Capitolo 3: Processi

CONCETTO DI PROCESSO: Un processo, 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 operativo esegue una varietà di programmi. Un **sistema a lotti** (batch) esegue **lavori** (job), mentre un sistema a partizione del tempo esegue **programmi utenti** o **task**.

Spesso i termini **lavoro** e **processo** sono usati in modo quasi intercambiabile.

Informalmente, un **processo** può essere considerato un programma in esecuzione e 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), può includere uno **heap**, ossia della memoria dinamicamente allocata durante l’esecuzione del processo. Ma sottolineiamo che un programma di per sé non è un processo: un programma è un’entità *passiva*, mentre il **processo** è un’entità *attiva*.

STATO DEL PROCESSO: Un processo durante l’esecuzione è soggetto a cambiamenti di **stato**, definiti in parte dall’attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati:

* **Nuovo**. Si crea il processo;
* **Esecuzione**. Un’unità di elaborazione esegue le istruzioni del relativo programma.
* **Attesa**. Il processo attende che si verifichi qualche evento.
* **Pronto**. Il processo attende di essere assegnato ad un’unità di 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** (PCB). Possono essere contenute le seguenti informazioni:

* **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 pile (*stack pointer*), registi d’uso generale e registri contenenti informazioni relativi 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.** Registri di base e di limite. Le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato.
* **Informazioni di contabilizzazione delle risorse.** Tempo di CPU, tempo reale d’utilizzo della stessa, i limiti di tempo, i numeri dei processi, e così via.
* **Informazioni sullo stato dell’I/O.** La lista dei dispositivi I/O assegnati a un determinato processo, l’elenco dei file aperti, e così via.

SCHEDULING DEI PROCESSI: L’obiettivo della multiprogrammazione consiste nel disporre dell’esecuzione contemporanea di alcuni processi in modo da massimizzare l’utilizzo della CPU. Per raggiungere questi obiettivi, lo **scheduler dei processi** seleziona un processo da eseguire dell’insieme di quelli disponibili.

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 di essere eseguiti, si trovano in una lista detta **coda dei processi pronti** (*ready queue*). Generalmente questa coda si memorizza come un alista concatenata.

Il sistema operativo ha anche altre code. Infatti, per esempio, una richiesta di I/O può essere diretta ad 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. L’elenco dei processi che attendono la disponibilità di un particolare dispositivo di I/O si chiama **coda del dispositivo**, ogni dispositivo ha la propria coda. 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 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.

Come possiamo immaginare quindi, i processi si spostano tra varie code.

SCHEDULER: Lo **scheduler** seleziona i processi e li carica in memoria assegnando la CPU in modo da ottimizzarne l’utilizzo. Spesso, in un sistema a lotti, accade che si sottopongano più processi di quanti ne possano eseguire immediatamente. Questi lavoro si trasferiscono in dispositivi di memoria secondaria, dove si tengono fino al momento dell’esecuzione (*spooling*). Abbiamo due tipi di **scheduler**:

* Lo **scheduler a lungo termine** (*job schedule*r), sceglie i lavori da questo insieme e li carica in memoria affinché vengano 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 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, infatti, 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.

La maggior parte dei processi si può caratterizzare in due prevalenze principali:

* Un **processo con prevalenza di I/O** che impiega la maggior parte del proprio tempo nell’esecuzione di operazioni di I/O.
* Un **processo con prevalenza d’elaborazione** che impiega la maggior parte del proprio tempo nelle elaborazioni.

Uno scheduler per rendere efficiente il suo lavoro è tenuto a selezionare una buona combinazione di processi I/O bound – CPU bound, altrimenti il sistema risulterebbe sbilanciato.

In alcuni sistemi operativi come quelli a partizione del tempo, si può introdurre un livello di scheduling intermedio. L’idea di base per lo **scheduler a medio termine** è che a volte può essere vantaggioso eliminare processi dalla memoria, riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta. Questo schema si chiama **avvicendamento dei processi in memoria** o **avvicendamento** (*swapping*).

CAMBIO DI CONTESTO: 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. In termini generali, si esegue un **salvataggio dello stato** corrente della CPU, in seguito, si attuerà un corrispondente **ripristino dello stato**. Questa procedura è nota col nome di **cambio di contesto** (*context switch*). Il tempo necessario al cambio di contesto è overhead, infatti il sistema non compie nessun lavoro direttamente utile alla sua computazione. La durata del cambio di contesto dipende molto dall’architettura dell’elaboratore.

OPERAZIONI SUI PROCESSI: Durante la propria esecuzione, un processo può creare numerosi nuovi processi. Il processo creante si chiama processo **genitore** o **padre**, 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.

La maggior parte dei sistemi operativi, identificano un processo per mezzo di un numero univoco, detto **identificatore del processo** o **pid** (*process identifier*). Quando un processo crea un sottoprocesso, quest’ultimo può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato ad un sottoinsieme delle risorse del processo genitore oppure ancora può non condividere nessuna risorsa del processo padre.

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.

In un sistema Unix, per creare un nuovo processo si una la chiamata **fork ()** mentre per caricare un programma nel processo figlio si usa la chiamata **exec ()** (dopo la chiamata **fork ()** ).

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 padre, che li riceve attraverso la chiamata di sistema **wait ()**.

La terminazione di un processo si può verificare anche in altri casi. Un processo può causare la terminazione di un altro per mezzo di un’opportuna chiamata di sistema ad esempio **abort ().** Un processo 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 padre. Generalmente è un intero maggiore di zero.

Un processo padre può porre termine all’esecuzione di uno dei suoi processi figli per diversi motivi:

* Il processo figlio ha ecceduto nell’uso delle risorse che gli erano state assegnate.
* Il padre non ha più bisogno del compito che svolgeva il figlio.
* Il padre termina e il sistema operativo non consente al figlio di continuare.

La maggior parte dei sistemi operativi non consentono ai processi figli di esistere dopo la terminazione del padre, quest’azione porta all’effetto di una **terminazione a cascata**.

COMUNICAZIONE TRA PROCESSI: I processi concorrenti nel sistema operativo possono essere **indipendenti** o **cooperanti**:

* Un processo è **indipendente** se non può influire su altri processi del sistema o impedirne l’uso.
* Un processo è **cooperante** se influenza o può essere influenzato da altri processi in esecuzione nel sistema.

I vantaggi della **cooperazione** tra processi sono:

* **Condivisione d’informazioni**. Poiché più utenti possono essere interessati alle stesse informazioni, è necessario offrire un ambiente che consenta un accesso concorrente a tali risorse.
* **Accelerazione del calcolo**. Ottenibile solo se il calcolatore dispone di più elementi capaci di attività d’elaborazione
* **Modularità**. Può essere necessaria la costruzione di un sistema modulare, che suddivide le funzioni di sistema in processi o thread distinti.
* **Convenienza**. Anche un solo utente può avere la necessità di compiere più attività contemporaneamente.

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di **comunicazione tra processi.** I modelli fondamentali sono due:

1. **A memoria condivisa.** Si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona.
2. **A scambio di messaggi.** La comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti.

SISTEMI A MEMORIA CONDIVISA: Per illustrare il concetto di cooperazione tra processi, si consideri il problema del **produttore** e del **consumatore**. Un processo **produttore** produce informazioni che sono consumate da un processo **consumatore**. L’esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal **produttore** e svuotato dal **consumatore**. Il buffer dovrà risiedere in una zona di memoria condivisa dai due processi. I processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un un’unità non ancora prodotta. Sono utilizzabili due tipi di buffer:

* **Buffer illimitato**: non pone limiti alla dimensione del buffer. Il consumatore può trovarsi ad attendere nuovi oggetti, ma il produttore può sempre produrne.
* **Buffer limitato**: presuppone l’esistenza di una dimensione fissa del buffer in questione che permetterà di memorizzare DIM\_BUFFER – 1 elementi. In questo caso, il **consumatore** deve attendere che il buffer sia vuoto; viceversa, il produttore deve attendere che il buffer sia pieno. Vediamo:

#define DIM\_BUFFER 10

Typedef struct {

…

} elemento;

elemento buffer[DIM\_BUFFER];

int inserisci = 0;

int preleva = 0;

Il buffer condiviso è realizzato come un array circolare con due puntatoli logici: **inserisci** e **preleva**. La variabile **inserisci** indica la successiva posizione libera nel buffer; **preleva** indica la prima posizione piena nel buffer. Il buffer risulterà vuoto se **inserisci == preleva**; pieno se ((**inserisci** + 1) % DIM\_BUFFER) == **preleva**. In particolare:

PROCESSO PRODUTTORE

elemento **appena\_Prodotto**;

while (true) {

/\*produce un elemento in **appena\_Prodotto**\*/

while (((**inserisci** + 1) % DIM\_BUFFER) == **preleva**);

/\*non fa niente\*/

buffer[**inserisci**] = **appena\_Prodotto**;

inserisci = (inserisci + 1) % DIM\_BUFFER;

}

PROCESSO CONSUMATORE

elemento **da\_Consumare**;

while (true) {

while (**inserisci == preleva**);

/\* non fa niente\*/

**da\_Consumare** = buffer[preleva];

preleva = (preleva + 1) % DIM\_BUFFER**;**

/\*consuma l’elemento in **da\_Consumare**\*/

}

Il processo produttore ha una variabile locale **appena\_Prodotto** contenente il nuovo elemento da produrre. Il processo consumatore ha una variabile lovale **da\_Consumare** in cui si memorizza l’elemento da consumare.

SISTEMI A SCAMBIO DI MESSAGGI: Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio degli indirizzi (Inter Process Communication – IPC).

Questo meccanismo per lo scambio di messaggi deve prevedere almeno due operazioni:

1. **send** (cioè, “invia messaggio”);
2. **receive** (cioè, “ricevi messaggio”).

I messaggi possono avere lunghezza fissa o variabile.

Se i processi **P** e **Q** vogliono comunicare, devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un **canale di comunicazione** (communication link), realizzabile in due modi:

1. **Realizzazione fisica** (memoria condivisa, bus, reti ecc..)
2. **Realizzazione logica**:
   1. **Comunicazione diretta o indiretta**;
   2. **Comunicazione sincrona o asincrona**;
   3. **Gestione automatica o esplicita del buffer**.

NOMINAZIONE: Con la **comunicazione diretta**, ogni processo che intenda comunicare deve nominare esplicitamente il ricevente o il trasmittente della comunicazione:

* **send** (P, messaggio): invia messaggio al processo P;
* **receive** (Q, messaggio): riceve, in messaggio, un messaggio dal processo Q.

All’interno di questo schema, un canale di comunicazione ha le seguenti caratteristiche:

* Tra ogni coppia di processi che intendono comunicare si stabilisce automaticamente un canale; i processi devono conoscere solo la reciproca identità;
* Un canale è associato esattamente a due processi;
* Il canale può essere unidirezionale, ma generalmente è bidirezionale.

Questo schema ha una **simmetria** nell’indirizzamento, vale a dire che per poter comunicare, il trasmettente e il ricevente devono nominarsi a vicenda. Una variante di questo schema di avvale dell’**asimmetria** dell’indirizzamento: soltanto il trasmittente nomina il ricevente:

* **Send** (P, messaggio): invia messaggio al processo P;
* **Receive** (id, messaggio): riceve, in messaggio, un messaggio da qualsiasi processo; nella variabile **id** si riporta il nome del processo con cui è avvenuta la comunicazione.

Con la **comunicazione indiretta**, i messaggi s’inviano a delle **porte** (dette anche *mail box*), che li ricevono. Una porta si può considerare in modo astratto come un oggetto in cui i processi possono introdurre e prelevare messaggi, ed è identificata in modo univoco, per esempio un valore intero. Due processi possono comunicare solo se condividono una porta, avremo dunque:

* **send** (A, messaggio): invia messaggio alla porta A;
* **receive** (A, messaggio): riceve, in messaggio, un messaggio dalla porta A.

In questo scema un **canale di comunicazione** ha le seguenti caratteristiche:

* Tra una coppia di processi si stabilisce un canale solo se entrambi i processi della coppia condividono una stessa porta;
* Un canale può essere associato a più di due processi;
* Tra ogni coppia di processi comunicanti possono esserci più canali diversi, ciascuno corrispondente ad una porta.

Per evitare incoerenze abbiamo alcune soluzioni:

* Si può fare in modo che un canale sia associato al massimo a due processi;
* Si può consentire l’esecuzione di un ‘operazione **receive** () a un solo processo alla volta;
* Si può consentire al sistema di decidere arbitrariamente quale processo riceverà il messaggio. In questo caso il sistema comunicherà l’identità del ricevente al trasmittente.

Il sistema operativo offre un meccanismo che permette al processo di:

* Creare una nuova porta;
* Inviare e ricevere messaggi tramite porta;
* Rimuovere una porta

SINCRONIZZAZINE: Lo scambio di messaggi può essere **sincrono** (o **bloccante**) oppure **asincrono** (o **non bloccante**).

* **Invio sincrono.** Il processo che invia il messaggio si blocca nell’attesa che il processo ricevente, o la porta, riceva il messaggio.
* **Invio asincrono.** Il processo invia il messaggio e riprende la propria esecuzione.
* **Ricezione sincrona.** Il ricevente si blocca nell’attesa dell’arrivo di un messaggio.
* **Ricezione asincrona.** Il ricevente riceve un messaggio valido oppure un valore nullo.

Se le primitive **send ()** e **receive ()** sono entrambe **bloccanti** si parla di **randez-vous** tra mittente e ricevente.

CODE DI MESSAGGI: Un canale ha una **capacità** che determina il numero dei messaggi che possono risiedere al suo interno, questa caratteristica può essere vista come una coda di messaggi legata al canale. Esistono tre tipi di code:

1. **Capacità zero.** La coda ha lunghezza massima **zero**, quindi il canale non può avere messaggi in attesa al suo interno. In questo caso il trasmittente deve fermarsi finché il ricevente prende in consegna il messaggio.
2. **Capacità limitata.** La coda ha lunghezza finita **n**, quindi al suo interno possono risiedere al massimo **n messaggi**. Se la coda non è piena, quando s’invia un nuovo messaggio, quest’ultimo è posto in fondo alla coda. Il trasmittente può proseguire la propria esecuzione senza essere costretto ad attendere. Il canale però se è pieno, il trasmittente deve fermarsi nell’attesa che ci sia spazio disponibile nella coda.
3. **Capacità illimitata.** La coda ha una lunghezza potenzialmente infinita, quindi al suo interno può attendere un numero indefinito di messaggi. Il trasmittente non si ferma mai.

Il caso **capacità zero** è talvolta chiamato sistema a scambio di messaggi **senza memorizzazione transitoria** (*no buffering*); gli altri due, sistemi **con memorizzazione transitoria automatica** (autmatic buffering).

CAPITOLO 4: THREAD

INTRODUZIONE: Un **thread** è l’unità di base d’uso della CPU, talvolta chiamato processo leggero (lightweight process). Un **thread** comprende un identificatore (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 del sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche **processo pesante**, è composta da un solo thread. Un processo **multithread** è in grado di lavorare a più compiti in modo concorrente. Molti programmi per i moderni PC sono predisposti per essere eseguiti da processi **multithread.** Di solito, un’applicazione si codifica come un nuovo processo a sé stante comprendente più thread di controllo. Il **multithreading** è la capacità di un sistema operativo si supportare thread di esecuzione multipli per ogni processo. Le idee di base del **modello a thread** sono:

* Permettere la definizione di “attività leggere” con costo di attivazione e terminazione limitato.
* Possibilità di condividere lo stesso spazio di indirizzamento.

In un modello a thread singolo la rappresentazione di un processo contiene il suo PCB e il suo spazio indirizzamento utente, lo stack utente e lo stack del kernel.

In un ambiente multithreading ogni processo ha associato:

* Un solo PCB;
* Un solo spazio di indirizzamento utente.

Ma ogni thread ha un proprio:

* Stack;
* Un blocco di controllo privato contenente l’immagine dei registri;
* Una priorità;
* Altre informazioni relative ai thread.

VANTAGI: I vantaggi della programmazione multithread si possono classificare rispetto a quattro fattori principali:

* **Tempo di risposta.** Un programma è in grado 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.
* **Condivisione delle risorse.** I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa o il passaggio di messaggi. 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.
* **Economia.** Assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo a cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto.
* **Scalabilità.** I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo (uno per ciascun processore).

Tuttavia però, questo modello, presenta anche alcuni svantaggi:

* Maggiore complessità di progettazione e programmazione:
  + I processi devono essere “pensati” paralleli.
  + Difficile sincronizzazione tra i thread.
* Il modello è inadatto per situazioni in cui i dati devono essere protetti.
* La condivisione di risorse accentua il pericolo di interferenze.

THREAD A LIVELLO UTENTE: I thread a livello utente sono gestiti come uno strato separato al di sopra del nucleo del sistema operativo. Sono realizzati attraverso una libreria di funzioni senza alcun intervento diretto del kernel, infatti il kernel non è necessariamente a conoscenza della presenza dei thread. Tutto il lavoro di gestione dei thread viene effettuato dall’applicazione. Se il nucleo di S.O. è a singolo thread e da uno dei thread utenti viene richiesta un’operazione bloccante (ad es. I/O), allora tutto il processo utente deve essere bloccato.

THREAD A LIVELLO KERNEL: Sono gestiti direttamente dal sistema operativo, che si occupa della creazione, scheduling e gestione dello spazio degli indirizzi. Non c’è codice di gestione dei thread ma un’API per la componente del kernel che gestisce i thread. Sono più lenti da creare e da gestire di quelli a livello utente. Tuttavia se un thread esegue un’operazione bloccante questa non blocca gli altri thread del processo. Se sono disponibili più CPU questi thread possono operare anche in parallelo.

VANTAGGI/SVANTAGGI LIVELLO UTENTE:

Vantaggi:

* Lo switching non coinvolge il kernel e quindi non ci sono cambiamenti della modalità di esecuzione.
* Maggiore liberta nella scelta dell’algoritmo di scheduling che può essere anche personalizzato.
* Siccome le chiamate possono essere raccolte in una libreria, c’è una maggiore portabilità del sistema operativo.

Svantaggi:

* Una chiamata al kernel può bloccare tutti i thread di un processo, indipendentemente dal fatto che solo uno dei suoi thread ha causato la chiamata bloccante.
* In sistemi multiprocessori, due processori non risulteranno mai associati a due thread del medesimo processo.

VANTAGGI/SVANTAGGI LIVELLO KERNEL:

Vantaggi:

* Il kernel può eseguire più thread dello stesso processo su più processori.
* Il kernel stesso può essere scritto multithread.

Svantaggi:

* Lo switching coinvolge chiamate al kernel, e questo ha un costo abbastanza alto.
* L’algoritmo di scheduling è meno personalizzabile e portabile.

MODELLI DI PROGRAMMAZIONE MULTITHREAD: Ci sono differenti modelli di gestione dei thread:

* **Modello da molti a uno**. Fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. Poiché si svolge nello spazio utente, la gestione dei thread risulta efficiente, ma l’intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante. Non c’è possibilità di parallelismo perché un solo thread alla volta può accedere al kernel.
* **Modelli da uno a uno**. Mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Questo modello offre un grado di concorrenza maggiore poiché se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread. Il modello permette anche l’esecuzione dei thread in parallelo in sistemi multiprocessore. L’unico svantaggio è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel.
* **Modello da molti a molti**. Mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel. Se un thread impiega una chiamata di sistema bloccante, il kernel può fare in modo che si esegua un altro thread. Nonostante permetta ai programmatori di creare tanti thread a livello utente quanti ne desiderino, non viene garantita una concorrenza reale, poiché il meccanismo di scheduling del kernel può scegliere un solo thread alla volta.

CAPITOLO 5: SCHEDULING DELLA CPU

CONCETTI FONDAMENTALI: Con la multiprogrammazione si cerca d’impiegare i 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 ad un altro processo. L’obiettivo del time-sharing è commutare l’uso della CPU tra i vari processi così frequentemente, che gli utenti possano interagire con ciascun programma in esecuzione.

CICLICITÀ DELLE FASI D’ELABORAZIONE E DI I/O: 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, infatti, si alternano tra questi due stati. Le durate delle sequenze di operazioni della CPU sono state misurate e la loro curva di frequenza è simile a quella esponenziale o iperesponenziale, con molte brevi sequenze di operazioni della CPU, e poche sequenze di operazioni di operazioni CPU molto lente.

SCHEDULER DELLA CPU: Lo **scheduler a breve termine**, o scheduler della CPU, tra i processi in memoria pronti per l’esecuzione, sceglie quello a cui assegnare la CPU. Le decisioni riguardanti lo **scheduler a breve termine** 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 segna 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.

Quando lo scheduling interviene nelle condizioni 1 e 4, si dice che lo schema di scheduling è **senza diritto di prelazione** (*non-preemptive)* o **cooperativo** (*cooperative*); altrimenti, lo schema di scheduling è **con diritto di prelazione** (*preemptive*).

Uno scheduling è **non-preemptive** se ogni processo a cui viene assegnata la CPU rimane in possesso della CPU fino a quando o termina la sua esecuzione oppure passa in uno stato waiting. Mentre se la politica di scheduling è **preemptive** allora bisognerà anche disporre di meccanismi per la sincronizzazione dei processi.

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**. Il **dispatcher** opera quindi:

* Il cambio di contesto;
* Il passaggio alla modalità utente;
* Il salto alla giusta posizione del programma utente per riavviarne l’esecuzione.

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: Riportiamo alcuni criteri di scheduling:

* **Utilizzo della CPU.** La CPU deve essere più attiva possibile.
* **Produttività.** La CPU è attiva quando svolge del lavoro. La **produttività** (*throughput*)è data dal numero dei processi completati nell’unità di tempo.
* **Tempi di completamento.** 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.** Il **tempo d’attesa** è la somma degli intervalli d’attesa passati nella coda dei processi pronti.
* **Tempo di risposta.** È dato dal tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta, che nella maggior parte dei casi non corrisponde all’output finale.

ALGORITMI DI SCHEDULING

SCHEDULING IN ORDINE D’ARRIVO: Il più semplice algoritmo di scheduling della CPU è l’algoritmo di **scheduling in ordine d’arrivo** (*scheduling 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 cosa FIFO.

Supponiamo che i processi arrivano nell’ordine P1, P2, P3 e sono serviti in ordine FCFS, si ottiene il risultato illustrato nel seguente **diagramma di Gantt**, un istogramma che illustra una data pianificazione includendo i tempi d’inizio e fine di ogni processo partecipante.

Processo Durata della sequenza

P1  24

P2 3

P3 3



Notiamo:

* Il tempo di attesa per P1: 0 millisecondi;
* Il tempo di attesa per P2: 24 millisecondi;
* Il tempo di attesa per P3: 27 millisecondi.

Quindi il tempo di attesa medio sarà la somma dei tempi di attesa diviso il numero dei processi eseguiti: (0 + 24 + 27) /3 = 17 millisecondi.

Supponiamo adesso che però i processi arrivano nel seguente ordine: P2, P3, P1, i risultati sarebbero quelli illustrati nel seguente diagramma di Gantt:



Notiamo ancora:

* Il tempo di attesa per P1: 6 millisecondi;
* Il tempo di attesa per P2: 0 millisecondi;
* Il tempo di attesa per P3: 3 millisecondi.

Quindi il tempo di attesa medio sarà: (6 + 0 + 3) /3 = 3 millisecondi.

Notiamo che è molto meglio dell’esempio precedentemente illustrato e quindi in FCFS il tempo di attesa medio può variare notevolmente al variare dei CPU burst dei processi e del loro ordine di arrivo. In questo caso abbiamo evitato l’**effetto convoglio**, cioè che tutti i processi attendono che un lungo processo liberi la CPU, che causa una riduzione dell’utilizzo della stessa e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi. Sottolineiamo che il criterio FCFS **non è preemtive**.

