16

Threads and Locks

In a Microsoft, Google or Amazon interview, it's not terribly common to be asked to implement an algorithm with threads (unless you're working in a team for which this is a particularly important skill). It is, however, relatively common for interviewers at any company to assess your general understanding of threads, particularly your understanding of deadlocks.

This chapter will provide an introduction to this topic.

Threads in Java

Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class. When a standalone application is run, a user thread is automatically created to execute the `main()` method. This thread is called the main thread.

In Java, we can implement threads in one of two ways:

- By implementing the `java.lang.Runnable` interface
- By extending the `java.lang.Thread` class

We will cover both of these below.

Implementing the Runnable Interface

The `Runnable` interface has the following very simple structure.

```
1  public interface Runnable {
2    void run();
3  }
```

To create and use a thread using this interface, we do the following:

1. Create a class which implements the `Runnable` interface. An object of this class is a `Runnable` object.
2. Create an object of type `Thread` by passing a `Runnable` object as argument to the `Thread` constructor. The `Thread` object now has a `Runnable` object that implements the `run()` method.

3. The `start()` method is invoked on the `Thread` object created in the previous step.

For example:

```
1  public class RunnableThreadExample implements Runnable {
2     public int count = 0;
3
4     public void run() {
5        System.out.println("RunnableThread starting.");
6        try {
7           while (count < 5) {
8              Thread.sleep(500);
9              count++;
10          }
11       } catch (InterruptedException exc) {
12          System.out.println("RunnableThread interrupted.");
13       }
14       System.out.println("RunnableThread terminating.");
15    }
16 }
17
18 public static void main(String[] args) {
19    RunnableThreadExample instance = new RunnableThreadExample();
20    Thread thread = new Thread(instance);
21    thread.start();
22
23    /* waits until above thread counts to 5 (slowly) */
24    while (instance.count != 5) {
25       try {
26          Thread.sleep(250);
27       } catch (InterruptedException exc) {
28          exc.printStackTrace();
29       }
30    }
31 }
```

In the above code, observe that all we really needed to do is have our class implement the `run()` method (line 4). Another method can then pass an instance of the class to `new Thread(obj)` (lines 19 - 20) and call `start()` on the thread (line 21).

Extending the Thread Class

Alternatively, we can create a thread by extending the `Thread` class. This will almost always mean that we override the `run()` method, and the subclass may also call the thread constructor explicitly in its constructor.

The below code provides an example of this.

```
1  public class ThreadExample extends Thread {
2     int count = 0;
3
4     public void run() {
```

```
5        System.out.println("Thread starting.");
6        try {
7           while (count < 5) {
8              Thread.sleep(500);
9              System.out.println("In Thread, count is " + count);
10             count++;
11          }
12       } catch (InterruptedException exc) {
13          System.out.println("Thread interrupted.");
14       }
15       System.out.println("Thread terminating.");
16    }
17 }
18
19 public class ExampleB {
20    public static void main(String args[]) {
21       ThreadExample instance = new ThreadExample();
22       instance.start();
23
24       while (instance.count != 5) {
25          try {
26             Thread.sleep(250);
27          } catch (InterruptedException exc) {
28             exc.printStackTrace();
29          }
30       }
31    }
32 }
```

This code is very similar to the first approach. The difference is that since we are extending the `Thread` class, rather than just implementing an interface, we can call `start()` on the instance of the class itself.

Extending the Thread Class vs. Implementing the Runnable Interface

When creating threads, there are two reasons why implementing the `Runnable` interface may be preferable to extending the `Thread` class:

- Java does not support multiple inheritance. Therefore, extending the `Thread` class means that the subclass cannot extend any other class. A class implementing the `Runnable` interface will be able to extend another class.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the `Thread` class would be excessive.

Synchronization and Locks

Threads within a given process share the same memory space, which is both a positive and a negative. It enables threads to share data, which can be valuable. However, it also creates the opportunity for issues when two threads modify a resource at the same

time. Java provides synchronization in order to control access to shared resources.

The keyword `synchronized` and the `lock` form the basis for implementing synchronized execution of code.

