1 Lezione del 30-09-25

Continuiamo la discussione della memoria RAM.

Avevamo introdotto il meccanismo della cache come una sorta di "memoizzazione" dei dati in occasione del primo accesso. In verità, nei moderni processori (dal Pentium in poi) abbiamo due cache separate:

- La **I-cache**, cioè *cache istruzioni*;
- La **D-cache**, cioè *cache dati*;

Il vantaggio di distinguere fra cache istruzioni e cache dati è che la I-cache enon ha bisongo di essere ricopiata in memoria alla fine dell'utilizzo, e probabilmente deve mantenere zone di memoria molto specifiche, per cui ha senso non rallentarne l'operazione chiedendole di mantenere anche informazioni sui dati.

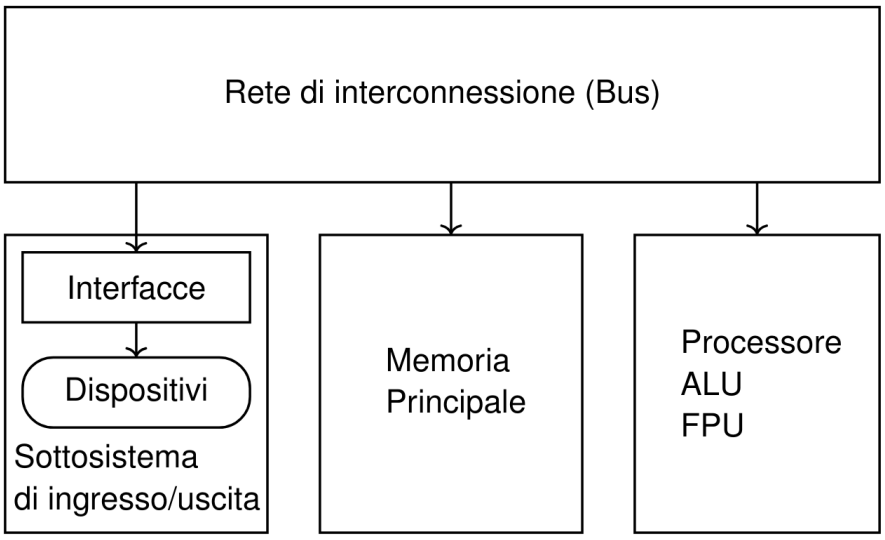
1.0.1 Gerarchie di memoria

Fra registri, RAM, dispositivi a blocchi, ecc... abbiamo visto diverse fonti di *memoria* che un calcolatore può utilizzare. Potrebbe avere senso organizzare queste memorie in una struttura gerarchica, magari per *dimensione crescente* in avanti (e di conseguenza per [velocità] all'indietro):

1. Registri interni;
2. Cache;
3. RAM;
4. Dischi;
5. Nastri.

1.1 Schema a blocchi di un semplice calcolatore

Possiamo quindi, dopo aver visto tutte le componenti che lo compongono, vedere lo schema a blocchi di un semplice calcolatore:



Vediamo quindi come questi componenti comunicano fra di loro:

- La rete di interconnessione (bus) serve tutti (a scapito della direzione delle frecce, può supportare la comunicazione *da* e *a* componenti);
- Il processore e la RAM si trovano sul bus;
- I dispositivi, cioè i trasduttori col mondo esterno, comunicano con il sistema attraverso le loro *interfacce*, che obbedisce da un lato alle regole del bus e dall'altro alle specifiche del dispositivo stesso per permettere la comunicazione.

1.2 Interfacce

Abbiamo visto come fra il calcolatore ed ogni dispositivo si trovi un'apposita *interfaccia*.

Di base, ogni interfaccia è caratterizzata da più registri (accessibili nello spazio di I/O), che possono essere scritti o letti dal calcolatore per dare o ottenere informazioni dal dispositivo. Notiamo che letture e scritture sui registri delle interfacce possono essere distruttive: spesso il dispositivo implementa particolari funzioni che vengono lanciate da operazioni di questo tipo (un registro che si azzera dopo esser letto, ecc...).

