Azzolini Riccardo 2019-11-14

Gestione della memoria

1 Obiettivi del SO

Nell'ambito della gestione della memoria, gli obiettivi del SO sono:

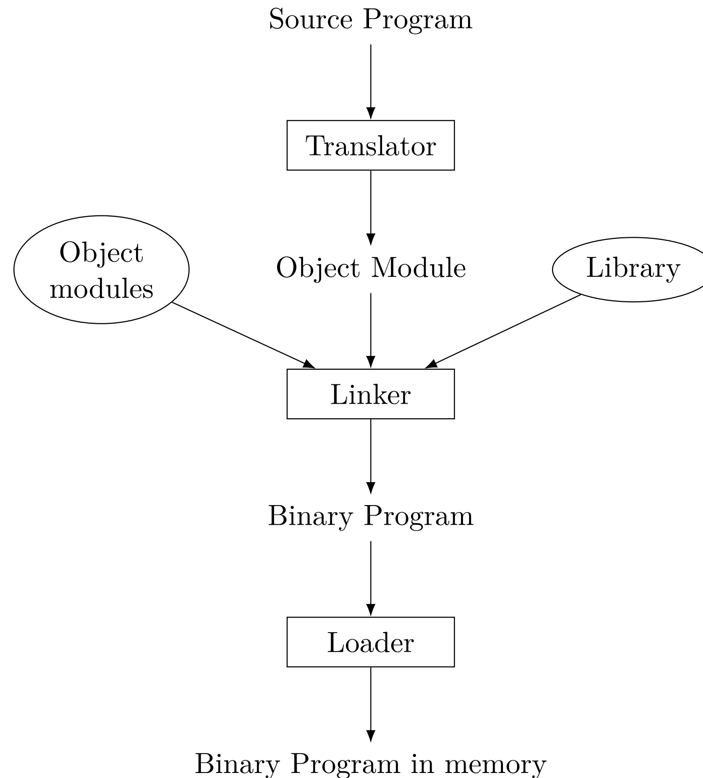
1. allocare più processi in memoria, per avere parallelismo,[1] e migliorare così la performance del sistema;
2. proteggere la memoria dei processi, impedendo che ciascuno di essi acceda alle aree degli altri processi.

Se il SO offre la **memoria virtuale**, all'obiettivo 1 si aggiunge la possibilità di eseguire al tempo stesso più processi di quanti ce ne starebbero in RAM.

2 Traduzione, linking, e loading

Il passaggio da programma sorgente a programma binario in memoria è composto da tre fasi: **traduzione**, **linking**, e **loading**:

[1] tra l'uso della CPU e l'attività I/O, nei sistemi uniprocessore, e anche tra l'uso di CPU da parte di più processi, nel caso dei sistemi multiprocessore.



1. **Source program**: il programma sorgente, scritto in un linguaggio come assembly, C, ecc. (non in linguaggio macchina).
2. Il **translator** (assembler/compilatore) riceve in input il programma sorgente e produce un **object module** (*modulo oggetto*): un programma in linguaggio macchina che solitamente non è ancora eseguibile, in quanto può contenere riferimenti a funzioni/dati definiti in altri moduli e/o librerie, i cui indirizzi non sono ancora noti.
3. Il **linker** collega i vari moduli oggetto e le librerie, producendo il programma eseguibile. In questa fase, gli indirizzi di memoria assegnati dal traduttore possono essere **rilocati** (cambiati), per evitare overlapping con gli indirizzi di altri moduli/librerie.
4. Quando si vuole eseguire il programma, il **loader** lo carica in memoria, associandolo a un processo. Può essere necessario effettuare un'ulteriore rilocazione degli indirizzi di memoria generati dal linker, per evitare overlapping con altri programmi già presenti in memoria.

2.1 Esempio

Il seguente programma P è scritto in un ipotetico linguaggio assembly:

```
        START   500
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A
LOOP    ...
        ...
        MOVE    R1, ALPHA
        BC      ANY, MAX
        ...
        ...
        BC      LT, LOOP
        STOP
A       DS      1
TOTAL   DS      1
        END
```

- `START 500` specifica che il programma occupa gli indirizzi di memoria a partire da 500.
- `ENTRY` dichiara un simbolo (variabile o etichetta) definito in questo modulo, ma visibile anche da altri moduli.
- `EXTRN` dichiara un simbolo definito in un altro modulo. L'indirizzo di tale simbolo dovrà quindi essere determinato dal linker.
- In fondo al programma vengono definite le variabili `A` e `TOTAL`.
- Il programma contiene varie istruzioni, che usano diversi tipi di operandi:
  - `READ`: una variabile;
  - `MOVE`: un registro e una variabile;
  - `BC`: una condizione e un'etichetta.

