Appunti di Sistemi Operativi

Enzo Mumolo

e-mail address :mumolo@units.it
web address :www.units.it/mumolo

Indice

Capitolo 1

La Gestione della Memoria

1.1 Introduzione

La gestione della memoria riguarda gli algoritmi e le tecniche usate dai Sistemi Operativi per risolvere le richieste di allocazione della memoria da parte dei processi. La richiesta piú importante riguarda l'allocazione in memoria dei processi stessi. Il problema della gestione della memoria puó essere descritta con la seguente Fig. 1.1:

Le richieste di memoria sono messe in coda in attesa; il Memory Manager decide quindi come soddisfare le richieste.

Oltre a queste richieste statiche, i processi richiedono tipicamente anche quantitá di memoria dinamicamente durante la loro esecuzione. La memoria richiesta dinamicamente viene allocata e rilasciate in zone di memoria particolari (Heap) e il rilascio viene gestito con algoritmi di 'Garbage Collection'. In questo capitolo non si descriveranno gli algoritmo per la gestione della memoria dinamica.

1.1.1 Caratteristiche generali di un gestore di memoria

Ogni gestore di memoria puó essere descritto con le seguenti caratteristiche generali:

Tempo di gestione I gestori di memoria richiedono un certo tempo di calcolo. É ovviamente importante che tale tempo sia piú piccolo possibile, perché la gestione della memoria é realizzata dal Sistema Operativo e il tempo richiesto si traduce quindi nel sovraccarico del SO.

Allocazione e rilascio Il tempo di gestione si divide tra tempo di allocazione di memoria e tempo di rilascio della memoria: in generale il tempo richiesto dagli algoritmi di allocazione non é uguale dal tempo richiesto per rilasciare. Inoltre bisogna tener conto della dimensione delle strutture dati utilizzate per richiesta/rlascio.

Strutture dati Ogni gestore della memoria richiede delle strutture dati per il suo funzionamento. Possono essere delle liste concatenate o delle tabelle o delle bitmap; in ogni caso é bene che le loro dimensione siano piccole.

Frammentazione Lo scopo del gestore di memoria é sí di risolvere le richieste di memoria da parte dei processi, ma sfruttando al massimo la memoria esistente, cercando quindi di lasciare meno spazio inutilizzato possibile. Tutti i gestori di memoria, tuttavia, lasciano un pó di memoria inutilizzata, nel senso che é impossibile sfruttare tutta la memoria esistente. Questo fatto viene descritto con la *percentuale di frammentazione* **f**, definita come

$$f = \frac{memoria\ inutilizzata}{memoria\ totale} \cdot 100$$

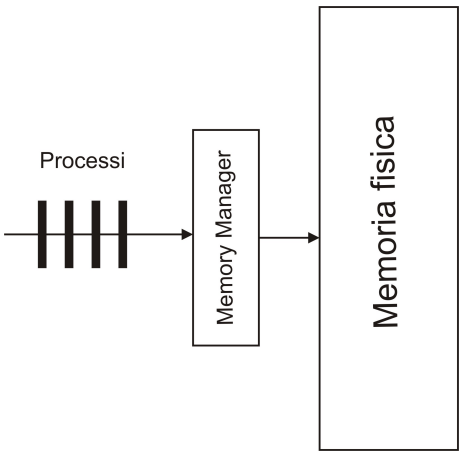

Figura 1.1: Il meccanismo della richiesta di memoria da parte dei processi

Naturalmente questa misura é relativa alla memoria utilizzata. Se si lavora con una partizione di dimensione fissa, ad esempio, la frammentazione si riferisce all'uso di memoria all'interno della singola partizione mentre se si lavora con partizioni variabili la frammentazione si riferisce all'intera memoria.

Indirizzi virtuali e fisici Un processo durante la sua esecuzione fá riferimento a indirizzi virtuali di memoria, che partono dal valore zero al massimo indirizzo riferito. Lo spazio di memoria che vá da zero al massimo valore riferito si chiama **spazio di indirizzamento del processo**. Gli indirizzi utilizzati in un programma sono rappresentati nella seguente Fig. 1.2:

Si ricorda che gli indirizzi relativi fanno riferimento alla differenza tra l'indirizzo attuale e l'indirizzo di destinazione e pertanto sono indipendenti dagli indirizzi assoluti cioé dalla posizione dove viene caricato il programma (**rilocabilitá**). Una volta che il codice rilocabile viene unito al codice di lireria, si ottiene il **codice eseguibile** che deve essere anch'esso rilocabile. Il codice eseguibile cosí ottenuto ha anche un'altra caratteristica: gli indirizzi partono dal valore 0 e si chiamano **indirizzi virtuali**. Per poter eseguire il codice, in sostanza, bisogna convertire gli indirizzi virtuali in indirizzi fisici.

Livelli di memoria I dispositivi di memorizzazione sono molteplici, e vanno per esempio dai Compact Disk ai dischi fissi ai dischi flessibili alla memoria centrale alla memoria contenuta entro le CPU, i registri etc. I sistemi di memorizzazione citati hanno delle capacitá di memoria via via decrescenti. Come regola generale i tempi di accesso nei sistemi di memorizzazione sono tanto piú bassi quanto meno capiente é il sistema di memorizzazione. Si va' cosí dalla capacitá tipica di un moderno disco fisso che puó essere di 20 GByte alla capacitá tipica di una moderna memoria centrale che puó essere intorno ai 512 MByte alla capacitá della memoria contenuta in una CPU che puó essere intorno ad 1Kbyte. Per contro il tempo d'accesso ad un disco fisso puó essere intorno a 10 ms, il tempo d'accesso alla memoria centrale puó essere intorno a 50 ns e il tempo d'accesso alla memoria contenuta in una CPU intorno a 1 ns. D'altra parte la velocitá di trasferimento puó essere intorno a qualche decna di Mbyte al secondo per il disco fisso, a qualche centinaio di Mbyte al secondo per la memoria a qualche migliaio per la memoria contenuta nella CPU. Si tenga conto che 1 ns puó essere il tempo di esecuzione di una istruzione nella CPU, considerando le attuali frequenze di clock. Se la CPU dovesse ad ogni istruzione caricare l'istruzione direttamente dalla memoria centrale passerebbe la maggior parte del tempo ad attendere il completamento dell'accesso ai dati. La soluzione attuata normalmente è allora quella di interporre tra la memoria e la CPU una memoria molto

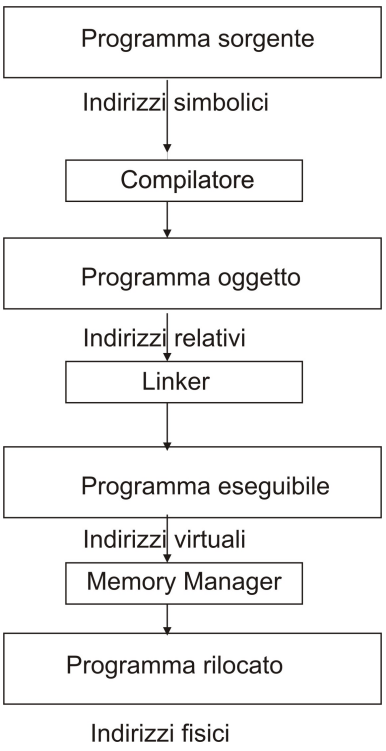

Figura 1.2: I vari tipi di indirizzi coinvolti nella esecuzione di un programma

più piccola ma allo stesso tempo molto veloce, in grado cosí di lavorare alla stessa velocità della CPU. In questa memoria-tampone, chiamata **memoria cache**, verranno memorizzate di volta in volta le informazioni utilizzate piú recentemente dalla CPU. Quando la CPU richiede un dato, prima di accedere alla memoria centrale, il dato viene cercato all'interno della cache. La cache é come noto una memoria indirizzata per contenuto. Se il dato è presente (evento di **hit**) allora puó essere fornito velocemente alla CPU. Se il dato non è presente (evento di **miss**) allora deve essere caricato dalla memoria centrale e messo in una qualche posizione della cache, eventualmente eliminando un altro dato (si ricordi che una cache è notevolmente più piccola dello spazio disponibile nella memoria centrale). L'interposizione di una cache tra CPU e memoria centrale in generale peggiora in qualche misura l'accesso alla memoria centrale rispetto al singolo accesso perché oltre al tempo di accesso vero e proprio occorre aggiungere il tempo di verifica della condizione di miss. Questo peggioramento però è largamente compensato dalla accelerazione che si ottiene nel caso si verifichi una condizione di hit. Il tempo di accesso effettivo é allora dato dalla relazione:

$$T_a = T_{cache} \cdot h + (T_{cache} + T_m)(1 - h)$$

dove T_a é il tempo effettivo di accesso, T_m il tempo d'accesso alla memoria centrale e h rappresenta la probabilitá di trovare il dato direttamente nella memoria cache. La relazione appena scritta puó essere cosí semplificata:

$$T_a = T_{cache} + T_m - h \cdot T_m$$

dalla quale risulta che la derivata rispetto ad h é pari a $-T_m$. Questo vuol dire che il tempo di accesso descresce con l'aumentare di h, non solo, ma anche che basta un piccolo innalzamento

di h per diminuire il tempo effettivo di accesso. Quindi é importante mantenere elevato il valore della probabilitá di hit. Ad esempi supponiamo che la memoria centrale abbia un tempo d'accesso di 50 ns e la memoria cache di 1 ns. In queste condizioni, se la probabilitá di hit fosse del 78% il tempo effettivio sarebbe di 12 ns, mentre se lo hit rate aumentasse di 12 punti percentuali, il tempo effettivo diventerebbe pari a 6 ns, con una diminuzione di 50 punti percentuali!.

Quanto detto per la memoria cache vale per tutti gli altri tipi di memoria: tra ogni tipo di memoria e l'altro, il primo caratterizzato da un tempo di accesso molto piú elevato del secondo perché piú capiente ci puó essere una zona di memoria veloce che funziona come memoria tampone. Questa memoria viene acceduta prima di accedere alla memoria lenta e capiente: se le informazioni cercate sono contenute nella memoria veloce si risparmia l'accesso alla memoria lenta. Cosí c'é una memoria cache tra la CPU e la memoria centrale, un'altra cache tra la memoria virtuale e la memoria centrale (TLB), una tra il disco e la CPU (buffer cache) e un'altra tra l'interfaccia del disk drive e il dispositivo meccanico del disco (disk cache).

