di Andrea de Prisco

Memoria virtuale, Paginazione, Segmentazione

Questo mese discuteremo sulla gestione della memoria dei calcolatori, come è ormai costume in questa rubrica, non troppo mini. Ovvero in che modo un calcolatore multiprogrammato mantiene in memoria i vari programmi da elaborare, senza che questi ne risentano di tale condivisione.

Ragioni storiche

Agli albori dell'informatica, anni '50, ovviamente nessuno pensava di multiprogrammare un calcolatore. Già era un problema mandare in esecuzione un programma, figuriamoci più d'uno in parallelo. La memoria di tali prototipi era così piccola a causa degli alti costi per bit che solo programmini davvero semplici potevano essere elaborati. A quei tempi non esistevano nemmeno le memorie di massa: si immetteva il programma su supporto cartaceo (schede o nastri perforati) e i risultati dell'elaborazione potevano essere o letti direttamente sulla carta della stampante o tuttalpiù si potevano far perforare nuove schede per usare i risultati del primo calcolo per elaborazioni future.

Poi arrivarono i nastri magnetici (nell'informatica) e i primi supporti veloci come dischi e tamburi, grazie ai quali si cominciarono a sfruttare meglio le potenzialità di calcolo di questi sistemi. Le memorie centrali continuavano però a costare molto, tant'è che si pensò di suddividere i programmi più grossi in piccole porzioni da caricare in memoria a turno, a seconda della relativa fetta di programma che in quell'istante doveva essere elaborata. Così il programmatore, conoscendo le dimensioni della memoria del calcolatore, provvedeva ad aggiungere in punti opportuni dei suoi programmi istruzioni tipo «carica questo pezzo», «scarica quest'altro» o roba simile.

Tali operazioni dovevano essere usate intelligentemente, pena un appesantimento troppo grave dell'elaborazione, che se andava avanti a carica e scarica di pezzi di memoria centrale in memoria secondaria (e viceversa), portava a tempi di elaborazione troppo lunghi per essere appena sopportabili.

Inoltre, un programma scritto per un calcolatore con una certa quantità di memoria difficilmente era trasportabile su un calcolatore compatibile al primo con quantità di memoria diversa, in quanto come minimo bisognava riorganizzare le suddivisioni per ottenere prestazioni accettabili sul secondo sistema.