Per eseguire la traduzione, si assume un linguaggio macchina in cui ogni istruzione ha la forma

opcode operand1 operand2

e occupa 4 byte, cioè 32 bit, rappresentabili con 8 cifre esadecimali:

1. *opcode* occupa 8 bit (2 cifre esadecimali):

Istruzione	*opcode*
`MOVE`	04
`BC`	06
`READ`	09

2. *operand1* occupa 4 bit (1 cifra esadecimale), e solitamente è un registro, ma nel caso di `BC` codifica un possibile valore del Condition Code:

CC	*operand1*
LT	1
ANY	6

3. *operand2* occupa 20 bit (5 cifre esadecimali), ed è un indirizzo di memoria, corrispondente a una variabile o a un'etichetta.

Inoltre, si assume che la memoria sia indirizzata a parole di 4 byte (cioè che ogni indirizzo corrisponda a 4 byte di RAM).

Alcune delle istruzione macchina che l'assembler genera per il programma P, qui riportate di fianco alle istruzioni assembly corrispondenti, sono:

```
        START   500
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A               09 0 00700
LOOP    ...
        ...
        MOVE    R1, ALPHA       04 1 00000
        BC      ANY, MAX        06 6 00000
        ...
        ...
        BC      LT, LOOP        06 1 00504
        STOP
A       DS      1
TOTAL   DS      1
        END
```

- L'indirizzo dell'etichetta `LOOP` è `00504`, perché il programma inizia a `00500`, e ci sono 4 istruzioni prima di `LOOP`.
- Si suppone che l'indirizzo di `A` sia `00700`.
- Nella traduzione di `READ A`, *operand2* è direttamente l'indirizzo di `A`, che l'assembler conosce perché la variabile è definita nello stesso modulo. Lo stesso vale per l'indirizzo dell'etichetta `LOOP` nella traduzione di `BC LT, LOOP`.
- Per la traduzione di `MOVE R1, ALPHA` e `BC ANY, MAX`, gli indirizzi del secondo operando sono al momento sconosciuti (dato che il programmatore ha "promesso" che questi simboli verranno definiti in altri moduli, dei quali l'assembler non è a conoscenza), quindi *operand2* viene temporaneamente impostato a `00000`: gli indirizzi corretti verranno poi inseriti dal linker.

Si assume poi di avere un altro modulo, P',

```
        START   200
        ENTRY   ALPHA
        ENTRY   MAX
        ...
        ...
```

che:

- contiene le definizioni di `ALPHA` e `MAX`;
- ha in totale 400 parole, quindi occupa gli indirizzi da 200 a 599.

Quando viene eseguito il linking di P e P', viene individuato un overlapping di indirizzi: P' finisce a 599, ma P inizia a 500. Il linker deve allora effettuare una rilocazione. In questo esempio, si ipotizza che:

- P' venga lasciato all'indirizzo 200;
- P venga spostato all'indirizzo 600.

Di conseguenza, oltre a inserire gli indirizzi di `ALPHA` e `MAX`, il linker deve anche modificare gli indirizzi interni a P, già inseriti dall'assembler, incrementandoli di 100:

```
        START   600                         // Modificato
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A               09 0 00800 // Modificato
LOOP    ...
        ...
        MOVE    R1, ALPHA       04 1 00201 // Inserito
        BC      ANY, MAX        06 6 00202 // Inserito
        ...
        ...
        BC      LT, LOOP        06 1 00604 // Modificato
        STOP
A       DS      1
TOTAL   DS      1
        END
```

L'output del linker viene salvato sul disco. Successivamente, quando è richiesta l'esecuzione del programma, il loader lo deve caricare in memoria. Per questo esempio, si suppone però che la memoria agli indirizzi da 200 a 299 sia già occupata. Allora, il loader effettua una rilocazione, aggiungendo 100 a ogni indirizzo contenuto in P e in P':

```
        START   700                           // Modificato
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A               09 0 00900 // Modificato
LOOP    ...
        ...
        MOVE    R1, ALPHA       04 1 00301 // Modificato
        BC      ANY, MAX        06 6 00302 // Modificato
        ...
        ...
        BC      LT, LOOP        06 1 00704 // Modificato
        STOP
A       DS      1
TOTAL   DS      1
        END
```

2.2 Osservazioni

- L'overlapping tra gli indirizzi di moduli/programmi diversi è normale e inevitabile, perché essi possono essere scritti da programmatori diversi, ciascuno dei quali non può conoscere gli indirizzi usati da tutti gli altri programmi esistenti.
- Quando un programma è scritto in un linguaggio ad alto livello, l'indirizzo iniziale viene scelto dal compilatore, che non è a conoscenza degli altri moduli.
- Un programma non viene sempre caricato nella stessa zona di memoria, dato che le zone disponibili variano in base a quali altri programmi sono in esecuzione.

