1 Lezione del 02-12-25

Riprendiamo dopo tempo immemore l'argomento delle periferiche.

1.1 Processi esterni

Il comportamento del controllore di un dispositivo è assimilabile ad un processo, che chiamiamo **processo esterno**. Il controllo del comportamento del processo esterno esterno è influenzato dai bit del registro di controllo, che vengono aggiornati da un corrispondente *processo interno*.

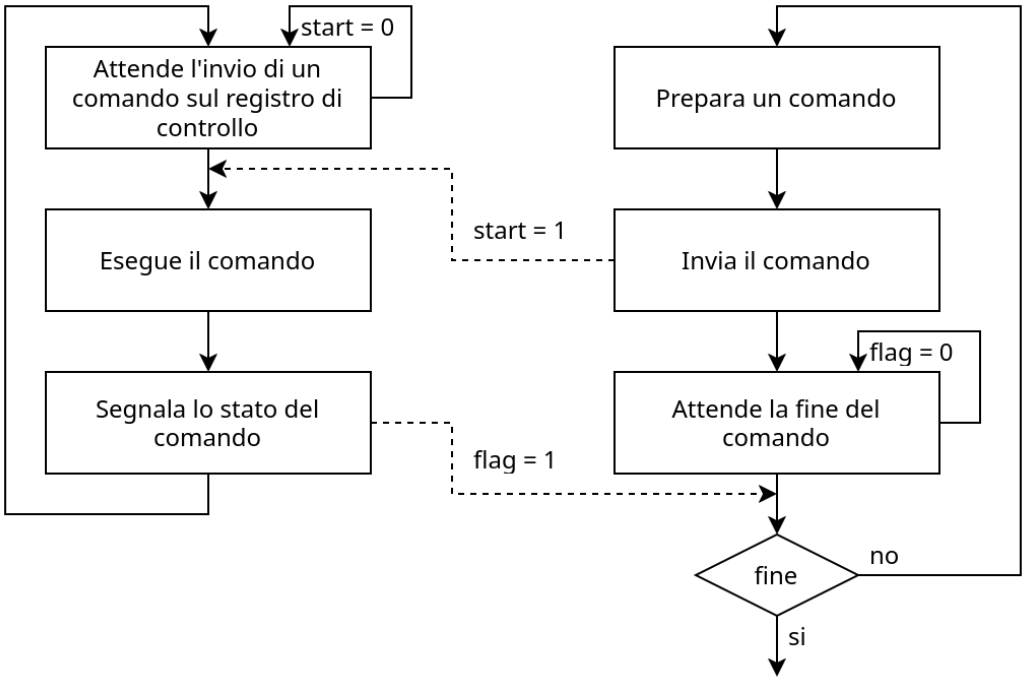
Ipotizziamo un processo esterno che si attiva quando viene alzato un dato bit di controllo, detto *bit di start*, e aggiorna un dato bit di stato, detto *bit di flag*. Il suo funzionamento potrebbe quindi essere il seguente:

1. Il PE attende l'invio di un comando sul registro di controllo;
2. Quando il bit di start transisce a 1, esegue il comando richiesto;
3. Successivamente all'esecuzione del comando aggiorna il bit di flag e torna allo stato 1.

Il corrispondente processo interno si comporta quindi come segue:

- Il PI prepara un comando;
- Invia il comando impostando il bit di start a 1;
- Attende la fine del comando aspettando che il bit di flag transisca a 1;
- Finisce o eventualmente ripete tornando allo stato 1.