SCHEDULING PER BREVITÀ: Un criterio diverso di scheduling della CPU si può ottenere con l’algoritmo di **scheduling per brevità** (*shortest-job-first*, SJF). Questo algoritmo associa ad ogni processo la lunghezza della successiva sequenza di operazioni della CPU. Quando la CPU è disponibile, viene assegnata al processo che ha il CPU burst successivo più breve. Se però capita che due processi hanno lo stesso CPU burst, la CPU viene assegnata al processo che è entrato per primo nella coda.

Abbiamo due schemi di **scheduling per brevità**:

* **Non-preemptive**: Quando la CPU è allocata al processo non viene deallocata fino al completamento del CPU burst del processo in esecuzione.
* **Preemptive**: Se arriva un processo con CPU burst più breve del CPU burst rimanente al processo correntemente in esecuzione, la CPU viene subito deallocata ed allocata al nuovo processo.

Questo criterio è ottimale perché permette di ridurre notevolmente il tempo di attesa medio, ma è molto difficile conoscere la durata della successiva richiesta di CPU.

* **SJF NON PREEMPTIVE:**

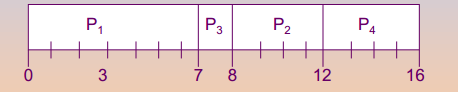
Processo Tempo di arrivo CPU burst

P1 0.0 7

P2 2.0 4

P3 4.0 1

P4 5.0 4



Il tempo di attesa adesso sarà la **somma degli intervalli di attesa passati nella coda dei processi pronti**. Il tempo di attesa:

* Il tempo di attesa per P1: 0 millisecondi;
* Il tempo di attesa per P2: (8 – 2) millisecondi;
* Il tempo di attesa per P3: (7 – 4) millisecondi.
* Il tempo di attesa per P4: (12 – 5) millisecondi.

Il tempo di attesa medio allora sarà: (0 + (8 – 2) + (7 – 4) + (12 – 5)) /4 = 4 millisecondi.

* **SJF PREEMPTIVE:**

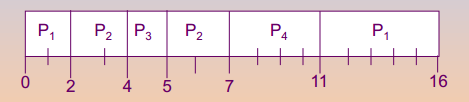
Processo Tempo di arrivo CPU burst

P1 0.0 7

P2 2.0 4

P3 4.0 1

P4 5.0 4



Il tempo di attesa adesso sarà:

* Il tempo di attesa per P1: 9 millisecondi;
* Il tempo di attesa per P2: 1 millisecondi;
* Il tempo di attesa per P3: 0 millisecondi.
* Il tempo di attesa per P4: 2 millisecondi.

Il tempo di attesa medio sarà: (9 + 1 + 0 + 2) /4 = 3 millisecondi.

DETERMINARE LA LUNGHEZZA DEL BURST SUCCESSIVO:

La difficoltà reale implicita nell’algoritmo SJF consiste nel conoscere la durata della successiva richiesta 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 cerca di ***predire***il suo valore, è probabile infatti che sia simile ai precedenti. Quindi, calcolando un valore approssimato della lunghezza, si può scegliere il processo con la più breve fra tali lunghezze previste.

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. Denotiamo:

1. tn = la lunghezza dell’*n*-esima sequenza di operazioni della CPU.
2. τn+1 = il valore previsto per la successiva sequenza di operazioni della CPU.
3. Definiamo:

Il valore tn contiene le informazioni più recenti; τn registra la storia passata. Il parametro α controlla il peso relativo sulla predizione della storia recente e quella passata.

Se α = 0, la storia recente non ha effetto, si suppone che le condizioni attuali siano transitorie; se invece α = 1, la storia passata è irrilevante. Più comune la condizione in cui α = ½, indica che la storia recente e la storia passata hanno lo stesso peso.

Per comprendere il comportamento della **media esponenziale**, si può sviluppare la formula per τn+1 sostituendo in τn in modo da ottenere:

Poiché sia α sia (1 – α) sono minori o uguali a 1, ogni termine ha peso inferiore a quello del suo predecessore.

SCHEDULING PER PRIORITÀ: L’algoritmo SJF è un caso particolare del più generale **algoritmo di scheduling per priorità**: si associa una priorità ad ogni processo e si assegna la CPU al processo con priorità più alta. Nella maggior parte dei casi la priorità viene identificata con un numero intero, più è grande il numero e più bassa è la priorità mentre più è basso il numero e più alta è la priorità.

Supponiamo che al tempo 0 arrivino nell’ordine P1, …, P5:

Processo Durata della sequenza Priorità

P1 10 3

P2 1 1

P3 2 4

P4 1 5

P5 5 2

Avremo questa situazione:



Il tempo di attesa medio sarà: (6 + 0 + 16 + 18 + 1) /5 = 8,2 millisecondi

Lo **scheduling per priorità** può essere sia con prelazione sia senza prelazione. Quando un processo arriva nella 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 dell’ultimo processo arrivato è superiore.

Un problema importante relativo agli algoritmi di scheduling per priorità è lo **starvation** (“inedia” o *attesa indefinita*), cioè che processi a bassa priorità potrebbero non essere mai eseguiti perché un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla CPU.

Una soluzione a questo problema 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: L’**algoritmo di scheduling circolare** (*round-robin,* **RR**) è stato progettato appositamente per i sistemi a tempo ripartito. 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. Allo scadere di questo tempo il processo viene sospeso e viene aggiunto alla fine della coda dei processi pronti.   
Lo scheduler della CPU individua il primo processo dalla coda dei processi pronti, gli assegna il **quanto di tempo** di CPU, e attiva il **dispatcher** per l’esecuzione del processo.

A questo punto possono accadere due cose:

* Il processo ha una sequenza di operazioni della CPU 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 il processo ha una sequenza sequenza di operazioni della CPU maggiore di un **quanto di tempo**, in questo caso il sistema operativo esegue un **cambio di contesto**, aggiunge il processo alla fine della coda dei processi pronti e, tramite lo scheduler della CPU, selezione il processo successivo nella coda dei processi pronti.

Se **n** è il numero dei processi presenti nella coda dei processi pronti e **q** è il quanto di tempo, possiamo dire che nessun processo rimane sospeso per più di (**n** – 1)**q** unità di tempo.

Il tempo di attesa medio per il criterio di scheduling RR è spesso abbastanza lungo.

Mostriamo adesso un esempio:

Processo Lunghezza del Burst

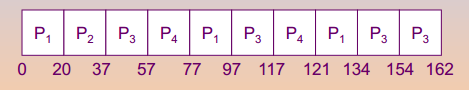
P1 53

P2 17

P3 68

P4 24

Il diagramma di Gantt per lo scheduling è:



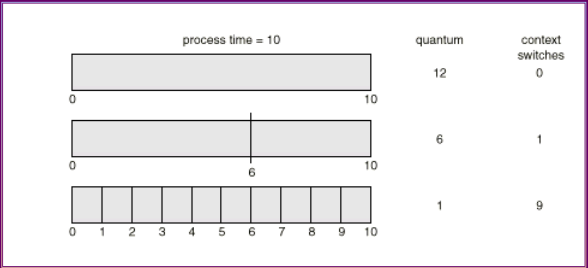
Il tempo di attesa:

* Il tempo di attesa per P1: 81 millisecondi;
* Il tempo di attesa per P2: 20 millisecondi;
* Il tempo di attesa per P3: 94 millisecondi.
* Il tempo di attesa per P4: 97 millisecondi.

Il tempo di attesa medio è: (81 + 20 + 94 + 97) /4 = 73 millisecondi

Generalmente ha tempo di completamento più alto dello scheduling SJF ma tempo di risposta migliore.

Possiamo notare banalmente che un **quanto di tempo** minore aumenta i campi di contesto:

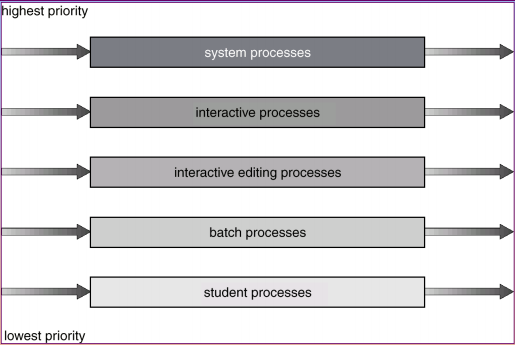


SCHEDULING A CODE MULTIPLE: È stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi si possono classificare facilmente in gruppi diversi. Una distinzione diffusa è per esempio quella che si fa tra i processi che si eseguono in **primo piano** (*foreground*), o **interattivi**, e i processi che si eseguono in **sottofondo** (*background*), o a **lotti** (*batch*).

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 ad una cosa. Ogni coda ha il proprio algoritmo di scheduling ed è inoltre necessario avere uno scheduling tra le code:

* Scheduling per priorità fissa: ad esempio, eseguire tutti i processi nella coda **foreground** e passare solo dopo alla coda **background**.
* Quanti di tempo: ciascuna coda ottiene un quanto di tempo della CPU con cui può schedulare i processi in coda.

Osserviamo:

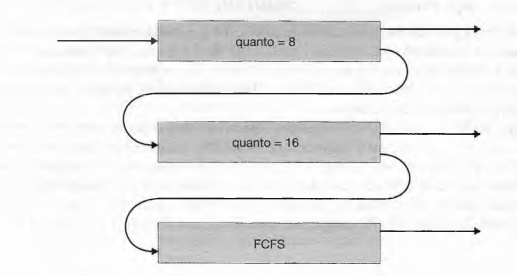


Ogni coda ha la priorità assoluta sulle code di priorità più bassa; nessun processo della coda dei processi in *sottofondo* può iniziare l’esecuzione finché le code per i processi di sistema, interattivi e interattivi di *editing* non siano tutte vuote. Se un processo interattivo di *editing* entrasse nella coda dei processi pronti durante l’esecuzione di un processo di sottofondo, si avrebbe la prelazione su quest’ultimo.

Oppure, come già detto, esiste 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 vari processi che la costituiscono.

SCHEDULING A CODE MULTIPLE CON RETROAZIONE: Lo **scheduling a code multiple con retroazione** (*multilevel feedback queue scheduling*), invece, permette ai processi di spostarsi fra le code. Se un processo usa per troppo tempo di elaborazione la CPU, viene spostato in una coda con priorità più bassa. Analogamente, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo. In questo modo si attua una forma d’**invecchiamento** che impedisce il verificarsi di un’attesa indefinita.

Graficamente avremo una situazione del genere:



All’ingresso nella coda dei processi pronti, i processi vengono assegnai alla coda 0 e ottengono un **quanto di tempo** di 8 millisecondi; i processi che non terminano entro tale **quanto di tempo**, vengono sposati alla fine della coda 1. Se la coda 0 vuota, si assegna un quanto di tempo di 16 millisecondi al processo alla testa della coda 1, ma se questo non riesce a completare la propria esecuzione, viene sottoposto a prelazione e messo nella coda 2. Se le code 0 e 1 sono vuote, si eseguono i processi della coda 2 secondo il criterio FCFS.

Generalmente uno **scheduler a code multiple con retroazione** è caratterizzato dai seguenti parametri:

* Numero di code;
* Algoritmo di scheduling per ciascuna coda;
* Metodo usato per determinare quando spostare un processo in una coda con priorità maggiore/minore;
* Metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

SCHEDULING PER SISTEMI DI ELABORAZIONE IN TEMPO REALE: Lo **scheduling a tempo reale stretto** (*hard – real- time*) è un sistema capace di garantire il completamento delle funzioni critiche entro un tempo definito. Un processo si presenta insieme ad una dichiarazione del tempo entro cui deve completare le proprie funzioni, se è possibile garantire il completamento allora lo scheduler accetta il processo, altrimenti rifiuta la richiesta (prenotazione delle risorse)

Lo **scheduler a tempo reale debole** (*soft real time*) è un sistema che garantisce una priorità più alta ai processi critici in modo tale che avranno sempre una prevalenza sui processi ordinari.

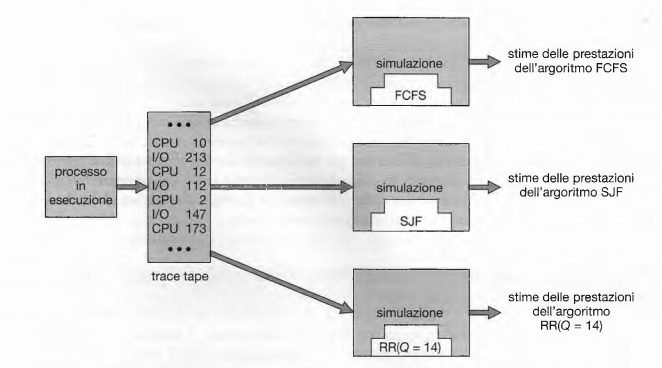
VALUTAZIONE DEGLI ALGORITMI: La scelta dell’algoritmo di scheduling da adottare è molto difficile. I criteri da usare per la scelta dell’algoritmo si definiscono spesso nei termini dell’utilizzo della CPU, del tempo di risposta o della produttività; poi occorre stabilire l’importanza relativa di queste misure. Per esempio:

* Rendere massimo l’utilizzo della CPU con il vincolo che il massimo tempo di risposta sia 1 secondo;
* Rendere massima la produttività in modo che il tempo di completamento sia, in media, linearmente proporzionale al tempo d’esecuzione totale.

Una volta definiti i criteri di selezione, è necessario valutare gli algoritmi considerati. Abbiamo vari metodi di valutazione:

* **Modello deterministico:** è un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e fornisce una formula o un numero che valuta le prestazioni di ciascun algoritmo per quel carico di lavoro. Necessitano in genere, però, di informazioni di conoscenze troppo dettagliate ed impossibili da ottenere.
* **Reti di code:** il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d’attesa. La CPU è un server con la propria coda dei processi pronti, ed il sistema di I/O ha le sue code dei dispositivi. Se sono noti l’andamento degli arrivi e dei servizi, si possono calcolare l’utilizzo, la lunghezza media delle code, il tempo medio d’attesa e così via. Questo tipo di studio si chiama **analisi delle reti di code** (*queueing-network-analysis*). Se il sistema è stabile, il numero di processi che lasciano la coda deve essere uguale al numero di processi che vi arrivano, quindi , quest’equazione è nota come **formula di Little** ed è utile soprattutto perché è valida per qualsiasi algoritmo di scheduling e distribuzione d’arrivi. Il risultato è un’approssimazione, non sempre esatta, del sistema reale.

SIMULAZIONI: Per riuscire ad avere una valutazione più precisa degli algoritmi di scheduling ci si può servire di **simulazioni**. Le **simulazioni** implicano la programmazione di un modello del sistema di calcolo; I processi sono simulati, ad esempio, attraverso generatori di numeri casuali che simulano le quantità di risorse necessarie ad ogni processo. Il simulatore dispone di una variabile che rappresenta un clock, con l’aumentare del valore di questa variabile, il simulatore modifica lo stato del sistema in modo da descrivere le attività dei dispositivi, dei processi e dello scheduler. La simulazione però non offre informazioni molto dettagliate, per rimediare a questo problema si può sottoporre il sistema reale ad un controllo continuo, con la registrazione della sequenza degli eventi effettivi, in questo modo si ottiene un cosiddetto **trace tape**, che poi si usa per condurre la simulazione:



Poiché spesso richiedono diverse ore del tempo d’elaborazione, le simulazioni possono essere molto onerose. Durante l’esecuzione della simulazione si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi.

REALIZZAZIONE: L’unico modo assolutamente sicuro per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo ed osservarne il comportamento nelle reali condizioni di funzionamento del sistema. Ci sono vari problemi però inerenti alla **realizzazione**:

* Costo molto elevato: le spese non sono dovute solo alla codifica dell’algoritmo e alle modifiche da fare al sistema operativo, ma anche alle relazioni degli utenti a fronti di costanti modifica del sistema operativo.
* Cambiamento dell’ambiente in cui lo si usa: l’ambiente si modifica anche in seguito alle prestazioni dello scheduler.

CAPITOLO 6: SINCRONIZZAZIONE DEI PROCESSI

INTRODUZIONE.1: I temi centrali della progettazione di un SO sono connessi con la gestione di processi e thread:

* Multiprogrammazione: processi multipli in un sistema monoprocessore;
* Multiprocessing: processi multipli in un sistema multiprocessore;
* Processi Distribuiti: gestione di processi multipli eseguiti su sistemi distribuiti.

Concetto fondamentale è quello di **concorrenza**, che comprende diversi aspetti di progettazione:

* Comunicazione tra processi;
* Condivisione e competizione per le risorse;
* La sincronizzazione delle attività di processi multipli;
* L’allocazione di tempo di processore ai processi, ecc...

La concorrenza appare almeno in tre diversi contesti:

* Applicazioni multiple: la multiprogrammazione è stata inventata per permettere di dividere dinamicamente il tempo di calcolo tra le applicazioni attive;
* Applicazioni strutturate: alcune applicazioni possono essere programmate efficacemente come insiemi di processi concorrenti;
* Struttura del SO: spesso anche i SO sono implementati come insiemi di processi.

In un sistema a singolo processore con multiprogrammazione i processi sono alternati nel tempo per dare l’illusione dell’esecuzione simulata. Nonostante l’overhead, l’esecuzione alternata porta benefici dal punto di vista dell’efficienza di esecuzione e della strutturazione dei programmi. In un sistema multiprocessore è possibile non solo alternare le esecuzioni dei processi ma anche sovrapporle (parallelo).

Nel caso di un singolo processore non è possibile predire le velocità relative dei processi, poiché queste dipendono dalle attività degli altri processi, dal modo in cui il SO gestisce gli interrupt e dalla politica di schedulazione del sistema. Si presentano queste difficolta:

* La condivisione di risorse globali è pericolosa:
  + Ad es. se due processi fanno uso della stessa variabile globale ed effettuano letture e scritture, allora l’ordine in cui le operazioni sono eseguite è critico.
* Per il SO è difficile e gestire l’assegnazione di risorse in maniera ottimale:
  + Un processo A può richiedere l’uso di un particolare canale di IO ed essere sospeso prima di poterlo usare.
  + Per il SP bloccare semplicemente il canale ed impedire l’uso da parte degli altri processi può essere inefficiente.
* Trovare un errore di programmazione diventa molto difficile perché i risultati e la sequenza di esecuzioni interlacciate dei vari processi non sono sempre facilmente riproducibili.

I principi di progettazione e di gestione che nascono a causa della concorrenza:

* Il SO deve tenere traccia dei processi attivi.
* Il SO deve allocare e deallocare varie risorse per ogni processo attivo:
  + Es tempo di elaborazione
  + Memoria
  + File
  + Dispositivi di IO
* Deve proteggere i dati e le risorse fisiche di ogni processo da interferenze da parte di altri processi.
* Il risultato di un processo deve essere indipendente dalla sua velocità di esecuzione, relativamente a quella degli altri processi concorrenti.

INTRODUZIONE.2: L’accesso concorrente a dati condivisi può creare problemi di **inconsistenza** dei dati. Per mantenere la consistenza dei dati sono necessari meccanismi che assicurino l’esecuzione ordinata e coordinata dei processi cooperanti. Tornando al problema del **produttore/consumatore**, 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 qualvolta che si inserisce un nuovo elemento nel buffer e si decrementa quando si preleva un elemento dal buffer.

Il codice si può modificare come segue:

#define DIM\_BUFFER 10

Typedef struct {

…

} elemento;

elemento buffer[DIM\_BUFFER];

int inserisci = 0;

int preleva = 0;

contatore = 0;

Il codice per il processo **produttore**:

elemento **appena\_Prodotto**;

while (1) {

/+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**:

elemento **da\_Consumare**;

while (1) {

while (contatore == 0);

/\*Non fa niente\*/

**da\_Consumare** = buffer[preleva];

preleva = (preleva + 1) % DIM\_BUFFER;

contatore--;

/\*consuma un elemento in **da\_Consumare**\*/

}

Se si considerano separatamente, le due procedure sono corrette, ma se si eseguono in modo **concorrente** possono non funzionare. Quindi deduciamo che lo statement **{contatore++, contatore--}** devono essere eseguiti *atomicamente*. Un’operazione si dice *atomica* se non può essere interrotta fino al suo completamento.

Lo statement **contatore++** si può codificare in un tipico linguaggio macchina come:

registro1 = contatore

registro1 = registro1 + 1

contatore = registro1

Analogamente per **contatore --**:

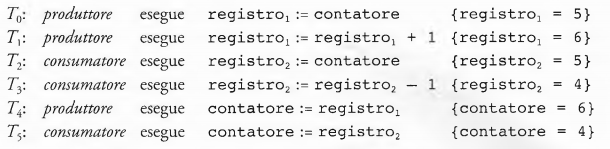
registro2 = contatore

registro2 = registro2 + 1

contatore = registro2

Se sia il **produttore** che il **consumatore** tentano di accedere concorrentemente al buffer, le esecuzioni in linguaggio macchina di **contatore++** e **contatore—**potrebbero interfogliarsi (*interleaving*), il risultato dell’*interleaving* dipende da come sono schedulati i due processi.

Supponiamo che **contatore** sia inizializzato a **5** e che ci siano un elemento prodotto e uno consumato, un possibile *interleaving* è:

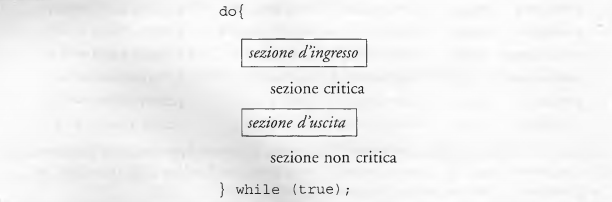


Il valore finale di **contatore** quindi è 4, ma il valore corretto che dovrebbe essere presente nella variabile è **5.**

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**. Per prevenire le **race condition**, i processi concorrenti devono essere sincronizzati tra di loro.

PROBLEMA DELLA SEZIONE CRITICA: Si consideri un sistema composto di **n** processi {P0, P1, …, Pn-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. Quindi, l’esecuzione delle sezioni critiche da parte dei processi è ***mutuamente esclusiva*** nel tempo. Ogni processo, quindi, 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**.

Struttura generale di un tipico processo Pi:



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

1. **Mutua esclusione**. Se il processo Pi è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. **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 dalle 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.
3. **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.

I processi possono usare delle variabili *condivise* per sincronizzarsi tra di loro.

Assumiamo che ogni processo sia eseguito ad una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla **velocità relativa** degli *n* processi.

SOLUZIONE DI PETERSON: Prima di arrivare alla **soluzione di Peterson** illustriamo due algoritmi per risolvere le **race condition,** a primo impatto sembrano funzionare ma poi vedremo perché invece non risolvono il nostro problema.

ALGORITMO 1:

Variabili condivise:

* **int turno**:
  + inizialmente **turno = 0**; potrà assumere valori 0 o 1.
* **turno == i**:
  + Pi può entrare nella sua sezione critica.

do {

while (**turno != i**);

*sezione critica;*

**turno == j**;

*sezione non critica*;

} while (1);

Problemi:

* Soddisfa la **mutua esclusione**, ma non il **progresso**;
* Se **turno == 0**, Pi non può entrane nella sua sezione critica anche se Pj è nella propria sezione NON critica.

ALGORITMO 2

Variabili condivise:

* **boolean pronto [2]**:
  + inizialmente pronto [0] = pronto [1] = false;
* **pronto [i] == true**:
  + Pi è pronto ad entrare nella sua sezione critica;

do {

**pronto [i] = true**;

while (**pronto [j]**);

*sezione critica*;

**pronto [i] == false**;

*sezione non critica*;

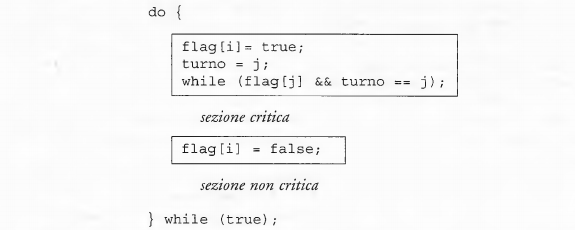
} while (1);

Problemi:

* Soddisfa la **mutua esclusione**, ma non il **progresso**;
* Se infatti consideriamo:
  + To: P0 assegna pronto [0] = true;
  + T1: P1 assegna pronto [1] = true;

Quindi i due processi P0 e P1 entrano in un ciclo infinito nelle rispettive istruzioni di while. Questo algoritmo dipende in modo decisivo dalla esatta temporizzazione dei due processi.

