1 Lezione del 28-03-25

Riprendiamo il discorso della memoria virtuale.

1.0.1 BASE e LIMIT

Avevamo detto che intendeveamo dividere la memoria utente fra processi, senza dover ricorrere al caricamento da disco della memoria relativa ad ognuno di essi.

Decidiamo quindi di dotare la CPU di due registri, **BASE** e **LIMIT**, che puntano rispettivamente all'inizio e alla fine della memoria dedicata al proesso, che questa dovrà controllare per prevenire accessi all'esterno della zona definita quando ci si trova in modalità utente.

Chiaramente potrebbero esserci problematiche rispetto a quali indirizzi i singoli programmi vogliono usare: questi non potranno chiaramente usare salti a posizioni arbitrarie in memoria.

Una prima soluzione può essere quella di rendere il **PIC** (*Position Indipendent Code*), cioè codice indipendente dalla posizione (dove ad esempio le `CALL` e le `JUMP` saltano ad *offset*, e non ad indirizzi assoluti). Un primo problema di questo approccio è che costringe i programmi di stare all'intero di una zona di $\sim$ 4GB, in quanto gli offset sono su 32 bit (e non è nemmeno detto che il kernel dedicherà ad ogni processo la stessa quantità di memoria).

Un altro approccio può essere quello di realizzare un *caricatore rilocante* (ad esempio impementato in MS-DOS): si fa in modo che il collegatore lasci gli indirizzi non completamente specificati, e si definiscono una volta nota la posizione al partire da cui il processo verrà caricato (semplicemente incrementando gli indirizzi a partire da 0 in modo che puntino alla stessa posizione relativa al nuovo punto di inizio del programma).

Notiamo però che una problematica si presenterà sempre se intendiamo spostare processi fra memoria e disco in posizioni diverse, in quanto un processo potrebbe ad esempio poter aver messo un indirizzo assoluto in un registro, pianificando di effettuarci successivamente un accesso.

Un'altra problematica è che abbiamo perso l'accesso alla *memoria condivisa*, a meno di non sovrapporre le regioni definite dal BASE e LIMIT di due processi, sempre però limitandosi a due regioni molto specifiche di soli due processi.

Decidiamo quindi di usare il seguente approccio: ogni accesso in memoria ad un indirizzo x richiesto dal programma viene trasformato in un accesso a BASE $+ x$. In questo modo il collegatore potrà far partire ogni programma dall'indirizzo 0: ogni indirizzo usato da quel processo non sarà quindi altro che un offset a partire dall'inizio della regione di memoria dedicata a tale processo.

Risolveremo così i problemi relativi agli indirizzi assoluti, ma resterà il problema della dimensione del codice e della memoria condivisa.

Trascurando per adesso questi due dettagli, vediamo che abbiamo effettivamente realizzato una **memoria virtuale**, dove una certa funzione f mappa indirizzi *virtuali* $x_1, ...$ ad indirizzi *fisici* $v_1, ...$:

$$\begin{array}{ll} x_1 & \\ x_2 & \xrightarrow{f(x)=\text{BASE}+x} \\ x_3 & \end{array} \begin{array}{l} v_1 \\ v_2 \\ v_3 \end{array}$$

Vediamo però che spostare processi nella memoria comporta comunque un gran dispendio di risorse in quanto la memoria dedicata ad un processo può raggiungere di-

mensioni considerevoli, problema che viene solo moltiplicato quando si inizia a lanciare sempre più processi.

1.0.2 Paginazione

Questo problema, assieme in qualche modo agli altri due che avevamo lasciato in sospeso, può essere risolto agendo sulla funzione f. Decidiamo infatti di dividere la memoria processo in una serie di **pagine**, di dimensione fissa (prendiamo 4 KB), che possono prese in qualsiasi ordine dalla memoria centrale, ad unità sempre da 4 KB che chiamiamo **frame** .

A questo punto non avremo più bisogno di **continuità** nella memoria dedicata ai processi, cioè non avremo problemi di *frammentazione esterna*, anche se in qualche modo avremo introdotto *frammentazione interna* dove ogni processo dovrà ottenere memoria a "pacchetti" di 4 KB (che è comunque più vantaggioso).

BASE e LIMIT non saranno chiaramente abbastanza per gestire una situazione di questo tipo, e avremo quindi bisogno di una **tabella di corrispondenza**, allocata da qualche parte in memoria, che contenga una riga per ogni pagina, contenente il frame corrispondente alla pagina. Ogni indirizzo x sarà quindi scomposto in due valori, il **numero di pagina** e l'**offset di pagina** al suo interno. Il numero di pagina verrà quindi trasformato nel frame corrispondente alla pagina, e si potrà procedere all'accesso.

Ogni processo avrà quindi bisogno della sua tabella di corrispondenza personale, che pensiamo per adesso di poter semplicemente caricare e scaricare da memoria assieme al processo stesso.

Risolveremo quindi anche il problema della memoria condivisa, in quanto basterà mettere alcuni frame in comune fra più processi (starà al kernel tenere conto di quali frame sono in uso da quali processi e cosi via). Inoltre, agendo sulle tabelle, possiamo anche immaginare come questo sistema porterebbe almeno via di un livello di astrazione l'accesso a regioni di memoria più grandi di 4 GB.

1.0.3 Memory Management Unit

Aggiungiamo quindi, lato hardware e posta fra processore e cache, un nuovo componente detto **MMU**, *Memory Management Unit*, che tiene conto delle tabelle di corrispondenza e trasforma tramite esse le pagine degli indirizzi nei frame giusti.

Assumeremo che tutti gli indirizzi che la CPU genera saranno indirizzi virtuali, e che tutti gli indirizzi che escono dalla MMU saranno indirizzi fisici. Decidere che tutti gli indirizzi generati dalla CPU sono virtuali solleva una questione riguardo al kernel: idealmente, vorremmo che questo veda l'interezza della memoria, senza paginazione. Prendiamo quindi la sua memoria come in testa allo spazio di memoria, con le regioni successive dedicate ai frame di pagina dei processi.

Potremmo allora avere una tabella dedicata al solo kernel, che tiene conto di tutto lo spazio indirizzabile. Inoltre, potremmo prendere la tabella del kernel come *identiva*, cioè dare al kernel la visione della sua memoria *così com'è*.

Corrediamo allora la tabella di pagina introducendo:

- **P:** un bit di presenza, che definisce l'esistenza o meno di una traduzione per quell'indirizzo: nel caso di accesso a pagine non traducibili si genera un ecceione, detta **page fault**, che comporta il caricamento della pagina richiesta o la terminazione forzata del programma per **segmentation fault**.

Ad esempio, se scegliamo 0 come la codifica del null pointer, vogliamo che la prima pagina (o le prime pagine, se vogliamo essere più larghi con accessi a strutture puntate da null pointer, che potrebbero avere offset negli struct anche considerevoli) sia non presente, e quindi si traduca in eccezione prima di effettuare accessi chiaramente erronei;

- **S/U:** *Sistema/Utente*, indica se una pagina è accessibile o meno ad un processo utente;
- **R/W:** *Read/Write*, indica se una pagina è accessibile solo in scrittura o solo in lettura per un certo processo. Questa può essere utile ad esempio per la sezione *text* del programma, che ricordiamo contiene il codice e non vogliamo venga modificata;
- **PCD** e **PWT**: indicano se ignorare completamente la cache (PCD) o se adottare una politica di scrittura *write-through* (PWT). Questo può essere utile nel caso di dispositivi mappati in memoria (come l'APIC o l'adattatore video);