Sistemi Operativi Teoria

Parte 3 – Gestione della memoria

Appunti a cura di Liccardo Giuseppe
Università degli Studi di Napoli "Parthenope"

Indice – Parte 3

CAPITOLO 10. Gestione della memoria 3

10.1 Gestione della gerarchia di memoria 3

10.2 Allocazione statica e dinamica della memoria (binding) 4

10.3 Esecuzione dei programmi 4

10.3.1 Associazione degli indirizzi 5

10.3.2 Indirizzi logici e indirizzi fisici 5

10.3.3 Rilocazione 6

10.3.4 Linking 6

10.4 Assegnazione della memoria 6

10.4.1 Protezione della memoria 6

10.4.2 Multiprogrammazione 7

10.4.3 Multiprogrammazione con partizionamento fisso 7

10.4.4 Swapping 8

10.4.5 Multiprogrammazione con partizionamento dinamico 8

10.5 Allocazione della memoria 8

10.6 Gestione della memoria 9

10.6.1 Uso di bitmap per la gestione della memoria 9

10.6.2 Uso di liste libere per la gestione della memoria 10

10.6.3 Tecniche di allocazione mediante free list 10

10.6.4 Buddy system e allocatori potenza del 2 10

10.6.5 Confronto tra allocatori di memoria 12

10.7 Frammentazione della memoria 12

10.7.1 Unione delle aree di memoria libera 13

10.7.2 Compattazione della memoria 13

10.8 Allocazione contigua e non contigua della memoria 13

10.6.1 Indirizzo logico, indirizzo fisico e traduzione di indirizzo 14

10.6.2 Approcci all'allocazione non contigua della memoria 14

10.6.3 Protezione della memoria 15

10.7 Paginazione 15

10.7.1 Caratteristiche delle pagine 15

10.7.2 Caratteristiche dei frame 15

10.7.3 Traduzione degli indirizzi logici in indirizzi fisici 15

10.7.4 Esempio 16

10.7.5 Dimensione ottimale delle pagine 17

10.7.6 Considerazioni finali 18

10.8 Segmentazione 18

10.8.1 Caratteristiche 18

10.8.2 Traduzione 19

10.8.3 Allocazione della memoria 19

10.9 Segmentazione con paginazione 19

10.9.1 Caratteristiche 19

10.7.5 Traduzione degli indirizzi 20

CAPITOLO 11. Memoria virtuale 21

11.1 Principi di base della memoria virtuale 21

11.2 Paginazione su richiesta 22

11.2.1 Paginazione su richiesta: concetti preliminari 23

11.2.2 Supporto hardware alla paginazione 25

11.2.3 Organizzazione pratica delle tabelle delle pagine 26

11.3 Il gestore della memoria virtuale 27

11.3.1 Panoramica sul funzionamento del gestore della memoria virtuale 28

11.4 Politiche di sostituzione delle pagine 28

11.4.1 Esempi realistici di politiche di sostituzione delle pagine 31

11.5 Allocazione dei frame a un processo 33

11.5.1 Il modello a Working Set 33

11.5.2 Working Set Clock 34

CAPITOLO 10. Gestione della memoria

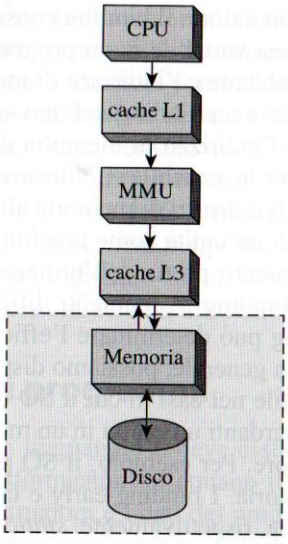
La memoria di un computer è condivisa da un grande numero di processi, per cui la gestione della memoria è tradizionalmente una parte molto importante di un SO. Le memorie diventano sempre meno costose e sempre più capienti; tuttavia, a tutt'oggi permangono le aspettative rispetto alla memoria come risorsa di un SO poiché sia la dimensione dei processi che il numero di processi che un SO deve gestire aumentano costantemente.

Le problematiche principali nella gestione della memoria sono l'uso efficiente della memoria, la protezione della memoria allocata a un processo contro gli accessi illegali da parte di altri processi, le prestazioni dei singoli processi e le prestazioni del sistema.

Questo capitolo è dedicato ai principi fondamentali della gestione della memoria. Comincia discutendo come la protezione della memoria è implementata nell'hardware utilizzando speciali registri nella CPU. Affronta le problematiche relative all'uso efficiente della memoria e le tecniche per l'allocazione e la de allocazione veloce della memoria. Successivamente vengono descritti gli approcci basati sull'allocazione non contigua della memoria chiamati *paginazione* e *segmentazione*. Il capitolo discute anche le speciali tecniche adottate dal kernel per gestire le proprie richieste di memoria in maniera efficiente.

10.1 Gestione della gerarchia di memoria

Come già detto nella parte 1, un calcolatore possiede una gerarchia di memoria: registri, cache, memoria RAM, cache del disco, hard disk, supporti di memorizzazione rimovibili, ecc.
Lo scopo di una gerarchia di memoria è quello di dare l'illusione di una memoria veloce e grande.



Levelli	Gestione	Prestazioni
Cache	Allocazione e uso gestiti in hardware	Garantire hit ratio elevati
Memoria	Allocazione gestita dal kernel ed utilizzo della memoria allocata gestita dalla libreria run-time	(1) Mantenere più processi in memoria, (2) Garantire hit ratio elevati
Disco	Allocazione ed uso gestiti dal kernel	Caricamento e memorizzazione veloci di parti dello spazio di indirizzamento di un processo

La CPU fa riferimento alla memoria più veloce, la *cache*, quando deve accedere a un'istruzione o a un dato. Se l'istruzione o il dato non è disponibile in cache, viene prelevato dal livello successivo della gerarchia di memoria, che potrebbe essere una cache più lenta oppure la *memoria RAM*. Se l'istruzione o il dato non è disponibile nemmeno al livello successivo della gerarchia, viene prelevato da un livello inferiore e così via.

L'aspetto principale delle prestazioni riguarda l'inserimento di più processi in memoria, in modo da migliorare sia le prestazioni del sistema che il servizio per l'utente. Per mantenere in memoria un elevato numero di processi, il kernel può decidere di mantenere in memoria anche solo una parte dello spazio di indirizzamento di ogni processo. A tal fine si utilizza la parte della gerarchia della memoria chiamata *memoria virtuale* che si compone della memoria RAM e dell'hard disk. Le parti dello spazio di indirizzamento di un processo non presenti in memoria vengono caricate dal disco quando necessario.

I registri, la memoria cache e la memoria RAM sono memorie volatili, cioè non sono utili per memorizzare permanentemente dati e programmi. Per questo scopo si utilizzano i dischi (o i supporti rimovibili).

10.2 Allocazione statica e dinamica della memoria (binding)

Un'operazione molto importante nella gestione della memoria è il ***binding***, ovvero il processo tramite cui si associano gli indirizzi di memoria alle entità di un programma. Un'entità di un programma, per esempio una funzione o una variabile, ha un insieme di attributi e ogni attributo ha un valore; il binding consiste nello specificare il valore di un attributo. Per esempio, una variabile in un programma ha attributi come il nome, il tipo, la dimensione. Un binding del nome specifica il nome della variabile e un binding del tipo specifica il suo tipo.

Questa operazione può essere effettuata in 3 momenti diversi:

1. *durante la compilazione*
2. *durante il caricamento*
3. *durante l'esecuzione*

Il momento esatto nel quale viene eseguito il binding può determinare l'efficienza e la flessibilità con cui l'entità del programma può essere utilizzata.

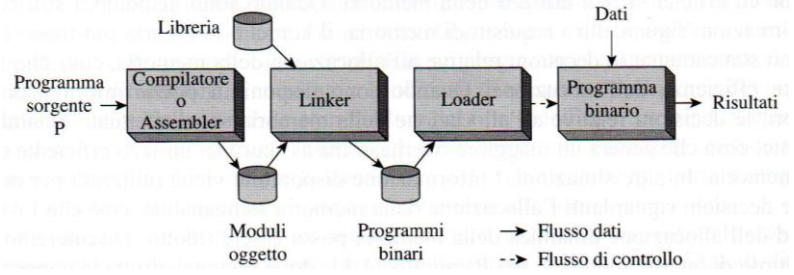
In generale possiamo distinguere tra ***binding statico*** (*early binding*) e ***binding dinamico*** (*late binding*).

Il *binding statico* è un binding eseguito prima dell'esecuzione di un programma (durante la compilazione o durante il caricamento).

Il *binding dinamico* è eseguito durante l'esecuzione di un programma.

10.3 Esecuzione dei programmi

Un programma P scritto in linguaggio L prima di essere eseguito deve subire varie trasformazioni. In particolare deve essere compilato → linkato → eseguito.
Ognuna di queste trasformazioni effettua il collegamento delle istruzioni e i dati del programma a un nuovo insieme di indirizzi.



La figura mostra uno schema delle tre trasformazioni eseguite sul programma P prima di poter essere caricato in memoria per l'esecuzione:

- ***Compilazione***: durante la compilazione, le istruzioni del codice sorgente sono tradotte in istruzioni macchina. Viene creato il modulo oggetto.
- ***Linking***: durante la fase di linking, il codice delle librerie viene incluso nel modulo oggetto. In pratica, vengono linkati al programma le funzioni che esso utilizzerà. Viene creato il codice binario.
- ***Caricamento***: durante il caricamento, il codice binario viene caricato in memoria per poter essere eseguito dalla CPU.

Solitamente le operazioni di linking e di caricamento sono effettuate da un unico programma chiamato *linker*. In alcune circostanze, la fase di caricamento viene eseguita dal *loader*, mentre il linker si occupa solamente della fase di linking.

10.3.1 Associazione degli indirizzi

Come detto, il binding può essere effettuato in 3 momenti diversi:

- ***al momento della compilazione***.
  Il binding viene eseguito in questo momento quando si sa dove il processo risiederà in memoria.
  Gli indirizzi che vengono assegnati saranno gli stessi ad ogni esecuzione del programma.
  Il codice generato prende il nome di codice assoluto.
  Se la posizione in memoria del processo dovesse cambiare, sarebbe necessaria la ricompilazione.
- ***al momento del caricamento***.
  La posizione in memoria del processo è fissa, ma è nota solo quando viene lanciata l'applicazione (non quando è compilata).
  Il codice generato è detto codice rilocabile.
  E' il loader che effettua la rilocazione.
- ***durante l'esecuzione***.
  La posizione in memoria del processo può variare durante l'esecuzione, quindi il programma può essere spostato da una zona all'altra della memoria durante l'esecuzione.

10.3.2 Indirizzi logici e indirizzi fisici

Se gli indirizzi sono generati nelle fasi di compilazione e di caricamento, allora indirizzi logici e indirizzi fisici corrispondono, per cui non è necessario effettuare la traduzione degli indirizzi.
Se gli indirizzi sono generati durante l'esecuzione, allora è necessaria la traduzione degli indirizzi.

E' importante fare la distinzione tra indirizzo logico ed indirizzo fisico.

Gli ***indirizzi logici*** sono quegli indirizzi utilizzati dalla CPU. in pratica, il processore fa riferimento alla memoria attraverso gli indirizzi logici.
Gli ***indirizzi fisici*** sono quegli indirizzi di memoria dove effettivamente risiede il dato o l'istruzione. L'accesso alla memoria fisica avviene utilizzando gli indirizzi fisici.

La traduzione da indirizzi logici a fisici, attuata durante l'esecuzione dei programmi, viene realizzata in hardware dalla MMU.

I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Se gli indirizzi sono generati in fase di esecuzione, invece, gli indirizzi logici non coincidono con quelli fisici. In questo caso ci si riferisce agli indirizzi logici col termine di ***indirizzi virtuali***.

10.3.3 Rilocazione

Abbiamo detto che gli indirizzi logici (o virtuali) non corrispondono necessariamente agli indirizzi dove sono effettivamente rese disponibili le informazioni cercate, ossia gli indirizzi fisici, per questo motivo è necessario che sia messo in atto un meccanismo che consenta di mettere in corrispondenza gli indirizzi logici con gli indirizzi fisici. Questo meccanismo è la **rilocazione**.

La ***rilocazione statica*** viene eseguita prima che abbia inizio l'esecuzione del programma. Questa politica permette di risparmiare l'overhead dovuto alla traduzione degli indirizzi, ma non è prestante in quanto non consente di cambiare l'area di memoria allocata al programma.

La ***rilocazione dinamica*** viene effettuata durante l'esecuzione del programma. Questa politica consente di cambiare l'area di memoria allocata al programma ma soffre di overhead: può essere effettuata sospendendo l'esecuzione del programma, eseguendo la rilocazione e riprendendo l'esecuzione del programma.

10.3.4 Linking

La differenza tra linker e loader è diventata sempre meno netta nei moderni SO. Tuttavia usiamo i termini nel seguente modo: un *linker* collega insieme i moduli per formare un programma eseguibile. Un *loader* carica un programma o una parte di esso in memoria per l'esecuzione.

