Introduction to Game Programming

Threads

CSC 329

Ubbo Visser

Threads

- What is a Thread?
- Creating and Running Threads in C#
  - Extending the Thread class
  - Implementing the Runnable Interface
  - Using Anonymous Inner Classes
  - The join() and sleep() Methods
- Synchronization
  - How to Synchronize
  - What to Synchronize
  - What not to Synchronize
  - Avoiding Deadlock

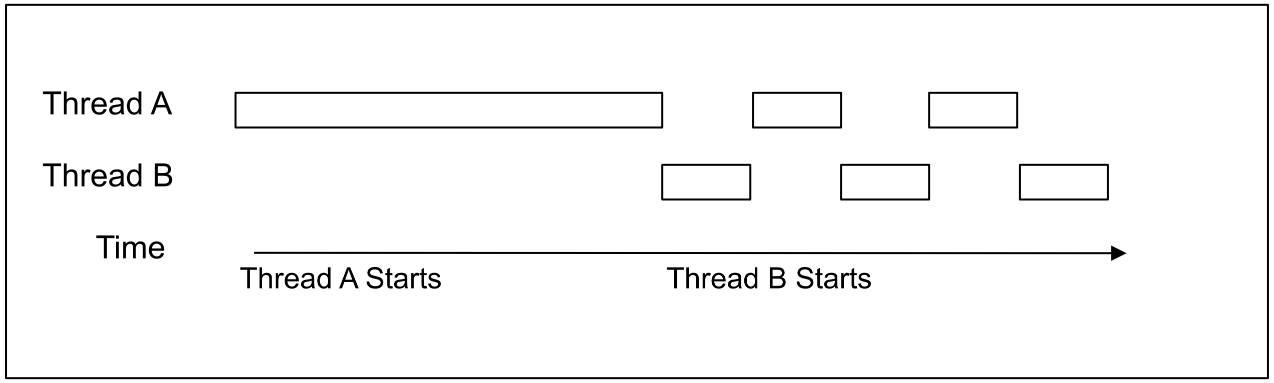
Java Threads (cont.)

- Using Wait() and Notify()
- The Event Model
- When to Use Threads
- When not to Use Threads
- Thread Pools

What is a Thread?

- A Thread is used to allow more than one task to be performed at one time.

- This is achieved by Time Slicing where one thread is executed for a short time, then pre-empted by another thread.



- Threads may run simultaneously on a machine with more than one processor.

Creating and Running Threads in Java

- There are three basic ways to use Threads in Java
  - Extend the Thread Class
  - Implement the Runnable Interface
  - Use anonymous inner classes

Extending the Thread Class

Extend the thread class and override the run() method

```
public class MyThread extends Thread
{
   public static void run()
   {
      System.out.println(“Do something cool here.”);
   }
}
```

Then create and start the thread:

```
Thread myThread = new MyThread();
myThread.start();
```

Implementing the Runnable Interface

Any object that implements the Runnable Interface can be passed as a parameter to the constructor of a Thread object.

```
public class MyClass extends SomeOtherClass implements
Runnable{
   public MyClass(){
      Thread thread = new Thread(this);
      thread.start();
   }

   public void run(){
      System.out.println(“Do something cool here.”);
   }
}
```

The MyClass class implements Runnable, passes itself into a new thread, then starts that thread which executes MyClass.run() .

Using Anonymous Inner Classes

An anonymous inner class can be used to start a Thread when inheriting the Thread class or implementing the Runnable interface is not desirable.

```
new Thread() {
   public void run() {
      System.out.println("Do something cool here.");
   }
}.start();
```

This piece of code creates an instance of a nameless class that inherits the Thread class and overrides the run() method.
This technique should be used carefully because it can easily become hard to read.

The join() and sleep() Methods

- Thread.join();
  If you are in one Thread and you want to wait for another Thread to finish then call the other Thread object's join() method. The current Thread will remain inactive until the outside Thread finishes its run() method.

- Thread.sleep(int);
  The sleep(int) method causes a Thread to be inactive for the specified number of milliseconds during which it will take up no clock cycles.

