Sistemi Operativi

Secondo semestre

Giovanni Tosini

Indice

Capitolo 1

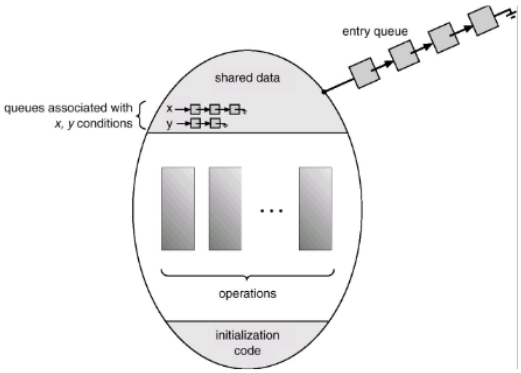
Monitor

Si tratta di una struttura simile ai semafori, implementa di default la mutua esclusione. Simile a una classe di un linguaggio a oggetti, può contenere:

- variabili(sono private)
- metodi(solo loro possono utilizzare le variabili definite all'interno del Monitor)
- costruttore

Quando un processo usa una entry del Monitor, gli altri non possono accedervi. Al suo interno posso dichiarare variabili di tipo CONDITION, non hanno valore, vengo usate esclusivamente con i metodi WAIT e SIGNAL.

```
CONDITION x
x.wait -> blocca il processo, non avvengono incrementi
          o decrementi come con i semafori
x.signal -> se fatto quando non ci sono processi in
            attesa, non fa niente
```



Il comando SIGNAL può rompere la mutua esclusione, perché un eventuale processo fermo potrebbe ripartire e in quel caso entrambi sarebbe all'interno della sezione critica, per evitare ciò il processo che la chiama:

- viene bloccato
- oppure SIGNAL deve essere l'ultima istruzione prima di uscire dal Monitor

Quale metodo usare? Dipende dal Monitor.

1.1 Problematiche

- esistono pochi linguaggi che li implementano
- possono essere applicati solo quando ci sono processi che condividono la memoria(stessa macchina)

Capitolo 2

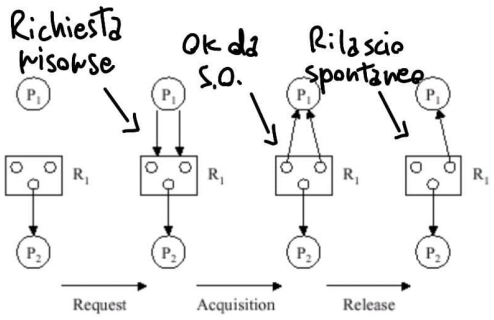
Deadlock

Causato quando un processo è in attesa di un evento causato a sua volta da un altro processo in attesa. Ci sono quattro condizioni che possono causare un Deadlock:

- mutua esclusione
- hold and wait: processo che detiene una risorsa ed è in attesa di una risorsa utilizzata da un altro
- no preemption: il processo deve rilasciare volontariamente la risorsa(non può essere ucciso)
- catena circolare: A attende B, che attende C, che attende A

In presenza di tutte e quattro esiste la possibilità che si verifichi. Tecnica di prevenzione: basta che una delle quattro non venga rispettata e il Deadlock non può avvenire.

Modello astratto RAG (Resource Allocation Graph)



- V = {{P1,P2},{R1}}
- $E_{iniziale}$ = {(R1,P2)} E_{finale} = {(R1,P1),(R1,P2)}

In generale, se abbiamo una sola istanza di ogni risorsa e siamo in presenza di un ciclo, il Deadlock è certo. Con risorse con più istanze invece non è assicurato.

2.1 Gestione Deadlock

Prevenzione statica: evitare ovvero che si verifichino una della quattro condizioni sopra riportate, se ci sono risorse disponibili non le assegno a prescindere.

- mutua esclusione: non posso toglierla
- hold and wait: accumulo tutte le risorse tutte in una volta altrimenti non procedo questo porta a delle problematiche:
  - basso uso delle risorse
  - starvation
  - occorre conoscere le risorse necessarie
- no preemption: un processo che richiede una risorsa non disponibile sarà costretto a rilasciare tutte le altre risorse che stava tenendo
- ogni risorsa avrà una priorità crescente, il processo può richiederle esclusivamente in ordine crescente

Prevenzione dinamica, durante l'esecuzione si blocca un processo in base al RAG, occorre conoscere a priori il caso peggiore in cui si causa un Deadlock e permette di sfruttare maggiormente le risorse per altri processi diversamente da quella statica. Guardando il RAG, come un processo richiede una risorsa

ipotizza di concederla e verifica se si presenta un ciclo, in caso affermativo la richiesta verrà rifiutata, il processo rimarrà in attesa e il sistema resterà in un stato SAFE.

x	Richieste	Possedute
p0	10	5
p1	4	2
p2	9	2

p0 avrà bisogno di 10 istanze delle quali già ne possiede 5,12 sono quelle presenti, 9 in uso e 3 libere. Stato SAFE o UNSAFE? Si va in ordine per processo:

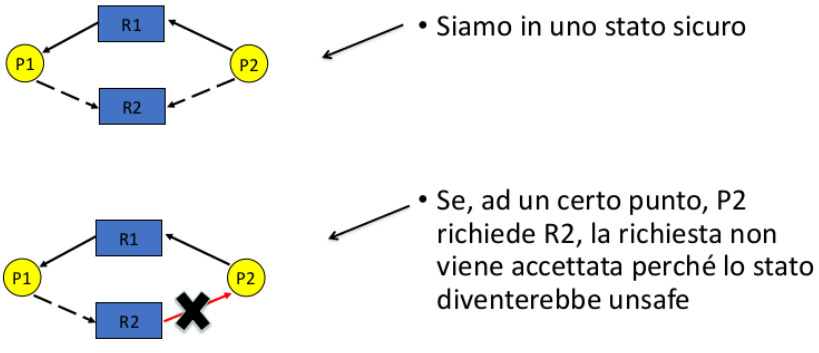
- p0 ne sta usando 5, ne occorrono altre 5, con le 3 libere non può essere soddisfatto, di conseguenza rimarrà in attesa
- p1 ne chiede 4, ne ha già 2, 3 sono libere, di conseguenza se andasse prima lui ne avremmo 5 libere dopo

Quindi saremmo in uno stato SAFE.
N.B.: lo scheduler in tutto ciò non ha rilevanza, lui si limita scegliere quali processi mettere nella CPU.

Lo svantaggio della previsione dinamica è l'uso delle risorse minore, perché ci possono esserne alcune che non vengono sfruttate.

1) L'algoritmo RAG funziona solo con risorse che possiedono un'unica istanza, funzionamento:

- aggiungo al RAG l'arco di reclamo, ovvero in futuro quel processo richiederà l'uso di quella risorsa molto probabilmente
- il S.O. permetterà al processo di usare la risorsa solo se ipotizzando che venga fatto, non si verifichi un ciclo



La freccia tratteggiata indica l'arco di reclamo.

2) L'algoritmo del banchiere è meno efficiente dell'algorimo RAG, ma funziona con qualsiasi numero di istanze delle risorse. Si divide in due parti, una che simula la cessione dell'istanza della risorsa e una che ne verifica l'effetto.

```
int available[m];  /* n°  di istanze di R_i disponibili */
int max[n][m];     /* matrice delle richieste di risorse */
int alloc[n][m];   /* matrice allocazione corrente */
int need[n][m];    /* matrice bisogno rimanente ovvero
                      need[i][j] = max[i][j] - alloc[i][j]*/
```

Algoritmo di allocazione (P_i)

```
void request(int req_vec[]) {
   if (req_vec[] > need[i][])
      error();   /* superato il massimo preventivato */
   if (req_vec[] > available[])
      wait();   /* attendo che si liberino risorse */
   available[] = available[] - req_vec[];
   alloc[i][] = alloc[i][] + req_vec[];
   need[i][] = need[i][] - req_vec[];
   if (!state_safe()) {/* se non è safe, ripristino il vecchio stato */
      available[] = available[] + req_vec[];
      alloc[i][] = alloc[i][] - req_vec[];
      need[i][] = need[i][] + req_vec[];
      wait();
   }
}
```

