1 Lezione del 15-04-25

Avevamo quindi visto come si era introdotto un nuovo modulo, il modulo di I/O, che aveva il compito di gestire da gestire l'I/O da livello sistema, fornendo primitive di I/O e processi (driver) al modulo utente e sfruttando primitive, sia accessibili a livello utente che accessibili a livello sistema (quindi solo a lui) del sistema.

1.0.1 Cambi di contesto a processi esterni

Avremo quindi che in un dato momento sulla macchina sono in esecuzione i processi utente (P_1, P_2, ecc...) e i processi *esterni* di I/O. Assumiamo per adesso di avere un solo processo di I/O, chiamato appunto IO.

Nel momento in cui uno dei processi utente (diciamo P_1) chiama una primitiva di I/O, come ad esempio la `read_n()` già nominata, viene messo in attesa e si mettono in esecuzione altri processi. Di qui in poi, all'arrivo di un'interruzione esterna relativa all'operazione di I/O corrente, viene eseguito l'handler (definito in sistema) e quindi da questo messo in esecuzione il processo IO, che riempie ad ogni chiamata il buffer, e all'ultima chiamata rimette in lista pronti (attraverso i meccanismi implementati coi semafori visti in 19.1.5) il processo P_1.

1.0.2 Implementazione degli handler

Un'idea di base potrebbe essere quella di dedicare un *processo esterno* ad ogni tipo di interruzione esterna nell'IDT. Abbiamo visto come l'APIC supporta 24 piedini di interruzione esterna (`IREQ`), il cui tipo di interruzione nell'IDT può essere scelto via software modificando appositi registri. Vorremo quindi creare l'associazione:

$$\text{piedino di interruzione} \rightarrow \text{processo esterno}$$

cioè all'arrivo dell'interruzione esterna, il processore deve eseguire l'handler corretto come definito nella IDT, e questa deve mettere in esecuzione il processo corrente.

Servira quindi anche un *handler* per ogni tipo di interruzione esterna, corrispondente al suo processo esterno (definiremo quindi `handler_0`, `handler_1`, ..., `handler_23`).

La struttura di un handler generico sarà la seguente:

```
handler_i:
  # gestiamo l'interruzione
  call salva_stato

  # sospensdiamo il processo in esecuzione
  call inspronti

  # trova il processo esterno corrispondente
  mov a_p+i*8, %rax
  # e mettilo in esecuzione
  movq %rax, esecuzione

  call carica_stato
  iretq
```

`a_p` conterrà l'indirizzo della prima entrata di una lista di puntatori ai descrittori dei processi esterni:

`a_p`

0	Processo esterno 0
⋮	//
23	Processo esterno 23

Questa si definisce nel sistema come:

```
// MAX_IRQ e' il numero di linee di interruzione (24)
des_proc* a_p[apic::MAX_IRQ];
```

1.0.3 Implementazione dei processi esterni

I processi esterni avranno quindi struttura simile. Di base, avremo bisogno (come abbiamo visto) di descrittori associati a periferiche che tengono conto dei tipi specifici alle periferiche, e delle informazioni riguardanti l'ultima richiesta di I/O:

```
des_io array_des_io[MAX_DES_IO];
```

Avremo quindi che un processo esterno è implementato come:

```
void estern_i(natq id) {
  des_io *d = &array_des_io[id];
  // non servono controlli, id e' fidato

  // i processi esterni non terminano mai, quindi facciamo un ciclo
    infinito
  for(;;) {
    // qui si gestisce l'interruzione
    // ...

    // aspetta prossima interruzione
    wfi();
  }
}
```

Notiamo che chiaramente non possiamo chiamare, finita la gestione, `schedulatore()`, in quanto questo processo (non atomico) non deve avere accesso a strutture dati sensibili, e proprio per questo è stato messo nel modulo di I/O.

Si rende quindi disponibile un'altra di quelle primitive non accessibili all'utente, cioè la `wfi()` (*Wait For Interrupt*), che appunto sospende il processo mettendolo in attesa fino alla prossima interruzione di sua competenza, e mettendo in esecuzione un altro processo (cioè facendo la chiamata a `schedulatore()` che volevamo fare in primo luogo).

Possiamo vedere l'implementazione, in assembler, della `wfi()`:

```
a_wfi:
  call salva_stato

  # intanto rispondiamo all'APIC
  call apic_send_EOI

  # non dobbiamo fare niente col processo esterno,
  # e' gia' nella tabella a_p

  call schedulatore

  call carica_stato
  iretq
```

Vediamo quindi il dettaglio di poter usare, questa volta, le primitive semaforiche all'interno del processo esterno (in quanto il processo deve essere interrompibile), cioè dire:

```
extern "C" void estern_i(natq id)
{
  des_io *d = &array_des_io[id];
  // non servono controlli, id e' fidato

  for (;;) {
    // gestisci l'interruzione

    if (d->quanti == 0) {
      sem_signal(ce->sync); // qui puoi usare la sem_signal()
    }

    wfi(); // ecco la wfi() per il ritorno
  }
}
```