Utilizzando le variabili condivise usate nei due precedenti algoritmi, vedremo che per accedere alla *sezione critica*, il processo Pi assegna innanzitutto a *flag [i]* il valore *true*; quindi attribuisce a *turno* il valore *j*, conferendo così all’altro processo la facoltà di entrare nella propria *sezione critica*. Qualora entrambi i processi tentino l’accesso contemporaneamente, all’incirca nello stesso momento sarà assegnato a *turno* sia il valore *i* sia il valore *j*. Soltanto uno dei due permane, mentre 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*:



Questo algoritmo rispetta entrambi i requisiti:

**Mutua esclusione**: **Pi**entra nella propria sezione critica solo se **pronto [j] == false** o **turno == i**. Se entrambi i processi fossero contemporaneamente in esecuzione nelle rispettive *sezioni critiche*, si avrebbe **pronto [i] = pronto [j] = true**. L’istruzione *while* non può essere eseguita da **Pi** e **Pj** contemporaneamente perché **turno** può assumere valori **i** o **j** ma non entrambi. Supponendo che **Pj** abbia eseguito con successo l’istruzione di *while* avremo che **pronto [j] == true** e **turno == j**, condizione che persiste fino a che **Pj**si trova nella propria sezione critica.

**Attesa limitata e Progresso**: Si può impedire ad un processo **Pi** di entrare nella *sezione critica* solo se questo è bloccato nel ciclo *while* dalla condizione **pronto [j] == true && turno == j**. Se **Pj**non è pronto ad entrare allora **pronto [j] == false && turno == j**, e **Pi** può entrare nella propria *sezione critica*. Se **Pj** invece è pronto per entrare, allora **pronto [j] == true**, poi se **turno == i** entra **Pi**, mentre se **turno == j** entra **Pj**. Se entra **Pj** all’uscita imposta **pronto [j] == false** e **Pi** può entrare.

Notiamo che **Pi** entra nella *sezione critica* (**Progresso**) al massimo dopo un ingresso da parte di **Pj** (**Attesa limitata**).

SOLUZIONE PER PIÙ PROCESSI: Consideriamo la *sezione critica* per **n** processi. Questo algoritmo si basa su uno schema di servizio usato “nelle panetterie”. Al suo ingresso nel negozio, ogni cliente riceve un nuovo e viene servito di volta in volta il cliente con il numero più basso. Questo algoritmo però non assicura che tutti abbiano ricevuto un numero diverso, e quindi nel caso in cui Pi e Pj hanno lo stesso numero allora viene servito prima il processo con *process ID* minore (i < j). Più formalmente diremo che, considerando la notazione (ticket #, process ID #):

1. Se oppure
2. Se

* è un numero k, tale che per i = 0, …, n – 1;

ALGORITMO DEL FORNAIO:

Variabili condivise:

* boolean **scelta [n]** = false;
* int **numero [i]** = 0;

do {

scelta[i] = true; /\* prendo un numero dal salta code ed ordino agli altri processi di non fare nulla \*/

numero[i] = max (numero[0], …, numero[n-1]) + 1; /\* sta assegnando ad un numero maggiore di tutti gli altri \*/

scelta[i] = false; /\* indica agli altri processi che ha preso il proprio numero \*/

for (j = 0; j < n; j++) {

while (scelta[j]); /\* se un altro processo sta prendendo un numero allora aspetta \*/

while ((numero[j] != 0) && ((numero[j],j) < (numero[i],i))); /\* controlla se j ha preso un numero di i, e poi mi confronto pure con me stesso \*/

}

*sezione critica*; /\* faccio quello che devo fare \*/

numero[i] = 0; /\* metto il mio numero con i = 0; così se qualcuno era bloccato nel while adesso può uscire \*/

*sezione non critica*;

} while (true);

CORRETTEZZA DELL’ALGORITMO:

**Mutua esclusione**: Se Pi si trova nella propria sezione critica e Pk (k != i) ha già scelto il proprio numero[k] != 0 allora (numero[i], i) < (numero[k], k).

Se Pi è nella propria sezione critica e Pk tenta di entrare nella propria sezione critica, il processo Pk esegue la seconda istruzione del while per j == i e trova:

* numero[i] != 0;
* (numero[i], i) < (numero[k], k);

Quindi continua il ciclo nell’istruzione while fino a che Pi lascia la propria sezione critica.

**Progresso e Attesa limitata**: Questi requisiti sono garantiti perché i processi entrano nella propria sezione critica secondo il criterio FCFS.

HARDWARE PER LA SINCRONIZZAZIONE: In generale, si può affermare che qualunque soluzione al problema richiede l’uso di un semplice strumento detto **lock** (lucchetto). 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.

In un sistema dotato di una singola CPU tale problema si potrebbe risolvere semplicemente se si evitasse che un processo venga interrotto mentre si modificano le variabili condivise. Questo può essere realizzato con apposite primitive del kernel per abilitare e disattivare gli interrupt:

Per esempio:

repeat

<disattiva le interruzioni>

<sezione critica>

.

.

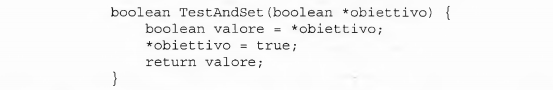
Forever

C’è un problema però, in questo modo la **mutua esclusione** è garantita, però il processore non può più alternare i programmi liberamente.

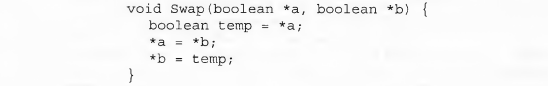
In sistemi con più unità di elaborazione, la disabilitazione delle interruzioni può comportare sprechi di tempo dovuti alla necessità di trasmettere richiesta di disabilitazione elle interruzioni a tutte le unità di elaborazione.

Per questo motivo molte delle moderne architetture offrono particolari istruzioni che permettono di controllare o modificare il contenuto di una parola di memoria (***TestAndSet***), oppure di scambiare il contenuto di due parole di memoria (***Swap***), in modo **atomico**, cioè senza essere interrotte.

L’istruzione ***TestAndSet*** è eseguita **atomicamente**, se si eseguono contemporaneamente due istruzioni ***TestAndSet()***, ciascuna in un’unità di elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario:

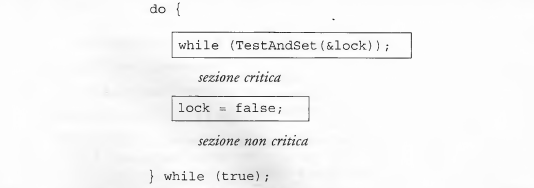


L’istruzione **Swap()**:

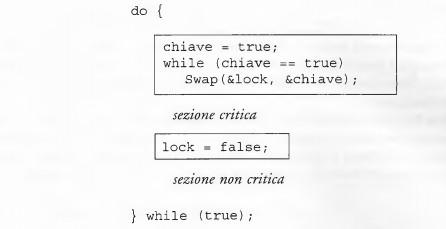


**Mutua esclusione** con ***TestAndSet()***:

Variabili condivise: **boolean look = false;**



**Mutua esclusione** con ***Swap()***:



Come vediamo, questi algoritmi soddisfano entrambi il requisito della **mutua esclusione**, ma non quello dell’**Attesa limitata**. Definiamo una soluzione che soddisfa tutti i requisiti:

Variabili condivise:

* **boolean** blocco = false;
* **boolean** attesa[n] = false;

do {

attesa[i] = true; /\* Pi è il primo scelto da sistema ed andrà a fare *TestAndSet \*/*

chiave = true;

while (attesa[i] && chiave)

chiave = **TestAndSet** (&blocco); /\* trova che blocco = false e lo modifica a true, poi setta chiave = false \*/

attesa[i] = false; /\* significa che non verrà bloccato durante le prossime istruzioni \*/

*sezione critica*;

j = (i + 1) % n; /\* passa la palla ad un altro processo \*/

while ((j != i) && !attesa[j]) /\* va a vedere, in maniera circolare, tutti i processi a

j = (j + 1) % n; destra di Pi, se c’è uno che vuole andare in sezione critica lo manda \*/

if (j == i) /\* esco dal while o perché ne ho trovato uno, oppure

blocco = false; perché li ho scorsi tutti e sono ritornato a me stesso,

else se nessuno vuole andare ristabilisco la situazione

attesa[j] = false; inziale con blocco = false, se invece l’ho trovato, metto

*sezione non critica*  attesa[j] = false cosi sblocco il processo Pj dal primo

} while (true); while e andrà in sezione critica \*/

**Mutua esclusione**: Un processo può entrare in sezione critica solo se attesa[i] == false oppure se chiave == false, chiave è impostata a false solo se si esegue il **TestAndSet**. Infatti il primo processo che segue il **TestAndSet** setta chiave a false e tutti gli altri processi devono attendere, infine attesa[i] può diventare false solo se un altro processo esce dalla propria sezione critica, e notiamo che solo una variabile attesa[i] vale false.

**Progresso**: Un processo quando esce dalla sezione critica, o imposta blocco a false cioè che non ha trovato un altro processo che vuole andare nella propria sezione critica e quindi ristabilisce la situazione inziale, oppure imposta attesa[j] a false, consentendo quindi ad un altro processo di andare in sezione critica.

**Attesa limitata**: Un processo, quando esce dalla propria sezione critica, va a controllare in modo circolare se un altro processo vuole andare nella propria sezione critica. Il primo processo che trova lo sblocca e così via per gli altri. Quindi ogni processo che chiede, entrerà in sezione critica entro **n – 1** turni.

SEMAFORI: Un **semaforo** S è una variabile intera cui si può accedere solo tramite due operazioni atomiche predefinite: ***wait ()*** e ***signal ()***:

**wait (S)** {

while (S ); //non -op

S--;

}

**signal (S)** {

S--;

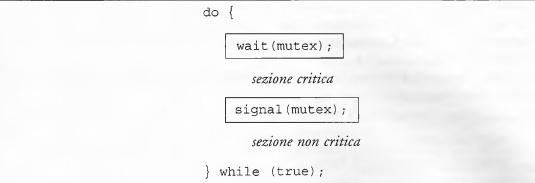
}

Il processo che sta aspettando cicla senza effettuare operazioni e non permette ad altri processi di avere il controllo della CPU, viene chiamata “attesa attiva” (busy waiting). I **semafori** serviranno per controllare le risorse; infatti, un processo che richiede l’utilizzo di un dispositivo, se S > 0 gli assegna quel dispositivo e decrementa S di uno, se invece non ci sono unità di risorse disponibili deve aspettare; quando finisce il suo lavoro col dispositivo si effettua la **signal ()** per liberare il dispositivo e incrementare S di uno perché adesso abbiamo liberato un dispositivo.

USO DEI SEMAFORI: Si usa distinguere due tipi di semafori:

* **semafori contatore**: il cui valore numerico è illimitato;
* **semafori binari**: il cui valore è 0 o 1.

In relazione ad alcuni sistemi i semafori binari sono anche detti **lock mutex** (*mutex locks*) perché fungono da “lock” che garantiscono la **mutua esclusione**. Usiamo come variabile condivisa **semaforo mutex** inizializzata a 1:



Il principale svantaggio della definizione di semaforo è che richiede una condizione di **attesa attiva** (busy waiting). Questa soluzione costituisce un problema per un sistema con multiprogrammazione, poiché la condizione di **attesa attiva** spreca cicli della CPU che un altro processo potrebbe sfruttare in modo produttivo. Per superare questo problema si possono modificare le definizioni delle operazioni **wait()** e **signal()**: quando un processo invoca l’operazione di **wait()** e trova che il valore del semaforo non è positivo, deve attendere andando a bloccare se stesso, assumiamo infatti l’esistenza della funzione **block(),** che sospende il processo che la invoca e pone il processo in una coda d’attesa associata al semaforo, e della funzione **wakeup(P)** che fa riprendere l’esecuzione di un processo P precedentemente bloccato. Quindi si può definire un semaforo come una struttura del linguaggio C:

typedef struct {

int valore; // “S” precedente

struct processo \*lista; // coda dei processi

} semaforo;

A ogni semaforo sono associati un **valore** intero e una **lista** di processi contenente i processi in attesa a un semaforo.

L’operazione di **wait()** sarà:

void wait (semaforo \*S) { //richiesta di un dispositivo

S.valore--; //decrementiamo i dispositivi disponibili

If (S.value < 0) { //non ci sono dispositivi, bloccami

Aggiungi questo processo a S ->lista;

**block();** //blocca il processo

}

}

L’operazione di **signal()** sarà:

void signal (semaforo \*S) { //restituiamo un dispositivo

S.valore++; //c’è un dispositivo in più libero

If (S.value <= 0) {

Togli un processo P da S->lista;

**wakeup()**; //sveglia il processo prelevato dalla lista

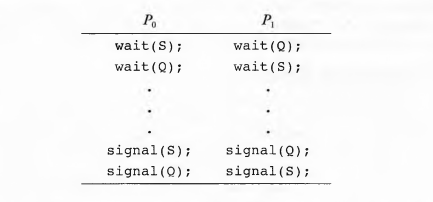
}

}

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

STALLO 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 attendere **indefinitamente** un evento che può essere causato solo da uno dei processi dello stesso insieme. Quando si verifica questa situazione si dice che i processi sono in **stallo** (*deadlocked*).

Ad esempio, si consideri un insieme di due processi, P0 e P1, ciascuno dei quali ha accesso a due semafori S e Q, inizializzati a 1:



Si supponga che P0 esegua **wait(S)** e quindi P1 esegua **wait(Q)**; eseguita **wait(Q)**, P0 deve attendere che P1 esegua **signal(Q)**; analogamente, quando P1 esegue **wait(S)**, deve attendere che P0 esegua **signal(S)**. Poiché queste operazioni **signal()** non si possono eseguire, P0 e P1 sono in stallo.

Un’altra questione connessa alle situazioni di stallo è quella dell’**attesa indefinita** (nota come *starvation*), che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio LIFO.

DUE TIPI DI SEMAFORI: Come già detto precedentemente abbiamo due tipi di semafori: **contatore** e **binario.** Possiamo però costruire un semaforo **contatore** S tramite semafori **binari**.

Per fare ciò abbiamo bisogno di sapere cosa ci serve per implementarlo.

Strutture dati:

* QuQusemaforo binario S1, S2;
* int C.

Inizializzazione:

* S1 = 1;
* S2 = 0;
* C = valore iniziale del semaforo contatore S.

Il valore intero C conterrà il numero dei dispositivi iniziali a disposizione.

Con S1 gestiamo la mutua esclusione e con S2 la coda dei processi.

Andiamo a vedere lo pseudocodice…

Operazione di **wait()**:

wait(S1); //il processo va a vedere se il semaforo binario S1 è verde e ne troverà al massimo uno ed è rosso per tutti gli altri

C--; //decremento il valore del semaforo

if (C < 0) { //Se il valore di C è minore di zero

signal(S1); //sblocco S1 tramite una signal(S1) e permette ad altri processi di accedere al contatore

wait(S2); //il processo si deve bloccare e mettersi in attesa

}

signal(S1); //devo utilizzare l’unità di risorsa disponibile, ma prima di farlo sblocco il semaforo che avevo messo a rosso tramite la wait(S1)

Operazione di **signal()**:

wait(S1); //regolo la mutua esclusione sul contatore, se S1 è verde incremento il contatore altrimenti aspetto

C++; //incremento il contatore

if (C <= 0) { //Se C è minore o uguale di 0

signal(S2); //se dopo l’incremento, il valore del contatore è minore uguale di zero, allora sveglio un processo facendo una signal(S2), un qualsiasi processo avrà la possibilità di accedere al dispositivo, infatti andrà a sbloccare la wait(S2) dove si era bloccato e sbloccherà il semaforo S1 che regola l’accesso in mutua esclusione

} else {

Signal(S1); //se dopo l’incremento, invece, il valore del contatore è maggiore di zero, significa che non c’è nessuno in attesa e rimetto il segnale a verde.

PROBLEMI TIPICI DI SINCRONIZZAZIONE: Esaminiamo adesso alcuni problemi di sincronizzazione come esempi di una vasta classe di problemi connessi al controllo della concorrenza. I problemi sono:

1: Produttori e consumatori con memoria limitata.

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con *n* posizioni, ciascuna capace di contenere un elemento:

itemvettore [DIM\_VETTORE]

semaforo **piene**; /\*numero di posizioni pene\*/

semaforo **vuote**; /\*numero di posizioni vuote\*/

semaforo **mutex**; /\*garantisce la mutua esclusione\*/

Inizialmente avremo questa situazione:

**piene** = n **vuote** = 0 **mutex** = 1

Processo produttore:

do {

…

Produce un elemento in **appena\_Prodotto**;

…

wait (**vuote**);

wait (**mutex**);

…

inserisci in **itemvettore** l’elemento in **appena\_Prodotto**;

signal (**mutex**);

signal (**piene**);

} while (true);

Processo consumatore:

do {

wait (**piene**);

wait (**mutex**);

…

rimuovi un elemento da **itemvettore** e mettilo in **da\_Consumare**;

…

signal (**mutex**);

signal (**vuote**);

…

consuma l’elemento in **da\_Consumare**;

…

} while (true);

2: Problema dei lettori – scrittori

Si supponga di avere una base di dati da condividere con numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura (**lettori**), altri possono richiedere l’aggiornamento e quindi una lettura e una scrittura (**scrittori**), dello stesso oggetto. Naturalmente, se due **lettori** accedono nello stesso momento alla base di dati, non si ha alcun effetto negativo; viceversa, se uno **scrittore** accede, gli altri processi non possono accedere, quindi gli **scrittori** devono avere accesso esclusivo.

Variabili condivise:

int **numlettori**; //numero di processi che stanno attualmente leggendo

semaforo **mutex**; //mutua esclusione per aggiornamento numlettori

semaforo **scrittura**; //mutua esclusione scrittori

Inizialmente avremo:

**numlettori** = 0 **mutex** = 1 **scrittura** = 1

Processo scrittore:

do {

wait (**scrittura**);

…

esegui l’operazione di scrittura;

…

Signal (**scrittura**);

} while (true);

Processo lettore:

do {

wait (**mutex**); //accede in mutua esclusione

**numlettori++**;

if (**numlettori** == 1) //prima del processo non c’era nessun altro lettore e quindi lui è il primo ad accedere al dispositivo

wait (**scrittura**); //vediamo se c’è uno scrittore in esecuzione, se c’è bloccati altrimenti vai avanti

signal (**mutex**); //sblocca l’accesso a **numlettori**

…

esegui l’operazione di lettura;

…

wait (**mutex**) //accede in mutua esclusione

**numlettori--**; //ho finito, quello che dovevo fare l’ho fatto

if (**numlettori** == 0) //se è vera vuol dire che lui era l’ultimo dei lettori

signal (**scrittura**); //sblocca il semaforo **scrittura** e da questo momento se qualcuno che vuole scrivere può farlo

signal (**mutex**); //sblocca l’accesso a **numlettori**

} while (true);

3: Problema dei cinque filosofi

Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso, e la tavola è apparecchiata con cinque bacchette. Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quella alla sua destra che condivide con il filosofo a destra e quella alla sua sinistra che condivide con il filosofo a destra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani del suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a mangiare.

Utilizziamo un array di 5 semafori tutti inizializzati a uno. Il filosofo i-esimo va a vedere se è disponibile la bacchetta a destra e quella a sinistra, si usa il modulo perché è circolare. Se riesce a prendere le due bacchette mangia altrimenti aspetta. Dopo che è riuscito a mangiare, rilascia le bacchette con due signal, dopo aver mangiato pensa e poi torna a mangiare. Possiamo avere però un problema di starvation ma anche un problema di deadlock perché se tutti i filosofi hanno in mano una bacchetta, quella è una situazione di stallo e stanno tutti fermi e moriranno di fame.

Per correttezza vediamo la struttura del filosofo i:

do {

wait (bacchetta [i]);

wait (bacchetta [(i + 1) % 5]);

…

Mangia

…

Signal (bacchetta [i]);

signal (bacchetta [(i + 1) % 5]);

…

Pensa

….

} while (true);

MONITOR: Neanche l’uso dei semafori, purtroppo, esclude la possibilità che si verifichi qualche errore di sincronizzazione. Per rimediare a questi errori, i ricercatori hanno sviluppato costrutti con un linguaggio ad alto livello. Un costrutto fondamentale di sincronizzazione ad alto livello è il tipo **monitor**.

Sintassi:

monitor **nome­**\_**monitor**{

*dichiarazioni di variabili condivise*

procedure P1 (…) {

…

}

Procedure P2 (…) {

…

}

.

.

.

Procedure Pn (..) {

…

}

*Codice d’inizializzazione* (…) {

…

}

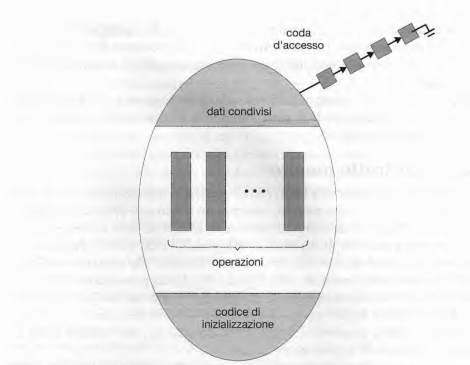
}

Un **monitor** è un modulo software che contiene una o più procedure, una sequenza di inizializzazione e dei dati locali. Le caratteristiche principali di un **monitor** sono:

* Le variabili locali sono accessibili solo dalle procedure del monitor e non dalle procedure esterne;
* Un processo entra nel monitor chiamando una delle sue procedure;
* Solo un processo alla volta può essere in esecuzione nel monitor;
* Ogni altro processo che ha chiamato il monitor è sospeso nell’attesa che questo diventi disponibile.

Con questo sistema garantiamo che nel monitor ci sia uno ed un solo processo in esecuzione e quindi la **mutua esclusione**.

Schema di un **monitor**:



Un monitor deve contenere degli strumenti di sincronizzazione, sono perciò necessari ulteriori meccanismi, in questo caso, un monitor fornisce la sincronizzazione mediante l’uso di **variabili di condizione** (*condition*) accessibili solo all’interno del modem.

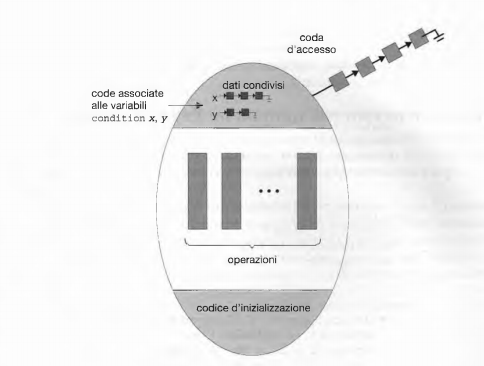
Un programmatore che deve scrivere un proprio schema di sincronizzazione può definire una o più **variabili condizionali** tramite il costrutto **condition**:

**condition** x, y;

Le uniche operazioni eseguibili su una variabile condition sono wait() e signal().

L’operazione **x.wait()** implica che il processo che la invoca rimanga sospeso finché un altro processo non invochi l’operazione **x.signal()** che risveglia esattamente un processo sospeso.

Le due operazioni sono quindi diverse da quelle sui semafori; infatti, se non ci sono processi sospesi **signal** non ha alcun effetto, contrariamente alla signal sui semafori.