The join() and sleep() Methods

- Thread.join();
  - If you are in one Thread and you want to wait for another Thread to finish call the other Thread object’s join() method.
  - The current Thread will remain inactive until the outside Thread finishes its run() method.

- Thread.sleep(int);
  - The sleep(int) method causes a Thread to be inactive for the specified number of milliseconds during which it will take up no clock cycles.

Synchronization

- Data corruption
  - Multiple threads are accessing and/or changing the same object.
  - One thread may be using data from an object when it gets pre-empted by another thread which changes the value of the data. When control returns to the previous thread the data is no longer valid.
- Synchronization
  - ... is the coordination of multiple threads that must access shared data.

How to Synchronize

- **`synchronized`**
  - The keyword *synchronized* is used to denote that a method or block of code can only be used by one thread at a time.
  - Threads acquire a lock on the object associated with the code. Only one thread can have a lock on an object at a time.
  - Locks are released when the associated code finishes or an exception is thrown.

How to Synchronize (2)

- The **`synchronized`** keyword is used in the two following ways in practice:

```
// in a method def. associating it with “this” object
public synchronized void DoSomethingCrazy() {
   int x = 3657;
}
```

```
// in a block of code assoc. with a specified object
synchronized(MyObject) {
   System.out.println(“Something to print out.”)
}
```

Dos and don’ts: Synchronization

- What to synchronize
  - Synchronize a piece of code any time two or more threads will access the object or field.

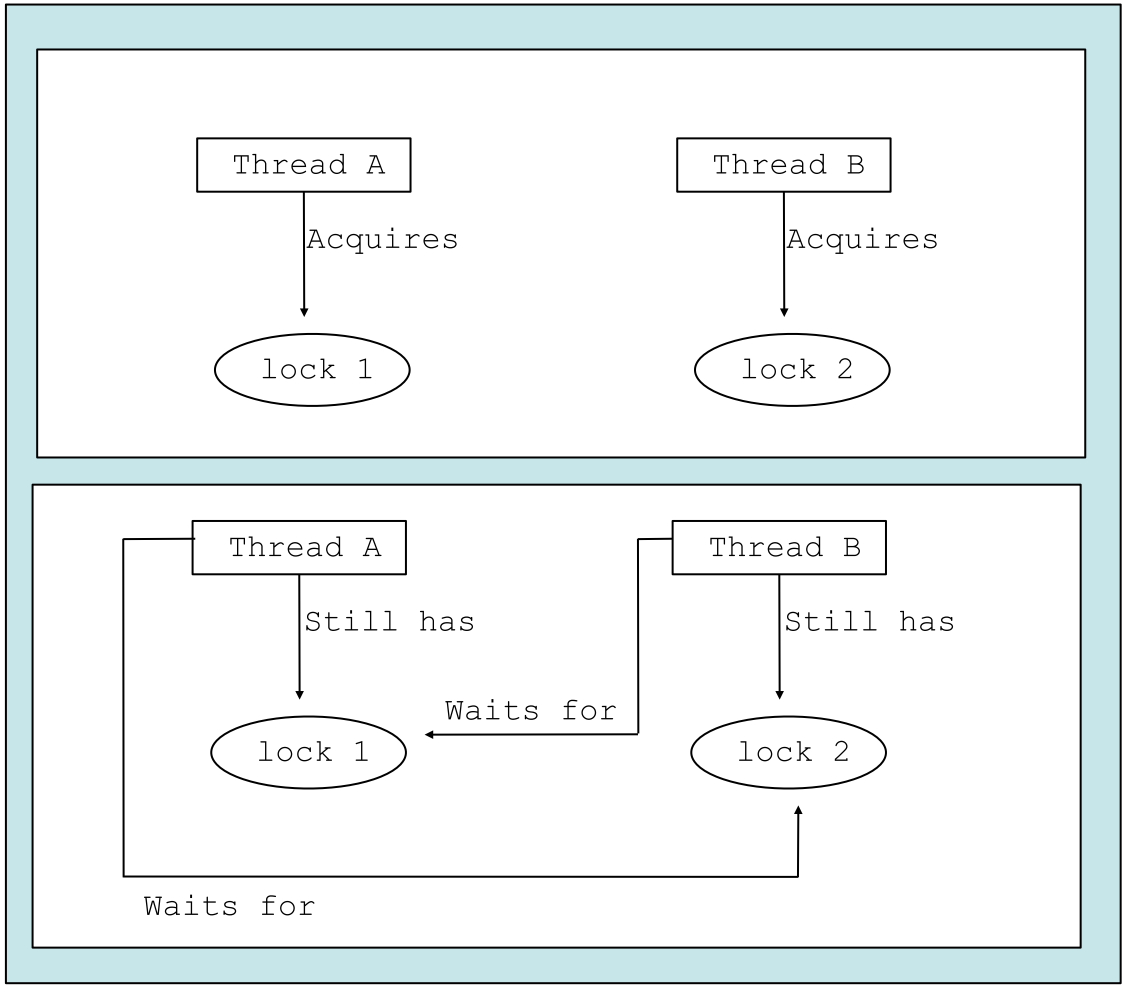
- What not to Synchronize
  - Do not *oversynchronize*, using synchronization too much causes delays, and can lead to further problems.
  - Only synchronize when you need to prevent threads from accessing the same data at the same time.