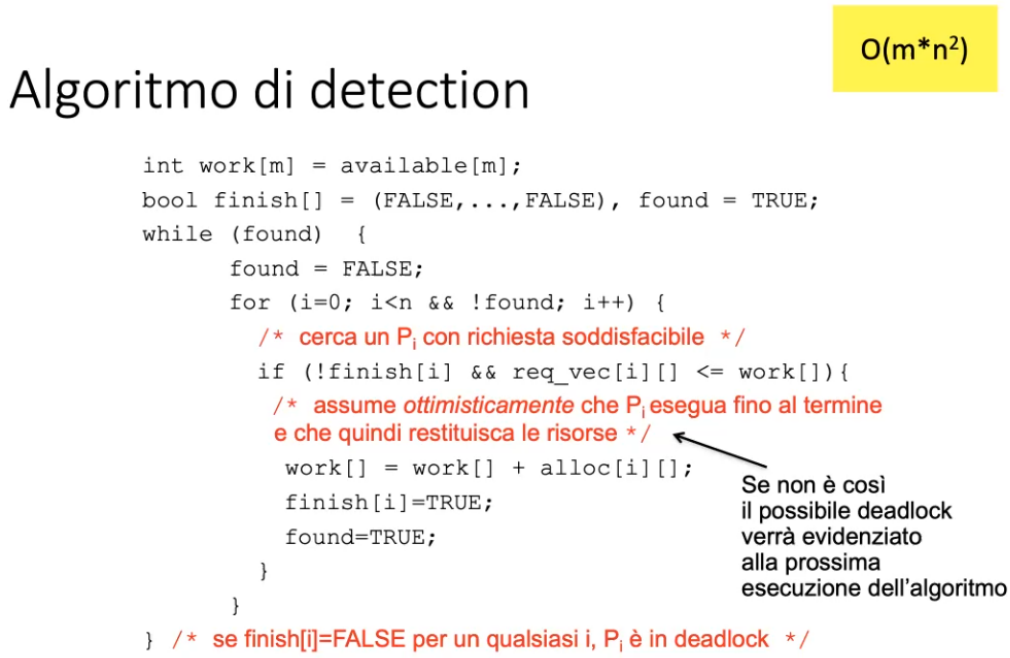
Richieste del processo P_i

"simulo" l'assegnazione

rollback

3) L'algoritmo di rilevazione permette che ci siano dei Deadlock, ogni tanto verifica che il sistema sia caduto in Deadlock in un metodo simile all'algoritmo del banchiere senza la necessità di conoscere la matrice max, verifica solo se il sistema è in uno stato SAFE. Usa strutture dati simili a quello del banchiere.

```
int available[m]; //n° istanze di R_i disponibili
int alloc[n][m]; //matrice allocazione corrente
int req_vec[n][m]; //matrice di richiesta,
                   //stesse dimensione della max,
                   // è una max istantanea
```



Parte dal presupposto che tutti i processi siano bloccati, verifica se il singolo processo non ha finito e se le risorse che richiede siano inferiori a quelle disponibili. come trova un processo che può proseguire con la sua esecuzione significa che non si è in Deadlock, non verifica la situazione futura, solo quella attuale. Esempio:

Rilevazione - esempio

- 5 processi: P_0,P_1,P_2,P_3,P_4
- 3 tipi di risorsa:
  - A (7 istanze), B (2 istanze), C (6 istanze)
- Fotografia al tempo T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Usando l'algoritmo si passa man mano ogni singolo processo e si verifica se nello stato attuale può finire la sua esecuzione oppure è fermo a causa di un'attesa di risorse. Man mano che si trovano processi che possono concludere, si considerano le risorse in loro possesso come possibili risorse future disponibili.

L'algoritmo di rilevazione può essere lanciato:

- dopo ogni richiesta di risorsa
- ogni N secondi
- quando la percentuale di utilizzo della CPU cala sotto una soglia T

La prima è particolarmente costosa, non si fa prevenzione, ma si verifica sempre, che può essere problematico essendo un algoritmo che può arrivare ad avere una complessità cubica. La terza opzione potrebbe non rilevare dei Deadlock causati da piccoli gruppi di processi che non causano grossi cali d'uso della CPU.

Quando ci si accorgerà del Deadlock si potrà:

- uccidere tutti o alcuni dei processi coinvolti

- prelazionare tutti i processi coinvolti

In entrambi i casi sarebbe un grosso danno, perché i processi hanno lavorato fino a quel momento per niente. Per la prima, invece di uccidere tutti i processi si può scegliere di ucciderne uno alla volta in base a alle risorse allocate, a quante ne mancano, a quanto tempo mancava, etc e verificare dopo se il Deadlock è tuttora presente o meno. La seconda possibilità invece porta il problema che prelazione un singolo processo equivale quasi a ucciderlo visto che non può continuare normalmente come prima, una possibile soluzione è il **rollback** che comporta un lavoro in precedenza eseguito dalla CPU per salvare uno stato in cui il processo non era bloccato. Successivamente quest potrebbe causare un fenomeno di starvation se si andrà a fare **rollback** dei soliti processi, in quel caso una possibile soluzione potrebbe essere quella di considerare il numero di **rollback** nei fattori di costo.

Capitolo 3

Gestione della memoria

Un processo ha bisogno sia di memoria RAM che di memoria di massa per poter lavorare. La memoria serve ai processi per via dell'uso del **Program Counter** da parte della CPU . Possono nascere svariati problemi nella gestione della memoria per i processi:

- l'allocazione della memoria dei singoli processi
- protezione dello spazio allocato
- condivisione dello spazio allocato
- gestione dello swap
- gestione della memoria virtuale

Ogni programma per essere eseguito deve essere messo in memoria e trasformato in processo, da quel momento la CPU preleverà istruzioni dalla memoria in base al **Program Counter**, questo può portare all'ulteriore prelievo di dati dalla memoria fino alla fine dell'esecuzione con l'eventuale scrittura in memoria del risultato, una volta che il processo terminerà la memoria verrà rilasciata.

Come avviene il passaggio da programma a processo? La trasformazione avviene tramite una trasformazione di indirizzi di memoria.

```
int x = 3;
```

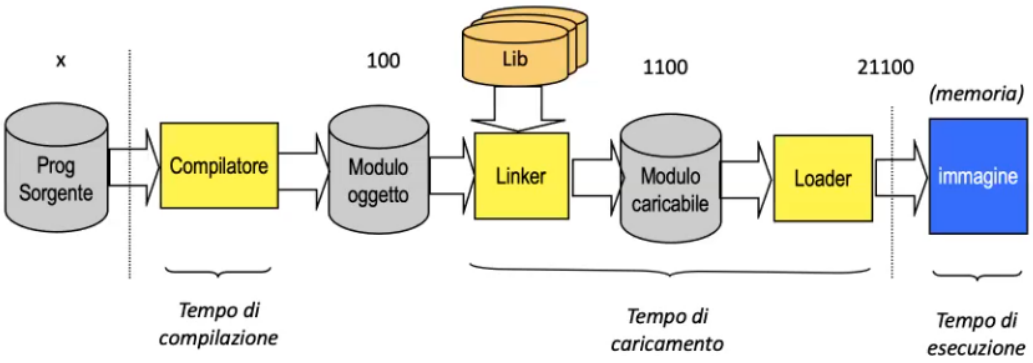
non è altro che un riferimento a un indirizzo di memoria, quando il programma verrà eseguito, quell'indirizzo **simbolico** verrà tradotto in un indirizzo **fisico**.

Chi compie questa trasformazione?

il compilatore: prende l'indirizzo **simbolico** e lo trasforma in una serie di indirizzi **rilocabili** ovvero che si possono posizionare da un'altra parte

il linker e/o il loader: prendono l'indirizzo **rilocabile** e lo trasformano in indirizzo fisico assoluto

Possono essere coinvolti tutti o no, in base a come avviene la trasformazione.



La variabile X viene presa dal **compilatore** e trasformata da indirizzo **simbolico** a indirizzo **rilocabile**, viene creato un modulo oggetto, in cui alla variabile viene assegnato un indirizzo di memoria, X non sarà più X, sarà l'area di memoria all'indirizzo 100. Il **linker** collegherà l'indirizzo 100 con gli indirizzi delle eventuali librerie incluse nel programma, ciò causerà la creazione di un modulo ricaribile in memoria. La X dall'indirizzo 100 sarà spostata all'indirizzo 1100 per fare spazio alle eventuali librerie. Il modulo caricabile è quello che effettivamente verrà caricato in memoria dal **loader**. Il SO assegnerà uno spazio all'interno della memoria tenendo conto dello spazio effettivamente occupato al momento. Eventualmente se l'indirizzo 1100 fosse occupato, il SO potrà spostarlo a un altro indirizzo libero, fatto questo il processo verrà eseguito. Se si troverà un'istruzione

```
int a = x + b;
```

vedendo che l'operando da prelevare è X, verrà ripescato dall'indirizzo di memoria, tale azione che collega indirizzi **simbolici** a indirizzi **fisici** viene denominata **binding degli indirizzi**.

3.1 Tempi di compilazione

compile-time: avviene a tempo di compilazione,in quel caso ci sarà solo il **compilatore** che non può sapere se l'indirizzo che assegna al programma sia occupato o meno, di conseguenza in caso in cui fosse occupato il programma semplicemente non verrà eseguito. Ciò funziona bene se la memoria è ottimizzata al massimo. Per cambiare la locazione del programma l'unica opzione sarà quellla di ricompilarlo.

load-time: avviene a tempo di caricamento, il **loader** può notare che l'area di memoria in cui vuole caricare sia occupata e di conseguenza cambia l'area in cui andare a caricare. Il codice sarà rilocabile, anche se non permette di spostare il processo mentre è in esecuzione. Se cambiasse l'indirizzo di riferimento sarà necessario ricaricare.

run-time: permette di spostare l'indirizzo di memoria allocato mentre il processo è in esecuzione, è necessario un supporto hardware perché decisamente più efficiente.