Soluzione al problema dei cinque filosofi per mezzo di monitor:

monitor fc {

enum {pensa, affamato, mangia} stato [5];

//i tre possibili stati in cui si può trovare un filosofo;

condition auto [5];

void prende (int i) { //supponiamo che il processo i-esimo decide di prendere una bacchetta

stato[i] = affamato; //dice agli altri che è affamato

verifica(i); //va a vedere se sono disponibili le bacchette

if (stato[i] != mangia) // se sono disponibili andrà in stato mangia altrimenti

deve attendere

auto[i].wait();

}

void posa (int i) { //supponiamo che un filosofo ha finito di mangiare e vuole posare le bacchette

stato[i] = pensa; //dice agli altri che ha mangiato e si mette a pensare

verifica((i + 4) % 5); //verifica il filosofo a sinistra se vuole mangiare e gli molla la bacchetta sinistra

verifica((i + 1) % 5); //verifica il filosofo a destra se vuole mangiare e gli molla la bacchetta destra

}

void verifica (int i) { //verifica per il filosofo i se le bacchette alla sua destra e alla sua sinistra sono disponibili

//controlla se il filosofo a sinistra e quello a destra di i stanno mangiando, se si significa che le bacchette non sono disponibili per mangiare

if((stato[(i + 4) % 5] != mangia) &&

(stato[i] == affamato) &&

(stato[(i + 1) % 5] != mangia)) {

stato[i] = mangia; //se sono disponibili mangio

auto[i].signal(); //sblocco il filosofo i se trovo che uno è bloccato

}

}

In questo modo risolviamo il problema dello stallo, ma abbiamo possibilità di attesa indefinita.

CAPITOLO 7

STALLO DEI PROCESSI

INTRODUZIONE: In un ambiente multiprogrammazione più processi possono competere per ottenere un numero finito di risorse; se una risorsa non è correntemente disponibile, il processo richiedente passa allo stato d’attesa. Se le risorse richieste sono trattenute da altri processi, a loro volta nello stato d’attesa, il processo potrebbe non cambiare più il suo stato. Situazioni di questo tipo sono chiamate di **stallo** (*deadlock*).

MODELLO DEL SISTEMA: Un sistema è composto da un numero finito di risorse da distribuire tra più processi in competizione. Le risorse sono suddivise in tipi differenti, ciascuno formato da un certo numero di istanze.

Prima di adoperare una risorsa, un sistema deve richiederla e, dopo averla usata, deve rilasciarla. Nelle ordinarie condizioni di funzionamento un processo può servirsi di una risorsa soltanto se rispetta la seguente sequenza:

1. **Richiesta**. Se la richiesta non si può soddisfare immediatamente, per esempio. Perché la risorsa è attualmente in possesso di un altro processo, il processo richiedente deve attendere finché non possa acquisire tale risorsa.
2. **Uso**. Il processo può operare sulla risorsa, per esempio, la risorsa è una stampante, il processo può effettuare la stampa.
3. **Rilascio**. Il processo rilascia la risorsa.

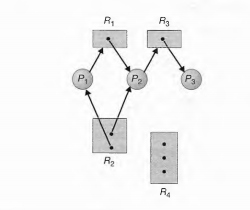
La **richiesta** ed il **rilascio** di risorse avvengono tramite chiamate di sistema. Come per esempio le chiamate di sistema request() e release(), open() e close(), allocate() e free(). La richiesta e il rilascio di altre risorse si possono eseguire per mezzo delle operazioni wait() e singnal().

CARATTERIZZAZIONE DELLE SITUAZIONI DI STALLO: Una situazione di **stallo** (*deadlock*) può verificarsi solo se si verificano contemporaneamente tutte e quattro le seguenti condizioni:

1. **Mutua esclusione**. Almeno una risorsa deve essere non condivisibile, vale a dire che è utilizzabile da un solo processo alla volta. Se un altro processo richiede tale risorsa, si deve ritardare il processo richiedente fino al rilascio della risorsa.
2. **Possesso e attesa.** Un processo in possesso di almeno una risorsa attende di acquisire risorse già in possesso di altri processi.
3. **Impossibilità di prelazione.** Non esiste un diritto di prelazione sulle risorse, vale a dire che una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. **Attesa circolare.** Deve esistere un insieme {P0,P1,…,Pn} di processi, tale che P0 attende una risorsa posseduta da P1, P1 attende una risorsa posseduta da P2, …, Pn-1 attende una risorsa posseduta da Pn e Pn attende una risorsa posseduta da P0.

GRAFO DI ASSEGNAZIONE DELLE RISORSE: Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una presentazione detta **grafo di assegnazione delle risorse**. Graficamente ogni processo Pi si rappresenta con un cerchio e ogni tipo di risorsa Rj si rappresenta con un rettangolo. Giacché il tipo di risorsa Rj può avere più istanze, ciascuna di essa si rappresenta con un puntino all’interno del rettangolo. Quando una freccia parte dal processo Pi e punta ad un tipo di risorsa Rj parliamo di **arco di richiesta**, mentre quando una freccia parte da un’istanza di una risorsa Rj e punta ad un processo Pi parliamo di **arco di assegnazione**.

Esempio:



Stati dei processi:

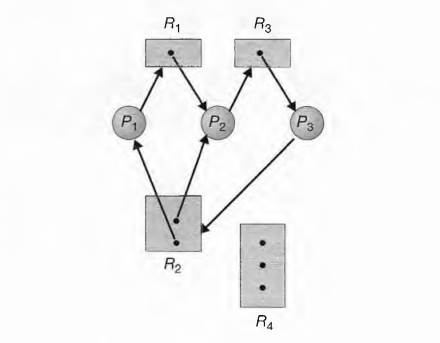
* Il processo P1 possiede un’istanza del tipo di risorsa di R2 e attende un’istanza del tipo di risorsa R1;
* Il processo P2 possiede un’istanza del tipo di risorsa R1 ed R2 e attende un’istanza del tipo di risorsa di R3;
* Il processo P3 possiede un’istanza del tipo di risorsa R3.

Nel momento in cui P3 ha finito di usare l’istanza del tipo R3 la restituisce e viene assegnata a P2 che a sua volta, quindi, può completare la sua esecuzione, a questo punto le istanze del tipo R1, R2 ed R3 vengono rilasciate e a questo punto R1 viene assegnata a P1 che può completare la sua esecuzione.

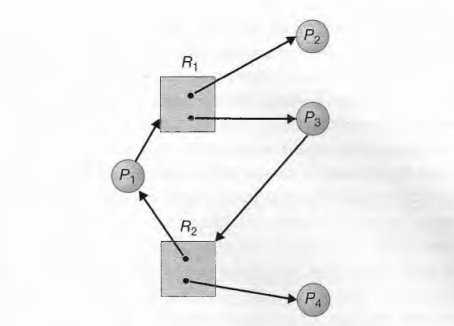
Data la definizione di **grafo di assegnazione delle risorse**, è facile mostrare che, se il grafo non contiene cicli, nessun processo del sistema subisce uno stallo; se il grafo contiene un ciclo, può sopraggiungere uno stallo:

1. Se ciascun tipo di risorsa ha esattamente un’istanza, allora l’esistenza di un ciclo implica la presenza di uno stallo;
2. Se ogni tipo di risorsa ha più istanze, un ciclo non implica necessariamente uno stallo.

Grafo di assegnazione delle risorse con un ciclo in presenza di deadlock:



Grafo di assegnazione delle risorse con un ciclo, ma senza stallo:



METODI PER LA GESTIONE DELLE SITUAZIONI DI STALLO: Essenzialmente, il problema delle situazioni di stallo si può affrontare impiegando tre metodi:

* Si può usare un protocollo per prevenire o evitare le situazioni di stallo, assicurando che il sistema non entri **mai** in stallo.
  + **Prevenire le situazioni di stallo** significa far uso di metodi atti ad assicurare che non si verifichi almeno una delle quattro condizioni necessarie che procurano deadlock (mutua esclusione, possesso e attesa, impossibilità di prelazione e attesa circolare.
  + Per **evitare le situazioni di stallo** occorre che il sistema operativo abbia in anticipo informazioni aggiuntive riguardanti le risorse che un processo richiederà e userà durante le sue attività.

Ciò comporta un basso utilizzo di risorse e un throughput ridotto.

* Si può permettere al sistema di entrare in stallo, individuarlo, e quindi eseguire il ripristino. In questo caso bisogna saper:
  + **Determinare la presenza di un deadlock**;
  + **Ripristinare il sistema da un deadlock**;

Ultima soluzione:

* Si può ignorare del tutto il problema, “fingendo” che le situazioni di stallo non possano mai verificarsi nel sistema.

Quest’ultima è la soluzione usata dalla maggior parte dei sistemi operativi, compresi UNIX e Windows.

CAPITOLO 8

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. 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*).

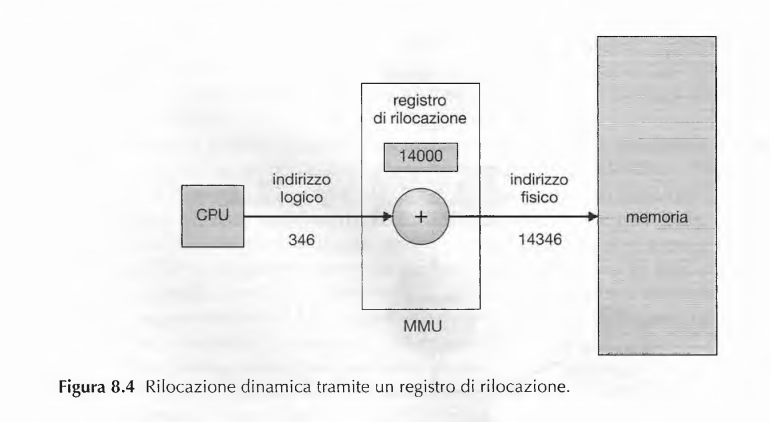
La procedura normale consiste nello scegliere uno dei processi appartenenti alla coda d’ingresso e nel caricarlo in memoria. Il processo durante l’esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, la memoria che gli era stata allocata viene deallocata. Generalmente gli indirizzi del programma sorgente sono simbolici. Un compilatore di solito **associa** (*bind*) questi indirizzi simbolici a indirizzi rilocabili. L’editor dei collegamenti (*linkage editor*), o il caricatore (*loader*), fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti.

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, in un secondo momento, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice.
* **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.

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** (*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 della fase d’**esecuzione**, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso si ci riferisce, di solito, agli indirizzi logici col termine **indirizzi virtuali**.

L’associazione nella fase d’**esecuzione** dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria** (*MMU*). 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. Il programma utente non considera mai gli indirizzi fisici reali e l’utente genera solo indirizzi logici e “pensa” che il processo sia eseguito nelle posizioni da 0 a *max*.

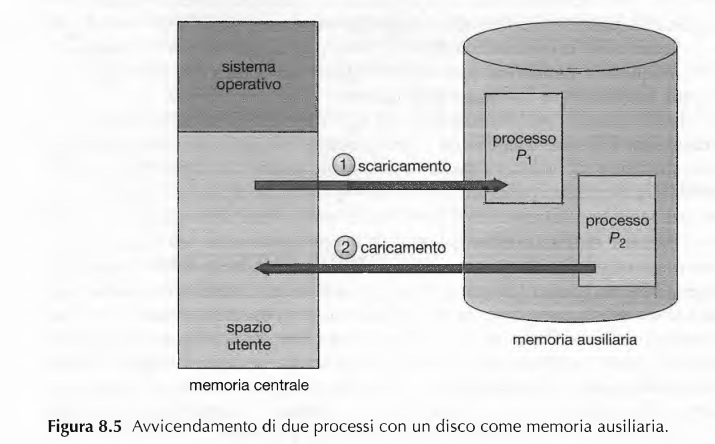


CARICAMENTO DINAMICO: Per migliorare l’utilizzo della memoria si può ricorrere al **caricamento dinamico** (*dynamic loading*), mediante il quale si carica una procedura solo quando viene richiamato; tutte le procedure si tengono in memoria secondaria in un formato di caricamento rilocabile. Si carica 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 **loader** 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. Non richiede un intervento particolare del sistema operativo, infatti spetta agli utenti progettare i programmi in modo da trarre vantaggio da un metodo di questo tipo.

COLLEGAMENTO DINAMICO E LIBRERIE CONDIVISE: Alcuni sistemi operativi consentono solo il **collegamento statico**, con il **collegamento dinamico**, invece, viene posticipato il collegamento alle librerie fino al momento dell’esecuzione. Con il collegamento dinamico 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. Con questo metodo tutti i processi che usano una libreria del linguaggio si limitano a eseguire la stessa copia del codice della libreria. Il collegamento dinamico richiedere 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.

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. In una situazione di timesharing, per esempio, ci sono normalmente molti utenti e la memoria centrale non è sempre sufficiente a contenere tutti i loro processi. Per eseguire questi processi bisogna spostarli dal disco alla memoria centrale e viceversa. Questo metodo prende il nome di **avvicendamento dei processi** e ovviamente richiede una memoria ausiliaria, tale memoria deve essere abbastanza ampia da contenere le copie di tutte le immagini in memoria di tutti i processi utenti, e deve permettere un accesso diretto a dette immagini di memoria.

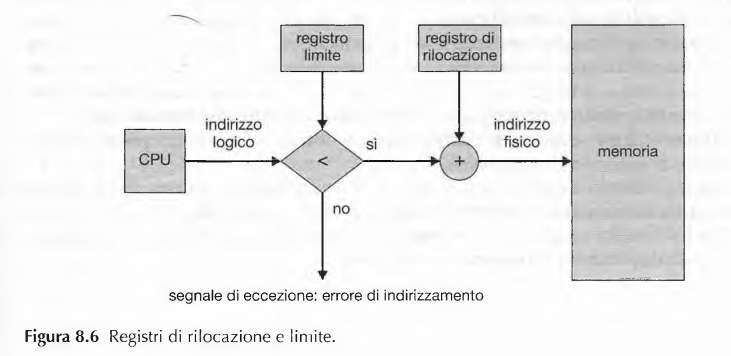


Una variante di questo criterio s’impiega per gli algoritmi di scheduling basati sulle priorità Se si presenta un processo con priorità maggiore, il gestore della memoria può scaricare dalla memoria centrale il processo con priorità inferiore per fare spazio all’esecuzione del processo con priorità maggiore. Quando il processo con priorità maggiore termina, si può ricaricare in memoria quello con priorità minore e continuare la sua esecuzione. Questo swapping è talvolta chiamato **roll out**, **roll in**. Se l’associazione degli indirizzi logici agli indirizzi fisici si effettua nella fase di assemblaggio o caricamento, il processo non può essere ricaricato in posizioni diverse. Se invece si effettua nella fase d’esecuzione, un processo può essere riservato in uno spazio di memoria diverso, poiché gli indirizzi fisici si calcolano nella fase d’esecuzione. Il sistema mantiene una **coda dei processi pronti** (*ready queue*) formata da tutti i processi pronti per l’esecuzione. 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, allora 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. La maggior parte del tempo d’avvicendamento è data dal tempo di trasferimento. Il tempo di trasferimento totale è direttamente proporzionale alla quantità di memoria interessata. Perciò sarebbe utile sapere esattamente quanta memoria sia effettivamente usata da un processo utente e non solo quanta questo potrebbe teoricamente usarne. Per scaricare un processo, inoltre, è necessario essere certi che sia completamente inattivo.

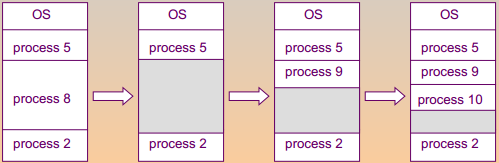
ALLOCAZIONE DELLA MEMORIA: La memoria centrale di solito si divide in due partizioni:

* **Sistema operativo residente**, generalmente mantenuto nella parte bassa della memoria, il fattore che incide in modo decisivo su tale scelta è generalmente la posizione del vettore delle interruzioni.
* **Processi utente**, generalmente mantenuti nella parte alta della memoria.

Per quanto riguarda la memoria, la protezione del Sistema Operativo dai processi utenti e la protezione dei processi utenti dagli altri processi utenti si può usare usando un registro di rilocazione insieme con un registro limite. Il **registro di rilocazione** contiene il valore dell’indirizzo fisico minore, il **registro limite** contiene l’intervallo di indirizzi logici e ciascun indirizzo logico deve essere minore del contenuto del registro limite.



ALLOCAZIONE CONTINUA DELLA MEMORIA: Uno dei metodi più semplici per l’allocazione della memoria consiste nel suddividere la stessa in **partizioni** di dimensione fissa. Ogni partizione deve contenere esattamente un processo. Con il **metodo delle partizioni multiple** quando una partizione è libera può essere occupata da un processo presente nella coda d’ingresso; terminato il processo, la partizione diviene nuovamente disponibile per un altro processo. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un **buco** (*hole*). Ovviamente la memoria contiene una serie di buchi di diverse dimensioni. Quando un processo viene caricato in memoria, gli viene assegnata la memoria di un **buco** sufficientemente grande da 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; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formare uno più grande. Questa procedura di divisione della memoria è chiamata **frammentazione** perché si “frammenta” lo spazio libero in tante piccole parti.



I criteri più usati per scegliere un buco libero tra quelli disponibili nell’insieme sono:

* **First - fit**. Si assegna il **primo** buco abbastanza grande. La ricerca può cominciare sia dall’inizio dell’insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s’individua il 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.

Con l’uso di simulazioni si è dimostrato che sia **first – fit** sia **best – fit** sono miglior rispetto a **worst – fit** in termini di risparmio di tempo e di utilizzo della memoria.

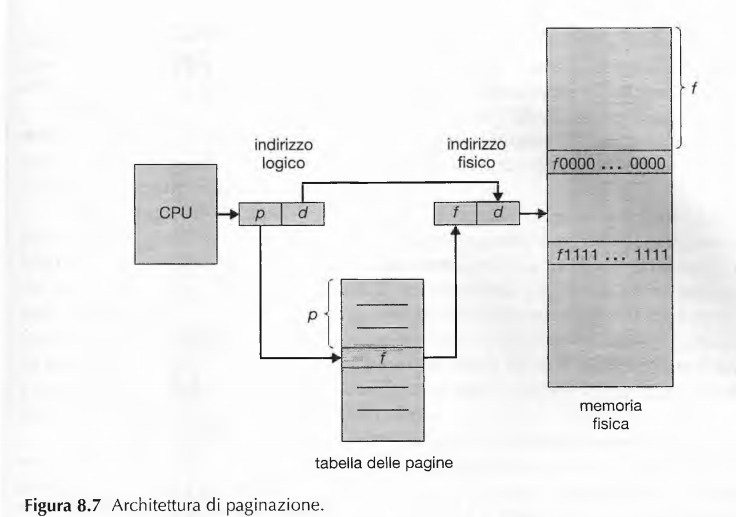
FRAMMENTAZIONE: Possiamo avere due tipi di frammentazione:

* **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.
* **Frammentazione interna**: quando la memoria allocata è leggermente più grande di quella richiesta, questa memoria residua, interna alla partizione, resterà inutilizzata.

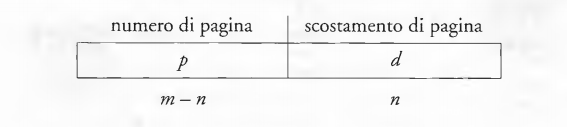
Una soluzione al problema della frammentazione esterna è data dalla **compattazione**. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. La compattazione tuttavia non è sempre possibile, infatti non si può realizzare se la rilocazione è statica e si compie nella fase di assemblaggio o di caricamento, è possibile solo se la rilocazione è dinamica e si compie nella fase d’esecuzione.

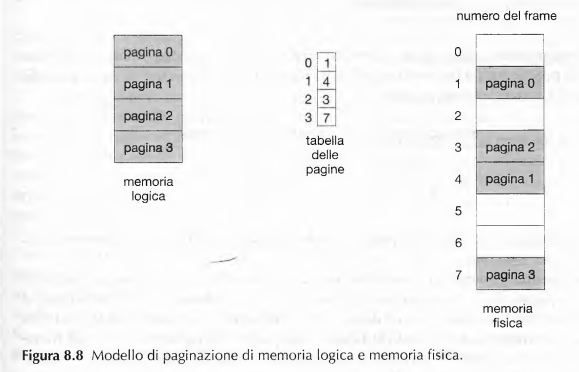
PAFINAZIONE: La **paginazione** è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo.

Il metodo 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. Ogni indirizzo generato dalla CPU è diviso in due parti: un **numero di pagine** (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 l’offset di pagina per definire l’indirizzo della memoria fisica, che s’invia all’unità di memoria.



La **paginazione** non è altro che una forma di rilocazione dinamica. Ogni indirizzo logico è associato ad un indirizzo fisico dell’hardware di paginazione. La paginazione è simile all’utilizzo di una tabella di registri di base, uno per ciascun frame di memoria. Con la paginazione ciascun frame libero può essere utilizzato ed allocato ad un processo che deve essere eseguito. I frame vengono allocati come unità, se i requisiti di memoria non combaciano con i limiti di pagina, l’ultimo frame allocato può non essere completamente pieno. Il caso peggiore si presenta quando un processo necessita di **n** pagine più **un byte**, anche in questo caso bisogna allocare **n + 1** pagine. Di solito, in media, con la paginazione si ha una frammentazione interna di mezza pagina per processo. 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. Se la dimensione dello spazio degli indirizzi logici è 2m e la dimensione di una pagina è di 2n unità di indirizzamento, allora **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 quindi ha la forma seguente:

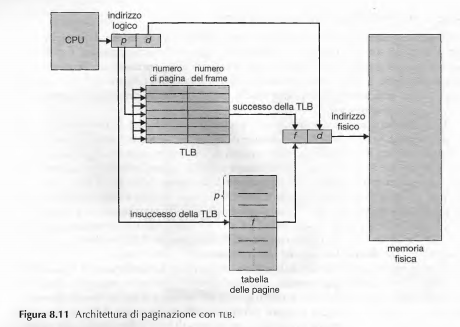




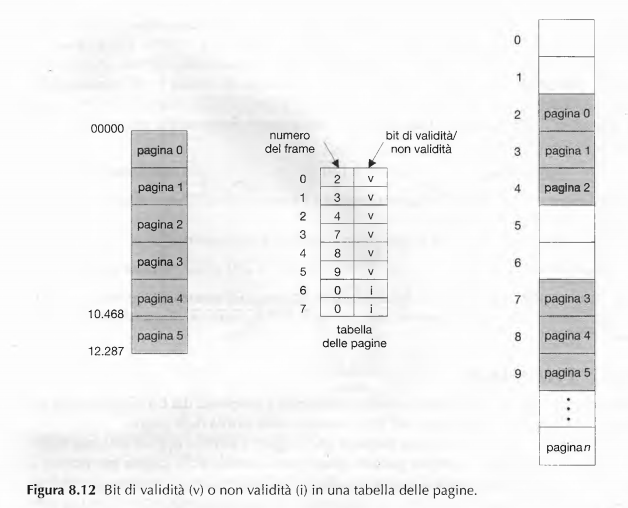
ARCHITETTURA DI PAGINAZIOE: Ogni sistema operativo segue metodi propri per memorizzare la tabella delle pagine. La maggior parte dei sistemi impiega una tabella delle pagine per ciascun processo. L’architettura d’ausilio della tabella delle pagine si può realizzare in modi diversi. Nel caso più semplice, si usa uno specifico insieme di **registri**. L’uso dei registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola. La maggior parte dei calcolatori contemporanei usa comunque tabelle molto grandi, 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. Questo metodo presenta un problema connesso al tempo necessario di accesso a una locazione della memoria utente. Infatti, per accedere a un byte occorrono **due** accessi alla memoria, uno per l’elemento nella tabella delle pagine e uno per il byte stesso. La soluzione a questo tipico 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 in 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 nella TLB non è presente il numero di pagina, situazione nota come **insuccesso della TLB** (*TLB miss*), si deve consultare la tabella delle pagine in memoria. La percentuale di volte che un numero di pagina si trova nella TLB è detta **tasso di successi** (*hit ratio*).

