4.17 Modify the socket-based date server (Figure 3.26) in Chapter 3 so that the server services each client request in a separate thread.

DateThreads.java /Users/grasshopper/430/Homework

```
1   import java.net.*;
2   import java.io.*;
3
4   public class DateThreads
5   {
6       public static void main(String args[])
7       {
8           try
9           {
10              ServerSocket sock = new ServerSocket(6013);
11              // now listen for connections
12              while (true)
13              {
14                  Socket client = sock.accept();
15                  new DateTask(client).start();
16              }
17          }
18          catch (IOException ioe)
19          {
20              System.err.println(ioe);
21          }
22      }
23  }
24
25  public class DateTask implements Runnable
26  {
27      Socket client;
28
29      public DateTask(Socket client)
30      {
31         this.client = client;
32      }
33
34      public void run()
35      {
36          try
37          {
38              PrintWriter pout = new
39              PrintWriter(client.getOutputStream(), true);
40              // write the Date to the socket
41              pout.println(new java.util.Date().toString());
42              // close the socket and resume
43              // listening for connections
44              client.close();
45
46          }
47          catch (InterruptedException e) {};
48      }
49  }
```

4.18 Modify the socket-based date server (Figure 3.26) in Chapter 3 so that the server services each client request using a thread pool.

• DatePools.java /Users/grasshopper/430/Homework

```
1   import java.net.*;
2   import java.io.*;
3   import java.util.concurrent.Executors;
4   import java.util.concurrent.ExecutorService;
5
6   public class DatePools
7   {
8       public static void main(String args[])
9       {
10          try
11          {
12              ServerSocket sock = new ServerSocket(6013);
13              ExecutorService threadExecutor = Executors.newCachedThreadPool();
14              // now listen for connections
15              while (true)
16              {
17                  Socket client = sock.accept();
18                  DateTask dt = new DateTask(client);
19                  threadExecutor.execute(dt);
20              }
21          }
22          catch (IOException ioe)
23          {
24              System.err.println(ioe);
25          }
26      }
27  }
28
29  public class DateTask implements Runnable
30  {
31      Socket client;
32
33      public DateTask(Socket client)
34      {
35         this.client = client;
36      }
37
38      public void run()
39      {
40          try
41          {
42              PrintWriter pout = new
43              PrintWriter(client.getOutputStream(), true);
44              // write the Date to the socket
45              pout.println(new java.util.Date().toString());
46              // close the socket and resume
47              // listening for connections
48              client.close();
49
50          }
51          catch (InterruptedException e) {};
52      }
53  }
```

5.4 What advantage is there in having different time-quantum sizes at different levels of a multilevel queuing system?

Some processes are short enough to start and finish within a single time quantum. These processes can be most efficiently executed in a queue with a short quantum. Other processes are more computationally intensive and require more CPU time to complete. These processes are better suited in a queue with a longer quantum so that they can execute as much work as possible without incurring the expensive overheads of context switching.