Sistemi Operativi I Modulo
Esercizi Lezione 06/12/2021

Igor Melatti

Talvolta le applicazioni concorrenti sono divise in fasi con la regola che nessun processo può proseguire se prima tutti i suoi simili non sono pronti a farlo. Le barriere implementano questo concetto: un processo che ha terminato la sua fase chiama una primitiva `barrier()` e si blocca. Quando tutti i processi coinvolti hanno terminato il loro stadio di esecuzione invocando anch'essi la primitiva `barrier()`, il sistema li sblocca tutti permettendo di passare ad uno stadio successivo. Un'applicazione concorrente composta da N processi utilizza più volte durante la sua esecuzione il meccanismo della `barrier()`. Dovendo migrare l'applicazione in questione ad un sistema operativo che non fornisce tale meccanismo, ma che fornisce condivisione di memoria e semafori, siete costretti a scrivere voi stessi la primitiva `barrier()` in termini dei meccanismi disponibili.

Variante 1: anziché i semafori, supporre che sia disponibile solo l'istruzione atomica `exchange` (e ovviamente la memoria condivisa).

Variante 2: anziché i semafori, supporre che sia disponibile solo l'istruzione atomica `compare_and_swap` (e ovviamente la memoria condivisa).

Variante 3: anziché i semafori, supporre che sia disponibile solo il meccanismo del passaggio di messaggi tra processi con mailbox. Inoltre, fare in modo che l'unica memoria condivisa usata sia costituita dalle mailbox.

Una soluzione con i semafori è mostrata in Figura 1. Tuttavia, tale soluzione è sbagliata. Trovare un controesempio, partendo dal fatto che è possibile chiamare 2 volte la funzione `barrier` e che il dispatcher vada in esecuzione tra `signal(mutex)` e `wait(barrier_sem)`. Provare a correggere (è necessario introdurre un terzo semaforo).

Una soluzione con l'istruzione `compare_and_swap` è mostrata in Figura 2. Anche questa soluzione non è corretta nel caso più generale: cosa succede se il dispatcher interviene tra `mutex = 0` e `while(wait_here[ord] != 0);`? Ricavare una soluzione corretta a partire da quella con i semafori.

Infine, una soluzione con lo scambio messaggi è mostrata in Figura 3. Anche questa soluzione non è corretta: ricavare una soluzione corretta a partire da quella con i semafori.

```
semaphore mutex = 1;
semaphore barrier_sem = 0;
int n_barrier = 0;

barrier() {
  wait(mutex);
  n_barrier++;
  if (n_barrier < N) {
    signal(mutex);
    wait(barrier_sem);
  }
  else {
    n_barrier = 0;
    for (i = 1; i < N; i++)
      signal(barrier_sem);
    signal(mutex);
  }
}
```

Figure 1: Soluzione (sbagliata) con i semafori

```
int mutex = 0, mutex2 = 0;
int wait_here[N];
int n_barrier = 0;

barrier() {
  int ord;
  while(compare_and_swap(mutex, 0, 1) == 1);
  n_barrier++;
  if (n_barrier < N) {
    ord = n_barrier;
    wait_here[ord] = 1;
    mutex = 0;
    while(wait_here[ord] != 0);
  }
  else {
    n_barrier = 0;
    mutex = 0;
    for (i = 1; i < N; i++)
      wait_here[i] = 0;
  }
}
```

Figure 2: Soluzione (sbagliata) con funzione `compare_and_swap`

```
mailbox box_n, box_ok, box_unbl, box_previous;

init() {
  box_n = create_mailbox();
  box_ok = create_mailbox();
  box_unbl = create_mailbox();
  box_previous = create_mailbox();
  nbsend(box_n, 0);
  nbsend(box_previous, null);
}

barrier() {
  int n_barrier, n_unblocked;
  receive(box_previous, null);
  nbsend(box_previous, null);
  receive(box_n, n_barrier);
  nbsend(box_n, n_barrier + 1);
  if (n_barrier + 1 < N) {
    receive(box_ok, null);
    receive(box_unbl, n_unblocked);
    nbsend(box_unbl, n_unblocked + 1);
    if (n_unblocked + 1 == N - 1)
      nbsend(box_previous, null);
  }
  else {
    nbsend(box_unbl, 0);
    receive(box_previous, null);
    for (i = 1; i < N; i++)
      nbsend(box_ok, null);
    nbsend(box_n, 0);
  }
}
```

Figure 3: Soluzione (sbagliata) con messaggi