I primi due tipi di **binding** sono considerati **statici** mentre l'ultimo è considerato **dinamico**.

3.2 Linking

Può essere diverso indipendentemente dal tipo di **binding**.

statico: tradizionale, prima di mandare in esecuzione il processo, si è creato un'immagine di memoria con tutte le librerie incluse.

dinamico: il sistema viene preparato per accogliere un'immagine del codice e delle librerie, ma il linking vero e e proprio avverrà solo al momento dell'esecuzione, l'eseguibile sarà più piccolo perché non conterrà le librerie, ma solo il loro riferimento.

Un programma linkato staticamente occuperà una maggior memoria rispetto a un programma linkato dinamicamente, dal punto di vista di caricamento il dinamico vince sullo statico, in quel caso gli stub creati dal linkato dinamico dovranno essere completati a **run-time** e ciò porterà il programma a essere più lento rispetto a quello linkato staticamente.

Ricapitolando:

linking statico:
- occupa maggior memoria

- lento nel caricamento
- veloce nell'esecuzione

linking dinamico:
- occupa meno memoria
- più rapido nell'avvio
- più lento nell'esecuzione

3.3 Loading

Lo stesso concetto di statico e dinamico può essere applicato al **loader**.

statico: tutto il codice viene caricato in memoria al tempo di esecuzione, quindi deve essere disponibile in memoria prima di eseguire il programma

dinamico: i moduli che servono al processo vengono caricati al primo utilizzo, se una parte non serve non verrà caricata di conseguenza

Uno dei malus dello **statico** è la possibilità di programmi in cui una parte del codice non verrà mai eseguita (magari per via di condizioni), non conviene di conseguenza caricare completamente il programma, in quel caso il loading **dinamico** diventa decisamente più utile.

3.4 Spazio di indirizzamento logico

Consiste nello spazio visto dalla CPU quando esegue un processo, mappato su uno spazio di indirizzamento fisico in RAM. Il SO deve mappare indirizzi logici o virtuali su indirizzi fisici reali, quando si fa **binding** a **compile-time** o **load-time** l'indirizzo logico e fisico coincidono. Invece con un **binding** a **run-time** l'indirrizzo logico generato dalla CPU potrebbe non coincidere con quello fisico presente in RAM, tutto ciò viene sempre gestito dal SO. Una gestione dinamica come questa ha un costo, ovvero il tempo che ci mette la MMU (Memory Management Unit) che tiene conto del riferimento tra indirizzo logico a quello fisico a tradurre la richiesta.

La MMU può essere composta da un registro e un sommatore che somma gli indirizzi logici con un offset, potrebbe essere problematico con processi che dovrebbero essere divisi all'interno dello spazio di memoria.

Capitolo 4

Schemi di gestione della memoria

4.1 Allocazione contigua

L'immagine di memoria del processo sarà tutto allocata in un'area contigua per l'appunto, senza buchi in mezzo. Per gestire tale l'allocazione in questo modo esistono delle varianti:

4.1.1 Partizioni fisse

La memoria è divisa a blocchi, ciascun blocco con una determinata dimensione che rimarrà fissa per sempre, tra di loro possono differire in dimensione, ma rimarranno fisse. Il processo che arriva verrà posizionato nella prima partizione che permette di contenerlo. Un contro è che se un blocco viene occupato solo in parte, quella libera sarà sprecata, perché il blocco è fisso e non può essere partizionato. Esistono delle opzioni:

- un'unica coda di attesa, con i processi in fila e come si libera uno spazio il processo in testa verificherà se riesce a entrarci o meno
- tante code, una per partizione, il SO fa un preselezione del processo che verrà messo nella coda più adatta a lui

Entrambe hanno vantaggi e svantaggi, una coda per partizione causerà la possibilità di avere delle code vuote e altre piene, in quel caso si avrebbe tanta memoria sprecata e pochi processi in coda. Va bene solo nel caso in cui il carico dei vari processi è ben distribuito e di varie dimensioni. Se tutte le code fossero piene, sarebbe ottimale per l'uso della memoria.

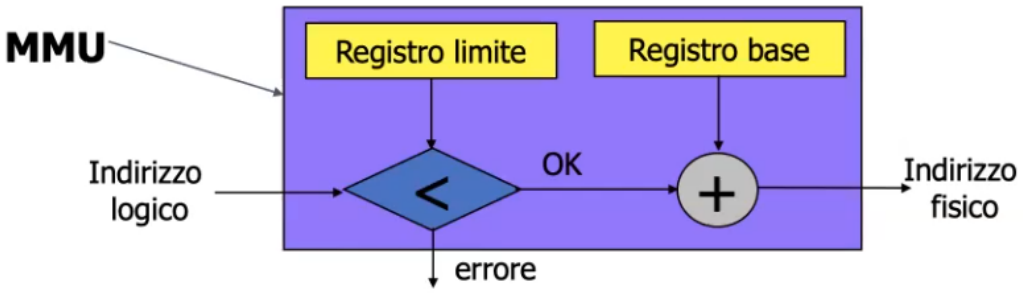
Nell'alternativa con un'unica coda, la gestione sarà più complicata, perché un processo troppo grande per stare in partizioni troppo piccole rimarrebbe

in attesa in caso di algoritmo FCFS, bloccando tutti gli altri processi dietro di lui che potrebbero entrare nelle partizioni libere. Una scansione della coda quando si libera una partizione risolve il problema, ci sono due varianti

Best-fit: scansiona tutta la coda scegliendo quello più adatto alla partizione che si è liberata ovvero quello che si avvicina a occuparla maggiormente, se la coda è molto lunga il tempo impiegato sarà decisamente elevato;

Best-available-fit: viene scelto il primo processo che può stare nella partizione che si è liberata, non minimizza lo spreco, perché potrebbe esserci un processo che occuperebbe meglio la partizione.

Lo schema della MMU con un allocazione contigua a partizioni fisse è il seguente:



Pro e contro:

Pro: semplicità, il SO deve solo tenere traccia di una tabella in cui si definisce in quale partizione andrà il processo

Contro: il livello di multiprogrammazione è vincolato dal numero di partizioni, inoltre c'è uno spreco di memoria causato dalla frammentazione:

Interna: causata dall'assegnamento di un processo a una partizione che non viene occupata al 100

Esterna: la somma della parte libera di partizioni parzialmente occupate potrebbe essere grande a sufficienza per un processo che invece sarà costretto ad attendere

la frammentazione riduce l'efficacia nell'uso della memoria, la tecnica con partizioni fisse soffre di entrambe, è più adatta per processi delle dimensioni giuste ed esatte alle partizioni che si vogliono occupare, quindi più verso i sistemi embedded.

4.1.2 Partizioni variabili

Il processo che arriva in memoria si prenderà una fetta di memoria grande esattamente quanto lo spazio che occuperebbe. La frammentazione esterne rimarrà, esempio:

- si crea una partizione da 100, una da 50, una da 30 e una da 20
- si libera quella da 50 e da 20
- un processo che occupa 70 potrebbe entrare se le posizioni da 50 e 20 venissero sommate, ma non si può a causa dell'allocazione contigua

Si genera un effetto a "groviera".

4.1.3 Tecnica del Buddy System

Un compromesso tra partizioni fisse e variabili, ovvero ha partizioni fisse, ma anche variabili, le dimensioni non sono customizzate, ma non rimangono fisse. Bisogna immaginare la memoria un blocco fisso, come arriva un processo parte il sequente schema:

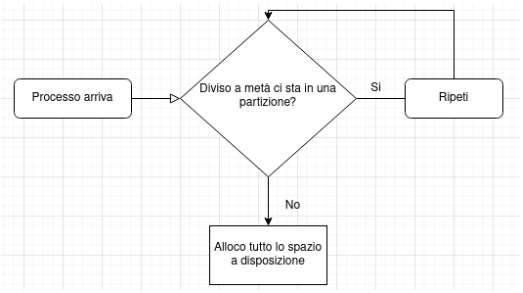


Figura 4.1: Flowchart Buddy System

Questo crea liste di blocchi liberi in base alla dimensione, al processo successivo si andrà a guardare nella lista se ci sta altrimenti si ripete la divisione.

4.2 Paginazione

Elimina totalmente la frammentazione esterna. La memoria fisica viene suddivisa in tante piccole aree della stessa dimensione detti **frame**. Lo spazio di allocazione non deve essere contiguo, al processo che arriva si daranno piccole aree fino a occupare tutte quelle necessarie. Il processo viene diviso, quando un processo dovrà essere eseguito in CPU, SO dovrà ricomporlo, quindi occorre una mappa delle locazioni.

