**Si consideri il problema dei produttori e consumatori. Si assuma che il buffer sia un array di 10 interi gestito circolarmente. Si assumano i seguenti vincoli aggiuntivi:**

* **il valore 3 non deve essere presente più di una volta nell’array;**
* **il valore 5 non deve essere presente più di due volte nell’array;**

**I processi che tentano di effettuare operazioni al momento non consentite devono essere messi in attesa. Programmare il sistema sfruttando i semafori con la semantica tradizionale.**

**Una soluzione possibile:**

## Semafori:

empty = 10; //numero di posizioni libere

full = 0; //numero di posizioni occupate semProd = 1; //semaforo produttori

semCons = 1; //semaforo consumatori

sem3 = 1; //semaforo per le occorrenze del valore 3 sem5 = 2; //semaforo per le occorrenze del valore 5

## Variabili

int i = 0; //indice usato dai produttori int j = 0; //indice usato dai consumatori

Produttore:

int item;

while (true) {

…

item = produceItem();

if (item == 3) {wait sem3;} if (item == 5) {wait sem5;} wait (empty);

wait (semProd); buffer[i] = item; i = i + 1 % 10;

signal (semProd); signal (full);

…

Consumatore:

int item;

while (true) {

…

wait (full);

wait (semProd); item = buffer[j]; j = j + 1 % 10;

signal (semCons);

if (item == 3) {signal (sem3);} if (item == 5) {signal (sem5);} signal (empty);

…

}

}

**Si consideri il classico problema dei produttori e consumatori, con il buffer implementato con un array di interi di**

**dimensione 100. Si assuma, per semplicità, che gli interi prodotti e consumati siano tutti strettamente positivi oppure uguali a 0, >= 0.**

**Si considerano le seguenti condizioni aggiuntive:**

* **gli interi pari possono occupare solo le posizioni di indice pari (0, 2, 4, …, 98);**
* **gli interi dispari divisibili per 3 possono occupare solo le posizioni di indice dispari divisibile per 3 (3, 9, 15, 21,**

**…, 99);**

* **gli interi dispari che non sono divisibili per 3 possono occupare solo le posizioni di indice dispari non divisibili per 3 (1, 5, 7, 11, 13, 17, …, 97);**

**Quando un processo tenta di effettuare un’operazione al momento non consentita (per esempio produrre un intero che il buffer non può al momento ospitare), il processo deve essere sospeso. Programmare il sistema sfruttando i**

**semafori con la semantica tradizionale. Si assume l’array inizializzato con interi NEGATIVI.**

**Una possibile soluzione:**

## Semafori:

emptyP = 50; //numero posizioni libere che possono ospitare numeri pari

emty3 = 17; //numero posizioni libere che possono ospitare numeri dispari div. per 3

emptyD = 33; //numero posizioni libere che possono ospitare numeri dispari non div. per 3 mutexP = 1; //m.e. su zone dell’array che ospitano numeri pari

mutex3 = 1; //m..e su zone dell’array che ospitano numeri dispari div. per 3 mutexD = 1; //m.e. su zone dell’array che ospitano numeri dispari non div. per 3

Produttore:

int item;

while (true) {

item = produceItem (); if (item & 2 == 0) {

wait (emptyP); wait (mutexP); buffer[i] = item; i = (i + 2) % 100;

signal (mutexP);

}

if (item % 2 != 0 & item % 3 == 0) { wait (empty3);

wait(mutex3); buffer[j] = item;

if (j == 99) {j = 3} else {j = j + 6;} signal (mutex3);

Consumer:

int item;

while (true) { wait (full);

wait (mutexP); wait (mutex3); wait (mutexD);

while (buffer[h] < 0) {h++;} item = buffer[h];

buffer[h] = -1;

if (item % 2 == 0) {signal (emptyP);}

if (item % 2 != 0 & item % 3 == 0) {signal (empty3);} if (item % 2 != 0 & item % 3 != 0) {signal (emptyD);} signal (emptyD);

signal (empty3); signal (emptyP);

}

}

if (item % 2 != 0 & item % 3!= 0) { wait (emptyD);

wait(mutexD); buffer[k] = item;

while (k % 2 == 0 | k % 3 == 0) { k = (k + 1) % 100;

}

signal (mutexD);

}

signal (full);

**Si consideri il classico problema dei produttori e consumatori, con il buffer implementato con un array di interi di dimensione 100. Si assuma, per semplicità, che gli interi prodotti e consumati siano >= 0.**

**Si considerano le seguenti condizioni aggiuntive:**

* **gli interi pari possono occupare solo le posizioni da 0 a 49;**
* **gli interi dispari possono occupare solo le posizion da 49 a 99;**
* **vi è una nuova categoria di processi, i processi ConsPari, che consumano tutti gli interi pari, se l’array contiene almeno 20 elementi pari;**

**Quando un processo tenta di effettuare un’operazione al momento non consentita (per esempio produrre un intero che il buffer non può al momento ospitare), il processo deve essere messo in attesa. Programmare il sistema sfruttando i semafori con la semantica tradizionale.**

**Una possibile soluzione:**

## Variabili:

attesaProd = 0; //numero di produttori in attesa attesaCons = 0; //numero di consumatori in attesa

attesaCons20 = 0; //numero di consumatori del tipo ConsPari in attesa numPari = 0; //numeri pari presenti nell’array

## Semafori:

semCons = 0; //semaforo per mettere in attesa i consumatori semProd = 0; //semaforo per mettere in attesa i produttori

semCons20 = 0; //semaforo per mettere in attesa i consumatori del tipo ConsPari mutex = 1; //semaforo per garantire m.e. sulle variabili condivise

Produttore:

while (true) {

int item = produceItem (); if (item % 2 == 0) {

wait (mutex) {

if (numPari == 50) { attesaPari++;

signal (mutex); wait (semProd); wait (mutex);

}

buffer[i] = item; i = (i + 1) % 50;

numPari++;

if (attesaCons > 0) { attesaCons--;

signal (semCons);

}

if (numPari == 20 & attesaCons > 0) { attesaCons20--;

signal (semCons20);

}

signal (mutex);