Localitá Quanto detto funziona grazie al **principio di localitá** che fá s'ıche la probabilitá di hit sia alta se in cache si mantengono le informazioni richieste piú recentemente. Il vantaggio in termini di velocità ottenuto usando le cache si basa teoricamente sul principio di **localitá temporale**: **una CPU tende a richiedere dati che sono stati usati di recente**. Se i dati usati di recente vengono mantenuti nella cache allora il tempo per accedere a questi dati (quelli maggiormente usati dalla CPU) sará notevolmente minore del tempo necessario per recuperare i dati usati raramente presenti nella memoria centrale. Attraverso cache organizzate a blocchi e trasferimenti paralleli tra memoria centrale e cache è possibile migliorare le prestazioni del sistema sfruttando il principio di **localitá spaziale**: **una CPU tende a richiedere dati contenuti in un indirizzo vicino a quelli usati di recente**. Nel caso si verifichi una condizione di miss invece di caricare un singolo dato nella cache viene caricato un blocco di più dati spazialmente vicini, ad esempio la parola richiesta e un certo numero di parole successive (organizzando opportunamente l'architettura delle memoria centrale e delle linee di comunicazione con la cache è possibile trasferire un blocco di dati in un tempo paragonabile a quello necessario per trasferire una sola parola). I successivi accessi alla cache avranno in questo caso più probabilità di trovare il dato richiesto aumentando così la frequenza di hit. Inoltre, chiamando $k = \frac{T_m}{T_{cache}}$ le relazioni viste nel punto precedente possono essere semplicemente sviluppate in questa forma:

$$T_a = T_{cache} + T_m(1-h) = T_{cache}(1 + k(1-h)) = T_m(1/k + (1-h))$$

In altri termini, il valore di k é alto allora $T_a = kT_{cache}(1-h) = T_m(1-h)$, cioé il tempo di accesso é dominato dalla probabilitá di **miss** (1-h).

Caricamento statico e dinamico Il caricamento del programma eseguibile in memoria puó essere fatto o staticamente o dinamicamente.

Il primo corrisponde al fatto che le librerie vengono caricate **saticamente** assieme al resto del codice. Il codice eseguibile ha dunque una dimensione piuttosto elevata.

Il secondo metodo é quello di caricare le librerie **dinamicamente** durante l'esecuzione del programma. In questo caso il codice eseguibile ha dimensioni piú ridotte e le librerie vengono caricate nell'ambiente dove esegue il programma. Il vantaggio é che le librerie possono essere cambiate o aggiornate senza dover ripetere la compilazione e il caricamento dell'intero programma. Lo svantaggio é una certa lentezza della esecuzione perché le librerie vengono caricate dal disco. Il programma che usa una libreria dinamica deve fare riferimento ad una informazione che dice dove sono memorizzate le librerie. Questa sezione é chiamata STUB.

In ogni modo il codice deve essere **rilocabile**, cioé puó fuzionare correttamente indipendentemente dalla zona di memoria fisica dove viene caricato.

1.2 Memoria contigua

La gestione della memoria contigua rappresenta il caso piú importante perché caratteristica di tutti i sistemi operativi dedicati, che sono in grande numero. Tutte le applicazioni di un certo pregio, come ad esempio un lettore DVD, un proiettore con collegamento fireware, un disco fisso per calcolatore, un cellulare etc etc, tipicamente sono controllate da un sistema operativo in tempo reale che usa sempre una memoria contigua. Da questo punto di vista la meoria contigua rappresenta il caso piú numericamente importante nella gestione della memoria.

Nella memoria contigua un processo deve essere completamente caricato in memoria per poter essere eseguito, e quindi la memoria fisica deve avere una dimensione maggiore o uguale alla dimensione dei processi. La memoria contigua ha il pregio di avere un comportamento deterministico, nel senso che il tempo d'accesso é semplicemente quello della memoria e non intervengono eventi asincroni che introducono variazioni di durata temporale. Per questo motivo nei sistemi dedicati non viene tipicamente utilizzata la memoria cache, che puó introdurre dei ritardi asincroni per la gestione del miss.

I fattori piú importanti della memoria contigua sono: la riduzione della frammentazione per il recupero della memoria fisica, visto che i processi devono essere caricati completamente in memoria, e il tempo di allocazione e di rilascio che deve essere il piú piccolo possibile.

Il primo tipo di gestione della memoria é relativo alle partizini fisse.

1.2.1 Partizioni fisse

In questo caso la memoria viene divisa in partizioni di dimensioni fisse, stabilite a secondo delle caratteristiche dei processi coinvolti. La divisione della memoria in diverse partizioni é ovviamente relativa alla necessitá di avere piú processi contemporaneamente in memoria, cioé alla necessitá di gestire la multiprogrammazione. La modalitá piú semplice di utilizzazione é quella nella quale ci sono diverse code, una per ogni partizione, a seconda della distribuzione dello spazio di indirizzamento richiesto dai singoli processi, come visualizzato in Fig.1.3.

Questo modo di operare é altamente inefficiente. Supponiamo che ci sia una coda con alcuni processi in attesa di allocazione e alcune partizioni libere. I processi non possono peró essere allocati perché non sono nella coda relativa alle partizioni libere.

L'aumento della efficienza si puó ottenere gestendo la memoria divisa in partizioni fisse con una sola coda illustrata in Fig.**??**; i processi vengono cos'ıallocati nella partizione che piú adatta.

Come si puó decidere in che partizione allocare il processo? In generale ci sono almeno tre modi per allocare un processo in una partizione: mettere il processo nella prima partizione che puó contenerlo (o analogamente nella prima che segue l'ultima allocazione), in quella che lascia meno spazio libero o in quella che lascia piú spazio libero. Le tre politiche si chiamano rispettivamente **First Fit** (e **Next Fit**, **Best Fit** e **Worst Fit**. É chiaro che la politica migliore potrebbe essere quella di allocare il processo nella partizione che lascia meno spazio libero. Decidere la partizione sulla base dello spazio libero, peró, richiede tempo.

Un altra considerazione riguarda la protezione della memoria, cioé rilevare le condizioni nelle quali un indirizzo di memoria fisico generato accede ad uno spazio esterno alla partizione. Nella memoria contigua a partizioni fisse il controllo é particolarmente semplice: intanto la trasformazione da indirizzo virtuale a indirizzo fisico viene realizzata sommando all'indirizzo virtuale (che inizia da zero) il valore iniziale della partzione dove viene allocato il processo. Il controllo sullo sforamento della partizione viene semplicemente realizzato confrontando l'indirizzo fisico con il limite della partizione fissa, come si visualizza nella Fig. 1.5, dove α e β sono gli indirizzi virtuali e fisico.

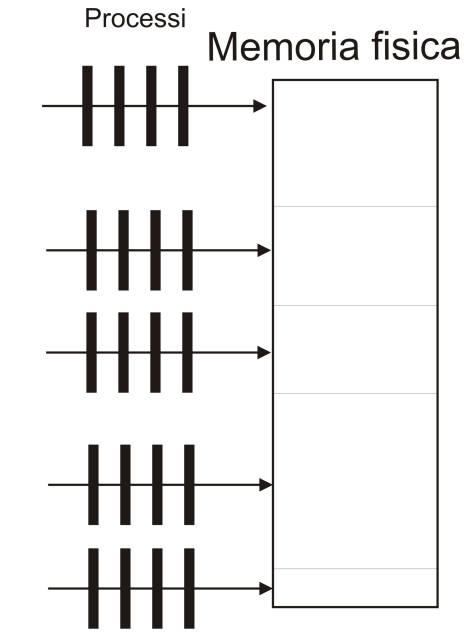

Figura 1.3: Memoria contigua a partizioni fisse con una coda per ogni partizione

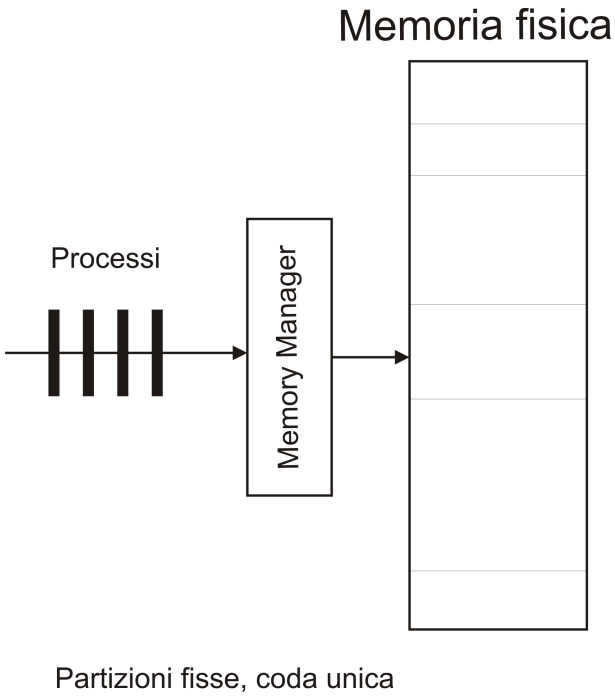

Figura 1.4: Memoria contigua a partizioni fisse con una coda unica

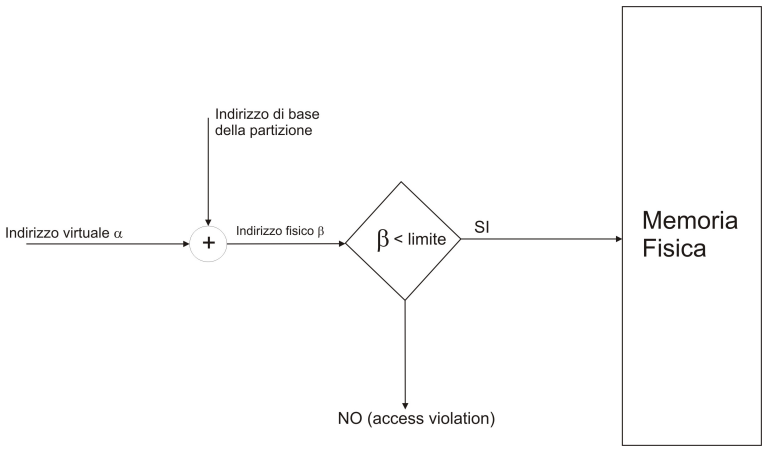

Figura 1.5: Il controllo della violazione accessi realizzato nelle partizioni fisse

In questo caso la **frammentazione é interna** alle partizioni. Visto che le partizioni sono fisse, non ci sono altre possibilitá di abbassare la frammentazione, oltre alli'utilizzo della allocazione BEST FIT: se nelle partizioni viene allocato un processo di piccole dimensoni (Worst Fit), lo spazio lasciato inutilizzato in ciascuna partizione diventa elevato, ma non si puó utilizzare perché la partizione é impegnata. In ogni modo, nelle partizioni fisse anche se si usa la best fit lo spazio lasciato inutilizzato non puó essere utilizzato, per cui é di solito utilizzata una allocazione First Fit che, anche se non particolarmente utile per la frammentazione, almeno ha un basso tempo di allocazione. In generale migliori prestazioni si possono ottenere dividendo la memoria in partizioni variabili.

