Multithreading

by Terry Sergeant

Contents

1 What and Why

In this course we view multithreading as a programming technique that allows a single program to instruct the computer to perform multiple sections of code concurrently. In a single-threaded program the statements of the program are performed in a specific order and we assume that once a statement is issued, it completes before the next statement begins. In a multi-threaded program we can launch multiple "threads" of execution that happen "at the same time" with no guarantee as to which will be complete first.

In many sections of code we want the assurance that statements will happens in a specified order because some statements require results from previous statements. So, multi-threaded programs are not a replacement for single-threaded programs but instead are used in specific situations.

Some examples of when multi-threaded programs might be useful:

- Suppose you have a menu-driven program and you also want to display a clock at the top of the screen. You can have one thread be responsible for updating the clock and another thread to handle the rest of the program. These two threads should run at the same time.
- Suppose you have a web-based game that allows players to connect as the game proceeds. You can assign a separate thread to handle communication with each player rather than trying to figure out how to handle multiple connections within your code. That is, your code can focus on one connection at a time.

2 Mechanics of Multithreading

2.1 Background and Context

In operating systems lingo a *process* is a program that has been started. A typical desktop computer will have dozens of processes that are started. In a single CPU (single core) system only one of the started processes can be actually running at a time. The OS will assign a process to be run for a short period of time and then remove the process from the running state and will assign another process to be run. By switching among the processes rapidly it creates the illusion that they are all running at the same time.

A *thread* is a "lightweight process". Threads that have the same parent will share memory and other resources so creating additional threads is faster than creating a completely new/separate process. Also, since threads share memory and files they can communicate with one another easily using these shared resources.

So, when we write a multi-threaded program we have a parent process that will create multiple threads. The threads will be executed concurrently (by having the OS assign them to the CPU for short periods).

2.2 A Java Example

Java supports a variety of ways to create multithreaded programs:

- Inherit from the `Thread` class
- Implement the `Runnable` interface
- Use the `Callable` and `Future` classes

In our examples we will focus on implementing the `Runnable` interface because it is perhaps simpler to learn initially and is flexible enough for our purposes. If you want to become a multithreading guru then you might consider mastering the use of `Callable` and `Future`.

NOTE: This and other examples are available online at: `https://josephus.hsutx.edu/classes/ds/source/multithreading/`

ThreadDemo3.java

```
1  /**
2   * Demonstrates how to create two threads and assign each to
3   * calculate a path length for the provided hailstone sequence.
4  */
5
6  import java.util.Scanner;
7
8  public class ThreadDemo3
9  {
10    public static void main(String [] args) throws Exception
11    {
12       Scanner kb= new Scanner(System.in);
13       Data mydata1= new Data(5);
14       Data mydata2= new Data(3);
15       Thread doit1= new Thread(new HailStone(mydata1));
16       Thread doit2= new Thread(new HailStone(mydata2));
17       doit1.start();
18       doit2.start();
19
20       doit1.join();
21       doit2.join();
22
23       System.out.println(""+doit1.getName()+" says " + mydata1);
24       System.out.println(""+doit2.getName()+" says " + mydata2);
25    }
26 }
27
28 // When implementing Runnable we must provide a method called
```

```
29 // run() with void return type and no parameters.
30 class HailStone implements Runnable
31 {
32    private Data data;
33    private int mynum;
34
35    public HailStone(Data d)
36    {
37       data= d;
38       mynum= 1;
39    }
40
41    @Override
42    public void run()
43    {
44       Thread me= Thread.currentThread();
45       int m;
46       int pathLength= 0;
47       String state;
48
49       System.out.println("I am "+me.getName() +
50             " and I am working on stuff ...");
51       m= data.n;          // given a number
52       while (m > 1) {
53          pathLength++;    // determine number of
54          if (m%2==0)      // iterations thru the
55             m= m / 2;     // "hailstone" function
56          else             // before reaching 1
57             m= 3*m+1;
58       }
59       data.pathLength= pathLength;
60       System.out.println("I, "+me.getName() +
61             ", am done with my work ...");
62    }
63 }
64
65
66 class Data
67 {
68    public Data(int num) { n= num; pathLength= 0; }
69    public int n;           // value to calculate
70    public int pathLength;  // length of path to reach 1
71    public String toString() {
72       return ""+n+" takes "+pathLength+" steps to fall";
73    }
74 }
```