Per questi motivi, la rilocazione (sia in fase di linking, che di loading) è indispensabile.

2.3 Rilocazione e linking dinamici

Nell'esempio precedente, il loader ha eseguito una **rilocazione statica**, modificando gli indirizzi del programma prima dell'esecuzione. In alternativa, si può avere la **rilocazione dinamica**, che sfrutta un opportuno supporto hardware, costituito ad esempio da un Relocation Register, e da una MMU (Memory Management Unit) che somma il valore del RR all'indirizzo richiesto quando viene eseguita un'istruzione di accesso alla memoria.

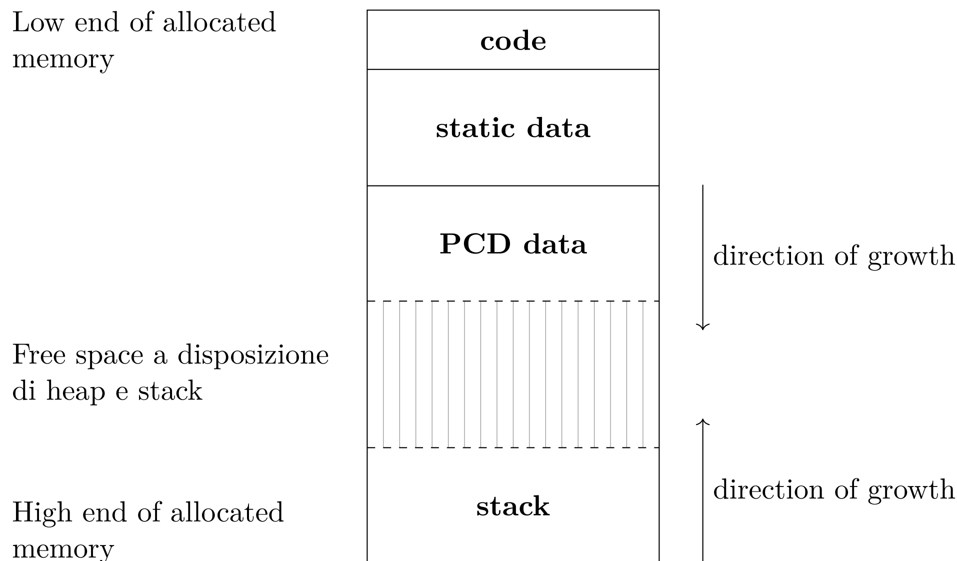
Anche il linking è statico nell'esempio precedente. Con il *linking dinamico*, invece, l'associazione degli indirizzi tra moduli viene creata solo quando serve: il linker sostituisce ogni riferimento a un simbolo definito in un modulo esterno con la logica per effettuare il calcolo dell'indirizzo corrispondente durante l'esecuzione del programma. Questo è il caso, ad esempio, delle DLL.

3 Modello di allocazione della memoria

La memoria che il SO alloca per un processo è suddivisa in aree che hanno funzioni diverse:

- il *codice* e i *dati statici* (le variabili globali) sono allocati in un'area di dimensione statica;
- l'area di *heap* e lo *stack* condividono un'area di dimensione statica, ma individualmente hanno dimensione variabile, e "crescono" in direzione opposta, occupando progressivamente uno spazio "vuoto" che si trova in mezzo.

Infatti, la dimensione dello stack varia con la creazione e rimozione dei record di attivazione delle procedure, mentre la dimensione dello heap è variabile perché quest'area contiene i **dati PCD** (*Program Controlled Dynamic data*), cioè i dati creati dinamicamente dal programma mediante appositi costrutti forniti dal linguaggio di programmazione (come ad esempio `new` in Java, o `malloc` e `calloc` in C).