1.2.2 Partizioni variabili

In questo caso la dimensione delle partizioni é determinata dalla dimensione dello spazio richiesto per l'allocazione dei processi. In Fig.1.6 é rappresentata una situazione di questo tipo. Quando un processo deve essere allocato, viene riservata una partizione esattamente uguale alla dimensione del processo. In questo modo non si ha frammentazione interna. Il problema della frammentazione sussiste quando il processo termina, lasciando libera la partizione nella quale era allocato: questa é la frammentazione esterna.

Come si può affrontare il problema della frammentazione? Naturalmente avere una minore frammentazione vuol dire sfruttare al meglio la memoria fisicamente disponibile. A prima vista puó sembrare che un modo semplice per ridurre la frammentazione sia quello di compattare le partizioni utilizzate. Si consideri dunque la seguente Fig.1.7, dove le partizioni libere sono rappresentate con una ombreggiatura:

In questa figura viene rappresentata una situazione esemplificativa: in memoria contigua a partizioni variabili sono allocati quatro processi, ma nessuna partizione libera é sufficiente per contenere il prossimo processo. Il compattamento risolverebbe il problema. L'eventuale operazione di compattameno, tuttavia, é estremamente costosa in termini di tempo. Infatti si tenga conto del fatto che il compattamento richiede lo spostamento di una quantitá di memoria tanto maggiore quanti piú processi sono presenti, come visualizzato in Fig.**??**.

Nella Fig.1.8sono presenti quattro situazioni in memoria; anche in questo caso la memoria libera é rapprentata con una ombreggiatura. La prima figura a sinistra rappresenta il caso in cui ci sono due processi caricati in memoria. Se é richiesto di caricare un terzo processo in memoria, supponiamo di 50KB, lo spazio libero é sufficiente per la sua allocazione, in particolare nella seconda partizione libera, di 100K. Nella seconda é visualizzata una richiesta di allocazione di un quarto processo di 60K. In questo caso non c'é piú spazio sufficiente, a meno che non si compattino i tre processi P1, P2

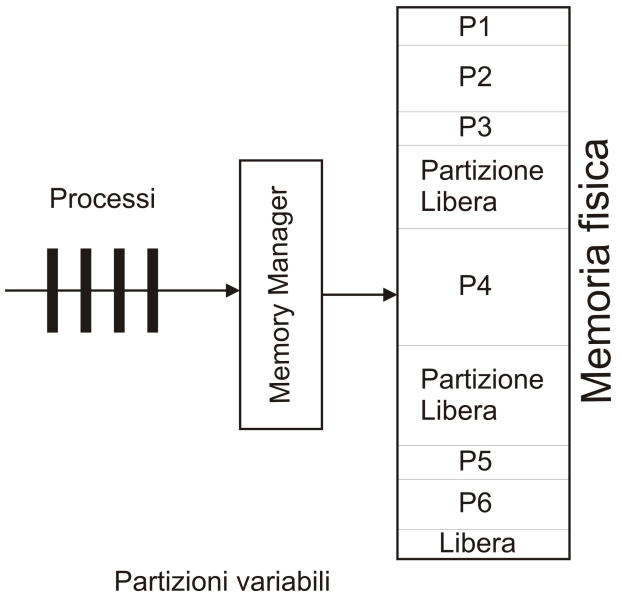

Figura 1.6: Una situazione in memoria a partizioni variabili

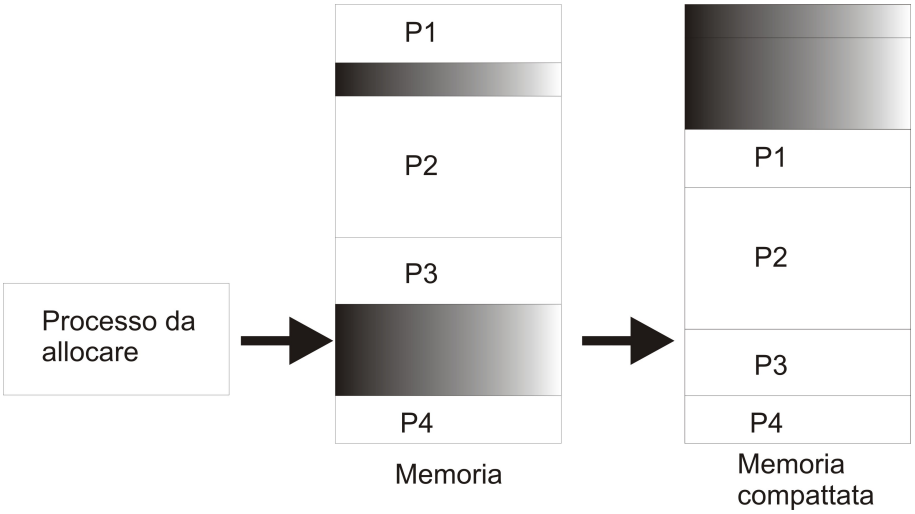

Figura 1.7: Compattamento della memoria; le partizioni ombreggiate sono partizioni libere

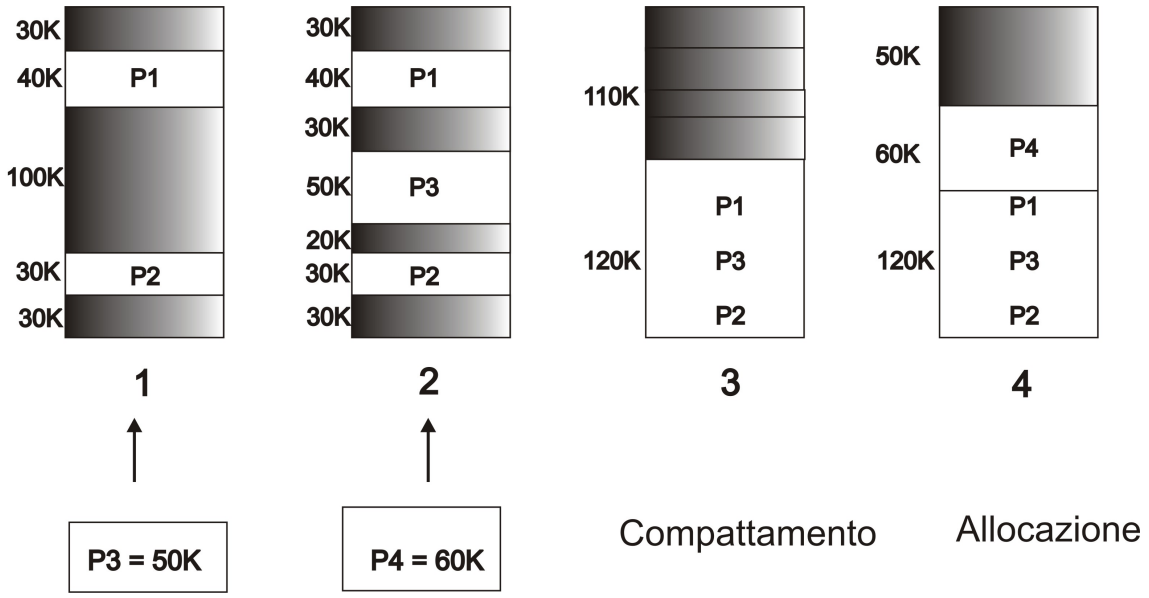

Figura 1.8: Compattamento della memoria

e P3, arrivando cosí alla situazione riportata nella terza figura. Questo vuol dire spostare un totale di 120KByte, cioé leggere e riscrivere in un altro indirizzo ciascuno dei 120KByte. Supponendo che il tempo di lettura/scrittura di 1 byte nella architettura considerata sia pari a 10ns, ad esempio, questo vuol dire che il compattamento richiederebbe $120 \cdot 10^3 \cdot 10 \cdot 10^{-9} \cdot 2 = 1.2ms$. Cioé il sistema resterebbe fermo per 1.2 ms secondi, perché durante lo spostamento i processi non possono eseguire in concorrenza. Il compattamento, peraltro, dovrebbe essere eseguita continuamente o comunque quando la frammentazione sale oltre un crto valore, perché ci sono continuamente processi che terminano la loro esecuzione e nuovi processi che vengono creati. Tenendo conto che i tempi della CPU sono di alcuni ordini di grandezza piú veloci, ci si rende conto che questo é normalmente inaccettabile.

Queste considerazioni possono essere facilmente espresse in modo piú formale valutando il **costo del compattamento**, che puó essere definito come un rapporto tra il tempo di compattamento e quello di riempimento. Avere un alto costo vuol dire che il tempo di compattamento é maggiore di quello di riempimento.

Il riempimento della memoria é legato alla multiprogrammazione, e rappresenta il tempo nel quale si riempie la memoria per una data frequenza di arrivo di nuovi processi, λ [processi/secondo]. Il tempo che ci vuole per compattare la memoria dipende da quanta memoria deve essere compattata. Supponiamo che il coefficiente di frammentazione sia $f = \frac{memoria\ libera}{memoria\ totale}$ e che la memoria totale sia M. Allora la memoria libera é fM e quella occupata é $(1-f)M$. Se la dimensione media dei processi é p[byte/processo], il tempo di riempimento ádato da $\frac{fM}{p\lambda}$ cioé $[byte]/[\frac{byte}{processo}][\frac{processi}{secondo}]$ mentre il tempo di compattamento é $(1-f)M2\tau$ dove τ é il tempo di lettura/scrittura di un byte in [secondo/byte]. Cioé il costo del compattamento é:

$$Costo = tempocompattamento/temporiempimento = 2(1-f)M\tau/fMp\lambda = 2(1-f)\tau/fp\lambda$$

Se la frammentazione fosse nulla, il costo andrebbe all'infinito mentre se la frammentazione fosse uguale a 1 il costo scenderebbe a zero. Queste considerazioni qualitative possono essere visualizzate nella Fig.1.9.