Il comportamento appena descritto può essere riassunto dal seguente schema, dove si mettono in evidenza (con linee tratteggiate), i cambiamenti di stato che rappresentano una comunicazione fra PE e PI:

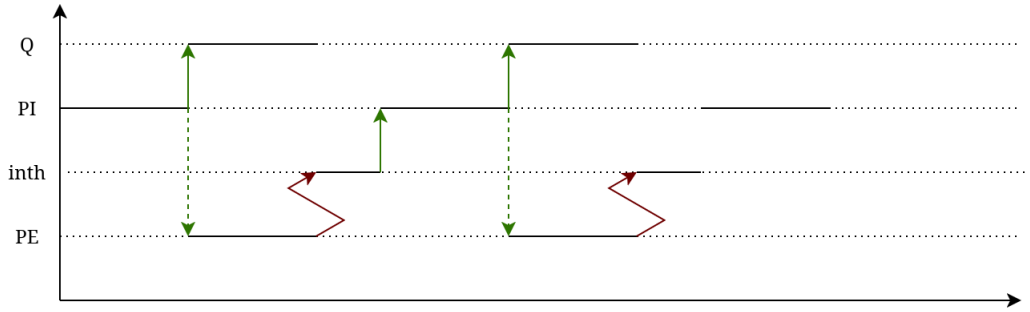


1.1.1 Gestione dei processi esterni

Assumiamo che i processi esterni modellizzano il comportamento di periferiche che eseguono il loro codice, o comunque portano avanti le loro operazioni, in parallelo al sistema. Sarà quindi vero che i PE eseguno solitamente in parallelo ad altri processi (anche solo processi utente).

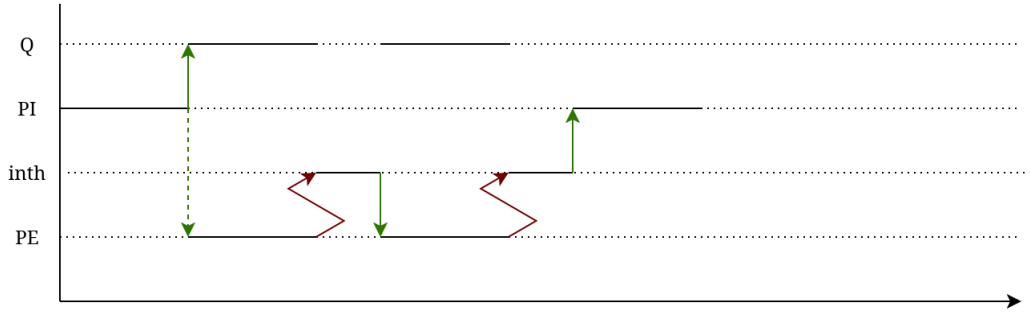
La transizione da processo utente a PI, quando un PE si mette in attesa (magari alzando un bit di flag), viene fatta sfruttando il meccanismo dell'*interruzione*. In particolare, prevediamo di associare un'interruzione alla modifica dei bit di flag da parte del dispositivo, e di predisporre nel sistema un handler per tale interruzione che rimetta in esecuzione il PI (che chiamiamo *inth*).

Possiamo visualizzare tale comportamento su un grafico:



Notiamo che l'esecuzione continua dell'inth, e della transizione dall'inth al PI, porta ad avere diversi cambi di contesto per ogni byte (o quanto di informazione su cui è tarato il buffer del dispositivo), cosa che chiaramente porta ad un overhead non indifferente. Possiamo pensare di migliorare la situazione includendo all'interno del inth stesso routine per la gestione di buffer da più di un byte (cioè fare in modo che sia l'inth a tenere conto dei byte letti / scritti finora, e a leggere dal / fornire al dispositivo il prossimo se necessario).

In tal caso il grafico ha il seguente aspetto:



1.1.2 Descrittori di dispositivo

All'interno del sistema, i dispositivi andranno raccontati da appositi **descrittori**. Questi descrittori vivranno all'interno dei driver (cioè assieme alla routine pensate per manipolarli, e per usarli per modificare lo stato dei dispositivi).

Routine di lettura scrittura fornite al programmatore sotto forma di API come `read()` e `write()`, nonché l'inth stesso, dovranno interagire con questo descrittore per portare avanti operazioni di trasferimento su e da dispositivo.

Vediamo quindi la struttura,a grandi linee, del descrittore di dispositivo:

- **Indirizzi dei registri**, relativi per ogni funzione alla solita tripla:
  - Indirizzo del registro di *controllo* (chiamiato `ctl`);
  - Indirizzo del registro di *stato* (chiamato `sts`);
  - Indirizzo del registro di *dati* (chiamato `dat`).