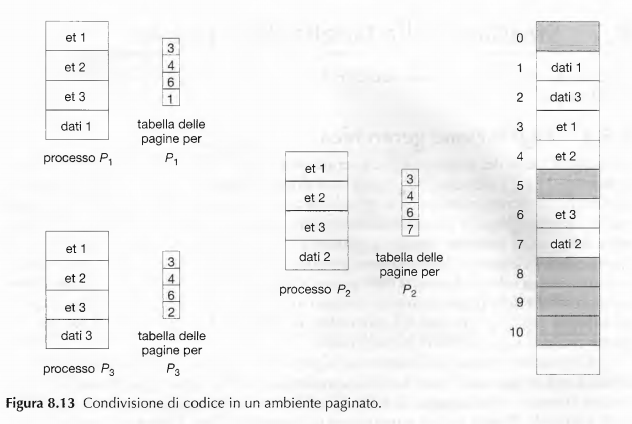
Se la TLB è piena, il sistema operativo deve scegliere un elemento da sostituire con il nuovo. I criteri di sostituzione variano dalla scelta dell’elemento usato meno recentemente (LRU) alla scelta casuale. Inoltre alcune TLB consentono che certi elementi siano **vincolati** (*wired down*), cioè non si possano rimuovere dalla TLB.



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. Nel caso si tenti di scrivere su una pagina di sola lettura viene generato un **trap** dell’hardware al sistema operativo. 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 di indirizzi logici del processo. Alcune architetture dispongono di registri, detti **registri di lunghezza della tabella delle pagine** (*page-table lenght register,* PTLR), per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare se quest’ultimo si trovi nell’intervallo valido per il 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. Si consideri un sistema con **n** utenti, ciascuno dei quali usa un elaboratore di testi. Se il codice rientrante, può essere condiviso, in questo caso, un elaboratore di testi viene condiviso da più processi, ciascuno dei quali dispone della propria pagina dei dati.



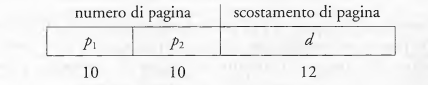
Il **codice rientrante**, detto anche **codice puro**, è un codice non automodificante, cioè che non cambia durante l’esecuzione. Quindi, due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari alla propria esecuzione.

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: La maggior parte dei moderni calcolatori dispone di uno spazio d’indirizzi logici molto grande. In un ambiente di questo tipo, la stessa tabella delle pagine finirebbe per diventare eccessivamente grande. 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:

* **Paginazione gerarchica;**
* **Tabella delle pagine di tipo hash;**
* **Tabella delle pagine invertita.**

PAGINAZIONE GERARCHICA: Il metodo consiste nell’adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata. 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.



…dove p1 è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e p2 è lo scostamento all’interno della pagina indicata dalla tabella esterna delle pagine. Poiché la traduzione degli indirizzi logici si svolte dalla tabella esterna delle pagine verso l’interno, questo metodo è anche noto come **tabella delle pagine ad associazione diretta**.

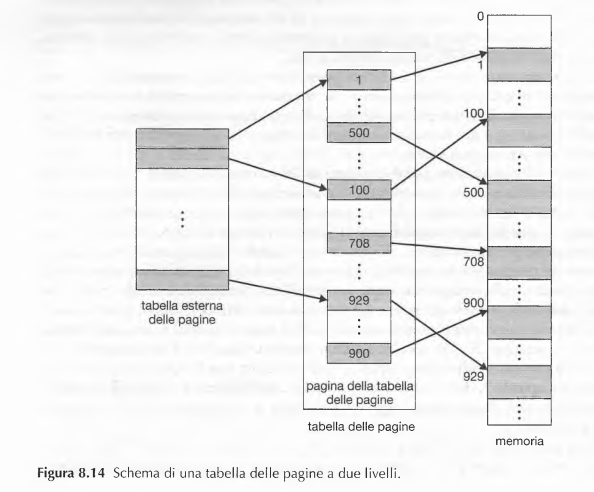


TABELLA DELLE PAGINE DI TIPO HASH: Un metodo di gestione molto comune degli spazi d’indirizzi relativi ad architetture oltre i 32 bit consiste nell’impiego di una **tabella delle pagine di tipo hash**, in cui l’argomento della funzione hash è il numero della pagina virtuale. Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione. Ciascun elemento è composto da tre campi:

1. **Il numero della pagina virtuale;**
2. **L’indirizzo del frame 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 del primo elemento della lista concatenata corrispondente. Se i valori coincidono si usa l’indirizzo del relativo frame per generare l’indirizzo fisico desiderato. Altrimenti, l’algoritmo esamina allo stesso modo gli elementi successivi alla lista concatenata.

Le tabelle delle pagine di tipo hash sono particolarmente utili per gli spazi d’indirizzi **sparsi**, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio d’indirizzi.

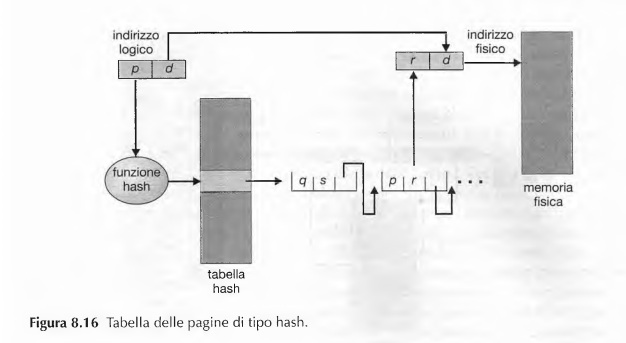
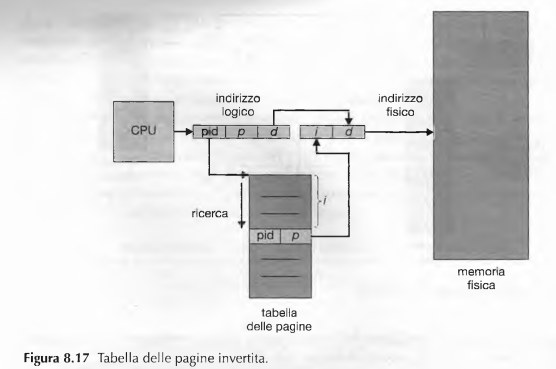


TABELLA DELLE PAGINE INVERTITA: Generalmente si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, oppure un elemento per ogni indirizzo virtuale a prescindere dalla validità di quest’ultimo. Questa è una rappresentazione 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 il valore. Uno degli inconvenienti è costituito dalla dimensione di ciascuna tabella delle pagine, che può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria proprio per sapere com’è impiegata la rimanente memoria fisica.

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 dall’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. Ciascun indirizzo virtuale è una tripla del tipo:

*< id – processo, numero di pagina, scostamento >*

Ogni elemento della tabella delle pagine invertita è una coppia <id-processo, numero di pagina> dove l’**id–processo** assume il ruolo di identificatore dello spazio d’indirizzi. Quando si fa un riferimento alla memoria, si presenta una parte dell’indirizzo virtuale, formato da <id-processo, numero di pagina>, al sottosistema di memoria. Quindi si cerca una corrispondenza nella tabella delle pagine invertita. Se si trova tale corrispondenza, per esempio sull’elemento **i**, si genera l’indirizzo fisico <**i**, scostamento>. In caso contrario è stato tentato un accesso illegale a un indirizzo.

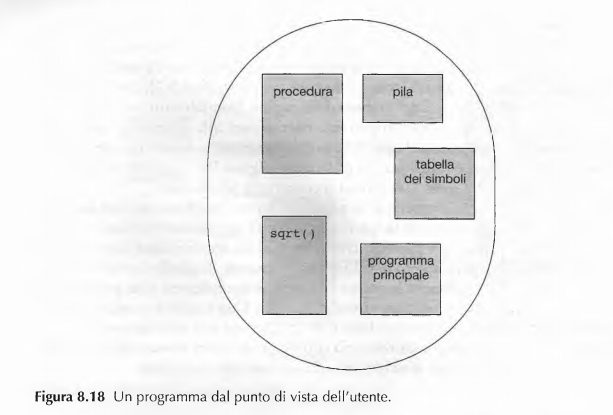


Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferimento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza occorre esaminare tutta la tabella: questa ricerca richiede molto tempo. Per limitare il problema si può impiegare una tabella hash, che riduce la ricerca a un solo, o pochi, elementi della tabella delle pagine.

SEGMENTAZIONE: 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 di indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indici specificano sia il nome sia lo scostamento all’interno del segmento, quindi l’utente fornisce ogni indirizzo come una coppia di valori: un nome di segmento e uno scostamento. 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 >

Un compilatore per il linguaggio C può creare segmenti distinti per i seguenti elementi di un programma: il codice, le variabili globali, lo heap, le pile, la libreria standard del C, ecc.

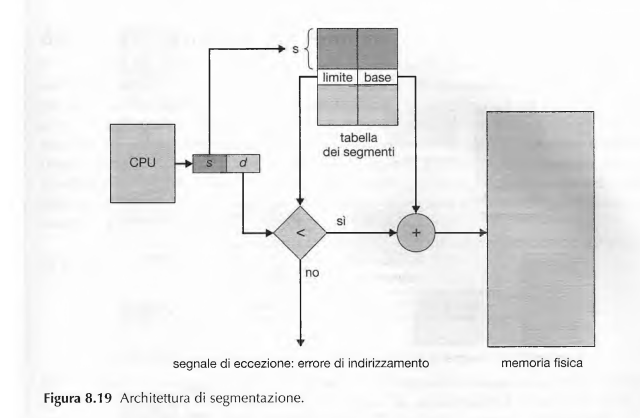


ARCHITETTURA DI SEGMENTAZIONE: La **tabella dei segmenti** mappa, in uno spazio ad una dimensione (memoria fisica), gli indirizzi logici. Ogni suo elemento è una coppia ordinata:

* **La base del segmento:** contiene l’indirizzo fisico iniziale della memoria dove il segmento risiede;
* **Il limite del segmento**: contiene la lunghezza del segmento.

Avremo anche:

1. **Segment-table base register (STBR)**: punta alla locazione in memoria della tabella dei segmenti.
2. **Segment-table lenght register (STLR)**: indica il numero di segmenti di un processo.

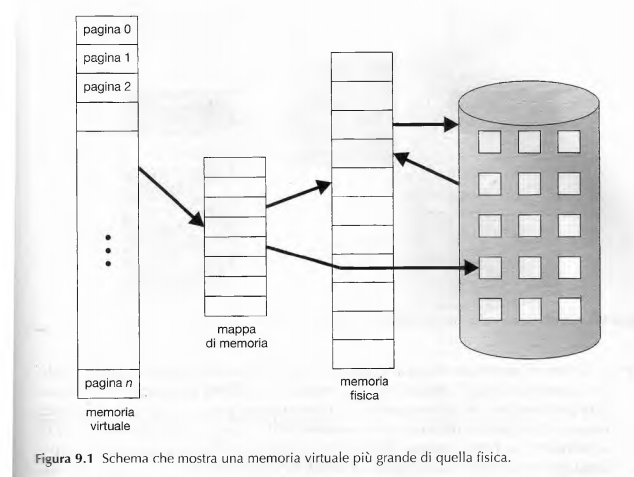


CAPITOLO 9

INTRODUZIONE: La **memoria virtuale** è una tecnica che permette di eseguire processi che possono anche non risiedere completamente in memoria. La **memoria virtuale** si fonda sulla separazione della memoria logica percepita dall’utente dalla memoria fisica. Questa separazione permette di offrire ai programmatori una memoria virtuale molto ampia, anche se la memori fisica disponibile è più piccola. Solo parte del programma viene effettivamente caricato in memoria per essere eseguito. Lo spazio di indirizzi logici quindi può essere più grande dello spazio di indirizzi fisici. Permette di condividere lo stesso spazio di indirizzi tra più processi e tecniche più efficienti per la creazione di processi.

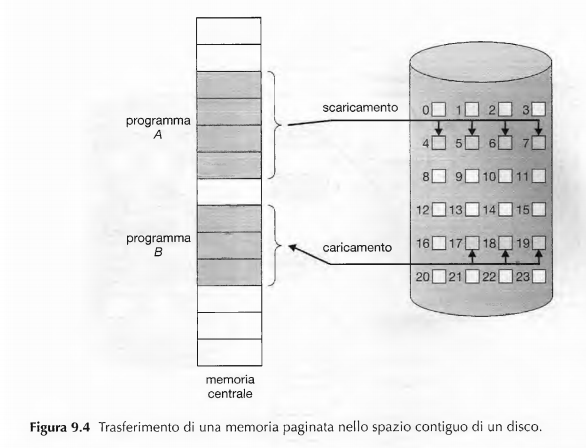
La memoria virtuale può essere implementata in due modi:

* **Paginazione su richiesta:** anziché effettuare swapping sulla memoria di tutto il processo, si utilizza uno swapper pigro, che non riversa mai una pagina in memoria a meno che non sia necessaria (**pager**);
* **Segmentazione su richiesta.**



PAGINAZIONE SU RICHIESTA: Anziché caricare in memoria l’intero processo, si caricano le pagine nel momento in cui servono realmente; si tratta di una tecnica detta **paginazione su richiesta**, comunemente adottata dai sistemi con memoria virtuale. Se il processo cerca di accedere ad una pagina che non è stata ancora caricata in memoria, il sistema operativo genere un’eccezione di pagina mancante (**page fault trap**):

* Se il riferimento non era valido, si termina il processo (abort);
* Se il riferimento era valido, si porta la pagina in memoria (swap in).

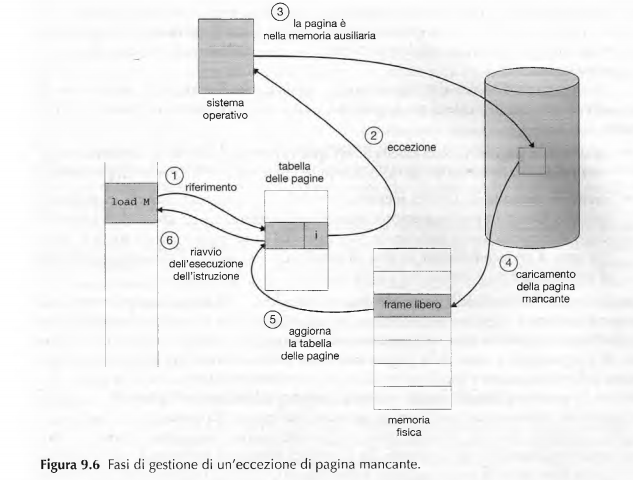


CONCETTI FONDAMENTALI: Quando un processo sta per essere caricato in memoria, il paginatore ipotizza quali pagine saranno usate, prima che il processo sia nuovamente scaricato dalla memoria. Così anziché caricare in memoria tutto il processo, si trasferisce in memoria solo le pagine che ritiene necessarie. 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, in questo caso, il bit impostato come “valido” significa che la pagina corrispondente è valida ed è presente in memoria; il bit impostato come “non valido” indica che la pagina non è valida oppure è valida ma è attualmente nel disco. Se l’ipotesi del paginatore è esatta e si caricano 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*).

La procedura di gestione dell’eccezione di pagina mancante è chiara, e corrisponde ai passi seguenti:

1. Si controlla una tabella interna per questo processo; in genere 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 segna 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.



I meccanismi d’ausilio alla paginazione su richiesta che l’architettura del calcolatore deve offrire sono quelli richiesti per la paginazione e l’avvicendamento dei processi in memoria:

* **Tabella delle pagine**. Questa tabella ha la capacità di contrassegnare un elemento come non valida attraverso un bit di validità oppure un valore speciale dei bit di protezione;
* **Memoria secondaria**. Questa memoria conserva le pagine non presenti in memoria centrale. Generalmente la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo d’avvicendamento; la selezione del disco usata a questo scopo si chiama **area d’avvicendamento**, o **area di scambio** (*swap space*).

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. Attualmente, nella maggior parte dei calcolatori il tempo d’accesso alla memoria, che si denota con **ma**, va da 10 a 200 nanosecondi. Finché non si verifichino assenze di pagine, il tempo d’accesso effettivo è uguale al tempo d’accesso alla memoria.

Supponendo che **p** sia la probabilità che si verifichi un’assenza di pagina (), 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:

*tempo d’accesso effettivo* =((1 – **p**) x **ma**) + (**p** x *tempo di gestione dell’assenza di pagina*))

Per calcolare il tempo d’accesso effettivo occorre conoscere il tempo necessario alla gestione di un’assenza di pagina. Alla presenza di un’assenza di pagina si esegue la seguente sequenza:

1. Segnale d’eccezione al sistema operativo;
2. Salvataggio dei registri utente e dello stato del processo;
3. Verifica che l’interruzione sia dovuta o meno a una pagina mancante;
4. Controllo della correttezza del riferimento alla pagina e determinazione della locazione della pagina nel disco;
5. Lettura dal disco e trasferimento in un frame libero:
   1. Attesa nella coda relativa a questo dispositivo finché la richiesta di lettura non sia servita;
   2. Attesa del tempo di posizionamento e latenza del dispositivo;
   3. Inizio del trasferimento della pagina in un frame libero;
6. Durante l’attesa, allocazione della CPU a un altro processo utente (scheduling della CPU, facoltativo);
7. Ricezione di un’interruzione del controllore del disco (I/O completato);
8. Salvataggio dei registri e dello stato dell’altro processo utente (se è stato eseguito il passo 6);
9. Verifica della provenienza dell’interruzione dal disco;
10. Aggiornamento della tabella delle pagine e di altre tabelle per segnalare che la pagina richiesta è attualmente presente in memoria;
11. Attesa che la CPU sia nuovamente assegnata a questo processo;
12. Recupero dei registri utente, dello stato del processo e della nuova tabella delle pagine, quindi ripresa dell’istruzione interrotta.

Non sempre sono necessari tutti i passi sopra elencati. In ogni caso, il tempo di servizio dell’eccezione di pagina mancante comporta tre operazioni principali:

1. Servizio del segnale di eccezione di pagina mancante;
2. Lettura della pagina;
3. Riavvio del processo.

Considerando un tempo medio di servizio dell’eccezione di pagina mancante di 25 millisecondi e un tempo d’accesso alla memoria di 200 nanosecondi, il tempo effettivo d’accesso in nanosecondi è il seguente:

*tempo d’accesso effettivo* = (1 – **p**) x 200 + **p** (25 millisecondi)

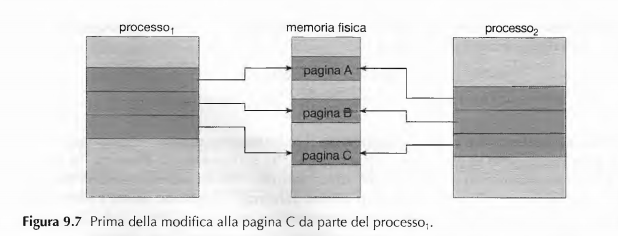
= (1 – **p**) x 200 + **p** x 25.000.000

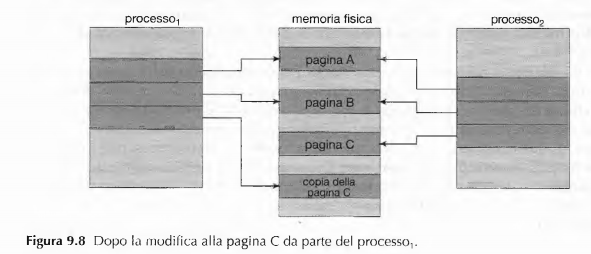
= 200 + **p** x 24.999.900

Il tempo d’accesso effettivo è direttamente proporzionale alla **frequenza delle assenze di pagine** (*page-fault rate*). Se un accesso su 100 accusa un’assenza di pagina, il tempo d’accesso effettivo è di 25 microsecondi. Se si desidera un rallentamento inferiore al 10 %, occorre che **p** sia minore di 0,0000004. Ciò significa, permettere un’assegna di pagina ogni 2.500.000 accessi alla memoria.

COPIATURA SU SCRITTURA: La memoria virtuale offre altri vantaggi per quel che riguarda la creazione ed esecuzione dei processi:

* **Copiatura in 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 scrittore, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina. Se uno dei due processi modifica una pagina, solo allora viene duplicata. Quando una pagina deve essere duplicata, si attinge ad un pool di pagine libere, che di solito si assegnano quando la pila o il cosiddetto **heap** di un processo devono espandersi. L’allocazione di queste pagine di solito avviene secondo una tecnica nota come **azzeramento su richiesta** (zero-fill-on-demand).Questa tecnica permette una creazione dei processi molto rapida e minimizza il numero di pagine che si devono assegnare ad un nuovo processo.





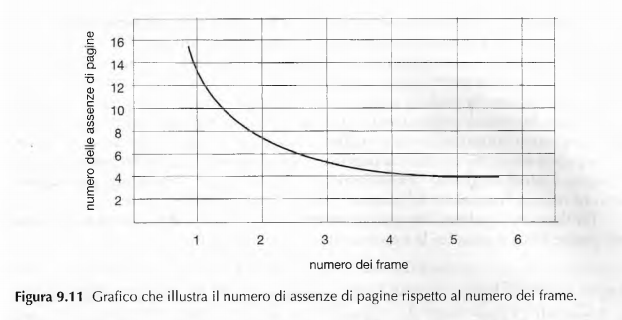
* **Associazione dei file alla memoria** (Memory-Mapped Files). Permette di trattare l’I/O di file su disco come un generico accesso a memoria facendo corrispondere un blocco del disco ad una pagina (o più pagine) di memoria. L’accesso a file avviene inizialmente tramite l’ordinario meccanismo di paginazione su richiesta, cui segue, però, il caricamento in una pagina fisica di una porzione di file della dimensione di una pagina, leggendola dal file system. Le operazioni successive di lettura e scrittura si gestiscono come normali accessi alla memoria. Questa tecnica permette inoltre l’associazione dello stesso file alla memoria virtuale di più processi allo scopo di condividere i dati. Le chiamate a sistema per l’associazione dei file alla memoria possono anche disporre della funzione di copiatura su scrittura, permettendo ai processi di condividere un file per la lettura, mantenendo però una propria copia dei dati da esso modificati.

SOSTITUZONE DELLE PAIGNE: In presenza di un page fault dovuto ad un riferimento ad una pagina non in memoria, quindi, si dovrà portare la pagina in memoria. Se nessun blocco di memoria è libero, si potrà liberarne uno inutilizzato e modificare la tabella delle pagine per indicare che quella pagina non è più in memoria. Il blocco di memoria liberato si potrà usare per caricare la pagina che ha causato l’eccezione.

L’architettura fisica del sistema di calcolo può disporre di un **bit di modifica** (modify-dirty bit) associato ad ogni pagina, che mette a 1 automaticamente ogni volta che quella pagina viene modificata. In questo modo solo le pagine che vengono modificate andranno riscritte sul disco.

SCHEMA DI BASE: La procedura si servizio dell’eccezione di pagina mancante viene modificata in modo da includere l’eventuale sostituzione della pagina:

1. Si individua la locazione nel disco della pagina richiesta;
2. Si cerca un blocco di memoria libero:
   1. Se esiste, lo si usa;
   2. Altrimenti si impiega un algoritmo di sostituzione delle pagine per scegliere un blocco di memoria “vittima”;
   3. Si scrive la pagina “vittima” nel disco; si modificano adeguatamente la tabella delle pagine e dei blocchi di memoria.
3. Si scrive la pagina richiesta nel blocco di memoria appena liberato e si modificano le tabelle delle pagine e dei blocchi di memoria;
4. Si riavvia il processo utente.



Esistono diversi tipi di algoritmi usati per la sostituzione delle pagine:

SOSTITUZIONE DELLE PAGINE SECONDO L’ORDINE D’ARRIVO (FIFO)

Questo algoritmo associa ad ogni pagina l’istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si selezione quella presente in memoria da più tempo.

