*Arkadiusz Paryz (COS)*

Thread Programming and

Non-Blocking Java Servers

General Part:

Explain about Thread Programming including:

* When and why we will use Threads in our programs?

First of it’s so that we can utilize the CPU to the full extent for the given task. Let’s say that you’re running a workhorse of a CPU with 16 or 32 cores but only utilizing a few of them, making use of the rest would provide you with better performance. “Performance” is another great reason why we do this. We are more able to run different tasks at once like event handling & GUI applications and since we most of the time are going to deal with multiple clients(multi-threaded) on a server (like just now in class) it will result in much better response time & as above mentioned “Performance” in general.

* Explain about the Race Condition Problem and ways to solve it in Java

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". Ex:

if (x == 5) // The "Check"

{

y = x \* 2; // The "Act"

// If another thread changed x in between "if (x == 5)" and "y = x \* 2" above,

// y will not be equal to 10.

}

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no authentic way of knowing.

To prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

// Obtain lock for x

if (x == 5)

{

y = x \* 2; // Now, nothing can change x until the lock is released.

// Therefore y = 10

}

// release lock for x

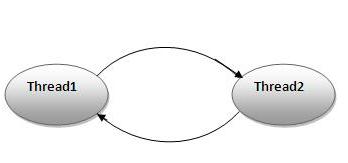
* Explain how we can write reusable non-blocking Java Controls using Threads

We can make other threads run child windows, Ex. JFrame to make the end-user be able to input some data or Swing-control in Java like jButton where the event handling is accomplished with implementing the observer pattern by using events and event listeners.

* Explain about deadlocks, how to detect them and ways to solve the Deadlock Problem

The example beneath shows an example of the deadlock problem that can occur while multi-threading in Java.

It happens when a thread is waiting for an object lock, that is already acquired by another thread. They therefore enter a state where one thread is waiting for its lock to release and vice versa, essentially creating a deadlock.



Preventing this can be obvious at some points when working with Threads or very subtle and sneaky. One of the ways to stay clear of deadlocks is self-explanatory: Only acquire one lock at a time. Unfortunately, that is not always possible. So, the other way is to Document every step in the program, and make sure it’s well documented so you know what’s happening at every step.

An example of a sneaky deadlock could be this transferMoney threading program:

public void transferMoney(Account fromAccount,

Account toAccount,

DollarAmount amountToTransfer) {

synchronized (fromAccount) {

synchronized (toAccount) {

if (fromAccount.hasSufficientBalance(amountToTransfer) {

fromAccount.debit(amountToTransfer);

toAccount.credit(amountToTransfer);

}

}

}

}

Even if all methods that operate on two or more accounts use the same ordering, listing 3 contains the seeds of the same deadlock problem as Listings 1 and 2, but in an even subtler way. Consider what happens when thread A executes:

transferMoney(accountOne, accountTwo, amount);

(Continuation on next page)

While at the same time, thread B executes:

transferMoney(accountTwo, accountOne, anotherAmount);

Again, the two threads try to acquire the same two locks, but in different orders; the deadlock risk still looms, but in a much less obvious form.

A crucial step, if you want to be completely sure, AGAIN with the other methods above mentioned with acquiring one lock at a time, and documenting. Make an ordering to acquire locks in a fixed sequence.

Like this (using the above-mentioned example to create an ordering):

public void transferMoney(Account fromAccount,

Account toAccount,

DollarAmount amountToTransfer) {

Account firstLock, secondLock;

if (fromAccount.accountNumber() == toAccount.accountNumber())

throw new Exception("Cannot transfer from account to itself");

else if (fromAccount.accountNumber() < toAccount.accountNumber()) {

firstLock = fromAccount;

secondLock = toAccount;

}

else {

firstLock = toAccount;

secondLock = fromAccount;

}

synchronized (firstLock) {

synchronized (secondLock) {

if (fromAccount.hasSufficientBalance(amountToTransfer) {

fromAccount.debit(amountToTransfer);

toAccount.credit(amountToTransfer);

}

}

}

}

Now the order in which the accounts are specified in the call to transferMoney() doesn't matter; the locks are always acquired in the same order.