Programming Examples In Java for the Teaching of Operating Systems

Charles Ashbacher
Mount Mercy College
Cedar Rapids, Iowa
Charles Ashbacher Technologies
http://www.ashbacher.com

Abstract

Operating systems is a class where a great deal of theoretical material is presented and practical experience is sometimes lacking. For my classes I have developed a series of programming examples and exercises that demonstrate the actions of operations such as scheduling time sharing processes, priority queues and conflict avoidance. Java is an excellent language for the implementation of such programs due to the full support of threads. This makes it possible to have separate threads running to simultaneously add and remove processes from the schedule queue. I use these examples in my operating system course as well as for the contract training in Java that I do for corporations.

Introduction

Operating systems is a course where programming assignments can do a great deal to help the students solidify the knowledge. Unfortunately, most textbooks do not include such problems, except as unsolved exercises. I have found that the presentation of programs really helps the students understand the principles, and they enjoy the programs as well. Most of them take it after completing the standard cycle of introductory programming classes followed by data structures. Therefore, they are accustomed to solving problems with programs, a skill that needs to be further emphasized in their training.

The Fundamental Class For Processes

The assumption used in introducing the various scheduling algorithms is to assume that the amount of time it takes for a process to run to completion is known. Therefore, the simulation programs start with a simple class that contains a process ID and the time it will take to run to completion. Students are introduced to the concept of time-sharing rather early, so the basic assumption is that each process in the queue will be allocated one slice of time. Therefore, in addition to methods to return the ID and remaining time for the process, there is a method that decrements the time by one. If the process has run to completion, this method returns a true, leaving the decision to remove the process to the queue manager. The code of this class is given in listing 1.

Listing 1

```
public class ThreadProcessEntry
 {
  private int processID;
  private float processTimeRequired;

  public ThreadProcessEntry(int theID,float theTime)
  {
   processID=theID;
   processTimeRequired=theTime;
  }

  public int getProcessID()
  {
```

```
   return processID;
   }

   public float getProcessTimeRequired()
   {
    return processTimeRequired;
   }

// This method reduces the time to completion by 1.0. If the value of time
// to completion has been set to zero, the method returns a true, indicating that
// it should be deleted from the queue.
   public boolean updateTimeRequired()
   {
    boolean stopProcess=false;
    if(processTimeRequired<=1.0f)
    {
     processTimeRequired=0.0f;
     stopProcess=true;
    }
    else
    {
     processTimeRequired--;
    }
    return stopProcess;
   }

   public String toString()
   {
    String str1="   "+processID+"          "+processTimeRequired;
    return str1;
   }

  }
```

The Time-Sharing Program

Java supports the Vector class, which is a sequential data structure that can dynamically grow. To start the process, a Vector is filled with processes where the time to completion is randomly assigned.

Two threads are created to alter the elements in the queue. One will add processes to the queue at a regular rate if there is room and is given in listing 2.

Listing 2

```
import java.io.*;
import java.util.*;
import javax.swing.*;

public class AddProcessesThread extends Thread
{
 private Vector theVector;
 ThreadProcessEntry theEntry;
 Random theRandom=new Random();
 public final long THEDELAY=4000;
 public final int QUEUESIZE=15;
 float theTime;
```

```
 int theSize;
 static int theNumber;
 JList theList;

 public AddProcessesThread(Vector inVector, JList inList)
 {
  theVector=inVector;
  theList=inList;
  theNumber=theVector.size()+1;
 }

 public void run()
 {
  try
  {
   while(true)
   {
    sleep(THEDELAY);
    theTime=20.0f*theRandom.nextFloat();
    theSize=theVector.size();
    if(theSize<=QUEUESIZE)
    {
     theEntry=new ThreadProcessEntry(theNumber,theTime);
     theNumber++;
     theVector.add(theEntry);
     theList.paint(theList.getGraphics());
    }
// Used for examination when needed.
//    else
//    {
//     JOptionPane.showMessageDialog(null,"The process queue is full");
//    }
   }
  }
  catch(InterruptedException ie)
  {
   JOptionPane.showMessageDialog(null,"The thread to add processes has been interrupted");

  }
 }

}
```

The long time for the delay is so that the user can see the changes being made in the queue. The first input to the constructor is the Vector containing the data and the second is the JList where the process queue will be displayed. The thread repaints the JList after the process is added.