Infine, vediamo a scopo di esempio il processo esterno per una periferica nota, la *CE* del 19.1.5:

```
extern "C" void estern_ce(natq id)
{
  // trova il descrittore
  des_ce *ce = &array_ce[id];

  for (;;) {
    // gestisci l'interruzione
    ce->quanti--;
    if (ce->quanti == 0) {
      outputb(0, ce->iCTL);
    }
    char b = inputb(ce->iRBR);
    *ce->buf = b;
    ce->buf++;

    if (ce->quanti == 0) {
      sem_signal(ce->sync); // qui puoi usare la sem_signal()
    }

    wfi(); // ecco la wfi() per il ritorno
  }
}
```

Tastiera e hard disk dispongono di processi esterni simili, definiti rispettivamente in `estern_kbd()` e `estern_hd()`.

Vediamo un dettaglio: durante l'esecuzione di un processo esterno, non potrà accadere che questo viene interrotto da un interruzione esterna relativa allo stesso processo: questo perchè le interruzioni esterne relative a quel dispositivo sono effettivamente disattivate fino alla chiamata di `wfi()`, cioè quando viene chiamata la `apic_send_EOI()` (e non si può interrompere la `wfi()` perchè è una primitiva di sistema, quindi atomica).

Esiste invece la possibilità che un interruzione di livello più alto nella IDT vada ad interrompere il processo esterno (quindi si metta in esecuzione il processo esterno relativo a *quella* interruzione), ma questo è desiderabile in quanto corrisponde in maniera naturale alla priorità di gestione delle interruzioni esterne secondo l'APIC.

1.0.4 Creazione di processi esterni

Vediamo quindi nel dettaglio la `activate_pe()`, di uso concesso solo al modulo di I/O, che viene usata per creare i processi esterni. Questa sarà diversa dalla comune `activate_p()`,

in quanto dovrà anche impostare la tabella `a_p` per gli handler.

La funzione avrà quindi firma: `activate_pe(void (*f)(natq), natq id, natl prio, natl livello, natb irq)`, cioè si specifica:

- La **funzione** (`f`) che realizza il processo stesso;
- L'**indice** (`id`) di processo;
- La **priorità** (`prio`) del processo (che per i processi determinerà anche il tipo di interruzione nell'IDT);
- Il **livello** (`livello`) del processo, fin qui tutto normale;
- La **linea di interruzione** (`IRQ`) che tale processo esterno gestisce.

Come anticipato, un discorso importante va fatto sulla priorità di questi processi. Abbiamo infatti la priorità *da noi* definita, cioè `prio`, e la priorità nell'APIC (quindi nella IDT), data dal codice di interruzione assegnato ad una linea `IRQ`. Questa seconda priorità, che chiamiamo **tipo**, viene ricavata direttamente dalla proprietà `prio` secondo la seguente formula:

$$\texttt{prio} = \texttt{MAX\_PRIO\_UTENTE} + \texttt{tipo}$$

Da qui in poi, le parti di creazione vera e propria del processo potranno essere messe in comune con la `activate_p()`, e la `activate_pe()` avrà il solo compito aggiuntivo di predisporre un handler, con relativo setup dell'APIC e caricamento dell'handler nell'IDT.

1.0.5 Implementazione del modulo I/O

Vediamo quindi la struttura generaled del modulo I/O stesso. Questo dispone delle sue primitive, che definisce per l'utente e carica nella IDT attraverso la primitiva sistema `fill_gate()` (potrebbe farlo da sé, in quanto gira a livello utente, ma non lo fa per *"educazione"* nei confronti del modulo sistema, cioè si centralizza la gestione dell'IDT per evitare errori).

Le primitive definite gestiscono quindi essenzialmente tastiera, video ed hard disk, cioè si dispone delle primitive:

```
# inserimento delle primitive di I/O nell'IDT
fill_io_gate  IO_TIPO_HDR     a_readhd_n
fill_io_gate  IO_TIPO_HDW     a_writehd_n
fill_io_gate  IO_TIPO_DMAHDR  a_dmareadhd_n
fill_io_gate  IO_TIPO_DMAHDW  a_dmawritehd_n
fill_io_gate  IO_TIPO_RCON    a_readconsole
fill_io_gate  IO_TIPO_WCON    a_writeconsole
fill_io_gate  IO_TIPO_INIC    a_iniconsole
fill_io_gate  IO_TIPO_GMI     a_getiomeminfo
```

I processi esterni per le periferiche (tastiera e hard disk) sono quindi definiti come nel paragrafo precedente, cioè si ha:

```
// processo esterno associato alla tastiera
void estern_kbd(natq) {
  // ...
}

// processo esterno per le richieste di interruzione dell'hard disk
void estern_hd(natq) {
  // ...
}
```

1.0.6 Inizializzazione del modulo I/O

Possiamo quindi vedere un ulteriore dettaglio riguardo alla sequenza di boot, cioè quella che riguarda l'inizializzazione dell'I/O.

