# 1) Semafori

Per accedere ad una palestra servono dei gettoni.

Ogni utente necessita di un numero di gettoni che dipende dalle attività che andrà a fare.

Abbiamo 3 tipi di utenti:

* Utenti di tipo 1: per accedere alla palestra devono acquisire 1 gettone.
* Utenti di tipo 2: per accedere alla palestra devono acquisire 2 gettoni.
* Utenti di tipo 3: per accedere alla palestra devono acquisire 3 gettoni.

La palestra ha 100 gettoni.

Se un utente di tipo n vuole accedere alla palestra, devono esserci n gettoni disponibili, che torneranno ad essere disponibili quando l’utente uscirà dalla palestra.

Programmare l’ingresso e l’uscita dalla palestra di ogni utente, nel rispetto di quanto segue:

- un utente che vuole entrare in palestra ma che non ha gettoni disponibili va in attesa.

- se un utente di tipo n è in attesa, allora i gettoni disponibili sono meno di n.

Possibile soluzione:

Uso le seguenti variabili:

token = 100; // numero gettoni disponibili

waiting1 = waiting2 = waiting3 = 0; // numero utenti in attesa, per ogni tipologia.

Uso i seguenti semafori:

mutex=1;

S1 = S2 = S3 = 0;

## Utente1:

while(true){

// attività non rilevante

wait(mutex);

if(token>0){ // se c’è almeno un token

token - - ; // lo consumo e entro

signal(mutex);

}

else{ // altrimenti

waiting1++;

signal(mutex);

wait(S1); // vado in attesa

}

// training activity

wait(mutex);

if(waiting1>0){ // se c’è un U1 in attesa lo sveglio

waiting1 - - ;

signal(S1);

}

else{ // altrimenti

if(waiting2>0 & token==1){ // se c’è un token disponibile e un U2 in attesa lo sveglio

token - - ; // l’utente U2 usa il token disponibile e il mio

waiting2 - - ;

signal(S2);

}

else{ // altrimenti

if(waiting3>0 & token==2){ // se ci sono due token disponibili e un U3 in attesa lo sveglio

token=token-2; // l’utente U3 usa il mio token e i due disponibili

waiting3 - - ;

signal(S3);

}

else{

token + +; // se non sveglio nessuno il mio token diventa disponibile

}

}

}

signal(mutex);

}

// attività non rilevante

}

## Utente2:

while(true){

// attività non rilevante

wait(mutex);

if(token>1){ // se ci sono almeno 2 token disponibili

token = token -2; // li consumo e entro

signal(mutex);

}

else{ // altrimenti

waiting2++;

signal(mutex);

wait(S2); // vado in attesa

}

// training activity

wait(mutex);

if(waiting2>0){ // se c’è un U2 in attesa lo sveglio

waiting2 - - ;

signal(S2);

}

else{ // altrimenti

if(waiting3>0 & (token==1 || token==2){ // se ci sono 1 o 2 token disponibili e un U3 in attesa lo sveglio

token - - ;

waiting3 - - ;

signal(S3);

}

else{ // altrimenti

token = token + 2;

while(waiting1>0){ // sveglio, se in attesa, al massimo due U1

token=token-1;

waiting1 - - ;

signal(S1);

}

}

}

signal(mutex);

}

// attività non rilevante

}

## Utente3:

while(true){

// attività non rilevante

wait(mutex);

if(token>2){ // se ci sono almeno 3 token disponibili li uso e entro

token = token - 3;

signal(mutex);

}

else{ // altrimenti vado in attesa

waiting3++;

signal(mutex);

wait(S3);

}

// training activity

wait(mutex);

if(waiting3>0){ // se c’è un U3 in attesa lo sveglio

waiting3 - - ;

signal(S3);

}

else{ // altrimenti

token = token + 3; // con i miei token posso svegliare U2 e/o U1

while(waiting2>0 & token>1){

token = token -2;

waiting2 - - ;

signal(S2);

}

while(waiting1>0 & token>0){ //

token = token -1;

waiting1 - - ;

signal(S1);

}

}

signal(mutex);

}

// attività non rilevante

}

# 2) Race condition

Assumiamo che due thread condividano una variabile double X, con valore iniziale X=100.

Il primo thread esegue l’istruzione X=50;

Il secondo thread esegue l’istruzione if (X>70) { X=200; } else { X=0; }

Discutere, formalmente, se è possibile che si verifichino race condition sulla variabile X.

L’operazione eseguita dal primo thread può essere formalizzata con la funzione f1: R—> R definita come segue (x è un valore reale, non va confuso con la variabile X):

f1(x) = 50.

L’operazione eseguita dal secondo thread può essere formalizzata con la funzione f2: R—> R definita, per parti, come segue:

f2(x) = 200, se x > 70

f2(x) = 0, se x <= 70

Pertanto, componendo le funzioni come segue, spieghiamo il comportamento dei thread quando la loro attività viene sequenzializzata in ordine arbitrario:

f1(f2(x)) = f1(200) = 50, se x > 70

f1(f2(x)) = f1(0) = 50, se x <= 70

e

f2(f1(x)) = f2(50) = 0

In particolare, tenendo conto che il valore iniziale di X è 100, vale che:

f1(f2(100)) = f1(200) = 50

e

f2(f1(x)) = f2(50) = 0

Possiamo pertanto dire che avremo race condition se e solo se X può acquisire valori diversi da 50 e 0.

Una possibile esecuzione è la seguente: il secondo thread testa la guardia (X>70), che risulta vera, perde il processore, il secondo thread esegue l’assegnamento, il primo thread riacquisisce il processore e assegna il valore 200 alla variabile X, che non verrà modificato in seguito.

Essendo 200 diverso da 50 e da 0, concludiamo che questa esecuzione da luogo a race condition.