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 leggiamo `quanti` byte nel buffer `buf`;

- **Scrittura:**

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

  dove prendiamo la periferica `id`, e scriviamo `quanti` byte dal buffer `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**.

Per realizzare il meccanismo di **sincronizzazione** sfruttiamo *semaforo*: vogliamo che il processo 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.

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'altro 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.

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.

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 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`). Saremo quindi costretti a replicare in qualche modo la `c_sem_signal()` nel codice del driver.

In questo modo, il codice del driver sarà effettivamente atomico. Questo potrebbe essere problematico in quanto ci impedisce di gestire interruzioni innestate a priorità più alta.

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 che, a causa della grande quantità di problemi che questo controllo solleva (ad esempio, indirizzi che potrebbero fare wraparound, indirizzi virtuali non cointigui che potrebbero passare per traduzioni livello sistema, o addirittura indirizzi non tradotti che causerebbero page fault e quindi eccezioni che non possiamo gestire in routine sistema, ecc...), siamo costretti a controllare tutte le traduzioni.

Inoltre, vogliamo controllare di non accedere allo spazio di memoria privata del processo, in quanto quando il driver sarà in esecuzione questo conterrà la memoria pri-

vata del processo attualmente in esecuzione, e non del processo che ha effettivamente chiamato la primitiva.

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).

Scriveremo quindi una funzione di controllo del buffer del genere:

```
extern "C" bool c_access(vaddr begin, natq dim, bool writeable, bool
    shared = true)
{
  esecuzione->contesto[I_RAX] = false;

  if (!tab_iter::valid_interval(begin, dim))
    return false;

  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();

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

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.