La CPU è a conoscenza degli indirizzi logici, la memoria logica viene divisa in blocchi della stessa dimensione le **pagine**, il SO trasforma le pagine in frame. Entra in aiuto una tabella delle pagine, ovvero una mappa della memoria specifica per ogni singolo processo, attraverso la quale il SO riesce a tradurre l'indirizzo logico che serve alla CPU in un indirizzo fisico e recuperare di conseguenza le informazioni necessarie.

Esempio:

- dimensione della pagina = 1KB
- dimensione del programma = 2.3KB
- saranno necessarie 3 pagine, dell'ultima si userà solo 0.3KB

Come si può notare sarà ancora possibile la frammentazione interna.

Come avviene la ricerca nella tabella delle pagine? L'indirizzo logico richiesto dalla CPU è composto dal numero di pagina e da un offset, SO tramite il numero di pagina andrà alla ricerca nella tabella delle pagine del processo in questione, una volta trovato otterrà l'altra metà di indirizzo fisico da sommare all'offset per ottenere l'indirizzo fisico effettivo in memoria.

Ovviamente la tabella risiede in memoria, ed è come già detto specifica del processo. Nei registri della CPU sarà presente la locazione della tabella e anche la sua lunghezza, a ogni context switch tale valore viene aggiornato.

Questo causa un doppio accesso alla memoria: uno per la tabella delle pagine e uno per l'istruzione/dato necessari, per questo si usa una cache dedicata, **Translation look-aside table**(TLB), servirà a evitare il doppio accesso in memoria, in caso di cache miss, ovvero TLB non sarà in possesso della pagina, si dovrà per forza fare il doppio accesso in memoria.

Il costo effettivo di accesso in memoria sarà dato dalla somma del cache hit all'accesso in memoria:

$$\overbrace{(T_{MEM} + T_{TLB})\alpha(\text{parametro})}^{\text{cache hit}} + \overbrace{(2T_{MEM} + T_{TLB})(1-\alpha)}^{\text{accesso in memoria}}$$

La tabella delle pagine è utile anche per altro:

Bit di validità: identifica se l'area di memoria è stata mappata per il processo o no;

Caratteristiche: un bit che comunica se la pagina è solamente leggibile o no, per esempio il codice di una programma è in sola lettura, oppure se la pagina sia eseguibile o meno.

Quanto è grande una tabella della pagine? La dimensione è stabilita in base alla memoria massima e alla configurazione, ovvero quanto grande è ogni frame della memoria. In un ipotetico spazio di indirizzamento virtuale grande 2^{64}, se ogni pagina avesse una dimensione di 4KB ovvero 2^{12} potrei avere un numero di frame in memoria corrispondente al rapporto tra la memoria indirizzabile totale e la dimensione della singola pagina.

$$2^{64}/2^{12} = 2^{52}$$

Un numero decisamente enorme che occuperebbe una grossa fetta della RAM da moltiplicare per il numero di processi attivi. Si possono usare due metodi per evitare ciò:

- una tabella delle pagine invertita;
- oppure la paginazione multilivello.

4.2.1 Tabella delle pagine invertita

Una tabella unica nel sistema e non per processo, indicizzata per frame che contiene:

- l'indirizzo virtuale della pagina che occupa quel frame
- informazioni sul processo che usa la pagina

Il tempo di traduzione degli indirizzi cresce perché non sarà più una ricerca per indice, ma per contenuto. Perché la tabella è indicizzata per frame, mentre la CPU richiederà per pagina, il SO dovrà scorrere la tabella alla ricerca della pagina e solo così otterrà il frame corrispondente che sarà l'indice del contenuto. Questo viene facilitato tramite una struttura a tabella hash, invece di usare una ricerca lineare. Cambia come viene generato l'indirizzo richiesto dalla CPU, sarà diviso in tre parti:

- offset;
- numero di pagina;
- PID del processo.

Con il numero di pagina e il PID ho la chiave per la tabella hash, così da ottenere il valore ricercato per la traduzione ovvero l'indice, perché la tabella è indicizzata per frame!

4.2.2 Paginazione multilivello

Viene paginata anche la tabella delle pagine, quindi spezzettata per fare in modo che alcuni pezzi non siano in RAM, lasciando spazio libero in memoria per altro di più utile. Questa azione può essere ripetuta anche per i pezzi delle tabella delle pagine, per più livelli di paginazione. Esempio:

- indirizzo logico a 32 bit;
- dimensione della pagina = 4K = 2^{12};
- 12 bit per offset(d);
- 20 bit per numero di pagina:
  - 10 bit per l'indice della tabella delle pagine esterna;
  - 10 bit per l'offset all'interno della pagina della tabella delle pagine interna.

Il costo per un accesso in memoria con una paginazione multilivello cambia di conseguenza nello speficico, l'accesso in memoria sarà moltiplicato per $n + 1$ livelli di paginazione:

$$(T_{MEM} + T_{TLB})\alpha + ((n + 1)T_{MEM} + T_{TLB})(1 - \alpha)$$

Il +1 causato dall'accesso al dato.

4.3 Segmentazione

Il problema della paginazione è che lo spezzettare a blocchi il processo viene fatto senza rispettare la struttura dati creata dal compilatore, sarebbe più utile avere pagine contenenti blocchi interi utili: una pagina con il main, una per le funzioni, etc... Finché la pagina ha una dimensione fissa questo non si può ottenere, la **segmentazione** è analoga alla paginazione, ma non si vincola più alla dimensione di una pagina fissa. Dal punto di vista di SO, avremo una tabella dei segmenti che verranno tagliati a dimensione variabile. Un segmento di dimensioni variabili torna ad avere la problematica dell'allocazione contigua, ovvero che non può essere messo dovunque, ma in blocchi unici. Nella tabella dei segmenti, ogni entry conterrà oltre che all'indirizzo base del segmento anche la sua dimensione, che precedentemente non era necessaria con le pagine.

Come cambia la traduzione? La CPU richiederà un indirizzo composto da una parte segmento e una parte offset, l'offset verrà analizzato, per verificare se va oltre la dimensione del segmento o se rimane al suo interno (ecco da cosa viene causato il segmentation fault). La parte di offset verrà sommata al valore restituito dalla tabella dei segmenti, che ricordiamo contenere la dimensione del segmento e la base da sommare eventualmente all'offset. Esempio di traduzione:

Segmento	Limite	Base
0	600	219
1	14	2300
2	100	90
3	580	1327

A ogni segmento si avrà appunto il suo limite e la sua base, per esempio

- se venisse richiesto il segmento 0 e l'offset 430 si controlla il segmento 0, si nota che l'offset 430 è all'interno del limite di conseguenza l'indirizzo fisico sarà dato da: offset + base = 649.
- segmento 1 offset 20? segmentation fault perché 20 è superiore al limite 14

La segmentazione mi garantisce di gestire ancora meglio tutti gli aspetti visti nella paginazione, perché adesso avrò la certezza che il contenuto del segmento sarà omogeneo.

Svantaggi della segmentazione Torna il problema della frammentazione esterna, la pagina può essere messa in qualunque spazio, perché sono tutti uguali, mentre il segmento essendo variabile ma richiedendo un'allocazione contigua non può e ciò causa la frammentazione esterna. Una possibile soluzione al problema è l'unione della paginazione e della segmentazione, segmento il processo e ogni segmento lo pagino, segmentandolo prima mantengo l'unità logica che verrà paginata omogeneamente in multilivelli.

4.4 Paginazione vs Segmentazione

Vantaggi della paginazione:

- non esiste frammentazione, a parte una piccola frammentazione interna;
- l'allocazione dei frame non richiede algoritmi specifici;

Svantaggi della paginazione:

- non mantiene l'omogeneità;

Vantaggi della segmentazione:

- omogeneità;
- associazione di protezione/condivisione ai segmenti;

Svantaggi della segmentazione:

- richiede degli algoritmi di allocazione dinamica dello spazione, essendo i segmenti variabili;
- reintroduce la problematica della frammentazione esterna;

Unendole si porteranno dietro i vantaggi di ciascuna limitando gli svantaggi.

4.5 Segmentazione paginata

L'indirizzo logico sarà composto da segmento e offset, il numero di segmento verrà sommato al valore all'interno **Segment Table Base Register** (indica dove si trova la tabella dei segmenti all'interno della memoria) per andare a indicizzare la tabella dei segmenti. All'interno della tabella dei segmenti si troverà la lunghezza del segmento e la **Page Table Base** in cui si troverà l'indirizzo di locazione della tabella delle pagine specifica di quel segmento. L'indirizzo verrà combinato con l'offset, che è composto da due parti, una che indica il numero di pagina e una che indica l'offset effettivo. Il numero di pagina verrà sommata al **Page Table Base Register** per recuperare la riga effettiva della tabella delle pagine, una volta indicizzata la riga in questione, si sommerà il valore del frame all'offset effettivo e si farà l'accesso alla memoria per recuperare il dato necessario.