Synchronized Methods

Most commonly, we restrict access to shared resources through the use of the `synchronized` keyword. It can be applied to methods and code blocks, and restricts multiple threads from executing the code simultaneously *on the same object*.

To clarify the last point, consider the following code:

```
1  public class MyClass extends Thread  {
2     private String name;
3     private MyObject myObj;
4
5     public MyClass(MyObject obj, String n) {
6        name = n;
7        myObj = obj;
8     }
9
10    public void run() {
11       myObj.foo(name);
12    }
13 }
14
15 public class MyObject {
16    public synchronized void foo(String name) {
17       try {
18          System.out.println("Thread " + name + ".foo(): starting");
19          Thread.sleep(3000);
20          System.out.println("Thread " + name + ".foo(): ending");
21       } catch (InterruptedException exc) {
22          System.out.println("Thread " + name + ": interrupted.");
23       }
24    }
25 }
```

Can two instances of `MyClass` call `foo` at the same time? It depends. If they have the same instance of `MyObject`, then no. But, if they hold different references, then the answer is yes.

```
1  /* Difference references - both threads can call MyObject.foo() */
2  MyObject obj1 = new MyObject();
3  MyObject obj2 = new MyObject();
4  MyClass thread1 = new MyClass(obj1, "1");
5  MyClass thread2 = new MyClass(obj2, "2");
6  thread1.start();
7  thread2.start()
8
9  /* Same reference to obj. Only one will be allowed to call foo,
```

```
10   * and the other will be forced to wait. */
11  MyObject obj = new MyObject();
12  MyClass thread1 = new MyClass(obj, "1");
13  MyClass thread2 = new MyClass(obj, "2");
14  thread1.start()
15  thread2.start()
```

Static methods synchronize on the *class lock*. The two threads above could not simultaneously execute synchronized static methods on the same class, even if one is calling `foo` and the other is calling `bar`.

```
1   public class MyClass extends Thread  {
2      ...
3      public void run() {
4         if (name.equals("1")) MyObject.foo(name);
5         else if (name.equals("2")) MyObject.bar(name);
6      }
7   }
8
9   public class MyObject {
10     public static synchronized void foo(String name) {
11        /* same as before */
12     }
13
14     public static synchronized void bar(String name) {
15        /* same as foo */
16     }
17  }
```

If you run this code, you will see the following printed:

```
Thread 1.foo(): starting
Thread 1.foo(): ending
Thread 2.bar(): starting
Thread 2.bar(): ending
```

Synchronized Blocks

Similarly, a block of code can be synchronized. This operates very similarly to synchronizing a method.

```
1   public class MyClass extends Thread  {
2      ...
3      public void run() {
4         myObj.foo(name);
5      }
6   }
7   public class MyObject {
8      public void foo(String name) {
9         synchronized(this) {
10           ...
11        }
12     }
```

```
13 }
```

Like synchronizing a method, only one thread per instance of `MyObject` can execute the code within the `synchronized` block. That means that, if `thread1` and `thread2` have the same instance of `MyObject`, only one will be allowed to execute the code block at a time.

Locks

For more granular control, we can utilize a lock. A lock (or monitor) is used to synchronize access to a shared resource by associating the resource with the lock. A thread gets access to a shared resource by first acquiring the lock associated with the resource. At any given time, at most one thread can hold the lock and, therefore, only one thread can access the shared resource.

A common use case for locks is when a resource is accessed from multiple places, but should be only accessed by one thread *at a time*. This case is demonstrated in the code below.

```
1  public class LockedATM {
2     private Lock lock;
3     private int balance = 100;
4
5     public LockedATM() {
6        lock = new ReentrantLock();
7     }
8
9     public int withdraw(int value) {
10       lock.lock();
11       int temp = balance;
12       try {
13          Thread.sleep(100);
14          temp = temp - value;
15          Thread.sleep(100);
16          balance = temp;
17       } catch (InterruptedException e) {      }
18       lock.unlock();
19       return temp;
20    }
21
22    public int deposit(int value) {
23       lock.lock();
24       int temp = balance;
25       try {
26          Thread.sleep(100);
27          temp = temp + value;
28          Thread.sleep(300);
29          balance = temp;
30       } catch (InterruptedException e) {      }
31       lock.unlock();
```

```
32        return temp;
33    }
34 }
```

Of course, we've added code to intentionally slow down the execution of `withdraw` and `deposit`, as it helps to illustrate the potential problems that can occur. You may not write code exactly like this, but the situation it mirrors is very, very real. Using a lock will help protect a shared resource from being modified in unexpected ways.