Quindi per avere basso costo bisogna avere una frammentazione alta cioé molta memoria libera rispetto alla memoria totale, mentre il costo é improponibile se la memoria libera é poca rispetto alla

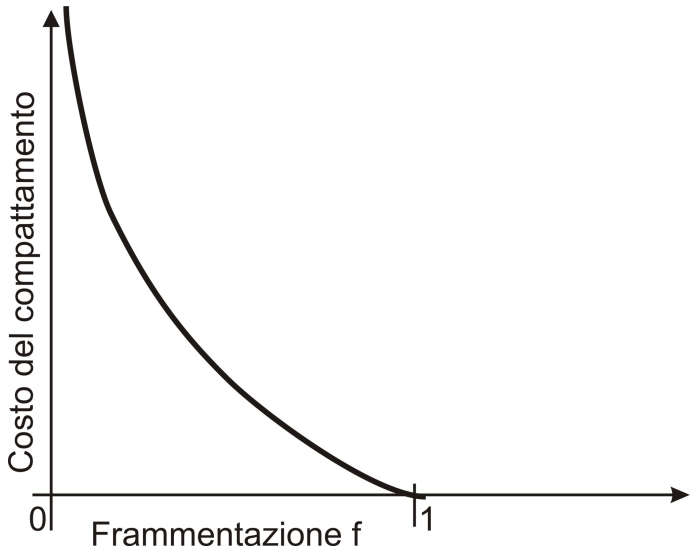

Figura 1.9: Il costo del Compattamento della memoria

memoria totale. Cioé il compattamento ha basso costo quando non é necessario perché c'é molta memoria, e diventa improponibile se la memoria libera é poca, cioé proprio quando servirebbe. Per questi motivi il **compattamento della memoria non é praticabile** per via software. É richiesto, eventualmente e nel caso si voglia usare questo approccio, un **Supporto Hardware**, cioé un sistema aggiuntivo che realizza il compattamento della memoriaper via hardware. Naturalmente ogni complicazione della architettura comporta un costo elevato.

1.2.3 Regola del 50%

In questa sezione si riporta una relazione sul rapporto tra frammentazione e rapporto fra la dimensione delle partizioni libere e la dimensione delle partizioni occupate. Questa relazione fornisce una chiave interpretativa per valutare le possibili allocazioni.

Consideriamo dunque la situazione raffigurata in Fig.1.10

In Fig.1.10 sono riportati due esempi di allocazione di processi in una memoria contigua a partizioni variabili. Ci sono alcuni processi, qualcuno allocato senza partizioni libere in mezzo, qualcuno isolato. Nella situazione di sinistra abbiamo sei processi e tre partizioni libere. Notiamo che i processi terminano e rilasciano la memoria occupata, quindi a regime é ben difficile che ci siano molte partizioni allocate contiguamente, una in seguito all'altra. Nella situazione raffigurata a destra ci sono 7 processi e 5 partizioni libere. In ogni caso é facile osservare che ci sono partizioni occupate (indicate con A in figura e che sono in numero n_A) che, quando rilasciate, aumentano il numero delle pertizioni libere, altre che diminuiscono questo numero (indicate con B e che sono in numero n_B) e altre che lo lasciano invariato (chiamae C; ci sono in memoria n_C partizioni di questo tipo). Nella figura a sinistra, ad esempio, $n_A = 2, n_B = 1, n_C = 3$ e $N = 6$. É altresí chiaro che per avere stabilitá in memoria, cioé una situazione tale che i vari processi di allocazione e disallocazione non comportino un aumento incontrollato né delle partizioni libere né di quelle occupate, all'equilibri statistico il numero medio delle partizioni che provocano aumento delle partizioni libere deve essere uguale al numero medio delle partizioni occupate che provocano diminuzione delle partizoni libere. Cioé all'equilibrio statistico deve essere $n_A = n_B$. In definitiva, il numero delle partizioni occupate é $N = n_A + n_B + n_C = 2n_B + n_C$. Il numero delle partizioni libere é poi dato dalla seguente considerazione: ogni partizione di tipo A non é associata a nessuna partizione libera, mentre c'é una partizione libera per ogni partizione occupata di tipo C. Ogni partizione di tipo B poi porta

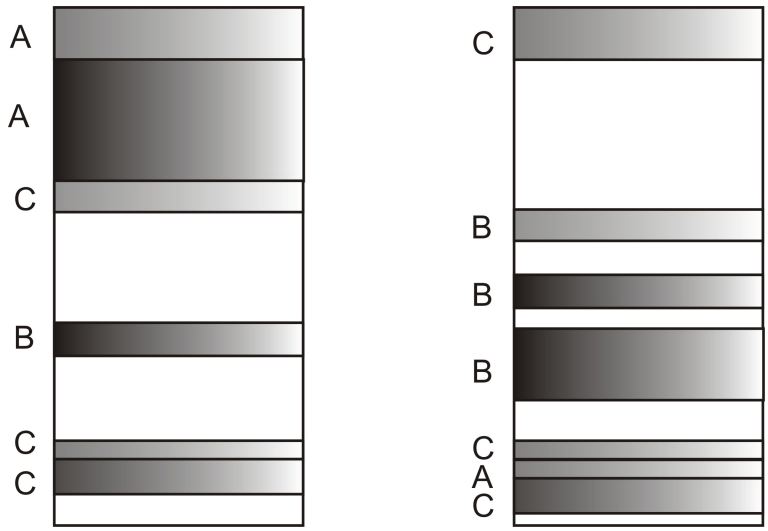

Figura 1.10: Una tipica situazione in memoria contigua a partizioni variabili

con sé due partizioni libere; peró ogni partizione di tipo B ha in corrispondenza una di tipo C. Quindi, indicando con M il numero di partizioni libere, $M = \frac{2n_B+n_C}{2} + a$ dove a é un termine per aggiustare il conto per situazioni nelle quali la divisione intera puó dare un risultato approssimato. Comunque, il termine di aggiustamento é sempre molto piccolo e puó essere trascurato nei casi reali nei queli ci sono molti processi. Quindi, visto che $N = 2n_B + n_C$ e che $M = \frac{2n_B+n_C}{2}$ ovviamente $M = \frac{N}{2}$ sempre all'equilibrio statistico e per un alto numero di processi. In altri termini, il numero di partizioni libere é la metá delle partizioni occupate.

Questo risultato porta ad una considerazione importante sul rapporto tra la dimensione media delle partizioni libere e quella delle partizioni occupate. Chiamando k questo rapporto, si puó dire che la frammentazione, cioé il rapporto tra dimensione della memoria libera e dimensione memoria totale é, chiamando D la dimensione media delle parizione occupate, cioé dei processi:

$$f = \frac{mem.libera}{mem.totale} = \frac{MkD}{MkD + ND} = \frac{NkD/2}{NkD/2 + ND} = \frac{k/2}{k/2 + 1} = \frac{k}{k+2}$$

cioé la frammentazione é una funzone crescente con k, che cresce lentamente da 0 a 1 per k che va da 0 a k molto grande. Quindi, per ridurre la frammentazione, é importante che la **dimensione delle partizioni libere sia piccola rispetto a quella delle partizioni occupate**.

1.2.4 Buddy System

In molti casi, nei queli il problema dello spreco di memoria é meno importante rispetto al problema della velocitá di allocazione e rilascio, ci sono degli approcci che vanno sotto il nome di **Buddy Systems** o **Metodi dei Compagni**. La memoria é divisa in blocchi di 2^k byte. Ci sono cosí blocchi lunghi $2^0 = 1$ byte, $2^1 = 2$ byte, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$ e cosí via.

Una allocazione esemplificativa partendo da una memoria di 1Mbye é rappresentata in Fig.1.11, dove sono ipotizzate tre richieste consecutive di 100K, 50K e 90K.

Il fatto chiave di questo metodo é che se conosciamo l'indirizzo di un blocco, cioé dell'indirizzo del primo byte del blocco, e ne conosciamo la dimensione, allora conosciamo subito l'indirizzo del suo compagno: questo porta ad un ricongungimento molto veloce delle partizioni libere. Man mano che l'algoritmo procede, infatti, *l'indirizzo di un blocco di dimensione* 2^k *é un multiplo di* 2^k. In generale, l'indirizzo del compagno di dimensione 2^k che parte all'indirizzo x é dato dalla sseguente regola:

