Azzolini Riccardo 2019-11-21

Gestione della memoria

1 Riuso della memoria

Il problema dell'allocazione della memoria può essere visto a due livelli:

- il SO deve allocare memoria per i processi;
- all'interno della memoria di un processo, il RTS del linguaggio deve allocare memoria per i dati PCD.

In entrambi i casi, è necessario riutilizzare la memoria liberata. Per poterlo fare, bisogna tener traccia di quali blocchi di memoria sono occupati e quali sono invece liberi. Una possibile soluzione è l'uso di una **free list**:

- si ha un puntatore al primo blocco libero;
- ogni blocco libero contiene:
  - un campo che ne indica la dimensione;
  - un puntatore al blocco libero successivo;
  - opzionalmente, un puntatore al blocco libero precedente (se si vuole realizzare una *double linked list*).

Quando la free list viene usata dal sistema operativo (per tenere traccia della memoria assegnabile ai processi), il puntatore al primo blocco è una variabile situata nella kernel area.

2 Tecniche di allocazione

Quando si vuole avviare un programma, in un sistema con allocazione contigua, il SO deve trovare un blocco libero abbastanza grande da contenere l'intero programma.

In generale, in un sistema che gestisce la memoria libera mediante una free list, per allocare un'area contigua di k byte è necessario cercare nella lista un blocco libero di dimensione $\geq k$. Se la sua dimensione è d, dopo aver allocato i k byte richiesti si ha ancora un'area libera di $d - k$ byte, che rimane nella free list.

Esistono almeno tre tecniche diverse per effettuare la ricerca:

first fit: si cerca la prima area libera di dimensione sufficiente ($\geq k$);

best fit: si cerca il blocco libero più piccolo tra quelli di dimensione $\geq k$;

next fit: si adotta lo stesso criterio di ricerca della first fit, ma la scansione della free list parte dal blocco successivo a quello in cui è stata effettuata l'ultima allocazione.

2.1 Confronto

- Con la first fit, un blocco si può spezzare più volte di seguito, quindi c'è una tendenza alla formazione di aree libere di piccole dimensioni, che rischiano di essere inutili.
- Con la best fit, almeno in teoria, vengono preservate le aree di dimensione maggiore, che potrebbero essere indispensabili per allocazioni grandi, ma, a lungo andare, il problema si presenta ugualmente. Inoltre, la ricerca nella free list è costosa, perché è necessario scorrere l'intera lista a ogni allocazione.
- La next fit è un compromesso: evita di spezzare più volte di seguito lo stesso blocco, ma senza il costo di ricerca associato alla best fit.

Nella pratica, la first fit e la next fit risultano più performanti della best fit.

3 Frammentazione

Definizione: L'esistenza di aree di memoria[1] non utilizzabili in un sistema di computazione è detta **frammentazione**.

Alcune tecniche per prevenire/contrastare la frammentazione sono:

- **Boundary tags**:
  - le aree libere sono collegate da una catena di puntatori;
  - all'inizio e alla fine di ogni area viene messo un tag, che ne indica lo stato (libera/occupata) e la dimensione;
  - quando un'area occupata viene liberata, la si unisce con le eventuali aree libere adiacenti (modificando i boundary tags, ed eventualmente aggiornando alcuni dei puntatori che formano la catena).
- **Memory compaction**:
  - le aree libere sono collegate da una catena di puntatori;

[1] Il problema della frammentazione vale non solo per la RAM, ma, in generale, per tutti i dispositivi di memoria.

  - a determinati intervalli di tempo, la memoria viene compattata, "spostando" le aree occupate in modo da unire tutte le aree libere, formando un unico blocco libero;
  - la compattazione comporta un cambiamento degli indirizzi dei processi, quindi è ragionevole solo la rilocazione non è un'operazione costosa (ad esempio, con la rilocazione dinamica mediante Relocation Register, è sufficiente modificare il valore del RR per i processi spostati, mentre, con la rilocazione statica, sarebbe necessario aggiornare gli indirizzi contenuti nei programmi in memoria);
  - quando il SO compatta la RAM (il che può richiedere anche alcuni minuti), la macchina è inutilizzabile.
- **Powers-of-two allocator**.
- **Buddy systems**.

4 Allocazione contigua

Si ha **allocazione contigua** della memoria quando ogni processo è allocato in una singola area di memoria contigua.

- La protezione della memoria è semplice: si può ottenere con i registri LBR/UBR.
- La rilocazione dei processi è semplice: può essere effettuata mediante il registro RR.
- Si può allocare in memoria un processo che richiede k byte solo se è disponibile un'area libera di dimensione $\geq k$. Si ha quindi il problema della frammentazione: potrebbe capitare che lo spazio complessivamente disponibile sia sufficiente, ma nessuna singola area libera sia abbastanza grande per il processo da allocare. In questo caso, una possibile soluzione è la compattazione della memoria.

4.1 Swapping

L'allocazione contigua può essere integrata con lo **swapping**: i processi (ready o waiting) che non ci stanno in RAM vengono spostati sullo *swap device* (una porzione del disco).

