ESAME 1

Assumiamo un array di interi A.

L’array è condiviso da thread di tre tipi:

Thread di tipo WP: eseguono for(int i=0 ; i<A.length ; i=i+2){ A[i]=A[i]+r.nextDouble(); }

con r un oggetto di classe Random privato

Thread di tipo WD: eseguono for(int i=1 ; i<A.length ; i=i+2){ A[i]=A[i]+r.nextDouble(); }

con r un oggetto di classe Random privato

Thread di tipo O: eseguono quicksort(A,0,A.length-1);

Scrivere il codice per i 3 tipi di thread, usando i semafori con la semantica tradizionale, prevenendo race condition e busy waiting e assicurando che non ci siano mai thread in attesa a meno che ciò non sia necessario per prevenire race condition.

Usiamo le seguenti variabili condivise:

boolean wrkP = false; // wrkP is true if and only if one WP is working on A

boolean wrkD = false; // wrkD is true if and only if one WD is working on A

boolean wrkO = false; // wrkO is true if and only if one O is working on A

i

int waitingP= 0; // waitingP is n if and only if n WP are waiting for working on A

int waitingD= 0; // similar

int waitingO= 0; // similar

Uso i seguenti semafori:

sP = sD = sO = 0;

mutex=1;

thread WP (thread WD is similar):

…

Random r = new Random();

…..

while(true){

….

wait(mutex);

if(!wrkP & !wrkO){ // guard true if and only if this thread can work on A

wrkP=true;

signal(mutex);

}

else{

waitingP++;

signal(mutex);

wait(sP); // it the thread cannot work on A, it waits on sP

}

for(int i=0 ; i<A.length ; i=i+2){A[i]=A[i]+r.nextDouble();}

wait(mutex);

if(waitingP>0){ // if there are threads WP waiting for working on A, one of them is waken up

waitingP - - ;

signal(sP);

}

else{

if(waitingO>0 & !wrkD){ // otherwise, if there are threads O waiting for working on A, and

waitingO - - ; // no thread WD is working on A, one thread WP is waken up

wrkO=true;

wrkP=false;

signal(sO);

}

}

signal(mutex)

….

}

thread WO:

while(true){

….

wait(mutex);

if(!wrkP & !wrkD & !wrkO){ // guard true if and only if this thread can work on A

wrkO=true;

signal(mutex);

}

else{

waitingO++;

signal(mutex);

wait(sO); // it the thread cannot work on A, it waits on sO

}

quicksort(A,0,A.length-1);

wait(mutex);

if(waitingO>0){ // if there are threads WO waiting for working on A, one of them is waken up

waitingO - - ;

signal(sO);

}

else{

wrkO=false;

if(waitingP>0 ){

waitingP - - ;

wrkP=true;

signal(sP);

}

if(waitingD>0 ){

waitingD - - ;

wrkD=true;

signal(sD);

}

}

signal(mutex)

….

}

Supponiamo di avere un array A di interi condiviso da due thread che eseguono il seguente metodo:

public void m( ){ int temp = A[0] ; A[0] = A[1] ; A[1] = temp; }

Spiegare, formalmente, se sull’array A sono possibili race condition.

Osserviamo che ogni thread avrà una variabile temp nel proprio stack.

Informalmente, temp non è una variabile condivisa, ma ogni thread ha una “propria” temp.

Gli effetti che si hanno sull’array eseguendo il metodo m possono essere spiegati con la funzione

f: (Z x Z)—> (Z x Z) (nota: Z è l’insieme dei numeri interi)

definita come segue:

f(u , v) = (v , u) per ogni u e v appartenenti a Z.

Pertanto, se i due thread eseguono il metodo non in concorrenza, l’effetto della loro esecuzione è spiegato come segue:

f(f(u , v)) = f(v , u) = (u , v).

Di conseguenza, in base alla definizione di race condition, se A[0] = u e A[1] = v sono i valori iniziali dei due interi, abbiamo race condition se dopo che entrambi i thread hanno eseguito il metodo, A[0] ha valore diverso da u oppure A[1] ha valore diverso da v.

Potrebbe capitare quanto segue: il primo thread esegue la prima istruzione, memorizzando il valore u nella propria variable temp, il secondo thread esegue le tre istruzioni, assegnando v a A[0] e u a A[1], quindi il primo thread esegue le due istruzioni rimanenti, assegnando il valore u a A[0] e u (che è il valore in temp) a A[1]. Pertanto abbiamo race condition.

Spiegare il TLB.  
  
  
ESAME 2

Assumiamo un array di interi A di dimensione 2.

L’array è condiviso da thread di 4 tipi:

Thread di tipo W0: eseguono l’operazione A[0]=A[0]+7; A[0] = A[0]\*2,

Thread di tipo W1: eseguono l’operazione A[1]=A[1]+7; A[1] = A[1]\*2;

Thread di tipo R: eseguono l’operazione Y = A[0]+A[1], dove Y è una variabile privata.

Thread di tipo E: eseguono l’operazione Y= A[0] ; A[0] = A[1] ; A[1] = Y, dove Y è una variabile privata.

Scrivere il codice per i 4 tipi di thread, usando i semafori con la semantica tradizionale, prevenendo race condition busy waiting e assicurando che non ci siano mai thread in attesa a meno che ciò non sia necessario per prevenire race condition.

