Mutua esclusione - soluzioni software

Index

Introduction

Le soluzione software è usabile solo per problemi di concorrenza semplici come ad esempio la mutua esclusione.
In questo caso dunque non possiamo fare affidamento alle istruzioni macchina, ma solamente all'assegnamento a variabili e simili. Come contro però si ha il fatto che tutte le operazioni sono in attesa attiva (non possono essere bloccati i processi)

Prove

Facciamo dei tentativi per provare a implementare la mutua esclusione via software

Primo tentativo

```
   /* PROCESS 0 /*                  /* PROCESS 1 */
.                                .
.                                .
while (turn != 0)                while (turn != 1)
   /* do nothing */ ;               /* do nothing */;
/* critical section*/;            /* critical section*/;
turn = 1;                        turn = 0;
.                                .
```

Questa soluzione però è applicabile solo a 2 processi (non a più processi)

Problemi

Una soluzione come questa risolve il problema della mutua esclusione ma con dei problemi.
Il maggiore di tutti sta nel fatto che funziona se ci sono due processi, ma se ce ne fosse uno solo (`PROCESS 1`) e `turn` fosse inizializzato a 1 non si uscirebbe mai del processo

Secondo tentativo

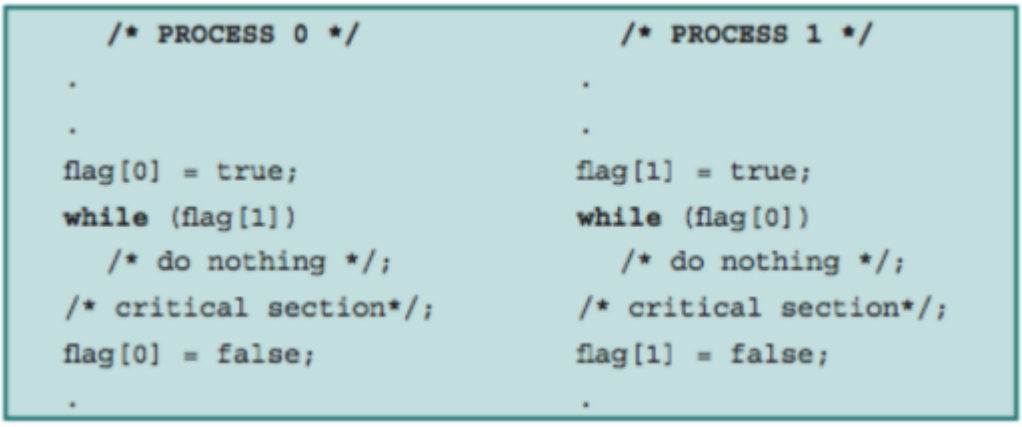
```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                .
.                                .
while (flag[1])                  while (flag[0])
   /* do nothing */;                 /* do nothing */;
flag[0] = true;                  flag[1] = true;
/*critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
.                                .
```

Il `PROCESS 0` legge la variabile di `PROCESS 1` e scrive la propria, mentre `PROCESS 1` fa il contrario

Problemi

In questo caso se lo scheduler interrompe `P0` immediatamente prima della modifica di `flag[0]` per passare a `P1` anche lui potrebbe entrare nella critical section e quindi si avrebbe una race condition

Terzo tentativo

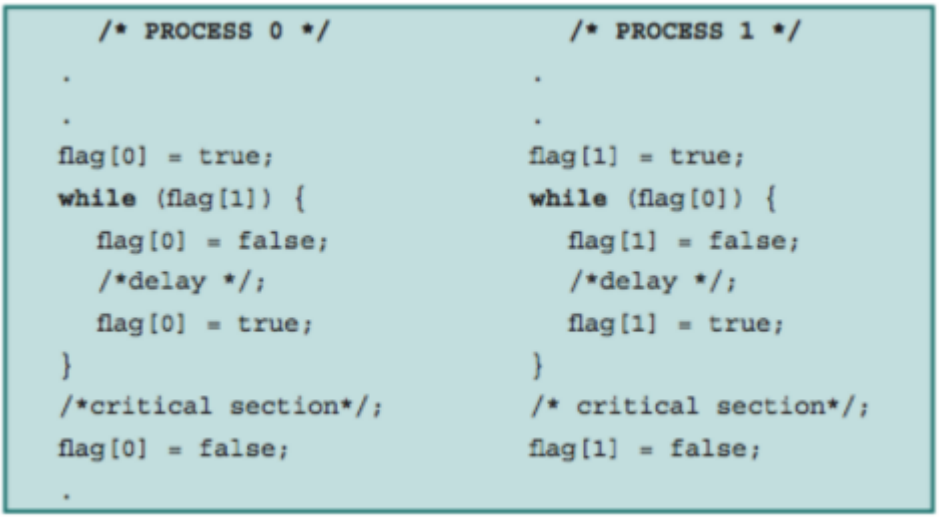


```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                .
.                                .
flag[0] = true;                  flag[1] = true;
while (flag[1])                  while (flag[0])
   /* do nothing */;                 /* do nothing */;
/* critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
.                                .
```