Some things to note about this sample program:

- The parent program (`main()`) creates two threads (`doit1` and `doit2`).
- Data is shared between `main()` and the threads by passing an object to each threads constructor.
- The threads begin executing on lines 17–18. Calling the `start()` method causes the `run()` method to be called.
- Calling the `join()` method for a thread causes `main()` to wait until the given thread finishes.
- When implementing the `Runnable` interface you must defined a method called `run()` that has a void return type and take no parameters and throws no exceptions.
- Because the `Data` objects were created in `main()` and then passed to each thread (via the constructor), any changes to the objects are seen in `main()`. That is, the threads can share memory with `main()` (and with each other).

Ideas for experimentation:

- Run the program several times and observe that the order in which various statements are output can change.
- Remove the `.join()` commands and then run the program several times observing the output.

3 Issues with Multithreading

Any time two or more threads are executing concurrently and accessing a shared resource you need to be aware that we don't have a guarantee regarding when threads will access those resources relative to one another. This can sometimes lead to problems.

A *critical section* is any shared resource (or section of code) that requires mutually exclusive access (i.e., can only be accessed by one thread at a time). If two thread enter/access a critical section at the same time then we have a problem. Enforcing mutual exclusion is tricky because of the fact that threads are constantly being started and stopped by the OS scheduler (and we don't have control over the scheduler). Suppose:

1. thread A enters a critical section and then is stopped by the OS scheduler
2. the scheduler then allows thread B to run
3. thread B enters the same critical section
4. now both threads are in the critical section (which should not happen)

3.1 We Have a Problem

Consider the following section of code:

ThreadCSProblem.java

```
1  /**
2   * Demonstrates what might happen when we don't enforce mutual
3   * exclusion in a multithreaded program.
4  */
5
6  public class ThreadCSProblem
7  {
8     public static void main(String [] args) throws Exception
9     {
10       Data mydata= new Data();
11       Thread [] pool= new Thread[10];
12       int i;
13       long start,stop;
14
15       // create 10 threads all sharing the same data
16       for (i=0; i<10; i++)
17          pool[i]= new Thread(new AddEmUp(mydata));
18
19       start= System.currentTimeMillis();
20       // start all the threads
21       for (i=0; i<10; i++)
22          pool[i].start();
23
24       // wait for them to finish
25       for (i=0; i<10; i++)
26          pool[i].join();
27       stop= System.currentTimeMillis();
28
29       System.out.println("Time elapsed: " +
30             (stop-start)/1000.0+" seconds");
31       System.out.println("The variable is now: " + mydata);
32    }
33 }
34
35 class AddEmUp implements Runnable
36 {
37    private Data data;
```

```
38
39    public AddEmUp(Data d)
40    {
41       data= d;
42    }
43
44    public void run()
45    {
46       int i;
47
48       for (i=0; i<10000000; i++)
49          data.addOne();
50    }
51 }
52
53 class Data
54 {
55    private int n;
56    public Data() { n= 0; }
57    public void addOne() { n++; }
58    public int getNum() { return n; }
59    public String toString() { return ""+n; }
60 }
```

This program shares a single integer among 10 threads. Each thread uses a loop to increment the integer 10,000,000 times. So when the program finishes we expect the value in the integer to be 100,000,000. Here is the output from one of the times I ran the program:

```
Time elapsed: 0.075 seconds
The variable is now: 45249154
```

So, why the strange, smaller-than-expected number? The `addOne()` method represents a critical section. This is because reading the current value of the number, adding one, and storing the answer back into the variable requires multiple steps. If two threads both read the same value and then add one to the value and store then answer then the counter will be incremented only once when it should have been incremented twice. Likewise, if a thread is postponed for quite a while then it will set the value of the integer to an "old" value when it is running again.

