Chapter 10

Multitasking

10.1 Multithreading in Java

Multiprocessing is a very old idea in operating systems. Multiprocessing involves a lot of overhead. Threads are a lightweight implementation of the same idea. All the threads of a process share the same address space. Multithreading capability involves the following:

creating threads: A thread should be able to start a new thread. The new thread starts an independent locus of control.

data sharing: In most applicatins, threads don't execute in isolation. A group of threads cooperate – share data – to solve a problem. Without proper precautions, data sharing can result in inconsitent object states. This is called a *race condition*. Meaningful protocols are needed for consistent sharing of data between threads.

event communication: Minimal implementations of data sharing can result in loss of efficiency through *busy-wait* loops. So language support is needed for signaling events.

10.2 Creating threads

When a Java program executes, a single main thread is created first. Additional threads can be created in two ways:

1. by implementing the `Runnable` interface.
2. by extending the `Thread` class

The following code shows thread creation.

```
public class ExampleThread extends Thread {
  private static int threadCount = 0;
  private String say;
```

```
  public ExampleThread(String s) {  // constructor
    say = s;
    System.out.println("Making " + threadCount++);
  }
  public void run() { //  provides the actual behavior
    try {
      while(true) {
        System.out.println(say);
        sleep(1000);
      }
    } catch (Exception e) {
     System.out.println(e);
    }
  }
  public static void main(String[] args) {
      ExampleThread e1 = new ExampleThread("BLAH");
      ExampleThread e2 = new ExampleThread("blah");
      e1.start(); // initialize and call "run"
      e2.start();
      System.out.println("All Threads Started");
  }
}
```

Threads have an associate priority. The priorities can be changed.

10.3 Synchronization

When two or more threads share a common data structure, it is essential to ensure that only one thread has access to the object at a time. Otherwise inconsistent updates of the data can result. (This condition is called *racing*.)
Consider the example of a bank account. The following class can represent the bank account.

```
class Account {
   private int balance = 0;

   public int getBalance() {
      return balance;
   }

   public void deposit(int a) {
      balance += a;  // need not be an atomic execution
```

```
   }

   public void withdraw(int a) {
      balance -= a;  // need not be an atomic execution
   }
}
```

What happens in case of simultaneous deposit and withdrawal by different threads on the same account? Both threads may retrieve the same `balance`, update it, and write it back. In this case one of the operations will be lost. To ensure that only one thread accesses the object, both methods should be declared `synchronized` as shown below.

```
public synchronized void deposit(int a) {  /* operations */ }
public synchronized void withdraw(int a) {  /* operations */ }
```

Each object in Java has its own associated monitor. A monitor is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. To ensure exclusive access or the lock to the associated object (`this`), a method should declare itself `synchronized`. In this case, the caller need not bother about synchronization. The runtime makes sure that only one of the synchronized method possesses the object at a time. This ensures mutual exclusion. Classes which ensure synchronized access to their objects are said to be *thread safe*.
Suppose we are using a class which is not thread safe. Suppose the class `Account` was written without multithreading in mind. (None of its methods is synchronized.) Can we still use it in a multithreaded context (without changing it)? For this the called should use `synchronized` on the object to ensure mutual exclusion as shown below.

```
Account account;
synchronized (account) {
         /* use account */
}
```

As mentioned before Java uses monitors to implement mutual exclusion. When a thread acquires the lock, it is said to have **enter**ed the monitor. Other threads which want to enter the monitor are **suspend**ed till the first thread **exit**s the monitor.
Careless use of shared resources can lead to deadlocks or starvation.

10.4 Event communication

Consider a producer-consumer interaction. In this case, there are two threads. The Producer thread produces data. The Consumer thread uses the data. How do the Producer and Consumer communicate? Through a shared data object, say a queue. The producer puts the data it produces into the queue and the consumer reads the data from the queue. Since the resource is shared, access to it should be synchronized.

Now consider the following situation. The Consumer enters a monitor and does not find any data in the queue. (This can happen if the Producer is slow.) Can it can poll the queue indefinitely? No. (Why?) It issues a `wait()` which suspends the process. When the Producer produces some data, it puts it into the queue. The Producer needs some mechanism for informing the waiting Consumer about the arrival of data. The producer uses the call `notify()` to wake up the Consumer. In case there are multiple processes waiting, the Producer can use `notifyAll()` to wake up all threads. The following code shows an example of event communication.

```
class Store {
    private int a;
    boolean stored = false;

    synchronized void put(int i) {
        try {
           if (stored) wait();
        } catch (Exception e) {
            System.out.println(e);
        }
        a = i;
        stored = true;
        notify();
    }

    synchronized int get() {
        try {
           if (!stored) wait();
        } catch (Exception e) {
           System.out.println(e);
        }
        stored = false;
        notify();
        return a;
    }
}

class Producer extends Thread {
    private Store s;

    Producer (Store s) {
       this.s = s;
    }
```

```
    public void run() {
        for (int i=0; i < 10; i++) {
           s.put(i);
           try {
              sleep((int) (100*Math.random()));
           } catch (Exception e) {
              System.out.println();
           }
        }
    }
}

class Consumer extends Thread {
   Store s;

   Consumer(Store s) {
      this.s = s;
   }

   public void run() {
      try {
         for (int i=0; i < 10; i++) {
             int x = s.get();
             System.out.println(">>> " + x);
             sleep(1000);
         }
      } catch (Exception e) {
          System.out.println(e);
      }
   }

   public static void main(String[] args) {
       Store s = new Store();
       Producer p = new Producer(s);
       Consumer c = new Consumer(s);

       p.start();
       c.start();
   }
}
```

10.5 Deadlocks

Deadlocks occur when there is a circular dependency on the use of shared objects. Suppose there are two shared resources, A and B, between two threads P and Q. Suppose P locks these objects in the order A and B and Q in the order B and A. Can this succeed? By the time P tries to reserve B, Q might have reserved it and now both processes wait indefinitely for an object locked by the other. This is called a *deadlock*. Deadlocks are difficult to detect and debug because they might be scheduling dependent.