Problemi

Anche qui, se lo scheduler interrompe subito dopo aver impostato il `flag` i processi rimarrebbero bloccati nel `while` (deadlock)

Quarto tentativo



```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                .
.                                .
flag[0] = true;                  flag[1] = true;
while (flag[1]) {                while (flag[0]) {
   flag[0] = false;                 flag[1] = false;
   /*delay */;                      /*delay */;
   flag[0] = true;                  flag[1] = true;
}                                }
/*critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
.
```

Problemi

In questa soluzione si tenta di risolvere il problema di deadlock modificando nuovamente il `flag` dentro il `while`, ma in questo caso devo sperare che lo scheduler interrompa il processo prima della fine del `delay`. In questo caso più che deadlock, si parla di **livelock**, ovvero i processi continuano a fare qualcosa ma non di utile

---

Algoritmo di Dekker

Nell'algoritmo di Dekker si implementano le due soluzioni, una variabile condivisa e una locale

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

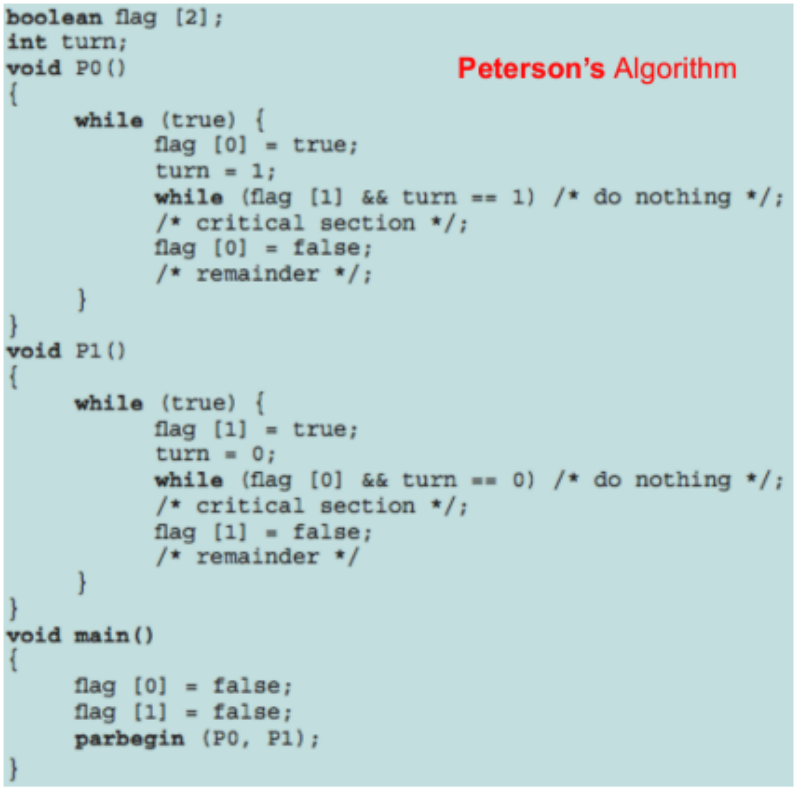
Qui fin dall'inizio dichiaro di voler entrare nella sezione critica. Se il `wants_to_enter` dell'altro processo è `false` entro nella sezione critica. Nel caso in cui invece il valore è `true`, si ha una variabile `turn` condivisa. Per il `P0` se `turn` è 0 (non tocca a me), rimetto a falso il fatto che voglio entrare e faccio un'attesa attiva finché il `turn` è 1. Una volta finita l'attesa ri-imposto il fatto che voglio entrare a `true`.

Questo algoritmo vale solo per 2 processi estendibile a N (seppur non banale, non so a cosa impostare `turn`). Garantisce inoltre la non-starvation (grazie a `turn`) e il non-deadlock (ma è busy-waiting, se ci sono delle priorità fisse, ci può essere deadlock). Non richiede alcun supporto dal SO, ma in alcune moderne architetture hanno ottimizzazioni hardware che riordano le istruzione da eseguire, nonché gli accessi in memoria, risulta dunque necessario disabilitare tali ottimizzazioni

---

Algoritmo di Peterson

L'algoritmo di Peterson permette di risolvere lo stesso problema in maniera più semplice



In questo caso il processo dice che a dover passare è l'altro processo ed entrando nel `while` solamente se è il turno dell'altro processo e inoltre vuole entrare (non si hanno problemi se in esecuzione si ha un solo processo). Anche qui, facendo un **interleaning** perfetto, non si avrebbero problemi in quando viene mandato in esecuzione il penultimo processo che ha impostato `turn`

Anche questo vale solo per 2 processi e l'estensione a N processi è più semplice rispetto al Dekker. Come per Dekker non si ha starvation, deadlock ma problemi per le CPU che riordinano gli accessi in memoria. Si ha inoltre il **bounded-waiting** tramite il quale un processo può aspettare l'altro al più una volta (vale anche per Dekker ma non per la generalizzazione a $N > 2$ processi)