There are two common ways to enforce mutual exclusion in a multithreaded program: semaphores and monitors.

3.2 Semaphores

A semaphore is a special type of counter with two supported operations:

acquire() This operation attempt to access the semaphore and if successful will decrement its counter. If the counter has reached zero then the thread calling this method will sleep until awakened by another process.

release() This operation will release an acquired semaphore. If one or more threads is sleeping (because they tried and failed to acquire the semaphore) then calling this method will cause one of them to be awakened (at which time they acquire the semaphore). If no threads are sleeping then the counter is incremented.

So, to enforce mutual exclusion using a semaphore you do something like this:

```
Semaphore sem= new Semaphore(1); // set counter to 1 initially
...
sem.acquire();
// critical section code goes here
sem.release();
```

Here we use a semaphore to solve the issue with `ThreadCSProblem.java` presented above.

ThreadCSSemaphore.java

```
/**
 * One way to address the problem given in ThreadCSProblem.java:
 * use semaphores.
*/

import java.util.Scanner;
import java.util.concurrent.Semaphore;

public class ThreadCSSemaphore
{
   public static void main(String [] args) throws Exception
   {
      Scanner kb= new Scanner(System.in);
      Data mydata= new Data();
      Thread [] pool= new Thread[10];
      long start,stop;

      int i;

      for (i=0; i<10; i++)
         pool[i]= new Thread(new AddEmUp(mydata));

      start= System.currentTimeMillis();
      for (i=0; i<10; i++)
         pool[i].start();

      for (i=0; i<10; i++)
         pool[i].join();
      stop= System.currentTimeMillis();

      System.out.println("Time Elapsed       : " +
            (stop-start) / 1000.0 + " seconds");
      System.out.println("The variable is now: " + mydata);
   }
}

class AddEmUp implements Runnable
{
   private Data data;

   public AddEmUp(Data d) {
      data= d;
   }

   public void run() {
      for (int i=0; i<10000000; i++) {
         data.addOne();
      }
   }
}

class Data
{
   private Semaphore cs;
   private int n;
   public Data() {
      n= 0;
      cs= new Semaphore(1); // set counter to 1
   }
   public void addOne() {
      try {
         cs.acquire();
         n++;
         cs.release();
      } catch (InterruptedException e) {
```

```
66          System.out.println(e);
67       }
68    }
69    public int getNum() { return n; }
70    public String toString() { return ""+n; }
71 }
```

Here is the output I recorded for this program:

```
Time Elapsed       : 2.68 seconds
The variable is now: 100000000
```

Some things to notice about this solution:

- we get the correct answer
- the time required to get the solution went from under a 10th of a second to nearly 3 seconds.
- We had to handle the `InterruptedException` that can be thrown when using semaphores.
- The semaphore is shared among all the threads. If each thread had used their own semaphore then mutual exclusion would not be enforced.

3.3 Monitors

A monitor is a language construct that allows you to designate a method as a "monitor". The compiler will add necessary code to ensure that only one process will be allowed in the monitor at a time. Monitors are a higher-level construct and are therefore easier to use that semaphores in many cases. However, they also are not as flexible as semaphores.

To designate a method as a monitor in Java you simply add the keyword `synchronized` in front of the method.

Here we use a monitor to solve the issue with `ThreadCSProblem.java` presented above.