Capitolo 5

Memoria virtuale

5.1 Motivazioni

L'immagine di memoria di un processo è davvero necessaria che sia **tutta** caricata in RAM? In generale no, serve solo una parte, ovvero quella che rappresenta la località spazio temporale necessaria. Quella che non serve verrà lasciata sul disco, questo permetterà di lasciare spazio libero in memoria per altre immagini di memoria più utili e avvantaggiare la multiprogrammazione. Lo spazio logico sarà più grande dello spazio in memoria, perché sarà composto dalla somma di tutte le immagini dei processi che saranno in parte in RAM e in parte sul disco.

5.2 Paginazione su domanda

Una pagina si troverà in memoria solo quando necessaria. I vantaggi sono i seguenti:

- meno memoria occupata da ogni processo;
- se il livello di multiprogrammazione fosse troppo alto, l'eventuale swap out verrà fatto solo a una parte delle pagine riducendo i tempi di I/O;

Per poter implementare occorre sapere se la pagina richiesta dalla CPU sia valida o no, entra in gioco il bit di validità presente nella tabella delle pagine. Quel bit segnalerà se la pagina si trova in RAM, su disco oppure se la pagina non si trova nell'intervallo di memoria. Quando una pagina è mappata su un frame che non si trova in RAM, bit validità = 0, avviene un **page fault**. Quindi:

1. controllando sulla tabella delle pagine risulterà che la pagina è invalida, non è in RAM, di conseguenza sarà su disco;
2. parte un interrupt, il **page fault**;
3. il S.O. verificherà che la pagina si trovi nella parte di disco dedicata alla funzione di RAM, (**Backing Store** o **Area di Swap**);
4. localizza la pagina e la riporta nella memoria, cercherà un frame e caricherà la parte di memoria recuperata;
5. reset della **page table** per indicare che ora la pagina è in memoria con annessa aggiunta dell'indirizzo e modifica del bit validità;
6. il processo verrà rimesso in ready queue per essere pronto all'esecuzione.

Questo avviene a ogni **page fault**.

5.2.1 EAT di un Page Fault

Introduciamo un parametro p (tasso di page fault), che vale in questo intervallo $0 \leq p \leq 1$, dove

- p = 0, nessun page fault;
- p = 1, ogni acceso è un page fault.

$$EAT = (1 - p) * t_{mem} + p * t_{\text{page fault}}$$

La parte $p*t_{\text{page fault}}$ è la moltiplicazione tra la percentuale di page fault e il tempo richiesto per recuperare il dato su disco, mentre t_{mem} è il tempo di accesso alla memoria tramite il meccanismo di gestione dell'allocazione di memoria ai processi. Il $t_{\text{page fault}}$ è composto dalla gestione dell'interrupt che arriva, il context switch per permettere al S.O. di gestire la situazione, torna al processo che era stato interrotto (context switch).

- gestione dell'interrupt (primo context switch);
- swap in, lettura della pagina mancante da disco a memoria;
- costo di riavvio del processo (secondo context switch);
- (opzionalmente) swap out della pagina che devo estrarre per fare spazio a quella richiesta.

Ci sono ben due context switch.

5.2.2 Rimpiazzamento delle pagine

In caso di assenza di frame liberi in cui mettere la pagina recuperata:

- tramite un algoritmo specifico di rimpiazzamento delle pagine si trova una pagina da sostituire;
- avviene lo swap su disco della pagina sostituita;
- swap dell'altra pagina nel frame dal disco;
- modifica le tabelle (tabella delle pagine e bit di validità);
- ripristino dell'istruzione che ha causato il page fault.

Rimpiazzare una pagina costa il doppio di una normale gestione del page fault. Per questo entra in gioco l'utilità del **dirty bit** ovvero un bit che specifica se la pagina è stata modificata mentre si trovava in memoria o meno, in caso di una modifica occorre ricaricarla su disco. Rimpiazzare una pagina che non è stata modifica costa meno, perché si può semplicemente sovrascrivere.

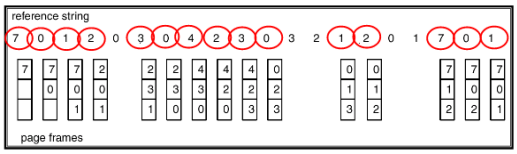
5.2.3 Algoritmi di rimpiazzamento delle pagine

Obiettivo: minimizzare il tasso di page fault, dipenderà dalla scelta delle pagine da caricare in memoria. Un metodo è l'uso di una stringa di riferimenti. Ipotizzando pagine grandi 100 byte, la stringa sarà composta dalla cifra delle centinaia delle pagine che verranno richieste. Esempio:

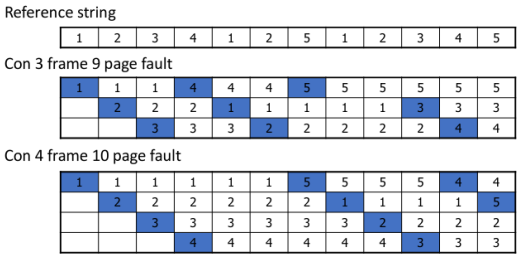
- indirizzi generati: 100, 604, 128, 130, 256, 260, 264, 268;
- la stringa di riferimento sarebbe composto da: 1, 6, 1, 1, 2 ,2, 2, 2;
- in realtà sarà composta da: 1, 6, 1, 2. Perché il page fault sarà possibile con la prima pagina, sicuramente non con quelle successive.

5.2.4 Algoritmo FIFO

Prima pagina che carico sarà la prima scelta a essere rimossa per liberare eventualmente spazio, non è adatto nè per lo scheduling nè per la gestione della memoria. Può causare l'anomalia di Belady, ovvero aumentando la disposizione di frame in memoria potrebbero aumentare i page fault diversamente da quello che ci si aspetterebbe.



(a) ***Esempio con algoritmo FIFO, causa 15 page fault***



(b) ***Anomalia di Belady, all'aumentare dei frame aumentano i page fault***

5.2.5 Algoritmo Ideale

L'**algoritmo ideale**, dovrebbe essere in grado di scegliere la pagina che non userò per più tempo, garantendo che per un po' non avrò page fault.

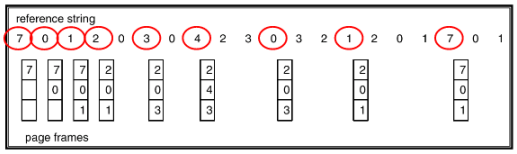


Figura 5.1: Esempio di algoritmo ideale

5.2.6 Algoritmo LRU

L'algoritmo **Least Recently Used**, approssima un possibile algoritmo ideale guardando al passato, rimpiazza la pagina che non viene utilizzata da più tempo.

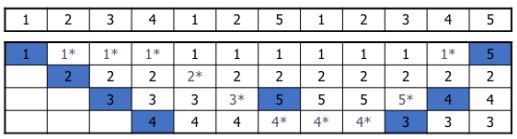
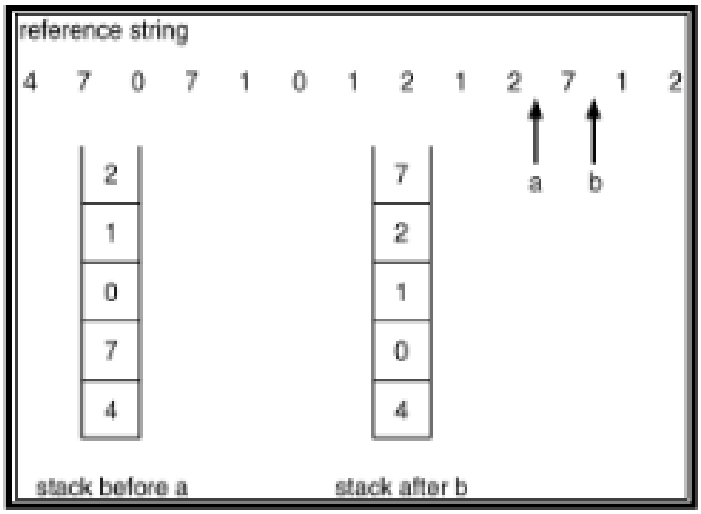


Figura 5.2: Esempio di algoritmo LRU, uso dell'asterisco per indicare la pagina meno utilizzata da più tempo

Lo si implementa tramite un contatore, all'interno si salverà il time stamp cioè il clock di sistema, indica da quanto non viene usata una pagina, il tutto verrà memorizzato nella tabella delle pagine. Inoltre quando avviene un page

fault, sarà necessario far partire una ricerca tra tutte le pagine per vedere quale ha il time stamp più vecchio. Il tutto è decisamente costoso perché appunto sarà necessario avere il time stamp di tutte le pagine e successivamente bisognerà controllarle tutte alla ricerca di quelle più vecchio.

Una possibile approssimazione per risparmiare spazio e tempo perdendo un po' di precisione è tramite l'uso dello stack. Si evita di memorizzare il clock per ogni pagina, ma si crea una pila, uno stack in cui si tiene un ordine di accesso. In fondo sarà presente la pagina più vecchia, man mano che viene utilizzata si metterà in cima, trovando in fondo la meno usata e in cima la più usata. Sarà comodo perché la vittima è sempre quella in fondo, ma scomodo perché bisognerà passare l'intero stack alla ricerca della pagina da mettere in cima.