}

}

else {facile!}

}

Consumatore:

Consumer (boolean pari) { while (true) {

if (pari) {

wait (mutex);

while (numPari == 0) { attesaCons++;

signal(mutex); wait (semCons); wait (mutex);

}

int item = buffer[j]; j = (j + 1) % 50;

numPari--;

if (attesaProd > 0) { attesaProd--;

signal (semProd);

}

signal (mutex);

}

Consumatore 20 pari:

Consumer20 () { while (true) {

wait (mutex);

while (numPari < 20) { attesaCons20++;

signal (mutex);

wait (semCons20); wait (mutex);

}

consumaTutto (); j = 0; numPari = 0; int k = 0;

while (attesaProd > 0 & k > 50) { attesaProd--;

signal (semProd); k++;

}

signal (mutex);

}

else {facile!} }

}

**Si implementi il classico problema produttori/consumatori con le seguenti modifiche:**

* **esiste una classe dei consumatori “speciali” che consumano due elementi anziché uno;**
* **se un consumatore speciale va in waiting allora da questo momento in poi al massimo 5 consumatori normali potranno consumare prima che almeno un consumatore speciale venga risvegliato;**

**Una possibile soluzione:**

## Variabili:

F = N; //F = numero di elementi free dell’array

A = 0; //A = numero di elementi available dell’array

bonus = 5; //serve per implementare la priorità ai consumatori speciali

## Semafori:

semProd = semCons = semConsSpec = 0; prodWaiting = consWaiting = consSpecWaiting = 0;

Produttore:

produttore () { wait (mutex); if (F == 0) {

prodWaiting++; signal (mutex); wait (semProd);

}

else {

F = F – 1;

signal (mutex);

}

**produzione classica**; wait (mutex);

if (bonus > 0 & consWaiting > 0) { consWaiting--;

signal (mutex);

}

else {

if (consSpecWaiting > 0 & A == 1) { consSpecWaiting--;

A = 0;

bonus = 5;

signal (semConsSpec);

Consumatore normale:

consumatore () { wait (mutex);

if (A == 0 OR bonus == 0) { consWaiting++;

signal (mutex); wait (consSem); wait (mutex);

}

else {A = A – 1;}

if (consSpecWaiting > 0) {bonus--;} signal (mutex);

**consumazione classica**; wait (mutex);

if (prodWaiting > 0) { prodWaiting--;

signal (semProd);

}

else {F = F + 1;} signal (mutex);

}

}

else {A = A + 1;}

}

signal (mutex);

}

consumatoreSpeciale () { wait (mutex);

if (A <= 1) {

consSpecWaiting++; signal (mutex);

wait (semConsSpec);

}

else {

A = A – 2;

signal (mutex);

}

**consumazione classica di due item**; wait (mutex);

if (prodWaiting > 1) { prodWaiting--; prodWaiting--;

signal (semProd); signal (semprod);

}

else {

if (prodWaiting == 1) { prodWaiting--;

signal (prodWaiting); F = F + 1;

}

else {F = F + 2;}

}

signal (mutex);

}

**Si programmi un sistema in cui un insieme di processi condivide un array A di 3 interi (indici 0, 1 e 2). I valori inziali dell’array sono tutti 0 (A [0] = A [1] = A [2] = 0). Esistono le seguenti classi di processi:**

* **processi scrittori: iterativamente selezionano un indice 0 <= i <= 2 ed un valore interno v ed assegnano tale valore v ad A [i];**
* **processi lettori: iterativamente calcolano A [0] + A [1] + A [2]; Condizioni:**
* **le race condition devono essere impossibili:**
* **quando uno o più lettori stanno lavorando sull’array ed uno scrittore va in waiting perché tenta di accedere all’array (va in waiting perché altrimenti potrebbe verificarsi una race condition), da questo momento in poi al più 10 lettori che tentano di accedere all’array possono accedere prima che almeno uno scrittore in attesa venga risvegliato (questa condizione garantisce che gli scrittori in attesa non rimangano in attesa all’infinito “per colpa” dei lettori);**
* **quando uno o più scrittori stanno lavorando sull’array (ovviamente in posizioni diverse se si tratta di più scrittori) ed un lettore va in waiting perché tenta di accedere all’array (va in waiting perché altrimenti**

**potrebbe verificarsi una race condition), da questo momento in poi al più 4 scrittori che tentano di accedere ad A [0], al più 4 scrittori che tentano di accedere ad A [1] ed al più 4 scrittori che tentano di acceder ad A [2] possono accedere prima che almeno un lettore venga risvegliato (questa condizione garantisce che i lettori non rimangano in attesa all’infinito “per colpa” degli scrittori);**

Una possibile soluzione:

## Variabili:

waitingR; //numero di lettori in waiting workingR; //numero di lettori in working bonusR; //bonus per i lettori

waiting\_i; //numero di scrittori in waiting su A [i] working\_i; //true IF uno scrittore sta lavorando su A [i] bonus\_i; //bonus per gli scrittori su i

## Semafori:

mutex = 1; //semaforo per m.e.

semR = 0; //semaforo per i lettori

sem\_i = 0; //semaforo per gli scrittori su A [i]

Reader:

while (true) { wait (mutex);

while ((working\_0 OR workin\_1 OR working\_2) OR ((waiting\_0 > 0 OR waiting\_1 > 0 OR waiting\_2 > 0) & bonusR == 0)) {

waitingR = waitingR + 1; signal (mutex);

wait (semR); wait (mutex);

}

workingR = workingR + 1;

(\*) bonus\_0 = bonus\_1 = bonus\_2 = 4;

if (waiting\_0>0 OR waiting\_1>0 OR waiting\_2>0) { bonusR = bonusR - 1;

}

signal (mutex);

int Z = A [0] + A [1] + A [2];

wait (mutex);

workingR = workingR - 1; if (waitingR == 0) {

(\*\*) bonusR = 10;

for (int i = 0; i < 3; i = i + 1) { if (waiting\_i > 0) {

waiting\_i = waiting\_i - 1; singal (sem\_i);

}

}

}

singal (mutex);