A separate class that extends Thread is created to run the processes in the queue and the code of that class appears in listing 3.

Listing 3

```
import javax.swing.*;
import java.util.*;

public class RunProcessesThread extends Thread
```

```
{
 Vector theVector;
 JList theList;
 public final long THEDELAY=2000;

 public RunProcessesThread(Vector inVector,JList inList)
 {
  theVector=inVector;
  theList=inList;
 }

 public void run()
 {
  ThreadProcessEntry theEntry;
  boolean testRemove;

  try
  {
   while(!theVector.isEmpty())
   {
    sleep(THEDELAY);
    for(int i=0;i<theVector.size();i++)
    {
     theEntry=(ThreadProcessEntry)theVector.get(i);
     testRemove=theEntry.updateTimeRequired();
     if(testRemove==true)
     {
      theVector.remove(i);
     }
     }
     theList.paint(theList.getGraphics());
    }
   }
    catch(InterruptedException ie)
    {
     JOptionPane.showMessageDialog(null,"The thread to increment processes has been interrupted");
    }
  JOptionPane.showMessageDialog(null,"The queue is empty");
 }

}
```

This thread simply moves through the processes in the Vector, decrementing the time to completion and removing any processes that have run to completion. The thread repaints the JList after all the processes have had their times decremented and those completed have been removed from the queue.

The window of the program contains a JList with associated ListModel and CellRenderer. Each entry in the JList is an extension of the JLabel and one instance of each of the threads to alter the queue are created and started. Therefore, the user can watch the procession of the processes through the queue. Simple changes in the sleep() times as well as altering the priority of the threads can further demonstrate the running of the queue.

Running Multiple Queues

In my discussions of process priorities and the avoidance of starvation, the inherent tradeoffs of giving some processes priority versus starvation are described. To demonstrate this, three queues are constructed, and given three different priorities. This requires an alteration of the structure so that the structure of the JList is encapsulated in a class, which is given in listing 4.

Listing 4

```
import ThreadProcessEntry.*;
import AddProcessesThread.*;
import RunProcessesThread.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.util.*;
import javax.swing.event.*;

public class DisplayQueue extends JList
{
 public final int THESIZE=15;
 private Vector theVector;
 private TheListModel theModel;
 private TheCellRenderer theRenderer=new TheCellRenderer();
 private Random theRandom=new Random();
 private float theTime;
 private Border listEdge;
 private Border listBorder;

 public DisplayQueue(Vector inVector)
 {
  theVector=inVector;
  theModel=new TheListModel();
  setModel(theModel);
  listEdge=BorderFactory.createLineBorder(Color.red,3);
  listBorder=BorderFactory.createTitledBorder(listEdge,"  ID        Time Remaining");
  setBorder(listBorder);
  setCellRenderer(theRenderer);
  setFixedCellWidth(160);
  ThreadProcessEntry theEntry;
  for(int i=0;i<THESIZE;i++)
  {
   theTime=20.0f*theRandom.nextFloat();
   theEntry=new ThreadProcessEntry(i+1,theTime);
   theVector.add(theEntry);
  }
 }

 public Vector getVector()
 {
  return theVector;
 }

 public class TheListModel extends AbstractListModel
 {
  public int getSize()
  {
```

```
  return theVector.size();
 }

 public Object getElementAt(int index)
 {
  String theElement=((ThreadProcessEntry)theVector.get(index)).toString();
  return(theElement);
 }
}

class TheCellRenderer extends JLabel implements ListCellRenderer
{

 public TheCellRenderer()
 {
  setOpaque(true);
 }

 public Component getListCellRendererComponent(JList list, Object value,
                        int index,boolean isSelected,boolean cellHasFocus)
 {
  if(isSelected)
  {
   setBackground(Color.white);
   setForeground(Color.black);
  }
  else
  {
   setBackground(Color.blue);
   setForeground(Color.red);
  }
  setText(value.toString());
  return this;
 }
}
}
```

The constructor of the class accepts a Vector, which contains the data of the queue that the list is to display.