Deadlocks and Deadlock Prevention

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds (or an equivalent situation with several threads). Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever. The threads are said to be deadlocked.

In order for a deadlock to occur, you must have all four of the following conditions met:

1. *Mutual Exclusion:* Only one process can access a resource at a given time. (Or, more accurately, there is limited access to a resource. A deadlock could also occur if a resource has limited quantity.)
2. *Hold and Wait:* Processes already holding a resource can request additional resources, without relinquishing their current resources.
3. *No Preemption:* One process cannot forcibly remove another process' resource.
4. *Circular Wait:* Two or more processes form a circular chain where each process is waiting on another resource in the chain.

Deadlock prevention entails removing any of the above conditions, but it gets tricky because many of these conditions are difficult to satisfy. For instance, removing #1 is difficult because many resources can only be used by one process at a time (e.g., printers). Most deadlock prevention algorithms focus on avoiding condition #4: circular wait.

Interview Questions

16.1 What's the difference between a thread and a process?

pg 416

16.2 How would you measure the time spent in a context switch?

pg 416

16.3 In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

pg 418

16.4 Design a class which provides a lock only if there are no possible deadlocks.

pg 420

16.5 Suppose we have the following code:

```
public class Foo {
   public Foo() { ... }
   public void first() { ... }
   public void second() { ... }
   public void third() { ... }
}
```

The same instance of `Foo` will be passed to three different threads. `ThreadA` will call `first`, `threadB` will call `second`, and `threadC` will call `third`. Design a mechanism to ensure that `first` is called before `second` and `second` is called before `third`.

pg 425

16.6 You are given a class with synchronized method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time? Can they execute A and B at the same time?

pg 427

Threads and Locks

Knowledge Based: Solutions

Chapter 16

16.1 *What's the difference between a thread and a process?*

pg 159

SOLUTION

Processes and threads are related to each other but are fundamentally different.

A process can be thought of as an instance of a program in execution. A process is an independent entity to which system resources (e.g., CPU time and memory) are allocated. Each process is executed in a separate address space, and one process cannot access the variables and data structures of another process. If a process wishes to access another process' resources, inter-process communications have to be used. These include pipes, files, sockets, and other forms.

A thread exists within a process and shares the process' resources (including its heap space). Multiple threads within the same process will share the same heap space. This is very different from processes, which cannot directly access the memory of another process. Each thread still has its own registers and its own stack, but other threads can read and write the heap memory.

A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads.

16.2 *How would you measure the time spent in a context switch?*

pg 159

SOLUTION

This is a tricky question, but let's start with a possible solution.

A context switch is the time spent switching between two processes (i.e., bringing a waiting process into execution and sending an executing process into waiting/terminated state). This happens in multitasking. The operating system must bring the state information of waiting processes into memory and save the state information of the currently running process.

In order to solve this problem, we would like to record the timestamps of the last and first instruction of the swapping processes. The context switch time is the difference in the timestamps between the two processes.

Let's take an easy example: Assume there are only two processes, P_1 and P_2.

P_1 is executing and P_2 is waiting for execution. At some point, the operating system must swap P_1 and P_2—let's assume it happens at the Nth instruction of P_1. If $t_{x,k}$ indicates the timestamp in microseconds of the kth instruction of process x, then the context switch would take $t_{2,1} - t_{1,n}$ microseconds.

The tricky part is this: how do we know when this swapping occurs? We cannot, of

course, record the timestamp of every instruction in the process.

Another issue is that swapping is governed by the scheduling algorithm of the operating system and there may be many kernel level threads which are also doing context switches. Other processes could be contending for the CPU or the kernel handling interrupts. The user does not have any control over these extraneous context switches. For instance, if at time $t_{1,n}$ the kernel decides to handle an interrupt, then the context switch time would be overstated.

