1 Lezione del 11-04-25

Riprendiamo il discorso dell'I/O nel kernel.

1.1 Primitive di I/O

L'ipotesi generale è che una primitiva di I/O vuole trasferire dati da e alla memoria o da e allo spazio di I/O. La forma generica della primitive che vorremo fornire all'utente sarà quindi del tipo:

- **Lettura:**

```
read_n(id, char* buf, natq quanti);
```

  dove prendiamo la periferica `id`, e la usiamo per leggere `quanti` byte e inserirli nel buffer (in memoria condivisa) `buf`;

- **Scrittura:**

```
write_n(id, const char* buf, natq quanti);
```

  dove prendiamo la periferica `id`, e ci scriviamo `quanti` byte dal buffer (in memoria condivisa) `buf`.

Avremo quindi che un certo processo P_1, a qualche punto della sua esecuzione, chiama una primitiva di I/O. A questo punto il controllo passa al kernel, che si occupa quindi di gestire l'operazione, sfruttando prima di tutto le istruzioni privilegiate a cui ha accesso `IN`, `OUT`, ecc... e le possibilità di scheduling di cui dispone per eseguire, mentre attende per la sincronia col dispositivo, altri processi (mettendo quindi quello che aveva chiamato la primitiva in attesa). Quando P_1 viene rimesso in esecuzione, non vede niente di diverso nel suo contesto privato, se non il fatto che il buffer contiene adesso l'input desiderato.

Le primitive di sistema che gestiscono un certo dispositivo prendono il nome, abbiamo detto, di **driver**.

1. Per realizzare il meccanismo di **sincronizzazione** sfruttiamo *semaforo*: vogliamo che il la primitiva di I/O chiami la `sem_wait()` al momento dell'inizio dell'operazione di I/O, e che la `sem_signal()` venga chiamata sullo stesso semaforo alla fine dell'operazione per segnalare che l'operazione è finita (dal driver stesso).

2. Per realizzare invece la **mutua esclusione** si sfrutta il procedimento inverso: si parte con un semaforo inizializzato a `sem_wait()`, e successivamente ogni processo che inizia un'operazione di I/O ne prende un *"gettone"*, restituendolo a termine operazione, così che nessun processo possa iniziare una operazione di I/O contemporaneamente a un altro.

1.1.1 Primitive nel kernel

Avevamo visto che il problema era usare le `sem_wait()` e le `sem_signal()` lato kernel, in quanto queste effettivamente interrompono routine sistema e ne violano l'atomicità.

Vediamo però che, se ci limitiamo a non manipolare le strutture dati sistema nelle primitive `read_n()` e `write_n()`, non incappiamo nei rischi per cui avevamo introdotto l'atomicità delle routine sistema in primo luogo (le uniche a manipolare la lista processi saranno le `read_n()` e `write_n()`).

L'unico problema resterà il discorso del contesto, che verrebbe salvato così 2 volte. Rimuoviamo allora le `salva_stato` e `carica_stato`.

Trasformiamo quindi le primitive di I/O in primitive effettivamente *non atomiche*, che eseguono nello stesso contesto (a livello di registri) del processo chiamante.

1.1.2 Driver

Vediamo come si svolge la situazione lato driver. Potremmo pensare iniziatutto di sfruttare un certo descrittore di *dispositivo*, `des_io`, realizzato ad esempio come:

```
des_io {
  char* buf;
  natq quanti;

  // semafori
  natl mutex;
  natl sync;
};
```

cioè contenente tutte le informazioni necessarie al driver per soddisfare la richiesta del processo che ha richiesto l'I/O, scrivendo i dati ottenuti nel buffer `buf` giusto all'indice `quanti` (che incrementerà o decrementerà da solo) giusto.

- `buf` e `quanti` vengono forniti alla primitiva di I/O e cambiano per ogni richiesta da parte dei processi.
- `mutex` e `sync` sono gli indici dei semafori e vengono inizializzati una sola volta per ogni dispositivo. In particolare, stando a quanto abbiamo detto nello scorso paragrafo, vorremo:

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

Notiamo che in ogni caso il buffer rappresenta i dati lato utente, cioè come vedremo contenuti nella memoria condivisa dei processi, e nulla riguardo al dispositivo vero e proprio: i suoi dati verranno ottenuti, a controllo d'interruzione, dalla primitiva stessa attraverso i suoi procedimenti specifici. In particolare, il buffer della `read_n` sarà quello dove il driver dovrà scrivere cosa legge così che il processo lo veda, mentre il buffer (costante) della `write_n` sarà quello dove il driver dovrà leggere per restituire poi al dispositivo.