}

Writer:

while (true) {

int i = …; int v = …; wait (mutex);

while (workingR > 0 OR working\_i OR (waitingR > 0 & bonus\_i == 0)) {

waiting\_i = waiting\_i + 1; signal (mutex);

wait (sem\_i); wait (mutex);

}

working\_i = true; (\*) bonusR = 10; if (waitingR > 0) {

bonus\_i = bonus\_i - 1;

}

signal (mutex); A [i] = v;

wait (mutex); working\_i = false;

if (bonus\_i > 0 & waiting\_i > 0) { waiting\_i = waiting\_i -1;

signal (sem\_i);

}

if (! working\_((i + 1) % 3) & ! working\_((i + 2) % 3) & ! working\_i) {

(\*\*) bonus\_0 = bonus\_1 = bonus\_2 = 4; while (waitingR > 0) {

waitingR = waitingR - 1; signal (semR);

}

}

signal (mutex);

}

ESERCIZIO 6

**Un parcheggio offre 30 posti auto, non tutti uguali:**

* **10 posti di tipo “S”: possono ospitare solo vetture di tipo “S”;**
* **10 posti di tipo “M”: possono ospitare vetture “S” oppure “M”;**
* **10 posti di tipo “L”: possono ospitare vetture “S” oppure “M”;**

**(“S”, “M”, “L”, rappresentano le tre possibili dimensioni vetture/posti auto).**

**Le vetture che non possono accedere al parcheggio vengono bloccate all’ingresso. In questo caso, quando queste vetture verranno sbloccate potranno accedere solamente a parcheggi del tipo corrispondente (vetture “S” in posti “S”, vetture “M” in posti “M”, vetture “L” in posti “L”).**

**Abbiamo le seguenti restrizioni:**

* **una vettura “S” che arriva all’ingresso può accedere ad un posto “M” solo se non ci sono posti liberi “S” e almeno 3 posti “M” sono occupati da vetture “M”;**
* **una vettura “S” che arriva all’ingresso può accedere ad un posto “L” solo se non ci sono posti liberi “S” e la vettura non può accedere a posti “M”;**
* **una vettura “M” che arriva all’ingresso può accedere ad un posto “L” solo se non ci sono posti liberi “M” e le vetture “M” già presenti nel parcheggio sono meno di 15;**

**Programmare l’ingresso e l’uscita delle vetture di ogni tipo usando i semafori con la semantica tradizionale. Una possibile soluzione:**

## Variabili:

busyS = busyM = busyL = 0; //numero di slot occupati, sempre compreso tra 0 e 10 inM = 0; //numero di auto M nel parcheggio (slot M o L)

inMM = 0; //numero di auto M nel slot M wS = wM = wL = 0; //numero di auto in attesa

## Semafori:

mutex = 1; //m.e. sulle variabili condivise semS = semM = semL = 0; //semafori per bloccare le auto

boolean canSS () {return busyS < 10;} boolean canMM () {return busyM < 10;} boolean canLL () {return busyL < 10;}

boolean canSM () {return (busyM < 10 & inMM >= 3);} //condizione busyS >=10 non verificata boolean canSL () {return (busyL < 10 & ! canSM());} //condizione busyS >=10 non verificata boolean canML () {return (busyL < 10 & inMM < 15);} //condizione busyM > 0 non verificata

void inSS () {busyS++;} //auto S entra slot S

void inSM () {busyM++;} //auto S entra slot M

void inSL () {busyL++;} //auto S entra slot L void inMM () {busyM++; inM++; inMM++;} //auto M entra slot M void inML () {busyL++; inM++;} //auto M entra slot L

void inLL () {busyL++;} //auto L entra slot L

void outS () {if (wS > 0) {wS--; signal (semS);} else {busyS--;}} //auto S esce slot S

void outM () {if (wM > 0) {wM--; inM++; inMM++; signal (semM);} else {busyM-;}} //auto S o M esce slot M void outL () {if (wL > 0) {wL--; signal (semL);} else {busyL--;}} //auto S o M o L esce slot L

void carS () {

wait (mutex);

if (canSS) {busyS++; signal (mutex); <park> wait (mutex); outS (); signal (mutex);)} else {

if (canSM) {busyM++; signal (mutex); <park> wait (mutex); outM (); signal (mutex);} else {

if (canSL) {busyL++; signal (mutex); <park> wait (mutex); outL (); signal (mutex);} else {

wS++; signal (mutex); wait (semS); <park> wait (mutex); outS (); signal (mutex);

}

}

}

}

void carM () { wait (mutex);

if (canMM) {busyM++; inM++; inMM++; signal (mutex); <park> wait (mutex); inM--; inMM--; outM(); singal (mutex);}

else {

if (canML) {busyL++; inM++; signal (mutex); <park> wait (mutex); inM--; outL (); signal (mutex);} else {

wM++; signal (mutex); wait (semM); <park> wait (mutex); outM (); signal (mutex);

}

}

}

void carL () {

wait (mutex);

if (canLL) {busyL++; signal (mutex); <park> wait (mutex); outL (); signal (mutex);} else {

wL++; signal (mutex); wait (semL); <park> wait (mutex); outL(); signal (mutex);

}

}

ESERCIZIO 7

**Un distributore automatico è fornito di confezioni di biscotti e cracker. Dispone di 50 cestelli, ognuno dei quali può ospitare una sola confezione.**

**Esistono 4 categorie di processi:**

* **fornitori di biscotti: inseriscono una confezione di biscotti nel distributore;**
* **consumatori di biscotti: acquistano una confezione di biscotti dal distributore;**
* **fornitori di cracker: inseriscono una confezione di cracker nel distributore;**
* **consumatori di cracker: acquistano una confezione di cracker dal distributore;**

**Quando un fornitore di cracker vuole inserire una confezione e ci sono cestelli liberi, non può farlo se sono valide entrambe le seguenti condizioni:**

* **il numero di confezioni di cracker già presenti è >= del numero di confezioni di biscotti già presenti;**
* **il numero di confezioni di cracker già presenti è >= 3;**