Nel caso più semplice, in ogni caso, un'interfaccia dispone di almeno 3 registri:

- Registro di **stato**, che segnala lo stato corrente dell'interfaccia (se è di uscita può segnalare che è pronta a ricevere dati, se di entrata che ci sono dati pronti, ecc...);
- Registro di **controllo**, che permette al calcolatore di comandarne l'operazione (se di entrata può impedire che nuovi dati arrivino in ingresso, ecc...);
- Uno o più **buffer dati**, resi accessibili attraverso un registro di lettura. Solitamente si dice **TBR** (*Transfer Buffer Register*) il registro che accede al buffer di uscita e **RBR** (*Receive Buffer Register*) il registro che accede al buffer di entrata. Nel caso di interfacce di ingresso/uscita TBR e RBR stanno alla stessa porta dello spazio di I/O, e quale viene reso disponibile al processore varia in base al tipo di operazione che esso richiede (TBR per uscita, RBR per ingresso).

1.3 Interruzioni

Veniamo quindi al meccanismo dell'**interruzione**. Nella formulazione originale di Dijkstra queste servivano a risparmiare al processore l'attesa "attiva" (*busy wait*) dei bit di stato delle interfacce, delegando questo invece ad una segnalazione esplicita da parte dell'interfaccia, che viene *gestita* dal processore mettendo in esecuzione un determinato *handler* di interruzione.

Per gestire correttamente le interruzioni abbiamo bisogno di un po' di infrastruttura in più:

- Una nuova fase processore, successiva all'esecuzione, che si occupa di controllare le richieste di interruzioni in arrivo (nei sistemi x86 *la* richiesta, che è stata inoltrata da un sottosistema detto APIC);
- Una zona di memoria dove viene allocata la **IVT** (*Interrupt Vector Table*), che associa ad un indice (cioè il tipo di interruzione) l'inizio dell'handler relativo a tale tipo;
- Una nuova istruzione, `IRET`, che si occupa di ritornare da un gestore di interruzione.

  Non potremmo usare la semplice RET in quanto ogni interruzione salva dello stato aggiuntivo oltre al semplice IP sulla pila: di base, salveremo anche il registro dei FLAG.

1.3.1 Tipi di interruzione

Abbiamo visto nel corso di calcolatori elettronici come il meccanismo delle interruzioni può essere sfruttato per implementare molta più funzionalità di quelle relative alla gestione dei dispositivi. In particolare, i calcolatori moderni dispongono di più tipi di interruzioni:

- Interruzioni **esterne**, del tipo che abbiamo appena visto, che si distinguono ulteriormente in:
  - Interruzioni esterne **mascherabili**, cioè che possono essere ignorate variando il flag IF;
  - Interruzioni esterne **non mascherabili**, cioè che vengono sempre gestite;
- Interruzioni **interne**, cioè lanciate da situazioni interne al processore (eccezioni);
- Interruzioni **software**, che possono essere lanciate dal programmatore attraverso l'apposita istruzione `INT`.

1.4 Meccanismi di protezione

Veniamo quindi ai meccanismi tipici del S/O in sé per sé. Se vogliamo la separazione fra processi e S/O che gestisce quei processi (e quindi ha l'accesso prioritario alle risorse di sistema), dobbiamo separare l'operazione del calcolatore in due modalità principali:

- Modalità **utente**: usata per la normale esecuzione dei programmi, non è possibile accedere a tutte le risorse di sistema;
- Modalità **supervisor**: usata per lo svolgimento delle chiamate sistema (primitive), tutte le risorse di sistema sono disponibili.

Importante è che il passaggio da modo utente a modo supervisor richieda al programma in esecuzione in modo utente di "abbandonare" il controllo, cedendolo ad una primitiva sistema. Vedremo come questo si può implementare agilmente sfruttando il meccanismo di interruzione.

1.5 Componenti del S/O

Vediamo quindi, come abbiamo fatto per l'hardware, quelli che sono i **componenti** del S/O e come questi sono organizzati.

Prendiamo come riferimento un sistema *Unix*, in quanto più semplice ed elegante per i nostri scopi.

