

Sistemi Operativi

Secondo semestre

Giovanni Tosini

Indice

1	Monitor	5
1.1	Problematiche	6
2	Deadlock	7
2.1	Gestione Deadlock	8
3	Gestione della memoria	15
3.1	Tempi di compilazione	17
3.2	Linking	17
3.3	Loading	18
3.4	Spazio di indirizzamento logico	18
4	Schemi di gestione della memoria	19
4.1	Allocazione contigua	19
4.1.1	Partizioni fisse	19
4.1.2	Partizioni variabili	21
4.1.3	Tecnica del Buddy System	21
4.2	Paginazione	21
4.2.1	Tabella delle pagine invertita	23
4.2.2	Paginazione multilivello	24
4.3	Segmentazione	24
4.4	Paginazione vs Segmentazione	25

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:

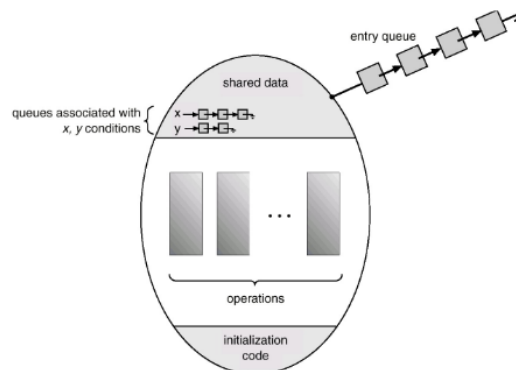
- variabili(sono private)
- metodi(solo loro possono utilizzare le variabili definite all'interno del Monitor)
- costruttore

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

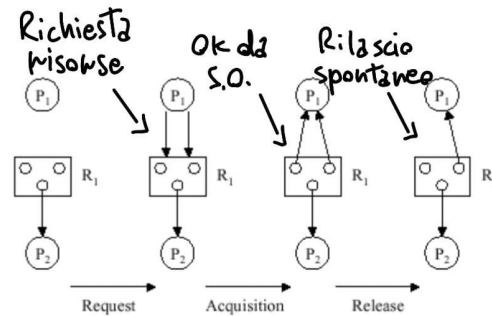
Deadlock

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)



- $V = \{\{P_1, P_2\}, \{R_1\}\}$
- $E_{iniziale} = \{\{R_1, P_2\}\}$ $E_{finale} = \{\{R_1, P_1\}, \{R_1, P_2\}\}$

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

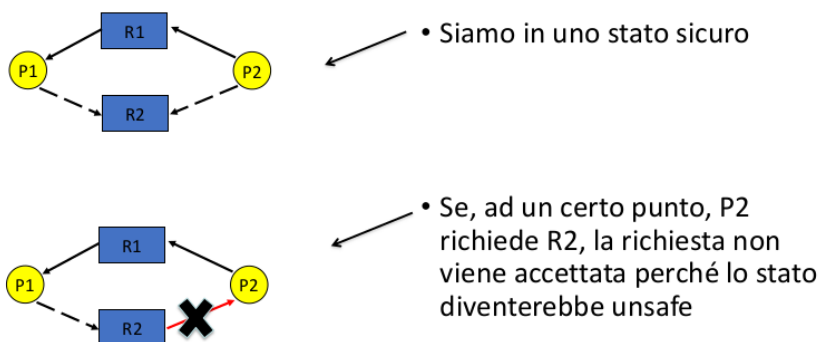
Quindi saremmo in uno stato SAFE.

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'algoritmo 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 Ri 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

Algoritmo di verifica dello stato

 $O(m \cdot n^2)$

```

boolean state_safe() {
    int work[m] = available[];
    boolean finish[n] = (FALSE, ..., FALSE);
    int i;
    while (finish != (TRUE, ..., TRUE)) {
        /* cerca Pi che NON abbia terminato e che possa
           completare con le risorse disponibili in work */
        for (i=0; (i<n)&&(finish[i] || (need[i][ ]>work[ ])); i++);
        if (i==n)
            return FALSE; /* non c'è → unsafe */
        else {
            work[ ] = work[ ] + alloc[i][ ];
            finish[i] = TRUE;
        }
    }
    return TRUE;
}

```

Ho già sottratto le richieste di P_i!

