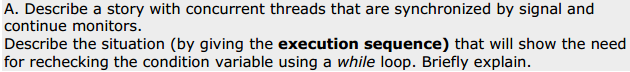
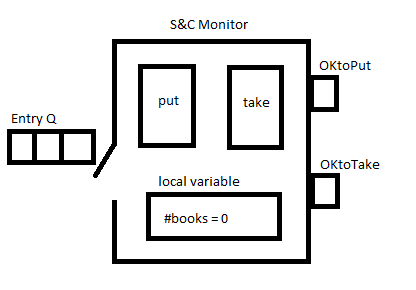
CS715 YUQIAN ZHANG id:23321201

Homework #2



Solution:

Story Introduction: There is a public bookshelf at school library. Students can put and take books from that shelf. The maximum holding number is 5 books. Each student is limited to put or take one book per time.



Two service methods: SC.put() --- SC.take() writing without while loop

Put(){

If (#books==5) {

wait(OKtoPut); }

#books++;

If (OKtoTake.queue!empty) signal(OKtoTake);

}

Take(){

If (#books==0){

wait(OKtoTake);}

#books--;

If (OKtoPut.queue!empty) signal (OKtoPut);

}

Consider following sequence: T1,T2,P1,T3 (T = taking; P = putting)

T1: waits on OKtoTake Q->OKtoTake Q: T1

T2: waits on OKtoTake Q ->OKtoTake Q: T1,T2

P1: #books++(#books =1) -> if statement satisfied, signal OKtoTake Q, release T1,T1 goes to the end of entry Q.

Now the entry Q is T3,T1

T3: #books—(#books=0) then gets out

T1: continues [ gets out of if statement, go directly to #books—without checking #books condition ] #books –(#books = -1)//which supposed to be waiting in the OKtoTake Q

So, revising the service method code with while loop.

Put(){

while (#books==5) {

wait(OKtoPut); }

#books++;

If (OKtoTake.queue!empty) signal(OKtoTake);

}

Take(){

while (#books==0){

wait(OKtoTake);}

#books--;

If (OKtoPut.queue!empty) signal (OKtoPut);

}

Consider the same sequence: T1,T2,P1,T3 (T = taking; P = putting)

T1: waits on OKtoTake Q->OKtoTake Q: T1

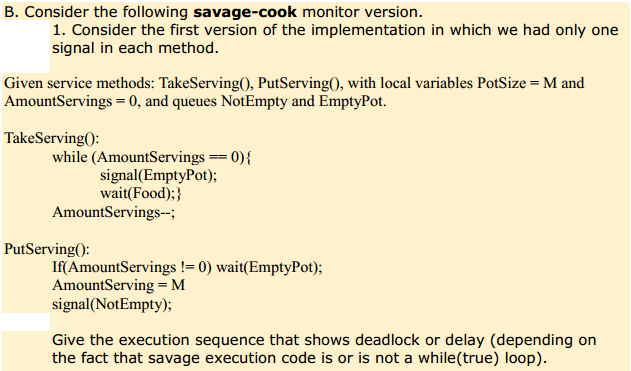
T2: waits on OKtoTake Q ->OKtoTake Q: T1,T2

P1: #books++(#books =1) -> if statement satisfied, signal OKtoTake Q, release T1,T1 goes to the end of entry Q.

Now the entry Q is T3,T1

T3: #books—(#books=0) then gets out

T1: continues from the original while loop. Condition not satisfied, gets blocked again and waits on OKtoTAKE Q. //The while loop helps checking condition variables when signaled thread continues executing from last blocking point



Solution:

S represents Savage, C represents Cook(And there is only 1 cook by assumption)

The following sequence will show deadlock when savage execution code is not a while(true) loop:

S1, S2, S3, C (M = 3, 3 Savages, AmountServings = 0 )

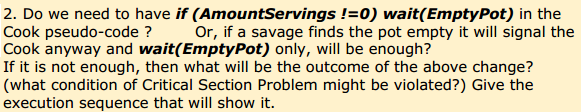
S1: because AmountServings = 0 is true, signal (EmptyPot); wait(Food);

S2: because AmountServings = 0 is true, signal (EmptyPot) ; wait(Food);

S3: because AmountServings = 0 is true, signal (EmptyPot) ; wait(Food);

C: (AmountServings!=0) is not satisfied, so goes to next part AmountServings =3, signal(NotEmpty)

S1: gets the lock and continues AmountServings—(AmountServings =2)//Here comes up deadlock since savage execution code is not in a while loop, after S1 is signaled by cook, it continues properly, but after then nobody is going to signal other savages waiting for food(even if the condition is satisfied AmountServings !=0). Without while (true)loop in savage execution code, savage cannot check by himself even though the condition is satisfied.