Fu così che nel 1961, in Inghilterra, un gruppo di persone pensò ad una

Figura 1

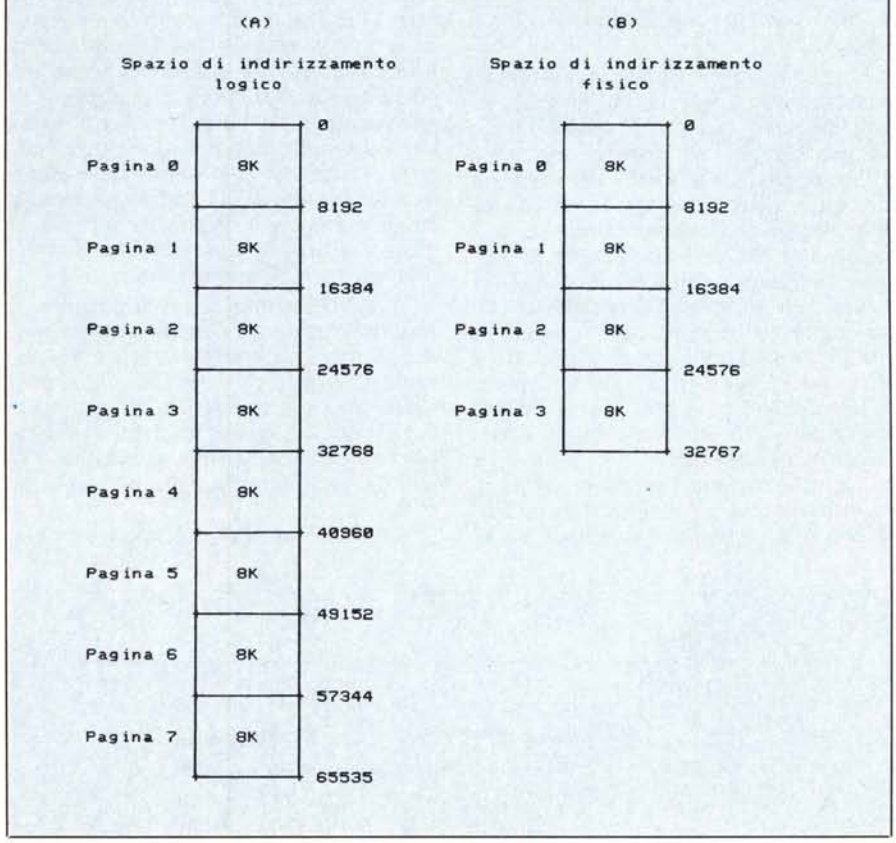
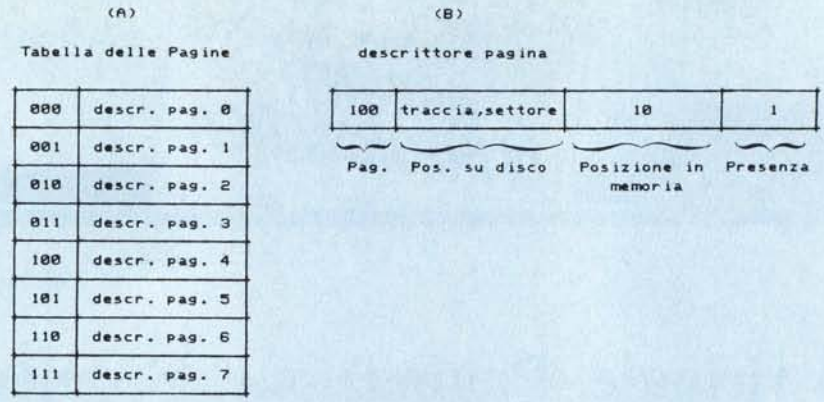


Figura 2



gestione automatica (ossia da parte del sistema) di questo procedimento di togli e metti porzioni di programma. In primo luogo sgravando il programmatore da tale onere e, non meno importante, aumentando la portabilità di un programma da un sistema all'altro, purché ambedue dotati del medesimo automatismo. Nacque così il concetto di...

Memoria virtuale

Prima di andare avanti vogliamo rispondere a quei lettori che fino a questo momento si sono chiesti cosa centra tutto questo con la multiprogrammazione, e soprattutto a quelli che hanno pensato: «Sì, va bene, ma ora la Ram costa poche migliaia di lire a Kappa...».

Il problema non è tanto l'effettiva quantità di memoria di cui un calcolatore dispone, quanto di come questa sia visibile ai vari processi in corso d'esecuzione. Infatti se è vero che un moderno calcolatore può facilmente disporre di qualche megabyte di memoria, è anche vero che su questo possono benissimo «girare» un centinaio di processi e un megabyte diviso 100 può ugualmente essere poco, come ai vecchi tempi. Quello che hanno fatto a Manchester nel 1961 è stato di separare il concetto di spazio logico di memoria da quello di spazio fisico. Cioè: «Caro programmatore, fai conto che la memoria disponibile sia di 64 Kappa, poi come sia possibile fare ciò in 32 non è affar tuo, se la sbriga il sistema». Questo ad esempio se lo spazio fisico di memoria (= centrale) sia pari a 32 Kappa e vogliamo che i programmi ne vedano il doppio.

Il primo approccio che considereremo è detto:

Paginazione

Il sistema considera il programma su disco come suddiviso in 8 pagine da 8 Kappa l'una (vedi figura 1A). Analogamente vedrà la propria memoria centrale suddivisa in pagine da 8 Kappa, questa volta in numero di 4 essendo disponibili solo 32768 byte (figura 1B). Dando RUN al programma, si carica in memoria la prima pagina da disco nella prima pagina fisica e si inizia l'elaborazione. Se in tale porzione il programma si riferisce a una cella non compresa nei primi 8 Kappa, occorre caricare da disco la pagina corrispondente che ad esempio verrà posta nella seconda pagina in memoria centrale. Posto che non vi sia più spazio occorrerà crearlo scaricando una pagina da memoria a disco, per fare largo al nuovo arrivo. Così via fino a completamento dell'esecuzione.

