The "missed wakeup problem" occurs when a thread calls an internal sleep() primitive to block, and another thread calls wakeup() to awaken the sleeping thread in an unsafe fashion. For example, consider the following pseudocode snippets for two threads:

**Sleeper thread**

```java
Thread sleeper = self();
listMx.lock();
list.put(sleeper);
listMx.unlock();
sleeper.sleep();
```

**Waker thread**

```java
listMx.lock();
Thread sleeper = list.get();
listMx.unlock();
sleeper.wakeup();
```

(a) What could go wrong? Outline how this code is vulnerable to the missed wakeup problem, and illustrate with an example schedule.
(b) How does blocking with monitors (condition variables) avoid the missed wakeup problem? Illustrate how the code snippets in (a) might be implemented using monitors, and outline why it works.

(c) Now we want to design a scheme that is safe from the missed wakeup problem, but using semaphores only. The first step is to implement locks (i.e., mutexes such as the listMx in the snippets of (a)). Implement locks using semaphores. As always: “any kind of pseudocode is fine, as long as its meaning is clear.”
(d) Next implement sleep() and wakeup() primitives using semaphores. These primitives are used as in the code snippets in part (1a) above. Note that sleep() and wakeup() operate on a specific thread. Your implementation should be “safe” in that it is not vulnerable to the missed wakeup problem.
Part 2. Piling on the heap, again (60 points)

For Lab #1 you built a heap manager for use in single-threaded processes. This question addresses the problem of how to adapt a heap manager for use in a multi-threaded process, in which multiple threads may invoke the heap manager API (malloc/freeNode) concurrently. You may answer with reference to your code for Lab #1, or to an idealized heap manager implementation (one that works).

(a) If you use the heap manager with no added concurrency control, what could go wrong? Give three examples. Please be brief but specific. Feel free to illustrate with pseudocode or drawings.
(b) Outline a concurrency control scheme for your heap manager and briefly discuss its tradeoffs. You may illustrate with pseudocode if you wish. Be sure to address the following questions. How many locks? What data is covered by the locks? What are the performance implications for your locking scheme? Do you need any condition variables? If so, how would you use them, i.e., under what conditions would a thread wait or signal? Is deadlock or starvation a concern? Does your scheme require any changes to the heap abstraction, i.e., any changes to the API or to a program that uses the heap manager?
Part 3. Sharing the plumbing (60 points)

The Computer Science building at the Old School was built for a bunch of men. But today many of the computer scientists are women. Unfortunately, the building has only one restroom. In keeping with the Old Traditions, the community has decided to coordinate use of the restroom by the following policy: the restroom may be visited concurrently by up to N individuals, provided they are of the same gender (M or F).

This problem asks you to implement a module to coordinate use of the restroom, for use in a simulation. Each individual is represented by a thread. A thread requests use of the restroom by calling one of the methods arriveM() or arriveF(), according to its gender. A thread waits in its arrive*() method until the restroom is available for its use. A thread departs the restroom by calling the method departM() or departF(). All threads are well-behaved.

Implement pseudocode for your coordination scheme. Be sure your solution is free of races and avoids starvation and deadlock.