In order to overcome these obstacles, we must first construct an environment such that after P_1 executes, the task scheduler immediately selects P_2 to run. This may be accomplished by constructing a data channel, such as a pipe, between P_1 and P_2 and having the two processes play a game of ping-pong with a data token.

That is, let's allow P_1 to be the initial sender and P_2 to be the receiver. Initially, P_2 is blocked (sleeping) as it awaits the data token. When P_1 executes, it delivers the token over the data channel to P_2 and immediately attempts to read a response token. However, since P_2 has not yet had a chance to run, no such token is available for P_1 and the process is blocked. This relinquishes the CPU.

A context switch results and the task scheduler must select another process to run. Since P_2 is now in a ready-to-run state, it is a desirable candidate to be selected by the task scheduler for execution. When P_2 runs, the roles of P_1 and P_2 are swapped. P_2 is now acting as the sender and P_1 as the blocked receiver. The game ends when P_2 returns the token to P_1.

To summarize, an iteration of the game is played with the following steps:

1. P_2 blocks awaiting data from P_1.
2. P_1 marks the start time.
3. P_1 sends token to P_2.
4. P_1 attempts to read a response token from P_2. This induces a context switch.
5. P_2 is scheduled and receives the token.
6. P_2 sends a response token to P_1.
7. P_2 attempts read a response token from P_1. This induces a context switch.
8. P_1 is scheduled and receives the token.
9. P_1 marks the end time.

The key is that the delivery of a data token induces a context switch. Let T_d and T_r be the time it takes to deliver and receive a data token, respectively, and let T_c be the amount of time spent in a context switch. At step 2, P_1 records the timestamp of the delivery of the token, and at step 9, it records the timestamp of the response. The amount of time elapsed, T, between these events may be expressed by:

$$T = 2 * (T_d + T_c + T_r)$$

This formula arises because of the following events: P_1 sends a token (3), the CPU context switches (4), P_2 receives it (5). P_2 then sends the response token (6), the CPU context switches (7), and finally P_1 receives it (8).

P_1 will be able to easily compute T, since this is just the time between events 3 and 8. So, to solve for T_c, we must first determine the value of $T_d + T_n$.

How can we do this? We can do this by measuring the length of time it takes P_1 to send and receive a token to itself. This will not induce a context switch since P_1 is running on the CPU at the time it sent the token and will not block to receive it.

The game is played a number of iterations to average out any variability in the elapsed time between steps 2 and 9 that may result from unexpected kernel interrupts and additional kernel threads contending for the CPU. We select the smallest observed context switch time as our final answer.

However, all we can ultimately say that this is an approximation which depends on the underlying system. For example, we make the assumption that P_2 is selected to run once a data token becomes available. However, this is dependent on the implementation of the task scheduler and we cannot make any guarantees.

That's okay; it's important in an interview to recognize when your solution might not be perfect.

16.3 *In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.*

pg 160

SOLUTION

First, let's implement a simple simulation of the dining philosophers problem in which we don't concern ourselves with deadlocks. We can implement this solution by having `Philosopher` extend `Thread`, and `Chopstick` call `lock.lock()` when it is picked up and `lock.unlock()` when it is put down.

```
1  public class Chopstick {
2     private Lock lock;
3
4     public Chopstick() {
5        lock = new ReentrantLock();
6     }
7
8     public void pickUp() {
9        void lock.lock();
```

```
    }

    public void putDown() {
        lock.unlock();
    }
}

public class Philosopher extends Thread {
    private int bites = 10;
    private Chopstick left;
    private Chopstick right;

    public Philosopher(Chopstick left, Chopstick right) {
        this.left = left;
        this.right = right;
    }

    public void eat() {
        pickUp();
        chew();
        putDown();
    }

    public void pickUp() {
        left.pickUp();
        right.pickUp();
    }

    public void chew() { }

    public void putDown() {
        left.putDown();
        right.putDown();
    }

    public void run() {
        for (int i = 0; i < bites; i++) {
            eat();
        }
    }
}
```

Running the above code may lead to a deadlock if all the philosophers have a left chopstick and are waiting for the right one.

