

Notes Software Engineering

Brendon Mendicino

March 2, 2023

Contents

1	Memoria	3
---	---------	---

1 Memoria

La memoria si compone si suddivide in:

- registri;
- cache
- memoria primaria;
- memoria secondaria;

Uno principali problemi di accesso alla memoria è quello di assicurarne la sua protezione, ovvero evitare che un programma in memoria riesca ad accedere alle zone di memoria di altri programmi. Una volta fatto il bootstrap del SO, sia esso che i programmi verranno caricati in memoria, per evetire che ogni processo abbia una vista al di fuori del suo scope, si possono usare dei controlli a livello della CPU, esistono dei registri chiamati **base** e **limit**, che attraverso dei meccanismi, riescono ad effettuare la protezione della memoria. Esiste anche un altro problema che è quello **relocation**, consiste nel come mappare gli indirizzi delle varie istruzioni di salto e dei vari puntari una volta che il programma viene caricato in memoria, se si utilizzaro degli indirizzi statici (creati solo in fase di compilazione e prendondo come riferimento di inizio del programma l'indirizzo 0), una volta che questo viene caricato in memoria le istrizioni non punterebbero più alle label corrispondenti ma sempre nello stesso punto, dove ad esempio si trova un altro programma. Per questo motivo quando si utilizzano valori di registri vengono sommati al base register, mentre le boundry del programma in memoria vengono salvate nel limit. Binding ... Si fa:

- **in compilazione:** fatto quando molto semplice, ad esempio quando esistono solo due programmi;
- **in fase di load:** viene fatta la rilocazione durante il caricamento in memoria;
- **in esecuzione:** viene fatto il binding degli indirizzi in modo dinamico;

Per risolvere questo problema ci si affida all'hardware, che è incaricato di fare la traduzione: l'indirizzo rimane lo stesso (logico) all'interno del processore, prima di arrivare all'address bus viene tradotto in indirizzo fisico, esiste dunque una dicotomia di indirizzo logico e fisico, in questo modo quando si scrive un programma, l'indirizzo parte sempre da 0. Esistono due tipi di indirizzamento che sono di tipo logico, dove l'intervallo degli indirizzi utilizzabili è logico, ed un indirizzamento fisico, dove il range è limitato dalla memoria del sistema. Per effettuare questa traduzione da indirizzo fisico a indirizzo logico e viceversa si utilizza una **MMU** (Memory Management Unit). Il modo più facile per realizzare una MMU, è quello di usare un **relocation register**, ovvero un registro che contiene il valore da aggiungere un indirizzo logico

per fare un indirizzo fisico, il modello più semplice di MMU è fatto da un sommatore ed un comparatore.

Per aumentare le prestazioni ed usare la memoria in modo più efficiente si possono usare delle tecniche dinamiche.

- Si parla di **dynamic loading** quando, un programma viene caricato in memoria principale in modo frammentato, utilizzando solo i componenti che effettivamente vengono chiamati;
- Si parla di **dynamic linking** quando i file che contengono le funzioni che devono essere linkate (come le librerie standard) non vengono inserite all'interno dell'eseguibile, ma gli indirizzi vengono risolti in modo dinamico durante l'esecuzione;
- Il **link statico** è quando si crea un eseguibile con tutte le funzioni dentro, di fatto il loader carica tutto quando in memoria.

Il **dynamic loading** vuol dire che una routine non è caricata finché non è necessaria, questo può essere fatto quando il programmatore ne è consapevole, infatti il processo di load non è trasparente:

```
1 void myPrintf(**args) {  
2     static int loaded = 0;  
3     if (!loaded) {  
4         load("printf");  
5         loaded = 1;  
6     }  
7     printf(args);  
8 }
```

Il **linker-assisted DL** si usa una chiamata fasulla che prima chiama la load, usando una **stub**.