Adottando un approccio *top-down*, dove per *top* intendiamo lo spazio dell'utente, vediamo le seguenti componenti:

- L'*userspace*, cioè gli applicativi utente veri e propri;
- Gli strumenti che il S/O fornisce all'utente per la gestione del sistema, cioè:
  - La **shell**;
  - **Compilatore** e **linker**;
  - Le **librerie** sistema (che si rivolgono ad API, ecc...).
- Il **kernel**, cioè la parte del S/O che effettivamente gestisce il sistema. Qui troviamo:

  - Il sottosistema **file**, che gestisce il filesystem su uno o più dispositivi a blocchi;
  - A sua volta il sottosistema file interagisce con i **driver** dispositivo (in particolare coi driver dei dispositivi a blocchi), che hanno il compito di gestire a livello hardware il comportamento dei dispositivi;
  - Inoltre, troviamo il sottosistema **controllo processi**, composto da:
    * Funzionalità **IPC** (*Inter Process Communication*) per la comunicazione fra processi;
    * Lo **scheduler**, che decide quali processi mandare in esecuzione;
    * Il sottosistema di **gestione memoria**, che gestisce lo spazio in memoria principale allocato per ogni processo, interagendo col meccanismo della *memoria virtuale*.

- Infine, il *kernel* si appoggia all'**hardware** della macchina.

1.5.1 Modello gerarchico

Un modello più complesso per S/O potrebbe elaborare su questa struttura, prevedendo più livelli intermedi di kernel che implementano *macchine virtuali* via via più vicine all'hardware. Ognuna di queste fornirà al livello superiore funzioni (effettivamente chiamate sistema) sempre più astratte, che implementeranno nel complesso le chiamate sistema rese disponibili ai processi utente.

1.5.2 Modello client-server

Un'altro modello possibile per sistemi distribuiti in rete è quello di avere più *nodi* collegati alla stessa rete. Ogni nodo disporrà del suo kernel, e in esecuzione su quel kernel avrà uno specifico processo (utente o sistema).

La funzionalità del S/O sarà quindi implementata interrogando la rete per il servizio richiesto: sarà quindi compito della macchina su quella rete che effettivamente implementa tale servizio rispondere e fornire, appunto, il servizio.

In ogni caso non analizzeremo sistemi di questo tipo in questo corso, relativi più che altro a sistemi su *cloud*, e quindi all'ambito delle reti informatiche.

1.6 Gestione dei processi

Informalmente, il termine processo viene usato per indicare un programma in esecuzione sulla macchina.

- Rappresenta la *sequenza di eventi* generati dall'elaboratore durante l'esecuzione;
- Identifica la più piccola *unità di esecuzione* dentro un S/O multiprogrammato: questo consentirà l'esecuzione di *più* processi concorrenti;

Un processo va necessariamente *descritto*, cioè bisogna definire un **descrittore** che lo rappresenta. Del processo ci interessa:

- Il **codice** del programma che esegue;
- I **dati**;
- Il valore dell'**IP**;

- Lo stato dei **registri**;
- Lo **stack**.

Inoltre, ad un certo processo potranno essere associate delle risorse:

- **Memoria** utilizzata;
- **File** aperti;
- **Dispositivi** di I/O a cui ha accesso.

1.6.1 Processi in memoria

Il processo in memoria ha a disposizione il suo *spazio di indirizzamento virtuale*. Viene detto *virtuale* perché verrà allocato in una memoria centrale fisica, le cui locazioni potrebbero non corrispondere esattamente con la memoria offerta al processo (attraverso il meccanismo della *memoria virtuale*).

Partendo dal basso, le regioni di memoria fornite al processo nel suo spazio di indirizzamento saranno:

1. `text`: contiene il codice del processo;
2. `data`: contiene i dati statici del programa (sezione `data` e `bss`, che contiene lo spazio riservato a variabili statiche non allocate);
3. `heap`: l'heap del processo, dove vengono allocati oggetti in memoria dinamica;
4. `stack`: si sviluppa verso il basso, rappresenta la pila del processo in esecuzione.