Uso le seguenti variabili condivise:

boolean wrk0 = false; // wrk0 is true if and only if one W0 is working on A

boolean wrk1 = false; // wrk1 is true if and only if one W1 is working on A

boolean wrkE = false; // wrkE is true if and only if one WE is working on A

int wrkR = 0; // wrkR is n if and only if n WR are working on A

int waiting0= 0; // waiting0 is n if and only if n W0 are waiting for working on A

int waiting1= 0; // similar

int waitingE= 0; // similar

int waitingR= 0; // similar

Uso i seguenti semafori:

S0 = S1 = SE = SR = 0;

mutex=1;

thread W0:

while(true){

….

wait(mutex);

if(!wrk0 & !wrkE & wrkR==0){ // guard true if and only if this thread can work on A

wrk0=true; // Notice that wrk1 is not checked, since

signal(mutex); // concurrency between W0 and W1 is legal

}

else{

waiting0++;

signal(mutex);

wait(s0); // it the thread cannot work on A, it waits on s0

}

A[0]=A[0]+7: A[0]=A[0]\*2; // this thread can work on A in concurrency with one thread of type W1

wait(mutex);

if(waiting0>0){ // if there are threads W0 waiting for working on A, one of them is waken up

waiting0 - - ;

signal(s0);

}

else{

if(waitingE>0 & !wrk1){ // otherwise, if there are threads WE waiting for working on A, and

waitingR - - ; // no thread W1 is working on A, one thread WE is waken up

wrkR=true;

wrk0=false;

signal(sE);

}

else{

wrk0=false;

while(waitingR>0 & !wrk1){ // otherwise, is if there are threads WR waiting for working on A

waitingR - - ; // and no thread W1 is working on A, all thread WR waiting for working are

workingR++; // waken up.

signal(sR));

}

}

signal(mutex)

….

}

Thread W1: come W0, scambiando i ruoli di 0 e 1.

Thread WR:

while(true){

int Y = ….;

….

wait(mutex);

if(!wrk0 & !wrk1 & !wrkE){ // the guard is true if and only if no thread is modifying A

wrkR + + ;

signal(mutex);

}

else{

waitingR++;

signal(mutex);

wait(wR);

}

Y=A[0]+A[1]; // this instruction can be performed in concurrency by several thread WR

wait(mutex);

wrkR - - ;

if(waiting0>0 & wrkR==0){

waiting0 - - ;

wrk0=true;

signal(s0);

}

if(waiting1>0 & wrkR==0){

waiting1 - - ;

wrk1=true;

signal(s1);

}

if(waitingE>0 & wrkR==0 & !wrk0 & !wrk1){

waitingE - - ;

wrkE=true;

signal(sR);

}

signal(mutex)

….

}

Thread WE:

while(true){

….

wait(mutex);

if(!wrk0 & !wrk1 & !wrkE & wrkR==0){ // guard true if nobody is working on A

wrkE + + ;

signal(mutex);

}

else{

waitingE++;

signal(mutex);

wait(wE);

}

Y=A[0]; A[0]=A[1]; A[1]=Y;

wait(mutex);

if(waitingE>0){

waitingE - - ;

signal(sE);

}

else{

wrkE=false;

if(waiting0>0){

waiting0 - - ;

wrk0=true;

signal(s0);

}

if(waiting1>0){

waiting1 - - ;

wrk1=true;

signal(s1);

}

while(!wrk0 & !wrk1 & waitingR>0){

wrkR++;

waitingR- - ;

signal(semR);

}

}

signal(mutex)

….

}  
  
ESAME 3

Supponiamo di avere un array A di interi condiviso da due thread che eseguono il seguente metodo:

public void m( ){ A[0] = A[1]+10 ; A[1] = A[0]+20; }

Spiegare, formalmente, se sull’array A sono possibili race condition.

Gli effetti che si hanno sull’array eseguendo il metodo m possono essere spiegati con la funzione

f: (Z x Z)—> (Z x Z) (nota: Z è l’insieme dei numeri interi)

definita come segue:

f(u , v) = (v + 10 , v + 30) per ogni u e v appartenenti a Z. (Nota: v+30 coincide con (v+10)+20.)

Pertanto, se i due thread eseguono il metodo non in concorrenza, l’effetto della loro esecuzione è spiegato come segue:

f(f(u , v)) = f(v + 10 , v + 30) = (v + 40 , v + 60).

Di conseguenza, in base alla definizione di race condition, se A[0] = u e A[1] = v sono i valori iniziali dei due interi, abbiamo race condition se dopo che entrambi i thread hanno eseguito il metodo, A[0] ha valore diverso da v+40 oppure A[1] ha valore diverso da v+60.

Potrebbe capitare quanto segue: il primo thread assegna v+10 a A[0], il secondo thread assegna v+10 a A[0], il secondo thread assegna v+30 a A[1], il primo thread assegna v+30 a A[1]. In tal modo, terminata l’esecuzione del metodo da parte di entrambi thread, A[0] ha valore v+10 e A[1] ha valore v+30. In questo caso, abbiamo race condition. Pertanto concludiamo che le race condition sono possibili.

(Nota: questa analisi finali potrebbe essere effettuata sostituendo u e v con due valori interi arbitrari.)|

ESAME 5

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

}