Avoiding Deadlock

Deadlock is the result of two threads that stall because they are waiting on each other to do something. For example:

Fig 1.
- Thread A acquires lock 1
- Thread B acquires lock 2

Fig 2.
- Thread B waits for lock 1 to be released
- Thread A waits for lock 2 to be released



Both threads are now waiting for the other to finish so neither will continue.

Using *wait()* and *notify()*

- The *wait()* and *notify()* methods provide a means for putting one thread to sleep until it is “woken up” by an outside source.

- *wait()* is a method of the java.lang.Object class. It is called in a synchronized block of code by an executing thread and causes the threads lock on the object to be released and the thread to be put to sleep.

- *notify()* is also a method of java.lang.Object. It notifies one thread that is waiting on the same lock. If several threads are waiting on the same lock one of them is notified randomly.

- There is also a *notifyAll()* method that notifies all waiting threads.

- The *wait()* method can also be given a maximum time in milliseconds to wait for however when control returns to the waiting thread there is no means to determine if it returned because of time or because it was notified.

The Java Event Model

- When your program operates in a graphical environment it can be accessed by at least two threads even if you are not using threads explicitly.

- The two threads are the *main thread* that runs your program and the *AWT event dispatch thread* which handles user input in order to allow event driven program design.

- Because of this you should always keep synchronization in mind even if you are not creating and using threads of your own.

When to Use Threads

- Game Design
  - Threads are useful to prevent lengthy operations from hindering the playing experience.

- Other examples of smart thread use include:
  - Loading files from the disk
  - Network communication, such as sending high scores to a server
  - Massive calculations, such as terrain generation

When Not to Use Threads

- Waste of resources
  - Running too many threads at once can drain the system
- Problems occur
  - An enemy could move in the middle of a draw operation, temporarily showing the enemy in two different places at once.
  - The time slices of each thread could be unbalanced, leading to jerky or inconsistent movement.
  - Synchronized code could lead to unneeded delays.

Thread Pools

- Thread Pool
  - A group of threads designed to execute arbitrary tasks, e.g.
    - Limit the number of threads used for simultaneous network or I/O connections
    - Control the maximum number of threads on the system for processor-intensive tasks.
- Example

```
ThreadPool myThreadPool = new ThreadPool(8);
myThreadPool.runTask(new Runnable() {
    public void run() {
        System.out.println("Do something cool here.");
    }
});
myThreadPool.join();
```

Summary

- Basics of how threads work and how to work with them.
- Now you can make sure your games don't have thread-synchronization problems or cause deadlock.
- We have also learned that you can't avoid dealing with threads in Java games because all graphical Java applications have at least two threads.
- Finally, we have discussed a generic ThreadPool class that can be useful in many situations.