Nel ***linking statico*** il linker collega tutti i moduli di un programma prima che cominci la sua esecuzione. Se più programmi usano lo stesso modulo di una libreria, ogni programma riceverà una propria copia del modulo, quindi diverse copie del modulo potranno essere presenti in memoria allo stesso tempo se i programmi che usano il modulo vengono eseguiti simultaneamente.
Il ***linking dinamico*** viene eseguito durante l'esecuzione di un programma binario. Il linker viene invocato quando, durante l'esecuzione, si incontra un riferimento esterno non assegnato. Il linker collega il riferimento esterno e riprende l'esecuzione del programma. Presenta molti vantaggi: i moduli non invocati durante l'esecuzione non vengono linkati; se un modulo è usato da più programmi è presente una sola volta in memoria; se si aggiorna una libreria di moduli, il programma utilizzerà automaticamente la nuova versione del modulo.

10.4 Assegnazione della memoria

La memoria centrale deve contenere sia il SO sia i vari processi utente, perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente possibile.
Solitamente la memoria centrale si divide in due partizioni, una per il SO e una per i processi utente. Nei moderni SO è richiesto che più processi utente risiedano contemporaneamente in memoria, perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda di ingresso che attendono di essere caricati in memoria.

10.4.1 Protezione della memoria

Come appena detto, in un sistema multiprogrammato, la memoria principale disponibile è di solito condivisa tra un certo numero di processi. Inoltre, viste le operazioni di swap-in e swap-out tra la memoria i dischi, esiste una certa difficoltà nel tenere separati i vari processi in memoria. In pratica, occorre assicurarsi che ogni processo abbia uno spazio di memoria separato che non interferisca con gli spazi di memoria degli altri processi. A tal fine occorre determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. La ***protezione della memoria*** è implementata mediante due registri di controllo della CPU chiamati *registro base* e *registro limite*.

Il *registro base* contiene l'indirizzo di partenza della memoria allocata a un programma, il *registro limite* contiene la dimensione della memoria allocata al programma.

L'hardware per la protezione della memoria genera un *interrupt di violazione di protezione della memoria* se un indirizzo di memoria utilizzato nell'istruzione corrente di un processo risiede al di fuori dei limiti degli indirizzi definiti dal contenuto dei registri base e limite.
E' il campo del PSW relativo all'*informazione di protezione della memoria* (MPI) che contiene i registri base e limite. Un processo utente, eseguito con la CPU in modalità utente, non può modificare il contenuto di questi registri poiché le istruzioni per caricare e salvare questi registri sono istruzioni privilegiate.

10.4.2 Multiprogrammazione

La monoprogrammazione attualmente non viene più utilizzata, a parte alcuni sistemi embedded; la maggior parte dei moderni SO consentono a diversi processi di girare contemporaneamente. In questo caso si parla di **multiprogrammazione**.

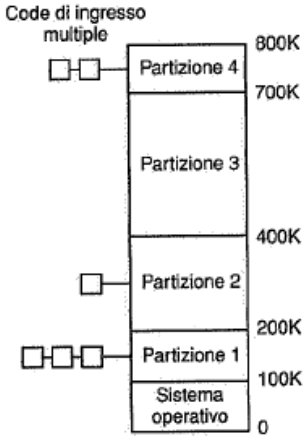
Essa consente di avere diversi processi in memoria; questo comporta che se, ad esempio, un processo è bloccato in attesa di I/O, un altro processo può utilizzare la CPU. Per questo motivo, si incrementano l'efficienza e le prestazioni di una macchina. Inoltre, la multiprogrammazione permette ad un programma di fare uso di due o più processi in modo tale da poter terminare prima le sue operazioni.

10.4.3 Multiprogrammazione con partizionamento fisso

Il modo più semplice per realizzare la multiprogrammazione è quello di dividere la memoria in *n* partizioni fisse di diversa dimensione. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato al numero di partizioni. Dal momento che le partizioni sono fisse, tutto lo spazio di una partizione non usato dal processo viene sprecato.

In questo tipo di multiprogrammazione vengono utilizzati due approcci per le code di processi che portano all'allocazione dei processi nelle varie partizioni della memoria.

CODE SEPARATE



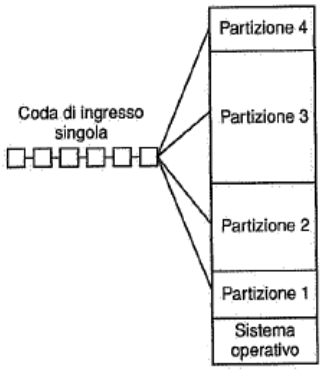
In questo approccio, ciascuna partizione ha una propria coda di ingresso.

All'avvio del sistema viene effettuata la suddivisione della memoria in partizioni fisse.

Quando arriva un processo, viene messo nella coda di ingresso della partizione più piccola che lo può contenere. Dal momento che le partizioni sono fisse, in questo schema, tutto lo spazio di una partizione non usato dal processo viene sprecato.

Lo svantaggio dell'organizzare i processi in ingresso in code separate diventa evidente nel caso in cui la coda per una partizione grande sia vuota ma quella per una partizione piccola sia piena, come nel caso delle partizioni 1 e 3 dell'immagine a sinistra: qui i piccoli processi devono aspettare per essere inseriti in memoria, sebbene la maggior parte della stessa sia libera.

CODA UNICA



In questo approccio, tutte le partizioni hanno una singola coda di ingresso "comune". La partizione della memoria viene sempre effettuata all'avvio del sistema.

Ogni qualvolta una partizione diventa libera, vi viene caricato il processo più vicino alla testa della coda che può entrare nella partizione, ed esso è quindi mandato in esecuzione.

Dal momento che non è desiderabile sprecare una partizione molto grande per far girare un processo piccolo, un'altra strategia è quella di cercare in tutta la

coda in ingresso ogni volta che si libera una partizione, e scegliere il job più grande che può entrarci. Questo algoritmo dà poca priorità ai processi piccoli, anche se di solito è preferibile dare a quest'ultimi una priorità alta.
Una possibile soluzione è quella di avere sempre a disposizione una partizione piccola che permette ai processi più piccoli di girare senza dover allocare per loro una partizione grande.

Il sistema con partizioni fisse veniva usato negli anni '80 e '90, è semplice da capire ed ugualmente semplice da implementare: i processi in arrivo vengono accodati finchè non è disponibile una partizione adatta, e a quel punto il processo viene caricato e mandato in esecuzione fino a che non termina.

10.4.4 Swapping

Nei sistemi multiutente e multiprogrammati, normalmente non vi è abbastanza memoria principale per mantenere tutti i processi attivi. Occorre, quindi, trasferire alcuni dei processi in eccesso dalla memoria sul disco, per poi introdurli in memoria successivamente.

Possono essere utilizzati due approcci generali di gestione della memoria a seconda dell'hardware disponibile: lo *swapping* o la *memoria virtuale*.

Lo ***swapping*** è la strategia più semplice e consiste proprio nel caricare interamente in memoria ogni processo, eseguirlo per un certo tempo e spostarlo nuovamente su disco. Lo spazio su disco dedicato ad "ospitare" i processi è chiamato *area di swap*.
Le operazioni svolte sono in genere:

- *swap-in*, quando si porta un processo dal disco in memoria
- *swap-out*, quando si porta un processo dalla memoria al disco

Nei sistemi multiprogrammati, lo swapping, a lungo andare, può creare molti buchi all'interno della memoria. Una delle soluzione a questo problema è la **compattazione della memoria**. Questa tecnica consiste nello spostare tutti i processi il più indietro possibile. Normalmente non viene eseguita visto che richiede un sacco di tempo di CPU.

10.4.5 Multiprogrammazione con partizionamento dinamico

Per superare alcune delle difficoltà che si incontrano nel partizionamento fisso, si può utilizzare il partizionamento dinamico. Questo genere di multiprogrammazione è quello utilizzato nei moderni SO.

Con il partizionamento dinamico, il numero, la locazione e la dimensione delle partizioni varia dinamicamente; quando un processo è caricato in memoria centrale, gli viene allocata tanta memoria quanta ne richiede. La flessibilità delle partizioni variabili migliora l'utilizzo della memoria, cioè non si hanno partizioni troppo piccole o troppo grandi come nel partizionamento statico.

Il problema delle partizioni variabili è che complicano rispetto alle partizioni fisse la procedura per tenere traccia dell'allocazione e de allocazione della memoria.

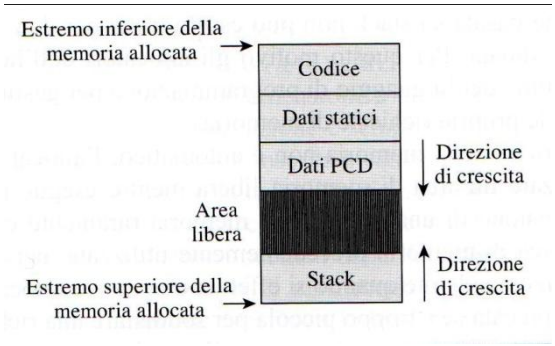
10.5 Allocazione della memoria

Quando si crea un processo o quando lo si carica (swap-in) in memoria, bisogna determinare quanta memoria deve essere allocata al processo stesso.

Se si utilizza un'***allocazione statica***, prima che il programma venga eseguito, si ipotizza la dimensione massima che può raggiungere il processo e la si alloca. Questo approccio può causare sprechi di memoria.

Nei moderni SO si utilizza un'***allocazione dinamica***. Viene allocata tanta memoria quanta realmente ne occorre al processo. Però, durante l'esecuzione di un processo, le strutture dati dello stesso possono crescere. Le componenti da tener presente per la quantità di memoria da allocare a un processo sono:

- codice e dati statici del programma
- segmento di dati PCD (dati dinamici controllati da programma)
- segmento stack



Come si può vedere dalla figura, alle componenti codice e dati statici del programma vengono allocate aree di memoria che corrispondono esattamente alla loro dimensione.
I dati PCD e lo stack condividono una sola grande area di memoria, ma eventualmente crescono in direzioni opposte se il processo "cresce" durante la sua esecuzione.
I dati PCD sono allocati partendo dalla estremità bassa di questa area mentre lo stack è allocato partendo dalla estremità alta dell'area. La memoria tra queste due componenti non è utilizzata, ma può essere usata per creare nuovi dati in entrambe le componenti: lo stack può "crescere" verso l'estremo inferiore della memoria allocata, mentre i dati PC "crescono" verso l'estremo superiore della memoria allocata.

10.6 Gestione della memoria

La memoria può essere gestita utilizzando le bitmap oppure le liste libere.

10.6.1 Uso di bitmap per la gestione della memoria

Quando la memoria viene assegnata dinamicamente, il SO deve gestirla. In generale, ci sono due modi per tenere traccia dell'utilizzo della memoria: i ***bitmap*** (*mappe di bit*) e le ***liste libere***.

Con una *mappa di bit*, la memoria viene divisa in unità di allocazione che possono essere lunghe solo poche parole o arrivare a qualche kilobyte; a ciascuna delle unità di allocazione viene associato un bit della mappa, che vale 0 se l'unità è libera e 1 se è occupata.

Le dimensioni delle unità di allocazione rappresentano una scelta di progettazione importante. Più piccola è l'unità di allocazione più grande è la mappa di bit; tuttavia anche con un'unità di allocazione di soli 4 byte, 32 bit in memoria richiedono solo 1 bit nella mappa di bit. Una mappa di bit di $32n$ bit userà n bit nella mappa di bit, così la mappa di bit prenderà solo 1/33 della memoria.

Se si sceglie un'unità di allocazione grande, allora la mappa di bit è piccola, ma viene sprecata una quantità significativa di memoria nell'ultima unità quando la dimensione dei processi non è un multiplo esatto dell'unità di allocazione.

Il problema principale, quando si utilizza l'allocazione contigua, sta nel fatto che quando si decide di caricare un processo di k unità il gestore della memoria deve esaminare la mappa di bit per trovare una sequenza consecutiva di k zeri consecutivi. Questa è un'operazione lenta per cui le bitmap sono poco utilizzate.

10.6.2 Uso di liste libere per la gestione della memoria

Quando un processo completa la propria esecuzione o rilascia la memoria allocata, il kernel riutilizza la memoria per soddisfare le richieste di altri processi.

Quando si fa uso del partizionamento della memoria, il SO conserva una tabella nella quale sono indicate le partizioni di memoria disponibili e quelle occupate. In pratica, il SO usa questa tabella per tener traccia dello stato della memoria. Questa gestione della memoria è chiamata ***gestione della memoria con liste concatenate***

Inizialmente tutta la memoria è a disposizione dei processi utente; si tratta di un grande blocco di memoria disponibile, un **buco**. Ma col passare del tempo, vengono eseguiti molti processi, dunque, solitamente, in memoria sono sparsi un *insieme* di buchi di diverse dimensioni.

Il kernel utilizza tre funzioni per garantire il riuso efficace della memoria.

PRIMA FUNZIONE: esso mantiene una ***free list*** contenente le informazioni relative a ogni area di memoria libera. Quando un processo libera parti di memoria, le informazioni relative alla memoria liberata vengono inserite nella free list. Quando un processo termina, ogni area di memoria a esso allocata e le informazioni relative vengono inserite nella free list.

SECONDA FUNZIONE: seleziona un'area di memoria per l'allocazione. Quando viene effettuata una nuova richiesta di memoria, il kernel seleziona l'area di memoria più adatta per soddisfare la richiesta.

TERZA FUNZIONE:unisce le aree di memoria libere. Due o più aree di memoria libere contigue possono essere unite per formare un'unica area libera più grande. Le aree da unire sono rimosse dalla free list e viene inserita al loro posto l'area più grande creata.