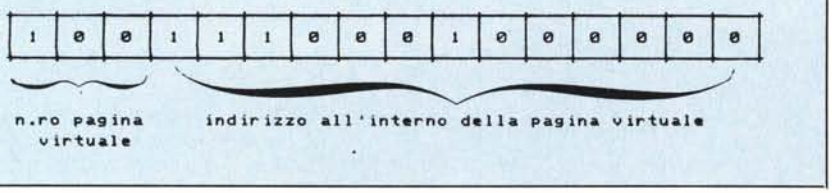
Questo in prima approssimazione. È importante notare come il sistema agisca in maniera completamente trasparente all'utente che, dal canto suo, può benissimo essere ignaro di tutto ciò. Ad esempio, il programmatore potrebbe riferire nelle prime istruzioni del suo programma alla cella di memoria n. 40000.

40000 diviso 8192, 8 Kappa, fa 4 col resto 7232: per il sistema il programmatore ha fatto un riferimento alla cella 7232 della pagina logica n. 4: se la pagina 4 è già presente in memoria centrale bene, altrimenti bisognerà caricarla da disco. Analogamente, se il processore sta elaborando le istruzioni della prima pagina logica e arrivati in fondo banalmente non vi è un goto a un indirizzo della stessa pagina, il sistema automaticamente caricherà la seconda pagina logica, se questa naturalmente non è già presente in memoria, per proseguire così l'elaborazione.

A questo punto è ovvio che il sistema in ogni momento deve conoscere un po' di informazioni riguardanti le pagine, fisiche e logiche. Innanzitutto deve essere al corrente di quali pagine logiche sono in memoria centrale e dove queste sono state poste. Se nell'esempio precedente la pagina logica 4 è stata precedentemente caricata nella pagina fisica n. 2, il riferimento cella 7232 della pagina logica 4 diventa cella 7232 della pagina fisica 2 che è poi quella che ci interessa. Analogamente, delle pagine non caricate in memoria deve conoscere dove queste siano locate su disco, ad esempio traccia e settore. Un'altra informazione utile potrebbe riguardare il fatto se una pagina sia stata modificata o meno dall'elaborazione del programma stesso. Questo può essere utile quando dobbiamo fare posto in memoria centrale per caricare una nuova pagina: se la vecchia non è stata modificata, non bisognerà scaricarla su disco dato che una copia perfettamente identica sarà già lì presente. Generalmente le pagine che contengono solo istruzioni non sono modificate dall'elaborazione, mentre quelle contenenti dati è più probabile che debbano essere scaricate su disco a causa di modifiche avvenute durante l'elaborazione.

Il sistema dovrà quindi conservare da qualche parte una tabella delle pagine dove mantenere le informazioni che gli servono: per ogni pagina logica

Figura 3



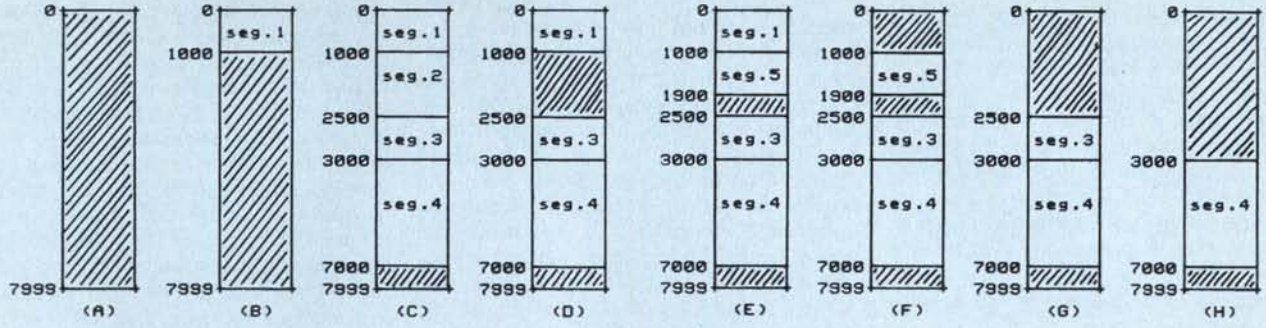


Figura 4 - Esempio di evoluzione di una memoria segmentata.

si servirà di un descrittore (vedi figg. 2 A e 2 B) che provvederà ad aggiornare man mano che evolverà lo stato della memoria virtuale (nuovi arrivi, modifica alle pagine e così via).