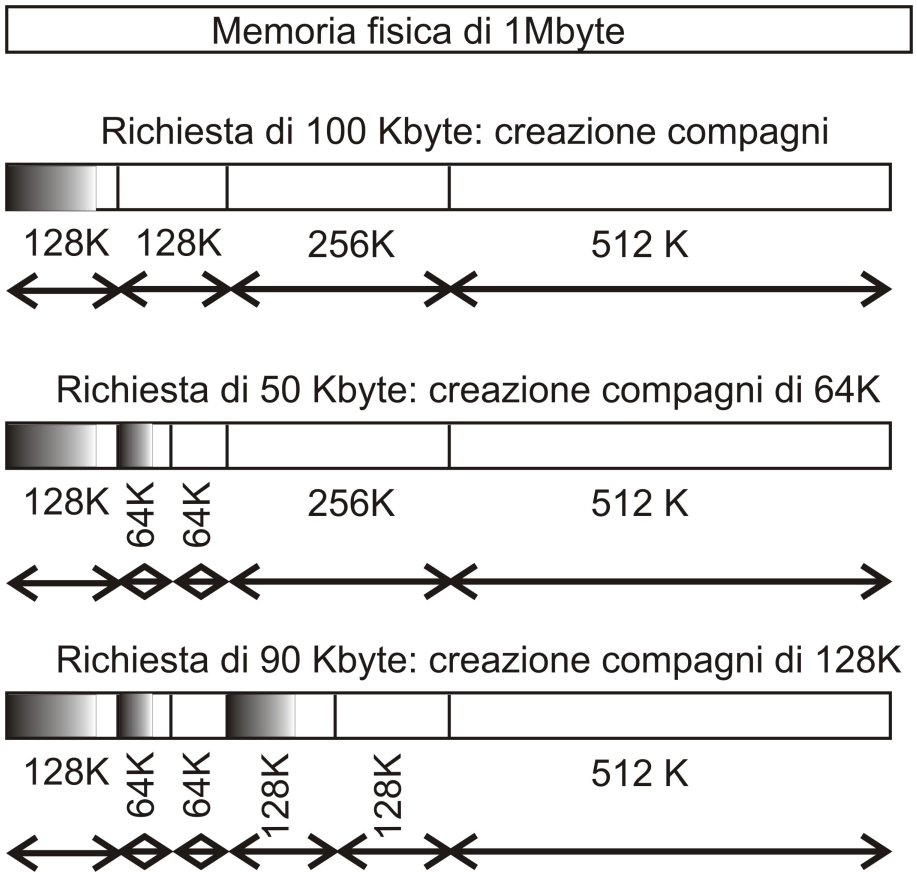

Figura 1.11: Esempio di allocazione con il metodo dei buddy-system

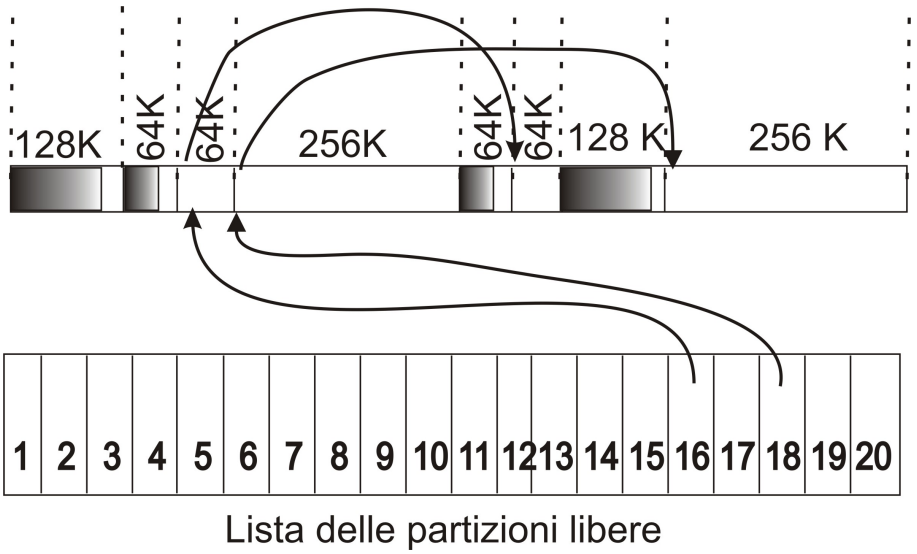

Figura 1.12: Strutture dati per il metodo dei compagni

- Se un blocco parte da x ed é $x\%2^{k+1} = 0$ allora l'indirizzo del compagno é $= x + 2^k$

altrimenti

- Se un blocco parte da x ed é $x\%2^{k+1} = 2^k$ allora l'indirizzo del compagno é $= x - 2^k$

Naturalmente l'operazione di modulo é effettuata con l'OR esclusivo, e quindi é estremamente veloce.

Facendo riferimento alla Fig.1.11 vediamo infatti che nella prima allocazione dall'alto ci sono due compagni da 128K (2^17), uno che inizia all'indirizzo x=0 e l'altro che inizia all'indirizzo 128K. Applicando la regola dobbiamo fare $0\%2^18$ che é 0. Infatti $0 = 0 \cdot 2^18 + 0$ cioé il quoziente e il resto sono entrambi a zero; quindi il suo compagno é quello che comincia a $x + 2^17 = 0 + 2^17 = 2^17 = 128K$. D'altra parte per il compagno con x=128K si ha: $2^17\%2^18 = 128K\%2^18 = 2^17$ infatti $2^17 = 0 \cdot 2^18 + 2^17$ cioé il quoziente é 0 e il resto 2^17. Allora il compagno é all'indirizzo $x - 2^k = 128K - 128K = 0$.

Nella seconda allocazione ci sono due compagni di 64K (2^16) uno all'indirizzo 128K e l'altro all'indirizzo 192K. Allora per il compagno di 2^16 che parte da 128K é $x\%2^17 = 0$ (infatti il quoziente della divisione 128k/256K é 0 e il resto é 128k) e per il compagno che parte da 192k é $x\%2^17 = 2^16$ (infatti il quoziente della divisione 192K/128K é 1 e il resto é 64K cioé 2^16.

Nella terza allocazione ci sono due compagni di 128K (2^17) di cui uno parte da x=256K e l'altro parte da x=384K. Se facciamo $x\%2^{k+1} = x\%2^18 = x\%256K$ si ha un resto di 0 per il primo e 128K per l'altro; quindi il compagno del primo parte all'indirizzo 384K e il compagno del secondo parte da 256K.

Vediamo innanzitutto delle strutture dati tipicamente necessarie per la gestione di questo algoritmo, supponendo di essere dopo una serie di allocazioni giá visualizzate in Fig.1.11. Naturalmente l'algoritmo puó essere costruito in molti modi, ma vediamo un modo ragionevole. La struttura dati é costituita essenzialmente da un array di puntatori alle partizioni libere di dimensione 2, 4, 8, 16, 32, 64, 128 etc. come illustrato in Fig.**??**. Nelle partizioni libere c'é il puntatore alla prossima partizione libera. Non bisogna dimenticare poi il bit occupato/libero che si trova in ogni partizione.

Osserviamo innanzitutto che ogni blocco contiene un bit che é settato a 1 se il blocco é libero, 0 se il blocco é occupato. Questo e' **l'overhead** in memoria dell'algoritmo. I blocchi liberi sono descritti da una lista concatenata bidirezionale allocata nei blocchi liberi stessi.

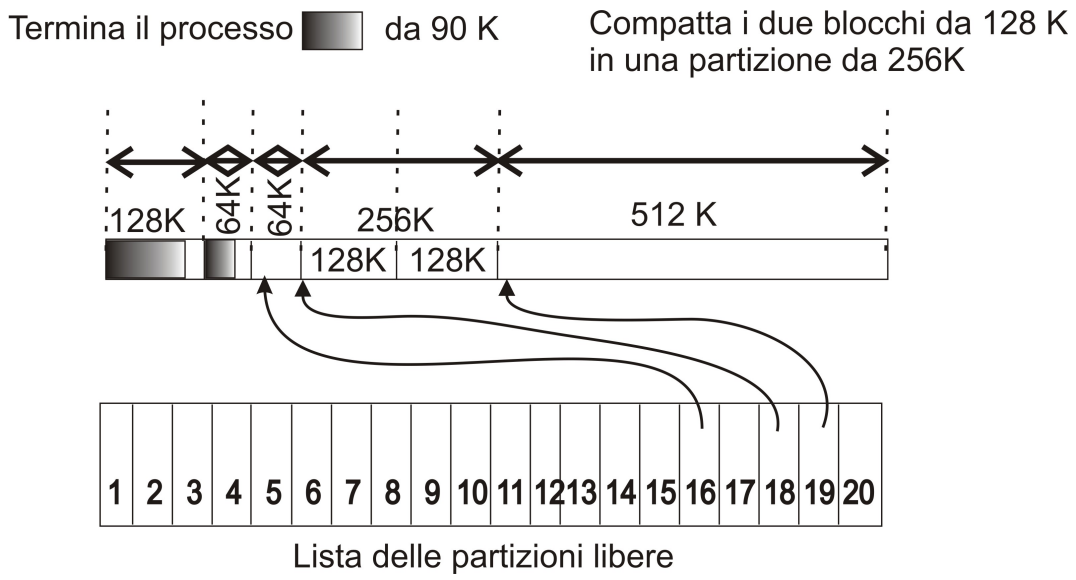

Figura 1.13: Compattamento di due compagni di 128K per costituire un compagno da 256K

Utilizzando il bit di occupazione, il rilascio con seguente compattamento viene fatto molto semplicemente: *quando termina un processo, rilasciando il suo blocco all'indirizzo x, si determina l'indirizzo del suo compagno e si vede se é libero o no. Se é libero, si accorpano i due compagni in un'unica partizione libera.* Questo é rappresentato in Fig.**??**.

1.2.5 Algoritmi di allocazione

Questa sezione si occupa di questa domanda: quando devo allocare un processo, in quale partizione libera é meglio metterlo? Ovviamente nella gestione della memoria contigua il processo sará messo nella partizione che puó contenerlo interamente, visto che i processi devono risiedere interamente in memoria.

I metodi fondamentali di allocazione sono:

- First-Fit: il processo viene allocato nella prima partizione libera che puó contenerlo. L'allocazione in questo caso é la piú veloce, ma niente si puó dire sullo spazio rimasto.
- Best-Fit: il processo viene allocato nella partizione libera che lascia meno spazio inutilizzato. Il tempo di allocazione é piú alto perché bisogna esaminare tutte le partizioni libere. Il rapporto tra dimensione delle partizioni occupate e partizioni libere é peró minimo.
- Next-Fit: il processo viene allocato nella prima partizione libera che segue la precedente allocazione.
- Worst-Fit: il processo viene allocato nella partizione che lascia piú spazio libero.

Queste allocazioni sono illustrate nell'esempio di Fig.1.14, dove sono esemplificate le situazioni in memoria dopo una prima allocazione di 4kbyte e una seconda di 16kbyte.

Seguendo l'esempio di Fig.1.14, il processo da 4K é allocato subito dopo la partizione libera di 22K. Il processo da 16K viene allocato nella First Fit nella partizione da 22K (lasciando una partizione libera di 6K, con la Best Fit in quella da 18 K lasciando 2K liberi, con la Next Fit in quella di 24K perché é la prima partizione dipo quella di 4K allocata in precedenza, e con la Worst Fit in quella da 24K perché é quella che lascia piú spazio libero (viene lasciata infatti una partizione libera di 8K).

