1 Lezione del 23-10-25

1.0.1 Atomicità delle primitive semaforiche

Ora che il problema della mutua esclusione è effettivamente risolto, interroghiamoci su come rendere effettivamente atomiche le `wait()` e `signal()`.

- In ambiente monoprocessore, basterà rendere tali funzioni *primitive* di sistema, e quindi eseguirle con le interruzioni disattivate, per assicurarne l'atomicità.
- In ambienti multiprocessore, si potrebbero invece avere collisioni fra le `wait()` e `signal()` chiamate da processi concorrenti in esecuzione parallela.

  Approfondiamo il problema: posto un semaforo di mutex, ad esempio, questo si troverà in memoria condivisa. Più di un processore potrà accedere alla memoria condivisa attraverso il bus. Se ci limitiamo a rendere "atomiche" le primitive `wait()` e `signal()` disattivando le interruzioni su *un* processore, non risolviamo il caso dove più processori vogliono accedere al semaforo contemporaneamente.

  Possiamo risolvere il problema usando la tecnica introdotte in 11.1.2, cioè dotandoci di un meccanismo hardware di *bloccaggio* del bus, fornito dall'istruzione `TSL`. Potremo infatti usare le primitive `lock()` e `unlock()` per bloccarci sulla risorsa semaforo, cioè dicendo:

```
prologo:
  {
    lock(mutex);
    wait(mutex);
    unlock(mutex);
  }
  // sezione critica
epilogo:
  {
    lock(mutex);
    signal(mutex);
    unlock(mutex);
  }
```

  Notiamo che questa soluzione non è propriamente corretta: infatti il processo non chiamerà la `unlock()` nel prologo finché non verrà svegliato dalla `wait()`: questo significa che potrebbe tenersi il lock sul semaforo, impedendo ad altri processi di segnalare sul semaforo e liberarlo!

  La soluzione corretta sarà quindi quella di ridefinire le primitive semaforiche come segue:

  - `wait(s)`:

```
void wait(s) {
  lock();
  if(s.value == 0) {
    // metti il chiamante in attesa
    insert(s.queue, /* chiamante */);
  } else {
    s.value--;
  }
  unlock();
}
```

– `signal(s)`:

```
void signal(s) {
  lock();
  if(!isEmpty(s.queue)) {
    primo = extract(s.queue); // e' una coda fifo
    // inserici primo in coda pronti
  } else {
    s.value++;
  }
  unlock();
}
```

1.1 Produttori e consumatori

Ipotizziamo adesso una situazione dove:

- Un processo, detto **produttore**, deposita un messaggio in un *buffer*;
- Un'altro processo, detto **consumatore**, preleva il messaggio dal *buffer*.

La policy sul buffer sarà la seguente:

- Il produttore non deve inserire un messaggio nel buffer se questo è pieno;
- Il consumatore non deve prelevare un messaggio dal buffer se questo è vuoto.

Possiamo usare 2 semafori per realizzare una prima soluzione:

- `spazio_disponibile`, con $s_0 = 1$, segnalerà quando il buffer è vuoto;
- `messaggio_disponibile`, con $s_0 = 0$, segnalerà quando il buffer è pieno.
- A questo punto il processo produttore dovrà controllare che il buffer sia vuoto (e aspettare che lo sia se non lo è), inserire il messaggio e segnalare che un nuovo messaggio è disponibile. In pseudocodice:

```
// produttore
do {
  // produci messaggio
  wait(spazio_disponibile);
  buffer.insert(messaggio);
  signal(messaggio_disponibile);
} while(!fine);
```

- Il consumatore dovrà invece controllare che il buffer abbia un nuovo messaggio (e aspettare che lo abbia se non lo ha), prelevare il messaggio e segnalare che il buffer è nuovamente vuoto. In pseudocodice:

```
// consumatore
do {
  wait(messaggio_disponibile);
  messaggio = buffer.extract();
  signal(spazio_disponibile); // prima segnala e poi consuma!
  // consuma messaggio
} while(!fine)
```

1.1.1 Più produttori e consumatori

Complichiamo la situazione: introduciamo un buffer ad n elementi, e prevediamo la presenza contemporanea di più produttori e consumatori.

In questo caso dovremmo assicurare, oltre che la sincronizzzazione coi due semafori appena visti, la mutua esclusione attraverso un mutex. Inoltre, dovremmo prevedere che il semaforo `spazio_disponibile` abbia $s_0 = n$, e non 1. Possiamo fidarci che il meccanismo dei semafori assicura il corretto ordinamento dei processi (per ogni messaggio che inseriamo, si libera uno e un solo consumatore).

- Avremo quindi che lo pseudocodice del produttore sarà:

```
// produttore
do {
  // produci messaggio
  wait(spazio_disponibile);
  {
    wait(mutex);
    buffer.insert(messaggio);
    signal(mutex);
  }
  signal(messaggio_disponibile);
} while(!fine);
```

- Mentre lo pseudocodice del consumatore sarà:

```
// consumatore
do {
  wait(messaggio_disponibile);
  {
    wait(mutex);
    messaggio = buffer.extract();
    signal(mutex);
  }
  signal(spazio_disponibile); // prima segnala e poi consuma!
  // consuma messaggio
} while(!fine)
```

1.1.2 Semafori distinti

Accorgiamoci che in questo sistema, i produttori si bloccano su `spazio_disponibile`, i consumatori si bloccano su `messaggio_disponibile`, ed entrambi si possono bloccare sul `mutex`. Questo non è particolarmente elegante e può portare a situazioni di rallentamento.

Possiamo risolvere questo problema usando, anzichè uno, 2 semafori di mutex.