  La configurazione di tali registri viene fatta in fase di boostrap del sistema (ricordiamo ad esempio che il bus PCI prevede uno spazio di indirizzamento specifico, quello di *configurazione*, per il rilevamento dei dispositivi e la configurazione dei loro registri);
- Dati relativi alla **sincronizzazione** sul dispositivo (o meglio sul processo esterno), fra cui:
  - Un **semaforo** che indica se un dato è disponibile, detto appunto `dato_disponibile` (su cui chiameremo le classiche `signal()` e `wait()`);
  - Un **contatore** che indica il numero di dati da trasferire, che chiamiamo semplicemente `contatore`.
  - Vorremo poi un **puntatore al buffer** in memoria su o da cui stiamo facendo trasferimento, che chiamiamo semplicemente `puntatore`.
- Infine, terremo traccia di un qualche flag rappresentante l'**esito del trasferimento**, che chiamiamo `esito`.

1.1.3 Vista di un device driver

Vediamo quindi l'implementazione di un semplice device driver, assumendo un dispositivo in sola lettura (si implementa solo la `read()`).

Notiamo che ci aspettiamo di implementare il secondo approccio visto in 19.1.1, cioè quello dove è l'inth a occuparsi di tenere conto dei byte letti / scritti finora, e a leggere dal / fornire al dispositivo il prossimo se necessario).

Infine, per i valori che restituiscono errore (come `esito` del descrittore di dispositivo, o i registri di stato del dispositivo), assumiamo la costante `ERR_CODE` $\neq 0$ come codice di errore generico.

```
// descrittore del dispositivo
struct des {
  int ctl;
  int sts;
  int dat;

  sem dato_disponibile;
  int contatore;
  char* puntatore;

  int esito;
}

// primitiva read, legge @cont byte in @pbuf dal disp. @fd
int read(int fd, char* pbuf, int cont) {
  // ottiene il descrittore di dispositivo
  // (assumimamo un'array indicizzata)
  des* disp = &descr[fd];

  // imposta  il buffer
```

```
  disp->contatore = cont;
  disp->puntatore = pbuf;

  // attiva dispositivo
  out(disp->ctl, 1);

  // aspetta per il termine dell'operazione
  wait(disp->dato_disponibile);

  // raccoglie l'esito
  if(disp->esito == ERR_CODE) return -1;

  // restituisce il numero di byte ancora da leggere
  return(cont - disp->contatore);
}

// interrupt handler
void inth() {
  // assumiamo lettura
  // inoltre, assumiamo che des* disp sia noto

  // legge registro di stato
  char sts;
  in(disp->sts, sts);
  if(sts == ERR_CODE) {
    < gestione di errore (device-specific) >

    if(< errore non recuperabile >) {
      // se non si puo recuperare, chiudi la comunicazione qui
      disp->esito = ERR_CODE;
      signal(disp->dato_disponibile);
    }
  }

  // legge byte da dispositivo
  char b;
  in(disp->dat, b);

  // copia nel buffer utente e aggiorna descrittore
  *(disp->puntatore++) = b;
  disp->contatore--;

  // ha terminato?
  if(disp->contatore != 0) {
    // se no, chiede altro byte
    out(disp->ctl, 1);
  } else {
    // se si, segnala
    disp->esito = 0; // corretta terminazione
    signal(disp->dato_disponibile);
  }
}
```

1.2 Dispositivo timer

Vediamo nel dettaglio una periferiche reale, cioè il **timer**, che dovrà essere un *generatore di eventi programmabile*, su base temporale.

Il timer può essere molto utile, ad esempio solo per realizzare una primitiva di `sleep()` all'interno del sistema. Ci occupiamo adesso di definire il comportamento del controllore e realizzarne un driver.

1.2.1 Descrittore del timer

Vediamo quindi il descrittore del dispositivo timer.