Sono arrivato in fondo, senza trovare finish[i]=FALSE e need[i][] <= work[]

L'ordine non ha importanza. Se + processi possono eseguire, ne posso scegliere uno a caso, gli altri eseguiranno dopo, visto che le risorse possono solo aumentare

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 Ri disponibili
int alloc[n][m]; //matrice allocazione corrente
int req_vec[n][m]; //matrice di richiesta,
                  //stesse dimensioni della max,
                  //è una max istantanea

```

$O(m \cdot n^2)$

Algoritmo di detection

```

int work[m] = available[m];
bool finish[] = (FALSE,...,FALSE), found = TRUE;
while (found) {
    found = FALSE;
    for (i=0; i<n && !found; i++) {
        /* cerca un Pi con richiesta soddisfacibile */
        if (!finish[i] && req_vec[i][] <= work[]){
            /* assume ottimisticamente che Pi esegua fino al termine
            e che quindi restituisca le risorse */
            work[] = work[] + alloc[i][];
            finish[i]=TRUE;
            found=TRUE;
        }
    }
}
} /* se finish[i]=FALSE per un qualsiasi i, Pi è in deadlock */

```

Se non è così
il possibile deadlock
verrà evidenziato
alla prossima
esecuzione dell'algoritmo

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 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
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;
```

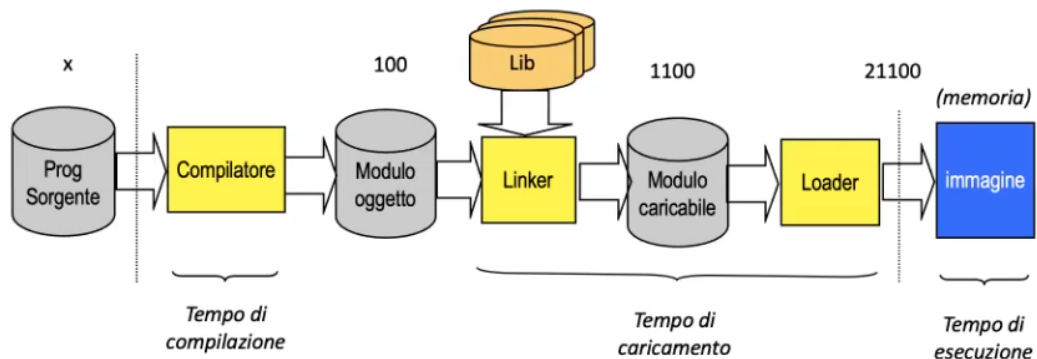
non è altro che un riferimento a un indirizzo di memoria, quando il programma verrà eseguito, quell'indirizzo **simbolico** verrà tradotto in un indirizzo **fisico**.

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 ricaricabile 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 quella 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 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'indirizzo 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.

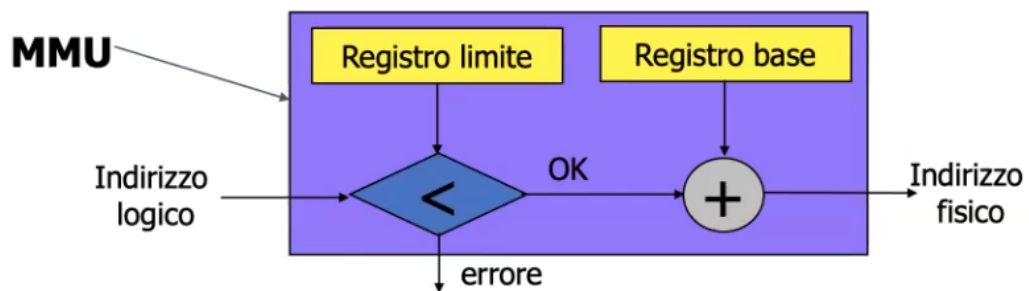
Nell'alternativa con un'unica coda, la gestione sarà più complicata, perché un processo troppo grande per stare in partizioni troppo piccole rimarrebbe

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 seguente schema:

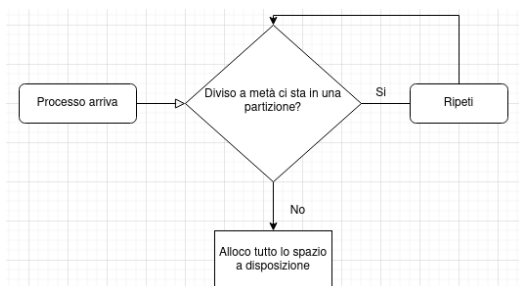


Figura 4.1: Flowchart Buddy System

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} potresti 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 specifico, 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: $\text{offset} + \text{base} = 649$.
- 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

Paginazione:

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

Svantaggi: • non mantiene l'omogeneità;

Segmentazione:

Vantaggi: • omogeneità;

- associazione di protezione/condivisione ai segmenti;

Svantaggi: • richiede degli algoritmi di allocazione dinamica dello spazio, essendo i segmenti variabili;

- reintroduce la problematica della frammentazione esterna;

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