10.6.3 Tecniche di allocazione mediante free list

Come detto, al tempo t esistono in memoria n buchi di differenti dimensioni. Quando un processo è in attesa di memoria, esistono quattro tecniche per scegliere un buco libero tra quelli disponibili:

1. ***First-fit***: questo è l'algoritmo più semplice nel quale si assegna al processo il primo buco abbastanza grande da contenerlo a partire dall'inizio della lista. Quindi le dimensione dell'area di memoria (buco) sono maggiori delle dimensioni richieste dal processo; questo algoritmo divide l'area di memoria in due parti, parte viene assegnata al processo, la restante viene reinserita nella free list.
2. ***Next-fit***: questo algoritmo è un caso particolare di *First-fit*. La differenza è che in questo algoritmo si assegna al processo il primo buco abbastanza grande da contenerlo a partire dal punto in cui era terminata la ricerca precedente.
3. ***Best-fit***: questo algoritmo assegna al processo il più piccolo buco capace di contenerlo. Si può facilmente intuire che bisogna ricercare il buco scorrendo completamente la free list.
4. ***Worst-fit***: questo algoritmo assegna il buco più grande al processo. Anche qui si deve esaminare l'intera lista. Lo scopo di questo algoritmo è quello di produrre buchi più grandi, cercando di evitare la frammentazione esterna.

Per rendere più veloce l'allocazione della memoria è possibile mantenere aggiornata la *free list*, ordinandola per dimensione. In questo modo, la ricerca nel best fit è effettuata in modo più veloce.

10.6.4 Buddy system e allocatori potenza del 2

Prima abbiamo descritto alcune tecniche di allocazione della memoria, ora ne descriveremo altre due che risultano essere più veloci, e quindi più efficienti, di quelle descritte precedentemente.

Il buddy system e gli allocatori potenza del 2 eseguono l'allocazione di memoria in blocchi di piccole dimensioni predefinite. Questa caratteristica porta alla frammentazione interna poiché parte della memoria in ogni blocco allocato può essere sprecata. Tuttavia, consente all'allocatore di mantenere free list separate

per blocchi di dimensioni differenti. Questa organizzazione evita ricerche costose nella free list e porta ad allocazioni e deallocazioni veloci.

BUDDY SYSTEM

Un allocatore buddy divide e ricombina i blocchi di memoria in modo predeterminato durante l'allocazione e la deallocazione. I blocchi creati dividendo un blocco sono chiamati *blocchi buddy*. I blocchi buddy liberi vengono uniti per ricreare il blocco da cui erano stati creati. Questa operazione prende il nome di *fusione*. Secondo questo sistema, i blocchi liberi contigui che non sono buddy non sono fusi.

FUNZIONAMENTO:

Qui descriviamo il ***buddy system binario***.

In un buddy system binario, i blocchi di memoria disponibile hanno dimensione 2^K, con $L \leq K \leq U$, dove:

- 2^L – blocco allocato di dimensione più piccola
- 2^U – blocco allocato di dimensione più grande; generalmente 2^U è la dimensione dell'intera memoria disponibile per l'allocazione

Inizialmente l'intero spazio disponibile è trattato come un singolo *blocco buddy* di dimensione 2^U.
Quando deve essere allocata un'area di dimensione S:

- se $2^{U-1} < S \leq 2^U$ allora viene allocato l'intero blocco di dimensione 2^U; altrimenti il blocco viene diviso in due parti, ciascuna di dimensione 2^{U-1}
- se $2^{U-2} < S \leq 2^{U-1}$ allora uno dei due blocchi è allocato per intero; altrimenti uno dei due blocchi è diviso in due metà di dimensione 2^{U-2}
- questo processo viene ripetuto fino ad ottenere il più piccolo blocco buddy di dimensione $\geq S$

Per evitare che i blocchi buddy diventino di dimensione troppo piccola viene inserito un limite minimo, che una volta raggiunto non consente di dividere a metà l'i-esimo blocco buddy.

CONSIDERAZIONI:

L'allocatore buddy associa un tag di 1 bit ad ogni blocco per indicare se il blocco è *allocato* o *libero*. Il tag può trovarsi nel blocco stesso o può essere memorizzato separatamente.

L'allocatore mantiene molte liste di blocchi liberi; ogni free list contiene blocchi liberi di uguale dimensione, ad esempio c'è la free list che contiene i blocchi di dimensione 2^Z, poi quella dei blocchi di dimensione 2^V, poi 2^U, poi 2^T, e così via…

Quando un processo richiede un blocco di memoria di dimensione *M*; il sistema cerca la più piccola potenza del 2 che sia maggiore o uguale a *M*. Chiamiamola 2^I. Se la free list che contiene i blocchi di dimensione 2^I è non vuota, allora alloca al processo il primo blocco libero della lista e cambia il tag portandolo da *libero* a *allocato*; se la free list è vuota, vuol dire che non sono stati creati blocchi di dimensione I, ed in questo caso l'allocatore controlla la lista per i blocchi di dimensione I+1, estrae un blocco da questa lista e lo divide in due parti, in modo tale da creare due blocchi buddy per la free list dei blocchi di dimensione I. uno di questi verrà impiegato per il processo (e sarà dunque *allocato*) mentre l'altro risulterà *libero*. Se la free list per i blocchi di dimensione I+1 è vuota, allora si procede controllando la free list dei blocchi di dimensione I+2, I+3, e così via.

Quando un processo libera un blocco di memoria di dimensione 2^I, il processo è analogo, solo che invece di dividere in due parti un blocco di dimensione 2^{I+1}, non fa altro che verificare se il blocco accanto a quello liberato (il "fratello") è libero, se lo è allora i due blocchi di dimensione 2^I vengono uniti per formare un blocco di dimensione 2^{I+1}. Questo procedimento viene ripetuto fino a che non si trova che il blocco "fratello" è occupato.

In questo allocatore vengono dunque compiute molte divisioni e molte unioni.

ALLOCATORE POTENZA DEL 2
Come nel buddy system binario, le dimensioni dei blocchi di memoria sono potenze del 2 e vengono mantenute free list separate per blocchi di diverse dimensioni. Tuttavia, le similitudini con il buddy system terminano qui.

Ogni blocco contiene un header che, a sua volta, contiene l'indirizzo della free list alla quale dovrebbe essere aggiunto nel caso venga deallocato.

Quando viene eseguita una richiesta per M byte, l'allocatore dapprima controlla la free list contenente i blocchi la cui dimensione è 2^I tale che $2^I \geq M$. Se questa free list è vuota, controlla la lista contenente i blocchi con dimensione potenza del 2 successiva e così via. Un intero blocco è allocato a una richiesta, ovvero non viene eseguita nessuna divisione dei blocchi.

Inoltre, non viene eseguita la fusione dei blocchi adiacenti per creare blocchi più grandi; quando un blocco viene rilasciato, viene semplicemente restituito alla sua free list.

FUNZIONAMENTO:
I blocchi vengono creati dinamicamente sia quando l'allocatore non ha a disposizione più blocchi di una data dimensione sia quando una richiesta non può essere soddisfatta.
All'inizio, il sistema comincia creando blocchi della dimensione desiderata e inserendoli nelle free list appropriate.

10.6.5 Confronto tra allocatori di memoria

Gli allocatori di memoria possono essere confrontati sulla base della velocità di allocazione e l'uso efficiente della memoria.

VELOCITA'
Gli allocatori buddy e potenza del 2 sono più veloci degli allocatori first-fit, best-fit, next-fit e worst-fit poiché evitano le ricerche nelle free list. L'allocatore potenza del 2 è più veloce dell'allocatore buddy poiché non esegue le divisioni e le unioni dei blocchi.

EFFICIENZA DELLA MEMORIA
Gli allocatori che utilizzano le tecniche first-fit, next-fit, best-fit e worst-fit non generano frammentazione interna. Tuttavia, la frammentazione esterna limita le loro prestazioni nel caso peggiore poiché i blocchi liberi possono essere troppo piccoli per soddisfare una richiesta.

Gli allocatori buddy e potenza del 2 allocano blocchi le cui dimensioni sono potenze del 2, per cui esiste frammentazione interna a meno che le richieste non coincidano con le dimensioni dei blocchi. Inoltre, questi allocatori utilizzano memoria aggiuntiva per memorizzare i tag (buddy system) o gli header (potenza del 2) dei blocchi.

10.7 Frammentazione della memoria

Il fenomeno della ***frammentazione*** si verifica quando si inseriscono e si rimuovono processi dalla memoria RAM, oppure file da una memoria di massa (hard disk).
Se si considera la memoria RAM, ripetute aggiunte o rimozioni di sequenza di dati di dimensioni diverse all'interno della stessa, comportano una "frammentazione" dello spazio libero disponibile, che quindi non risulta più essere contiguo.

Esistono due tipi di frammentazione. La tabella seguente descrive le due forme.

Tipologia di frammentazione	Descrizione
Frammentazione esterna	Alcune aree di memoria sono troppo piccole per essere allocate.
Frammentazione interna	Viene allocata più memoria rispetto a quella richiesta, dunque una parte della memoria allocata resta inutilizzata.

Vengono generalmente individuati due tipi di frammentazione:

- **frammentazione interna**, si ha frammentazione interna quando viene allocata, ad un processo, più memoria di quanto richiesto dal processo stesso; questo problema si verifica se un allocatore gestisce blocchi di memoria di dimensioni prefissate. Questa frammentazione non influisce sulle prestazioni del sistema ma comporta uno spreco di memoria.
- **frammentazione esterna**, si ha frammentazione esterna quando un'area di memoria rimane inutilizzata poiché troppo piccola per essere allocata. Si verifica con la gestione delle partizioni dinamiche.

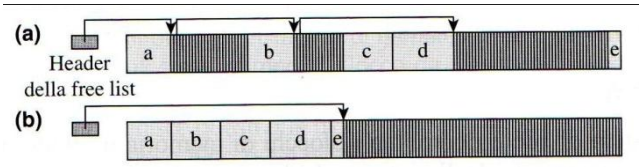
10.7.1 Unione delle aree di memoria libera

In questo paragrafo e nei prossimi mostreremo diverse tecniche per evitare o minimizzare la frammentazione della memoria.

La frammentazione esterna può essere evitata unendo le aree di memoria libera per formare aree di memoria più grandi. Ogni volta che viene liberata una nuova area di memoria, si può tentare l'*unione delle aree di memoria libere*. In pratica, si verifica se l'area successiva a quella liberata è anch'essa libera; in caso affermativo, si uniscono le aree. Questa operazione viene ripetuta fino a che non si incontra un'area allocata ad un processo. Tuttavia, questo metodo è costoso poiché coinvolge la ricerca nella free list ogni volta che si libera una nuova area di memoria.

10.7.2 Compattazione della memoria

In questo approccio tutte le aree di memoria libere vengono unite per formare un'unica area di memoria libera. Questo risultato può esser ottenuto "impacchettando" tutte le aree di memoria allocata verso un'estremità della memoria.



La compattazione non è semplice come potrebbe apparire, poiché potrebbe coinvolgere spostamenti di molti byte di memoria alla volta.

10.8 Allocazione contigua e non contigua della memoria

In passato, le vecchie architetture di computer utilizzavano l'allocazione contigua della memoria. Nell'***allocazione contigua della memoria***, ciascun processo è contenuto in una singola sezione contigua della memoria.

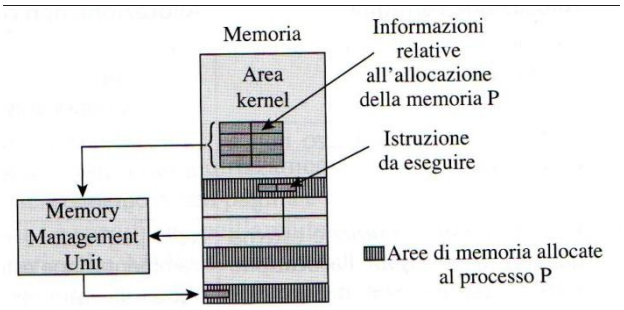
Le moderne architetture di computer utilizzano il modello di ***allocazione non contigua della memoria***, in cui le varie parti di un processo sono contenute in più aree di memoria, anche sparpagliate.
Questo modello di allocazione della memoria consente al kernel di riutilizzare le aree di memoria libere più piccole rispetto alla dimensione di un processo, per cui si può ridurre la frammentazione esterna.

Questo tipo di allocazione richiede l'uso di una ***memory management unit*** (MMU) in hardware.

10.6.1 Indirizzo logico, indirizzo fisico e traduzione di indirizzo

Come già detto in precedenza, un ***indirizzo logico*** è l'indirizzo di un'istruzione o di un dato utilizzato dalla CPU; un ***indirizzo fisico*** è l'indirizzo in memoria in cui sono presenti un'istruzione o un dato.

L'insieme di indirizzi logici in un processo costituisce lo *spazio di indirizzamento logico* del processo.
L'insieme di indirizzi fisici nel sistema costituisce lo *spazio di indirizzamento fisico* del sistema.



La figura mostra come la CPU ottiene l'indirizzo fisico corrispondente all'indirizzo logico.