- In questo caso lo pseudocodice del produttore sarà:

```
// produttore
do {
  // produci messaggio
  wait(spazio_disponibile);
  {
    wait(mutex_produttore);
    buffer.insert(messaggio);
    signal(mutex_produttore);
  }
  signal(messaggio_disponibile);
} while(!fine);
```

- Mentre lo pseudocodice del consumatore sarà:

```
// consumatore
do {
  wait(messaggio_disponibile);
  {
    wait(mutex_consumatore);
    messaggio = buffer.extract();
    signal(mutex_consumatore);
  }
  signal(spazio_disponibile); // prima segnala e poi consuma!
  // consuma messaggio
} while(!fine)
```

Questo chiaramente ci porta a dover fare delle considerazioni sulle modalità in cui si implementa il buffer. In particolare, vorremo che le operazioni `insert()` ed `extract()` siano completamente disaccoppiate e non possano collidere: questo perché la configurazione adottata permette a queste di essere eseguite contemporaneamente (l'una dal produttore e l'altra dal consumatore, che si bloccano su semafori diversi).

Se si adotta la classica implementazione ad array, questo problema non si pone. Se si usa una struttura più sofisticata come una lista, la situazione è più complicata. Vediamo nel dettaglio.

La soluzione che possiamo immaginare è di avere una lista con puntatore alla coda, dove le inserzioni (produttore) si fanno in coda, e le estrazioni (consumatore) si fanno in testa.

In questo caso, per liste con più di un elemento, operazioni di estrazione ed inserzione agiranno su oggetti completamente distinti in memoria, e non avremo problemi. Il problema sarebbe però quando si vuole avere un inserzione ed un estrazione parallela su una lista con un solo elemento.

1.2 Primitive di comunicazione

Tralasciamo per adesso i sistemi in memoria condivisa, e parliamo dei sistemi distribuiti, composti da nodi con memorie locali. Questo è ad esempio il caso delle reti di calcolatori.

Il problema che ci poniamo è come sfruttare un certo **canale di comunicazione** orientato per realizzare due primitive, la primitiva `send(destinazione, messaggio)`, e la primitiva `receive(origine, messaggio)`. Notiamo che la `send()` è *asincrona* (o ugualmente, non *sincrona* o non *bloccante*): quando si ritorna dalla chiamata, non si può avere la sicurezza che il messaggio sia stato recapitato. Una primitiva *bloccante*, di contro, avrebbe sospeso il chiamante fino all'arrivo del messaggio: questo chiaramente implica l'attesa di un ACK da parte del destinatario.

La `receive()` è invece necessariamente bloccante: il processo chiamante viene messo in attesa finché un messaggio non è stato effettivamente ricevuto e può essere recapitato.

Una soluzione più esplicita per il programmatore a cui forniamo le `send()` e `receive()` potrebbe essere quella di imporre il ricevimento dell'ACK, cioè:

```
// invia
send(destinazione, messaggio);
// aspetta l'ACK, e' bloccante
ack = receive(destinazione);
```

Questo sfrutta il fatto che la `receive()` è bloccante e effettivamente risolve il nostro problema, permettendoci di mantenere la `send()` asincrona. Chiaramente, però, richiede al programmatore di scrivere codice più complicato (e corretto!).

1.2.1 Formato del messaggio

Per dotarci di primitive di comunicazione, abbiamo bisogno di stabilire un **formato standard** per i messaggi che andiamo ad inviare. Questo è solitamente diviso in:

- **Intestazione**: contiene informazioni su:
  - **Origine** del messaggio;
  - **Destinazione** del messaggio;
  - **Tipo** del messaggio;
  - **Lunghezza** (in byte) del messaggio;
  - *Informazioni di controllo* varie sul messaggio.
- **Corpo**: contiene il messaggio vero e proprio, o *payload*, che vogliamo trasmettere.

1.2.2 Produttori e consumatori remoti

Vediamo un primo esempio di come le primitive di comunicazione potrebbero essere usate, ad esempio per realizzare un sistema produttore e consumatore.

In particolare, vediamo 2 varianti di comunicazione, **diretta simmetrica** e **diretta asimmetrica**:

- Comunicazione **diretta simmetrica**:
  - Lato produttore si avrà:

```
pid consumatore = /* ... */;
main() {
  msg mess;
  do {
    produci(&mess);
    send(consumatore, mess);
  } while(!fine);
}
```

  - Lato consumatore si avrà:

```
pid produttore = /* ... */;
main() {
  msg mess;
  do {
    receive(produttore, &mess);
    consuma(M);
  } while(!fine);
}
```

  Vediamo come in questo tipo di comunicazione non è necessario prevedere un *buffer*: si invia un messaggio per volta e si aspetta, lato consumatore, per ogni messaggio.

- Comunicazione **diretta asimmetrica**:
  - Lato produttore si avrà:

```
pid consumatore = /* ... */;
main() {
  msg mess;
  do {
    produci(&mess);
    send(consumatore, mess);
  } while(!fine);
}
```

– Lato consumatore si avrà:

```
main() {
  msg mess;
  pid produttore;
  do {
    receive(&produttore, &mess);
    consuma(M);
  } while(!fine);
}
```

In questo caso il produttore non è già noto al consumatore, che invece si mette in ascolto per il primo messaggio disponibile.

1.2.3 Modello client-server

Il discorso fatto finora su produttori e consumatori può essere sviluppato introducendo il paradigma (probabilmente già noto) **client-server**.

In questo caso prevediamo più processi, detti **client** (o *clienti*) che richiedono servizi ad un solo processo, detto **server** (o *servitore*). I client accedono al server tramite una determinata **porta**, che per quanto ci riguarda si occupa anche di *bufferizzazione* delle richieste dei client prima che queste arrivino al processo server vero e proprio.