ThreadCSSync.java

```
1  /**
2   * One way to address the problem given in ThreadCSProblem.java:
3   * monitors.
4  */
5
6  public class ThreadCSSync
7  {
8     public static void main(String [] args) throws Exception
9     {
10       Data mydata= new Data();
11       Thread [] pool= new Thread[10];
12       long start,stop;
13       int i;
14
15       for (i=0; i<10; i++)
16          pool[i]= new Thread(new AddEmUp(mydata));
17
18       start= System.currentTimeMillis();
19       for (i=0; i<10; i++)
20          pool[i].start();
21
22       for (i=0; i<10; i++)
23          pool[i].join();
24       stop= System.currentTimeMillis();
25
26       System.out.println("Time Elapsed       : " +
27             (stop-start) / 1000.0 + " seconds");
28       System.out.println("The variable is now: " + mydata);
29    }
30 }
31
32 class AddEmUp implements Runnable
33 {
34    private Data data;
35
36    public AddEmUp(Data d)
37    {
38       data= d;
39    }
40
41    public void run()
42    {
43       int i;
44
45          for (i=0; i<10000000; i++)
46             data.addOne();
47    }
48 }
49
50 class Data
51 {
52    private int n;
53    public Data() { n= 0; }
54    public synchronized void addOne() { n++; }
55    public int getNum() { return n; }
56    public String toString() { return ""+n; }
57 }
```

Here is the output I recorded for this program:

```
Time Elapsed       : 2.09 seconds
The variable is now: 100000000
```

Some things to notice about this solution:

- We get the correct answer.
- The time required to get the solution was about 2 seconds (slightly faster than semaphore solution and much slower than the original version that doesn't work).
- The only change we very simple to employ! Just designate the `addOne()` method as a monitor by adding the keyword `synchronized` to the method definition (line 53).

LESSON: If you can use a monitor then use it. Otherwise use a semaphore.

4 Classic Problems and Their Solutions

We will consider two classic synchronization problems. In both cases we will present semaphore solutions because monitor solutions require the use of condition variables which add complexity.

4.1 Readers-Writers Problem

Imagine an online database that allows users to read data and write data. Reading data is a safe operation in the sense that it is fine to have multiple users reading concurrently. When writing data, however, we begin to have issues such as encountered in `ThreadCSProblem.java` above. So, when a user is writing they should have exclusive access (no other readers or writers allowed).

There are quite a few variations of this problem. Here are a couple:

Reader Preference In this solution no reader is kept waiting unless a writer has already been granted access.

Writer Preference In this solution no writer is kept waiting unless a reader has already been granted access.

Each of these solutions can allow "starvation" of processes that are not preferred. The term *starvation* refers to a process that is being indefinitely postponed from acting because it is waiting on other processes. A more complex solution would be to avoid starvation (perhaps by keeping of order of arrival), but this is beyond the scope of our inquiry.

Imagine we have multiple concurrent processes that may occasionally read or write to a shared database as follows:

```
loop
   do_some_work()
        acquire_read_lock()
        db.read()
        release_read_lock()

   do_some_work()
        acquire_write_lock()
        db.write()
        release_write_lock()
end loop
```

So, our goal is to create code for these methods: `acquire_read_lock()`, `release_read_lock()`, `acquire_write_lock()`, `release_write_lock()`. Of course, these must be written to enforce the behavior required by our problem statement. **Before you look at the solutions proposed below take a moment to jot down ideas of your own.**

4.1.1 A Reader-Preference Solution

Here is one semaphore-based solution (written in pseudocode) that gives preference to readers:

Reader-Preference

```
Semaphore mutex = 1
Semaphore db = 1
int numReaders = 0

function acquire_read_lock() {
   mutex.acquire()
   numReaders = numReaders + 1
   if (numReaders == 1) {
      db.acquire()
   }
   mutex.release()
}

function release_read_lock() {
   mutex.acquire()
   numReaders = numReaders - 1
   if (numReaders == 0) {
      db.release()
   }
   mutex.release()
}

function acquire_write_lock() {
   db.acquire()
}

function release_write_lock() {
   db.release()
}
```

Notice that the semaphores `mutex` and `db` are both initially 1. So, only one process can acquire them at a time. The `db` semaphore is acquired by the first reader and is released by the last reader. So, if a writer attempts to acquire it while any reader is still reading, it will sleep. Likewise, if a writer has acquired the `db` semaphore then the next arriving reader will sleep when it tries to acquire it.

