```
25
26    public void eat() {
27       pickUp();
28       chew();
29       putDown();
30    }
31
32    public void pickUp() {
33       left.pickUp();
34       right.pickUp();
35    }
36
37    public void chew() { }
38
39    public void putDown() {
40       right.putDown();
41       left.putDown();
42    }
43
44    public void run() {
45       for (int i = 0; i < bites; i++) {
46          eat();
47       }
48    }
49 }
```

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

Solution #1: All or Nothing

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.

```
1  public class Chopstick {
2     /* same as before */
3
4     public boolean pickUp() {
5        return lock.tryLock();
6     }
7  }
8
9  public class Philosopher extends Thread {
10    /* same as before */
11
12    public void eat() {
13       if (pickUp()) {
14          chew();
15          putDown();
16       }
17    }
18
19    public boolean pickUp() {
20       /* attempt to pick up */
21       if (!left.pickUp()) {
22          return false;
23       }
24       if (!right.pickUp()) {
```

```
25          left.putDown();
26          return false;
27        }
28        return true;
29      }
30    }
```

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.

One issue with this is that if all the philosophers were perfectly synchronized, they could simultaneously pick up their left chopstick, be unable to pick up the right one, and then put back down the left one—only to have the process repeated again.

Solution #2: Prioritized Chopsticks

Alternatively, we can label the chopsticks with a number from `0` to `N - 1`. Each philosopher attempts to pick up the lower numbered chopstick first. This essentially means that each philosopher goes for the left chopstick before right one (assuming that's the way you labeled it), except for the last philosopher who does this in reverse. This will break the cycle.

```
1   public class Philosopher extends Thread {
2     private int bites = 10;
3     private Chopstick lower, higher;
4     private int index;
5     public Philosopher(int i, Chopstick left, Chopstick right) {
6       index = i;
7       if (left.getNumber() < right.getNumber()) {
8         this.lower = left;
9         this.higher = right;
10      } else {
11        this.lower = right;
12        this.higher = left;
13      }
14    }
15
16    public void eat() {
17      pickUp();
18      chew();
19      putDown();
20    }
21
22    public void pickUp() {
23      lower.pickUp();
24      higher.pickUp();
25    }
26
27    public void chew() { ... }
28
29    public void putDown() {
30      higher.putDown();
31      lower.putDown();
32    }
33
34    public void run() {
35      for (int i = 0; i < bites; i++) {
36        eat();
```

```
37        }
38     }
39  }
40
41  public class Chopstick {
42     private Lock lock;
43     private int number;
44
45     public Chopstick(int n) {
46        lock = new ReentrantLock();
47        this.number = n;
48     }
49
50     public void pickUp() {
51        lock.lock();
52     }
53
54     public void putDown() {
55        lock.unlock();
56     }
57
58     public int getNumber() {
59        return number;
60     }
61  }
```

With this solution, a philosopher can never hold the larger chopstick without holding the smaller one. This prevents the ability to have a cycle, since a cycle means that a higher chopstick would "point" to a lower one.

15.4 Deadlock-Free Class: Design a class which provides a lock only if there are no possible deadlocks.

pg 180

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 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 find 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:

```
boolean checkForCycle(locks[] locks) {
   touchedNodes = hash table(lock -> boolean)
   initialize touchedNodes to false for each lock in locks
   for each (lock x in process.locks) {
      if (touchedNodes[x] == false) {
         if (hasCycle(x, touchedNodes)) {
            return true;
         }
      }
   }
   return false;
}

boolean hasCycle(node x, touchedNodes) {
   touchedNodes[r] = true;
   if (x.state == VISITING) {
      return true;
   } else if (x.state == FRESH) {
      ... (see full code below)
   }
}
```