To prevent deadlocks, we can implement a strategy where a philosopher will put down his left chopstick if he is unable to obtain the right one.

```
public class Chopstick {
    /* same as before */

```

```
    public boolean pickUp() {
      return lock.tryLock();
    }
}

public class Philosopher extends Thread {
  /* same as before */

  public void eat() {
    if (pickUp()) {
      chew();
      putDown();
    }
  }

  public boolean pickUp() {
    /* attempt to pick up */
    if (!left.pickUp()) {
      return false;
    }
    if (!right.pickUp()) {
      left.putDown();
      return false;
    }
    return true;
  }
}
```

In the above code, we need to be sure to release the left chopstick if we can't pick up the right one—and to not call `putDown()` on the chopsticks if we never had them in the first place.

16.4 *Design a class which provides a lock only if there are no possible deadlocks.*

pg 160

SOLUTION

There are several common ways to prevent deadlocks. One of the popular ways is to require a process to declare upfront what locks it will need. We can then verify if a deadlock would be created by issuing these locks, and we can fail if so.

With these constraints in mind, let's investigate how we can detect deadlocks. Suppose this was the order of locks requested:

```
A = {1, 2, 3, 4}
B = {1, 3, 5}
C = {7, 5, 9, 2}
```

This may create a deadlock because we could have the following scenario:

```
A locks 2, waits on 3
B locks 3, waits on 5
C locks 5, waits on 2
```

We can think about this as a graph, where 2 is connected to 3, 3 is connected to 5, and 5 is connected to 2. A deadlock is represented by a cycle. An edge `(w, v)` exists in the graph if a process declares that it will request lock `v` immediately after lock `w`. For the earlier example, the following edges would exist in the graph: `(1, 2)`, `(2, 3)`, `(3, 4)`, `(1, 3)`, `(3, 5)`, `(7, 5)`, `(5, 9)`, `(9, 2)`. The "owner" of the edge does not matter.

This class will need a `declare` method, which threads and processes will use to declare what order they will request resources in. This declare method will iterate through the declare order, adding each contiguous pair of elements (v, w) to the graph. Afterwards, it will check to see if any cycles have been created. If any cycles have been created, it will backtrack, removing these edges from the graph, and then exit.

We just have one final component to discuss: how do we detect a cycle? We can detect a cycle by doing a depth first search through each connected component (i.e., each connected part of the graph). Complex algorithms exist to select all the connected components of a graph, but our work in this problem does not require this degree of complexity.

We know that if a cycle was created, one of our new edges must be to blame. Thus, as long as our depth first search touches all of these edges at some point, then we know that we have fully searched for a cycle.

The pseudocode for this special case cycle detection looks like this:

```
1  boolean checkForCycle(locks[] locks) {
2     touchedNodes = hash table(lock -> boolean)
3     initialize touchedNodes to false for each lock in locks
4     for each (lock x in process.locks) {
5        if (touchedNodes[x] == false) {
6           if (hasCycle(x, touchedNodes)) {
7              return true;
8           }
9        }
10    }
11    return false;
12 }
13
14 boolean hasCycle(node x, touchedNodes) {
15    touchedNodes[r] = true;
16    if (x.state == VISITING) {
17       return true;
18    } else if (x.state == FRESH) {
19       ... (see full code below)
20    }
21 }
```

In the above code, note that we may do several depth first searches, but touchedNodes is only initialized once. We iterate until all the values in touchedNodes are false.