Solution:

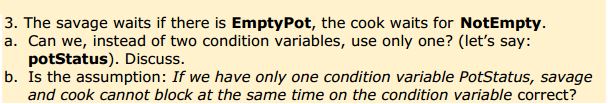
Sequence: S1, S2, S3, C (M = 3, 3 Savages, AmountServings = 0 )

S1: because AmountServings = 0 is true, signal (EmptyPot); wait(Food);

S2: because AmountServings = 0 is true, signal (EmptyPot); wait(Food);

S3: because AmountServings = 0 is true, signal (EmptyPot); wait(Food);

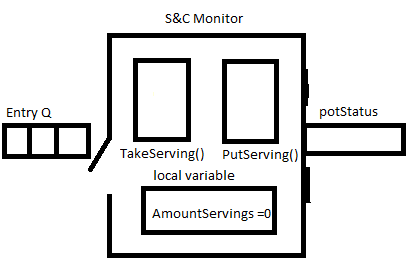
C: wait(EmptyPot) without any check// leads to deadlock even though under this condition(AmountServings = 0), cook should put servings at this time anyway.



Solution:

1. Yes.

The relationship between two condition variables EmptyPot and NotEmpty is contradict. We can combine them in just one queue: potStatus.



Execution code:

TakeServing() {

while(AmountServings == 0) {

wait(PotStatus);}

AmountServings --;

while(PotStatus.Queue!Empty) Signal(PotStatus);

}

PutServings() {

If (AmountServings != 0) {

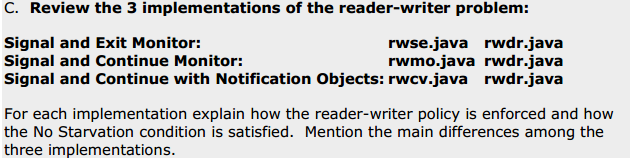
wait(PotStatus);}

AmountServings = M;

while(PotStatus.Queue!Empty) Signal(PotStatus);

}

1. Yes, it is correct. Since the conditions are contradict. Once Cook is in the blocking queue, #Servings != 0 will keep at least one savage active outside and release the whole blocking queue right after it’s done. Same way applies to Savages. If some savages are in the blocking queue, once a cook comes to the head of entry queue, after his serving, he will release the whole blocking queue immediately. They are one or the other.



Solution:

* **Signal and Exit Monitor:**

public synchronized void startRead(int i) {

if (isWriting) wait(OKtoRead);//check if there is a writer writing now

else if (!empty(OKtoWrite)) {// check if there is writer waiting to write, gives priority to a waiting writer

wait(OKtoRead); // new incoming readers cannot starve writers

}

numReaders++;

notify(OKtoRead); // when a writer finishes, all waiting readers start, here is no starvation satisfied. Readers are calling next one until empty

}

public synchronized void endRead(int i) {

numReaders--;

if (numReaders == 0) notify(OKtoWrite);// when there is no readers, signal waiting writers

}

public synchronized void startWrite(int i) {

if (numReaders != 0 || isWriting) wait(OKtoWrite);

isWriting = true;

}

public synchronized void endWrite(int i) {

isWriting = false;

if (!empty(OKtoRead)) notify(OKtoRead);//keep releasing waiting readers

else notify(OKtoWrite); // nor do writers starve readers

}

In this code, readers give priority to writers and with no starvation implemented as follows. When the last reader exits the monitor, it checks first is there any waiting writers, it might also be waiting readers, but readers signaled from writers. Likewise, when a writer gets out of Database, the waiting readers are still waiting writers but it will release all waiting readers. Only when there is no waiting reader, we will release writers. So, it is like readers are released in a group once after a writer arrives.

