Process Synchronisation

Java Synchronisation (1)

```
public class Server {
   Server () {}
   synchronized public void method1() { /* do something */ }
   public void method2 () { /* do something else */ }
}
```

```
public class Server {
   Server () {}
   synchronized public void method1() { /* do something */ }
   synchronized public void method2() { /* do something else */ }
}
```

Java Synchronisation (2)

```
public class Server {
   Object o = new Object();
   Server () {}
   public void method1() { synchronized (o) { /* do something */ } }
   synchronized public void method2() { /* do something else */ }
}
```

```
public class Server {
   Server () {}
   public void method1() { synchronized (this) {/*do something*/}  }
   synchronized public void method2() { /* do something else */ }
}
```

Java Synchronisation (3)

- wait(), notify(), notifyAll()
- Condition synchronisation?
- Condition variables
- Thread stop() ?
- Semaphore & lock idioms
- Thread pools

For contemplation (1)

- The wait() statement in all Java program examples in this chapter is part of a while loop. Explain why you would always need to use a while statement when using wait() and why you would never use an if statement.
- Under which circumstances can a semaphore be used to solve the critical section problem. Clearly demonstrate that in these circumstances the semaphore solution satisfies the conditions for a solution to the critical section problem. Provide code that shows how the semaphore is used to protect the critical section.

For contemplation (2)

- The **Singleton** design pattern ensures that only one instance of an object is created. For example, assume we have a class called Singleton and we only wish to allow one instance of it. Rather than creating a Singleton object using its constructor, we instead declare the constructor as private and provide a public static method—such as getInstance() —for object creation:

  Singleton sole = Singleton.getInstance();

- The figure provides one strategy for implementing the Singleton pattern. The idea behind this approach is to use **lazy initialization**, whereby we create an instance of the object onlywhen it is needed—that is, when getInstance() is first called. However, the figure suffers from a race condition. Identify the race condition.
- The following figure shows an alternative strategy that addresses the race condition by using the **double-checked locking idiom**. Using this strategy, we first check whether instance is null. If it is, we next obtain the lock for the Singleton class and then double-check whether instance is still null before creating the object. Does this strategy result in any race conditions? If so, identify and fix them. Otherwise, illustrate why this code example is thread safe.

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton { }

   public static Singleton getInstance() {
     if (instance == null)
       instance = new Singleton();

     return instance;
   }
}
```

Figure 6.44 First attempt at Singleton design pattern.

```
public class Singleton {
   private static Singleton instance = null;

   private Singleton { }

   public static Singleton getInstance() {
     if (instance == null) {
       synchronized(Singleton.class) {
         if (instance == null)
           instance = new Singleton();
       }
     }

     return instance;
   }
}
```

Figure 6.45 Singleton design pattern using double-checked locking.

For contemplation (3)

- Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and—once finished—will return them. As an example, many commercial software packages provide a given number of licenses, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned. The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:
  - #define MAX RESOURCES 5
  - int available resources = MAX RESOURCES;
- When a process wishes to obtain a number of resources, it invokes the decrease count() function:
  - /* decrease available resources by count resources */
  - /* return 0 if sufficient resources available, */
  - /* otherwise return -1 */
  - int decrease count(int count) {
  -   if (available resources < count) return -1;
  -   else { available resources -= count; return 0; }
  - }
- When a process wants to return a number of resources, it calls the decrease count() function:
  - /* increase available resources by count */
  - int increase count(int count) {
  -   available resources += count; return 0;
  - }
- The preceding program segment produces a race condition. Do the following:
  - Identify the data involved in the race condition.
  - Identify the location (or locations) in the code where the race condition occurs.
  - Using Java synchronization, fix the race condition. Also modify decreaseCount() so that a thread blocks if there aren't sufficient resources available.

Deadlocks

For contemplation (1)

- Consider the deadlock situation that could occur in the dining philosophers problem when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.
- Java's locking mechanism (the synchronized statement) is considered reentrant. That is, if a thread acquires the lock for an object (by invoking a synchronized method or block), it can enter other synchronized methods or blocks for the same object. Explain how deadlock would be possible if Java's locking mechanism were not reentrant.

For contemplation (2)

- Consider the traffic deadlock depicted in the figure.
  - Show that the four necessary conditions for deadlock indeed hold in this example.
  - State a simple rule for avoiding deadlocks in this system.