Chapter 9 Multithreading

Chapter Topics

- Thread Basics
- Thread Synchronization
- Animations

9.1 Thread Basic

9.1.1 Terminologies

- A process is an independent executing unit that contains its own state information and uses its own address space. Processes only interact with each other via inter-process communication mechanism.
- A single process might contains multiple threads; all threads within a process share the same state and same memory space, and can communicate with each other directly, because they share the same variables.

9.1.2 Threads and the Runnable Interface type

1) Basic setting up

```
public interface Runnable
{
    void run();
}
```

1. Define class that implements Runnable
2. Place the code for the task into run method of the class.
3. Create an object of the Runnable class
4. Construct a Thread object and supply the Runnable object in the constructor
5. Call the start method of the Thread object to start the thread.

2) Runnable class

```
public class MyRunnable implements Runnable
{
    public void run()
    {
        thread action
    }
}
```
3) Program with main method that creates and starts threads

Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();

4) Example

/**
   An action that repeatedly prints a greeting.
*/
public class GreetingProducer implements Runnable {
    private String greeting;
    private static final int REPETITIONS = 10;
    private static final int DELAY = 1000;

    /**
     Constructs the producer object.
     @param aGreeting the greeting to display
     */
    public GreetingProducer(String aGreeting) {
        greeting = aGreeting;
    }

    public void run() {
        try {
            for (int i = 1; i <= REPETITIONS; i++)
            {
                System.out.println(i + " : " + greeting);
                Thread.sleep(DELAY);
            }
        } catch (InterruptedException exception) {
        }
    }
}

/**
   This program runs two threads in parallel.
*/
public class ThreadTester {
    public static void main(String[] args) {

Runnable r1 = new GreetingProducer("Hello, World!");
Runnable r2 = new GreetingProducer("Goodbye, World!");

Thread t1 = new Thread(r1);
Thread t2 = new Thread(r2);

t1.start();
t2.start();

Class Diagram

Sequence Diagram
9.1.3 Thread States and Priorities

Each thread runs for a time slice. Then, the thread scheduler activates another thread. The thread scheduler gives no guarantee about the order in which threads are executed.

(1) Thread States

- New (before start is called)
- Runnable
- Blocked

- Reasons for blocked state:
  - Sleeping
  - Waiting for I/O
  - Waiting to acquire lock (later)
  - Waiting for condition (later)
- Unblocks only if the reason for block goes away

- Dead (after the run method exits)
(2) Thread priorities
- Every thread has a priority. By default a thread inherits the priority of the thread that constructed it.
- Thread priorities are highly system dependent.
  - Application programmers should generally not adjust thread priority. (The setPriority method can be used to change a default priority.) You should never write your programs so that their correct functioning depends on thread priority levels.

(3) Thread scheduling algorithm
- Scheduler activates the next thread if
  - a thread has completed its time slice
  - a thread has blocked itself
  - a thread with higher priority has become runnable
- Scheduler determines new thread to run
  - looks only at runnable threads
  - picks one with highest priority

9.1.4 Terminating Threads
- In a normal way, a thread terminates when the run method exits.
- Sometimes it is necessary to terminate running thread
  - Don't use deprecated stop method
Interrupt thread by calling interrupt

- Calling t.interrupt() doesn't actually interrupt t; just sets a flag
- Interrupted thread must sense interruption and exit its run method
- Interrupted thread has chance to clean up

### 9.1.6 Sensing Interruptions

- Use a public method of Thread that checks if the “interrupted flag”. If the flag is set, the method clears the flag and throws InterruptedException. The most practical strategy is to surround the entire work portion of the run method with a try block that catches the InterruptedException.

```java
public class MyRunnable implements Runnable {
    public void run() {
        try {
            while (...) {
                do work
                Thread.sleep(...);
            }
        }
        catch (InterruptedException e) {} // clean up
    }
}
```

- Manually check if the “interrupted flag” is set.

```java
if (Thread.currentThread().isInteruppted()) ... 
```