Notiamo che questo dispositivo sarà capace di mantenere più timer *virtuali* contemporaneamente. Ciò sarà implementato gestendo più timer virtuali nel descrittore e aggiornandoli cumulativamente all'arrivo di eventi (interruzione) da parte del singolo timer fisico installato nel sistema.

- **Indirizzi dei registri**, relativi per ogni funzione alla solita tripla:
  - Indirizzo del registro di *controllo* (chiamiato `ctl`);
  - Indirizzo del registro di *stato* (chiamato `sts`);
  - Indirizzo del registro di *dati* (chiamato `dat`).
- Dati relativi alla **sincronizzazione** sul timer, fra cui:
  - Un'array di **semafori** che indicano se i timer hanno terminato, detti `fine_attesa[N]`;
  - Un'array di interi che rappresenta i ritardi di ogni timer, detta `ritardi[N]`.
- Infine, il classico flag di `esito`.

1.2.2 Driver del timer

Vediamo allora un semplice driver per il timer appena visto:

```
// descrittore del timer
struct des {
  int ctl;
  int sts;
  int dat;

  sem fine_attesa[N];
  int ritardi[N];

  int esito
}

des tim; // istanza globale

// primitiva delay, aspetta per @ritardo
void delay(int ritardo) {
  // usiamo il proc. corrente per indicizzare il timer
  int proc = < processo corrente >;

  // configura il timer
  tim.ritardo[proc] = ritardo;

  // aspetta il timer
  wait(tim.fine_attesa[proc]);
}

```

```
// interrupt handler
void inth() {
  for(int i = 0; i < N; i++) {
    if(descr.ritardo[i] != 0) {
      descr.ritardo[i]--;
      if(descr.ritardo[i] == 0)
        signal(descr.fine_attesa[i]);
    }
  }
}
```

1.2.3 Dispositivi a blocchi

Iniziamo a parlare dei *dispositivi a blocchi*, e in particolare dei **dischi**. Questi sono dispositivi che permettono di memorizzare, seppur in maniera più lenta rispetto alla RAM, vaste quantità di dati per l'archiviazione a lungo termine.

Storicamente (ed ancora oggi) i dischi venivano realizzati come dischi magnetici veri e propri (**HDD**, da *Hard Disk Drive*), mentre oggi si stano diffondendo sempre più dischi allo stato solido (**SSD**, da *Solid State Drive*). Una discussione più approfondita delle specifiche hardware si può trovare in `https://raw.githubusercontent.com/seggiani-luca/appunti-ce/638d3abf2e1d473632b575401582203c3b113c82/master/master.pdf`.

Ciò che basta sapere in questo contesto è che un disco è formato da più **tracce** disposte radialmente, ed ogni traccia è divisa in **settori**. Spesso, inoltre, più dischi sono sovrapposti fra di loro a formare **cilindri**.

1.2.4 Scheduling di dischi

I dischi sono dispositivi ad accesso *sequenziale*. Questo significa che è necessario un'algoritmo di **scheduling** degli accessi a disco, che minimizzi il movimento della testina di lettura e quindi i tempi medi diaccesso.

Ne vediamo alcuni:

- **FCFS** (*First Come First Served*): è l'algoritmo più semplice, dove gestiamo le richieste di accesso nell'ordine in cui arrivano. Lato software è estremamente semplice e veloce, ma lato hardware richiede potenzialmente il numero massimo di spostamenti della testina;
- **SSSF** (*Shortest Seek Time First*): sceglie l'accesso più vicino alla posizione corrente della testina. In questo caso riduciamo il numero di movimenti della testina, ma l'overhead non è più trascurabile:
  - Ogni volta che gestiamo una richiesta dobbiamo scorrere tutta la coda delle richieste per individuare quella più vicina;
  - C'è il rischio di starvation (se entrano spesso richieste con seek time minore).
- **SCAN** (o *algoritmo dell'ascensore*): è ispirato dal funzionamento degli ascensori. Si decide una direzione di andamento della testina, e si inizia a gestire le richieste seguendo tale direzione. Una volta arrivati ad un estremo (alla richiesta di indice più basso o più alto) si cambia direzione.