**I fornitori che tentano di rifornire prodotti ma non possono farlo devono essere messi in attesa.**

**I consumatori che desiderano acquistare prodotti non presenti, rinunciano (non devono essere messi in attesa). Programmare il sistema usando i semafori con la semantica tradizionale.**

**Una possibile soluzione:**

## Variabili:

free = 50; //numero di cestelli liberi

waitB = 0; //numero di fornitori di biscotti in attesa waitC = 0; //numero di fornitori di cracker in attesa

totB = 0; //numero di confezioni di biscotti nel distributore totC = 0; //numero di confezioni di cracker nel distributore

## Semafori:

semi = 0; //semaforo per i fornitori di biscotti

mutex = 1; //semaforo per la m.e. sulle variabili condivise

fornitore biscotti { wait (mutex); if (free == 50) {

wiatB++;

signal (mutex); wait (semB);

{inserisco i biscotti}

}

else {

free = free – 1; totB = totB + 1;

if (totB == totC + 1 & waitC > 0 & free > 0) { waitC = waitC – 1;

signal (semC); totC = totC + 1; free = free – 1;

}

signal (mutex);

{inserisco i biscotti}

}

}

fornitore cracker { wait (mutex);

if (free == 0 | (totC >= totB & totC >= 3)) { waiC = waitC + 1;

signal (mutex); wait (semC);

{inserisco I cracker}

}

else {

totC = totC + 1; free = free – 1; signal (mutex);

{inserisco I biscotti}

}

}

consumatore biscotti { wait (mutex);

if (totB == 0) {signal (mutex); “non metto in attesa”} else {

totB = totB – 1; if (waitB > 0) {

waitB = waitB – 1; tot = totB + 1;

signal (semB);

}

else {

if (waitC > 0 & (totC < totB && totC <= 3)) { waitC = waitC – 1;

totC = totC + 1; signal (semC);

}

else {free = free + 1;}

}

signal (mutex);

}

}

consumatore cracker { wait (mutex);

if (totC == 0) {signal (mutex); “non metto in attesa”} else {

totC = totC – 1;

if (waitC > 0 & (totC < totB && totC <= 3)) { waitC = waitC – 1;

totC = totC + 1; signal (semC);

}

else {

if (waitB > 0) {

waitB = waitB – 1; totB = totB + 1;

signal (semB);

}

else {free = free + 1;}

}

signal (mutex);

}

}

**Un parcheggio con una disponibilità totale di 50 posti viene usato da vetture bianche e da vetture di altro colore. Quando una vettura bianca vuole entrare, il numero di vetture bianche non può diventare maggiore del numero delle vetture di altro colore. Il parcheggio ha un unico gate, usato dalle vetture per entrare ed uscire. Le vetture che tentano di entrare devono essere messe in coda di attesa. Non possono entrare per la mancanza di posti disponibili, oppure per la violazione della condizione sui colori sopra descritta.**

**Non deve capitare che ci siano vetture in attesa senza ragione. Programmare l’ingresso e l’uscita del parcheggio delle vetture, usando i semafori con la semantica tradizionale.**

**Una possibile soluzione:**

## Variabili:

free = 50; //numero di posti disponibili

waitN = 0; //numero di vetture non bianche in attesa waitB = 0; //numero di vetture bianche in attesa

totB = 0; //numero di vetture bianche nel parcheggio

totN = 0; //numero di vetture non bianche nel parcheggio

## Semafori:

semB = 0; //semaforo per le vetture bianche

semN = 0; //semaforo per le vetture non bianche

mutex = 1; //semaforo per la m.e. sulle variabili condivise

EnteringN:

wait (mutex); if (free > 0) {

free = free – 1; totN = totN + 1;

if (totN = totB + 1 & waitB > 0 & free > 0) { waitB = waitB – 1;

signal (semB);

}

signal (mutex);

} else {

waitN = waitN + 1; signal (mutex);

wait (semN);

EnteringB:

wait (mutex);

if (free == 0 | totB >= totN) { waitB = waitB + 1;

signal (mutex); wait (semB);

}

totB = totB + 1; free = free – 1; signal (mutex);

}

ExitingN:

wait (mutex); totN = totN – 1; if (waitN > 0) {

waitN = waitN – 1; signal (semN);

} else {

if (waitB > 0 & totB < totN) { waitB = waitB – 1;

signal (semB);

} else {free = free + 1} signal (mutex);

}

ExitingB:

wait (mutex); totB = totB – 1;

if (waitB > 0 & totB < totN) { waitB = waitB – 1;

signal (semB);

} else {

if (waitN > 0) {

waitN = waitN – 1; signal (semN);

} else {free = free + 1;}

}

signal (mutex);

1. **Assumiamo un array di 100 interi A inizializzato con [0,0,0,0,0,0] condiviso da thread che appartengono a tre tipi:**
   * **thread di tipo 1: ciclicamente generano un numero random k ed eseguono l’operazione for (int i = 0; i < A.length; i++) {A[i] = A[i] + k + i;}, che deve essere indivisibile sul dato A;**
   * **thread di tipo 2: ciclicamente generano un numero random k ed effettuano l’operazione for for (int i = A.lenght-1; i >= 0; i = i - 2) {A[i] = A[i] + k – i;}, che deve essere indivisibile sul dato A;**
   * **thread di tipo 3: ciclicamente eseguono**

**int x = 0; for (int i = 0; i < A.length; i++) {x = x + A[i]; System.out.println(x);}**

**Usando i semafori con la semantica tradizionale, scrivere il codice dei 3 tipi di thread, rispettando i seguent i vincoli: un thread può essere in waiting su un semaforo solo se ciò è necessario per garantire le indivisibilità delle operazioni dei thread di tipo 1 e 2, oppure se è necessario per evitare race condition su variabili condivise.**

**Inoltre, quando un thread di tipo 1 termina la propria operazione indivisibile su A, se vi sono thread in attesa, viene data priorità a thread di tipo 2, poi a thread di tipo 3, poi a thread di tipo 1.**

**Analogamente, quando un thread di tipo 2 termina la propria operazione indivisibile su A, se vi sono thread in attesa, viene data priorità a thread di tipo 1, poi a thread di tipo 3, poi a thread di tipo 2.**