Un'altra approssimazione potrebbe basarsi sui i bit di reference, alla pagina se ne assocerebbero di meno, con un limite inferiore uguale a 1, mentre il massimo il numero di bit necessari a memorizzare il time stamp. Si userà per individuare le pagina da eliminare in caso di necessità, quindi varrà 0 o 1, se 0 verrà eliminata.

Un'ulteriore approssimazione è tramite l'algoritmo dell'**orologio** o algoritmo **secondo chance**. Al posto delle ore si metteranno i numeri di pagina, la prima e la seconda lancetta si muoveranno senza mai modificare l'angolo tra di loro. Come la prima lancetta raggiungerà una pagina, metterà il bit di reference a 0, quindi un unico bit per ogni pagina. La pagina ha tempo tanto quanto l'angolo formato tra le lancette per tornare a 1, in caso contrario significherà che quella pagina non è stata riferita, nel momento in cui servirà una "vittima" da rimuovere, come passerà la seconda lancetta verrà scelta quella pagina.

Una quarta opzione si basa non più sul tempo, ma sulla frequenza in cui una pagina verrà riferita. Qui si avranno due opzioni:

- Algoritmo LFU (**Least Frequently Used**), conto quante volte viene riferita una pagina, nel caso in cui mi serva una vitta sceglierò quella con il conteggio più basso;
- Algoritmo MFU (**Most Frequently Used**), l'esatto opposto, ipotizza che la pagina con meno riferimenti sia stata caricata da poco e che potrebbe venire utilizzata maggiormente in futuro.

Capitolo 6

Allocazione dei frame

Quanti frame assegnare a un processo? Occorre sapere qual è il numero minimo di frame che si possono assegnare per dargli la garanzia che riesca a eseguire un'istruzione alla volta. Quanti frame richiede un'istruzione? Potrebbe richiederne uno come anche due nel caso in cui l'istruzione sia a cavallo di due pagine, a questi bisogna aggiungere gli eventuali operandi che in base all'architettura se fossero tre, potrebbero essere anche loro su due pagine diverse arrivando a un totale possibile di 8 frame necessari. Il minimo è dato dal massimo numero di indirizzi specificabili da un'istruzione, che cambia da architettura ad architettura.

Dal minimo si può ovviamente salire, ma di quanto? Ci può essere anche qui un'allocazione **fissa** e un'allocazione **variabile**.

Allocazione fissa: ogni processo riceve un numero di frame definito, non per forza uguale per tutti, ma che non varia una volta definito;

Allocazione variabile: il numero di frame di un processo può variare durante la sua esecuzione, è decisamente più difficile da gestire;

6.1 Allocazione fissa

Si divide in due ulteriori categorie:

In parti uguali: i frame vengono assegnati in maniera equa a ogni processo, anche se un processo ne avrebbe bisogno di un numero maggiore;

Proporzionale: in base alla grandezza in frame dei processi richiedenti i frame disponibili vengono assegnati in maniera proporzionale;

Un contro della proporzionale è che un processo che richiede molti frame in realtà poi all'interno a righe di codice che gli portano a dover usare molti frame in meno rispetto a quella che sembrava inizialmente.

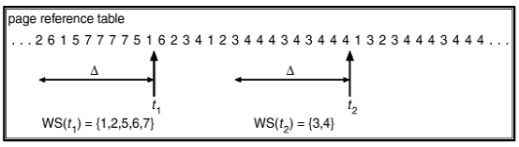
6.2 Allocazione variabile

Una soluzione migliore, allocare a runtime è decisamente meglio, ma è molto difficile da gestire. Ci sono due possibili soluzioni:

- tramite calcolo del **working set**;
- tramite calcolo del PFF (**Page Fault Frequency**);

6.2.1 Working set

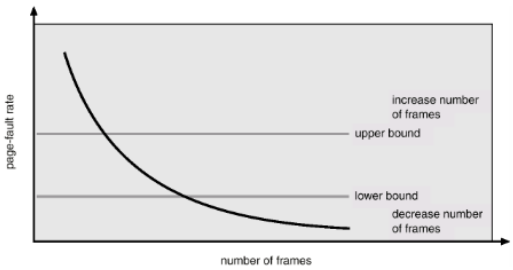
Segue il principio di località, i frame verranno assegnati in base alla località. Come lo misuro?



Periodicamento, al tempo t_1, t_2, etc..., si andrà a guardare a quante pagine ci si è riferiti negli istanti di tempo ovvero Δ, all'istante t_1 il working set è di 5 pagine, ma all'istante successivo come si può vedere calerà a 2 pagine. In base a ciò vengono assegnati i frame. Sarà fondamentale scegliere il parametro Δ, troppo piccolo diventa poco significativo, troppo grande potrebbe essere a cavallo di due località.

Cosa succede se la somma dei working set richiede un numero di frame maggiori di quelli a disposizione? Si verifica il fenomeno del **thrashing**, il SO si accorge che l'uso della CPU è sceso e ci sono molti processi in coda per fare accesso al disco e fare swap in e swap out. Una cattiva gestione porterebbe a fare entrare pronti altri processi, che avrebbero necessità di memoria e di conseguenza tale memoria verrebbe rimossa a quelli già presenti, i quali vedranno i pochi frame a loro disposizione calare ulteriormente e di conseguenza richiederebbero ulteriori swap in e swap out e ciclicamente nuovi processi che entrano e ulteriori frame rimossi fino a un blocco totale.

6.2.2 Page Fault Frequency



L'obiettivo è di mantenere tutti i processi nella fascia intermedia tra upper bound e lower bound, il SO farà scelte per fare in modo che se ci sono processi sotto la soglia lower bound, gli toglierà dei frame perché il numero di page fault è molto basso, quindi potrà assegnare frame ai processi che superano la soglia di upper bound.

Capitolo 7

Memoria secondaria

Necessaria per implementare la memoria virtuale e per memorizzare a lungo termine dati e programmi necessari. Diventa rilevante il concetto di tempo di accesso al disco, dipende da tre parametri:

- **Seek Time**, tempo di spostamento della testina, meccanico, parte più lenta;
- **Latency Time**, tempo di attesa che il disco sottostante porti il settore di interesse sotto la testina, il disco ruota continuamente;
- **Transfer Time**, tempo necessario per spostare i dati da disco a controllore o viceversa.

Essendo il Seek Time il momento che fa perdere pià tempo in assoluto, conviene muoversi a cilindri lungo i dischi, così da non dover spostare la testina, ma bisogna anche considerare che non si riesce a memorizzare un file tutto contiguo. Inolte poichè il disco è uno, ma i processi che leggono e scrivono sono tanti, il SO riceverà molte richieste di lettura e scrittura su zone diverse del disco.

Queste problematiche svaniscono tramite l'uso di dispositivi a stato solido, perché non ci sono più componenti meccaniche ma prettamente elettroniche.

7.1 Scheduling degli accessi a disco

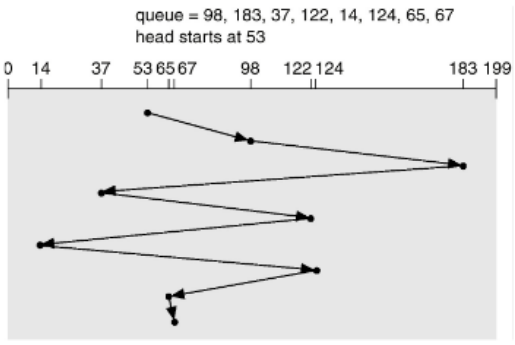
Il disco è diviso in piatti concentrici, suddiviso in tracce e a loro volta suddivise in settori della stessa dimensione. Dal punto di vista logico come un vettore di elementi, un array di blocchi ordinati, tutti della stessa dimensione. Quando avviene un accesso al disco, ci sarà una coda di richieste con

indirizzi e blocchi che indicano dove devono essere fatti e annessa quantità di dati da leggere o scrivere.

Il SO ragiona in termini di costi ed efficacia per decidere riguardo gli accessi. Esempio:

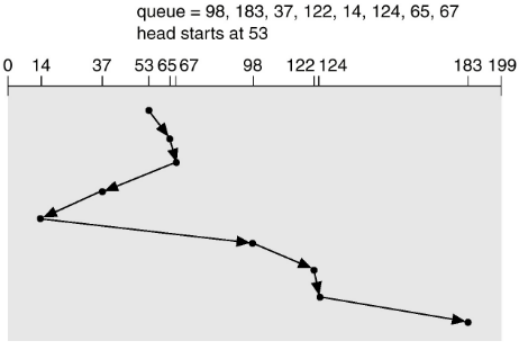
- disco con 200 intervalli di valori ammissibili [0, 199];
- la testina di lettura e scrittua si trova sulla testina 53;
- le richieste di accesso sono per i valori: 98, 183, 37, 122, 14, 124, 65 e 67;

Applicando il meccanismo **FIFO**, la testina si sposterà a zig zag per eseguire tutti gli accessi.