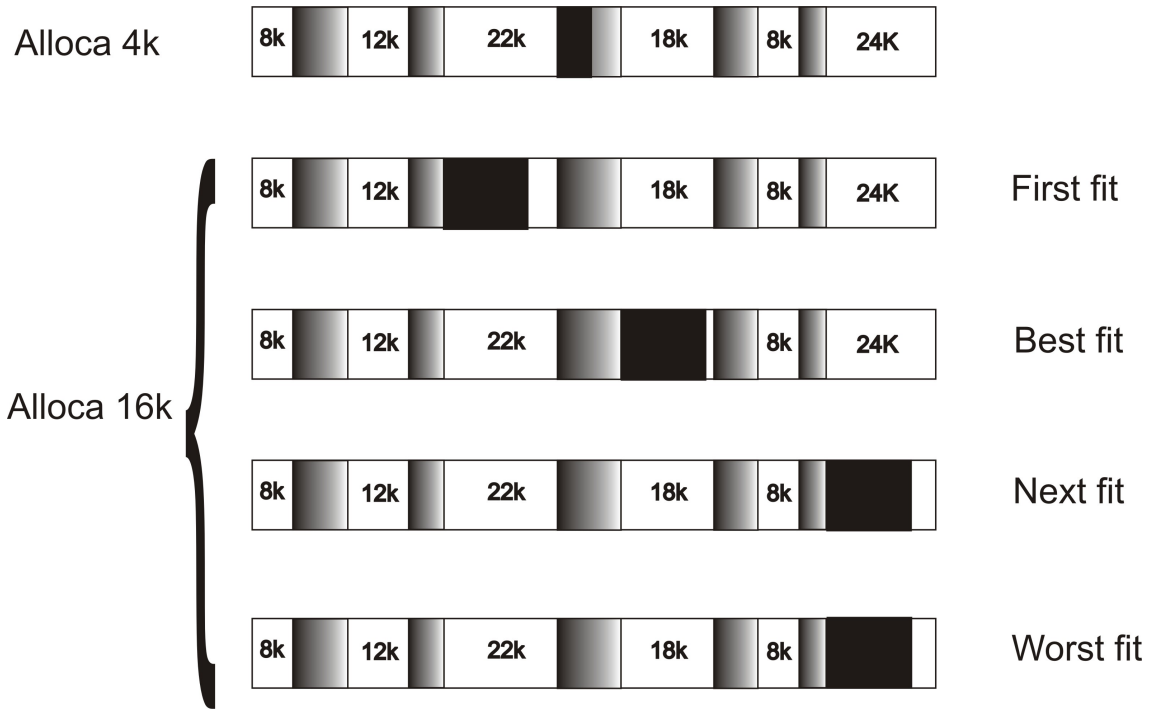

Figura 1.14: Allocazione first-fit, best-fit, next-fit, worst-fit. Attenzione: i processi allocato sono rappresentati con l'area ombreggiata; le zone bianche sono partizioni libere.

L'algoritmo che ha il minor rapporto tra diensione delle partizioni libere e dimensione delle partizioni occupate é il Best-Fit che porta a una minore frammentazione. Tuttavia il suo overhead in termini di tempo di gestione non é trascurabile.

Le strutture dati piú naturali per gestire una memoria contigua sono le liste concatenate. Facendo riferimento alla memoria rappresentata in Fig.1.15, che rappresenta una situazione in memoria contigua, dove ci sono tre processi e due partizioni libere, é naturale descrivere la memoria la seguente lista concatenata come rapresentato in Fig.1.16.

Gli algoritmi di gestione devono cercare la partizione opportuna in fase di allocazione, devono rilasciare la partizione occupata in caso di terminazione del processo e devono eventualmente fondere assieme due partizioni libere se sono adiacenti. In altri termini:

allocazione A seconda del criterio scelto, Best-Fit, First-Fit o Next-Fit, si seleziona la partizione giusta per l'allocazione. Viene inserito un elemento nella lista marcato come 'Occupato' con il campo 'start' corrispondente all'indirizz della partizione libera e 'size' uguale alla dimensione del processo, prima dell'elemento corrispondente alla partizione libera selezionata, per mantenere l'ordine di indirizzo. L'elemento della partizione libera viene modificato con il seguente algoritmo:

$$start = start + size_{processo}; \quad size_{libera} = size_{libera} - size_{processo}$$

rilascio Quando un processo termina la memoria occupata viene rilasciata, e diventa partizione libera. Questo vuole dire semplicemente che il flag O/L che era O (occupato) diventa L (Libero) e nessuna altra modifica viene fatta.

fusione In questa fase si fondono assieme le partizioni libere. Ovviamente questo viene fatto se ci sono due partizioni libere consecutive. Quindi, una volta che una partizione viene rilasciata, e

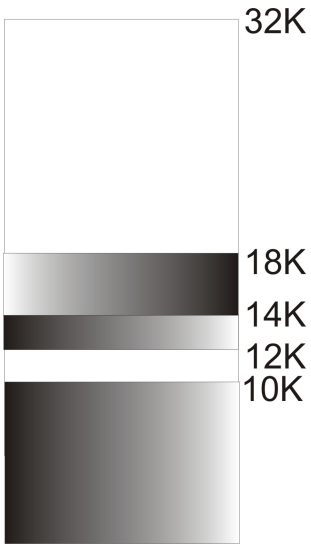

Figura 1.15: Un esempio di una memoria contigua con tre processi caricati e due partizioni libere

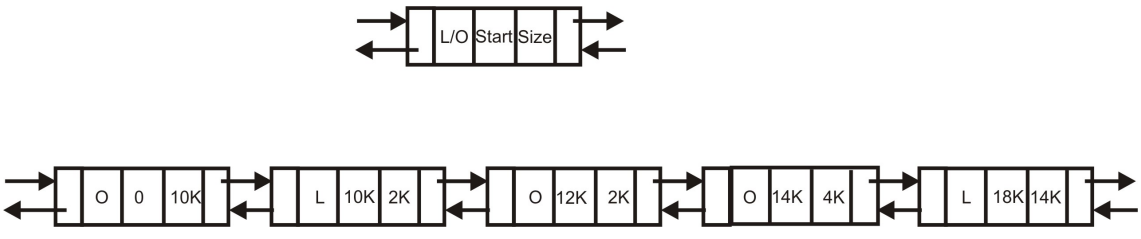

Figura 1.16: Allocazione first-fit, best-fit, next-fit, worst-fit

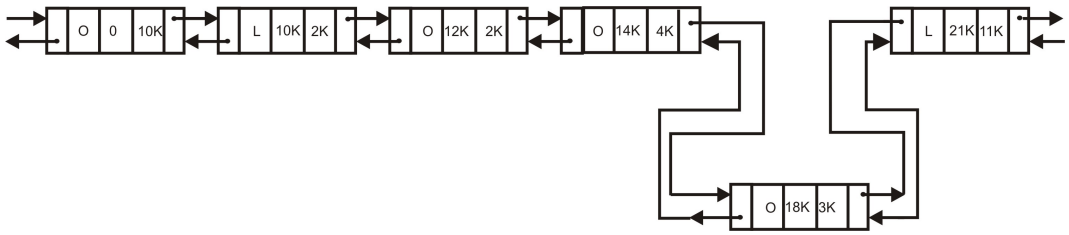

Figura 1.17: Allocazione di un processo di 3K e modifica della partizione libera

l'elemento corrispondente viene aggiornato con flag L, si ricerca a destra e sinistra per vedere se ci sono partizioni libere. In questo caso si controlla se la somma $start + size$ del blocco libero a sinistra nella lista é uguale all'indirizzo $start$ del bocco appena liberato. In questo caso, si fondono insieme eliminando l'elemento del nuovo blocco e modificando il campo size del blocco di sinistra, come $size = size + size_{bloccoappenaliberato}$. Analogamente per l'elemento a destra.

Un esempio si puó vedere in figura Fig.1.17. In questo caso si cerca di allocare un processo di 3K, che viene messo nella partizione libera di 14K e lasciando 11K liberi.

Una riduzione del tempo di gestione della memoria si puó ottenere se si considerano solo liste delle partizioni libere; dopo tutto, le informazioni sulla allocazione dei processi sono contenute nel PCB. In questo caso la visita della lista diventa molto piú veloce perché ci sono meno elementi. L'uso della lista delle sole partizioni libere nel caso della allocazione di un processo da 3K é visualizzato in Fig.1.18. L'algoritmo di gestione diventa:

allocazione ricerca la partizione in accordo con il criterio di allocazione. I campi dell'elemento selezionato vengono cosí modificati: $start = start + size_{processo}$, $size = size - size_{processo}$.

rilascio la terminazione di un processo si traduce nella inserzione di un elemento con start e size uguali a quelli del processo terminato.

fusione la fusione viene fatta nello stesso modo del caso precedente: dopo aver inserito il nuovo elemento si guarda a destra e sinistra per vedere se ci sono adiacenze.

La semplificazione ottenuta in questo caso é evidente giá dalla sola figura, rispetto al caso precedente.

É importante osservare che la lista delle partizioni libere puó essere allocata nelle ultime locazioni delle partizioni libere, perché sicuramente l'allocazione di un processo nelle partizioni libere lascia un pó di spazio nella partizione (frammentazione esterna).

1.2.6 Allocazione con Bitmap

Una alternativa all0uso delle liste consiste nell'uso di **BitMap**, cioé di matrici di bit, dove ciascun bit é associato ad un blocco di memoria. Se il bit é settato il blocco di memoria é occupato, altrimenti é libero. Se per esempio si ha una memoria di 10 Mbyte gestita con una bitmap, dove ogni bit é associato a un blocchetto di 1 byte, la dimensione della matrice é di 10 Mbit, cioé 1,25 Mbyte. Se ogni bit é associato a un blocchetto di 4 bit, allora é necessaria una memoria di 2,5 Mbyte e cosí via. La dimensione della bitmap é compensata dalla velocitá delle operazioni di allocazione/rilascio: per allocare basta vedere se nella matrice c'é un numero sufficiente di bit a zero; il rilascio corrisponde al reset del valore dei bit. Tuttavia queste operazioni di bit necessitano generalmente di un supporto hardware, perché possono essere laboriose se effettuate unicamente via software.