ESERCIZIO 1

## Variabili:

wrk12: numero di thread di tipo 1 oppure 2 che stanno lavorando. Valore iniziale 0. Valori possibili: 0,1 wrk3: numero di thread di tipo 3 che stanno lavorando. Valore iniziale 0. Valori possibili: 0,1,2,3, …

tw1: numero di thread di tipo 1 in waiting. Valore iniziale 0 tw2: numero di thread di tipo 2 in waiting. Valore iniziale 0 tw3: numero di thread di tipo 3 in waiting. Valore iniziale 0

## Semafori:

mutex: valore iniziale 1. Garantisce m.e. sulle variabili condivise sopra

s1: valore iniziale 0. Serve per mettere in waiting i thread di tipo 1 s2: valore iniziale 0. Serve per mettere in waiting i thread di tipo 2 s3: valore iniziale 0. Serve per mettere in waiting i thread di tipo 3

Thread tipo 1:

while (true) {

//some work having nothing to do with our array wait (mutex);

if (wrk12 > 0 || wrk3 > 0) {tw1++; signal (mutex); wait (s1);} else {wrk12++; signal (mutex);}

k = …;

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

wait (mutex); wrk12--;

if (tw2 > 0) {tw2--; wrk12++; signal (s2);}

else {while (tw3 > 0) {tw3--; wrk3++; signal (s3);}

if (wrk3 == 0 & tw1 > 0) {tw1--; wrk12++; signal (s1);}} signal (mutex);

// some work having nothing to do with our array

}

Thread tipo 2: analogo al thread di tipo 1

Thread tipo 3:

while (true) {

//some work having nothing to do with our array wait (mutex);

if (wrk12 > 0) {tw3++; signal (mutex); wait (s3);} else {wrk3++; signal (mutex);}

int x = 0;

for (int i = 0; I < A.length; i++) {x = x + A[i]; System.out.println (x);}

wait (mutex); wrk3--;

if (wrk3 == 0 & tw1 > 0) {tw1--; wrk12++; signal (s1);}

if (wrk3 == 0 & wrk12 == 0 & tw2 > 0) {tw2--; wrk12++; signal (s2);} signal (mutex);

//some work having nothing to do with our array

**Assumiamo un array di 6 interi A inizializzato con [0, 0, 0, 0, 0, 0] condiviso da thread che appartengono a tre tipi:**

* + **thread di tipo 1: ciclicamente generano un numero random k ed effettuano l’operazione**

**A[0] = A[0] + k; A[1] = A[1] + k; A[2] = A[2] + k, che deve essere indivisibile dul dato A[0, 2]**

* + **thread di tipo 2: ciclicamente generano un numero random k ed effettuano l’operazione**

**A[3] = A[3] + k; A[4] = A[4] + k; A[5] = A[5] + k, che deve essere indivisibile sul dato A[3, 5]**

* + **thread di tipo 3: ciclicamente stampano il valore A[0] + A[1] + A[2] + A[3] + A[4] + A[5].**

**Usando i semafori con la semantica tradizionale, scrivere il codice dei 3 tipi di thread, rispettando il seguente vincolo: un thread può essere in waiting su un semaforo solo se ciò è necessario per garantire le indivisibilità delle operazioni dei thread di tipo 1 e 2.**

**ESERCIZIO 1**

# VARIABILI CONDIVISE:

**tw1: numero di thread di tipo 1 in waiting. Valore iniziale 0. tw2: numero di thread di tipo 2 in waiting. Valore iniziale 0. tw3: numero di thread di tipo 3 in waiting. Valore iniziale 0.**

**twrk1: numero di thread di tipo 1 che stanno lavorando. Valore iniziale 0. Valori possibili 0, 1. twrk2: numero di thread di tipo 2 che stanno lavorando. Valore iniziale 0. Valori possibili 0, 1.**

**twrk3: numero di thread di tipo 3 che stanno lavorando. Valore iniziale 0. Valori possibili 0, 1, 2, 3, …**

# SEMAFORI:

**s1: valore iniziale 0. Serve per mettere in waiting i thread di tipo 1. s2: valore iniziale 0. Serve per mettere in waiting i thread di tipo 2. s3: valore iniziale 0. Serve per mettere in waiting i thread di tipo 3.**

Thread tipo 1:

while (true) {

//some work having nothing to do with our array wait (mutex);

if (twrk1 > 0 ||twrk3 > 0) {tw1++; signal (mutex); wait (s1);} else {twrk1++; signal (mutex);}

k = …;

A[0] = A[0] + k; A[1] = A[1] + k; A[2] = A[2] + k;

wait (mutex); twrk1--;

if (tw1 > 0) {tw1--; twrk1++; signal (s1);}

else {while (tw3 > 0 & twrk2 == 0) {tw3--; twrk3++; signal (s3);}} signal (mutex);

//some work having nothing to do with our array

}

Thread tipo 2: analogo a thread di tipo 1

Thread di tipo 3:

while (true) {

//some work having nothing to do with our array wait (mutex);

if (twrk1 > 0 || twrk2 > 0) {tw3++; signal (mutex); wait (s3);} else {twrk3++; signal (mutex);}

print (A[0]…A[6]);

wait (mutex); twrk3--;

if (twrk3 == 0 & tw1 > 0) {tw1--; twrk1++; signal (s1);} if (twrk3 == 0 & tw2 > 0) {tw2--; twrk2++; signal (s2);} signal (mutex);

//some work having nothing to do with our array

}

1. **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 tre 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;**

ESERCIZIO 1:

# VARIABILI:

waiting1 = waiting2 = waiting3 = 0; //numero di utenti in attesa token = 100; //numero di gettoni disponibili

# SEMAFORI:

s1 = s2 = s3 = 0; //semaforo per mettere in waiting i thread, per ogni tipologia

mutex = 1; //semaforo per garantire m.e. alle variabili condivise

Utenti di tipo 1:

while (true) {

//attività non rilevante

wait (mutex);

if (token > 0) {token--; signal (mutex);} //se c’è almeno un token, lo consumo ed entro else {waiting1++; signal (mutex); wait (s1);} //altrimenti, vado in attesa