3.1 Esempio di dati PCD in C

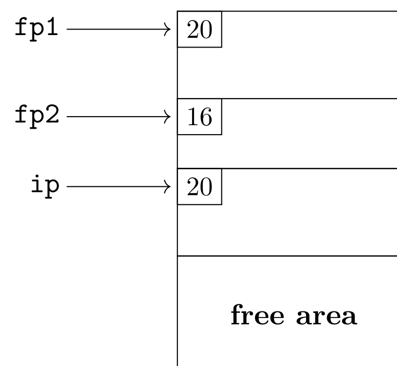
```c
// ...
float *fp1 = (float *) calloc(5, sizeof(float));
float *fp2 = (float *) calloc(4, sizeof(float));
int *ip = (int *) calloc(10, sizeof(int));
free(fp2);
// ...
```

- `fp1` e `fp2` sono puntatori a `float`, mentre `ip` è un puntatore a `int`. Queste variabili potrebbero trovarsi in un record d'attivazione (e quindi nello stack), se sono locali, oppure nell'area dati statici, se sono globali.
- `calloc` richiede l'allocazione di un'area nello heap, e restituisce un puntatore a tale area. La dimensione dell'area da allocare è specificata dai due parametri, che sono:

  1. il numero di elementi da allocare;
  2. la dimensione di un singolo elemento.

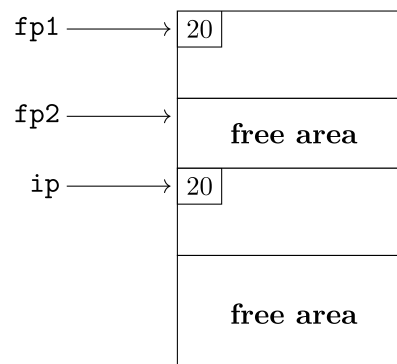
  Quindi, per esempio, `calloc(5, sizeof(float))` alloca un'area che può contenere 5 valori di tipo `float`.
- `free` chiede di liberare un'area di memoria allocata in precedenza.

Dopo le tre chiamate `calloc`, lo stato dello heap potrebbe essere, ad esempio:



Ogni area allocata ha un campo contenente la dimensione, che serve per facilitare la deallocazione.

Dopo la chiamata `free`, l'area di memoria puntata da `fp2` diventa libera:



3.2 Heap allocator e RTS

Le procedure di gestione dello heap sono funzioni di libreria. In particolare, esse fanno parte del **RTS** (**RunTime Support**, cioè supporto a tempo di esecuzione) del linguaggio, e vengono collegate al programma dal linker.

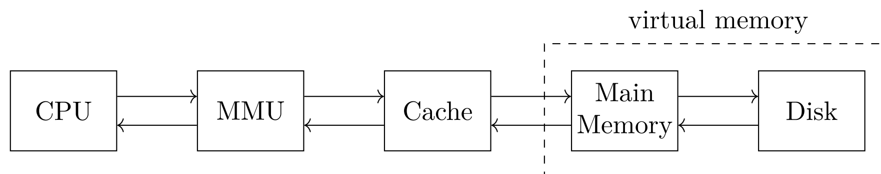
Le implementazioni delle procedure per allocare/deallocare memoria nello heap (come ad esempio `malloc`/`calloc` e `free`) sono chiamate, rispettivamente, **heap allocator/deallocator**.

Nota: Oltre alla gestione dello heap, il RTS di un linguaggio può contenere anche altre funzioni. Ad esempio, in Java, esso comprende anche il *garbage collector*, che libera la memoria allocata nello heap quando non esistono più puntatori che fanno riferimento a essa.

4 Gerarchia di memoria

Nel mondo ideale, l'unità di memoria centrale sarebbe abbastanza veloce da non rallentare l'operazione della CPU, e, al tempo stesso, in grado di ospitare grosse quantità di dati.

In realtà, invece, le unità di memoria più veloci sono anche più costose, e quindi realizzate con capacità minori. Perciò, si fa uso di una **gerarchia di memoria**, costituita sia da unità più piccole e veloci, che da unità più grandi e lente.



I programmi contengono riferimenti alle aree di testo, dati, stack, e heap. Ciascuna di queste può essere memorizzata parzialmente a vari livelli della gerarchia, ma ciò avviene in modo trasparente al programmatore, che ragiona come se tutte queste aree si trovassero fisicamente su una singola unità di memoria.

- Nei sistemi con memoria virtuale (che è presente in tutti quelli moderni), un'area di memoria può essere parzialmente in RAM e parzialmente sul disco. La memoria virtuale è trasparente al programmatore in quanto gestita interamente dal SO.
- Per velocizzare gli accessi, una parte della RAM viene ricopiata nella *cache*, che è più piccola e veloce: quando si esegue un accesso, l'indirizzo richiesto viene prima cercato in cache, e solo se non viene trovato si accede poi alla RAM. La cache è

trasparente sia al programmatore che al SO,[2] perché viene gestita direttamente a livello hardware.

[2]È comunque possibile sviluppare i programmi/SO in modo che facciano uso più efficiente della cache.