Il kernel memorizza le informazioni relative alle aree di memoria allocate al processo P in una tabella e le rende disponibili alla *memory management unit* (MMU). La CPU invia gli indirizzi logici di ogni dato o istruzione utilizzati nel processo alla MMU. LA MMU utilizza le informazioni relative all'allocazione della memoria, memorizzate nella tabella, per calcolare i corrispondenti indirizzi fisici. Questo indirizzo è chiamato *indirizzo di memoria effettivo* del dato o dell'istruzione. La procedura per calcolare tale indirizzo a partire da un indirizzo logico è chiamata *traduzione dell'indirizzo*.
Quindi, chi si occupa della traduzione da indirizzo logico a indirizzo fisico è la MMU.

10.6.2 Approcci all'allocazione non contigua della memoria

Esistono fondamentalmente due approcci per implementare l'allocazione non contigua della memoria:

- **paginazione**
- **segmentazione**

Nella *paginazione* ogni processo viene diviso in parti di dimensione fissata, chiamate *pagine*.
Nella *segmentazione* un programmatore identifica le componenti di un processo chiamate *segmenti*.

Nella seguente tabella vengono confrontate l'allocazione di memoria contigua e l'allocazione di memoria non contigua:

Funzione	Allocazione contigua	Allocazione non contigua
Allocazione della memoria	Il kernel alloca una singola area di memoria a un processo.	Il kernel alloca diverse aree di memoria a un processo – ogni area contiene una componente del processo.
Traduzione dell'indirizzo	La traduzione dell'indirizzo non è necessaria.	La traduzione dell'indirizzo è eseguita dalla MMU durante l'esecuzione del programma.
Frammentazione della memoria	Si verifica frammentazione esterna quando viene usata l'allocazione first-fit, best-fit o next-fit. Si verifica frammentazione interna se l'allocazione della memoria viene eseguita in blocchi di poche dimensioni standard.	Nella paginazione non si verifica la frammentazione esterna ma può verificarsi la frammentazione interna. Nella segmentazione si verifica la frammentazione esterna, ma non si verifica la frammentazione interna.
Swapping	Se il computer non ha un registro di rilocazione, un processo swappato deve essere rimesso nell'area originariamente allocata a esso.	Le componenti di un processo swappato possono essere caricate in una qualunque parte della memoria.

10.6.3 Protezione della memoria

Ogni area di memoria allocata a un programma deve essere protetta contro le interferenze da parte di altri programmi. E' la MMU che si occupa di implementare tale funzione mediante il controllo dei limiti.

Nel tradurre un indirizzo logico ($comp_i$, $byte_i$), la MMU controlla se $comp_i$ è un indirizzo del programma e se $byte_i$ è presente in $comp_i$. Viene generato un interrupt di violazione della protezione se uno di questi controlli fallisce.

10.7 Paginazione

Abbiamo detto che, nella paginazione, ogni processo viene diviso in parti di dimensione fissata chiamate *pagine*, la cui dimensione viene definita dall'architettura del sistema.

La memoria può memorizzare un numero intero di pagine e viene partizionata in aree o blocchi di memoria, dette ***frame***, che hanno la stessa dimensione di una pagina. Quindi, ogni area di memoria (frame) è esattamente della stessa dimensione della pagina, per cui non si crea frammentazione esterna nel sistema. La frammentazione interna può crearsi poiché all'ultima pagina di un processo viene allocato un frame della dimensione di una pagina anche se è più piccolo della dimensione di una pagina.

10.7.1 Caratteristiche delle pagine

Lo spazio in memoria consiste un'organizzazione di pagine.

Ogni pagina è composta da s byte, dove s è potenza del 2. Il suo valore è specificato nell'architettura del computer. Abbiamo già detto che i processi usano indirizzi logici.

La MMU prende questo indirizzo logico e lo scompone nella coppia (pi, bi) dove:

- pi è il numero di pagina
- bi è il numero di un byte all'interno della pagina

Le pagine in un programma e i byte in una pagina sono numerati a partire da 0; per questo motivo:

- $pi \geq 0$
- $0 \leq bi \leq s$

10.7.2 Caratteristiche dei frame

L'hardware partiziona la memoria in aree chiamate ***frame***; anche questi sono numerati, in memoria, a partire da 0. Ogni frame ha la stessa dimensione della pagina.

A ogni istante, alcuni frame risultano essere allocati alle pagine dei processi, mentre altri frame risultano essere liberi. Il kernel mantiene una lista, chiamata ***lista dei frame liberi***, per tenere traccia dei frame liberi. Nel caricare un processo per l'esecuzione, il kernel consulta la lista dei frame liberi e alloca un frame libero a ogni pagina del processo.

10.7.3 Traduzione degli indirizzi logici in indirizzi fisici

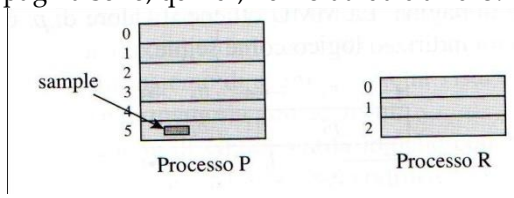
Per facilitare la traduzione dell'indirizzo, il kernel costruisce una ***tabella delle pagine*** (PT) per ogni processo. Questa tabella ha un elemento per ogni pagina del processo, che indica il frame allocato alla pagina.

Durante la traduzione di un indirizzo logico (pi, bi), la MMU utilizza il numero di pagina pi per indicizzare la tabella delle pagine del processo, ottiene il numero del frame della pagina allocata a pi e calcola l'indirizzo di memoria effettivo secondo l'equazione:

indirizzo memoria effettivo di (pi, bi) = indirizzo di avvio del frame allocato a pi + numero dei byte di bi nella pagina pi

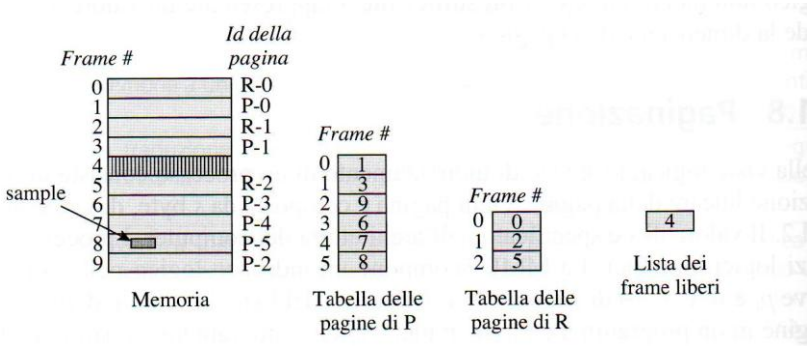
10.7.4 Esempio

Si considerino due processi P ed R in un sistema che usa una pagina di dimensione 1 KB. I byte in una pagina sono, quindi, numerati da 0 a 1023.



Il processo P ha indirizzo di avvio 0 e dimensione 5500 byte. Dunque avrà 6 pagine numerate da 0 a 5; le pagine 0, 1, 2, 3 e 4 avranno 1024 byte, per un totale di 5120 byte; mentre la pagina 5 avrà solo 380 byte, i restanti sono sprecati (causano frammentazione interna).
Se un dato *sample* ha indirizzo 5248, ovvero 5 x 1024 + 128, vuol dire che si trova nella pagina 5; la MMU vede il suo indirizzo come la coppia (5, 128).

Il processo R ha dimensione 2500 byte. Dunque avrà 3 pagine, numerate da 0 a 2; le pagine 0 e 1 avranno 1024 byte, mentre la pagina 2 avrà 452 byte.



La figura mostra la vista fisica dell'esecuzione dei processi P ed R.

Visto che i frame hanno la stessa dimensione delle pagine, allora avranno dimensione pari a 1 KB. Il computer ha una memoria di 10 KB per cui i frame sono numerati da 0 a 9. Sei frame sono occupati dal processo P e tre frame sono occupati dal processo R.

Le pagine contenute nei frame sono mostrate come P-0, … , P-5 per P; come R-0, … , R-2 per il processo R.

La lista dei frame liberi contiene un solo elemento, perché il solo frame 4 risulta libero.
La tabella delle pagine di P riporta il frame allocato a ogni pagina di P.
Analogo discorso vale per la tabella delle pagine di R che riporta il frame allocato a ogni pagina di R.

Supponiamo che ad esempio, il processo P usi l'indirizzo logico (5, 128) della variabile *sample* durante la sua esecuzione, questo sarà tradotto nell'effettivo indirizzo di memoria usando l'equazione scritta in precedenza:

indirizzo di memoria effettivo di (5, 128)
= indirizzo di avvio del frame allocato a 5 + numero dei byte, cioè "frame #8 + 128"
= 8 x 1024 + 128
= 8320

La dimensione di una pagina è potenza di 2, così il calcolo dell'indirizzo effettivo è eseguito tramite ***concatenazione di bit***, che è molto più veloce dell'addizione appena vista.

Usiamo la seguente notazione per descrivere come viene eseguita la traduzione dell'indirizzo:
s dimensione di una pagina;
l lunghezza di un indirizzo logico (ovvero, numero di bit in esso contenuti);

l_p lunghezza dell'indirizzo fisico;
n_b numero di bit utilizzati per rappresentare il byte in un indirizzo logico;
n_p numero di bit utilizzati per rappresentare il numero di pagina in un indirizzo logico;
n_f numero di bit utilizzati per rappresentare il numero di frame in un indirizzo fisico.

La dimensione di una pagina è s, che è una potenza del 2. n_b viene scelto in modo tale che $s = 2^{n_b}$.

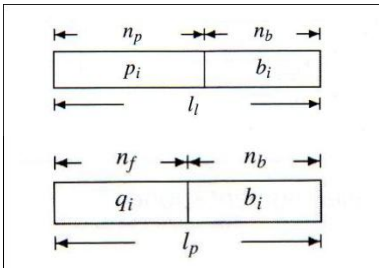
Preso un indirizzo logico costituito da l_l bit, alcuni di questi bit (quelli più significativi) saranno usati per rappresentare il numero di pagina dell'indirizzo logico, gli altri bit (quelli meno significativi) saranno usati per rappresentare lo scostamento (in byte) in quella pagina.

In pratica:
- gli n_p bit più significativi danno la pagina *pi*
- gli n_b bit meno significativi danno lo scostamento *bi*

Sia la pagina *pi* allocata nel frame *qi*, poiché hanno la stessa dimensione, allora l'indirizzo fisico (*qi, bi*) sarà costituito da:
- gli n_f bit più significativi danno il frame *qi*
- gli n_b bit meno significativi danno lo scostamento *bi*

Il numero del frame viene preso dalla tabella delle pagine e corrispondeva agli n_f bit. A questi bit vengono poi concatenati gli altri n_b bit che rappresentano lo scostamento.

In definitiva, basta sostituire la sequenza di bit usata per rappresentare la pagina con la sequenza di bit usata per rappresentare il frame corrispondente.



La MMU ottiene questo indirizzo semplicemente concatenando *qi* e *bi* per ottenere un numero di bit l_p.

10.7.5 Dimensione ottimale delle pagine

La dimensione delle pagine è un parametro molto importante del sistema.
Se si usano pagine piccole si ha il vantaggio di avere uno spreco di memoria minore (a causa dell'inutilizzo parziale dell'ultima pagina).
Se si usano pagine grandi si ha il vantaggio di avere tabelle delle pagine piccole. Inoltre, un altro vantaggio è legato al fatto che il tempo di trasferimento necessario per trasferire una pagina grande è simile a quello necessario per trasferire una pagina piccola.

Siano:
- *S* la dimensione media dei processi
- *P* la dimensione delle pagine
- *E* la dimensione di un elemento nella tabella delle pagine

Allora:
- *S/P* sarà il numero di pagine in memoria
- *SE/P* sarà la dimensione della tabella
- *P/2* sarà la memoria sprecata (è una stima)

Il costo sarà:

$COSTO = SE/P + P/2$

Derivando rispetto a P, si ottiene che la dimensione ottimale della pagina è:

P = sqrt (2SE)

Solitamente la dimensione delle pagine và da 1 KB a 64 KB.

10.7.6 Considerazioni finali

In pratica, se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è di 2^n unità di indirizzamento, allora m-n bit più significativi di un indirizzo logico indicano il numero di pagina, e gli n bit meno significativi indicano lo scostamento di pagina.

La paginazione non è altro che una forma di rilocazione dinamica: a ogni indirizzo logico l'architettura di paginazione fa corrispondere un indirizzo fisico. Con la paginazione si evita la frammentazione esterna ma tuttavia si può avere la frammentazione interna visto che lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, quindi l'ultimo blocco di memoria assegnato può essere non completamente pieno.
Attualmente la dimensione delle pagine è compresa tra 4 KB e 8 KB; ciascun elemento della tabella delle pagine di solito è lungo 4 byte, ma questa dimensione può variare.

10.8 Segmentazione

Nella segmentazione un programma è diviso in vari segmenti di diversa dimensione. Ogni segmento rappresenta un'entità logica del programma come una funzione o una struttura dati.

Visto che la dimensione dei segmenti è variabile, allora può essere generata frammentazione esterna. Per questo motivo il sistema deve usare tecniche di riutilizzo della memoria come la *first-fit* o la *best-fit*.