Chiamando `proc` la primitiva che si occupa di realizzare la richiesta I/O (nell'esempio in 19.1, potrebbe essre sia `read_n` che `write_n`), e `driver` la primitiva del driver, si ha che i semafori di un certo descrittore `ce` si evolvono quindi come segue (in relazione alla lista di 19.1):

1. `proc` aspetta che arrivi `ce->mutex` (2);
2. `proc` prende `ce->sync` e viene messo in attesa (1);

3. `driver` restituisce `ce->sync` (1);

4. `proc` restituisce `ce->mutex` (2)

e che quindi `proc` e `driver` sono strutturati pressappoco come segue, prima `proc`:

```
extern "C" void proc(natl id, ...) {
  // ottieni ce
  sem_wait(ce->mutex); // (2)
  sem_wait(ce->sync); // (1)

  // qui saremo messi in attesa, e' compito di driver restituire ce->sync

  sem_signal(ce->mutex); // (2)
}
```

e poi `driver`:

```
extern "C" void driver() {
  // ottieni ce

  // intanto restituirai un byte
  ce->quanti--;

  if(ce-quanti == 0) {
    // appunto, restituiamo
    c_sem_signal(ce->sync); (1)
  }

  // qui gestiamo l'interruzione esterna del dispositivo
  // e restituiamo effettivamente il byte
}
```

Nei prossimi paragrafi andremo via via a definire i dettagli tecnici e raffinare l'implementazione.

1.1.3 Driver e primitive

La domanda è se il driver può chiamare le primitive semaforo che gestiscono i semafori di indice `mutex` e `sync`. In particolare, avevamo detto che il driver avrà il compito di chiamare la `sem_signal(sync)` per segnalare al processo che l'operazione di I/O è finita. Abbiamo però il problema della `salva_stato`, che andrà a sovrascrivere, quando chiamata da una routine sistema (e quindi anche da un driver) il contesto del processo in `esecuzione` (che potrebbe essere arbitrario) con i valori correnti della routine sistema, facendo evidentemente danni.

Potremmo allora pensare di usare direttamente la sua implementazione, cioè chiamare la `c_sem_signal()`. Il problema sarà che la `c_sem_signal()` usa la `sem_valido()` per controllare la validità del semaforo cercato, usando la `liv_chiamante()`, che non sarebbe significativa se chiamata senza passare da un interruzione (sfrutta il CS salvato presumibilmente in pila dalla `INT`). In particolare, vediamo che si ha:

```
extern "C" void c_sem_signal(natl sem)
{
  // una primitiva non deve mai fidarsi dei parametri
  if (!sem_valido(sem)) {
    flog(LOG_WARN, "semaforo errato: %u", sem);
    c_abort_p();
    return;
  }
```

```
9
10    // corpo di esecuzione effettiva, alla 11.2.1
11  }
```

percorrendo il backtrace al contrario:

```
1  bool sem_valido(natl sem)
2  {
3    // dal momento che i semafori non vengono mai deallocati,
4    // un semaforo e' valido se e solo se il suo indice e' inferiore
5    // al numero dei semafori allocati
6
7    int liv = liv_chiamante(); // qui vogliamo distinguere il contesto
8    return sem < sem_allocati_utente ||
9      (liv == LIV_SISTEMA && sem - MAX_SEM < sem_allocati_sistema);
10 }
```

e infine:

```
1  int liv_chiamante()
2  {
3    // (ci aspettiamo che) salva_stato ha salvato il puntatore
4    // alla pila sistema subito dopo l'invocazione della INT
5    natq* pila = ptr_cast<natq>(esecuzione->contesto[I_RSP]);
6
7    // -> peccato che nessuno ha chiamato la salva_stato!
8    //    siamo passati da c_sem_signal() e non a_sem_signal()
9
10   // la seconda parola dalla cima della pila contiene il livello
11   // di privilegio che aveva il processore prima della INT
12   return pila[1] == SEL_CODICE_SISTEMA ? LIV_SISTEMA : LIV_UTENTE;
13 }
```