- Lo *swap out* (spostamento dalla RAM allo swap device) e lo *swap in* (dallo swap device alla RAM) vengono eseguiti da un processo di sistema, e richiedono tempo (soprattutto se abbinati a una compattazione), durante il quale la macchina è inutilizzabile.

- Quando si fa lo swap in di un processo, questo può essere facilmente caricato anche in un'area di RAM diversa da quella che occupava in precedenza (se si dispone del registro RR).

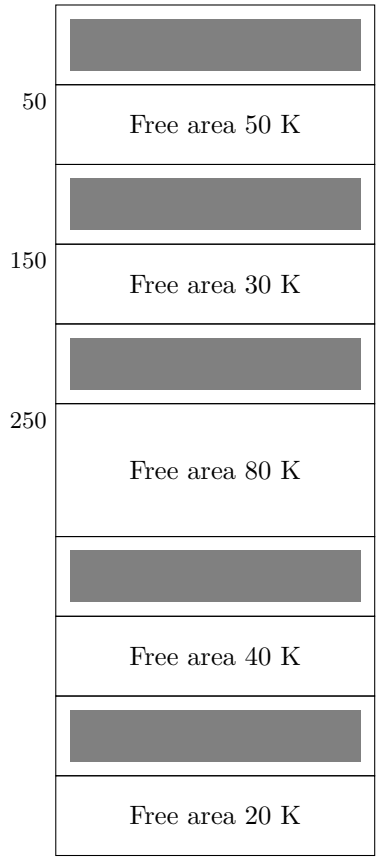
5 Allocazione non contigua

Si ha **allocazione non contigua** della memoria se ogni processo può essere allocato in più aree di memoria non adiacenti. Ogni porzione del processo che viene allocata in un'area contigua è detta **componente**.

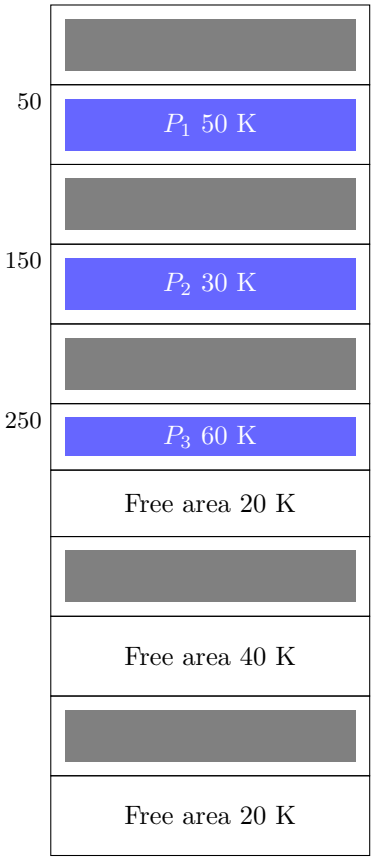
- Per allocare in memoria un processo di k byte è sufficiente che lo spazio complessivamente disponibile sia $\geq k$: non serve una singola area libera in grado di contenere l'intero processo. Perciò, il problema della frammentazione è in gran parte risolto.
- La protezione della memoria e la traduzione da indirizzi logici a indirizzi fisici sono più complicate rispetto al caso dell'allocazione contigua (non bastano LBR/UBR e RR).

5.1 Idea

Per illustrare l'idea dell'allocazione non contigua, si suppone che lo stato attuale della memoria sia il seguente:



Quando viene allocato un processo, esso viene diviso in componenti, in base alla dimensione delle aree libere disponibili. Ad esempio, se si alloca un processo P da 140 K, questo viene diviso in tre componenti (P_1, P_2 e P_3):



Si assume poi che il processo P abbia una variabile `x`, situata all'indirizzo logico 51448 (cioè 51448 byte dopo l'inizio del programma). Se un'istruzione fa riferimento a `x`, bisogna determinare l'indirizzo fisico che tale istruzione deve considerare:

1. la componente P_1 ha dimensione 51200 byte (50 K), cioè contiene gli indirizzi logici da 0 a 51199;
2. la componente P_2 ha dimensione 30720 byte (30 K), quindi contiene gli indirizzi logici da 51200 a 81919, tra i quali è compreso anche l'indirizzo di `x`, 51448.

Allora, l'indirizzo logico 51448 si trova nella componente P_2, con offset $51448 - 51200 = 248$ dall'indirizzo fisico iniziale di P_2, che è 150 K.

Questa traduzione da indirizzo logico a indirizzo fisico viene effettuata dalla MMU, che per farlo accede alle informazioni relative all'allocazione di P memorizzate dal SO: in particolare, sfrutta una tabella in cui, per ogni componente, sono indicati l'indirizzo fisico iniziale e la dimensione:

Indirizzo	Dimensione
50 K	50 K
150 K	30 K
250 K	60 K

Con questa tecnica, però, ogni accesso alla RAM richiederebbe molti altri accessi e calcoli per effettuare la traduzione dell'indirizzo. Per questo motivo, essa non è impiegata dalle implementazioni reali, che usano invece metodi più efficienti, come ad esempio la *paginazione*.