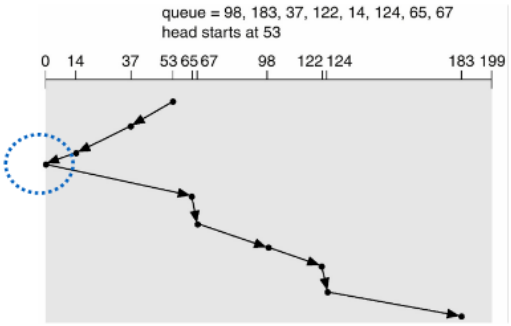
Lo spostamento totale finale sarà di 640 tracce.

Una prima soluzione sarebbe quella di spostarsi verso la zona più vicina a quella in cui ci si trova, implementando l'algoritmo **SSTF** (Shortest Seek Time First), riduce lo spostamento della testina rispetto alla posizione attuale. Questo algoritmo è decisamente ottimo rispetto al **FIFO**, ma non il migliore.



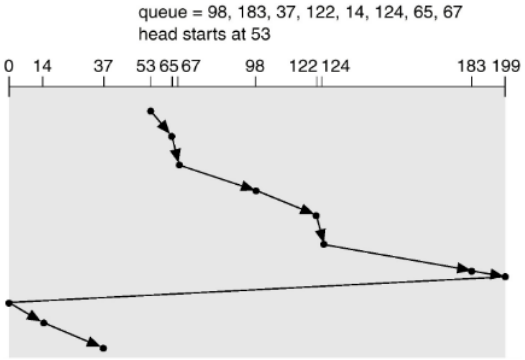
Se l'elenco di richieste accesso in lettura e scrittura fosse stabile, andrebbe bene, ma durante l'accesso al primo blocco molto probabilmente arriverebbero altre richieste di accesso, quindi la coda di richieste è in continup aggiornamento.

Una possibile alternativa a questa è l'implementazione dell'algoritmo dell'ascensore o **SCAN**. Si sposta da un'estremità all'altra servendo tutto quello che si trova in mezzo.



Tale algoritmo si può migliorare, per esempio lo spostamento nella traccia 0 non ha senso siccome non sono avvenute richieste minore della 14.

L'algoritmo **C-SCAN** o algoritmo dello spalatore di neve è simile allo SCAN, ma la differenza sta che una volta raggiunta l'ultima traccia, ritorna alle tracce iniziali che molto probabilmente saranno state richieste dai processi. Il tutto ovviamente senza servire le eventuali richieste intermedie.



SCAN e **C-SCAN** evitano la starvation? No, perché le richieste agli estremi rimangono sempre le ultime.

Una soluzione migliore che implementa il meglio degli altri è una variante dello SCAN, ovvero **N-step SCAN**, la coda delle richieste viene divisa in partizioni grandi N, un valore parametrizzato, in questo modo saturate le richieste la lista rimane congelata e le servo una dopo l'altra senza fermarmi a metà strada, le richieste che arrivano nel mentre verranno salvate in

una coda diversa. Se il valore di N diventasse troppo grande si regredirebbe all'algoritmo SCAN, mentre un con valore uguale a 1 si tornerebbe all' algoritmo FIFO. Con un valore intermedio le cose verrebbero gestite decisamente meglio.

La variante più semplicemte e **FSCAN** (Fast SCAN) che ha solo due code, si riempie una prima coda e nel mentre che viene servita si riempie la seconda, alternandosi tra le due.

L'algoritmo **LIFO** (Last In First Out), in alcuni casi avrebbe senso schedulare gli accessi nell'ordine opposto di arrivo, per dare preferenza a richieste con elevata località. Pone un alto riscio di starvation, è molto specifico.

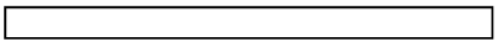
Ci sono vari fattori che influenza quale possa essere l'algoritmo migliore, nel general purpose il caso medio è quello che interessa, mentre in un caso specifico ovviamente varia.

7.2 Formattazione dei dischi

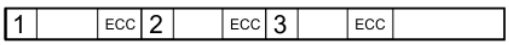
Un disco per essere utilizzare deve essere formattato, esistono due tipi di formattazione:

- basso livello o fisica;
- logica.

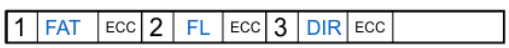
La prima permette di partizionarlo e prepararlo a ricevere i dati e iniziare a essere utilizzato. La seconda prepara le strutture dati che verranno usate dal SO per gestire il disco. Inoltre la formattazione logica va a inserire il programma di Bootstrap per caricare i driver dei dispositivi e lanciare il SO. La formattazione fisica divide in settori, ogni settore avrà uno spazio per i



(a) ***Disco nudo, preformattazione fisica e logica***



(b) ***Disco post formattazione fisica***



(c) ***Disco post formattazione logica***

dati e uno per la correzione d'errore, permettono di ripristinare o identificare

un certo numero di errori. Se il numero di errori fisici o logici è inferiore al numero di errori che possono essere corretti allora si potrà continuare a leggere anche in presenza di errori.

La formattazione logica andrà a inserire le strutture necessarie per la gestione del file system, capire quali blocchi sono liberi e quali sono occupati.

7.3 Gestione dei blocchi difettosi

Un blocco si può danneggiare per motivi fisici, urti o usura, ma anche dei danni software, come un corruzione del file system.

Esistono dei meccanismi di gestione degli errori online, un blocco che non funziona lo rimappo in un blocco di scorta. Il dato non verrà recuperato, però evita di riutilizzare il blocco difettoso. I processi continueranno a chiedere accessi al blocco difettoso e il SO saprà che dovrà rigirare tutto al blocco rimappato. Lo svantaggio è che andrà a inficiare i meccanismi di scheduling visti sopra, loro saranno convinti di andare al blocco con un certo numero, che essendo corrotto sarà reindirizzato a un altro blocco che magari si trova molto distante causando un aumento del **Seek Time**.

7.4 Gestione dello spazio di swap

Si tratta di un area in cui vengono salvate pagine e frame, non file e cartelle, ma viene gestita alla stessa maniera. Le opzioni sono:

- definisco l'area non come il file system definendo delle primitive, system call diverse per accedervi;
- definisco l'area di memoria virtuale ma la tratto come fosse il file system.

Così avremo un sistema meno efficiente perché si avrà a che fare con struttre dati inutili per quello che viene salvato lì, ma risparmio il dover definire delle system call apposite. Viceversa con una partizione separata occorerranno le system call ad hoc, ma le prestazioni aumenteranno notevolmente.

Capitolo 8

File System

Consiste di una struttura dati che ci permette di memorizzare e organizzare dati e programmi nel modo migliore possibile per poterli trovare al momento del bisogno. La struttura è simile a quella di un albero, i rami sono le cartelle mentre i file sono le foglie.

8.1 Interfaccia

Cos'è un file? Un file è un'idea astratta di uno spazio omogeneo in cui memorizzo dati. Idea astratta perché non si sa fisicamente come sia fatto o dove si trovi. Un file è uno spazio di indirizzamento logico e contiguo. L'unica cosa che lo identifica è il nome. Il file può essere di tipi diversi, ha una struttura omogenea rispetto all'utilizzo che se ne farà.

Cosa contiene un file? Oltre ai dati che si vogliono memorizzare contiene una parte di attributi, una sorta di Process Control Block. Gi attributi non sono memorizzati dentro al file stesso, ma vengono memorizzate all'interno della cartella. Tali attributi sono:

- il nome;
- il tipo, eseguibile, non eseguibile,...;
- la posizione, dove si trova in memoria;
- la dimensione;
- la protezione, chi può farci e che cosa;
- tempo di creazione;

- tempo di ultimo accesso;
- tempo di ultima modifica;
- identità del proprietario.

Questi attributi vengono memorizzati in una struttura `inode`, che ha bisogno di spazio e viene memorizzata all'interno della cartella.

8.1.1 Operazioni su file

Creazione : il SO deve mettere a disposizione delle system call per creare file, che vanno a cercare dello spazio a disposizione per contenere il file.

Scrittura e lettura : necessità di sapere la posizione in cui andare a scrivere, analogamente la lettura.

Cancellazione : semplicemente rimuove gli attributi dalla directory, ma i dati rimangono in memoria, cambia soltanto che non si sa più come raggiungerli. I dati spariranno effettivamente con una sovrascrizione che non si può sapere quando avverrà effettivamente.

Troncamento : lascia il file con gli attributi inalterati, ma azzera la dimensione, ovviamente così il contenuto non viene cancellato.

Apertura : impone la ricerca del file all'interno della struttura della cartella su disco, la copia in memoria del contenuto e la creazione di due tabelle all'interno delle strutture dati del SO. Una per ogni file aperto e una che lega il file al processo che l'ha aperto. Finchè il file è in uso, una sua copia viene caricato in RAM, così da velocizzare lettura e scrittura, la chiusura del file porta al salvataggio permanente su disco.