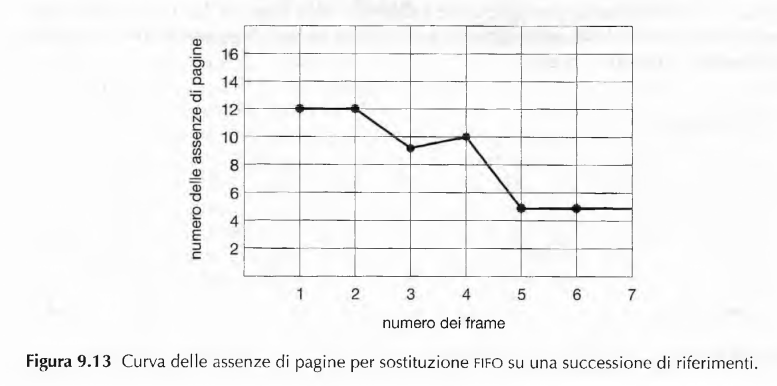
Esempio:



I frame iniziali sono tre, e sono vuoti. Considerando che inizialmente il caricamento di pagine in frame vuoti causano una eccezione di pagina, nel nostro esempio avremo ben 15 assenze di pagine.

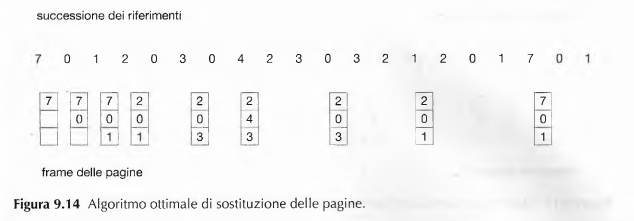
Le prestazioni di questo algoritmo non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe anche contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso.

In generale, può accadere che aumentando il numero di frame a disposizione, il numero di assenze di pagine aumenta a differenza di prima. Questo inatteso risultato è noto col nome di **anomalia di Belady**.



SOSTITUZIONE OTTIMALE 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. Semplicemente **si sostituisce la pagina che non si userà per il periodo di tempo più lungo**.

Esempio:



Notiamo che le assenze di pagina con questo algoritmo risultano essere solo nove.

Ignorando le prime tre assenze di pagine, che si verificano con tutti gli algoritmi, la sostituzione ottimale è due volte migliore rispetto all’algoritmo FIFO; nessun algoritmo di sostituzione può gestire questa successione di riferimenti a tre blocchi di memoria con meno di nove assenze di pagine. Sfortunatamente l’algoritmo è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti.

SOSTITUZIONE DELLE PAGINE USATE MENO RECENTEMENTE (LRU)

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 **algoritmo LRU** (last recently used).

La sostituzione LRU associa ad 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.

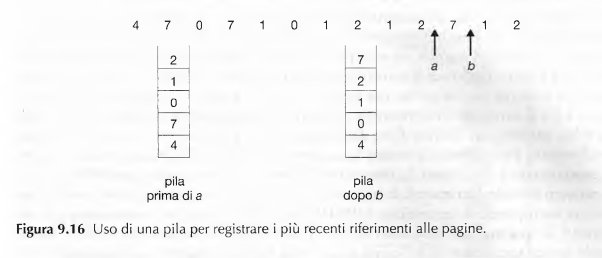
Esempio:



L’algoritmo LRU produce 12 assenze di pagine. Occorre notare che le prime cinque assenza di pagine solo le stesse della sostituzione ottimale.

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. Questo schema implica una ricerca all’interno della tabella delle pagine per individuare la pagina usata meno recentemente, e una scrittura in memoria per ogni accesso alla memoria. Occorre infine considerare il superamento della capacità del contatore (overflow).
* **Pila**. UN altro metodo prevede la presenza di una pila dei numeri delle pagine. Ogni volta che si fa riferimento ad 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. 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.



Esiste una classe di algoritmi chiamati **algoritmi a pila**, che non presenta l’anomalia di Belady. Un algoritmo a pila è un algoritmo per il quale è possibile mostrare che l’insieme delle pagine in memoria per **n** frame è sempre un *sottoinsieme* dell’insieme delle pagine che dovrebbero essere in memoria per **n + 1** frame.

SOSTITUZIONE DELLE PAGINE PER APPROSSIMAZIONE A LRU

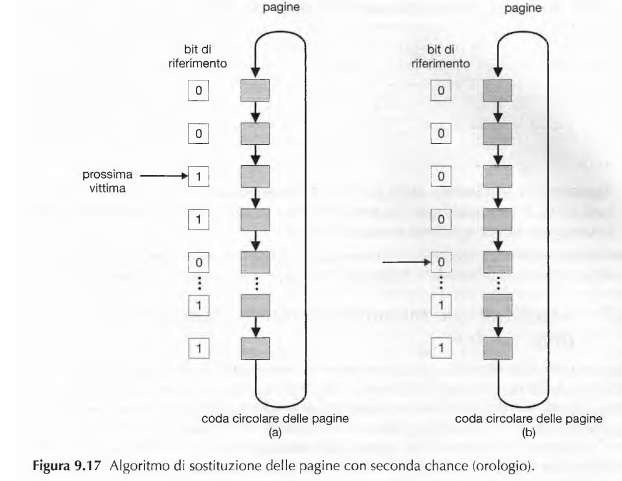
Molti sistemi tuttavia possono fornire un aiuto: un **bit di riferimento**. Il bit di riferimento a una pagina è impostato automaticamente dall’architettura del sistema ogni volta che si fa un riferimento a quella pagina, che sia una lettura o una scrittura. Inizialmente, il sistema operativo azzera tutti i bit. Quando si inizia l’esecuzione di un processo utente, l’architettura 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.

1. ALGORITMO CON BIT SUPPLEMENTARI DI RIFERIMENTO

Ulteriori informazioni sull’ordinamento si possono ottenere registrano 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 un segnale d’interruzione del time del sistema trasferisce il controllo al sistema operativo. A questo punto, sposta il **bit di riferimento** per ciascuna pagina nel bit più significativo della sequenza, traslando gli altri bit a destra di uno e scartando il bit meno significativo. Questi registri a scorrimento, ad esempio di 8 bit, contendono l’ordine d’uso delle pagine relativo agli ultimi otto periodi di tempo. Se il registro a scorrimento contiene 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. In ogni caso l’unicità dei numeri non è garantita, infatti, si possono sostituire tutte le pagine con il valore minore, oppure si può ricorrere a una selezione FIFO. Il numero dei bit può essere variato: si stabilisce secondo l’architettura disponibile per accelerare 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**.

1. ALGORITMO CON SECONDA CHANCE

Dopo aver selezionato una pagina, si controlla il bit di riferimento, se il suo valore è 0, si sostituisce la pagina; se invece è 1, si dà una seconda **chance** alla pagina e la selezione passa alla successiva pagina FIFO. Quando una pagina riceve una seconda chance, si azzera il suo bit di riferimento e si aggiorna il suo istante d’arrivo al momento attuale. 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 a 0; a ogni passo si azzera il bit di riferimento appena esaminato. Una volta trovata la pagina “vittima”, la si sostituisce e si inserisce la nuova pagina nella coda circolare nelle posizioni corrispondente. Se tutti i bit sono a 1, la sostituzione con seconda chance si riduce a una sostituzione FIFO.



1. ALGORITMO CON SECONDA CHANCE MIGLIROATO

L’algoritmo con seconda chance si può migliorare considerando i bit di riferimento e di modifica come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi:

1. (0, 0): né recentemente usato né modificato
   1. migliore pagina da sostituire;
2. (0, 1): non usato recentemente, ma modificato
   1. la pagina non così buona poiché prima di essere sostituita deve essere scritta in memoria secondaria;
3. (1, 0) usato recentemente ma non modificato
   1. probabilmente la pagina sarà presto ancora usata;
4. (1, 1) usato recentemente e modificato
   1. probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Ogni pagina rientra in una di queste quattro classi. Alla richiesta di una sostituzione di pagina, si usa lo stesso schema impiegato nell’algoritmo a orologio, ma anziché controllare se la pagina puntata ha il bit di riferimento a 1, si esaminano le classi cui la pagina appartiene e si sostituisce la prima pagina che si trova nella classe minima non vuota.

SOSTITUZIONE DELLE PAGINE BASATA SU CONTEGGIO

Si potrebbe usare un contatore del numero d riferimenti fatti a ciascuna pagina, e sviluppare i seguenti schemi:

1. **Algoritmo di sostituzione delle pagine meno frequentemente usate** (least frequently, LFU); richiede che si sostituisca la pagina con il conteggio più basso. La ragione di questa scelta è che una pagina usata attivamente deve avere un conteggio di riferimento alto. Il punto debole di questo algoritmo è rappresentato dai casi in cui una pagina è usata molto intensamente durante la fase iniziale di un processo, ma poi non viene più usata. Il suo conteggio sarà alto, quindi rimane in memori anche se non è più necessaria. Una soluzione è quella di formare un conteggio per l’uso medio con esponente decrescente;
2. **Algoritmo di sostituzione delle pagine più frequentemente usate** (most frequently, MFU); è basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è stata ancora usata.

ALGORITMI CON MEMORIZZAZIONE TRANSITORIA DELLE PAGINE

I sistemi hanno generalmente un gruppo di frame liberi (*pool of free frames*). Quando si verifica un’assenza di pagina, si sceglie innanzi tutto un frame vittima, ma prima che sia scritta in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima sia scritta in memoria secondaria. Quando nel seguito si scrive la vittima in memoria secondaria, si aggiunge il suo frame al gruppo dei frame liberi.

APPLICAZIONI E SOSTITUZIONE DELLA PAGINA

Alcuni Sistemi operativi permettono a certi programmi di utilizzare una partizione del disco come un array sequenziale di blocchi logici, senza ricorrere alle strutture di dati del file system. Un simile array è anche detto **disco di basso libello** (raw disk), e il relativo I/O è denominato I/O **di basso livello** (raw I/O). Il disco di basso libello salta tutti i servizi del file system.

ALLOCAZIONE DEI FRAME

Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame. Naturalmente, col diminuire del numero di frame allocati a ciascun processo aumenta la frequenza delle assenze di pagine, con conseguente rallentamento dell’esecuzione del processo.

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, **m**/**n** frame. Questo schema è chiamato **allocazione uniforme**.

Un’alternativa consiste nel conoscere che diversi processi hanno bisogno di quantità di memoria diverse. Per risolvere questo problema è possibile ricorrere all’**allocazione proporzionale**, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione.

Si supponga che **si** sia la dimensione della memoria virtuale per il processo **pi.** Si definisce la seguente quantità:

Quindi, se il numero totale dei frame disponibili è **m**, al processo **pi** si assegnano **ai** frame, dove **ai** è approssimativamente

Naturalmente è necessario scegliere ciascun ai in modo che sia un intero maggiore del numero minimo di frame richiesti dalla struttura della serie di istruzioni di macchina e in modo che la somma di tutti gli ai non sia maggiore di **m**.

Usando l’**allocazione proporzionale**, per suddividere 62 frame tra due processi, uno di 10 e uno di 127 pagine, si assegnano rispettivamente 4 e 57 frame, infatti:

10/137 x 62 ≡ 4

127/137 x 62 ≡ 57

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** permette che per un processo si scelga un frame per la sostituzione di tutti i frame, anche se quel frame è correntemente allocato ad un altro processo; un processo può dunque sottrarre un frame ad un altro processo. Può accadere che per un certo processo 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 risente di un problema, cioè, un processo non può controllare la propria frequenza di assenze di pagine, perché il loro comportamento dipende anche dal comportamento di paginazione di altri processi.
* **Sostituzione locale** richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame. Il numero di blocchi di memoria assegnati a un processo non cambia. Con l’algoritmo di sostituzione locale non abbiamo lo stesso problema precedente. 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.

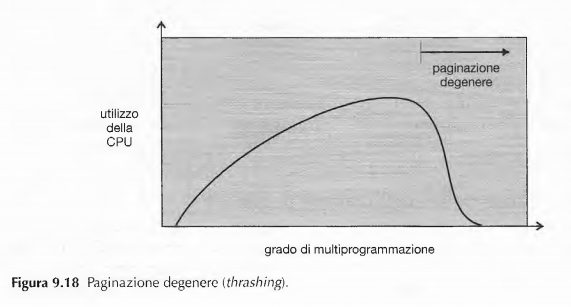
PAGINAZIONE DEGERENE (thrashing)

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. Questo comporta:

* Una riduzione dell’utilizzo della CPU;
* Il sistema operativo ritiene che sia necessario aumentare il livello di multiprogrammazione;
* Un altro processo aggiunto al sistema.

Anche se tecnicamente si può ridurre al valore minimo il numero dei frame allocati, esiste un certo numero di pagine in uso attivo. Nel momento in cui si deve sostituire una pagina, tutte le sue pagine sono attive, e quindi si deve sostituire una pagina che sarà immediatamente necessaria e di conseguenza si verificano subito parecchie assenze di pagine. Questa intensa quanto degenere paginazione, nota come **thrashing** si verifica quando si spende più tempo per la paginazione che per l’esecuzione dei processi.

Aumentando il grado di multiprogrammazione aumenta anche l’utilizzo della CPU, anche se più lentamente, fino a raggiungere un massimo. Se a questo punto si aumenta ulteriormente il grado di multiprogrammazione, l’attività di paginazione degenera e fa crollare l’utilizzo della CPU.

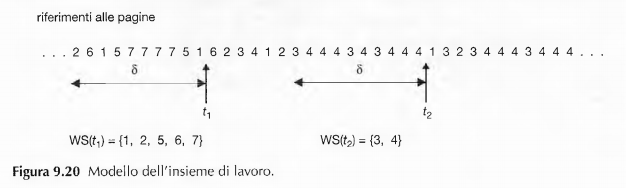


Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame “servano” a un processo si impiegano diverse tecniche.

Il metodo dell’insieme di lavoro (*working-set*), comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo metodo definisce il **modello di località** d’esecuzione di un processo. Questo modello stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente. Generalmente un programma è formato da parecchie località diverse, che sono sovrapponibili.

MODELLO DELL’INSIEME DI LAVORO

Questo modello usa un parametro, Δ, per definire la **finestra dell’insieme di lavoro**. L’idea consiste nell’esaminare i più recenti Δ riferimenti alle pagine. L’insieme di pagine nei più recenti Δ riferimenti è l’**insieme di lavoro**.



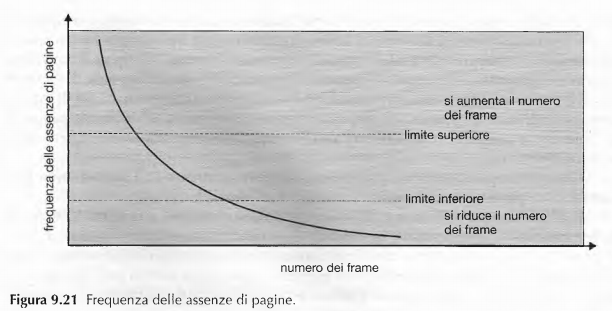
Considerando l’esempio nella figura in alto, data la successione di riferimenti alla memoria, se Δ = 10 riferimenti alla memoria, l’**insieme di lavoro** nell’istante t1 è {1, 2, 5, 6, 7}. All’istante t2 l’**insieme di lavoro** è {3, 4}.

La caratteristica più importante dell’insieme di lavoro è la sua dimensione. Calcolando la dimensione dell’insieme di lavoro, **WSS**, per ciascun processo **pi** del sistema, si può determinare la richiesta totale di frame, cioè **D**:

Ogni processo usa attivamente le pagine del proprio insieme di lavoro, Quindi il processo **i** necessita di **WSSi** frame. Se la richiesta totale è maggiore del numero totale di frame liberi (D > m), la paginazione **degenera**, poiché alcuni processi non dispongono di un numero sufficiente di frame.

FREQUENZA DELLE ASSENZE DI PAGINE

La strategia sulla **frequenza delle assenze di pagine** (*page fault frequency*, PFF) è basata sulla 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 ha bisogno di più frame. Se, invece, è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la frequenza effettiva delle assenze di pagine. Se oltrepassa il limite superiore, occorre allocare a quel processo un altro frame; se la sequenza scende sotto il limite inferiore, si sottrae un frame al processo.



CAPITOLO 11

STRUTTURA DEL FILE SYSTEM: Il **file system** è l’insieme delle strutture dati e dei metodi che ci permettono la registrazione e l’accesso a dati e programmi presenti in un sistema di calcolo. Per fornire un efficiente e conveniente accesso al disco, il sistema operativo fa uso di uno o più **file system** che consentono di memorizzare, individuare e recuperare facilmente i dati. Lo stesso file system è generalmente composto da molti livelli distinti.

Il livello più basso, il ***controllo dell’I/O***, costituito dai **driver dei dispositivi** e dai gestori dei segnali d’interruzione, si occupa del trasferimento delle informazioni tra memoria centrale e memoria secondaria.

Il **file system di base** deve soltanto inviare dei generici comandi all’appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco. Ogni blocco fisico si identifica col suo indirizzo numerico nel disco.

Il **modulo di organizzazione dei file** è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Conoscendo il tipo di allocazione dei file usato e la locazione dei file, può tradurre gli indirizzi dei blocchi logici, che il file system di base deve trasferire, negli indirizzi dei blocchi fisici.

Infine, i **file system logico** gestisce i metadati, si tratta di tutte le strutture del file system, eccetto gli effettivi dati. Il file system logico gestisce la struttura della directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita, dato un nome simbolico di file. Mantiene le strutture di file tramite i **blocchi di controllo dei file**, contenenti informazioni sui file, come la proprietà, i permessi e la posizione del contenuto.

REALIZZAZIONE DEL FILE SYSTEM: Per realizzare un file system si usano parecchie strutture dati, sia nei dischi sia in memoria. Fra le strutture presenti nei dischi ci sono le seguenti.

* **Il blocco di controllo dell’avviamento**. Contenente le informazioni necessarie al sistema per l’avviamento di un sistema operativo da quel volume.
* **I blocchi di controllo dei volumi**. Ciascuno di loro contiene dettagli riguardanti il relativo volume (o partizione).
* **Le strutture delle directory** usate per organizzare i file.
* **I blocchi di controllo di file**. Contenenti molti dettagli dei file, compresi i permessi d’accesso ai relativi file, i proprietari, le dimensioni e le locazioni dei blocchi di dati.

Le strutture che vi possono essere incluse sono di diverso tipo:

* La tabella di montaggio interna alla memoria che contiene informazioni relative a ciascun volume montato;
* La struttura della directory, tenuta in memoria, contenente le informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente;
* La tabella generale dei file aperti, contenente una copia del blocco di controllo del file per ciascun file aperto, insieme con altre informazioni;
* La tabella dei file aperti per ciascun processo, contenente un puntatore all’appropriato elemento della tabella generale dei file aperti;
* I buffer conservano blocchi del file system durante la loro lettura o scrittura sul disco.

Una volta creato un file, per essere usato per operazioni di I/O deve essere **aperto**. La chiamata di sistema open() passa un nome di file al file system. Una volta aperto il file, se ne ricerca il nome all’interno della directory. Successivamente, si crea un elemento nella tabella dei file aperti del processo con un puntatore alla tabella generale e con alcuni altri campi. Il nome dato all’elemento della tabella è **descrittore di file** (*file descriptor*) in UNIX, e **handle del file** in Windows. Quando un processo chiude il file, si cancella il relativo elemento nella tabella dei file aperti del processo e si decrementa il contatore associato al file nella tabella generale. Se tutti i processi che avevano aperto il file lo hanno chiuso, si riscrive l’informazione aggiornata sul file nella struttura delle directory nei dischi e si cancella il relativo elemento nella tabella generale dei file aperti.

FILE SYSTEM VIRTUALI: Un metodo ovvio ma non ottimale per realizzare più tipi di file system è scrivere procedure di gestione di file e directory separate per ciascun tipo di file system. In questo modo la realizzazione del file system si articola in tre strati principali:

* Il primo strato è l’interfaccia del file system, basata sulle chiamate di sistema open(), read(), write() e close() e sui descrittori dei file;
* Il secondo strato si chiama strato del **file system virtuale** e svolge due funzioni importanti:
  1. Separa le operazioni generiche del file system dalla loro realizzazione definendo un’interfaccia uniforme. Nello stesso calcolatore possono coesistere più interfacce del file system virtuale, che permettono un accesso trasparente a diversi tipi di file system montati localmente.
  2. Permette la rappresentazione univoca di un file su tutta la rete. Il file system virtuale è basato su una struttura di rappresentazione dei file detta **vnode** che contiene un indicatore numerico unico per tutta la rete per ciascun file.

REALIZZAZIONE DELLE DIRECTORY: La selezione degli algoritmi di allocazione e degli algoritmi di gestione delle directory ha un grande effetto sull’efficienza, le prestazioni e l’affidabilità del file system.

**LISTA LINEARE**: Il più semplice metodo di realizzazione di una directory è basato sull’uso di una lista lineare contenente i nomi dei file con puntatori ai blocchi di dati. Questo metodo è di facile programmazione, ma la sua esecuzione è onerosa in termini di tempo. Il vero svantaggio dato da una lista lineare di elementi di directory è dato dalla ricerca lineare di un file.

**TABELLA HASH**: Un’altra struttura dati che si usa per realizzare le directory è la **tabella hash**. La tabella hash riceve un valore calcolato usando come operando il nome del file e riporta un puntatore al nome del file nella lista lineare. L’inserimento e la cancellazione sono abbastanza semplici, anche se occorre prendere provvedimenti per evitare **collisioni**, cioè situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione.

METODI DI ALLOCAZIONE: Esistono tre metodi principali per l’allocazione dello spazio di un disco; può essere infatti contigua, concatenata o indicizzata.

**ALLOCAZIONE CONTIGUA**: Per usare il metodo di **allocazione contigua**, ogni file deve occupare un insieme di blocchi contigui del disco. L’allocazione contigua dello spazio è definita dall’indirizzo del primo blocco e dalla lunghezza. Accedere a un file il cui spazio è assegnato in modo contiguo è facile. L’allocazione contigua però presenta alcuni problemi, uno dei quali riguarda l’individuazione dello spazio per un nuovo file che ricorda il problema generale dell’**allocazione dinamica della memoria**. I più comuni criteri di scelta sono: best-fit, first-fit, worst-fit. Questi algoritmi, però, soffrono della **frammentazione esterna**: assegnando e liberando lo spazio per i file, lo spazio libero dei dischi viene frammentato in tanti piccoli pezzi. Se poi l’ultimo blocco assegnatogli non è completamente pieno, si avrà anche il fenomeno della **frammentazione interna**.

Per ridurre al minimo questi inconvenienti, alcuni sistemi operativi fanno uso di uno schema di allocazione contigua modificato: inizialmente si assegna una porzione di spazio continuo, e se questa non è abbastanza grande si aggiunge un’altra porzione di spazio, un’**estensione**.

**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 e ogni blocco contiene un puntatore al blocco successivo.

Con l’allocazione concatenata non esiste frammentazione esterna e per soddisfare una richiesta si può usare qualsiasi blocco libero della lista. Inoltre non è necessario dichiarare la dimensione di un file al momento della sua creazione e infatti può continuare a crescere finché sono disponibili blocchi liberi.

Ci sono però comunque alcuni svantaggi. Il problema principale riguarda il fatto che può essere usata in modo efficiente solo per i file ad accesso sequenziale. Per trovare l’i-esimo blocco di un file occorre partire dall’inizio del file e seguire i puntatori finché non si raggiunge l’i-esimo blocco.

La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi contigui in **cluster**, e nell’allocare i cluster anziché i blocchi. Questo metodo permette che la corrispondenza tra blocchi logici e fisici rimanga semplice, ma migliora la produttività del disco.

Un altro problema riguarda l’affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi per tutto il disco, se un puntatore andasse perduto o danneggiato è la fine. Una soluzione parziale a tale problema consiste nell’usare liste doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco.

