# 1) Semafori

Si consideri una versione del problema produttori/consumatori con due buffer condivisi: due array di interi, A e B.

Diciamo che l’intero A[i] (o l’intero B[i]) è presente se è stato prodotto da un produttore e, dopo che è stato prodotto, non è stato ancora consumato da nessun consumatore.

Inizialmente, nessun A[i] è presente e nessun B[i] è presente.

Gli array A e B sono condivisi da 3 tipi di thread:

1. consumatori: se almeno un A[i] oppure almeno un B[i] è presente, il thread consuma TUTTI gli interi presenti in A e in B e stampa la loro somma.

2. A-produttori : se esiste almeno un indice i tale che A[i] non è presente, il thread seleziona uno di questi i e produce un intero in A[i].

In particolare, l’intero viene calcolato chiamando un metodo mA( i ), con l’istruzione A[ i ] = mA( i ).

Altrimenti, il thread va in attesa e vi rimane fintantoché tutti gli A[i] sono presenti.

3. B-produttori: se esiste almeno un indice i tale che B[i] non è presente, il thread seleziona uno di questi i e produce un intero in B[i].

In particolare, l’intero viene calcolato chiamando un metodo mB( i ), con l’istruzione A[ i ] = mB( i ).

Altrimenti, il thread va in attesa e vi rimane fintantoché tutti i B[i] sono presenti.

Scrivere il codice dei tre tipi di thread, usando i semafori con la semantica tradizionale e garantendo che:

- non si possano verificare race condition su A e su B

- un thread può essere in waiting solo nei seguenti casi:

\* l’attesa è necessaria per prevenire race condition.

\* l’attesa è imposta dalle specifiche ai punti 1, 2 e 3.

Possibile soluzione:

Variabili condivise usate:

int k = 0; // posizione di A in cui il prossimo PA può produrre. Se k = A.length allora nessuno può produrre.

int h = 0; // posizione di B in cui il prossimo PB può produrre. Se k = B.length allora nessuno può produrre.

int workingPA = 0; // numero di PA working, cioè numero di PA che stanno producendo.

int workingPB = 0; // numero di PB working, cioè numero di PB che stanno producendo.

boolean workingC = false; // true se e solo se un c'è working, cioè sta consumando.

int waitingPA = 0; // numero di PA in attesa

int waitingPB = 0; // numero di PB in attesa

int waitingC= 0; // numero di C in attesa

Semafori:

mutex = 1; // serve per usare le variabili di cui sopra nelle sezioni critiche

mutexPA =1; // serve per consentire ai PA di determinare la posizione in cui produrre all’interno di sezioni critiche

mutexPB = 1; // analogo a PA

semPA = 0; // serve per mettere in attesa i PA che non possono produrre.

semPB = 0; // serve per mettere in attesa i PB che non possono produrre.

semC = 0; // serve per mettere in attesa i C che non possono consumare.

## ProduttoreA (analogo per ProduttoreB):

while(true){

… // something having nothing to do with A and B

wait(mutex); // sulle variabili condivise workingC, workingPS, eccetera, lavoro all’interno di sezioni critiche

if(workingC || k + workingPA == A.length ) { // vado in attesa (sul semaforo semPA) se un C sta consumando oppure se A è pieno/verrà riempito da PA che hanno già controllato di poter produrre.

waitingPA++ ;

signal(mutex);

wait(semPA);

}

else{

workingPA++ ;

signal(mutex);

}

wait(mutexPA)

int i = k ; // determino la posizione in cui produrre

k++; // aggiorno la posizione in cui il prossimo PA può produrre

signal(mutexPA);

A[i] = mA( i ); // vari PA possono produrre in concorrenza con altri PA e con PB

wait(mutex);

workingPA - - ;

if( workingPA==0 & workingPB==0 & waitingC > 0 ) { // controllo se posso svegliare un C

waitingC - - ;

workingC = true;

signal(semC);

}

signal(mutex);

}

## Consumatore:

while(true){

… // something having nothing to do with A and B

wait(mutex);

if( workingPA>0 || workingPB>0 || (k==0 & h==0)) { // se ci sono PA o PB che stanno producendo, oppure non ci sono elementi da consumare, vado in attesa

waitingC + + ;

signal(mutex);

wait(semC);

}

else {

workingC=true;

signal(mutex);

}

int sum = 0;

for(int j = 0 ; j<k ; j++) { sum= sum+A[j]; }

for(int j = 0 ; j<h ; j++) { sum= sum+A[j]; }

System.out.println(sum);

wait(mutex);

k=h=0; // aggiorno k e h perché gli array ora sono vuoti

int c = 0;

while(waitingPA >0 & c < A.length) { // sveglio al massimo A.length PA in attesa

waitingPA - - ;

workingPA + + ;

signal(semPA);

}

c = 0;

while(waitingPB >0 & c < B.length) { // sveglio al massimo B.length PB in attesa

waitingPB - - ;

workingPB + + ;

signal(semPB);

}

workingC = false;

signal(mutex);

}

# 2) Race condition

Si assuma la variabile X condivisa da due thread che invocano il seguente metodo M:

public void M( ) { if ( X >= 0 ){ X = - 5; } else { X = 12 ; }

Discutere formalmente se si possono verificare race condition su X.

L’effetto sulla variabile X dell’esecuzione di ognuno dei due thread può essere formalizzato con la funzione f: R—> R definita, per parti, come segue:

- f(y) = 12, se y < 0.

- f(y) = -5, se y >= 0.

Pertanto, il comportamento dei due thread che eseguono sequenzialmente può essere formalizzato applicando due volte f:

- f(f(y)) = f(12) = -5, se y < 0.

- f(f(y)) = f(-5) = 12, se y >= 0.

Pertanto, se il valore iniziale di X è non negativo, per non avere race condition la variabile X deve assumere valore finale 12.

Altrimenti, se il valore iniziale di X è negativo, per non avere race condition la variabile X deve assumere valore finale -5.

Assumiamo che inizialmente X valga 10. Un thread potrebbe testare la guardia e perdere il processore prima di eseguire il ramo “then”.

L’altro thread potrebbe eseguire tutto il metodo. Il primo thread riprenderebbe in seguito l’esecuzione ed eseguirebbe il ramo “then”.

Alla fine X varrebbe -5, ma l’unico valore ammissibile partendo da X=10 è 12, pertanto in questo caso avremmo race condition.

# 3) Teoria

Che cos'è una PCB e qual è il suo ruolo?