La rinominazione **non** è un'operazione su file, avviene sulla struttua della cartella, perché cambiare il nome di un file consiste nel cambiare la sua posizione, il suo pathname.

8.1.2 Metodi di accesso

Un compilatore avrà necessità di accedere al file in modo sequenziale, mentre un gestore di base dati dovrà accedervi randomicamente

Sequenziale

Un processo che vuole accedere a un file in modo sequenziale avrà bisogno di operazioni come:

- `read next`;
- `write next`;
- `reset`.

In questa modalità di accesso l'operazione di `rewrite` non è ammissibile, perché si creerebbe inconsistenza rispetto al contenuto precedente, condizione necessaria per un accesso sequenziale.

Diretto

Non interessa partire dall'inizio del file e leggerlo in sequenza, quindi ogni elemento del file sarà leggibile e scrivibile separatamente dagli altri, il file sarà strutturato a blocchi. Le operazioni permesse saranno:

- `read n`;
- `write n`;
- `position n`;
- `read next`;
- `write next`;
- `rewrite n`.

Non c'è rischio di creare inconsistenza perché ogni file è diviso dagli altri(blocchi).

8.2 Struttura delle directory

Il file system è organizzato a volumi o partizioni, è una vista logica, quindi indipendente dal numero dei dispositivi fisici. Esempio: un disco partizionato in due dischi, quindi dal punto di vista logico se ne vedranno due. Vale anche la casistica opposta, due dischi visti dal sistema come un unico disco solo.

All'interno delle partizioni, troviamo una collezione di nodi contenenti informazioni sui file. Le directory sono elementi che puntano ai file. All'interno della directory verranno memorizzati gli attributi dei file visti sopra.

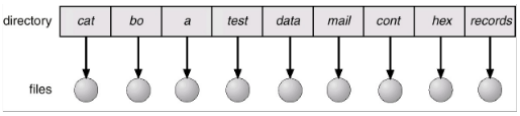
Operazioni su directory Una volta creata la directory con al suo interno gli `inode` dei file che vi risiedono si possono compiere le seguenti operazioni.

- aggiungere file;
- cancellare file;
- visualizzarne il contenuto;
- rinominare un file;
- cercare un file;
- attraversare il file system.

Obiettivi Gli obiettivi per chi vuole implementare la gestione delle directory sono un'elevata efficienza per un accesso rapido al file, una nomenclatura che permette agli utenti di avere un modo conveniente di dare un nome ai file e ricordarsi come si chiamano, infine il raggruppamento, ovvero poter classificare logicamente i file per criterio(tipo, protezione, etc...).

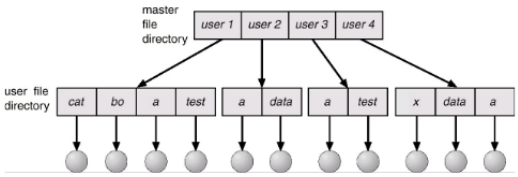
8.2.1 Directory a un livello

Unica per tutti gli utenti, la nomenclatura era un problema perché ogni file doveva avere un nome differente rispetto agli altri, considerando che la lunghezza massima dei file al tempo era di 8 caratteri.



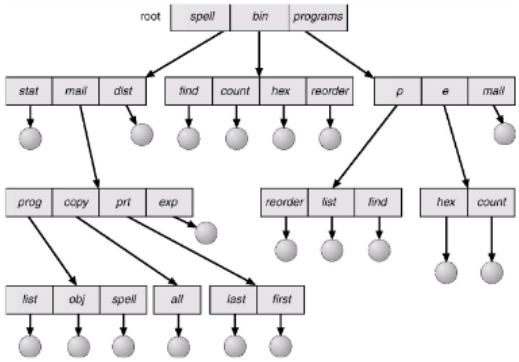
8.2.2 Directory a due livelli

Una prima evoluzione è stata la separazione delle cartelle a livello utente, è nato un primo concetto di cartella `home` per ogni utente e al suo interno ogni utente metteva i loro file. La problematica sulla nomenclatura si spostava, ora non era più condivisa tra tutti gli utenti, ma era per il singolo che all'interno della sua cartella non poteva crearne altre e di conseguenza era limitato con la rinomina dei file. Nasce anche il problema di decidere dove posizionare i programmi di sistema, se replicarli all'interno di ogni singola cartella o no, da cui nasce a sua volta il concetto di variabile di ambiente, che permettono di specificare cose come il PATH. Sulle macchine Linux tale variabile d'ambiente contiene cartelle e sottocartelle con programmi eseguibili che permettono di essere lanciati da terminale.



8.2.3 Directory ad albero

Permettono una ricerca efficiente, attraversando l'albero aggiungendo la possibilità di raggruppare sotto cartelle, nasce l'idea della directory corrente o directory '.', di comandi `cd` o `pwd`. Nasce il concetto di percorso assoluto e relativo ovvero relativo alla posizione attuale. Uno '/' all'inizio del percorso indicherà un percorso assoluto, ovvero dalla radice, mentre ometterlo indicherà un percorso relativo ovvero partendo dalla locazione attuale in cui ci si trova.



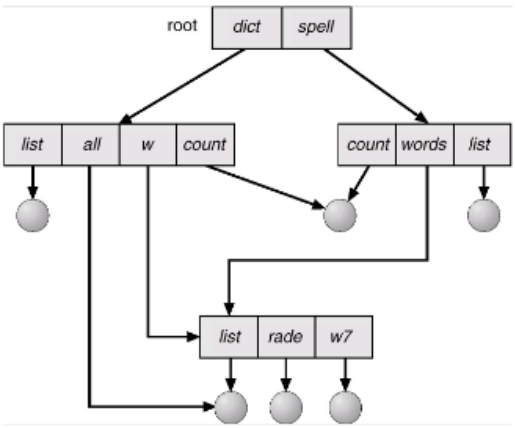
8.2.4 Directory a grafo aciclico

Ci saranno dei rami che si ricongiungono, facilitando la condivisione tra utenti. Da questo nasce il concetto di link, simbolici e hard link.

Link simbolico: un altro file con all'interno il pathname del file a cui si collega;

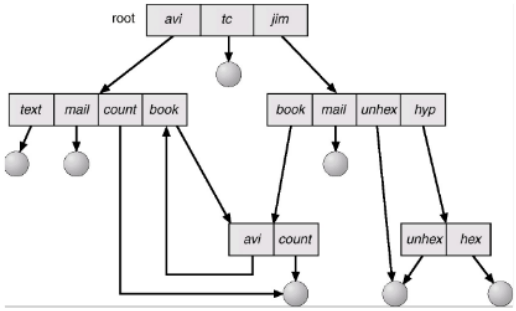
Hard link: il file è uno solo, al quale fa riferimo un `inode1` e quello del link `inode2`, un unico file con due strutture di attributi diversi.

Cancellando un `inode` a un hard link il file rimarrà, fintanto che avrà un collegamento con un altro `inode`. Se si cancella un link simbolico, il file originale rimarrà, ma se si cancellasse il file a cui il link simbolico punta, avremo un errore tentando di aprirlo siccome non punterà più a nulla. Una modifica eseguita a un file tramite hard link sarà visibile pure tramite l'altro link.



8.2.5 Directory a grafo generico

Con questa struttura ci sono possono essere dei cicli, ciò potrà causare dei problemi eventualmente con la ricerca. In tal caso bisognerà usare degl algoritmi di controllo dei cicli per verificare se nel link creato sia nato un ciclo o meno.



8.2.6 Mount di file system

Meccanismo che permette di collegare un punto del file system corrente a un altro file system, in modo da poterlo raggiungere e attraversare come se facesse parte del nostro file system, esempio: quando si collega una chiavetta USB.

8.3 Protezione

Potendo creare file system grandi a piacere con cartelle a grafo, occorre pensare anche ai livelli di protezione per impedire che un utente faccia cose indesiderate sul file system. Esistono varie possibilità per permettere al

proprietario del file di fare qualsiasi cosa. Si parte decidendo che tipo di operazioni controllare, dando dei permessi:

- lettura;
- scrittura;
- esecuzione;
- append;
- cancellazione;
- ...

Un primo livello di protezione è un inserimento in ogni directory di un elenco di chi può fare cosa. Molto complicato da gestire ed estremamente lungo. Un meccanismo più efficiente e meno fine è il raggruppamento in 3 classi usato dai sistemi Unix. Ogni file apparterrà all'utente che l'ha creato, quell'utente appartiene a un gruppo di utenti che condividono un certo insieme di privilegi. Gli utenti sono divisi in 3 categorie:

- proprietario;
- il gruppo di utenti a cui appartiene il proprietario;
- tutti gli altri.

Ogni utente può appartenere a più gruppi contemporaneamente. I permessi per ogni classe verranno identificati tramite i caratteri `r`, `w` e `x`. Per un cartella il permesso di execute `x`, corrisponde alla possibilità di poterla attraversare tramite il comando `cd`.