1 Lezione del 25-03-25

1.1 Attesa

Esiste un altra primitiva, la `delay`, che viene usata per sospendere un processo per un certo istante temporale.

Il kernel sfrutta di per sé il timer 1 per generare interruzioni periodiche, che lo assistano anche solamente a tenere traccia del tempo trascorso durante l'esecuzione. A questo punto la `delay(natl n)` si limita ad aspettare `n` clicli del timer. Un'implementazione naive del timer è quindi quella di una lista di strutture, che rappresentano **richieste**, che tengono conto del loro conteggio attuale e del processo che le ha invocate. Un processo crea una richiesta sfruttando la primitiva `delay`, che risulta nella creazione di una richiesta e dello spostamento del processo nella lista bloccati. Il kernel dovrà quindi limitarsi ad aggiornare ad ogni ciclo di timer le richieste, decrementandole, e quindi ad riportare il processo in esecuzione una volta che il conteggio raggiunge 0.

Un modo più efficiente di fare la stessa cosa è quello di memorizzare non il conteggio di ogni richiesta, ma il conteggio *successivo* alla richiesta precedente nella lista d'attesa. Questo, chiaramente, implicherà un possibile riordinamento della lista in fase di inserzione (chi arriva prima sta in testa). In questo caso basterà decrementare solo il primo elemento della lista, e in occasione di raggiungimento di 0 eliminare quel processo e i successivi con conteggio aggiuntivo pari a 0.

1.1.1 Implementazione delle primitive d'attesa

Vediamo quindi l'implementazione vera e propria della primitiva `delay()`, secondo quanto detto finora. Questa si basa prima di tutto sulla definizione di una struttura `richiesta`, e dal mantenimento di una lista di tali richieste:

```
struct richiesta {
  // tempo di attesa aggiuntivo rispetto alla richiesta precedente
  natl d_attesa;
  // puntatore alla richiesta successiva
  richiesta* p_rich;
  // descrittore del processo che ha effettuato la richiesta
  des_proc* pp;
};

// Coda dei processi sospesi
richiesta* sospesi;
```

A questo punto serviranno due primitive, la `delay()` vera e propria, e la `driver_td()`, che si occupa effettivamente di avanzare temporalmente le richieste quando ha luogo un impulso di timer.

- `delay()`:

```
// primitiva di delay
extern "C" void c_delay(natl n)
{
  // caso particolare: se n e' 0 non facciamo niente
  if (!n)
    return;

  richiesta* p = new richiesta;
  p->d_attesa = n;
  p->pp = esecuzione;
```

```
11
12    inserimento_lista_attesa(p);
13    schedulatore();
14  }
```

- `driver_td()`:

```
1  // driver del timer
2  extern "C" void c_driver_td(void)
3  {
4    inspronti();
5
6    if (sospesi != nullptr) {
7      sospesi->d_attesa--;
8    }
9
10   while (sospesi != nullptr && sospesi->d_attesa == 0) {
11     inserimento_lista(pronti, sospesi->pp);
12     richiesta* p = sospesi;
13     sospesi = sospesi->p_rich;
14     delete p;
15   }
16
17   schedulatore();
18 }
```

1.2 Memoria dinamica

Vediamo alla gestione della memoria dinamica, in particolare alla parola chiave `new` fornita dal linguaggio. Per noi le `new` non si tradurranno in altro che chiamate di funzione, che cercano una zona di memoria libera dove allocare il dato desiderato. Di contro, la `delete` si occuperà di deallocare lo stesso dato.

Una domanda che potremmo porci è dove si trova questa memoria. Per quanto riguarda il **sistema**, una porzione dedicata viene inizializzta all'avvio e resta tale durante l'esecuzione dello stesso. Le allocazioni e deallocazioni si fanno quindi con le `alloc()` e `dealloc()` (che ridefiniscono gli operatori corrispondenti, `new` e `delete`), definite all'interno di `libce.h`.

Per quanto riguarda l'**utente**, invece, si dedica un altra porzione di memoria, condivisa fra i processi. Questa condivisione implica che più processi non possono fornire le loro versioni della funzione `new` e `delete`, in quanto se queste venissero interrotte (le funzioni utente non sono mai atomiche), lascerebbero la memoria dinamica in uno stato inconsistente per altri processi intenzionati a modificarla.

Si usa quindi un semaforo che tiene conto di chi sta scrivendo in memoria. In particolare, vediamo le:

- `new`: implementata per l'utente come:

```
1  // alloca un oggetto nello heap utente
2  void* operator new(size_t s)
3  {
4    void* p;
5
6    sem_wait(userheap_mutex);
7    p = alloc(s);
8    sem_signal(userheap_mutex);
9
```

```
  return p;
}
```

- `delete`: implementata per l'utente come:

```
// dealloca un oggetto dallo heap utente
void operator delete(void* p)
{
  sem_wait(userheap_mutex);
  dealloc(p);
  sem_signal(userheap_mutex);
}
```

Queste vengono semplicemente fornite in un apposita libreria (`lib.h`) al programma utente, che può servirsene per scrivere, assieme agli altri processi, nell'heap utente.

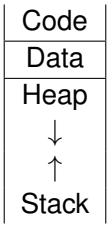
1.3 Memoria virtuale

Veniamo quindi all'ultimo argomento chiave del corso. Abbiamo detto che la memoria di sistema è divisa fra sistema e utente. In ogni momento ci aspettiamo che la memoria utente occupata da due porzioni: una **parte pubblica**, che rappresenta l'heap condiviso fra processi, e la **parte privata**, che rappresenta la memoria relativa ad un *singolo* processo, quello attualmente in esecuzione. La memoria privata degli altri processi è stata quindi intesa finora come memorizzata separatamente, magari nel disco rigido, con conseguente scaricamento del processo corrente e caricamento del successivo in memoria in fase di cambio di contesto fra processi.

Per i nostri scopi, possiamo assumere anche l'heap come parte della memoria privata. Il problema principale sarà infatti quello di poter memorizzare le immagini della memoria di *più* processi contemporaneamente. Infatti, storicamente, per *sistema multiprogrammato* si intendeva proprio il sistema in grado di mantenere più processi *in memoria* (il sistema visto finora sarebbe stato detto *multiprocesso*).

Decidiamo quindi di dividere la porzione di memoria utente in più sezioni, associate ad ogni processo. Potremmo intanto chiederci qual'è la memoria da dedicare ad ogni processo. La porzione dati e il codice di un programma sono infatti fissi in dimensioni, mentre pila e heap non lo sono. Storicamente, la memoria richiesta veniva specificata dal programmatore in fase di scrittura del programma. Quali metodologie si usino oggi non ci è immediatamente di interesse.

Si crea quindi per ogni processo una struttura di questo tipo:



Dove lo stack e l'heap si espandono in una sola regione, da estremità opposte.