### 9.2 Thread Synchronization

Note: I include the part of BoundedQueue program that will be used in this section.

```java
public class BoundedQueue<E>
{
    private E[] elements;
    private int head;
    private int tail;
    private int count;
```
/**
 * Constructs an empty queue.
 * @param capacity the maximum capacity of the queue
 * @precondition capacity > 0
 */
public BoundedQueue(int capacity)
{
    elements = (E[])new Object[capacity];
    count = 0;
    head = 0;
    tail = 0;
}

/**
 * Remove object at head.
 * @return the object that has been removed from the queue
 * @precondition size() > 0
 */
public E remove()
{
    E r = elements[head];
    head = (head + 1) % elements.length;
    count--;
    return r;
}

/**
 * Append an object at tail.
 * @param anObject the object to be appended
 * @return true since this operation modifies the queue.
 * (This is a requirement of the collections framework.)
 * @precondition !isFull()
 */
public boolean add(E anObject)
{
    elements[tail] = anObject;
    tail = (tail + 1) % elements.length;
    count++;
    return true;
}

/**
 * Checks whether this queue is full.
 * @return true if the queue is full
 */
public boolean isFull()
{
    return count == elements.length;
}

Producer (Runnable)

Instance fields
    greeting
    queue
    greetingCount

Consumer (Runnable)

Instance fields
    queue
    greetingCount

int i = 1;
while (i <= greetingCount)
{
    if (!queue.isFull())
    {
        queue.add(i + ": " + greeting);
        i++;
    }
    Thread.sleep(some delay);
}

queue.add

elements[tail] = newValue;
tail++;

queue.remove

E result = elements[head];
head++;
return result;

/**
 * This program runs three threads in parallel.
 */
public class ThreadTester
public static void main(String[] args)
{
    BoundedQueue<String> queue = new BoundedQueue<String>(10);
    final int GREETING_COUNT = 100;
    Runnable run1 =
        new Producer("Hello, World!", queue, GREETING_COUNT);
    Runnable run2 =
        new Producer("Goodbye, World!", queue, GREETING_COUNT);
    Runnable run3 = new Consumer(queue, 2 * GREETING_COUNT);

    Thread thread1 = new Thread(run1);
    Thread thread2 = new Thread(run2);
    Thread thread3 = new Thread(run3);

    thread1.start();
    thread2.start();
    thread3.start();
}

9.2.1 Race Condition

A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

A race condition scenario
1. The first thread calls the add method of the BoundedQueue class and executes the following statement:

   `elements[tail] = newValue;`

2. Second thread calls `add` and executes

   ```
   elements[tail] = newValue;
   tail++;  
   ```

3. First thread executes

   `tail++;`
To fix the race conditions, you need to ensure that only one thread manipulates the queue at any given moment.

### 9.2.2 Reentrant Locks

- To remove race condition, you can have a thread acquire a lock.
- Locking mechanism
  - While a thread owns the lock, no other thread can acquire the same lock.
  - When another thread tries to acquire same lock, it blocks
  - When first thread releases lock, other thread is unblocked and tries again
- Two kinds of locks
  - Objects of the ReentrantLock or another class that implements java.util.concurrent.Lock interface type
  - Locks that are built into every Java object
private Lock queueLock = new ReentrantLock();

```
public void add(E newValue)
{
    queueLock.lock();
    elements[tail] = newValue;
    tail++;
    queueLock.unlock();
}
```

- Does using a lock eliminate the race condition? Yes.

1. First thread calls `add` and acquires lock, then executes
   `elements[tail] = newValue;`
2. Second thread calls `add` and tries to acquire lock, but it is blocked
3. First thread executes
   `tail++;`
4. First thread completes `add`, releases lock
5. Second thread unblocked
6. Second thread acquires lock, starts executing protected code

### 9.2.3 Avoiding deadlocks

**Deadlock**

- It is desirable to include the test for full queue in the add method.