* **Signal and Continue Monitor:**

private int numReaders = 0;

private int numWriters = 0;

private int numWaitingReaders = 0;

private int numWaitingWriters = 0;

private boolean okToWrite = true;

private long startWaitingReadersTime = 0;

public synchronized void startRead(int i) {

long readerArrivalTime = 0;

if (numWaitingWriters > 0 || numWriters > 0) {

numWaitingReaders++;

readerArrivalTime = age();

while (readerArrivalTime >= startWaitingReadersTime)

try {wait();} catch (InterruptedException e) {}

numWaitingReaders--;

}

numReaders++;

}

public synchronized void endRead(int i) {

numReaders--;

okToWrite = numReaders == 0;

if (okToWrite) notifyAll();//No starvation

}

public synchronized void startWrite(int i) {

if (numReaders > 0 || numWriters > 0) {

numWaitingWriters++;

okToWrite = false;

while (!okToWrite)

try {wait();} catch (InterruptedException e) {}

numWaitingWriters--;

}

okToWrite = false;

numWriters++;

}

public synchronized void endWrite(int i) {

numWriters--; // ASSERT(numWriters==0)

okToWrite = numWaitingReaders == 0;

startWaitingReadersTime = age();

notifyAll();

}

}

In the signal and continue monitor, it’s more fair to both readers and writers in the sense that as soon as writer arrives then no more newly arriving readers are allowed to start reading. However, there are a lot of local variables added to support to satisfy no starvation condition. Observed from code above, all wait()methods are written in the try{} block. This is because once there is a interrupt sent to reader(which is in wait() status) will release and exit. Thus, keep it blocking in the try{} block enforced reader and writer work properly. At the same time, code for this monitor is more complex, for adding reader/writer arrival &waiting time to implement communication among local variables.

* **Signal and Continue with Notification Objects**

class Database extends MyObject {

private int numReaders = 0;

private boolean isWriting = false;

private Vector waitingReaders = new Vector();

private Vector waitingWriters = new Vector();

public Database() { super("rwDB"); }

public synchronized String toString() {

return getName() + " numR=" + numReaders

+ " numRw=" + waitingReaders.size() + " isW="

+ isWriting + " numWw=" + waitingWriters.size();

}

public void startRead(int i) {

Object convey = new Object();

synchronized (convey) {

if (cannotReadNow(convey))

while (true) // wait to be notified, not interrupted

try { convey.wait(); break; }

// notify() after interrupt() race condition ignored

catch (InterruptedException e) { continue; }

}

}

private synchronized boolean cannotReadNow(Object convey) {

boolean status;

System.out.println(" cannotR <: " + this);

if (isWriting || waitingWriters.size() > 0) {

waitingReaders.addElement(convey);

status = true;

} else {

numReaders++;

status = false;

}

System.out.println(" cannotR >: " + this);

return status;

}

public void startWrite(int i) {

Object convey = new Object();

synchronized (convey) {

if (cannotWriteNow(convey))

while (true) // wait to be notified, not interrupted

try { convey.wait(); break; }

catch (InterruptedException e) { continue; }

}

}

private synchronized boolean cannotWriteNow(Object convey) {

boolean status;

System.out.println(" cannotW <: " + this);

if (isWriting || numReaders > 0) {

waitingWriters.addElement(convey);

status = true;

} else {

isWriting = true;

status = false;

}

System.out.println(" cannotW >: " + this);

return status;

}

public synchronized void endRead(int i) {

System.out.println(" endR <: " + this);

numReaders--;

if (numReaders == 0 && waitingWriters.size() > 0) {

synchronized (waitingWriters.elementAt(0)) {

waitingWriters.elementAt(0).notify();

}

waitingWriters.removeElementAt(0);

isWriting = true;

}

System.out.println(" endR >: " + this);

}

public synchronized void endWrite(int i) {

System.out.println(" endW <: " + this);

isWriting = false;

if (waitingReaders.size() > 0) {

while (waitingReaders.size() > 0) {

synchronized (waitingReaders.elementAt(0)) {

waitingReaders.elementAt(0).notify();

}

waitingReaders.removeElementAt(0);

numReaders++;

}

} else if (waitingWriters.size() > 0) {

synchronized (waitingWriters.elementAt(0)) {

waitingWriters.elementAt(0).notify();

}

waitingWriters.removeElementAt(0);

isWriting = true;

}

System.out.println(" endW >: " + this);

}

}

In this code, it is creating notification objects to simulate conditional variable OKtoRead and OktoWrite. By having threads blocking on different objects, when notify, instead of notifying a mixture of threads, it is notify in a more easy way by related variables in a random order. It’s a closer way to get to conditional variable names. By using the notification objects, comparing to Signal and Continue Monitor, it helps us to implement the same idea with less local variable declared.