A questo punto, scendiamo un po' nei dettagli per vedere più da vicino come avviene la trasformazione dell'indirizzo logico (es. cella 40000) nel corrispondente indirizzo fisico in una delle pagine di memoria centrale del calcolatore preso in esempio. Poniamoci come prima nel caso in cui la pagina logica 4 sia già presente nell'area fisica n. 2 a seguito di un precedente riferimento. Dicevamo che lo spazio logico indirizzabile ammonta a 64 Kappa: ciò vuol dire che per accedere ad una qualsiasi cella delle 65536 possibili occorrerà un indirizzamento a 16 bit, essendo 2 elevato a 16 = 65536.

40000 in notazione binaria, come mostrato anche in figura 3 è 1001110001000000: le prime tre cifre binarie (a cominciare da sinistra) di tale numero rappresentano il numero della pagina virtuale (100) che in decimale diventa 4. Le rimanenti 13 cifre rappresentano invece la posizione all'interno della pagina, nel nostro caso 7232. Come questo sia possibile è assai semplice: 8192, la dimensione di una pagina, è proprio una potenza di 2, quindi il quoziente e il resto di tale divisione si ottiene semplicemente guardando il numero binario da dividere. Se non vi torna, come già detto nel primo articolo di Appunti di Informatica, è solo perché avete paura dei numeri binari, tutto qui. Essi sono isomorfi ai numeri naturali in base 10 che sappiamo ben manipolare sin dal tempo delle elementari: prendiamo un numero decimale qualsiasi, 1234567 ad esempio e proviamo a dividerlo pr 1000, questa volta ovviamente una potenza di 10: un coro unanime urlerà 123 col resto di 4567, semplicemente guardando il dividendo.

Torniamo a noi: l'100 (uno-zero-zero, non cento, mi raccomando) prelevato ci servirà per accedere al corrispondente elemento nella tabella delle pagine, mostrato sempre in figura 2A. Lì dentro, potremo prelevare la posizione in memoria centrale della pagina logica (10, in decimale 2) e sostituire questo valore al posto dell'100 precedente ottenendo 101110001000000 pari a 23616 in decimale che è la cella fisica corrispondente, nel nostro caso, alla cella logica da cui siamo partiti.

Working set

Ciò che abbiamo appena descritto è quanto avviene nel caso in cui la pagina di memoria che stiamo riferendo è presente in memoria centrale essendo stata riferita precedentemente. Come già detto, l'informazione circa la presenza o meno in memoria centrale, la troviamo nella tabella delle pagine quando vi accediamo per trasformare l'indirizzo logico in indirizzo fisico con l'algoritmo appena discusso. È ovvio che può benissimo capitare di non trovare la pagina in memoria e in questo caso occorrerà caricarla da disco per procedere con l'elaborazione. Altrettanto ovvio è che quanto più carichiamo e scarichiamo pagine di memoria tanto più sarà faticoso potare a termine una elaborazione. Nel 1968 Denning (un altro dei «potenti» in informatica) osservò come la maggior parte dei programmi non indirizzino uniformemente tutto lo spazio logico di memoria, ma prevalentemente un certo insieme di pagine detto working set che con l'avanzare dell'elaborazione, tende a stabilizzarsi verso determinati valori. Se quindi il numero delle pagine fisiche disponibili per ospitare pagine logiche di una elaborazione è più grande del working set del programma, la gestione della memoria virtuale non ci creerà molti problemi in quanto a continui togli e metti. Al contrario se il numero delle pagine è inferiore al working set del programma, potremo rassegnarci all'idea di assistere a molto movimento memoria-disco prima di vedere terminata l'elaborazione.