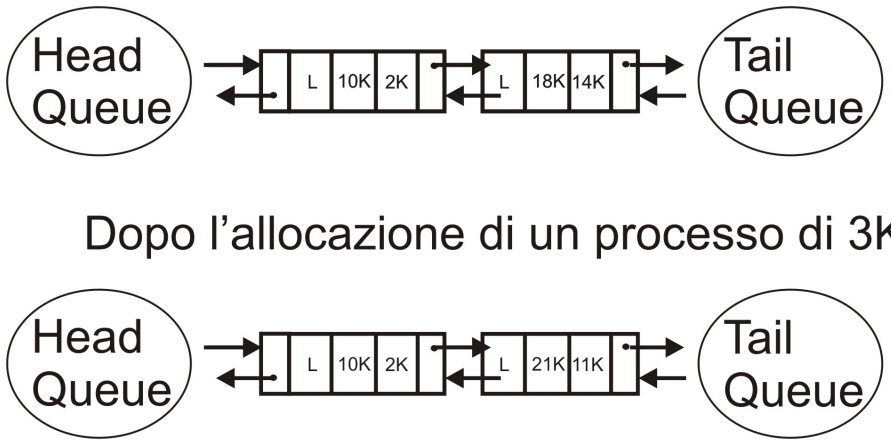

Figura 1.18: Allocazione di un processo di 3K e modifica della lista delle sole partizioni libere

1.3 Memoria virtuale

La gestione della memoria mediante memoria virtuale riguarda un tipo di indirizzamento che permette **l'esecuzione di processi piú grandi della memoria fisica**. Alternativamente, agevola la multiprogrammazione in cui sono caricati in memoria molti processi. Comunemente questo viene realizzato mediante l'algoritmo della memoria paginata.

1.3.1 Memoria paginata

Questo tipo di algoritmo considera che i processi siano divisi in sezioni di una certa dimensione, che puó essere 1024 byte, o di 4096 byte e cosí via, restando comunque su valori piuttosto limitati. La memora fisica viene anche divisa in pagine della stessa dimensione, che sono chiamate **page frames**. Questa strategia comporta un tipo di frammentazione interna, che é legata al fatto che i processi non possono generalmente essere divisi in un numero intero di pagine virtuali. La frammentazione mediamente é quindi pari a metá della dimensione della pagina.

Vediamo con un esempio il funzionamento della memoria paginata. Consideriamo il seguente frammento di programma assembler (assolutamente esemplificativo):

```
start:
    move 5,R1
    move 8,R2
    jmp  $05DE ; salto assoluto
```

Supponiamo inoltre che le istruzioni siano 16 bit e che il programma sia caricato all'indirizzo 0. In memoria ci sono quindi i codici operativi delle istruzioni. Quando la Cpu emette il primo indirizzo, cioé $0000, il codice operativo della prima istruzione viene letto dalla memoria e decodificato. La seconda istruzione é all'indirizzo $0002 e la terza all'indirizzo $0004, supponendo che la memoria sia organizzata a byte. Si ricorda che questi sono indirizzi virtuali. L'istruzione `jmp $05DE` accede alla memoria all'indirizzo $05DE nel senso che la prossima istruzione da eseguire é memorizzata in memoria all'indirizzo $05DE appunto. Nella memoria paginata viene caricata il programma in pagine, a seconda del loro utilizzo. Se le pagine sono di 1024 byte, dopo il caricamento della prima pagina in memoria si trovano le prime 512 istruzioni se ogni istruzione é a 16 bit, la prima all'iindirizzo 0, la seconda 2, la 512-esima all'indirizzo $03FE (l'ultima istruzione di 16 bit é allocata agli indirizzi $03FE-$03FF). In altri termini l'istruzione all'indirizzo $05DE non si trova in memoria e deve essere caricata da disco. Supponiamo che nella memoria fisica ci sia una page frame libera

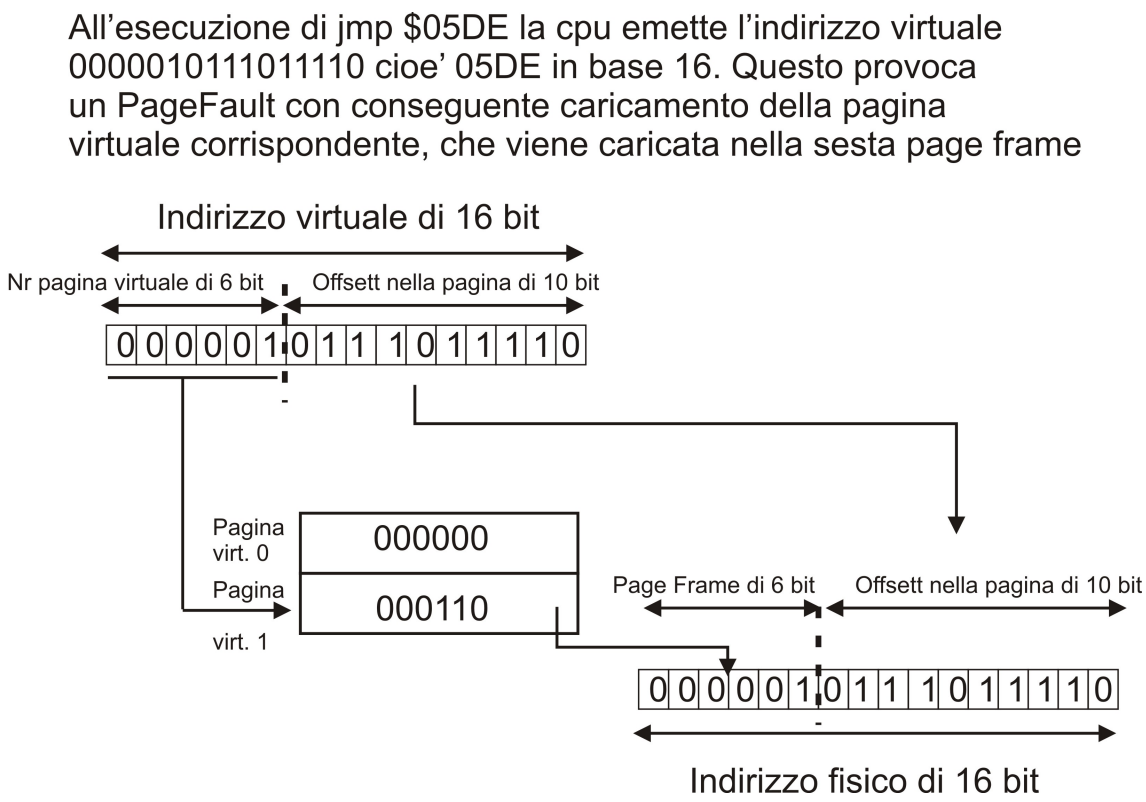

Figura 1.19: Esempio di traduzione secondo quanto descritto nel testo

all'indirizzo 0001100000000000 cioé $1800 in base 16, e in questa page frame viene caricata. Cioé, riassumendo, abbiamo supposto di trovarci in una situazione in cui che gli indirizzi virtuali da 0 a $3FF sono caricati nella page frame 0, che vá dall'indirizzo fisico $0000 a $3FF (per una dimensione di pagine pari a $400, o 1024 byte in base 10), e che la pagina virtuale seguente, cioé da $400 a $7FF sia caricata nella page frame 1, che va' dall'indirizzo fisico $1800 a $1BFF. L'accesso all'indirizzo $05DE Questa situazione é dunque descritta nella prossima figura, Fig.1.19:

La tabella fulcro della traduzione si chiama **Page Map Table**. É una tabella che risiede in memoria centrale, e l'accesso alle informazioni viene fatto con i tempi d'accesso alla memoria centrale. Per questo motivo si semplifica l'accesso mediante memorie **cache** dove sono memorizzate le ultime coppie (numero pagina virtuale, numero page frame). Se la probabilitá di HIT in cache é sufficientemente alta, la traduzione avviene quindi in tempi rapidissimi. Queste cache sono chiamate **TLB** per Translation Lookaside Buffer. Sono contenute nel modulo hardware chiamato **MMU** (Memory Management Unit) che era inizialente un circuito integrato separato dalla CPU, ora é normalmente integrato all'interno.

La tabella PMT contiene tutte le informazioni della pagina virtuale, in particolare:

- un bit di validitá che dice se la pagina é residente in memoria o meno
- l'indirizzo disco, cioé le informazioni su dove recuperare la pagina su disco
- i flag di protezione della pagina che dicono se la pagina puó essere letta o scritta
- il numero di page frame dove é caricata la pagina virtuale
- bit di riferimento e di modifica. Questi bit vengono settati quando la pagina é riferita e quando viene modificata rispettivamente, e sono utilizzati nella gestione veloce dei rimpiazzamenti.

La dimensione della PMT é pari al numero di pagine virtuali nelle quali é diviso un processo, che é la spazio di idirizzamento diviso la dimensione della pagina virtuale. Questi valori possono essere notevoli; per questo motivo é usuale dividere la PMT su piú livelli, per avere pagine di dimensioni piccole, che possono essere memorizzate nella memoria virtuale.

É importante osservare che se una pagina richiesta non é in memoria (bit di validitá a zero) deve essere caricata da disco. Questo evento, **PAGE FAULT**, comporta un tempo molto elevato e quindi ha un costo molto alto. Questi sistemi di gestione memoria devono quindi minimizzare la probablitá di avere page fault. Il costo é alto per i seguenti motivi: intanto un processo che ha un page fault deve essere messo in coda d'attesa su disco, e quando la pagina é letta da disco deve essere rimesso in coda d'attesa dei processi pronti in attesa ce lo schedultori lo selezioni. Poi, si consideri questo fatto: dove viene messa la pagina appena letta da disco? se si seleziona una pagina che é stata modificata (bit di modifica) bisogna scrivere la pagina su disco, passando di nuovo per le code d'attesa.

Da quanto visto, nella memoria paginata gli indirizzi virtuali vengono divisi in due parti: il numero di pagina virtuale di n_1 bit e l'offsett nella pagina di n_2 bit. La dimensione della pagina é dunque di 2^{n_2} bit. Per questo motivo é usuale rappresentare lì'indirizzo virtuale α come:

$$\alpha = x \cdot c + w$$

dove x é il numero di pagina virtuale, c la dimensione della pagina e w l'offsett nella pagina. D'altra parte, l'indirizzo fisico é:

$$\beta = f(x) \cdot c + w$$

dove f(x) é la funzione di traduzione, realizzata con la PMT.

