1 Lezione del 24-03-25

1.1 Primitive

Abbiamo introdotto il concetto di primitiva, cioè di routine svolte dal sistema al servizio di un dato programma. Queste verranno implementate come gestori di interruzioni, quindi non propriamente funzioni, in quanto implicano un passaggio di contesto. Ciò nonostante, in un linguaggio come il C++ le primitive saranno comunque rappresentate da funzioni, dette *funzioni interfaccia*, scritte in assembly e che hanno il solo compito di usare la funzione `INT` con i parametri necessari alla chiamata di una specifica primitiva primitiva.

1.1.1 Primitiva di creazione di un processo

Abbiamo visto che la creazione di un processo consiste nell'inizializzazione dalla memoria ad esso dedicata in contesto utente e sistema, alla creazione del suo descrittore e all'inserzione di questo in pila *"pronti"*. Se la pila è rappresentata come una linked list, l'operazione dovra quindi essere quella di un *inserimento in testa*.

Notiamo che questa operazione non può essere divisa da altre interruzioni, in quanto richiede necessariamente almeno due passaggi, dove fra un passaggio e l'altro la lista viene lasciata in uno stato inconsistente:

- Prima si fa puntare il processo al resto della lista;
- Poi si fa puntare il puntatore della lista pronti al processo inserito.

anche invertendo l'ordine delle operazioni, dopo la prima la lista è inconsistente (in questo caso perché il processo inserito non viene effettivamente visto, nel caso opposto perché non vengono visti tutti gli altri).

Nel caso di routine di sistema basterà abbassare il flag `IF`, disabilitando effettivamente le interruzioni, durante tutta la durata della routine. A questo punto basterà evitare di generare eccezioni, e non usare mai l'istruzione `INT`, per ottenere una routine che viene eseguita dal processore nella sua interezza senza il rischio di interruzioni. Chiamiamo codice di questo tipo **codice atomico**. Il kernel Linux, ad esempio, *non* è atomico.

1.1.2 Disposizione delle primitive

La memoria del calcolatore conterrà in qualsiasi momento la tabella IDT, di cui abbiamo detto le prime 32 entrate rappresentano le eccezioni. Siamo quindi liberi di usare i gate dal 33 in poi per implementare le primitive. Per queste primitive dobbiamo impostare i parametri:

- **P:** 1, per attivare il gate;
- **L:** sistema, in quanto le primitive devono essere svolte a livello sistema;
- **DPL:** utente, in quanto le primitive devono essere accessibili all'utente;
- L'indirizzo effettivo della routine, implementata (in assembly, serve `IRET`), che deve trovarsi da qualche altra parte;
- **I/T:** tipo interrupt (interruzioni esterne mascherabili disabilitate).

Notiamo che l'interruzione esterna non mascherabile 2 è comunque in grado di bloccare le nostre istruzioni atomiche. Questo non è importante, in quanto abbiamo detto la useremo per casi particolarmente catastrofici (dove magari la salvaguardia dei dati dell'utente e del sistema e di maggiore priorità rispetto allo stato dei processi).

La struttura della routine sarà quindi tipicamente:

```
primitiva:
  CALL salva_stato
  CALL c_primitiva
  CALL carica_stato
  IRETQ
```

dove `c_primitiva` è una funzione, scritta in C++, che termina con una `RET` e lascia quindi che `primitiva` restituisca il controllo all'utente con `IRETQ`.

Per chiamare la primitiva da C++, come abbiamo detto, ci doteremo di una funzione di interfaccia del tipo:

```
primitiva_i:
  INT $ tipo %il tipo di primitiva
  RET
```

1.1.3 Passaggio di parametri alla primitiva

Supponiamo di voler passare dei parametri alla nostra primitiva. La funzione di interfaccia dovrà semplicemente essere modificata per accettare dati parametri (`primitiva_i(params...)`).

A questo punto la `primitiva_i` potrà svolgere il passaggio effettivo sfruttando i registri, solitamente il solo registro `%EAX` (in quanto `salva_stato` non modifica i registri).

1.1.4 Passaggio di parametri dalla primitiva

Per avere una restituzione di parametri da parte della primitiva la situazione è più complicata, in quanto abbiamo una chiamata a `carica_stato` prima del ritorno della primitiva per `IRETQ`.

Abbiamo però accesso al contesto di processo, nel descrittore di processo, e possiamo quindi modificare i registri che ci interessano direttamente lì.

1.2 Semafori

Per gestire l'accesso condiviso ad una risorsa, nel nostro kernel adotteremo il meccanismo dei **semafori**.

Introdotti da Dijsktra nel 1962, questi si possono meglio modellizzare come una scatola piena di gettoni: ogni utente può mettere un gettone o prelevare un gettone dalla scatola, con la condizione che questa operazione sia atomica: se si tenta di prendere un gettone che non esiste, si resta in attesa finché quel gettone non viene effettivamente immesso nella scatola.

I problemi che vogliamo risolvere sfruttando i semafori sono effettivamente die due categorie:

- Problemi di **mutua esclusione:** assicurarsi che solo un processo possa accedere ad una risorsa in un dato momento.

  In questo caso si associa un gettone alla risorsa: accedere alle risorsa significa prendere il gettone, restituire la risorsa significa reinserire il gettone. L'esistenza di un

singolo gettone assicura che solo un processo abbia accesso alla risorsa in un dato momento. Al momento della reimmissione del gettone, il processo che ne vince l'accesso sarà nel nostro kernel quello a priorità più alta.

Notiamo inoltre che un processo che cerca di estrarre un gettone da una scatola vuota (tenta l'accesso ad una risorsa occupata o comunque non disponibile) dovrà aspettare che questa risorsa si renda disponibile: rappresenterà quindi il caso perfetto di **blocco** del processo, che può essere realizzato con **preemption** nei sistemi che la supportoìano;

- Problemi di **sincronizzazione:** esistono più attività, e ci interessa che alcune attività vengano fatte prime di altre (ordinamento *parziale*).

  Prendiamo l'esempio di avere due processi, A e B, e di volerci assicurare che A $\rightarrow$ B. In questo caso creiamo un semaforo associato al processo A, che parte vuoto. A mette il suo gettone nel semaforo quando finisce la sua esecuzione. A questo punto, B preleva il gettone ed esegue. Se B avesse provato ad entrare in esecuzione prima che A avesse terminato, non sarebbe riuscito a prelevare il gettone e avrebbe fallito.

  Nel caso di 2 processi (sempre A e B, con A che scrive e B che legge) che devono scambiare dati fra di loro ciclicamente, potremmo usare 2 semafori per realizzare un *handshake*. Ad esempio, definiamo quelle che effettivamente sono due variabili logiche sfruttando i semafori, che intendiamo come "buffer scritto" e "buffer letto". Il processo A dovrà semplicemente attivare il semaforo "buffer scritto" in fase di scrittura, e il processo B attivare il semaforo "buffer letto" in fase di lettura. Abbassando questi semafori al termine delle rispettive operazioni, e assicurandosi, osservando l'altro semaforo, di poter effettivamente procedere ad una nuova operazione, potremmo realizzare il paradigma desiderato.

Dal punto di vista di implementazione, il kernel fornisce una primitiva `sem_ini(int val)` che inizializza un semaforo con `val` gettoni iniziali, restituendone l'indirizzo. Da qui in poi i processi hanno accesso alle primitive `sem_wait` e `sem_signal`, che si occupano rispettivamente di richiedere e restiture un gettone.