Producer

```
if (!queue.isFull())
{
    queue.add(greeting);
}
```

```
queue.add(i + ": " + greeting);
queue.add(i + ": " + greeting);
```
BoundedQueue

```java
public void add(E newValue) {
    queueLock.lock();
    elements[tail] = newValue;
    tail++;
    queueLock.unlock();
}

public void remove() {
    queueLock.lock();
    E result = elements[head];
    head++;
    return result;
    queueLock.unlock();
}
```

- How can a thread wait for more space? If you use the sleep method for this purpose, deadlock occurs.

If a producer thread sleeps after acquiring a lock, no consumer can removes item from the queue. A consumer calls remove but will be simply blocks because the consumer thread attempts to obtain the same lock the producer owns. The producer cannot wake up until any consumer removes an item.

Avoiding Deadlocks

- Condition object manages set of threads that wait for the condition to change.
• Calling to `await()` on a condition object temporarily release a lock and blocks the current thread. The current thread is added to a set of threads that are waiting for the condition. To unblock the thread, another thread must execute the `signalAll` method on the same condition object.

• The `signalAll()` method unblocks all threads waiting for the condition, making them all Runnable again.

BoundedQueue.java

```java
private Object[] elements;
private int head;
private int tail;
private int size;

private Lock queueLock = new ReentrantLock();
private Condition spaceAvailableCondition = queueLock.newCondition();
private Condition valueAvailableCondition = queueLock.newCondition();
```

```java
public void add(E newValue) throws InterruptedException {
    queueLock.lock();
    try {
        while (size == elements.length)
            spaceAvailableCondition.await();
        elements[tail] = newValue;
        tail = (tail+1)%elements.length;
        size++;
        valueAvailableCondition.signalAll();
    } finally {
        queueLock.unlock();
    }
}
```

```java
public E remove() throws InterruptedException {
    queueLock.lock();
    try {
        while (size == 0)
            valueAvailableCondition.await();
        E r = (E) elements[head];
        head = (head + 1)%elements.length;
        size--;
        spaceAvailableCondition.signalAll();
        return r;
    } finally {
        queueLock.unlock();
    }
}
```
9.2.4 Primitive Object Locks

- Each Java object has a lock
- Calling a synchronized method acquires lock of implicit parameter
- Leaving the synchronized method releases lock
- `Object.wait` blocks current thread and adds it to wait set
- `Object.notifyAll` unblocks waiting threads
- Synchronized methods were invented for a specific purpose: to remove race condition by ensuring the integrity of a data structure (i.e. atomic operation)

```java
public class BoundedQueue<E>
{
    private Object[] elements;
    private int head;
    private int tail;
    private int size;

    /**
     * Constructs an empty queue.
     * @param capacity the maximum capacity of the queue
     */
    public BoundedQueue(int capacity)
    {
        elements = new Object[capacity];
        head = 0;
        tail = 0;
        size = 0;
    }

    /**
     * Removes the object at the head.
     * @return the object that has been removed from the queue
     */
    public synchronized E remove() throws InterruptedException
    {
        while (size == 0) wait();
        E r = (E) elements[head];
        head = (head + 1) % elements.length;
        size--;
        notifyAll();
        return r;
    }

    /**
     * Appends an object at the tail.
     * @param newValue the object to be appended
     */
    public synchronized void append(E newValue)
    {
        elements[tail] = newValue;
        tail = (tail + 1) % elements.length;
        size++;
        notifyAll();
    }
}
```
public synchronized void add(E newValue) throws InterruptedException
{
    while (size == elements.length) wait();
    elements[tail] = newValue;
    tail = (tail + 1) % elements.length;
    size++;
    notifyAll();
}

9.2.5 Algorithm Animation

Merge Sort

1. Divide the unsorted list into two sublists of about half the size
2. Sort each of the two sublists
3. Merge the two sorted sublists back into one sorted list.