Three different JLists are created and displayed in the program window. Each contains its' own queue of processes and there is an instance of the threads to add and run the processes for each queue. The priorities of the threads for the different queues are set to MIN_PRIORITY, NORM_PRIORITY and MAX_PRIORITY respectively. Therefore, when the program runs, the students can observe the dynamic changes.

Adding the Processes To A Sorted Queue

After the initial concepts of time-sharing and how processes are executed via the allocation of time slices, the next algorithm covered is the shortest-job-first algorithm. In this case, jobs are loaded according to their time to run. This requires an underlying data structure for the queue that supports the easy inserting of an item into an arbitrary position in the list. Since Java supports the LinkedList data structure and the entries are sorted based on time to completion, it is a simple matter to change the underlying container to a LinkedList and create a method that adds the next entry to the proper location in the queue. The thread that runs the processes is then altered so it runs only the leading process of the queue to completion for each

iteration of the loop. This allows students to run various simulations and the modifications of the previous program are given as a programming assignment.

Adding Processes Based On Assigned Priorities

The next step in the operating system class is to assign each process a priority and then add the process to the single queue based on the ordering of the priority. This requires a very straightforward modification of the class representing the process, which is given in listing 5.

Listing 5

```
public class PriorityProcessEntry
{
 private int processID;
 private float processTimeRequired;
 private int processPriority;

 public PriorityProcessEntry(int theID,float theTime,int priority)
 {
  processID=theID;
  processTimeRequired=theTime;
  processPriority=priority;
 }

 public int getProcessID()
 {
  return processID;
 }

 public float getProcessTimeRequired()
 {
  return processTimeRequired;
 }

 public int getProcessPriority()
 {
  return processPriority;
 }

 public boolean updateTimeRequired()
 {
  boolean stopProcess=false;
  if(processTimeRequired<=1.0f)
  {
   processTimeRequired=0.0f;
   stopProcess=true;
  }
  else
  {
   processTimeRequired--;
  }
  return stopProcess;
 }
```

```
 public String toString()
 {
  String str1="The process ID is "+processID;
  str1=str1+" and the time to completion is "+processTimeRequired;
  return str1;
 }

}
```

The changes from the previous process class are marked in bold.

The method to insert a new process into the queue is altered so that it is added based on the priority and a typical method is given in listing 6.

```
public boolean insert(Object obj)
   {
    boolean theResult=false;
    int addPosition;
    if(theList.size()<PriorityQueue1.MAXIMUM_IN_QUEUE)
    {
     if((theList.isEmpty())||((!theList.isEmpty())&&
           (((PriorityProcessEntry)obj).getProcessPriority()>
           ((PriorityProcessEntry)theList.getFirst()).getProcessPriority())))
     {
      theList.addFirst(obj);
      theResult=true;
      return theResult;
     }
     ListIterator theIterator=theList.listIterator();
     boolean found=false;
     int positioncount=0;
     while((theIterator.hasNext())&&(found==false))
     {
      if(((PriorityProcessEntry)obj).getProcessPriority()>
            ((PriorityProcessEntry)theIterator.next()).getProcessPriority())
      {
       addPosition=theIterator.previousIndex();
       theList.add(addPosition,obj);
       found=true;
      }
     }
     if(found==false)
     {
      theList.addLast(obj);
     }
     theResult=true;
    }
    return theResult;
   }
```

Which is a standard algorithm for inserting an element into a sorted linked list.

Given these programs as a base, the students are then able to code simulations for several different scenarios. The following is a program that is commonly assigned as a programming assignment.

In this exercise, you will use a priority queue where the processes can have a priority from 1 through 300 with 300 the lowest. In this project, each process will run to completion when it starts. Immediately after a process runs, a new one is added to the priority queue.

A counter will also be kept so that after every fifth process is executed, all processes where the priority is greater than one will have their priority incremented by one. This will not change the order of the processes in the table.

Initialize the queue with a set of thirty processes where one process has a priority of 300. Note this process in some way.

Program a simulation where a set of processes is added until the process that had an initial priority of 300 is executed. The processes that are added will have their processes assigned in a random manner. Run this program three times and note how long it takes before the process with initial priority of 1 is executed.

Change the simulation so that each added process has a priority of 1. Run the simulation and note how long it takes before the process with an initial priority of 300 is executed.

Copies of these programs are available upon request.