CST8221 – Java Application Programming
Hybrid Activity #9
Thread Deadlocks

Terminology

A **deadlock** is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. Sometimes it is called *"deadly embrace."*

This situation may be likened to two loving siblings who want to make a bike tour of a city, but only have one bicycle and one helmet between them. If one person takes the bicycle and the the other takes the helmet, a deadlock occurs when the person with the bicycle needs the helmet and the person with the helmet needs the bicycle to make the tour without violating the city bylaw. Both needs cannot be satisfied, and here is the deadlock (and a family feud).

Starvation is a similar process to *deadlock*. Starvation occurs when a process is prevented from accessing a resource due to priority scheduling mechanism.

The Nature of Things

In computer science, ***deadlock*** refers to a specific condition when two or more threads or processes are each waiting for each other to release a resource. Deadlock is a common problem in multithreading where many threads share a specific type of mutually exclusive resource known as a software lock or soft lock. As you are already aware Java uses the soft lock mechanism for thread synchronization. Deadlocks are particularly troubling because there is no general solution to avoid soft deadlocks.

The following could create a deadlock situation (see references for more details):

- **Mutual exclusion**. Only a single process can hold a resource or modify shared information at one time.
- **Circular waiting**. A thread A can wait for a thread B, which in turn waits on C, which in turn waits on A. They form a loop.
- **Lack of preemption**. Once a resource has been granted to a thread it cannot be taken away.

The Java technology neither detects nor do attempts to avoid prevent deadlocks. The deadlocks are a design problem, not a language problem (similarly, memory leaks are not a language problem in C/C++). It is a programmer's responsibility to ensure that a deadlock will not happen. The design methods that can be used to prevent deadlock are (see references for more details):

- **Avoidance**. Do not enter a state where a deadlock can occur.
- **Detection and recovery**. If it can be detected a state is a deadlock state, than do not go there.
- **Prevention**. Prevent deadlock by removing one of the conditions mentioned above.

To stay out of trouble you can follow the following simple design principle: If you have multiple objects that you want to access using synchronization, first make a general decision about the order in which you will obtain these locks (serialization rule). Then adhere strictly to that order throughout the program. The locks must be released in the reverse order in which the have been obtained.

In conclusion, follow the simple rule of life: "Thread safely."

References

Textbook 1 – Chapter 23.
"Core Java – Volume I – Fundamentals" by G.S Horstman and G. Cornell, Prentice Hall
Links:
http://en.wikipedia.org/wiki/Deadlock

Code Examples

Four code examples are provided for this hybrid activity. You will find the examples in ***CST8221_HA10_code_examples.zip***. It consists of four folders containing the corresponding examples.

The first example, ***UnsynchBankTest***, demonstrates how data shared by multiple threads can be easily corrupted.

The second example, ***SynchBankTest2***, demonstrates how the data corruption can be prevented by using the "classic" (before Java 5) synchronization mechanism.

The third example, ***SynchBankTest***, demonstrates how the data corruption can be prevented by using the advanced (after Java 5) synchronization mechanism provided by the **java.util.concurrent** package.

The third example, ***DeadlockInterrupted***, demonstrates how to use thread interruption in order to resolve deadlocks. Pay attention to the role of the **java.util.Timer** class in this example. It also demonstrates "daemon" threads.

Exercises

Exercises 1 – Play with the examples

Run the code examples. Compare the code of the first three examples. It will give you an understanding of how to prevent data corruption of shared data.

Exercises 2 – Create a deadlock

Make the following changes to the ***SynchBankTest2.java*** in the **SynchBankTest2** example:

Replace the line

```
TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
```

with the following line

```
TransferRunnable r = new TransferRunnable(b, i, 2*INITIAL_BALANCE);
```

Next, replace the line

```
public static final int NACCOUNTS = 100;
```

with the following line

```
public static final int NACCOUNTS = 10;
```

Compile and run the program. The program will hang. Congratulations! You have just created a deadlock.
Press **Ctrl-Break** and it t does not work try Ctrl +\ or Ctrl-\. You will get a thread dump that lists all threads. Each thread has a stack trace telling you where it is currently blocked. Press **Ctrl-C** to terminate the program.

In the original program a dreadlock cannot occur because each transfer amount is for, at most, $1000. There are 100 accounts and the total of all of them is $100000. This means that at least one of the accounts must have more than $1000 at any time. The thread moving money out of that account can therefore proceed. But with the changes you have made asking the threads to remove more than the $1000 limit you have created a deadlock condition.

Exercises 3 – Create another deadlock

Make the following changes to the **SynchBankTest** code:

In ***SynchBankTest.java*** replace the line

```
public static final int NACCOUNTS = 100;
```

with the following line

```
public static final int NACCOUNTS = 10;
```

In ***TransferRunnable.java*** replace the line

```
bank.transfer(fromAccount, toAccount, amount);
```

with the following line

```
bank.transfer(toAccount,fromAccount, amount);
```

Compile and run the program. The program will hang. Congratulations! You did it again!
Press **Ctrl-Break** and it t does not work try Ctrl +\ or Ctrl-\. You will get a thread dump that lists all threads. Each thread has a stack trace telling you where it is currently blocked. Press **Ctrl-C** to terminate the program.

The reason for the deadlock now is that you have made the *i'th* thread now responsible to for putting money into the *i'th* account, rather than for taking it out of the *i'th* account. In this case, there is a great chance that all threads will "gang up" on one account, each trying to remove more money from it than it contains.

As you see, it is very easy to create a deadlock; it is not so easy to avoid it because sometime is difficult to apprehend the problem in advance.

Questions

Q1. Can multiple threads which are not sharing common resources create a deadlock?
Q2. Can a deadlock be prevented by using the Java *synchronized* primitive?

Submission

No submission is required for this activity.

Marks

No marks are allocated for this activity, but remember that understanding how the examples work is crucial for understanding how to avoid problems with threads.

And do not forget that it is very easy to create a deadlock:

" When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone." a statute passed by the Kansas Legislature