10.8.1 Caratteristiche

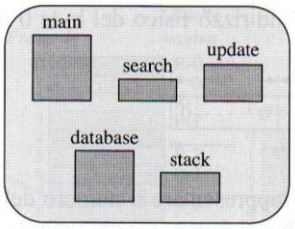
La segmentazione permette una facile condivisione del codice, dei dati e delle funzioni di un programma proprio perché questi sono organizzati in segmenti.
In pratica, un processo viene visto come un insieme di segmenti, che però sono sparsi in memoria. Per questo motivo indirizzi logici e indirizzi fisici non corrispondono.
Ogni indirizzo logico è rappresentato nella forma (*si*, *bi*), dove:

- *si* è l'ID di un segmento
- *bi* è lo scostamento in byte all'interno del segmento

Questa organizzazione permette al programmatore di vedere la memoria come un insieme di segmenti che possono essere di dimensione diversa, e soprattutto dinamica; offre dei vantaggi al programmatore rispetto alla paginazione:

1. semplifica la gestione di strutture dati che crescono, infatti il SO può espandere il segmento a seconda dei casi
2. permette la modifica e la ricompilazione indipendente dei programmi, senza richiedere che si faccia il link con tutto l'insieme dei programmi per ricaricarli
3. si presta alla condivisione dei processi; un programmatore può allocare un programma di utilità in un segmento, che può essere accessibile agli altri processi
4. si presta alla protezione; poiché un segmento può contenere un insieme ben definito di programmi o dati, il programmatore o l'amministratore di sistema può assegnare privilegi in modo conveniente

10.8.2 Traduzione



Processo Q

Nome	Dimensione	Indirizzo
main	476	23500
database	20240	32012
search	378	76248
update	642	91376
stack	500	54500

Tabella dei segmenti di Q

La figura mostra come il kernel gestisce il processo Q.

La parte sinistra mostra la visione logica del processo Q. Per facilitare la traduzione dell'indirizzo, il kernel costruisce una ***tabella dei segmenti*** per Q. Ogni elemento in questa tabella mostra la dimensione di un segmento e l'indirizzo dell'area di memoria a esso allocata. Il campo *dimensione* viene utilizzato per garantire la protezione della memoria.

Questa tabella viene usata dalla MMU per effettuare la traduzione degli indirizzi; visto che i segmenti non hanno dimensioni fissate, la traduzione non può essere realizzata come nella paginazione, cioè attraverso concatenazione di bit, ma il calcolo dell'indirizzo di memoria effettivo per l'indirizzo logico (*si*, *bi*) consiste nel sommare *bi* all'indirizzo di avvio del segmento *si*.

Esempio:
Consideriamo la procedura *get_sample* del segmento *update*. Supponiamo che questa stessa procedura abbia numero di byte 232.
Per ottenere il suo indirizzo di memoria effettivo occorre sommare 91376 + 232 = 91608.

10.8.3 Allocazione della memoria

L'allocazione della memoria per ogni segmento viene eseguita come nell'allocazione contigua della memoria. Il kernel mantiene una lista delle aree di memoria libera.

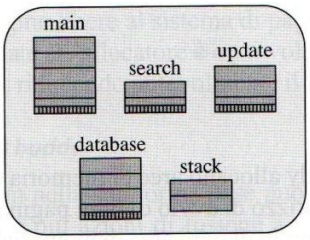
Nel caricare un processo, il kernel cerca in questa lista per eseguire l'allocazione first-fit o best-fit per ogni segmento del processo.
Quando un processo termina, le aree di memoria allocate ai suoi segmenti vengono aggiunte alla free list.

La frammentazione esterna può verificarsi poiché i segmenti hanno dimensioni differenti.

10.9 Segmentazione con paginazione

Un approccio ibrido, chiamato ***segmentazione con paginazione*** combina le caratteristiche della segmentazione e della paginazione. Cioè facilita la condivisione del codice, dei dati e dei moduli di programma tra i processi senza incorrere nella frammentazione esterna; tuttavia si verifica la frammentazione interna così come si verifica nella paginazione.

10.9.1 Caratteristiche



Processo O

Nome	Dimensione	Indirizzo della tabella delle pagine
main	476	
database	20240	
search	378	
update	642	
stack	500	

Tabella dei segmenti di Q

Secondo tale approccio, ogni segmento di un programma viene paginato separatamente; di conseguenza viene allocato a ogni segmento un numero intero di pagine.

Per ogni segmento viene creata una tabella delle pagine e l'indirizzo di questa tabella viene memorizzato nell'elemento relativo al segmento nella tabella dei segmenti.
Come si può vedere nella figura, ogni segmento viene paginato separatamente, per cui si verifica frammentazione interna nell'ultima pagina di ogni segmento. Ogni elemento della tabella contiene l'indirizzo della tabella delle pagine e la dimensione del segmento (utile per la protezione della memoria).

10.7.5 Traduzione degli indirizzi

La traduzione di un indirizzo logico (*si*, *bi*) avviene in due fasi.

Nella prima fase, si cerca l'elemento di *si* nella tabella dei segmenti e viene estratto l'indirizzo della sua tabella delle pagine. Il numero del byte *bi* è ora diviso in una coppia (ps_i, bp_i), dove:

- ps_i è il numero di pagina nel segmento *si*
- bp_i è il numero del byte nella pagina *pi*

Nella seconda fase, si completa il calcolo dell'indirizzo effettivo così come nella paginazione, ovvero si ottiene il numero del frame di ps_i, e bp_i viene concatenato ad esso per ottenere l'indirizzo effettivo.

CAPITOLO 11. Memoria virtuale

La ***memoria virtuale*** è una parte della gerarchia di memoria composta da una memoria e da un disco.

Durante l'esecuzione di un processo, alcune componenti del suo spazio di indirizzamento (codice e dati) si trovano in memoria, mentre altre risiedono su un disco e vengono caricate in memoria solo quando necessario durante l'esecuzione del processo.
Questa soluzione fa sì che la richiesta totale di memoria di un processo possa superare la dimensione della memoria del sistema. Permette, inoltre, che un maggior numero di processi risiedano in memoria contemporaneamente, perché ognuno di loro occupa meno memoria della propria dimensione.

Le prestazioni di un processo dipendono dalla percentuale delle sue porzioni che devono essere caricate in memoria dal disco.

In questo capitolo, verranno studiate le tecniche usate dal kernel per assicurare l'esecuzione efficiente di un processo e le buone prestazioni di sistema.

11.1 Principi di base della memoria virtuale

La memoria virtuale è ciò che il suo nome indica: è un'illusione di una memoria più grande di quella reale, ossia della RAM del computer. Il kernel implementa tale illusione tramite una combinazione di mezzi hardware e software, cioè attraverso la **MMU** (HW) e il **gestore della memoria virtuale** (SW).

Sono due le operazioni fondamentali nel funzionamento della memoria virtuale che utilizzano la paginazione: la ***traduzione degli indirizzi*** e il ***caricamento delle pagine su richiesta***.

TRADUZIONE DEGLI INDIRIZZI
Come già detto, la memoria virtuale consente a un processo di caricare in memoria, di volta in volta, solo le sue parti necessarie. Essa si basa sul modello di ***allocazione di memoria non contigua***, per tale motivo:

- le parti di un processo (*componenti*) possono essere caricate in aree di memoria non adiacenti
- l'indirizzo di ciascun operando o istruzione di un processo è un *indirizzo logico* del tipo (p_i, b_i); è la MMU che traduce questo indirizzo logico nell'indirizzo di memoria effettivo in cui si trova l'operando o l'istruzione

Questo modello di allocazione riduce la frammentazione di memoria, poiché un'area libera di memoria può essere riutilizzata anche se non è abbastanza grande da contenere l'intero spazio di indirizzamento di un processo. In questo modo possono essere caricati in memoria più processi utente.

CARICAMENTO SU RICHIESTA DI COMPONENTI DI UN PROCESSO
Il *gestore della memoria virtuale* si occupa di caricare in memoria solo una parte del processo.

Quando un processo viene mandato in esecuzione, il gestore della memoria virtuale carica solo quella porzione che contiene l'*indirizzo di start* del processo, cioè l'indirizzo dell'istruzione con cui la sua esecuzione comincia. Successivamente, carica altre porzioni del processo solo quando necessarie. Questa tecnica prende il nome di ***caricamento su richiesta***.
Per mantenere contenuta la quantità di memoria occupata da un processo, occasionalmente il gestore rimuove dalla memoria le componenti inutilizzate del processo.

PAGINAZIONE E SEGMENTAZIONE
E' possibile implementare la memoria virtuale attraverso la paginazione o mediante la segmentazione.

Nella *paginazione*, ciascuna porzione di spazio di indirizzamento è chiamata *pagina*; tutte le pagine hanno uguale dimensione, potenza di due. La dimensione della pagina è definita dall'hardware e la suddivisione delle pagine viene eseguita automaticamente.

Nella *segmentazione*, ogni porzione di uno spazio di indirizzamento è chiamato *segmento*. Un programmatore dichiara alcune entità logiche rilevanti (per esempio, strutture dati o oggetti) in un processo come segmenti. Quindi l'identificazione delle porzioni è a carico del programmatore, e i segmenti possono avere dimensioni differenti. Alcuni sistemi usano un approccio ibrido segmentazione con paginazione.

La seguente tabella mette a confronto la paginazione con la segmentazione:

Oggetto	Confronto
Concetto	Una pagina è una porzione a misura fissa di uno spazio di indirizzamento di un processo identificato dall'hardware della memoria virtuale. Un segmento è un'entità logica all'interno di un programma, ossia una funzione, una struttura dati o un oggetto. I segmenti sono identificati dal programmatore.
Dimensione delle componenti	Tutte le pagine sono della stessa dimensione. I segmenti possono essere di dimensioni differenti.
Frammentazione esterna	Non si verifica durante la paginazione perché la memoria è divisa in page frame la cui dimensione è uguale alla dimensione delle pagine. Si verifica nella segmentazione perché un'area libera di memoria può essere piccola per ospitare un segmento.
Frammentazione interna	Si verifica nell'ultima pagina di un processo durante la paginazione. Non si verifica nella segmentazione perché un segmento è allocato in un'area di memoria la cui dimensione è uguale alla dimensione del segmento.
Condivisione	La condivisione (sharing) di pagine è realizzabile subordinatamente ai vincoli sulla condivisione delle pagine di codice descritti successivamente nel Paragrafo 12.6. La condivisione di segmenti è possibile liberamente.

11.2 Paginazione su richiesta

Riprendiamo un concetto già discusso nel capitolo precedente, cioè la traduzione degli indirizzi nella paginazione.

Si considera che un processo sia composto da pagine, numerate da 0 in avanti. Ogni pagina è di s byte, dove s è una potenza di 2.
Si considera che la memoria di un computer sia composta da *frame di pagina*, dove un frame di pagina è un'area di memoria che ha la stessa dimensione di una pagina. Questi frame sono numerati da *0* a *#frame-1* dove *#frame* è il numero dei frame di pagina di memoria.

Dunque lo spazio di indirizzamento fisico è composto da indirizzi da *0* a *#frame x s – 1*.

In ogni momento, un frame può essere libero oppure può contenere una pagina di un processo.

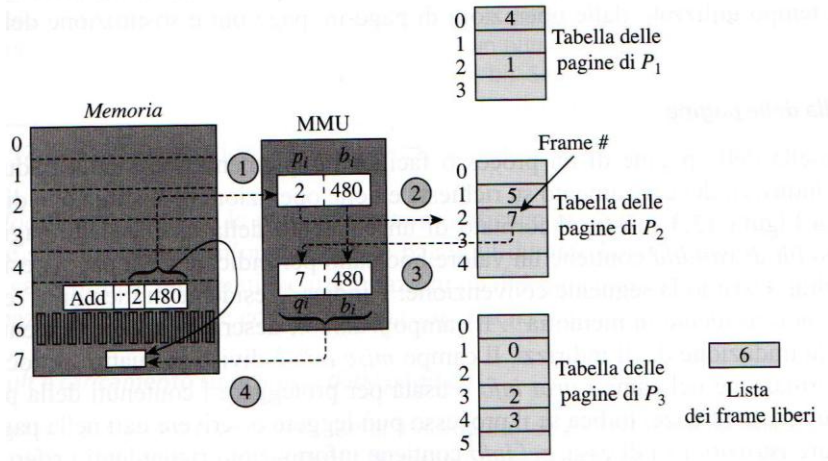
Ogni indirizzo logico usato in un processo è rappresentato da una coppia (*pi, bi*) dove *pi* è un numero di pagina e *bi* è il numero di byte nella pagina *pi* (il suo valore è $0 \leq bi \leq s$).

A partire dall'indirizzo logico, si può calcolare l'indirizzo di memoria effettivo, nel modo che segue:

indirizzo memoria effettivo di (pi, bi) = indirizzo di avvio del frame allocato a pi + numero dei byte di bi nella pagina pi

La dimensione di una pagina è potenza di 2, così il calcolo dell'indirizzo effettivo è eseguito tramite concatenazione di bit, che è molto più veloce dell'addizione.

Esempio:



La figura mostra uno schema della memoria virtuale che usa la paginazione in cui si assume che la dimensione di pagina sia 1 KB, ossia 1024 byte.

La memoria (sulla sinistra) contiene 8 frame numerati da 0 a 7.

Per ciascuno dei tre processi P1, P2 e P3 esiste una *tabella delle pagine*, nella quale sono contenute le informazioni sull'allocazione di memoria. In particolare, ogni elemento di una tabella contiene il numero del frame in cui una pagina risiede.
Come si può vedere dalle tabelle, i processi P1, P2 e P3 hanno alcune delle loro pagine in memoria:

- P1 ha le sue pagine 0 e 2 nei frame 4 e 1 della memoria
- P2 ha le sue pagine 1 e 2 nei frame 5 e 7 della memoria
- P3 ha le sue pagine 1, 3 e 4 nei frame 0, 2 e 3 della memoria

Un'ulteriore tabella, chiamata *tabella dei frame liberi* contiene la lista dei frame liberi in memoria.