La dimensione della pagina dipende da molti fattori, per esempio legati alla architettura del calcolatore, e alla dimensione dei blocchi su disco, e alla dimensione massima dello spazio di indirizzamento dei processi. Vediamo come si puó scegliere un valore ottimale di dimensione di pagina, considerando solo la richiesta di memoria dell'algoritmo di gestione della memoria. In questo caso, il costo C si puó ritenere legato alla dimensione della PMT e della frammentazione, cioé: $C = \frac{dim.massima\ spazio\ indirizzamento}{dim.pagina} + dim.pagina/2$. Derivando C rispetto alla dimensione di pagina e uguagliando a zero la derivata, si ottiene:

$$dim.pagina = \sqrt{dimensione\ riga\ PMT \cdot 2 \cdot dim.massima\ spazio\ indirizzamento}$$

Ad esempio, se ogni riga di PMT é lunga 8 byte, e la dim. massima spazio indirizzamento é 2^20, si ha che la dimensione della pagina é di 4Kbyte.

L'operazione di traduzione é ottenuta mediante il concatenamento di sequenze binarie, e quindi richiede un supporto hardware.

In generale, dunque, il meccanismo della traduzione si puó descrivere con la Fig.1.20.

Da quanto detto possiamo enfatizzare alcune caratteristiche della memoria virtuale:

- ogni processo ́diviso in pagine virtuali
- la memoria fisica é diviisa in pagine della stessa dimensione.
- ogni processo ha la SUA PMT
- le pagine fisiche possono essere condivise. Anzi la condivisione é conveniente per motivi di efficienza (basta pensare a piú versioni dello stesso processo eseguite da piú utenti) ma questo comporta possibili problemi di coerenza.
- la traduzione comporta accesso in memoria, che puó essere ridotto utilizzando cache di traduzine

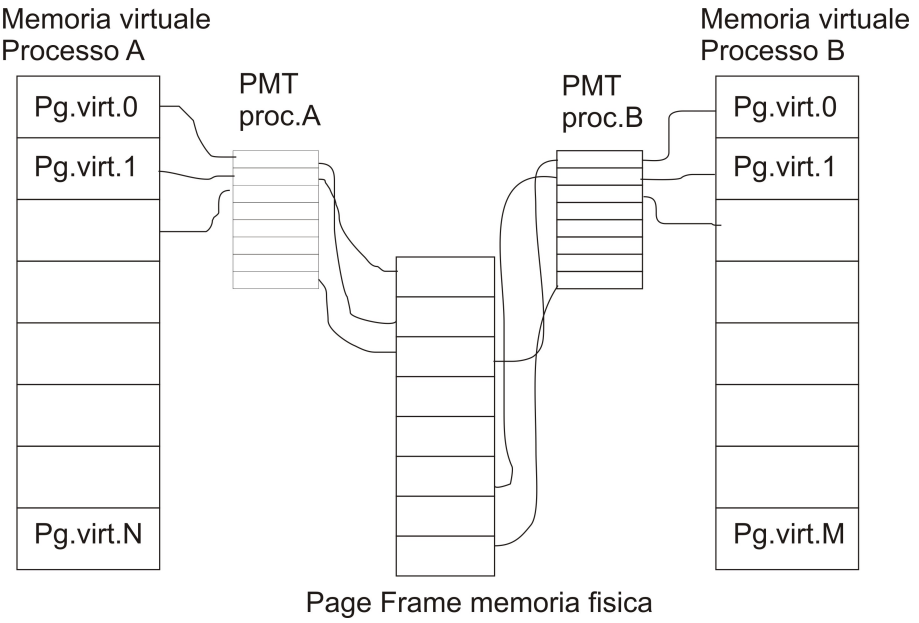

Figura 1.20: Il meccanismo della traduzione

1.3.2 Agoritmi di Rimpiazzamento

Gli algoritmi d gestione a memoria virtuale devono risolvere tre problemi:

1. cioé come identificare la pagina da caricare in memoria
2. Placement cioé dove mettere la pagina
3. Replacement cioé quale pagina rimpiazzare nel caso non ci siano page frame liberi

Visto che gli indirizzi virtuali si possono dividere in nr. di pagina virtuale e offsett, é usualmente sufficiente, per analizzare il comportamento di un sistema di memoria virtuale, considerare la sequenza delle pagine virtuali. Questa sequenza é chiamata **stringa di riferimenti**. Ad esempio, un processo puó richiedere la stringa dei riferimenti:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1

cioé le pagine virtuali secondo questa sequenza. All'interno delle pagine virtuali, poi, l'accesso é regolato secondo l'offsett. Il comportamento della memoria virtuale é determinato da questi interrogativi:

Fetch Cioé: quale pagina bisogna caricare in memoria? In realtá questa domanda é risolta automaticamente, cioé quando un processo richiede una pagina, é quella che deve essere caricata. Questo di chiama **Demand Paging**. Altre tecniche si possono utilizare, da quella di caricare non solo la pagina richiesta ma un intorno ad essa, e quella di caricare delle pagine in previsione al loro impiego futuro.

Placement . Cioé: dove devo mettere la pagina? Naturalmente le pagine vengono messe nel primo posto libero. Se non ci sono posti liberi bisogna rimpiazzare una pagina giá caricata, naturalmente salvandola su disco se é stata modificata. Ma si deve cercare in una zona della memoria dedicata al processo (strategie locali) o in tutta la memoria (strategie globali)? di solito si usano strateie gloali.

Replacement Cioé: Quale pagina devo eventualmente rimpiazzare? Questo é necessario se non ci sono posti liberi. In questo caso bisogna rimpiazzare una pagina. Ci sono molti criteri per

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2
x	x	x	x		x	x	x	x	x	x			x	x		

PF=12/17=0.7

Figura 1.21: Pagine virtuali caricate in memoria con rimpiazzamento FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2
x	x	x	x		x		x	x	x	x			x		x	

PF=11/17=0.65

Figura 1.22: Pagine virtuali caricate in memoria con rimpiazzamento LRU

decidere quale pagina rimpiazzare. I piú popolari sono FIFO e LRU (Least Recently Used). Nel primo caso si sostituisce la pagina piú vecchia, che risiede da piú tempo in memoria. Ma la pagina piú vecchia puó essere quella piú importante, cioé non solo quella piú riferita, ma quella recentemente utilizzata. Si sceglie in questo caso la pagina meno recentemente usata, o LRU.

Il rimpiazzamento LRU si puó vedere come una approssimazione del rimpiazzamento ottimo: la pagina da rimpiazzare é quella che verrá utilizzata tra piú tempo. Questo approccio non é fisicamente realizzabile, e puó essere utilizzato come riferimento perché fornisce risultati ottimi.

Vediamo degli esempi di probabilitá di page fault con questi rimpiazzamenti.

FIFO Supponiamo di avere 3 page frame e che la sequenza dei riferimenti sia quella giá vista. Allora, la sequenza viene elaborata secondo quanto illustrato in Fig.1.21, dove la prima riga contiene i riferimenti e l'ultima i casi in cui ho Page Fault.

Se aumento la memoria fisica posso avere un page frame minore o uguale a quelli ottenuti con minore quantitá di memoria. Questa é una **ANOMALIA** che contraddistingue FIFO.

LRU Anche in questo caso supponiamo di avere 3 page frame e che la sequenza dei riferimenti sia quella giá vista. Questo algoritmo é mostrato in Fig.1.22; la prima riga é la sequenza dei riferimanti e l'ultima segnala il page fault con 'x'.

Ottimo (o Belady) Scegliendo la pagina da rimpiazzare come quella meno utilizzata nel futuro, si ha un comportamento ottimo. In Fig.1.23 viene mostrato il comportamento di questo rimpiazzamento con 3 page frame, im modo da confrontare il risultato con i precedenti.

1.3.3 L'approccio Working Set

Questo tipo di approccio é basato sull'idea di considerare le pagine che sono usate in un intervallo temporale T, la finestra del working set. La memoria fisica contiene dunque le pagine

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	4	4	3	3	3	3	1	1	1	1
x	x	x	x		x		x		x							

PF=7/17=0.41

Figura 1.23: Pagine virtuali caricate in memoria con rimpiazzamento Ottimo

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3		0	0
		1	1	3	3	3	2	2	2	2		2	2	2	2	2
x	x	x	x		x		x	x	x	x		x	x	x	x	x

PF=14/17=0.82

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	7		3	3	3	3	3	3	3	3	3	3		
	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0
		1	1	1				2	2	2	2	2	2	2	2	2
			2	2	2	2	4	4	4	4			1	1	1	1

PF=8/17=0.47

Figura 1.24: Pagine virtuali caricate in memoria secondo l'approccio Working Set per 3 e 4 page frame

virtuali contenute nel working set in ogni istante di lavoro. Questa strategia richiede dunque la definzione di due parametri: la dimensione massima del Working Set e la dimensione della finestra temporale. Questo approccio é normalmente utilizzato in tutti i sistemi operativi a memoria virtuale per i seguenti motivi: é semplice da realizzare, é semplice da controllare (basta variare la dimensione della finestra di lavoro), fornisce delle prestazioni ottimali (utilizza il rimpiazzamento LRU). In Fig.1.24 é riportato un esempio di funzionamento di questa strategia sia per 3 che per 4 page frame.

Come si vede, il numero di pagine utilizzate nella memoria fisica é al massimo pari alla finestra temporale ma puó essere anche minore, a vantaggio degli altri processi.

1.3.4 Localitá dei programmi

Tra le moltissime considerazioni sulla gestione della memoria virtuale, concludiamo queste note con una osservazione legata alla relazione tra caricamento delle pagine e programmazione in termini di localitá. Suponiamo che un programma debba azzerare una matrice di 256 x 256, e che le matrici siano caricate in memoria per righe. Che differenza c'é tra questi due programmi?

```
for(i=0;i<256;i++)
```

```
   for(j=0;j<256;j++)
     A[i][j] = 0;

for(i=0;i<256;i++)
   for(j=0;j<256;j++)
     A[j][i] = 0;
```

La differenza é nulla dal punto di vista del risultato, ma dal punto di vista della velocitá puó essere mostruosa. Se le pagine sono di 256 byte, infatti, il primo programma produce 256 page fault, e il secondo ne produce 65536, comé facile rendersi conto. Esiste infatti un campo di ricerca che si chiama **Program restructuring** che ha come scopo quello di modificare il codice sorgente dei programmi in modo tale che massimizzino la localitá.