Una variante importante del metodo di allocazione concatenata consiste nell’uso della **tabella di allocazione dei file** (FAT). La FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco. L’elemento di directory contiene il numero del primo blocco del file. L’elemento della tabella indicizzato dal quel numero di blocco contiene a sua volta il numero del blocco successivo del file. Questa catena continua fino all’ultimo blocco, che ha come elemento della tabella un valore speciale di fine del file. I blocchi inutilizzati sono indiati nella tabella da un valore 0. L’allocazione di un nuovo blocco a un file implica semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l’indirizzo del nuovo blocco.

**ALLOCAZIONE INDICIZZATA**: In mancanza di una FAT, l’allocazione concatenata non è in grado di sostenere un efficiente accesso diretto. L’**allocazione indicizzata** risolve questo problema, raggruppando tutti i puntatori in una sola locazione: il **blocco indice**. Ogni file ha il proprio blocco indice: si tratta di un array d’indirizzi di blocchi del disco. L’i-esimo elemento del blocco indice punta all’i-esimo blocco del file. Per individuare e leggere l’i-esimo blocco occorre usare il puntatore che si trova nell’i-esimo elemento del blocco indice, per poi localizzare e leggere il blocco desiderato.

I possibili meccanismi per memorizzare il blocco indice sono:

* **Schema concatenato**. Un blocco indice è formato normalmente di un solo blocco di disco; perciò, ciascun blocco indice può essere letto e scritto esattamente con un’operazione. Per permettere la presenza di lunghi file è possibile collegare tra loro parecchi blocchi indice.
* **Indice a più livelli**. Una variante della rappresentazione concatenata consiste nell’impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file.
* **Schema combinato**. Un’altra possibilità, consiste nel tenere i primi 15 puntatori del blocco indice nell’*i-node* del file. I primi 12 di questi puntano a **blocchi diretti**, cioè contengono direttamente gli indirizzi di blocchi contenenti dati del file. Gli altri 3 puntatori puntano a **blocchi indiretti**. Il primo puntatore di blocco indiretto è l’indirizzo di un **blocco singolo**. Quindi c’è un puntatore di **blocco indiretto doppio** contenete l’indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati. L’ultimo puntatore contiene l’indirizzo di un **blocco indiretto triplo**.

GESTIONE DELLO SPAZIO LIBERO: Per tener traccia dello spazio libero in un disco, il sistema conserva una **lista dello spazio libero**; vi sono registrati tutti gli spazi liberi, cioè non allocati ad alcun file o directory.

**VETTORE DI BIT**: Spesso la lista dello spazio libero si realizza come una **mappa di bit**, o **vettore di bit**. Ogni blocco è rappresentato da un bit: se il blocco è libero, il bit è 1, se il blocco è assegnato il bit è 0. I vantaggi principali che derivano da questo metodo sono la sua relativa semplicità ed efficienza nel trovare il primo blocco libero o **n** blocchi liberi consecutivi nel disco. I vettori di bit sono efficienti solo se tutto il vettore è mantenuto in memoria centrale.

**LISTA CONCATENATA**: Un altro metodo di gestione degli spazi liberi consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria. Questo primo blocco contiene un puntatore al successivo blocco libero e così via. Questo schema non è efficiente; per attraversare la lista è infatti necessario leggere ogni blocco, con un notevole tempo di I/O.

**RAGGRUPPAMENTO**: Una possibile modifica del metodo della lista dello spazio libero prevede la memorizzazione degli indirizzi di **n** blocchi liberi nel primo di questi. I primi **n-1** di questi blocchi sono effettivamente liberi; l’ultimo blocco contiene gli indirizzi di altri **n** blocchi liberi, e così via.

**CONTEGGIO**: Anziché tenere una lista di **n** indirizzi liberi, è sufficiente tenere l’indirizzo del primo blocco libero e il numero **n** di blocchi liberi contigui che seguono il primo blocco. Ogni elemento della lista dello spazio libero è formato da un indirizzo del disco e un contatore.

EFFICIENZA E PRESTAZIONI

**EFFICIENZA**: I dischi tendono di solito a essere il principale collo di bottiglia per le prestazioni di un sistema, essendo i più lenti tra i componenti più rilevanti di un calcolatore. L’uso efficiente di un disco dipende fortemente dagli algoritmi usati per l’allocazione del disco e la gestione delle directory. Si devono tenere in considerazione anche il tipo di dati normalmente contenuti in un elemento di una directory. Di solito si memorizza la **data dell’ultima scrittura** per fornire informazioni all’utente e per determinare se per il file occorre la creazione o l’aggiornamento di una copia di riserva. Alcuni sistemi mantengono anche la **data dell’ultimo accesso** per consentire all’utente di risalire all’ultima volta che un file è stato letto.

**PRESTAZIONI:** Dopo aver scelto gli algoritmi fondamentali del file system le prestazioni possono essere migliorate in diversi modi. Alcuni sistemi riservano una sezione separata dalla memoria centrale come **cache del disco**, dove tenere i blocchi in previsione di un loro riutilizzo entro breve tempo. Altri sistemi impiegano una **cache delle pagine** per i file; si tratta di una soluzione che impiega tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi di file system. Questo metodo è noto come **memoria virtuale unificata**.

Alcune versioni di UNIX e Linux prevedono la cosiddetta **buffer cache unificata**. In questo caso, le chiamate di sistema read() e write() passano attraverso la buffer cache. La chiamata con associazione alla memoria richiede l’uso di due cache, la cache delle pagine e la buffer cache. Questa situazione è nota come **double caching** proprio perché i dati del file system richiedono un doppio passaggio di cache. Con una buffer cache unificata, sia l’associazione alla memoria sia le chiamate di sistema read() e write() usano la stessa cache delle pagine., con il vantaggio di evitare il double caching e di permettere al sistema di memoria virtuale di gestire i dati del file system.

Indipendentemente dalla gestione della cache per blocchi di disco oppure per pagine, l’algoritmo LRU è in generale ragionevole per la sostituzione dei blocchi o delle pagine. A causa dell’alta frequenza delle operazioni di I/O, quando la memoria libera diventa troppo esigua, il modulo di scansione delle pagine sottrae pagine ai processi anziché alla cache delle pagine.

Ci sono altri aspetti che possono influenza le prestazioni di I/O, come quelli che riguardano la necessità di scritture sincrone o asincrone. Le scritture **sincrone** avvengono nell’ordine in cui le riceve il sottosistema per la gestione del disco e non subiscono la memorizzazione transitoria. Nella maggior parte dei casi si usano le scritture **asincrone**, in cui si memorizzano i dati nella cache e si restituisce immediatamente il controllo alla procedura chiamante.

Gli accessi sequenziali si potrebbero invece ottimizzare con tecniche note come rilascio indietro e lettura anticipata. Il **rilascio indietro** rimuove una pagina dalla memoria di transito non appena si verifica una richiesta della pagina successiva. Con la **lettura anticipata** si leggono e si mettono nella cache la pagina richiesta e parecchie pagine successive. Il recupero di questi dati dal disco con un unico trasferimento e la memorizzazione nella cache consentono di risparmiare una quantità di tempo considerevole.

**RIPRISTINO**: Poiché i file e le directory sono mantenuti sia in memoria centrale sia nei dischi, è necessario aver cura di assicurare che il verificarsi di un malfunzionamento nel sistema non comporti la perdita di dati e la loro incoerenza.

**VERIFICA DELLA COERENZA**: Il **verificatore della coerenza** confronta i dati delle directory con quelli contenuti nei blocchi dei dischi, tentando di correggere ogni incoerenza.

**FILE SYSTEM CON REGISTRAZIONE DELLE MODIFICHE (ANNOTATI)**: Alcuni algoritmi sono stati applicati con successo al problema della verifica della coerenza, realizzando i **file system orientati alle transazioni e basati sulla registrazione delle modifiche**, noti anche come **file system annotati**. Fondamentalmente, tutte le modifiche dei metadati si annotano in modo sequenziale in un file di registrazione, detto ***log***. Ogni insieme di operazioni che eseguo uno specifico compito si chiama **transazione**. Quando un’intera transazione è stata completata, se ne rimuovono le annotazioni dal log, che è in realtà un buffer circolare.

**COPIE DI RISERVA E RECUPERO DEI DATI**: Poiché si possono verificare malfunzionamenti e perdite di dati anche nei dischi magnetici, è necessario preoccuparsene e provvedere affinché i dati non vadano persi definitivamente. A questo scopo si possono usare programmi di sistema che consentano di fare delle **copie di riserva** (backup). Il ripristino della situazione antecedente la perdita di un singolo file, richiederà **il recupero** (restore) dei dati dalle copie di riserva. Una tipica sequenza di gestione delle copie di riserva è:

* **Giorno 1**. Copiatura nel supporto di backup delle copie di riserve di tutti i file contenuti nel disco; detta **copiatura completa**.
* **Giorno 2**. Copiatura su un altro supporto di tutti i file modificati dal Giorno 1; si tratta di una **copiatura incrementale**.
* **Giorno 3**.

.

.

.

* **Giorno n**. Copiatura su un altro supporto di tutti i file modificati dal Giorno n – 1. Ritorno al Giorno 1.

CAPITOLO 12

STRUTTURA DEI DISPOSITIVI DI MEMORIZZAZIONE

**DISCHI MAGNETICI**: I **dischi magnetici** sono il mezzo fondamentale di memoria secondaria dei moderni sistemi di calcolo. Concettualmente sono relativamente semplici: i **piatti** dei dischi hanno una forma piana e rotonda come quella CD. Le testine di lettura e scrittura sono sospese su ciascuna superficie d’ogni piatto e sono attaccate al **braccio del disco** che lo muove in blocco. La superficie di un piatto è divisa logicamente in **tracce** circolari a loro volta suddivise in **settori**; l’insieme delle tracce corrispondenti a una posizione del braccio costituisce un **cilindro**.

La testina può danneggiare la superficie magnetica. Tale incidente, detto **crollo della testina**, di solito non può essere riparato. Un disco può essere **rimovibile**: ciò permette che diversi dischi siano montati secondo la necessità.

I moderni dischi sono considerati come grandi array monodimensionali di **blocchi logici**, dove un blocco logico è la minima unità di trasferimento. La dimensione di un blocco logico è di solito 512 byte, sebbene alcuni dischi si possano **formattare a basso livello** allo scopo di ottenere una diversa dimensione dei blocchi logici, ad esempio 1024 byte.

SCHEDULING DEL DISCO: Il tempo d’accesso si può scindere in due componenti principali: il **tempo di ricerca** e la **latenza di rotazione**.

L’**ampiezza di banda** è il numero totale di byte trasferiti diviso il tempo totale intercorso fra la prima richiesta e il completamento dell’ultimo trasferimento.

Ogni volta che devono compiere operazioni di I/O con un’unità a disco, un processo impartisce al sistema operativo una chiamata di sistema. La richiesta contiene diverse informazioni:

* Se l’operazione sia di immissione o di emissione di dati;
* L’indirizzo nel disco rispetto al quale eseguire il trasferimento;
* L’indirizzo di memoria rispetto al quale eseguire il trasferimento;
* Il numero di byte da trasferire.

**SCHEDULING IN ORDINE D’ARRIVO – FCFS**: La forma più semplice di scheduling è, naturalmente, l’algoritmo di servizio secondo l’ordine d’arrivo. Si tratta di un algoritmo equo, ma che in generale non garantisce la massima velocità di servizio. Le deficienze di quest’algoritmo sono palesate dai frequenti giganteschi salti per passare da un blocco all’altro.

**SCHEDULING PER BREVITÀ – SSTF**: Sembra ragionevole servire tutte le richieste vicine alla posizione corrente della testina prima di spostarla in un’area lontana per soddisfarne altre: questa considerazione è alla base dell’**algoritmo di servizio secondo il più breve tempo di ricerca**, che sceglie la richiesta che dà il minimo tempo di ricerca rispetto alla posizione corrente della testina. Lo scheduling SSTF è essenzialmente una forma di scheduling per brevità, e al pari di questo, può condurre a situazioni d’attesa indefinita (starvation) di alcune richieste.

**SCHEDULING PER SCANSIONE – SCAN**: Secondo l’algoritmo SCAN il braccio dell’unità a disco parte da un estremo del disco e si sposta nella sola direzione possibile, servendo le richieste mentre attraversa i cilindri, finché non giunga all’altro estremo del disco: a questo punto, il braccio inverte la marcia, e la procedura continua. Le testine attraversano continuamente il disco nelle due direzioni. L’algoritmo SCAN è a volte chiamato **algoritmo dell’ascensore**, perché il braccio dell’unità a disco si comporta proprio come un ascensore che serva per prima tutte le richieste in salita e poi tutte quelle in discesa.

**SCHEDULING PER SCANSIONE CIRCOLARE – C-SCAN**: L’algoritmo SCAN **circolare** è una variante dello scheduling SCAN. Anche l’algoritmo C-SCAN sposta la testina da un estremo all’altro del disco, servendo le richieste lungo il percorso; tuttavia, quando la testina giunge all’altro estremo del disco, ritorna immediatamente all’inizio del disco stesso, senza servire richieste durante il viaggio di ritorno.

**SCHEDULING LOOK**: Queste versioni dello SCAN e del C-SCAN sono dette **LOOK** e **C-LOOK**, perché “guardano” se ci sono altre richieste da soddisfare lungo la direzione attuale prima di continuare a spostare la testina in quella direzione. (NON SCORRE TUTTO IL DISCO)

**SCELTA DI UN ALGORITMO DI SCHEDULING**: Un algoritmo molto comune e naturalmente attraente è l’SSTF poiché aumenta le prestazioni rispetto all’FCFS; lo SCAN e il C-SCAN danno migliori prestazioni in sistemi che sfruttano molto le unità a disco, perché conducono con minor probabilità a situazioni d’attesa indefinita. Per qualunque algoritmo di scheduling, le prestazioni dipendono comunque in larga misura dal numero e dal tipo di richieste.

GESTIONE DELL’UNITÀ A DISCO

**FORMATTAZIONE DEL DISCO**: Un disco magnetico nuovo è **tabula rasa**: un insieme di uno o più piatti sovrapposti ricoperti di materiale magnetico; prima che possa memorizzare dati, deve essere diviso in settori che possano essere letti o scritti dal controllore. Questo processo si chiama formattazione di basso livello, o **formattazione fisica**. La **formattazione di basso livello** riempie il disco con una speciale struttura dati per ogni settore, tipicamente consistente di un’intestazione, un’area per i dati, e una coda. L’intestazione e la coda contengono informazioni usate dal controllore del disco, ad esempio il numero del settore e un **codice per la correzione degli errori**.

Per usare un disco come contenitore di informazioni, il sistema operativo deve ancora registrare le proprie strutture dati nel disco, cosa che fa in due passi. Il primo consiste nel suddividere il disco in uno o più gruppi di cilindri, detti **partizioni**. Il passo successivo alla suddivisione in partizioni è la **formattazione logica**, cioè la creazione di un file system: il sistema operativo registra nel disco le strutture dati iniziali relative al file system.

**BLOCCO D’AVVIAMENTO**: Affinché un calcolatore possa entrare in funzione, ad esempio quando viene acceso o riavviato, è necessario che esegua un programma iniziale, di solito, questo programma d’avviamento iniziale è piuttosto semplice. Per la maggior parte dei calcolatori, il programma d’avviamento è memorizzato in una **memoria a sola lettura** (ROM), il che è conveniente, perché la ROM non richiede inizializzazione, e ha un indirizzo iniziale fisso dal quale la CPU può cominciare l’esecuzione ogniqualvolta si accende o si riavvia la macchina. Il problema, però, è che cambiare il programma d’avviamento richiede in questo caso la sostituzione dei circuiti integrati ROM. A causa di questo inconveniente, molti sistemi memorizzano nella ROM un piccolo caricatore d’avviamento (*bootstrap loader*) il cui solo compito è quello di caricare da un disco il programma d’avviamento completo. Quest’ultimo si può facilmente modificare: se ne scrive semplicemente una nuova versione nel disco. Il programma d’avviamento completo è registrato in una partizione del disco denominata blocchi d’avviamento, posta in una locazione fissata del disco; un disco contenente una tale partizione si chiama **disco d’avviamento** o **disco di sistema**.

**BLOCCHI DIFETTOSI:** A volte si può verificare un guasto irreparabile, e l’unità a disco deve essere sostituita. Più di frequente, uno o più settori divengono malfunzionanti; in effetti, la maggior parte dei dischi messi in commercio contiene già **blocchi difettosi**. Essi sono trattati in diversi modi, nel caso dei dischi semplici come quelli gestiti da un controllore IDE, i blocchi difettosi sono gestiti “manualmente”. Unità a disco, invece, più complesse come i dischi SCSI hanno strategie di recupero dei blocchi difettosi più complesse. Il controllore mantiene una lista dei blocchi malfunzionanti dell’unità a disco che è inizializzata durante la formattazione fisica eseguita dal produttore, ed è aggiornata per tutto il periodo in cui l’unità a disco è operativa. La formattazione fisica mette anche da parte dei settori di riserva non visibili al sistema operativo: si può istruire il controllore affinché sostituisca da un punto di vista logico un settore difettoso con uno dei settori di riserve inutilizzati. Questa strategia è nota come **accantonamento di settori**. Un’alternativa è data da quei controllori capaci di sostituire i settori difettosi tramite la tecnica della **traslazione dei settori**.

GESTIONE DELL’AREA D’AVVICENDAMENTO

La **gestione dell’area d’avvicendamento** è un altro compito di basso livello del sistema operativo. La memoria usa lo spazio dei dischi come estensione della memoria centrale: poiché l’accesso alle unità a disco è molto più lento, l’uso di un’area d’avvicendamento riduce notevolmente le prestazioni del sistema.

**COLLOCAZIONE DELL’AREA D’AVVICENDAMENTO**: Le possibili collocazioni per un’area d’avvicendamento sono due: all’interno del normale file system, o in una partizione del disco a sé stante. Se l’area d’avvicendamento è semplicemente un grande file all’interno del file system, si possono usare le ordinarie funzioni del file system per crearla, assegnargli un nome, e allocare spazio per essa.

In alternativa, l’area d’avvicendamento si può creare in un’apposita partizione del disco non formattata: in essa non è presente alcuna struttura relativa al file system e alle directory, ma si usa uno speciale gestore dell’area d’avvicendamento per allocare e rimuovere i blocchi.

STRUTTURE RAID

Ci sono varie tecniche per l’organizzazione dei dischi, note col nome comune di **batterie ridondanti di dischi** (RAID), che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

**MIGLIORAMENTO DELL’AFFIDABILITÀ**: Consideriamo in primo luogo l’affidabilità. La possibilità che uno dei dischi in un insieme di **n** dischi si guasti è molto più alta della possibilità che uno specifico disco isolato presenti un guasto. La soluzione a questo problema sta nell’introdurre una certa **ridondanza**, cioè nel memorizzare informazioni che non sono normalmente necessarie, ma che si possono usare nel caso di un guasto a un disco per ricostruire le informazioni perse. Il metodo più semplice è quello della **copiatura speculare**; ogni disco logico consiste di due dischi fisici e ogni scrittura si effettua in entrambi i dischi. Se uno dei dischi si guasta, i dati si possono leggere dall’altro. I dati si perdono solo se il secondo disco si guasta prima della sostituzione del disco già guasto. Il tempo medio di guasto di un disco con copiatura speculare, dipende da due fattori: il tempo medio di guasto di un singolo disco e il **tempo medio di riparazione**, cioè il tempo richiesto (in media) per sostituire un disco guasto e ripristinarvi i dati.

**MIGLIORAMENO DELLE PRESTAZIONI TRAMITE IL PARALLELISMO**: L’accesso in parallelo a più dischi può provare vari vantaggi. Con la copiatura speculare, se si può accedere in parallelo a più dischi, la frequenza con la quale si possono gestire le richieste di lettura raddoppia. Attraverso l’uso di più dischi è possibile anche migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dischi. Nella sua forma più semplice, questa distribuzione, chiamata **sezionamento dei dati**, consiste nel distribuire i bit di ciascun byte su più dischi; in questo caso si parla di **sezionamento a livello dei bit**. Inoltre, il sezionamento non si deve realizzare necessariamente a livello dei bit di un byte: nel **sezionamento a livello dei blocchi**, ad esempio, i blocchi di un file si distribuiscono su più dischi.

LIVELLI RAID: Sono stati proposti numerosi schemi per fornire ridondanza usando l’idea del sezionamento combinata con i bit di parità. Questi schemi realizzano diversi compromessi tra costi e prestazioni e sono stati classificati in livelli chiamati **livelli RAID:**

* **RAID di livello 0**. Il livello 0 si riferisce a batterie di dischi con sezionamento a livello dei blocchi, ma senza ridondanza;
* **RAID di livello 1**. Il livello 1 si riferisce alla tecnica con la copiatura speculare.
* **RAID di livello 2**. Il livello 2 è anche noto come **organizzazione con codici per la correzione degli errori** (ECC) In un sistema di questo tipo, ogni byte di memoria ha associato un bit di parità che indica se i bit con valore 1 nel byte sono in numero pari (parità = 0) oppure dispari (parità = 1). Se si altera uno dei bit nel byte, la parità del byte cambia e quindi non concorda più con la parità memorizzata.
* **RAID di livello 3**. Con il livello 3, o **organizzazione con bit di parità intercalati** i controllori dei dischi possono rilevare se un settore è stato letto correttamente, così che un unico bit di parità si può usare sia per individuare gli errori sia per correggerli. Se uno dei settori è danneggiato, si conosce esattamente di quale settore si tratta e, per ogni bit nel settore, è possibile determinare se debba avere valore 1 o 0 calcolando la parità dei bit corrispondenti dai settori negli altri dischi. Se la parità dei rimanenti bit è uguale a quella memorizzata, il bit mancante è 0, altrimenti è 1. La batteria ha anche una cache RAM **non volatile** per memorizzare i blocchi mentre viene calcolata la parità e per memorizzare transitoriamente le scritture dal controllore ai dischi.
* **RAID di livello 4**. Nel livello 4, o **organizzazione con blocchi di parità intercalati**, s’impiega il sezionamento a livello dei blocchi e inoltre si tiene un blocco di parità in un disco separato per i blocchi corrispondenti presenti in **n** dischi diversi da questo. Se uno dei dischi si guasta, il blocco di parità si può usare insieme ai blocchi corrispondenti degli altri dischi per ripristinare i blocchi nel disco guasto.
* **RAID di livello 5**. Il livello 5, o **organizzazione con blocchi intercalati a parità distribuita**, i dati e le informazioni di parità sono distribuite tra gli **n + 1** dischi. Per ogni blocco, uno dei dischi memorizza la parità e gli altri i dati. Un blocco di parità non può contenere informazioni di parità per blocchi che non risiedono nello stesso disco, poiché un guasto al disco provocherebbe sia la perdita dei dati sia la perdita delle informazioni di parità e quindi i dati non sarebbero ripristinabili.
* **RAID di livello 6**. Il livello 6, detto anche **schema di ridondanza P + Q**, memorizza ulteriori informazioni ridondanti per poter gestire guasti contemporanei di più dischi. Invece di usare la parità, s’impegnano codici per la correzione degli errori, come i **codici di Reed – Solomon**.
* **RAID di livello 0 + 1**. Il livello 0 + 1 consiste in una combinazione dei livelli RAID 0 e 1. Il livello 0 fornisce le prestazioni, mentre il livello 1 l’affidabilità. Se si guasta un singolo disco, l’intera sezione dei dati diventa inaccessibile, lasciando disponibile solo l’altra sezione.
* **RAID di livello 1 + 0**. Il livello 1 + 0, si fa prima la copiatura speculare dei dischi a coppie, e poi il sezionamento su queste coppie. Con un guasto, il singolo disco diventa inaccessibile, ma il suo duplicato è ancora disponibile, come tutti gli altri dischi.