Tornando ai calcolatori multiprogrammati, possiamo ora notare come non sia arbitrario il numero di programmi mantenibili contemporaneamente in memoria: infatti essendo questa, sebbene grande, per definizione non illimitata, caricando troppi programmi dovremmo destinare ad ognuno di questi via via sempre un numero inferiore di pagine che oltrepassato il limite minimo del working set di ciascuno, come detto, farebbe degradare troppo le prestazioni dell'intero sistema.

Fate largo

Il terzo ed ultimo problema riguarda l'eventualità di non avere spazio disponibile in memoria centrale per caricare la nuova pagina richiesta: occorre scegliere una pagina da scartare per fare posto al nuovo arrivo.

Sceglierne una a caso certamente non sarà una buona idea anche perché se siamo sfortunati potremmo essere costretti a ricaricarla in memoria subito dopo.

Anche per questo problema non esistono soluzioni ottime in assoluto, ma si basano tanto per cambiare su osservazioni probabilistiche. Una prima idea potrebbe essere quella di scartare la pagina meno recentemente usata ossia quella che da più tempo non è stata riferita: con buone probabilità tale pagina non ci servirà più. Tale algoritmo detto LRU (Last-Recently-Used) non è molto agevole da implementare in quanto occorre tenere traccia dei riferimenti fatti alle pagine mantenute in memoria.

Un altro algoritmo detto FIFO (First-In-First-Out) come dice il suo nome semplicemente scarta la pagina che da più tempo risiede in memoria, indipendentemente dai riferimenti ad essa fatti nell'ultimo intervallo di tempo. In ogni caso, sia l'uno che l'altro possono miseramente fallire sotto al-

cune ipotesi: vediamo un piccolo esempio.

Immaginiamo di disporre di otto pagine in memoria centrale e che il nostro programma ne occupi nove (di logiche, naturalmente). Sia il caso, inoltre, che il nostro programma è composto semplicemente di un grosso loop: si tratta di esegure di filato le nove paginate di istruzioni e l'ultima istruzione è un goto all'inizio del programma. La sequenza di eventi sarà questa:

si carica la prima pagina in memoria e si esegue;

si carica la seconda pagina e si esegue;

si procede così fino all'ottava pagina;

a questo punto per caricare la nona occorre scaricarne una;

sia LRU che FIFO scaricheranno la prima;

si esegue la nona e... rioccorre la prima;

per caricare la prima sia LRU che FIFO scaricano la seconda;

si esegue la prima e... rioccorre la seconda...

e così via caricando e scaricando fino al termine dell'elaborazione.

Cosa è successo? Semplicemente il working set del nostro programma era di nove pagine secche e noi ne avevamo disponibili solo otto: con LRU e FIFO non potevamo aspettarci di meglio.

Segmentazione

Un'alternativa alla gestione della memoria virtuale appena vista, è la segmentazione. La differenza fondamentale tra i due metodi riguarda la lunghezza delle singole porzioni di memoria occupate, che nel primo caso ha dimensione fissa (es.: 8 Kappa) mentre con la segmentazione viene allocato di volta in volta uno spazio di lunghezza variabile, a seconda di quanto effettivamente ne serve.

Infatti, nell'esempio precedente, se il programma in questione invece di essere lungo 64 Kappa fosse stato lungo 60, l'ultima delle sue 8 pagine sarebbe ovviamente piena solo per metà nonostante il sistema allocasse in memoria centrale sempre 8 Kappa, per qualsiasi pagina.

D'altro canto, come vedremo, la segmentazione è un po' più pesante da implementare, in particolare per la gestione degli spazi vuoti, se si vuole godere di tutti i suoi benefici.

La lunghezza dei singoli segmenti, come detto prima, dipenderà dal caso: ad esempio possiamo allocare un segmento della lunghezza del programma principale, altri segmenti per i singoli sottoprogrammi, ancora allocheremo segmenti per le strutture dati, tutti ovviamente saranno presenti in memoria solo quando servono per non occupare inutilmente spazio.