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.

```
class LockFactory {
   private static LockFactory instance;

   private int numberOfLocks = 5; /* default */
   private LockNode[] locks;

   /* Maps from a process or owner to the order that the owner claimed it would
    * call the locks in */
   private HashMap<Integer, LinkedList<LockNode>> lockOrder;

   private LockFactory(int count) { ... }
   public static LockFactory getInstance() { return instance; }

   public static synchronized LockFactory initialize(int count) {
      if (instance == null) instance = new LockFactory(count);
```

```
16      return instance;
17    }
18
19    public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes,
20                            int[] resourcesInOrder) {
21      / *check for a cycle */
22      for (int resource : resourcesInOrder) {
23        if (touchedNodes.get(resource) == false) {
24          LockNode n = locks[resource];
25          if (n.hasCycle(touchedNodes)) {
26            return true;
27          }
28        }
29      }
30      return false;
31    }
32
33    / *To prevent deadlocks, force the processes to declare upfront what order they
34     * will need the locks in. Verify that this order does not create a deadlock (a
35     * cycle in a directed graph) */
36    public boolean declare(int ownerId, int[] resourcesInOrder) {
37      HashMap<Integer, Boolean> touchedNodes = new HashMap<Integer, Boolean>();
38
39      / *add nodes to graph */
40      int index = 1;
41      touchedNodes.put(resourcesInOrder[0], false);
42      for (index = 1; index < resourcesInOrder.length; index++) {
43        LockNode prev = locks[resourcesInOrder[index - 1]];
44        LockNode curr = locks[resourcesInOrder[index]];
45        prev.joinTo(curr);
46        touchedNodes.put(resourcesInOrder[index], false);
47      }
48
49      / *if we created a cycle, destroy this resource list and return false */
50      if (hasCycle(touchedNodes, resourcesInOrder)) {
51        for (int j = 1; j < resourcesInOrder.length; j++) {
52          LockNode p = locks[resourcesInOrder[j - 1]];
53          LockNode c = locks[resourcesInOrder[j]];
54          p.remove(c);
55        }
56        return false;
57      }
58
59      / *No cycles detected. Save the order that was declared, so that we can
60       * verify that the process is really calling the locks in the order it said
61       * it would. */
62      LinkedList<LockNode> list = new LinkedList<LockNode>();
63      for (int i = 0; i < resourcesInOrder.length; i++) {
64        LockNode resource = locks[resourcesInOrder[i]];
65        list.add(resource);
66      }
67      lockOrder.put(ownerId, list);
68
69      return true;
70    }
71
```

```
72    /* Get the lock, verifying first that the process is really calling the locks in
73     * the order it said it would. */
74    public Lock getLock(int ownerId, int resourceID) {
75       LinkedList<LockNode> list = lockOrder.get(ownerId);
76       if (list == null) return null;
77
78       LockNode head = list.getFirst();
79       if (head.getId() == resourceID) {
80          list.removeFirst();
81          return head.getLock();
82       }
83       return null;
84    }
85 }
86
87 public class LockNode {
88    public enum VisitState { FRESH, VISITING, VISITED };
89
90    private ArrayList<LockNode> children;
91    private int lockId;
92    private Lock lock;
93    private int maxLocks;
94
95    public LockNode(int id, int max) { ... }
96
97    /* Join "this" to "node", checking that it doesn't create a cycle */
98    public void joinTo(LockNode node) { children.add(node); }
99    public void remove(LockNode node) { children.remove(node); }
100
101   /* Check for a cycle by doing a depth-first-search. */
102   public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes) {
103      VisitState[] visited = new VisitState[maxLocks];
104      for (int i = 0; i < maxLocks; i++) {
105         visited[i] = VisitState.FRESH;
106      }
107      return hasCycle(visited, touchedNodes);
108   }
109
110   private boolean hasCycle(VisitState[] visited,
111                            HashMap<Integer, Boolean> touchedNodes) {
112      if (touchedNodes.containsKey(lockId)) {
113         touchedNodes.put(lockId, true);
114      }
115
116      if (visited[lockId] == VisitState.VISITING) {
117         /* We looped back to this node while still visiting it, so we know there's
118          * a cycle. */
119         return true;
120      } else if (visited[lockId] == VisitState.FRESH) {
121         visited[lockId] = VisitState.VISITING;
122         for (LockNode n : children) {
123            if (n.hasCycle(visited, touchedNodes)) {
124               return true;
125            }
126         }
127         visited[lockId] = VisitState.VISITED;
```

```
128       }
129       return false;
130    }
131
132    public Lock getLock() {
133       if (lock == null) lock = new ReentrantLock();
134       return lock;
135    }
136
137    public int getId() { return lockId; }
138 }
```

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.