Nella figura sono rappresentati anche i passi per la traduzione degli indirizzi: si sta traducendo l'indirizzo logico della pagina 2 del processo P2 nell'effettivo indirizzo di memoria. Questa operazione è compiuta dal MMU, che fa uso della tabella delle pagine di P2 per effettuare quest'operazione. La MMU:

1. Vede l'indirizzo dell'operando 2528 come la coppia (2, 480) visto che 2528 = 2 x 1024 + 480
2. Accede all'elemento relativo alla pagina 2 nella tabella delle pagine di P2
3. Questo elemento contiene il numero di frame 7, quindi la MMU calcola l'indirizzo effettivo 7 x 1024 + 480
4. Utilizza questo indirizzo per accedere alla memoria e, in effetti, accede al byte 480 del frame 7

11.2.1 Paginazione su richiesta: concetti preliminari

La memoria è divisa in parti chiamate *frame*, la cui dimensione è uguale alla dimensione delle pagine.

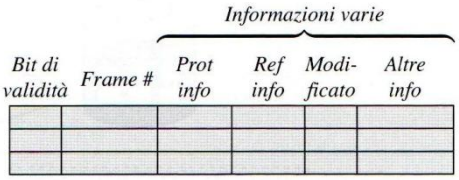
La ***paginazione su richiesta***, o *caricamento su richiesta di pagina*, è costituita da quelle azioni che effettua il gestore della memoria virtuale per tenere aggiornate le tabelle delle pagine dei vari processi.

TABELLA DELLE PAGINE
Il gestore della memoria virtuale tiene una *tabella delle pagine* per ciascun processo per "mappare" le pagine virtuali sui frame, cioè per indicare in quale frame di memoria si trovano le varie pagine.

Questa tabella facilita l'implementazione della traduzione degli indirizzi, del caricamento su richiesta e delle operazioni di sostituzione delle pagine.

Ciascuna tabella contiene un elemento (riga) per ogni pagina allocata al processo.
I campi (colonne) sono molteplici e contengono svariate informazioni.

Le dimensioni di una tabella variano da computer a computer, ma nei computer di ultima generazione sono di 32 o 64 bit.



Campo	Descrizione
Bit di validità	Indica se la pagina descritta dall'elemento attualmente è presente in memoria. Questo bit è anche chiamato bit di presenza.
Frame #	Indica quale frame di memoria è occupato dalla pagina.
Prot info	Indica le modalità di utilizzo del contenuto della pagina, se in scrittura, lettura o esecuzione.
Ref info	Fornisce informazioni riguardanti i riferimenti fatti alla pagina mentre era in memoria.
Modificato	Indica se la pagina è stata modificata mentre era in memoria, ovvero se è dirty. Questo campo è un singolo bit, detto dirty bit.
Altre info	Altre informazioni utili relative alla pagina, per esempio, la sua posizione nello spazio di swap.

Tra le informazioni più importanti occorre ricordare:

- il *bit di validità*, che indica se la pagina è presente o meno in memoria (0 non presente – 1 presente)
- il *#frame*, che indica il frame di memoria occupato dalla pagina
- il campo *modificato*, che indica se la pagina è stata modificata o meno, cioè se è *dirty*; quando una pagina viene modificata (cioè è dirty) deve essere copiata sul disco (page-out)

PAGE FAULT E CARICAMENTO SU RICHIESTA DELLE PAGINE

Abbiamo già parlato più volte della traduzione degli indirizzi effettuata dalla MMU. La tabella seguente descrive in modo sintetico i passi per la traduzione degli indirizzi:

Passo	Descrizione
1. Ricavare numero di pagina e numero di byte in una pagina	Un indirizzo logico è visto come una coppia (p_i, b_i), dove b_i è dato dagli n_b bit meno significativi dell'indirizzo e p_i dato dagli n_p bit più significativi (Paragrafo 11.8).
2. Ricercare nella tabella delle pagine	p_i è usato per indicizzare la tabella delle pagine. Un page fault viene generato se il *bit di validità* dell'elemento della tabella delle pagine contiene uno 0, ossia se la pagina non è presente in memoria.
3. Formare l'indirizzo effettivo di memoria	Il campo *frame #* dell'elemento della tabella delle pagine contiene un numero di frame rappresentato con un numero a n_f bit. Se è concatenato con b_i si ottiene l'effettivo indirizzo di memoria del byte.

Supponiamo che la CPU voglia usare un certo dato o una certa istruzione.
La MMU, nell'effettuare la traduzione degli indirizzi verifica, facendo uso della tabella delle pagine, se quella pagina è presente o meno in memoria (fa riferimento al campo *bit di validità*).
Se la pagina non è presente, si genera un *page fault* e il gestore della memoria virtuale carica la pagina in memoria e aggiorna la tabella delle pagine.

Infatti alcune pagine virtuali non possono avere corrispondenza in memoria fisica, il bit di validità è usato proprio per indicare se la pagina è presente o meno in memoria. Si ha **page fault** quando un processo tenta di usare una pagina non mappata, cioè non presente in memoria.
Il controllo del bit di validità è effettuato al passo 2: la MMU controlla il bit di validità della tabella delle pagine; se è 0 vuol dire che la pagina non è presenta in memoria, dunque la MMU genera un interrupt chiamato ***page fault***, che è un interrupt di programma. A causa dell'interrupt viene invocato il gestore della memoria virtuale che carica la pagina in memoria e aggiorna la tabella delle pagine di quel processo. In pratica, MMU e gestore della memoria virtuale interagiscono per decidere *quando* una pagina di un processo deve essere caricata in memoria.

OPERAZIONI DI PAGE-IN, PAGE-OUT E SOSTITUZIONE DELLE PAGINE

Quando un processo vuole usare una pagina non presente in memoria e ci sono frame liberi in memoria, il gestore della memoria virtuale carica la pagina dallo spazio di swap (sul disco) in memoria. Questa operazione prende il nome di ***page-in***.

Quando il gestore della memoria virtuale decide di rimuovere una pagina dalla memoria, copia la pagina nello spazio di swap (sul disco) Questa operazione prende il nome di ***page-out***.

Le operazioni di page-in e page-out costituiscono il *page I/O*, che è differente dalle operazioni di I/O dei processi in quanto, quando si eseguono operazioni di page I/O lo stato del processo diventa *blocked*.

Quando un processo vuole usare una pagina non presente in memoria e non ci sono frame liberi in memoria è necessario ricorrere alla ***sostituzione delle pagine***. La sostituzione di pagina diventa necessaria quando si verifica un page fault e non ci sono frame liberi in memoria.
Questa operazione viene svolta dal gestore della memoria virtuale che:

- seleziona una delle pagine attualmente in memoria utilizzando un *algoritmo di sostituzione delle pagine*
- modifica il campo *bit di validità* di questa pagina, settandolo a 0 (non presente); facendo questa operazione, la pagina diventa *dirty*
- effettua un'operazione di *page-out* per copiare la pagina sul disco in quanto la pagina è dirty
- effettua un'operazione di *page-in* per caricare in memoria, nel frame appena liberato, la pagina richiesta dal processo
- una volta conclusa l'operazione di page-in, modifica il campo *bit di validità* della pagina appena caricata, settandolo ad 1 (presente)

N.B. Durante queste operazioni, lo stato del processo viene impostato a *blocked*; solo dopo l'operazione di page-in viene ripristinata l'esecuzione del processo. Nel frattempo, la CPU è assegnata ad altri processi.

11.2.2 Supporto hardware alla paginazione

PROTEZIONE DELLA MEMORIA

Un interrupt di *violazione di protezione della memoria* viene generato quando un processo tenta di accedere a una pagina non esistente oppure se tenta di accedere a una pagina a cui non può accedere.

Inoltre, per gestire i privilegi di accesso di un processo a una pagina, occorre far riferimento al campo *prot_info* della tabella delle pagine del processo. In pratica, la MMU controlla se il contenuto della pagina alla quale si vuole accedere permette l'accesso in modalità scrittura, lettura o esecuzione. Nel caso in cui la modalità di accesso non è consentita, viene generato un interrupt.

BUFFER DI TRADUZIONE DEGLI INDIRIZZI

Le tabelle delle pagine vengono tenute in memoria per le loro grandi dimensioni, ma ciò può penalizzare fortemente le prestazioni, infatti una singola istruzione può fare riferimento a più indirizzi.
Se a questo aggiungiamo il fatto che i programmi tendono a fare la maggior parte dei riferimenti a un piccolo numero di pagine, allora si può concludere che solo una piccola parte degli elementi nella tabella delle pagine viene letta con maggior frequenza.

Una possibile soluzione per aumentare le prestazioni è quello di usare il ***TLB*** (Tanslation Look-aside buffer), o *memoria associativa*, che è una memoria cache veloce contenuta nella MMU che permette di velocizzare la traduzione degli indirizzi.
Questo TLB contiene porzioni della tabella delle pagine, cioè solo pochi elementi di quest'ultima. In particolare contiene gli elementi che sono stati usati più di recente. Di ogni elemento memorizza alcune informazioni principali come il numero il numero di pagina, il numero di frame, il bit di modifica e il campo per la protezione (*prot_info*).

FUNZIONAMENTO:
Quando alla MMU arriva un indirizzo virtuale da tradurre, l'hardware controlla prima se il relativo numero di pagina virtuale è presente nel TLB, confrontandolo simultaneamente (in parallelo) con tutti gli elementi.

Page hit
Se c'è corrispondenza, ovvero se viene trovato un elemento con lo stesso numero di pagina, la MMU completa la traduzione prendendo il numero del frame direttamente dal TLB, senza il bisogno di dover accedere alla tabella delle pagine.

Page miss
Se non c'è corrispondenza, ovvero se il numero di pagina virtuale non è presente nel TLB, la MMU esegue una normale ricerca nella tabella delle pagine; se la pagina è presente nella tabella delle pagine competa la traduzione, altrimenti genera un page fault, che attiva il gestore della memoria virtuale per caricare la pagina in memoria.
E' importante tener aggiornati gli elementi contenuti all'interno della TLB. Quando si verifica un page miss, la MMU continua la ricerca nella tabella delle pagine, quando ha concluso la traduzione degli indirizzi, scarica uno degli elementi del TLB e lo rimpiazza con l'elemento della tabella delle pagine appena trovato.
La scelta dell'elemento da sostituire viene effettuata usando un apposito algoritmo di sostituzione.

SUPPORTO PER LA SOSTITUZIONE DELLE PAGINE
Il gestore della memoria virtuale ha bisogno di due tipi di informazioni per minimizzare i page fault e il numero delle operazioni di page-in e page-out durante la sostituzione di una pagina:

1. l'ultimo istante di utilizzo di una pagina
2. se una pagina è *dirty*, ossia se un'operazione di scrittura è stata eseguita su qualche byte della pagina

L'istante di ultimo utilizzo indica quanto di recente una pagina è stata usata da un processo; è utile per selezionare una pagina da sostituire. Per questo scopo è utilizzato un campo di un singolo bit, *ref_info*.

Per indicare se una pagina è *dirty*, viene usato il campo *bit di modifica*. Se una pagina è *clean* vuol dire che la sua copia in memoria è ancora attuale e quindi non è necessaria alcuna operazione di page-out. Se una pagina è *dirty*, è necessaria un'operazione di page-out in quanto la sua copia è vecchia.

11.2.3 Organizzazione pratica delle tabelle delle pagine

Abbiamo detto che le tabelle delle pagine sono memorizzate nella memoria. Negli attuali computer, vengono mandati in esecuzione molti processi contemporaneamente, per questo motivo, gran parte della memoria RAM potrebbe essere usata per la memorizzazione delle tabelle delle pagine dei processi.
Per risolvere questo spreco, derivante dalla presenza costante in memoria di tabelle delle pagine molto grosse, vengono seguiti fondamentalmente due approcci per ridurre la dimensione in memoria assegnata alle tabelle delle pagine:

- *Tabella delle pagine invertita*
- *Tabella delle pagine multilivello*

In entrambi gli approcci, il TLB viene usato per ridurre il numero di riferimenti di memoria necessari per eseguire la traduzione degli indirizzi.

TABELLA DELLE PAGINE INVERTITA
La ***tabella delle pagine invertita*** (IPT) contiene un elemento per ogni frame (pagina fisica) di memoria. Ciascun elemento contiene il numero di pagina che occupa il frame e l'ID del processo. La tabella viene così denominata perché l'informazione in essa contenuta è invertita rispetto alla classica tabella delle pagine, che contiene un elemento per ogni pagina.

La dimensione di una IPT non dipende dal numero e dalle dimensioni dei processi, ma dipende dalla dimensione della memoria. Quindi questo approccio riduce la quantità di memoria occupata.

In questo approccio, però, è più difficile effettuare la traduzione degli indirizzi da indirizzo logico a indirizzo fisico. Supponiamo di avere un processo R. Abbiamo detto che ogni elemento di una IPT contiene due informazioni: l'ID del processo e il numero di pagina che occupa quel frame. Quindi una coppia (*R*, *pi*)

nella f_i –esima entry (riga) della tabella invertita indica che il frame *fi* è occupato dalla pagina *pi* di un processo R.
Se si vuole effettuare la traduzione di questo indirizzo, non si dovrà cercare più un indice ma una coppia di elementi (*R, pi*) nella tabella IPT. Per velocizzare questa ricerca, vengono usate la TLB, e una *hash table* nel caso in cui si verifichi un *page miss*, cioè se la pagina cercata non è nel TLB.