Anche per la segmentazione, abbiamo il problema di trasformare indirizzamenti logici (coppie segmento-posizione in questo) in indirizzi fisici (essenzialmente una ben precisa locazione della memoria), solo che in questo caso il giochetto di prima non è più applicabile proprio per l'arbitrarietà della posizione di inizio e della lunghezza dei segmenti stessi. Ci avvarremo di una memoria associativa ad alta velocità, detta cache memory, tramite la quale partendo dal segmento otteniamo l'indirizzo di inizio dello stesso in memoria e sommandoci la posizione relativa che ci interessa otteniamo l'indirizzo fisico. Per fare un esempio, immaginiamo di dover indirizzare la locazione 100 del segmento n. 3. Supponiamo inoltre che tale segmento sia già presente in memoria centrale a partire dalla locazione 2000. Passando 3 alla memoria associativa questa ci restituirà l'inizio del terzo segmento (2000), basterà sommarci 100 per ottenere 2100 che è l'indirizzo fisico corrispondente dunque alla cella cercata.

Oltre a tutto questo, come prima, abbiamo i problemi riguardanti il carico e lo scarico dei segmenti nonché l'onere di ricercare lo spazio necessario prima dell'operazione di caricamento.

Passiamo ora a considerare la figura 4: come indica la didascalia rappresenta un esempio di evoluzione, col passare del tempo, di memoria virtuale gestita con segmentazione, avendo disponibili solo 8000 celle di memoria centrale.

Si parte, naturalmente, con la memoria completamente libera come mostrato in figura 4 A. Carichiamo in memoria un primo segmento: figura 4 B; immaginiamo di caricare allo stesso modo altri segmenti necessari all'elaborazione del primo: la situazione è mostrata in figura 4 C, al momento l'unico spazio libero rimasto sono le 1000 celle a fine memoria. Supponiamo a questo punto che il segmento 2 non serva più e il sistema decida di scaricarlo rendendo libero lo spazio corrispondente. L'evoluzione procede poi con l'allocazione di 900 celle per il segmento 5, figura 4 E, la deallocazione dello spazio corrispondente al segmento 1, figura 4 F, poi del segmento 5, figura 4 G, lasciando la memoria come indicato in figura 4 H, dopo aver scaricato anche il segmento 3.

Ovviamente il sistema dovrà essere a consocenza dello stato della memoria, prima di caricare qualcosa da disco. Un modo per descrivere tale stato, è quello di tenere traccia degli spazi liberi tramite una opportuna lista: un elenco delle posizioni degli spazi vuoti con relativa lunghezza di ognuno di questi.

Inoltre tale lista dovrà essere manipolata intelligentemente, se non si vuole arrivare ben presto alla paralisi