//training activity wait (mutex);

if (waiting1 > 0) {waiting1--; signal (s1);} //se c’è un U1 in attesa, lo sveglio

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 ed il mio token 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 I due token disponibili ed il mio token waiting3--;

signal (s3);

} else {token++;} //se non svegio nessuno il mio token diventa disponibile

}

}

signal (mutex);

//attività non rilevante

}

while (true) {

//attività non rilevante

wait (mutex);

if (token > 1) {token = token – 2; signal (mutex);} //se ci sono almeno disponibili, li consumo ed entro else {waiting2++; signal (mutex); wait (s2);} //altrimenti, vado in attesa

//training activity wait (mutex);

if (waiting2 > 0) {waiting2--; signal (s2);} //se c’è un U2 in attesa, lo sveglio

else { //altrimenti

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

token--; 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

}

while (true) {

//attività non rilevante

wait (mutex);

if (token > 2) {token = token – 3; signal (mutex);} //se ci sono almeno 3 token disponibili li uso ed entro else {waiting3++; signal (mutex); wait (s3);} //altrimenti, vado in attesa

//training activity wait (mutex);

if (waiting3 > 0) {waiting3--; singal (s3);} //se c’è un U3 in attesa, lo sveglio

else { //altrimenti

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

while (waiting2 > 0 & token > 1) { //sveglio U2 waiting2--;

token = token – 2; signal (s2);

}

while (waiting1 > 0 & token > 0) { //sveglio U1 waiting1--;

token = token – 1; signal (s1);

}

}

signal (mutex);

//attività non rilevante

**Una palestra può ospitare al massimo 40 clienti. Esistono due tipologie di cliente: regular e premium. Un cliente premium può essere ammesso alla palestra solo se c’è almeno un posto libero.**

**Un cliente regular può essere ammesso alla palestra solo se sono verificate entrambe le seguenti due operazioni:**

* **c’è almeno un posto libero;**
* **la palestra ha < di 20 posti occupati, altrimenti, se la palestra ha già almeno 20 posti occupati, allora, la metà dei posti occupati deve essere occupata da clienti premium;**

**Programmare l’ingresso e l’uscita dalla palestra per entrambi i tipi di processo cliente, nel rispetto di quanto segue:**

* **se un cliente non può entrare, va messo in attesa;**
* **non può capitare che un cliente sia in attesa e il suo eventuale ingresso non violerebbe le regole sopra citate; Non è necessario programmare il comportamento dei clienti all’interno della palestra.**

# VARIABILI:

postiOccupati = 0; //posti occupati nella palestra waitRegular = 0; //numero di clienti regular in attesa waitPremium = 0; //numero di clienti premium in attesa

premiumIn = 0; //numero di clienti premium nella palestra

# SEMAFORI:

mutex = 1; //semaforo per la m.e. sulle variabili condivise semRegular = 0; //semaforo per i clienti regular

semPremium = 0; //semaforo per i clienti premium

entrataRegular () { wait (mutex);

if (postiOccupati == 40 || postiOccupati >= 20 & (premiumIn <= postiOccupati / 2)) {

waitRegular++; signal (mutex); wait (semRegular)

} else {

postiOccupati++; signal (mutex);

}

}

uscitaRegular () { wait (mutex);

if (waitPremium > 0) { waitPremium--; signal (semPremium);

} else if (waitRegular > 0 && (postiOccupati <= 20

|| (premiumIn >= postiOccupati / 2))) { waitPremium--;

signal (semPremium);

} else {

postiOccupati--;

}

signal (mutex);

}

entrataPremium () { wait (mutex);

if (postiOccupati == 40) { waitPremium++; signal (mutex);

wait (semPremium);

} else {

postiOccupati++; premiumIn++; signal (mutex);

}

}

uscitaPremium () { wait (mutex);

if (waitPremium > 0) { postiOccupati--; premiumIn--;

waitPremium--; signal (semPremium);

} else if (waitRegular > 0 && (postiOccupati <= 20

|| (premiumIn >= postiOccupati / 2))) { waitRegular--;

premiumIn--;

signal (semRegular);

} else {

postiOccupati--; premiumIn--;

}

signal (mutex);

**Un distributore di benzina ha N pompe e 1 serbatoio della capacità di M litri. Ogni automobile all’arrivo richiede una**

**specifica quantità di benzina. Il serbatoio è rifornito da una autobotte che lo riempie fino alla capacità massima e solo**

**se nessuna automobile sta facendo rifornimento.**

**Le automobili possono fare benzina solo se c’è una pompa libera, se la quantità di benzina richiesta è disponibile e se**

**l’autobotte non sta riempiendo il serbatoio.**

**Si descriva una soluzione in uno pseudo linguaggio che ottimizzi l’accesso alle risorse usando semafori e processi.**

# VARIABILI:

int disponibile = N; int occupate = 0;

String autobotte = “no”;

# SEMAFORI:

sem distributore = 1; sem serbatoio = 0;

sem auto = 1;

autobotte () {

wait (distributore); autobotte = “si”;

signal (distributore); wait (auto);

riempi (); disponibile = M; signal (auto);

autobotte = “no”;

}

automobile () {

wait (distributore);

if (occupate < N && autobotte.equals (“no”)) { occupate++;

if (occupate == 1) wait (auto); signal (distributore);

richiesta = rand (1 … 20); wait (serbatoio);

if (richiesta > disponibile) { signal (serbatoio);

}

else {

disponibile -= richiesta; signal (serbatoio);

faiBenzina ();

}

wait (distributore); occupate--;

if (occupate == 0) signal (auto); signal (distributore);

}

else {signal (distributore);}

**}**

**Un parcheggio ha 30 posti, due ingressi con sbarra A e B, ed un’uscita.**

**Quando un veicolo si presenta ad uno dei due ingressi, se c’è almeno un posto libero entra, parcheggia ed esce dal parcheggio, altrimenti prenota l’ingresso ed attende di poter entrare.**

**Se ci sono veicoli in attesa ad entrambi gli ingressi, vengono fatti entrare quando altri veicoli escono dal parcheggio, aprendo le due sbarre alternativamente.**