Saremo quindi costretti a replicare in qualche modo la `c_sem_signal()` nel codice del driver, cioè dire:

```
1  extern "C" void c_driver() {
2    // ottieni ce
3
4    // intanto restituirai un byte
5    ce->quanti--;
6
7    if(ce-quanti == 0) {
8      // qui (!) facciamo la sem_signal()
9      des_sem *s = &array_dess[ce->sync];
10
11     s->counter = 0;
12     des_proc* lavoro = rimozione_lista(s->pointer);
13     inspronti();  // preemption
14     inserimento_lista(pronti, lavoro);
15     schedulatore(); // preemption
16   }
17
18   // qui gestiamo l'interruzione esterna del dispositivo
19   // e restituiamo effettivamente il byte
20 }
```

In questo modo, il codice del driver dovrà essere effettivamente atomico (`c_sem_signal()` manipola le liste di processi, che sono strutture dati sensibili). Questo potrebbe essere problematico in quanto ci impedisce di gestire interruzioni innestate a priorità più alta. Decidiamo di continuare con questa limitazione.

1.1.4 Cavalli di Troia

Altre questioni di sicurezza potrebbero riguardare i cosiddetti **cavalli di Troia**: un utente potrebbe sfruttare la `write_n()` per scrivere in locazioni di memoria arbitrarie, dove lui da solo non avrebbe potuto scrivere. Si rende quindi necessario controllare gli indirizzi passati alle primitive di I/O lato software.

Vediamo i problemi che vogliamo controllare:

- Gli indirizzi potrebbero fare wraparound, costringendo il driver a saltare da M_2 a M_1, e quindi sovrascrivendo più memoria di quanta probabilmente si voleva impattare;
- Regioni di indirizzi virtuali (di per sé contigue) che potrebbero avere flag di scrivibilità non contigui (cioè si potrebbero incontrare flag di traduzioni modalità sistema o sola lettura);
- Indirizzi virtuali non tradotti, che quindi causerebbero page fault (e visto che abbiamo detto il driver deve essere atomico, non possiamo permetterci nessuna eccezione);
- Non vogliamo accedere alla memoria privata del processo richiedente l'I/O, perché l'albero di traduzione caricato al momento dell'esecuzione del driver non sarà sicuramente il suo (è stato messo in attesa dalla primitiva), ma quello di un altro processo (quello in `esecuzione`) con la sua memoria privata (che sicuramente non vogliamo toccare).

L'unica regione valida resta quindi quella che avevamo già detto, cioè la **memoria condivisa**. Notiamo che questa era comunque una necessità, in quanto vogliamo che il processo passi per forza un buffer corrispondente ad un indirizzo in memoria condivisa, appunto perchè chiedere alla primitiva driver di scrivere nel suo spazio privato sarebbe complesso (a meno di non passare un indirizzo fisico, e quindi passare dalla finestra FM, cosà però abbastanza complicata dal punto di vista dei controlli).

Per tutti questi controlli, quindi, siamo quindi costretti a controllare tutte le traduzioni.

Scriveremo quindi un'apposita funzione di controllo del buffer del tipo:

```
// restituisce true (nell'I_RAX del contesto) se la regione va bene,
// false altrimenti
extern "C" bool c_access(vaddr begin, natq dim, bool writeable, bool
    shared = true)
{
  esecuzione->contesto[I_RAX] = false;

  // l'intervallo e' valido?
  if (!tab_iter::valid_interval(begin, dim))
    return false;

  // siamo nella regione di memoria utente condivisa?
  if (shared && (!in_utn_c(begin) || (dim > 0 && !in_utn_c(begin + dim -
    1))))
    return false;

  // usiamo un tab_iter per percorrere tutto il sottoalbero relativo
  // alla traduzione degli indirizzi nell'intervallo [begin, begin+dim).
  for (tab_iter it(esecuzione->cr3, begin, dim); it; it.next()) {
    tab_entry e = it.get_e();
```

```
19
20      // interrompiamo il ciclo non appena troviamo qualcosa che non va
21      if (!(e & BIT_P) || !(e & BIT_US) || (writeable && !(e & BIT_RW)))
22        return false;
23    }
24    esecuzione->contesto[I_RAX] = true;
25    return true;
26  }
```

che fa prima i controlli detti su intervallo e regione processi, e poi percorre l'intero albero di traduzione, controllando che ogni traduzione sia scrivibile.