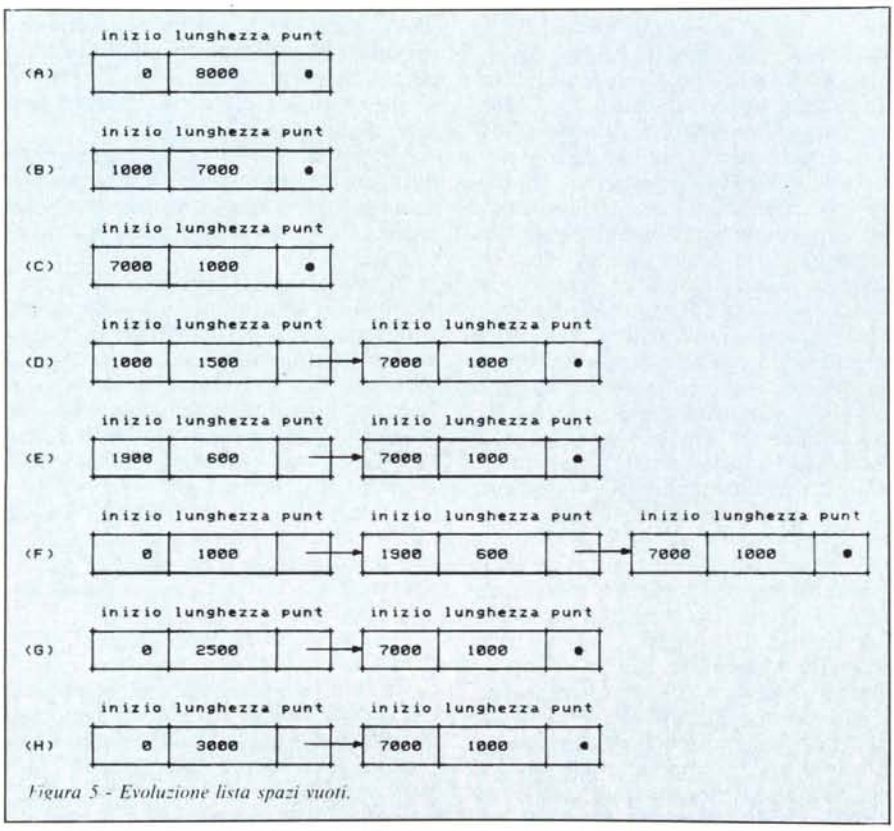


Figura 5 - Evoluzione lista spazi vuoti.

completa di tutto il sistema. Per esempio, tornando alla figura 4 F notiamo che sono presenti tre spazi: uno di 1000 celle all'inizio, uno di 600 celle a partire dalla posizione 1900 e uno di 1000 celle in fondo. Scaricando il segmento 5, come mostrato in figura 4 G gli spazi liberi non passano da 3 a 4 ma, badaben-badaben-badaben, diventano 2, il primo lungo 2500 celle, il secondo come prima di 1000 celle in fondo alla memoria. Per completezza, in figura 5 è mostrata la lista degli spazi vuoti corrispondenti alle otto situazioni di memoria mostrate in figura 4.

Strategie di allocazione

La lista degli spazi vuoti è usata dal sistema in tre casi: quando occorre trovare lo spazio per caricare un nuovo segmento, dopo aver caricato un segmento (e quindi lo spazio libero è diminuito) o dopo aver liberato spazio a causa di un rilascio. Per quanto riguarda il primo di questi tre punti, come per la paginazione possono presentarsi due altre eventualità: spazio disponibile e non. Ad esempio, per continuare l'elaborazione, potrebbe essere necessario il segmento 34, non presente in memoria centrale. D'altra parte, se non disponiamo di spazio a sufficienza (e il sistema può accorgersi di ciò consultando la lista «liberi») occorrerà scaricare qualche segmento, magari uno di quelli che è stato usato meno nell'ultimo intervallo di tempo.

Ancora, potrebbe verificarsi il caso in cui lo spazio in memoria ci sia, ma sparso un po' qua, un po' là in modo da non permettere un'allocazione contigua: in condizioni simili, generalmente non vengono scaricati altri segmenti, ma appositi algoritmi ricompattano la memoria, in modo da concentrare i «vuoti» in coda o in testa per poi caricare in quella zona il segmento ricercato.

Ultimo problema: posto che vi sia spazio a sufficienza ossia che la lista «liberi» sia ben nutrita e con molti tagli anche più grandi del segmento che dobbiamo caricare, come scelgo la zona di memoria da occupare?

Esistono essenzialmente tre algoritmi: il primo, first fit, scorrendo la lista alloca il primo spazio che trova sufficientemente capiente per contenere il segmento. Dà ottimi risultati, proprio per la sua intrinseca semplicità, specialmente se ricerche successive non ricominciano da inizio lista, ma ciclicamente scorrono questa a partire dall'ultima allocazione fatta: ciò permette di non concentrare pezzettini piccoli, non utilizzabili direttamente, in testa alla memoria.

Un secondo algoritmo, best fit, scorre tutte le volte l'intera lista per trovare il vuoto più piccolo capace di contenere il segmento richiesto: se abbiamo la fortuna di trovare spazi esattamente lunghi quanto il segmento sfrutteremo certamente bene la memoria, ma se ciò non avviene (e di solito è così) inevitabilmente aggiungeremo ogni volta nuovi vuoti piccoli che certamente non fanno piacere.

Complementare a questo, l'algoritmo worst fit, cerca il segmento più lungo che c'è in modo da lasciare conseguentemente anche un nuovo vuoto abbastanza grande quindi con ottime probabilità di essere adoperato in futuro. MC

Bibliografia

Andrew S. Tanembaum:
«Structured Computer Organization»
Prentice-Hall, 1976

A.N. Habermann:
«Introduction to Operating System Design»
Science Research Associates, 1976