The code below provides further details. For simplicity, we assume that all locks and processes (owners) are ordered sequentially.

```
1  public class LockFactory {
2    private static LockFactory instance;
3
4    private int numberOfLocks = 5; /* default */
5    private LockNode[] locks;
6
7    /* Maps from a process or owner to the order that the owner
8     * claimed it would call the locks in */
9    private Hashtable<Integer, LinkedList<LockNode>> lockOrder;
10
11   private LockFactory(int count) { ... }
12   public static LockFactory getInstance() { return instance; }
13
14   public static synchronized LockFactory initialize(int count) {
15     if (instance == null) instance = new LockFactory(count);
16     return instance;
17   }
18
19   public boolean hasCycle(
20         Hashtable<Integer, Boolean> touchedNodes,
21         int[] resourcesInOrder) {
22     /* check for a cycle */
23     for (int resource : resourcesInOrder) {
24       if (touchedNodes.get(resource) == false) {
25         LockNode n = locks[resource];
26         if (n.hasCycle(touchedNodes)) {
27           return true;
28         }
29       }
30     }
31     return false;
32   }
33
34   /* To prevent deadlocks, force the processes to declare upfront
35    * what order they will need the locks in. Verify that this
36    * order does not create a deadlock (a cycle in a directed
37    * graph) */
38   public boolean declare(int ownerId, int[] resourcesInOrder) {
39     Hashtable<Integer, Boolean> touchedNodes =
40       new Hashtable<Integer, Boolean>();
41
42     /* add nodes to graph */
43     int index = 1;
44     touchedNodes.put(resourcesInOrder[0], false);
```

```
45        for (index = 1; index < resourcesInOrder.length; index++) {
46           LockNode prev = locks[resourcesInOrder[index - 1]];
47           LockNode curr = locks[resourcesInOrder[index]];
48           prev.joinTo(curr);
49           touchedNodes.put(resourcesInOrder[index], false);
50        }
51
52        /* if we created a cycle, destroy this resource list and
53         * return false */
54        if (hasCycle(touchedNodes, resourcesInOrder)) {
55           for (int j = 1; j < resourcesInOrder.length; j++) {
56              LockNode p = locks[resourcesInOrder[j - 1]];
57              LockNode c = locks[resourcesInOrder[j]];
58              p.remove(c);
59           }
60           return false;
61        }
62
63        /* No cycles detected. Save the order that was declared, so
64         * that we can verify that the process is really calling the
65         * locks in the order it said it would. */
66        LinkedList<LockNode> list = new LinkedList<LockNode>();
67        for (int i = 0; i < resourcesInOrder.length; i++) {
68           LockNode resource = locks[resourcesInOrder[i]];
69           list.add(resource);
70        }
71        lockOrder.put(ownerId, list);
72
73        return true;
74     }
75
76     /* Get the lock, verifying first that the process is really
77      * calling the locks in the order it said it would. */
78     public Lock getLock(int ownerId, int resourceID) {
79        LinkedList<LockNode> list = lockOrder.get(ownerId);
80        if (list == null) return null;
81
82        LockNode head = list.getFirst();
83        if (head.getId() == resourceID) {
84           list.removeFirst();
85           return head.getLock();
86        }
87        return null;
88     }
89  }
90
91  public class LockNode {
92     public enum VisitState { FRESH, VISITING, VISITED };
93
94     private ArrayList<LockNode> children;
```

```
95    private int lockId;
96    private Lock lock;
97    private int maxLocks;
98
99    public LockNode(int id, int max) { ... }
100
101   /* Join "this" to "node", checking to make sure that it doesn't
102    * create a cycle */
103   public void joinTo(LockNode node) { children.add(node); }
104   public void remove(LockNode node) { children.remove(node); }
105
106   /* Check for a cycle by doing a depth-first-search. */
107   public boolean hasCycle(
108         Hashtable<Integer, Boolean> touchedNodes) {
109      VisitState[] visited = new VisitState[maxLocks];
110      for (int i = 0; i < maxLocks; i++) {
111         visited[i] = VisitState.FRESH;
112      }
113      return hasCycle(visited, touchedNodes);
114   }
115
116   private boolean hasCycle(VisitState[] visited,
117         Hashtable<Integer, Boolean> touchedNodes) {
118      if (touchedNodes.containsKey(lockId)) {
119         touchedNodes.put(lockId, true);
120      }
121
122      if (visited[lockId] == VisitState.VISITING) {
123         /* We looped back to this node while still visiting it, so
124          * we know there's a cycle. */
125         return true;
126      } else if (visited[lockId] == VisitState.FRESH) {
127         visited[lockId] = VisitState.VISITING;
128         for (LockNode n : children) {
129            if (n.hasCycle(visited, touchedNodes)) {
130               return true;
131            }
132         }
133         visited[lockId] = VisitState.VISITED;
134      }
135      return false;
136   }
137
138   public Lock getLock() {
139      if (lock == null) lock = new ReentrantLock();
140      return lock;
141   }
142
143   public int getId() { return lockId; }
144 }
```

As always, when you see code this complicated and lengthy, you wouldn't be expected to write all of it. More likely, you would be asked to sketch out pseudocode and possibly implement one of these methods.