**Quando un veicolo esce dal parcheggio, se ci sono veicoli in attesa ad almeno uno dei due ingressi, ne fa entrare uno.**

# VARIABILI:

int postiLiberi = 30; //totale posti disponibili nel parcheggio int bookA = 0; //veicoli in attesa alla sbarra A

int bookB = 0; //veicoli in attesa alla sbarra B

int turno = 0; //tiene conto di chi è il turno attuale

# SEMAFORI:

mutex = 1; //semaforo per garantire m.e. sulle variabili condivise sbarraA = 0; //semaforo per veicoli in attesa alla sbarra A

sbarraB = 0; //semaforo per i veicoli in attesa alla sbarra B

entraIngressoA () { wait (mutex);

if (postiLiberi > 0) { postiLiberi--;

signal (mutex);

}

else {

bookA++;

signal (mutex); wait (sbarraA);

}

}

entraIngressoB () { wait (mutex);

if (postiLiberi > 0) { postiLiberi--;

signal (mutex);

}

else {

bookB++;

signal (mutex); wait (sbarraB);

esci () {

wait (mutex);

## dovrei fare postiLiberi++ qui?? non dovrei quindi controllare postiLiberi == 0 perch se una macchina esce si ha già un posto libero.

if (postiLiberi == 0) {

if ((turno == 0 || bookB == 0) && bookA > 0) { bookA--;

turno = 1;

signal (sbarraA);

}

else {

if (bookB > 0) { bookB--; turno = 0;

signal (sbarraB);

}

else {postiLiberi++;}

}

else {postiLiberi++;}

}

signal (mutex);

}

}

}

1. **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.**

* 1. **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 B[i] = mB(i). Altrimenti, il thread va in attesa e vi rimane fintantoché tutti gli 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;** Soluzione alternativa:

# VARIABILI:

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 h == B.length allora nessuno può produrre;

int workingPA = 0; //numero di PA in working, cioè numero di PA che stanno producendo; int workingPB = 0; //numero di PB in working, cioè numero di PB che stanno producendo; boolean workingPC = false; //true se e solo se un PC è working, cioè sta’ consumando;

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

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

int waitingPC = 0; //numero di PC 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 delle sezioni critiche;

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

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;

semPC = 0; //serve per mettere in attesa i PC che non possono consumare;

Thread Pa

While(true){

//something having nothing to do with A and B

wait (mutex);

if (workingPC || k == A.length) { waitingPA++;

signal (mutex); wait (semPA); wait (mutexPA);

}

else {

workingPA++; signal (mutex);

wait (mutexPA);

}

int i = k; k++;

signal (mutexPA);

A[i] = mA(i);

wait (mutex); workingPA--;

if (workingPA == 0 && workingPB == 0 && waitingPC > 0) { waitingPC--;

workingPC = true; signal (semPC);

}

signal (mutex);

//something having nothing to do with A and B

}

Thread Pb

While(true){

//something having nothing to do with A and B

wait (mutex);

if (workingPC || h == B.lenght) { waitingPB++;

signal (mutex); wait (semPB); wait (mutexPB);

}

else {

workingPB++; signal (mutex);

wait (mutexPB);

}

int j = h; h++;

signal (mutexPB); B[i] = mB(i);

wait (mutex); workingPB--;

if (workingPA == 0; && workingPB == 0 && waitingPC > 0) { waitingPC--;

workingPC = true; signal (semPC);

}

signal (mutex);

//something having nothing to do with A and B

}

Thread PC:

while (true) {

//something having nothing to do with A and B

wait (mutex);

if (workingPA > 0 || workingPB > 0 || (k == 0 && h == 0)) { waitingPC++;

signal (mutex); wait (semPC);

}

else {

workingPC = true; signal (mutex);

}

int sum = 0;

for (int j = 0; j < A.length; j++) {sum = sum + A[i];} for (int j = 0; j < B.length; j++) {sum = sum + B[i];} System.out.println (sum);

wait (mutex); workingPC = false; k = h = 0;

while (waitingPA > 0 && workingPA < A.length) { waitingPA--;

workingPA++; signal (semPA);

}

while (waitingPB > 0 && workingPB < B.length) { waitingPB--;

workingPB++; signal (semPB);

}

signal (mutex);

//something having nothing to do with A and B

}

**Un vecchio ponte consente di attraversare un fiume nelle direzioni nord -> sud e sud -> nord, con i seguenti vincoli:**

* **per ragioni di peso, in ogni istante al più un veicolo può passare sul ponte;**
* **se un veicolo trova il ponte occupato, attende che si liberi (non è previsto che il veicolo decida di rinunciare ad attraversare il ponte);**
* **dopo che un veicolo ha attraversato il ponte in una direzione, se ci sono veicoli in attesa su entrambi i lati, allora deve passare per primo un veicolo che viaggia nella direzione opposta;**

**Programmare il veicolo che viaggia in senso nord -> sud (programma goingToSouth) ed il veicolo che viaggia in senso sud -> nord (programma goingToNorth).**

# VARIABILI:

int bookToNorth = 0; //numero di veicoli che aspettano per andare a nord int bookToSouth = 0; //numero di veicoli che aspettano per andare a sud

boolean ponteLibero = true; //true se e solo se ponte libero. Nessuna macchina sta passando sul ponte

# SEMAFORI

mutex = 1; //semaforo per m.e. sulle variabili condivise

semToNorth = 0; //semaforo per mettere in attesa veicoli che vogliono andare a nord semToSouth = 0; //semaforo per mettere in attesa veicoli che vogliono andare a sud

enteringNorth () { wait (mutex);

if (ponteLibero) { ponteLibero = false; signal (mutex);

}

else {

bookToSouth++; signal (mutex); wait (semToSouth);

}

}

goingToSouth () { enteringNorth (); crossingToSouth (); exitingSouth ();

}

goingToNorth () { enteringSouth (); crossingToNorth (); exitingNorth ();

}

exitingSouth () { wait (mutex);

if (bookToNorth > 0) { bookToNorth--; signal (semToNorth);

}

else {

if (bookToSouth > 0) { bookToSouth--; signal (semToSouth);