Now that you've seen an example solution to the reader-preference version of this problem, try to tweak this solution to produce a writer-preference version.

4.1.2 A Writer-Preference Solution

Here is one semaphore-based solution (written in pseudocode) that gives preference to writers:

Writer-Preference

```
Semaphore writer = 1
Semaphore mutex = 1
Semaphore db = 1
int numReaders = 0

function acquire_read_lock() {
   writer.acquire()
   mutex.acquire()
   numReaders = numReaders + 1
   if (numReaders == 1) {
      db.acquire()
   }
   mutex.release()
   writer.release()
}

function release_read_lock() {
   mutex.acquire()
   numReaders = numReaders - 1
   if (numReaders == 0) {
      db.release()
   }
   mutex.release()
}

function acquire_write_lock() {
   writer.acquire()
   db.acquire()
}

function release_write_lock() {
   db.release()
   writer.release()
}
```

This is a so-called "weak writer preference" solution in that when a writer finishes it is "up for grabs" as to whether the next process to run is a reader or a writer. The benefit of this solution is that when a writer wants to write it acquires the `writer` semaphore which blocks subsequent readers from acquiring the read lock. So, when the existing readers finish the writer will gain access to the database.

4.2 Producer-Consumer Problem

The *Bounded Buffer Producer-Consumer Problem* features two distinct types of processes: producers and consumer. There can be multiple instances of both types. Producers create some item (e.g., widgets) and then store them in a waiting area (i.e., the buffer). The waiting area (buffer) is of a fixed size and so can hold only a limited number of widgets. Consumers remove items from the waiting area and consume them.

So, we want producers and consumers to cooperate as follows:

- Both need to access the shared buffer.
- Producers should not continue to produce if the buffer gets full.

- Consumers should not continue to consume if the buffer is empty.
- Otherwise we want producers to continue producing and consumers to continue consuming.

4.2.1 A Partial Producer-Consumer Solution

Consider the following partial solution to the Producer-Consumer problem. This "solution" has a couple of problems. First, it does not treat access to the buffer (via `store()` and `get()`) as a critical section. Second, it performs a "busy wait". Suppose a consumer process is currently being executed by the CPU and that `numberOfItems` is 0. This consumer will continue executing the if-statement as long as it has the CPU and the if-statement will continue to be false. So, its entire time using the CPU is wasted.

Partial Producer-Consumer Solution

```
int numberOfItems = 0

function producer() {
   while (true) {
      if (numberOfItems < BUFFER_SIZE) {
         produce(item)
         numberOfItems = numberOfItems + 1
         store(item)
      }
   }
}

function consumer() {
   while (true) {
      if (numberOfItems > 0) {
         get(item)
         numberOfItems = numberOfItems - 1
         consume(item)
      }
   }
}
```

Take a moment to write a solution that uses semaphores to address the problems with this partial solution.

4.2.2 A Better Producer-Consumer Solution

Consider this semaphore-based solution:

Partial Producer-Consumer Solution

```
Semaphore mutex = 1
Semaphore used = 0
Semaphore available = BUFFER_SIZE

function producer() {
   while (true) {
      produce(item)
      available.acquire()
      mutex.acquire()
      store(item)
      mutex.release()
      used.release()
   }
}

function consumer() {
   while (true) {
      used.acquire()
      mutex.acquire()
      get(item)
      mutex.release()
      available.release()
      consume(item)
   }
}
```

The `mutex` semaphore is used to provide mutually exclusive access to the buffer. The `used` semaphore tracks how many slots in the buffer are filled and prevents a consumer from attempting to extract an item when the buffer is empty. The `available` semaphore prevents a producer from storing a produced item in the event the buffer is full.