STRUTTURE PER LA MEMORIZZAZIONE TERZIARIA

**DISHI RIMOVIBILI**: I dischi rimovibili sono un tipo di memoria terziaria. I **dischi magnetico-ottici** sono un altro tipo di dischi rimovibili. La testina di un disco magnetico-ottico è sospesa a una distanza dalla superficie del disco molto maggiore rispetto alla testina di un disco magnetico, e il materiale magnetico è protetto da uno spesso strato di plastica o di vetro; di conseguenza, il disco è molto più resistente a eventuali collisioni della testina. Un’altra categoria di dischi rimovibili è quella dei **dischi ottici**, i quali non sfruttano per niente il magnetismo, ma usano materiali speciali che la luce laser può alterare in modo da creare punti relativamente chiari o scuri. Il tipo di dischi descritti fin qui si possono riutilizzare: per questo sono detti **dischi a lettura e scrittura**. Per contro, i **dischi monoscrivibili** o costituiscono una categoria distinta.

**NASTRI**: In linea generale un nastro può contenere più dati di un disco ottico o magnetico rimovibile. Le unità a nastro e quelle a disco hanno velocità di trasferimento simili ma, poiché richiedono operazioni di avanzamento rapido o di riavvolgimento che possono durare decine di secondi o addirittura interi minuti, l’accesso diretto è molto più lento per un nastro che per un disco. I nastri si usano per contenere copie di riserva dei dati presenti nei dischi.

COMPITI DEL SISTEMA OPERATIVO: Due tra gli obiettivi primari di un S.O. sono la gestione dei dispositivi fisici e la presentazione di una macchina virtuale alle applicazioni. Il sistema operativo realizza due astrazioni concernenti i dischi: una è il dispositivo a basso livello, un semplice array di blocchi di dati; l’altra è il file system.

**INTERFACCIA PER LE APPLICAZIONI**: La maggior parte dei sistemi operativi gestisce i dischi rimovibili quasi come i dischi fissi. Quando s’inserisce un nuovo disco nella relativa unità, esso deve essere formattato, quindi si crea sul disco rimovibile un file system vuoto che si usa proprio come il file system di un’ordinaria unità a disco.

La gestione dei nastri, invece, è spesso differente; il sistema operativo di solito presenta un nastro come un supporto di memorizzazione a basso livello. Quando un’unità a nasco è presentata come dispositivo a basso livello, il sistema operativo non fornisce servizi del file system: è l’applicazione che deve decidere come usare l’array di blocchi.

**NOMI DEI FILE**: Nel caso di un disco fisso, nei PC i nomi dei file consistono di una lettera rappresentante un’unità seguita da un nome di percorso. Le cose si complicano nel caso s’intenda scrivere dati su un supporto rimovibile in un certo calcolatore e poi riutilizzare lo stesso mezzo in un altro calcolatore: se entrambi i calcolatori sono dello stesso tipo e hanno lo stesso tipo di unità, l’unico problema è quello di conoscere i contenuti e l’organizzazione dei dati; ma se i calcolatori o le unità sono di diverso tipo possono sorgere molte altre difficoltà.

**GESTIONE GERARCHICA DELLA MEMORIA**: Un **juke-box** **automatizzato** permette a un calcolatore di cambiare un nastro o un disco rimovibile senza l’intervento dell’utente. La tecnica più comune per estendere la gerarchia di memorizzazione fino alla memoria terziaria consiste nell’ampliare i file system. I file piccoli e frequentemente usati rimangono nei dischi magnetici, mentre i file vecchi, ingombranti e raramente necessari si archiviano nel juke-box.

Al giorno d’oggi, la **gestione gerarchica della memoria secondaria** si applica solitamente a sistemi con grandi quantità di dati usati di rado, o periodicamente. Le ricerche correnti in questo campo tentano di estendere quest’ultima con una piena **gestione del ciclo di vita delle informazioni**.

PRESTAZIONI

**VELOCITÀ**: La velocità della memoria terziaria è definita da due fattori: ampiezza di banda e latenza. La prima si misura in byte; in particolare, l’**ampiezza di banda sostenuta** è la velocità media di trasferimento nel caso di una rilevante quantità di dati, cioè il numero di byte diviso il tempo di trasferimento; l’**ampiezza di banda effettiva** è invece il numero di byte trasferiti diviso il tempo di I/O totale.

Il secondo fattore è la **latenza di accesso**. Rispetto a questo parametro i dischi sono molto più veloci dei nastri; un accesso al disco si compie semplicemente spostando il braccio verso il cilindro selezionato e aspettando che il settore interessato ruoti sotto la testina. Un accesso diretto a nastro richiede lo svolgimento o il riavvolgimento della bobina finché il blocco richiesto raggiunge la testina.

**AFFIDABILITÀ**: Si può dire che le unità a disco fisso sono più affidabili delle unità a nastri o a dischi rimovibili, e che i dischi ottici sono probabilmente più affidabili dei dischi e dei nastri magnetici.

**COSTI**: La memoria centrale è più costosa della memoria secondaria. Non si dispone più di una tecnologia di memoria terziaria che sia di più ordini di grandezza più economica dei dischi magnetici.

CAPITOLO 13

INTRUDUZIONE: Poiché i dispositivi di I/O sono così largamente diversi per funzioni e velocità, altrettanto diversi devono essere i metodi di controllo. Tali metodi costituiscono il **sottosistema di I/O** del nucleo; questo sottosistema separa il resto del nucleo dalla complessità di gestione dei dispositivi di I/O. I **driver dei dispositivi** offrono al sottosistema di I/O un’interfaccia uniforme per l’accesso ai dispositivi.

**ARCHITETTURE E DISPOSITIVI DI I/O**: I calcolatori fanno funzionare un gran numero di tipi di dispositivi. La maggior parte rientra nella categoria dei dispositivi di memorizzazione secondaria e terziaria (dischi, nastri), dispositivi di trasmissione (schede di rete, modem), interfacce uomo-macchina (schermi, tastiere, mouse).

Un dispositivo comunica con un sistema di calcolo inviando segnali attraverso un cavo o attraverso l’etere e comunica con il calcolatore tramite un punto di connessione (**porta**). Se uno o più dispositivi usano in comune un insieme di fili, la connessione è detta **bus**. Un **bus** è un insieme di fili e un protocollo rigorosamente definito che specifica l’insieme dei messaggi che si possono inviare attraverso i fili. Quando un dispositivo A ha un cavo che si connette a un dispositivo B e il dispositivo B ha un cavo che si connette a un dispositivo C che a sua volta è collegato a una porta di un calcolatore, si ottiene il cosiddetto **collegamento a margherita**, che di solito funziona come un bus. Una tipica struttura di bus di PC; si tratta di un **bus PCI** che connette il sottosistema CPU-memoria ai dispositivi veloci, e di un **bus di espansione** cui si connettono i dispositivi relativamente lenti. I dischi sono collegati a un bus SCSI inserito nel relativo controllore. Un **controllore** è un insieme di componenti elettronici che può far funzionare una porta, un bus o un dispositivo. Esistono controllori semplici, come un controllore di porta seriale. Ed altri, come il controllore SCSI, più complessi; infatti, il controllore del bus SCSI è spesso realizzato come una scheda circuitale separata, **adattatore**, che s’inserisce nel calcolatore. Esso contiene tipicamente un’unità d’elaborazione, microcodice e memoria privata. Alcuni dispositivi sono dotati di propri controllori incorporati. L’unità d’elaborazione dà comandi e fornisce dati al controllore per portare a termine trasferimenti di I/O tramite uno o più registri per dati e segnali di controllo. La comunicazione col controllore avviene attraverso la lettura e la scrittura di configurazioni di bit in questi registri. La comunicazione può avvenire in due modi:

* **Utilizzo di speciali istruzioni di I/O**. L’istruzione I/O attiva le linee di bus che selezionano il giusto dispositivo e trasferiscono bit nei registri del dispositivo stesso.
* **I/O associato alla memoria**. Ai registri del controllore viene assegnata una sequenza di indirizzi in memoria centrale, in modo che ogni operazione di lettura e scrittura a questi indirizzi comporta un trasferimento diretto di dati con i registri del dispositivo.

Una porta I/O permette la trasmissione di informazioni tra il sistema centrale e un dispositivo e consiste tipicamente di quattro registri:

1. Il registro **status** contiene alcuni bit che possono essere letti e indicano lo stato della porta;
2. Il registro **control** può essere scritto per attivare un comando o per cambiare il modo di funzionamento del dispositivo;
3. La CPU legge dal registro **data-in** per ricevere i dati;
4. La CPU scrive nel registro **data-out** per emettere i dati.

La tipica dimensione dei registri di dati varia tra 1 e 4 byte.

INTERROGAZIONE CICLICA: Il protocollo completo per l’interazione fra la CPU e un controllore può essere complesso, ma la fondamentale nozione di **negoziazione** è semplice. Si assuma l’uso di due bit per coordinare la relazione di tipo produttore – consumatore fra il controllore e la CPU. Il controllore specifica il suo stato per mezzo del bit **busy** del registro **status**; pone a 1 il bit **busy** quando è impegnato in un’operazione, e lo pone a 0 quando è pronto a eseguire il comando successivo. La CPU comunica le sue richieste tramite il bit **command-ready** nel registro **command**; pone questo bit a 1 quando il controllore deve eseguire il comando. In questo esempio, la CPU scrive in una porta coordinandosi con un controllore per mezzo della negoziazione come segue:

1. La CPU legge ripetutamente il bit **busy** fino a che esso non vale 0.
2. La CPU pone a 1 il bit **write** del registro dei comandi e scrive un byte nel registro **data-out**;
3. La CPU pone a 1 il bit **command-ready**;
4. Quando il controllore si accorge del fatto che il bit **command-ready** è posto a 1, pone a 1 il bit **busy**;
5. Il controllore legge il registro dei comandi e trova il comando **write**; legge il registro **data-out** per ottenere il byte da scrivere, e compie l’operazione di scrittura nel dispositivo;
6. Il controllore pone a 0 il bit **command-ready**, pone a 0 il bit **error** nel registro **status** per indicare che l’operazione di I/O ha avuto esito positivo, e pone a 0 il bit **busy** per indicare che l’operazione è terminata.

La sequenza si ripete per ogni byte. Durante l’esecuzione del passo 1, la CPU è in **attesa attiva** (busy-waiting) o in **interrogazione ciclica** (polling): itera la lettura del registro **status** fino a che il bit **busy** assuma il valore 0.

INTERRUZIONI: La CPU ha un contatto, detto **linea di richiesta dell’interruzione**, del quale la CPU controlla lo stato dopo l’esecuzione di ogni istruzione. Quando rileva il segnale di un controllore nella linea di richiesta dell’interruzione, la CPU memorizza lo stato dell’elaborazione corrente e salta alla **procedura di gestione delle interruzioni**, che si trova a un indirizzo di memoria fissato. Questa procedura determina le cause dell’interruzione, porta a termine l’elaborazione necessaria ed esegue un’istruzione **return from interrupt** per far sì che la CPU ritorni nello stato in cui si trovata prima della sua interruzione. Inoltre è necessario strutturare i segnali d’interruzione in diversi livelli, cosicché il sistema operativo possa distinguere tra interruzioni ad alta e a bassa priorità, e possa rispondere con l’appropriato grado di tempestività. In un calcolatore moderno queste tre caratteristiche sono fornite dalla CPU e dal **controllore delle interruzioni**. La maggior parte delle CPU ha due linee di richiesta delle interruzioni. Una è quella delle **interruzioni non mascherabili**, che è riservata a eventi quali gli errori di memoria irrecuperabili. La seconda linea è quella delle **interruzioni mascherabili**: che è usata dai controllori dei dispositivi per richiedere un servizio. Il meccanismo delle interruzioni accetta un **indirizzo**, un numero che seleziona da un insieme ristretto una specifica procedura di gestione delle interruzioni. Nella maggior parte delle architetture questo indirizzo è uno scostamento relativo a una tabella detta **vettore delle interruzioni**, che contiene gli indirizzi di memoria degli specifici gestori delle interruzioni. Il meccanismo delle interruzioni realizza anche un sistema di **livelli di priorità delle interruzioni**. Esso permette alla CPU di differire la gestione delle interruzioni di bassa priorità senza mascherare tutte le interruzioni, e permette a un interruzioni di priorità alta di sospendere l’esecuzione della procedura di servizio di un’interruzione di priorità bassa.

ACCESSO DIRETTO ALLA MEMORIA (DMA): In molti calcolatori si evita di sovraccaricare la CPU assegnando alcuni compiti a un’unità d’elaborazione specializzata, detta controllore dell’**accesso diretto alla memoria**. Per dare avvio a un trasferimento DMA, la CPU scrive nella memoria un comando strutturato per il DMA. Esso contiene un puntatore alla locazione dei dati da trasferire, un altro puntatore alla destinazione dei dati, e il numero dei byte da trasferire. La CPU scrive l’indirizzo di questo comando strutturato nel controllore del DMA, e prosegue con l’esecuzione di altro codice. In alcune architetture per realizzare la tecnica DMA si usano gli indirizzi della memoria fisica, mentre in altre s’impiega l’**accesso diretto alla memoria virtuale** (DVMA): in questo caso si usano indirizzi virtuali che poi si traducono in indirizzi fisici.

**INTERFACCIA DI I/O PER LE APPLICAZIONI**: È necessario avere un trattamento uniforme dei dispositivi di I/O, quindi le chiamate di sistema di I/O incapsulano il comportamento dei dispositivi in alcuni tipi generali. I dispositivi differiscono in molti aspetti:

* **Trasferimento a flusso di caratteri o a blocchi**;
* **Accesso sequenziale o diretto**;
* **Dispositivi sincroni o asincroni**;
* **Condivisibili o riservati**;
* **Velocità di funzionamento**;
* **Lettura e scrittura, solo lettura o solo scrittura**.

DISPOSITIVI CON TRASFERIMENTO A BLOCCHI O A CARATTERI: L’interfaccia per i **dispositivi a blocchi** sintetizza gli aspetti per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocco di dati. Di solito le applicazioni comunicano con questi dispositivi tramite un file system che funge da interfaccia. Il sistema operativo e certe applicazioni particolari come quelle per la gestione delle basi di dati possono trovare più conveniente trattare questi dispositivi come una semplice sequenza lineare di blocchi. In questo caso si parla di I/O a **basso livello**.

I file possono essere associati alla memoria, facendo coincidere una parte dello spazio indirizzi virtuale di un processo con il contenuto di un file. Gli effettivi trasferimenti di dati sono eseguiti solo quando sono necessari per soddisfare una richiesta d’accesso all’immagine nella memoria.

I dispositivi a **flussi di carattere** invece, generano o accettano uno stream di dati, ad esempio tastiera, mouse, porte seriali. Le chiamate di sistema fondamentali per le interfacce di questo tipo permettono a un’applicazione di acquisire (get) o inviare (put) un carattere.

DISPOSITIVI DI RETE: La maggior parte dei S.O. fornisce un’interfaccia per l’I/O di rete diversa da quella per i dischi. L’interfaccia per l’I/O di rete fornita su molti S.O. è l’interfaccia di rete **socket**. Le chiamate di sistema di un’interfaccia socket permettono a un’applicazione di creare una socket, collegare una socket locale all’indirizzo di un altro punto della rete, controllare se un’applicazione sia inserita nella socket locale, e inviare o ricevere pacchetti di dati lungo la connessione. Per permettere lo sviluppo di server, l’interfaccia socket fornisce una funzione chiamata **select** che gestisce un insieme di socket e restituisce informazioni sulle socket per le quali sono presenti pacchetti che attendono d’essere ricevuti, su quelle che hanno spazio per accettare un pacchetto da inviare.

OROLOGI E TEMPORIZZATORI: La maggior parte dei calcolatori ha temporizzatori e orologi che forniscono tre funzioni essenziali:

* Segnare l’ora corrente;
* Segnalare il tempo trascorso;
* Regolare un temporizzatore in modo da avviare l’operazione (x) al tempo (t).

Il dispositivo che misura la durata di un lasso di tempo e che può avviare un’operazione si chiama **temporizzatore programmabile**; si può, infatti, regolare in modo da attente un certo tempo e poi generare un segnale d’interruzione, e può anche ripetere questo processo per un numero prefissato di volte. Per l’ora corrente vi è un contatore di interrupt ad altra frequenza chiamato **orologio hardware**.

I/O BLOCCANTE O NON BLOCCANTE: Un altro aspetto delle chiamate del sistema è la scelta tra I/O bloccate o non bloccante. Una system call **bloccante** sospende l’esecuzione dell’applicazione che l’ha invocata, questa passa dalla coda dei processi pronti alla coda d’attesa del sistema; quando la chiamata a sistema termina l’applicazione torna nella coda dei processi pronti.

Alcuni processi necessitano di una forma **non bloccante** di I/O. Una system call di questo tipo non attesta l’esecuzione dell’applicazione per un lasso di tempo significativo. Al contrario, essa restituisce rapidamente l’esecuzione dell’applicazione, fornendo un parametro che indica quanti byte di dati sono stati trasferiti.

Una possibile alternativa alle chiamate del sistema non bloccanti è costituita dalle chiamate del sistema asincrone. Esse restituiscono immediatamente il controllo al sistema, ma il completamento dell’I/O viene successivamente comunicato all’applicazione per mezzo di una variabile o di un interrupt software o tramite una routine di ritorno.

**SOTTOSISTEMA PER L’I/O DEL NUCLEO**: Il nucleo fornisce molti servizi riguardati l’I/O, molti sono offerti dal sottosistema per l’I/O del nucleo, e sono realizzati a partire dai dispositivi e dai relativi driver.

SCHEDULING: Fare lo scheduling di un insieme di richieste di I/O significa stabilirne un ordine d’esecuzione efficace. Lo scheduling può migliorare le prestazioni complessive del sistema, distribuire equamente gli accessi dei processi ai dispositivi e ridurre il tempo d’attesa medio per il completamento di un’operazione di I/O.

I progettisti si S.O. realizzano lo scheduling mantenendo una coda di richieste per ogni dispositivo. Quando un’applicazione richiede l’esecuzione di una chiamata del sistema I/O bloccante, si aggiunge la richiesta alla coda relativa al dispositivo appropriato. Lo scheduler di I/O bloccante riorganizza l’ordine della coda per migliorare l’efficienza globale del sistema e il tempo d’attesa cui sono sottoposte le applicazioni

MEMORIZZAZIONE TRANSITORIA: Una **memoria di transito** (buffer) è un’area di memoria che contiene dati mentre essi sono trasferiti fra due dispositivi o tra un’applicazione e un dispositivo. La memorizzazione transitoria si usa per tre ragioni:

1. La prima è la necessità di gestire la differenza di velocità fra il produttore e il consumatore di un flusso di dati;
2. La seconda riguarda la gestione dei dispositivi che trasferiscono dati in blocchi di dimensioni diverse;
3. Il terzo è la realizzazione della **semantica delle copie** nell’ambito dell’I/O delle applicazioni.

La **semantica delle copie** garantisce che la versione dei dati scritta nel disco sia conforme a quella contenuta nell’area di memoria al momento della chiamata del sistema, indipendentemente da ogni successiva modifica.

CACHE: Una cache è una memoria ad alta velocità utilizzata dai dispositivi di I/O per rendere più efficace il trasferimento di dati: l’accesso a queste copie è più rapido dell’accesso agli originali. La differenza fra una memoria di transito e una cache consiste nel fatto che la prima può contenere dati di cui non esiste un’altra copia, mentre una cache mantiene su un mezzo più efficiente una copia di informazioni già memorizzate.

CODA E USO ESCLUSIVO DEI DISPOSITVI: Una coda di file da stampare (spool) è una memoria di transito contenete dati per un dispositivo che non può accettare flussi di dati intercalati, ad esempio la stampante. Il sistema operativo risolve questo problema filtrando tuti i dati per la stampante: i dati da stampare provenienti da ogni singola applicazione si registrano in uno specifico file in un disco, quando un’applicazione termina di emettere i dati da stampare, si aggiunge tale file alla coda di stampa; quest’ultima viene copiata sulla stampante, un file per volta. La tecnica dell’accodamento (spooling) viene realizzata utilizzando lo spool. Questa tecnica viene utilizzata per ottenere un uso esclusivo dei dispositivi. In alcuni S.O. lo spooling è gestito da un processo di sistema specializzato (deamon di spool), il altri da thread del kernel.

GESTIONE DEGLI ERRORI: Un sistema operativo che usi la protezione della memoria può proteggersi da molti tipi di errori. I dispositivi e i trasferimenti di I/O possono causare errori in molti modi, sia per motivi transitori, come il sovraccarico di una rete di comunicazione, sia per motivi ‘permanenti’, come nel caso in cui il controllore dell’unità a disco sai difettoso. Nel caso di situazioni transitorie, solitamente il S.O. può tentare di recuperare la situazione, in situazioni con errori permanenti, la gestione degli errori è più difficile.

Alcuni tipi di dispositivi possono fornire informazioni assai dettagliate sugli errori, ad esempio, l’insuccesso di un’operazione di un dispositivo SCSI è registrato dal protocollo SCSI usando un **codice di rilevazione**.

STRUTTURE DI DATI NEL NUCLEO: Il nucleo ha bisogno di mantenere informazioni sullo stato dei componenti coinvolti nelle operazioni di I/O, e usa a questo fine diverse strutture dati interne, ad esempio la tabella dei file aperti. Il nucleo usa molte strutture di questo tipo per tenere traccia dei collegamenti in rete, delle comunicazioni con i dispositivi a caratteri e di altre attività connesse all’I/O.

**TRASFORMAZIONE DELLE RICHIESTE DI I/O IN OPERAZIONI DEI DISPOSITIVI**: Per illustrare come si posso giungere dal nome del file al controllore del disco, conviene innanzi tutto esaminare un sistema relativamente semplice come l’MS-DOS. La prima parte di un nome di file identifica uno specifico dispositivo. Ad esempio, C: è la parte inziale di ogni nome di file residente nell’unità a disco principale. Grazie all’uso dei due punti come separatore, lo spazio per i nomi dei dispositivi è distinto dallo spazio per i nomi del file.

UNIX, invece, rappresenta i nomi dei dispositivi all’interno dell’ordinario spazio dei nomi del file system. In un nome di percorso dello UNIX non c’è alcuna chiara separazione fra il dispositivo interessato e il nome del file in senso proprio. UNIX impiega una particolare tabella, detta **tabella di montaggio**, per associare i prefissi dei nomi di percorso ai corrispondenti nomi di dispositivi. Quando deve risolvere un nome di percorso, il sistema esamina la tabella per trovare il più lungo prefisso corrispondente: questo elemento della tabella indica il nome del dispositivo voluto. Il nome del dispositivo viene rappresentato come una coppia di numeri <**principale**,**scondario**>; dove **principale**, individua il driver del dispositivo che deve essere usato per gestire l’I/O; e **secondario**, individua l’indirizzo della porta o l’indirizzo in memoria del controller del dispositivo.

**PRESTAZIONI**: L’I/O è uno tra i principali fattori che influiscono sulle prestazioni di un sistema: richiede un impegno notevole della CPU per l’esecuzione del codice dei driver e per uno scheduling equo ed efficiente dei processi quanto essi sono bloccati e riavviati.

Per migliorare l’efficienza dell’I/O si possono applicare diversi principi:

* Ridurre il numero dei cambi di contesto;
* Ridurre il numero di copiature dei dati nella memoria durante i trasferimenti tra dispositivi e applicazioni;
* Ridurre la frequenza delle interruzioni tramite i trasferimenti di grandi quantità di dati in un’unica soluzione, usare controllori intelligenti e l’interrogazione ciclica;
* Aumentare il tasso di concorrenza usando controllori DMA intelligenti o canali di I/O per sollevare la CPU dalle semplici copiature di dati;
* Realizzare le primitive direttamente tramite i dispositivi fisici, così da permettere che la loro esecuzione sia simultanea alle operazioni di bus e di CPU;
* Equilibrare le prestazioni della CPU, del sottosistema per la gestione della memoria, del bus e dell’I/O giacché il sovraccarico di uno qualunque di questi settori provoca l’inutilizzo degli altri.