1.1.5 Implementazione di un driver

Vediamo quindi l'implementazione di un semplice driver per un dispositivo fasullo cioè il dispositivo *CE*, dotato di soli 3 registri (e relative porte):

`0x?? + 0`	**CTL**, *Control Register*
`0x?? + 4`	**STR**, *Status Register*
`0x?? + 8`	**RBR**, *Receive Buffer Register*

Il `0x??` è dato dal fatto che il dispositivo è montato sul bus PCI, ergo per disporre i suoi registri nello spazio di I/O bisogna regolarne il **BAR**.

In ogni caso, il funzionamento del dispositivo è banale: si limita a controllare periodicamente CTL, e quando vi trova 1, stampare un carattere alfanumerico.

Disponiamo per tale dispositivo il descrittore:

```
1  struct des_ce {
2    // i registri, variabili perche in PCI
3    ioaddr iCTL, iSTS, iRBR;
4
5    // gia' visto
6    char *buf;
7    natl quanti;
8    natl sync;
9    natl mutex;
10 };
```

Avremo quindi una funzione `ce_init()`, che viene lanciata all'avvio del kernel, e che si occupa di impostare il BAR e i descrittori dispositivo.

A questo punto, forniremo all'utente la primitiva `c_cereadn()`, per la lettura:

```
1  // questa non pu  toccare i descrittori di processo, praticamente e' una
       chiamata di funzione,
2  // quindi e' interrompibile (non atomica)
3  extern "C" void c_cereadn(natl id, char *buf, natl quanti)
4  {
5    if (id == id_ce) {
6      flog(LOG_WARN, "ce non riconosciuto: %d", id);
7      abort_p(); // e' quella normale, nessuno ha ancora chiamato
       salva_stato
8    }
9
10   if (!quanti)
11     return;
12
13   if (!c_access(reinterpret_cast<vaddr>(buf), quanti, true, false)) {
14     flog(LOG_WARN, "buf non valido\n");
15     abort_p();
```

```
  }

  des_ce *ce = ce;
  sem_wait(ce->mutex);
  ce->buf = buf;
  ce->quanti = quanti;
  outputb(1, ce->iCTL); // qui attivi
  sem_wait(ce->sync);
  sem_signal(ce->mutex);
}
```

mentre imposteremo nella IDT una funzione assembly che chiama il seguente gestore per l'interruzione esterna associata a *CE*:

```
// questa e' atomica, e reimplementa in qualche modo sem_signal()
extern "C" void c_driver_ce(int id)
{
  // qui otteniamo il descrittore di dispositivo
  des_ce *ce = ce;
  ce->quanti--;

  // qui si termina se si e' finito
  if (ce->quanti == 0) {
    // qui disattiviamo le interruzioni
    outputb(0, ce->iCTL);

    // qui (!) facciamo la sem_signal()
    des_sem *s = &array_dess[ce->sync];

    s->counter = 0;
    des_proc* lavoro = rimozione_lista(s->pointer);
    inspronti();  // preemption
    inserimento_lista(pronti, lavoro);
    schedulatore(); // preemption
  }

  // qui gestiamo effettivamente il dispositivo

  // questa va fatta dopo che disattivi le interruzioni,
  // altrimenti potresti avere un'altra interruzione sotto (nell'IRR)
  char b = inputb(ce->iRBR);
  *ce->buf = b;
  ce->buf++;
}
```

dove notiamo il dettaglio che la gestione del dispositivo si fa dopo aver disattivato, eventualmente, il dispositivo impostando CTL a 0. In caso contrario, si potrebbe ottenere una nuova interruzione, messa in IRR dall'APIC, che andrebbe erroneamente a richiamare il driver una volta di troppo rispetto a quelle previste.

Lato assembler, il driver verrà impostato come segue:

```
% setup IDT
carica_gate INTR_TIPO_CE  a_driver_ce LIV_SISTEMA
```

dove la `a_driver_ce` è:

```
a_driver_ce:
  call salva_stato
  movq $0, %rdi
  call c_driver_ce
  call apic_send_EOI
  call carica_stato
```

```
  iretq
```

La `c_cereadn()` avrà invece la controparte assembler:

```
  .extern c_cereadn
a_cereadn:
  % qui nessuno chiama salva_stato
  call c_cereadn
  % qui nessuno chiama carica_stato
  iretq
```

Dove notiamo ancora meglio che praticamente si ha una semplice chiamata di funzione che ritorna con la `IRETQ` invece che con la `RET`.