— COMPITO CON SOLUZIONE —

Sistemi Operativi (modulo II / B)

17 gennaio 2013

Esercizio 1

Si consideri il generico thread `T(i)`, con $0 \leq$ `i` $<$ `MAX`, che esegue il seguente codice:

```
thread T(i) {
    P(s[i]);
    somma = somma + i;
    print somma;
    if (i < MAX-1)
        V(s[i+1]);
}
```

dove

- `s[MAX]` è un array di semafori binari condiviso dai thread. Tutti i semafori dell'array sono inizializzati a false ad eccezione di `s[0]` che inizializzato a true.
- `somma` è una variabile intera condivisa dai thread e inizializzata a 0.

Considerare MAX = 4 e un programma che manda in esecuzione, uno dopo l'altro, i seguenti thread:

1. `T(0), T(2), T(1)`
2. `T(2), T(1), T(0)`
3. `T(2), T(3), T(0)`

Per ogni caso indicare l'output spiegando brevemente (risposte senza spiegazione non verranno valutate).

SOLUZIONE: I thread attendono sul 'proprio' semaforo `s[i]`, aggiungono il proprio id alla somma e la stampano. Infine sbloccano il thread successivo facendo una `V(s[i+1])` (nel caso non si sia raggiungo il massimo). L'unico semaforo verde è `s[0]`, di conseguenza l'unico thread che non si blocca sulla prima P è `T(0)`. Tutti gli altri thread andranno in attesa sul 'proprio' semaforo `s[i]` e vengono sbloccati a catena dal precedente (in ordine di id). La coseguenza è che gli id vengono sommati in progressione alla somma. Se un thread non viene eseguito bloccherà anche tutti quelli successivi. In dettaglio:

1. `T(0), T(2), T(1)`. Stampa '0','1','3';
2. `T(2), T(1), T(0)`. Stampa '0','1','3';
3. `T(2), T(3), T(0)`. Stampa '0' e poi va in stallo perché manca il thread `T(1)`.

Esercizio 2

Considerare la seguente soluzione `software` (insoddisfacente) al problema della sezione critica per più processi:

```
while (locked) {}
locked=true;

< sezione critica >

locked=false;
```

in cui la variabile condivisa `locked` è inizializzata a `false`. Mostrare una esecuzione con 2 thread in cui NON viene garantita la *mutua esclusione*. Discutere una variante (corretta) della soluzione proposta basata su istruzioni hardware speciali.

SOLUZIONE: L'idea di garantire mutua esclusione acquisendo un lock ogni volta che si accede alla sezione critica: finché il lock (variabile `locked`) è attivo si attende in 'busy-waiting'; quando il lock è rilasciato lo si acquisisce (`locked=true`) e si accede alla sezione critica; all'uscita della sezione critica si rilascia il lock (`locked=false`). Il problema insorge se 2 thread trovano `locked` a false e superano 'contemporaneamente' il ciclo while (nel caso di uno scheduling 'sfortunato' o in esecuzione su core distinti): entrambi settano `locked` a true e entrano in sezione critica.

Una variante corretta è quella basata sull'istruzione hardware speciale `test-and-set`:

```
while (test-and-set(&locked)) {}

< sezione critica >

locked=false;
```

`test-and-set` setta il valore di `locked` a true e ritorna il valore precedente. In questo modo se `locked` vale true il thread resta in busy-waiting e il valore non viene modificato (è già true), se invece `locked` vale false il processo entra in sezione critica ma contemporaneamente (in modo atomico) acquisisce il lock: nessun altro processo entrerà in sezione critica.