TABELLA DELLE PAGINE MULTILIVELLO
La ***tabella delle pagine multilivello*** consiste nel suddividere una tabella delle pagine, di grosse dimensioni, in più livelli, evitando di tenere tutti i livelli in memoria per tutto il tempo. In pratica, in questo modo, si effettua una paginazione della tabella dei processi: una tabella delle pagine di alto livello viene utilizzata per accedere alle varie pagine della tabella delle pagine. Se la tabella delle pagine di alto livello è grande, potrebbe essere essa stessa paginata e così via.

11.3 Il gestore della memoria virtuale

Il *gestore della memoria virtuale* deve prendere due decisioni chiave che influenzano le prestazioni di un processo: quale pagina rimuovere dalla memoria per liberare spazio per una nuova pagina richiesta da un processo, e quanta memoria allocare a un processo.

Funzione	Descrizione
Gestione spazio di indirizzamento logico	Set up dello spazio di swap di un processo. Organizzare il suo spazio di indirizzamento logico in memoria attraverso le operazioni di page-in e page-out, e gestire la sua tabella delle pagine.
Gestione della memoria	Tenere traccia dei frame occupati e liberi in memoria.
Implementare la protezione di memoria	Conservare le informazioni utili per la protezione di memoria.
Raccogliere informazioni sui riferimenti alle pagine	Il supporto hardware alla paginazione fornisce le informazioni circa i riferimenti alle pagine. Queste informazioni sono tenute in appropriate strutture dati per l'uso da parte dell'algoritmo di sostituzione di pagine.
Eseguire sostituzione di pagine	Eseguire la sostituzione di una pagina quando viene generato un fault e tutti i page frame in memoria, o tutti i frame allocati per un processo, sono occupati.
Allocare memoria fisica	Decidere quanta memoria allocare a un processo e rivedere questa decisione di volta in volta per adattarsi alle necessità del processo e del SO.
Implementare la condivisione delle pagine	Organizzare la condivisione delle pagine da elaborare.

La tabella schematizza le funzioni del gestore della memoria virtuale.

Descriviamo di seguito alcune di queste operazioni, mentre nei paragrafi successivi descriveremo le funzioni più importanti.

GESTIONE DELLO SPAZIO DI INDIRIZZAMENTO LOGICO DI UN PROCESSO
Il gestore della memoria virtuale gestisce lo spazio di indirizzamento logico di un processo attraverso le seguenti sottofunzioni:

1. Creare una copia delle istruzioni e dei dati del processo nel suo spazio si swap
2. Gestire la tabella delle pagine
3. Eseguire le operazioni di page-in e page-out
4. Eseguire l'inizializzazione del processo

Come già menzionato nei paragrafi precedenti, una copia dell'intero spazio di indirizzamento logico di un processo è tenuta nello spazio di swap del processo. Quando si verifica un page fault, la pagina viene caricata dallo spazio di swap mediante un'operazione di page-in. Quando una pagina dirty deve essere rimossa dalla memoria, viene eseguita un'operazione di page-out per copiarla dalla memoria nello spazio di swap sul disco. In questo modo la copia di una pagina nello spazio di swap è attuale se quella pagina non è in memoria, o è in memoria ma non è stata modificata da quando è stata caricata.

GESTIONE DELLA MEMORIA
Il gestore della memoria virtuale si occupa anche di mappare i frame, cioè tenere traccia dei frame occupati e dei frame liberi. un frame viene eliminato dalla lista dei frame liberi quando viene effettuata un'operazione di page-in, mentre viene aggiunto nella lista quando viene effettuata un'operazione di page-out.
Quando un processo termina, tutti i frame ad esso allocati sono aggiunti nella lista dei frame liberi.

PROTEZIONE
Come già detto nei paragrafia precedenti, il gestore della memoria si occupa di conservare le informazioni utili per la protezione della memoria. Queste info vengono memorizzate nel campo *prot_info* della tabella delle pagine. In questo campo sono contenuti i privilegi di accesso (scrittura, lettura, esecuzione), dunque se si tenta di accedere ad una pagina con una modalità non consentita, viene generato un interrupt.

RACCOLTA DI INFORMAZIONI PER LA SOSTITUZIONE DI PAGINA
Il gestore della memoria virtuale si occupa anche di conservare le informazioni necessarie per la sostituzione delle pagine. Queste informazioni sono contenute nei campi *Ref_info* e *Modificato* della tabella delle pagine.

11.3.1 Panoramica sul funzionamento del gestore della memoria virtuale

Le due più importanti decisioni che prende il gestore durante il suo funzionamento sono:

- decidere quale pagina deve essere sostituita quando si verifica un page fault e non ci sono frame liberi in memoria
- decidere periodicamente quanta memoria, cioè quanti frame, allocare a ciascun processo

Queste decisioni vengono prese indipendentemente l'una dall'altra.

Quando si verifica un page fault, il gestore della memoria virtuale procede sostituendo una pagina attraverso un algoritmo di sostituzione delle pagine.
Quando decide di aumentare o diminuire la memoria allocata a un processo, specifica semplicemente il nuovo numero di frame che dovrebbero essere allocati a ciascun processo.

11.4 Politiche di sostituzione delle pagine

L'obiettivo di una politica di sostituzione delle pagine è quello di sostituire solo quelle pagine che non si useranno nell'immediato. Il problema, per certi versi, risulta essere simile al problema dello scheduling dei processi per ottenere l'uso della CPU.

Valutiamo le seguenti tre politiche di sostituzione delle pagine per vedere come si comportano rispetto all'obiettivo proposto:

- Politica di sostituzione delle pagine ottimale
- Politica di sostituzione delle pagine First-In First-Out (FIFO)
- Politica di sostituzione delle pagine Least Recently Used (LRU)

Per effettuare questa analisi ci basiamo sul concetto di *stringa dei riferimenti alle pagine*. Una stringa di riferimenti alle pagine di un processo è una sequenza di pagine a cui un processo ha fatto accesso durante la

sua esecuzione. Può essere costruita monitorando l'esecuzione di un processo e formando una sequenza di numeri di pagina.
Per convenienza, associamo una *stringa dei tempi dei riferimenti* t_1, t_2, t_3, … a ogni stringa dei riferimenti alle pagine. In questo modo, al k-esimo riferimento di pagina nella stringa dei riferimenti alle pagine è associato un istante di tempo t_k.

SOSTITUZIONE OTTIMALE DELLE PAGINE
La ***sostituzione ottimale*** consiste nel sostituire le pagine in modo tale che il numero totale di page fault durante l'esecuzione di un processo sia il minimo possibile. In pratica, un algoritmo del genere sostituisce solo quelle pagine che non si useranno per il periodo di tempo più lungo.

Naturalmente, una politica del genere è irrealizzabile in quanto al momento del page fault, il SO non ha nessun modo di sapere quando si farà riferimento a ciascuna delle pagine, cioè non può conoscere il comportamento futuro di un processo.

Esempio:
Stringa di riferimento 2,3,2,1,5,2,4,5,3,2,5,2

SOSTITUZIONE DELLE PAGINE FIFO
La ***politica di sostituzione delle pagine FIFO***, ad ogni page fault, sostituisce la pagina che è stata caricata in memoria prima di ogni altra pagina del processo, cioè quella che risiede in memoria da più tempo. In pratica, il SO mantiene una lista di tutte le pagine correntemente in memoria, dove la pagina di testa è la più vecchia e la pagina in coda è quella arrivata più di recente. Al momento del page fault, la pagina in testa viene rimossa anche se è la pagina più utilizzata, per questo motivo viene raramente utilizzato nella sua forma pura.

Per tenere traccia dell'ordine di arrivo si utilizza il campo *Ref_info* della tabella delle pagine.

Esempio:
Stringa di riferimento 2,3,2,1,5,2,4,5,3,2,5,2

SOSTITUZIONE DELLE PAGINE LRU
La ***politica di sostituzione delle pagine LRU***, a ogni page fault, sostituisce la pagina utilizzata meno di recente con la pagina richiesta. In pratica, viene scaricata la pagina usata meno di recente.
Questa politica è realizzabile ma non è conveniente, poiché per implementarla completamente, è necessario mantenere liste concatenate di tutte le pagine in memoria, con la pagina usata più di recente in testa alla lista; la difficoltà sta nel fatto che la lista va aggiornata ad ogni riferimento alla memoria.

ANALISI DELLE POLITICHE DI SOSTITUZIONE DI PAGINE

Il seguente esempio illustra il funzionamento delle politiche di sostituzione di pagina ottimale, FIFO e LRU.

Esempio 12.6 Funzionamento delle politiche di sostituzione di pagine

Si considerino la seguente stringa dei riferimenti alle pagine e la stringa dei tempi dei riferimenti per un processo *P*:

stringa dei riferimenti alle pagine	0, 1, 0, 2, 0, 1, 2, ...	(12.4)
stringa dei tempi dei riferimenti	t_1, t_2, t_3, t_4, t_5, t_6, t_7, ...	(12.5)

La Figura 12.15 illustra il funzionamento delle politiche di sostituzione delle pagine ottimale, FIFO e LRU per questa stringa dei riferimenti alle pagine con *alloc* = 2. Per convenienza, mostriamo solo due campi della tabella delle pagine, il *bit di validità* e *ref info*. Nell'intervallo tra t_0 e t_3 (incluso), vengono riferite solo due pagine distinte: le pagine 0 e 1. Possono stare entrambe in memoria nello stesso istante perché *alloc* = 2. t_4 è il primo istante di tempo in cui un page fault determina una sostituzione di pagina.

La colonna di sinistra mostra i risultati per la sostituzione di pagina ottimale. L'informazione sui riferimenti di pagina non è mostrata nella tabella delle pagine poiché l'informazione riguardante i riferimenti passati non serve per la sostituzione ottimale delle pagine. Quando si verifica un page fault all'istante di tempo t_4, la pagina 1 è sostituita perché il suo riferimento successivo è più lontano nella stringa dei riferimenti rispetto a quello di pagina 0. All'istante t_6 la pagina 1 sostituisce la pagina 0 perché il riferimento successivo della pagina 0 è più lontano rispetto a quello di pagina 2.

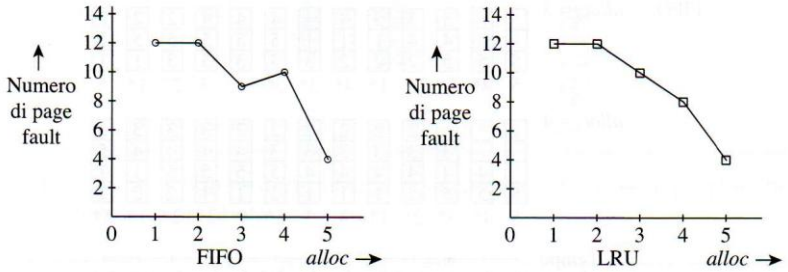
La colonna di centro della Figura 12.15 mostra i risultati per la politica di sostituzione FIFO. Quando si verifica un page fault all'istante di tempo t_4, il campo *ref info* mostra che la pagina 0 era stata caricata prima della pagina 1, e così la pagina 0 è sostituita dalla pagina 2.

L'ultima colonna della Figura 12.15 mostra i risultati per la politica di sostituzione LRU. Il campo *Ref info* della tabella delle pagine indica quando una pagina era stata riferita l'ultima volta. All'istante di tempo t_4, la pagina 1 è sostituita dalla pagina 2 perché l'ultimo riferimento della pagina 1 è precedente rispetto all'ultimo riferimento della pagina 0.

Il numero totale di page fault che si verificano con le politiche ottimale, FIFO e LRU sono 4, 6 e 5, rispettivamente. Per definizione, nessun'altra politica ha meno page fault rispetto alla politica di sostituzione ottimale delle pagine.

		Ottimale		FIFO		LRU	
Istante	Rif. pagina	*Bit di validità* / *Ref info*	Sostituzione	*Bit di validità* / *Ref info*	Sostituzione	*Bit di validità* / *Ref info*	Sostituzione
t_1	0	0: 1 ; 1: 0 ; 2: 0	–	0: 1, t_1 ; 1: 0 ; 2: 0	–	0: 1, t_1 ; 1: 0 ; 2: 0	–
t_2	1	0: 1 ; 1: 1 ; 2: 0	–	0: 1, t_1 ; 1: 1, t_2 ; 2: 0	–	0: 1, t_1 ; 1: 1, t_2 ; 2: 0	–
t_3	0	0: 1 ; 1: 1 ; 2: 0	–	0: 1, t_1 ; 1: 1, t_2 ; 2: 0	–	0: 1, t_1 ; 1: 1, t_2 ; 2: 0	–
t_4	2	0: 1 ; 1: 0 ; 2: 1	Sostituisci 1 con 2	0: 0 ; 1: 1, t_2 ; 2: 1, t_4	Sostituisci 0 con 2	0: 1, t_3 ; 1: 0 ; 2: 1, t_4	Sostituisci 1 con 2
t_5	0	0: 1 ; 1: 0 ; 2: 1	–	0: 1, t_5 ; 1: 0 ; 2: 1, t_4	Sostituisci 1 con 0	0: 1, t_5 ; 1: 0 ; 2: 1, t_4	–
t_6	1	0: 0 ; 1: 1 ; 2: 1	Sostituisci 0 con 1	0: 1, t_5 ; 1: 1, t_6 ; 2: 0	Sostituisci 2 con 1	0: 1, t_5 ; 1: 1, t_6 ; 2: 0	Sostituisci 2 con 1
t_7	2	0: 0 ; 1: 1 ; 2: 1	–	0: 0 ; 1: 1, t_6 ; 2: 1, t_7	Sostituisci 0 con 2	0: 0 ; 1: 1, t_6 ; 2: 1, t_7	Sostituisci 0 con 2

La figura in basso illustra come variano i page fault per gli algoritmi di sostituzione di pagina FIFO e LRU per la stringa dei riferimenti dell'esempio in alto.