Avremo infatti nel `main()` del modulo sistema la seguente chiamata all'entry point del modulo I/O:

```
extern "C" void main(natq) {
  flog(LOG_INFO, "Creo il processo main I/O");
  main_io = crea_processo(mio_entry, int_cast<natq>(&io_init_done),
    MAX_EXT_PRIO, LIV_SISTEMA);
  if (main_io == nullptr) {
    flog(LOG_ERR, "impossibile creare il processo main I/O");
    goto error;
  }
  processi++;
  flog(LOG_INFO, "Attendo inizializzazione modulo I/O...");

  // cediamo il controllo al modulo I/O e aspettiamo che setti
  // la variable io_init_done
  cedi_controllo(main_io);

  // sostanzialmente una sorta di sem_wait()
  while (!io_init_done)
    halt(); // abilita temporaneamente le interruzioni esterne
}
```

dove la funzione `cedi_controllo()` ha il compito di salvare lo stato corrente del kernel in pila in modo che vi si possa ritornare dopo l'esecuzione del *processo di inizializzazione* (è proprio un processo) di un altro modulo.

A questo punto il modulo di I/O prenderà il controllo dal suo entry point:

```
extern "C" void main_io(natq p)
{
  // questo e' il flag che sta guardando il main() di sistema
  int *p_io_init_done = ptr_cast<int>(p);

  // riempi la IDT delle primitive di I/O
  fill_io_gates();

  // inizializza l'heap
  ioheap_mutex = sem_ini(1);
  if (ioheap_mutex == 0xFFFFFFFF) {
    flog(LOG_ERR, "impossible creare semaforo ioheap_mutex");
    abort_p();
  }
  char* end_ = allinea_ptr(_end, DIM_PAGINA);
  heap_init(end_, DIM_IO_HEAP);
  flog(LOG_INFO, "Heap del modulo I/O: %llxB [%p, %p)", DIM_IO_HEAP,
      end_, end_ + DIM_IO_HEAP);

  // inizializza la console (tastiera + video)
  flog(LOG_INFO, "Inizializzo la console (kbd + vid)");
  if (!console_init()) {
    flog(LOG_ERR, "inizializzazione console fallita");
    abort_p();
  }

  // inizializza l'hard disk
  flog(LOG_INFO, "Inizializzo la gestione dell'hard disk");
  if (!hd_init()) {
```

```
    flog(LOG_ERR, "inizializzazione hard disk fallita");
    abort_p();
  }

  // avverti main che hai finito
  *p_io_init_done = 1;

  // termina
  terminate_p();
}
```

Abbiamo quindi una serie di funzioni (`heap_init()`, `console_init()`, `hd_init()`, ...) che hanno il compito di inizializzare ogni dispositivo. Queste, oltre ad inizializzare tutte le informazioni relative al descrittore di dispositivo, hanno il compito di creare il processo esterno di gestione di tale dispositivo.

Ad esempio, riguardo alle periferiche di tipo *CE* descritte in 19.1.5 (e ammesso quindi di avere più periferiche dello stesso tipo nel bus), potremmo definire la funzione di inizializzazione:

```
// trova le periferiche CE installate e crea i rispettivi processi esterni
bool ce_init()
{
  // scansiona il bus PCI per le periferiche di tipo CE
  for (natb bus = 0, dev = 0, fun = 0;
       pci::find_dev(bus, dev, fun, 0xedce, 0x1234);
       pci::next(bus, dev, fun))
  {
    if (next_ce >= MAX_CE) {
      flog(LOG_WARN, "troppi dispositivi ce");
      break;
    }

    // crea il descrittore
    des_ce *ce = &array_ce[next_ce];

    // configuralo
    ioaddr base = pci::read_confl(bus, dev, fun, 0x10);
    base &= ~0x1;
    ce->iCTL = base;
    ce->iSTS = base + 4;
    ce->iRBR = base + 8;

    if ( (ce->sync = sem_ini(0)) == 0xFFFFFFFF ) {
      flog(LOG_WARN, "ce%d: impossibile allocare semaforo sync", next_ce);
      break;
    }
    if ( (ce->mutex = sem_ini(1)) == 0xFFFFFFFF ) {
      flog(LOG_WARN, "ce%d: impossibile allocare semaforo mutex", next_ce)
    ;
      break;
    }

    // trova la linea di interruzione del dispositivo
    natb irq = pci::read_confb(bus, dev, fun, 0x3c);

    // attiva il procsso esterno
    if (activate_pe(estern_ce, next_ce, MIN_EXT_PRIO + 0x80, LIV_SISTEMA,
    irq) == 0xFFFFFFFF) {
      flog(LOG_WARN, "ce%d: impossibile attivare processo esterno",
    next_ce);
```

```
39        break;
40      }
41      // log...
42
43      next_ce++;
44    }
45
46    // restituisci true se non sei riuscito ad inizializzare
47    return next_ce != 0;
48  }
```

Implementazioni simili si trovano per la `console_init()`, che chiama a sua volta `vid_init()` e `kbd_init()`, e la `hd_init()`.

Come ultimo dettaglio, ricordiamo che il timer è comunque gestito da sistema, utilizzando i driver come definiti in `19.1.5` (in particolare, `driver_td()`).