1 Lezione del 29-10-25

1.1 Implementazione di un monitor

Veniamo quindi a come si può effettivamente implementare un monitor come descritto nella scorsa lezione.

Avere più funzioni in mutua esclusione significa effettivamente usare un semaforo di mutex `sem mutex = 1`, e avere il seguente prologo ed epilogo di funzione per ogni funzione interna al monitor:

```
func_monitor() {
  // prologo
  wait(mutex);

  // corpo func

  // epilogo
  signal(mutex);
}
```

Il problema diventa quindi la gestione delle variabili di condizione:

- Ricordiamo che `x.wait()` vuole che il processo attuale si sospenda;
- `x.signal()` potrebbe invece bloccare il processo e passare ad un altri (*signal and wait*) oppure continuare col processo corrente (*signal and continue*). Noi, come anticipato in 13.3.1, useremo la prima politica.

Avremo quindi un semaforo inizializzato a zero su ogni variabile di condizione (ad esempio `sem x_sem = 0`) per il blocco, e un contatore dei processi bloccati sulla variabile (ad esempio `int x_count = 0`).

- A questo punto la `x.wait()` sarà:

```
x.wait() {
  x_count++;

  signal(mutex); // devo sbloccare il mutex
  wait(x_sem);
}
```

Il problema è che facciamo una `signal(mutex)`, quando in verità vorremmo segnalare di proseguire ai processi già interni al monitor. Modifichiamo allora il moniotr, introducendo un semaforo `sem next = 0` e un contatore `int next_count = 0` per i processi *"bloccati"* al suo interno.

Prologo ed epilogo saranno allora:

```
func_monitor() {
  // prologo
  wait(mutex);

  // corpo func

  // epilogo
  if(next_count > 0) {
    signal(next); // prima fai uscire i processi interni
  } else {
    signal(mutex); // poi apri il monitor ad altri
  }
}
```

A questo punto il codice della `x.wait()` potrà essere:

```
x.wait() {
  x_count++;

  if(next_count > 0) { // c'e' qualcuno in attesa
    signal(next);
  } else { // non c'e' nessuno, sblocca il monitor
    signal(mutex);
  }

  wait(x_sem);
  x_count--; // uscito da wait()
}
```

- Implementiamo quindi la `x.signal()` secondo la politica signal and wait:

```
x.signal() {
  if(x_count > 0) {
    signal(x_sem);

    next_count++;
    wait(next); // sono uno dei processi del monitor
    next_count--;
  }
}
```

Ci dovrebbe quindi essere chiaro il funzionamento del monitor come un ambiente "ristretto" per i processi del sistema dove lo scheduling non è necessariamente FCFS (o qualsiasi fosse l'algoritmo usato dallo scheduler del sistema).

1.1.1 Conditional wait

Un'altra possibile politica che si può adottare all'interno dei monitor è la cosiddetta **conditional wait**, nella forma `x.wait(c)` dove `c` è un *numero di priorità*. I processi con numero di priorità più piccolo (priorità più alta) vengono schedulati per primi.

Un esempio dove potrebbe essere utile usare tale costrutto è il seguente, dove si implementa un monitor con il compito di allocare una certa risorsa:

```
monitor ResourceAllocator {
  boolean busy;
  condition x;
  void acquire(int time) {
    if(busy) x.wait(time);
    busy = TRUE;
  }

  void release() {
    busy = FALSE;
    x.signal();
  }

  initialization code() {
    busy = FALSE;
  }
}
```

In questo caso prendiamo come argomento `time`, cioè il tempo per cui occupiamo la risorsa (meno tempo $\rightarrow$ più priorità).

1.2 Deadlock

Veniamo quindi alla trattazione vera e propria dei **deadlock**, o *blocchi critici*, che avevamo introdotto in 13.2.

Di base, questi sono situazioni dove ciascun processo, in un insieme di processi, detiene una risorsa e ne desidera una di un altro. Sul grafo di allocazione, equivalentemente, significa che abbiamo un *ciclo*.

Ricordiamo che ci eravamo posti di implementare appropiate tecniche di deadlock **detection** (*rilevamento* di deadlock) e deadlock **avoidance** (*risoluzione* o *prevenzione* di deadlock).

Notiamo adesso che in verità esistono casi dove si possono avere cicli nel grafo di allocazione, ma non avere deadlock: questo è il caso di risorse con **istanze multiple**. In questocaso, un ciclo in un grafo di allocazione simboleggia la *possibilità* di aver deadlock, ma la situazione potrebbe comunque risolversi (lo si verifica per osservazione del grafo).

Una soluzione banale al problema del deadlock è obbligare il programmatore a dichiarare subito tutte le risorse di cui il programma ha bisogno: a questo punto, lato S/O, si potrà realizzare via mutex su tali risorse un sistema di bloccaggio che eviterà sempre i deadlock. Il problema è chiaramente che tale vincolo è estremamente restrittivo, e renderebbe non solo molto scomodo per il programmatore programmare una data applicazione, ma in generale abbatterebbe l'efficienza dell'intero sistema.

1.2.1 Condizioni di deadlock

Esistono 4 condizioni necessarie affinché si verifichi un deadlock:

1. **Mutua esclusione**: solo un processo per volta può usare una data risorsa;
2. **Hold and wait**: un processo che ha ottenuto almeno una risorsa si mette in attesa di altre risorse ottenute da altri processi;
3. **No preemption**: una risorsa può essere rilasciata solo *volontariamente* dai processi che la ottengono, quando questi completano la loro operazione sulla stessa;
4. **Attesa circolare**: esiste un insieme $\{p_0, p_1, ..., p_n\}$ di processi in attesa tali che p_0 aspetta una risorsa di p_1, p_1 aspetta una risorsa di p_2, ..., p_{n-1} aspetta una risorsa di p_n.

Si ha che difficilmente si può lavorare sulle prime 3 condizioni: la muta esclusione deriva dal fatto che ci sono fisicamente risorse limitate nel sistema, e non vogliamo imporre al programmatore vincoli su come e quali risorse vogliono ottenere, o constringerli a programmare meccanismi di recupero dal ritiro di risorse (*preemption* su risorse).