COMPITO CON SOLUZIONI

Sistemi Operativi (modulo II / B)

15 gennaio 2019

Esercizio 1

Considerare un thread `produttore`, che produce due elementi alla volta, e un thread `consumatore` che consuma un elemento alla volta. I thread comunicano tramite un `buffer` di dimensione `MAX`:

```
semaphore vuote=MAX, piene=0, mutex=1; // semafori per celle vuote, piene e mutua esclusione

thread produttore {
  while(true) {
    <produce d1 e d2>
    P(vuote); // alloca una cella vuota
    P(vuote); // alloca una cella vuota
    P(mutex); // mutua esclusione su buffer
    buffer.scrivi(d1);
    buffer.scrivi(d2);
    V(mutex); // rilascia mutua esclusione
    V(piene); // libera una cella piena
    V(piene); // libera una cella piena
  }
}
```

```
thread consumatore {
  while(true) {

    P(piene); // alloca una cella piena
    P(mutex); // mutua esclusione su buffer
    d = buffer.leggi();
    V(mutex); // rilascia mutua esclusione
    V(vuote); // libera una cella vuota

    <consuma d>
  }
}
```

Sincronizzare i thread tramite semafori in modo che: il `produttore` si blocchi se il `buffer` è pieno, il `consumatore` si blocchi se il `buffer` è vuoto, produttore e consumatore non leggano e scrivano contemporaneamente nel `buffer` (scrivere le P e le V direttamente nel codice sopra, indicando la inizializzazione dei semafori e spiegare brevemente): Utilizziamo un semaforo `mutex` inizializzato a 1 per garantire la mutua esclusione sull'utilizzo del `buffer`. Viene eseguita una `P(mutex)` prima di letture o scritture e una `V(mutex)` al termine di tali operazioni. Utilizziamo un semaforo `vuote` inizializzato a `MAX` per allocare le celle vuote al produttore. Poiché il produttore scrive due valori alla volta, eseguiamo due `P(vuote)` prima della `P(mutex)`. Il consumatore eseguirà una `V(vuote)` dopo che ha liberato una cella. In modo analogo, utilizziamo un semaforo `piene` per allocare le celle piene al consumatore, che eseguirà una `P(piene)` prima di leggere dal buffer. Il produttore, eseguirà due `V(piene)` dopo aver effettuato le scritture.

Esercizio 2

Considerare due thread che richiedono due risorse (0 e 1) tramite un monitor `risorse` nel modo seguente:

```
Thread T0 {
  monitor.richiedi(0); monitor.richiedi(1);
  ...
  monitor.rilascia(0); monitor.rilascia(1);
}
```

```
Thread T1 {
  monitor.richiedi(1); monitor.richiedi(0);
  ...
  monitor.rilascia(0); monitor.rilascia(1);
}
```

Implementare il monitor `risorse` in modo che garantisca mutua esclusione nell'allocazione delle risorse

```
Monitor risorse {
  r[2] = {true, true}; // risorse disponibili

  void richiedi(int i) {
    // se non disponibile attende
    while (!r[i]) wait();
    r[i] = false; // alloca la risorsa
  }
}
```

```
void rilascia(int i) {
  r[i] = true; // libera la risorsa
  notify();    // sblocca l'altro thread
}
```

Discutere la possibilità di stallo e una possibile soluzione che lo prevenga: Se `risorse.richiedi(0)` e `risorse.richiedi(1)` vengono eseguite rispettivamente dai due thread ci troviamo in una situazione di stallo in quanto le successive richieste saranno entrambi bloccanti, in modo analogo a quanto accade nel problema dei filosofi a cena (situazione di attesa circolare). Una possibile soluzione è ammettere la richiesta incrementale di risorse solo secondo un ordine prestabilito (allocazione gerarchica). Ad esempio possiamo stabilire che `r[0]` < `r[1]` e ritornare un errore nel caso le richieste non rispettino tale ordine. L'errore dovrà però essere gestito dai thread che invocano il metodo `risorse.richiedi()`.