Le **shared libraries** sono in grado di condividere le risorse, infatti se più processi utilizzano la stessa funzione essa viene messa a disposizione e per ogni nuova chiamata non ci sarà bisogno di chiamare una load.

Come si alloca memoria per un programma (immagine)? La più semplice è l'allocazione contigua, dove si vede la RAM come due partizioni, una per il SO una per i processi (indirizzi più alti), per caricare un processo si parte da un indirizzo di inizio ed un indirizzo di fine, la MMU vista prima funziona solo con i casi di allocazione contigua (basilare). L'**allocazione contigua con partizione variabile**: quando ci sono più buchi ci sono delle politiche differenti per inserire nuovi programmi:

- **first-fit**: il primo che si trova;
- **best-fit**: il buco con la dimensione più piccola;
- **worst-fit**: il buco con la dimensione più grande;

La frammentazione è sparsità dei buchi all'interno della memoria. Si dice **esterna** perché è al di fuori dei processi, si dice **interna** quando è interna al processo, ovvero che ha più memoria di quello che serve. La frammentazione ha bisogno di **compattazione**, il SO sposta i pezzi e poi si riparte, partizione della memoria in zona libera e zona occupata, per fare una **deframmentazione** (o compactazione) vuol dire creare solo due partizioni (parte processi e zona libera), per effettuare la compactazione bisogna che i processi si possano spostare, inoltre un processo non può essere spostato se in quel momento il processo sta effettuando delle operazioni di IO, una soluzione per il problema dell'IO è il **latch job** in cui quella parte non può essere spostata, oppure si utilizza un **buffer IO del kernel**, si fa IO solo in buffer del sistema operativo.

Con **backing store** si definisce uno spazio della memoria secondaria in cui non si salvano programmi ma vengono immagazzinati dei dati che altrimenti dovrebbero andare in RAM.

La **Paginazione** risolve i problemi della allocazione continua, rimuovendo l'allocazione contigua al suo posto si utilizzano delle pagine, che sono l'unità minima di allocazione, questo risolve il problema della frammentazione.

Definition 1.1 – Pagine

Le partizioni della memoria logica.

Definition 1.2 – Frame

Le partizioni della memoria fisica.

Definition 1.3 – Blocco

...

Tipicamente la loro dimensione è un multiplo di due, sia pagine che frame hanno la stessa dimensione, da qualche parte andranno salvate le informazioni di mapping tra pagine e frame, si utilizza un **frame table**, dove ogni riga corrisponde a una mappatura. Anche utilizzando la paginazione si ha frammentazione interna, l'ultima pagina soffrirà di frammentazione.

Un indirizzo generato dalla CPU si divide in:

- **numero di pagine (p):**
- **numero di offset (d):**
- **numero di frame (f):**

Un indirizzo è composto da: (p-d, d)

Example 1.1

Se si hanno frame piccoli diminuisce la frammentazione ma aumentano anche le righe della tabella.

Nelle righe della tabelle si aggiungono dei bit in più:

- parte di protezione, specifica che la parte di codice non può essere scritta;
- **modify bit**: pagina modifica;
- **page present/absent**:

Come si implementa una page table? La page table si trova in memoria RAM, per sapere dove si trova si utilizzano due registri: **page table base register** e **base table lenght register**, infatti la base table si trova in memoria contigua. Per velocizzare questa operazione si utilizza che TLB che è più piccola e si trova direttamente sulla CPU, si usa la **Translation Look-aside Buffer** (TLB), un tipo di memoria in cui si accede per contenuto. Si aggiunge anche un'altra informazione **ASID** in cui viene salvata l'informazione del processo a cui la pagina appartiene, se non è presente questa informazione si i processi si contentano la TLB. Quando avviene un **TLB miss** se reinserisce nella TLB e poi si ritenta, utilizzando una politica scelta.

Lo schema finale è la combinazione di TLB e page table.

Il **Tempo di accesso effettivo** (EAT) in memoria è il tempo che mi costa accedere alla RAM,