private static <E> void mergeSort(E[] a, int from, int to, Comparator<? super E> comp)
{
    if (from == to) return;
    int mid = (from + to) / 2;
    // Sort the first and the second half
    mergeSort(a, from, mid, comp);
    mergeSort(a, mid + 1, to, comp);
    merge(a, from, mid, to, comp); // the merge part of program is assumed and does not appear in this lecture note.
}
```java
Runnable r = new Sorter(values, panel);
Thread t = new Thread(r);
t.start();

public void run()
{
    Comparator<Double> comp = new Comparator<Double>()
    {
        public int compare(Double d1, Double d2)
        {
            1) panel.setValues(values, d1, d2);
            2) Thread.sleep(DELAY);
            3) return (d1).compareTo(d2);
        }
    };
    MergeSorter.sort(values, comp);
}
```

ArrayComponent panel
= new ArrayComponent();
frame.add(panel, BorderLayout.CENTER);
...
Runnable r = new Sorter(values, panel);
Thread t = new Thread(r);
t.start();

Calls compare method each time two elements are compared.

AnimationTester
Sorter

```java
public void run()
{
    Comparator<Double> comp = new Comparator<Double>()
    {
        public int compare(Double d1, Double d2)
        {
            1) panel.setValues(values, d1, d2);
            2) Thread.sleep(DELAY);
            3) return (d1).compareTo(d2);
        }
    };
    MergeSorter.sort(values, comp);
}
```
import java.awt.*;
import javax.swing.*;

/**
   This program animates a sort algorithm.
*/
public class AnimationTester
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ArrayComponent panel = new ArrayComponent();
        frame.add(panel, BorderLayout.CENTER);
        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setVisible(true);
        Double[] values = new Double[VALUES_LENGTH];
        for (int i = 0; i < values.length; i++)
            values[i] = Math.random() * panel.getHeight();
        Runnable r = new Sorter(values, panel);
        Thread t = new Thread(r);
        t.start();
    }

    private static final int VALUES_LENGTH = 30;
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 300;
}


default.*;
import java.util.*;

/**
   This runnable executes a sort algorithm.
   When two elements are compared, the algorithm
   pauses and updates a panel.
*/
public class Sorter implements Runnable
{
    private Double[] values;
    private ArrayComponent panel;
    private static final int DELAY = 100;

    /**
Constructs the sorter.
@param values the array to sort
@param panel the panel for displaying the array
*/
public Sorter(Double[] values, ArrayComponent panel)
{
    this.values = values;
    this.panel = panel;
}

public void run()
{
    Comparator<Double> comp = new Comparator<Double>()
    {
        public int compare(Double d1, Double d2)
        {
            panel.setValues(values, d1, d2);
            try
            {
                Thread.sleep(DELAY);
            }
            catch (InterruptedException exception)
            {
                //Thread.currentThread().interrupt();
            }
            return (d1).compareTo(d2);
        }
    }
    MergeSorter.sort(values, comp);
    panel.setValues(values, null, null);
}

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * This component draws an array and marks two elements in the array.
 */
public class ArrayComponent extends JComponent
{
    private Double[] values;
    private Double marked1;
private Double marked2;

public synchronized void paintComponent(Graphics g) {
    if (values == null) return;
    Graphics2D g2 = (Graphics2D) g;
    int width = getWidth() / values.length;
    for (int i = 0; i < values.length; i++) {
        Double v = values[i];
        Rectangle2D bar = new Rectangle2D.Double(width * i, 0, width, v);
        if (v == marked1 || v == marked2)
            g2.fill(bar);
        else
            g2.draw(bar);
    }
}

/**
 * Sets the values to be painted.
 * @param values the array of values to display
 * @param marked1 the first marked element
 * @param marked2 the second marked element
 */
public synchronized void setValues(Double[] values, Double marked1, Double marked2) {
    this.values = (Double[]) values.clone();
    this.marked1 = marked1;
    this.marked2 = marked2;
    repaint();
}