Monitors and Semaphores – review and practice

Semaphores

A semaphore is an object with an initial value N and two methods: down() and up(). Calls to down() can block, while calls to up() do not block.

Note that you don't implement semaphores - you instantiate them:

\[ S = \text{semaphore}(N) \] -- create a semaphore with initial value N

Semaphore behavior is defined thusly:

Given a semaphore \( S \) with initial value \( N \): at any point in time, if \( M \) threads have called \( S.\text{up()} \), then no more than \( M+N \) threads have returned from \( S.\text{down()} \).

Alternately we can describe this in terms of a semaphore count:

\[
\begin{align*}
S.\text{down}(): & \quad \text{if count is 0} \\
& \quad \text{wait until woken up} \\
& \quad \text{count} = \text{count} - 1
\end{align*}
\]

\[
\begin{align*}
S.\text{up}(): & \quad \text{increment count} \\
& \quad \text{if any threads are waiting} \\
& \quad \text{wake the first one up}
\end{align*}
\]

This is only a description; in particular, since the only semaphore methods are down() and up() there is no way to read the value of the semaphore count. [e.g. The description given in Downey handles the 'count' variable differently, but behaves identically]

Several things to remember, from Downey:

- In general, there is no way to know before a thread decrements a semaphore whether it will block or not. [not even if you add a method to examine the count, as another thread might enter between when you checked and when you called down()]
- After a thread increments a semaphore and another thread gets woken up, both threads continue running concurrently. There is no way to know which thread, if either, will continue immediately.
- When you signal a semaphore, you don’t necessarily know whether another thread is waiting, so the number of unblocked threads may be zero or one.
**Monitors**

A monitor is a particular kind of user-defined class, which has:

- regular variables
- methods
- conditions --- this is what normal classes don't have

For this class we will use the following pseudo-code syntax:

```pseudo
monitor <typename> { 
<type> <variable name> 
... 
condition <condition name> 
... 
<type> <method-name>(<arguments>) { 
  code...
  <condition variable>.wait()
  <condition variable>.signal()
  <condition variable>.broadcast()
 }
}
```

And we will use the following behavior definition, basically what is called "Mesa semantics":

Threads can:

- enter the monitor by calling any of the methods
- leave the monitor by returning from a method
- leave the monitor by calling wait()
- re-enter the monitor by returning from wait()

If there are any threads waiting on condition C:

- C.signal() will release one of them and
- C.broadcast() will release all of them.
- If there are no threads waiting, neither C.signal() nor C.broadcast() have any effect.

Only one thread can be *in the monitor* (i.e. entered but not left) at any given time. If thread A is in the monitor and thread B tries to (a) call a method or (b) return from wait(), thread B will block until thread A leaves the monitor. (or longer – see below)

When a thread calls signal() or broadcast() it does not leave the monitor, so that you can be sure that it will continue to run before any thread that it woke up.

**However**, if you wake thread A via signal() or broadcast(), and thread B tries to enter the monitor by calling a method, you don't know whether A or B will get to run first.
Question 1.

An operating system has a sys_read() system call which (for the purposes of writing pseudo-code) we can assume takes two arguments - an offset and a length - and returns a data buffer.

Create a function new_read() which:

- takes 1 argument, length
- reads <length> bytes immediately following those returned by the previous read

with the additional properties that:

- up to 4 threads are allowed to be calling sys_read() at once
- calls must return in order - i.e. if thread A calls new_read() before thread B, then thread A must return from new_read() before B. (even if thread B returns from sys_read() first)

Implement new_read():

a) With semaphores. Your answer will have some global variables (semaphores and regular variables) and a single function new_read() which uses these global variables and its own local variables.

b) as a monitor.

Question 2.

This is the same as question 1, except that instead of allowing 4 sys_read() requests to be outstanding, we allow requests totalling 10,000 bytes to be outstanding. (assume that no single request is larger than 10,000 bytes)

Implement as a monitor.

Question 3.

A barbershop consists of a waiting room with N chairs, and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs in the waiting room are occupied, then the customer leaves the shop. If the barber is busy, but chairs in the waiting room are available, then he/she sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

Implement a monitor with 2 methods, customer() and barber(). At runtime a single thread will call the barber() method, which loops forever, and then threads 2,3,... will arrive at various times and call the customer() method. The barber() method will use the function cut_hair(), which leaves the monitor and takes an arbitrary amount of time to return.