}

else {

ponteLibero = true;

}

}

signal (mutex);

}

**Per smaltire il traffico con maggiore efficienza, supponiamo ora che quando un veicolo è in attesa ad un lato del ponte perché altri veicoli stanno andando verso il lato in cui si trova, sia consentito ad ulteriori 10 veicoli di entrare dal lato opposto.**

# VARIABILI:

int goingToSouth = 0; int goingToNorth = 0; int extraToNorth = 0; int extraToSouth = 0;

boolean ponteLibero = true;

# SEMAFORI:

mutex = 1;

semToNorth = 0;

semToSouth = 0;

void enteringNorth () { wait (mutex);

if (goingToNorth == 0 && bookToNorth == 0) { goingToSouth++;

signal (mutex);

goingToSouth () { enteringNorth (); crossingToSouth (); exitingSouth ();

}

}

else {

if (goingToSouth > 0 && extraToSouth < 10) { extraToSouth++;

goingToSouth++; signal (mutex);

goingToNorth () { enteringSouth (); crossingToNorth (); exitingNorth();

}

}

else {

bookToSouth++; signal (mutex); wait (semToSouth);

}

}

}

void exitingSouth () { wait (mutex); goingToSouth--;

if (goingToSouth == 0) { if (extraToSouth >0) {

extraToSouth = 0;

}

while (bookToNorth > 0) { goingToNorth++; bookToNorth--;

signal (semToNorth);

}

}

signal (mutex);

}

ESERCIZIO 18

1. 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.

* 1. 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 B[i] = mB(i).

Altrimenti, il thread va in attesa e vi rimane fintantoché tutti gli 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;

ESERCIZIO 1:

Soluzione del prof:

# VARIABILI CONDIVISE:

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 h == A.length allora nessuno può produrre;

int workingPA = 0; //numero di PA in working, cioè numero di PA che stanno producendo; int workingPB = 0; //numero di PB in working, cioè numero di PB che stanno producendo; boolean workingPC = false; //true se e solo se un PC è working, cioè sta’ consumando;

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

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

int waitingPC = 0; //numero di PC 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

delle sezioni critiche;

mutexPB = 1; //serve per consentire ai PB di determinare la posizione in cui produrre all’interno

delle sezioni critiche;

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; semPC = 0; // serve per mettere in attesa i PC che non possono consumare;

//something having nothing to do with A and B

wait (mutex); //sulle variabili condivise. Lavoro all’interno di sezioni critiche. if (workingPC || k + workingPA == A.length) { //vado in attesa se un C sta consumando oppure se A è

waitingPA++; pieno/verrà riempito da PA che hanno già controllato di

signal (mutex); poter produrre. 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 & waitingPC > 0) { //controllo se posso svegliare un PC waitingPC--;

workingPC = true; signal (semPC);

}

signal (mutex);

//something having nothing to do with A and B

}

//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 waitingPC++; ci sono elementi da consumare, vado in attesa.

signal (mutex); wait (semPC);

}

else {

workingPC = 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);

}

while (waitingPB > 0 & c < B.length) { waitingPB--;

workingPB--; signal (semPB);

}

workingPC = false; signal (mutex);

//something having nothing to do with A and B

}

**Una palestra può ospitare al massimo 50 utenti, che sono classificati come utenti agonisti oppure utenti delettanti. Programmare ingresso ed uscita dalla palestra dei thread “utenti agonisti” e “utenti dilettanti”. E’ necessario rispettare i seguenti vincoli:   
a. Se i 50 posti sono esauriti, nessun utente può accedere alla palestra.  
b. Anche nel caso i 50 posti non siano esauriti, i dilettanti non possono entrare se nella palestra sono gia’ presenti almeno 15 dilettanti e i dilettanti sono più numerosi degli agonisti.  
c. L’utente che desidera accedere alla palestra ma non può farlo (a causa dei vincoli “a” e “b” deve essere messo in attesa.  
d. Non devono esserci utenti in attesa di accedere alla palestra senza ragione.  
e. I semafori hanno la semantica tradizionale.**

Variabili:  
AW: numero agonisti in waiting  
DW: numero dilettanti in waiting  
A:  numero agonisti in palestra  
D:  numero dilettanti in palestra  
  
Semafori:  
mutex=1, serve per garantire che l'accesso alle variabili condivise avvenga all'interno di sezioni critiche  
SA=0, serve per mettere gli agonisti in attesa  
DA=0, serve per mettere i dilettanti in attesa.  
  
  
  
  
Ingresso agonisti.  
  
  
wait(mutex);  
  
if(A+D == 50){ // palestra piena  
  AW++;  
  signal(mutex);  
  wait(SA);  
}  
else{  
  A++;  
  if(WD>0 & A+D<50 & D==A){ // l'ingresso di un A potrebbe autorizzare un D  
     signal(SD)  
  }  
  signal(mutex)  
}  
  
  
  
  
Uscita agonisti  
  
wait(mutex);  
  
if(AW>0){  // se ci sono A in attesa, uno può entrare  
  AW=AW-1;  
  signal(SA);  
}  
else{  
  if(DW>0 & (D<=A-1 || D <15)){ // altrimenti, se c'è un D in attesa   
                                // valuto se farlo entrare  
     DW=DW-1;  
     D=D+1;  
     A=A-1;  
  }  
  else{ A=A-1;  // altrimenti, non entra nessuno  
  }  
}  
  
signa(mutex);  
  
      
  
Ingresso dilettanti.  
  
wait(mutex);  
  
if(A+D == 50 || (D>A & D>=15)){ // palestra piena o dilettante non autorizzato  
  DW++;  
  signal(mutex);  
  wait(SD);  
}  
else{  
  D++  
  signal(mutex)  
}  
  
  
  
  
Uscita dilettanti  
  
wait(mutex);  
  
if(DW>0 & (D-1<15 || D-1<=A)){  
  DW=DW-1;  
  signal(SD);  
}  
else{  
  if(AW>0){  
     AW=AW-1;  
     A=A+1;  
     D=D-1;  
  }  
  else{ D=D-1;  
  }  
}  
  
signa(mutex);  
  
    