15.5 Call In Order: 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 180

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

```
21
22    public void second() {
23       try {
24          lock1.lock(); // wait until finished with first()
25          lock1.unlock();
26          ...
27
28          lock2.unlock(); // mark finished with second()
29       } catch (...) { ... }
30    }
31
32    public void third() {
33       try {
34          lock2.lock(); // wait until finished with third()
35          lock2.unlock();
36          ...
37       } catch (...) { ... }
38    }
39 }
```

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;
3
4     public Foo() {
5        try {
6           sem1 = new Semaphore(1);
7           sem2 = new Semaphore(1);
8
9           sem1.acquire();
10          sem2.acquire();
11       } catch (...) { ... }
12    }
13
14    public void first() {
15       try {
16          ...
17          sem1.release();
18       } catch (...) { ... }
19    }
20
21    public void second() {
22       try {
23          sem1.acquire();
24          sem1.release();
25          ...
26          sem2.release();
27       } catch (...) { ... }
28    }
29
30    public void third() {
31       try {
32          sem2.acquire();
```

```
33          sem2.release();
34          ...
35        } catch (...) { ... }
36    }
37  }
```

15.6 Synchronized Methods: 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 180

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.

Ultimately, the key concept to remember 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.

15.7 FizzBuzz: In the classic problem FizzBuzz, you are told to print the numbers from 1 to n. However, when the number is divisible by 3, print "Fizz". When it is divisible by 5, print "Buzz". When it is divisible by 3 and 5, print "FizzBuzz". In this problem, you are asked to do this in a multithreaded way. Implement a multithreaded version of FizzBuzz with four threads. One thread checks for divisibility of 3 and prints "Fizz". Another thread is responsible for divisibility of 5 and prints "Buzz". A third thread is responsible for divisibility of 3 and 5 and prints "FizzBuzz". A fourth thread does the numbers.

pg 180

SOLUTION

Let's start off with implementing a single threaded version of FizzBuzz.

Single Threaded

Although this problem (in the single threaded version) shouldn't be hard, a lot of candidates overcomplicate it. They look for something "beautiful" that reuses the fact that the divisible by 3 and 5 case ("FizzBuzz") seems to resemble the individual cases ("Fizz" and "Buzz").

In actuality, the best way to do it, considering readability and efficiency, is just the straightforward way.

```
1  void fizzbuzz(int n) {
```

```
2     for (int i = 1; i <= n; i++) {
3       if (i % 3 == 0 && i % 5 == 0) {
4         System.out.println("FizzBuzz");
5       } else if (i % 3 == 0) {
6         System.out.println("Fizz");
7       } else if (i % 5 == 0) {
8         System.out.println("Buzz");
9       } else {
10        System.out.println(i);
11      }
12    }
13  }
```

The primary thing to be careful of here is the order of the statements. If you put the check for divisibility by 3 before the check for divisibility by 3 and 5, it won't print the right thing.

Multithreaded

To do this multithreaded, we want a structure that looks something like this:

FizzBuzz Thread	Fizz Thread

if i div by 3 && 5
  print FizzBuzz
  increment i
repeat until i > n
	if i div by only 3
	  print Fizz
	  increment i
	repeat until i > n

Buzz Thread	Number Thread

if i div by only 5
  print Buzz
  increment i
repeat until i > n
	if i not div by 3 or 5
	  print i
	  increment i
	repeat until i > n

The code for this will look something like:

```
1   while (true) {
2     if (current > max) {
3       return;
4     }
5     if (/* divisibility test */) {
6       System.out.println(/* print something */);
7       current++;
8     }
9   }
```

We'll need to add some synchronization in the loop. Otherwise, the value of `current` could change between lines 2 - 4 and lines 5 - 8, and we can inadvertently exceed the intended bounds of the loop. Additionally, incrementing is not thread-safe.

To actually implement this concept, there are many possibilities. One possibility is to have four entirely separate thread classes that share a reference to the current variable (which can be wrapped in an object).

The loop for each thread is substantially similar. They just have different target values for the divisibility checks, and different print values.