16.5 *Suppose we have the following code:*

```
public class Foo {
  public Foo() { ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

The same instance of Foo will be passed to three different threads. ThreadA will call first, threadB will call second, and threadC will call third. Design a mechanism to ensure that first is called before second and second is called before third.

pg 160

SOLUTION

The general logic is to check if `first()` has completed before executing `second()`, and if `second()` has completed before calling `third()`. Because we need to be very careful about thread safety, simple boolean flags won't do the job.

What about using a `lock` to do something like the below code?

```
1  public class FooBad {
2     public int pauseTime = 1000;
3     public ReentrantLock lock1, lock2, lock3;
4
5     public FooBad() {
6        try {
7           lock1 = new ReentrantLock();
8           lock2 = new ReentrantLock();
9           lock3 = new ReentrantLock();
10
11          lock1.lock();
12          lock2.lock();
13          lock3.lock();
14       } catch (...) { ... }
15    }
16
17    public void first() {
18       try {
19          ...
20          lock1.unlock(); // mark finished with first()
21       } catch (...) { ... }
```

```
22    }
23
24    public void second() {
25       try {
26          lock1.lock(); // wait until finished with first()
27          lock1.unlock();
28          ...
29
30          lock2.unlock(); // mark finished with second()
31       } catch (...) { ... }
32    }
33
34    public void third() {
35       try {
36          lock2.lock(); // wait until finished with third()
37          lock2.unlock();
38          ...
39       } catch (...) { ... }
40    }
41 }
```

This code won't actually quite work due to the concept of *lock ownership*. One thread is actually performing the lock (in the FooBad constructor), but different threads attempt to unlock the locks. This is not allowed, and your code will raise an exception. A lock in Java is owned by the same thread which locked it.

Instead, we can replicate this behavior with semaphores. The logic is identical.

```
1  public class Foo {
2     public Semaphore sem1, sem2, sem3;
3
4     public Foo() {
5        try {
6           sem1 = new Semaphore(1);
7           sem2 = new Semaphore(1);
8           sem3 = new Semaphore(1);
9
10          sem1.acquire();
11          sem2.acquire();
12          sem3.acquire();
13       } catch (...) { ... }
14    }
15
16    public void first() {
17       try {
18          ...
19          sem1.release();
20       } catch (...) { ... }
21    }
22
23    public void second() {
```

```
24        try {
25           sem1.acquire();
26           sem1.release();
27           ...
28           sem2.release();
29        } catch (...) { ... }
30     }
31
32     public void third() {
33        try {
34           sem2.acquire();
35           sem2.release();
36           ...
37        } catch (...) { ... }
38     }
39  }
```

16.6 *You are given a class with synchronized method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time? Can they execute A and B at the same time?*

pg 160

SOLUTION

By applying the word `synchronized` to a method, we ensure that two threads cannot execute synchronized methods *on the same object instance* at the same time.

So, the answer to the first part really depends. If the two threads have the same instance of the object, then no, they cannot simultaneously execute method A. However, if they have different instances of the object, then they can.

Conceptually, you can see this by considering locks. A synchronized method applies a "lock" on *all* synchronized methods in that instance of the object. This blocks other threads from executing synchronized methods within that instance.

In the second part, we're asked if `thread1` can execute synchronized method A while `thread2` is executing non-synchronized method B. Since B is not synchronized, there is nothing to block `thread1` from executing A while `thread2` is executing B. This is true regardless of whether `thread1` and `thread2` have the same instance of the object or not.

Ultimately, the key concept to remember here is that only one synchronized method can be in execution per instance of that object. Other threads can execute non-synchronized methods on that instance, or they can execute any method on a different instance of the object.