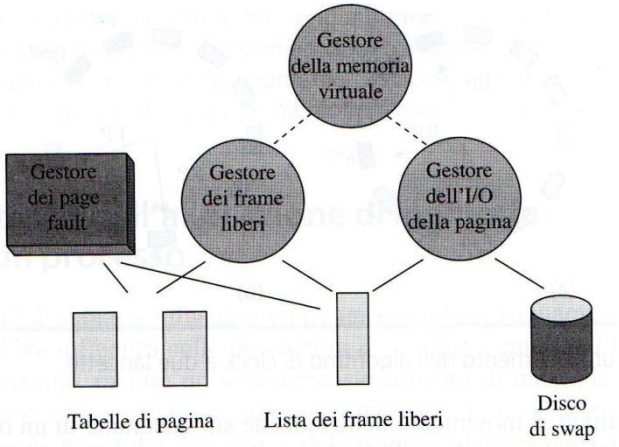


L'asse verticale mostra il numero totale di page fault, mentre sull'asse orizzontale è riportato l'*alloc*, cioè il numero di frame allocati al processo.
Come si può notare, c'è un'anomalia nel grafico della politica FIFO: il numero di page fault aumenta quando aumenta il numero di frame. Questo comportamento anomalo è noto come ***anomalia di Belady***.

Per questo motivo, il gestore della memoria non può utilizzare la politica di sostituzione FIFO perché incrementando i frame allocati al processo può aumentare la frequenza dei page fault.

Al contrario, nella politica LRU, il numero di page fault diminuisce all'aumentare dei frame.

11.4.1 Esempi realistici di politiche di sostituzione delle pagine



Il gestore della memoria virtuale mantiene una lista di frame liberi e prova a tenere pochi frame in questa lista a ogni istante di tempo. Esso consiste di due thread:

- *gestore dei frame liberi*
- *gestore dell'I/O della pagina*

La politica di sostituzione di pagina è implementata nel gestore dei frame liberi.

ALGORITMO NRU

La politica di sostituzione di pagina LRU sarebbe la scelta più conveniente, tuttavia, non è fattibile perché i computer non hanno sufficienti bit nel campo *Ref_info* per memorizzare il tempo di ultimo riferimento. Infatti, la maggior parte dei computer hanno un singolo bit di riferimento per raccogliere informazioni relative ai riferimenti di pagina. Per questo motivo, le politiche di sostituzione delle pagine devono essere implementate usando solo il bit di riferimento. Questo requisito ha portato a una classe di politiche chiamate ***politiche NRU*** (*Not Recently Used* – non usato di recente), in cui il bit di riferimento è utilizzato per determinare se una pagina è stata riferita di recente, e qualche pagina che non è stata riferita di recente viene sostituita.

Una semplice politica NRU è la seguente: il bit di riferimento di una pagina è inizializzato a 0 quando la pagina è caricata, ed è impostato a 1 quando la pagina viene riferita. Quando si rende necessaria una

sostituzione di pagina, se il gestore della memoria virtuale verifica che i bit di riferimento di tutte le pagine sono diventati 1, resetta i bit di tutte le pagine a 0 e arbitrariamente seleziona una delle pagine per la sostituzione; altrimenti, sostituisce una pagina il cui bit di riferimento è 0. Successive sostituzioni di pagina dipenderanno da quali pagine sono state riferite dopo il reset dei bit di riferimento.

L'attrattiva principale dell'algoritmo NRU è che è facile da capire, abbastanza semplice da implementare, e che dà delle prestazioni che spesso risultano soddisfacenti.

ALGORITMI DI CLOCK

Gli ***algoritmi di clock*** sono una sottoclasse ampiamente usata degli algoritmi NRU.
Essi forniscono la migliore discriminazione tra le pagine resettando i bit di riferimento delle pagine periodicamente, e non solo quando tutte diventano 1; in questo modo risulta possibile sapere se una pagina è stata riferita nell'immediato passato, piuttosto che dal momento in cui tutti i bit di riferimento sono stati resettati a 0.

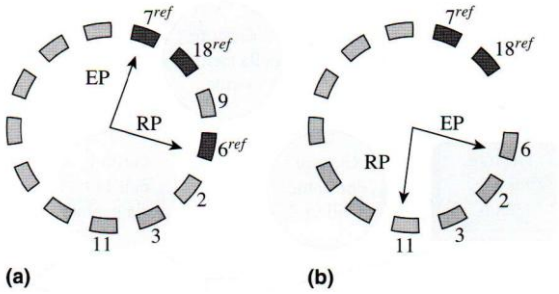
Per implementare questi algoritmi, le pagine di tutti i processi in memoria sono memorizzati in una lista circolare e i puntatori usati dagli algoritmi si muovono sulle pagine ripetutamente in modo analogo al movimento delle lancette di un orologio. Il bit di riferimento di una pagina è impostato a 1 quando:

- la pagina è caricata per la prima volta in un frame in memoria
- quando la pagina caricata nel frame viene utilizzata

Quando si verificare un page fault, si effettua una scansione della lista e si sostituisce il frame che ha il bit di riferimento posto a 0.

Nell'***algoritmo di clock a una lancetta***, una esecuzione è composta da due passi su ogni pagina. Nel primo passo, il gestore della memoria virtuale semplicemente resetta il bit di riferimento della pagina puntata dal puntatore. nel secondo passo cerca tutte le pagine i cui bit di riferimento sono ancora *off* e li aggiunge alla lista dei frame liberi.

Nell'***algoritmo di clock a due lancette***, sono utilizzati due puntatori. Un puntatore, chiamato *resetting pointer* (RP), serve per resettare i bit di riferimento, mentre l'altro puntatore, chiamato *examining pointer* (EP), viene utilizzato per controllare i bit di riferimento. entrambi i puntatore vengono incrementati simultaneamente. Il frame al quale il puntatore EP punta è aggiunto alla lista dei frame liberi se il suo bit di riferimento è *off*.



Esempio 12.8 Algoritmo di clock a due lancette

La Figura 12.20 illustra il funzionamento dell'algoritmo di clock a due lancette utilizzato dal gestore dei frame liberi di Figura 12.19. Il simbolo ^{ref} su una pagina implica che il bit di riferimento della pagina è settato ad 1; l'assenza di questo simbolo implica che il bit di riferimento è 0. Quando viene attivato il gestore dei frame liberi, esamina la pagina 7, che è puntata dall'examining pointer [Figura 12.20(a)]. Il suo bit di riferimento è 1, quindi vengono avanzati entrambi i puntatori, il resetting e l'examining. In questo istante, il bit di riferimento della pagina 6 è resettato a 0 perché RP stava puntando a esso. Il puntatore examining si sposta su pagina 18 (e il resetting si sposta su pagina 2) perché, anch'esso ha il suo bit di riferimento settato ad 1. Ora resta sulla pagina 9. La pagina 9 ha il suo bit di riferimento a 0, quindi è rimossa dalla lista delle pagine in memoria e aggiunta a quella dei frame liberi. I puntatori resetting e examining puntano ora alle pagine 6 e 11, rispettivamente [Figura 12.20(b)]. La di-

La distanza tra i puntatori RP ed EP fornisce proprietà diverse agli algoritmi. Se i puntatori sono molto vicini, sarà esaminata una pagina subito dopo il reset del suo bit di riferimento, di conseguenza solo le pagine usate di recente rimarranno in memoria. Se i puntatori del clock sono abbastanza lontani, solo le pagine che non sono state utilizzate da molto tempo saranno rimosse.

11.5 Allocazione dei frame a un processo

Diciamo molto superficialmente che sia il sovradimensionamento che il sottodimensionamento di memoria concesso a un processo porta a cali di prestazioni del sistema e scarsa efficienza della CPU.
Tuttavia, non è chiaro il modo in cui il gestore della memoria virtuale decide il giusto numero di frame da allocare a ogni processo, ossia il giusto valore di *alloc* per ogni processo.

Vengono utilizzati due approcci per controllare l'allocazione dei frame per un processo:

- ***allocazione di memoria fissa*** – l'allocazione di memoria per un processo è fissa; di conseguenza, la prestazione di un processo è indipendente dagli altri processi del sistema. Quando si verifica un page fault in un processo, viene sostituita una delle sue pagine. Questo approccio è detto *sostituzione di pagina locale*
- ***allocazione di memoria variabile*** – l'allocazione di memoria può essere variata in due modi. Quando si verifica un page fault, tutte le pagine di tutti i processi che sono in memoria possono essere prese in considerazione per la sostituzione. Ciò è identificato come *sostituzione globale delle pagine*. Alternativamente, il gestore della memoria virtuale può rivedere l'allocazione della memoria per un processo periodicamente sulla base della sua località e sul comportamento riguardo ai page fault, ma quando si verifica un page fault esegue una sostituzione locale di pagina.

Nell'allocazione fissa in ambito globale, le decisioni riguardanti l'allocazione di memoria sono eseguite staticamente. La memoria da allocare a un processo è determinata in base ad alcuni criteri quando il processo viene inizializzato. La sostituzione di pagina è sempre eseguita localmente. Questo metodo risente di tutti i problemi connessi a una decisione statica: un sottodimensionamento o un sovradimensionamento di memoria per un processo può influenzare la prestazione del processo stesso e quella del sistema.

Nell'allocazione variabile in ambito globale, l'allocazione per il processo attualmente in esecuzione può diventare troppo grande.

La miglior soluzione tra le tre è senza dubbio l'allocazione variabile in ambito locale che utilizza la sostituzione locale delle pagine, perché il gestore della memoria virtuale determina il giusto valore di *alloc* per un processo, di volta in volta.

11.5.1 Il modello a Working Set

Il concetto di un *working set* fornisce una base per decidere quanti e quali pagine di un processo dovrebbero essere in memoria per ottenere una buona prestazione di processo.

L'insieme di tutte le pagine di ogni processo viene detto ***working set*** (*insieme di lavoro*). É stato creato il modello working set per ridurre sensibilmente il tasso di page fault e consiste nell'assicurarsi che l'insieme di lavoro sia caricato totalmente in memoria prima di consentire ad un altro processo di andare in esecuzione (altrimenti si verificherebbero mille page fault per ogni processo).

Il numero di pagine nel working set può aumentare o diminuire, a seconda della disponibilità delle pagine stesse. Il working set aumenta come un processo page fault. Al contrario esso diminuisce con la diminuzione delle pagine disponibili. Per evitare la consumazione completa della memoria, le pagine devono essere rimosse dai working set del processo e trasformate in pagine disponibili per un loro eventuale utilizzo.

Il sistema operativo diminuisce i working set del processo nei seguenti modi:

- scrivendo su pagine modificate in un'area dedicata, su di un dispositivo memoria di massa (generalmente conosciuti come spazio di swapping o paging)
- contrassegnando pagine non modificate come libere (non vi è alcuna necessità di scrivere queste pagine su disco in quanto queste non sono state modificate)

Per determinare i working set appropriati per tutti i processi, il sistema operativo deve possedere tutte le informazioni sull'utilizzo per tutte le pagine. In questo modo, il sistema operativo, determina le pagine usate in modo attivo (risiedendo sempre nella memoria) e quelle non utilizzate (e quindi da rimuovere dalla memoria). In molti casi, una sorta di algoritmo non usato di recente, determina le pagine che possono essere rimosse dai working set dei processi.

11.5.2 Working Set Clock

L'algoritmo base dell'insieme di lavoro è ingombrante poiché ad ogni page fault deve essere analizzata tutta la tabella delle pagine, fino a quando non viene trovata una pagina candidata adatta. L'algoritmo ***WSClock*** (*working set clock*) è un algoritmo perfezionato, basato sull'algoritmo dell'orologio ma che utilizza anche le informazioni dell'insieme di lavoro; per la sua semplicità di implementazione e per le buone prestazioni è utilizzato largamente nella pratica.

La struttura dati necessaria è una lista circolare di pagine fisiche, come nell'algoritmo dell'orologio. Inizialmente la lista è vuota; quando viene caricata la prima pagina, essa viene aggiunta alla lista e ogni volta che viene caricata una pagina, questa va a finire nella lista in modo da formare un anello. Ogni elemento contiene il campo relativo al tempo di ultimo utilizzo derivante dall'algoritmo base dell'insieme di lavoro, e contiene inoltre il bit R ed il bit M.

Come nell'algoritmo dell'orologio, ad ogni page fault, la pagina puntata dalla lancetta viene esaminata per prima: se il bit R assume il valore 1, la pagina è stata usata durante il tick corrente, quindi non è una buona candidata ad essere rimossa, il bit R viene quindi impostato a 0, quindi si passa ad esaminare la pagina successiva e l'algoritmo viene ripetuto per quella pagina.