

COSC 301: Operating Systems

Lab 6: Locking and condition variable problems

1. A number of threads periodically call into the following routine, to make sure that a pipe that is shared between them has already been opened (after calling this routine, a thread might go ahead and call `write()` on that pipe, for example). Assume there is a global integer `pipe`, which is set to -1 when the pipe is closed, and a global lock `lock`, which is used for synchronization. Here is the code:

```
void
MakeSurePipeIsOpen() {
    mutex_lock(&lock);
    if (pipe == -1) {
        pipe = open("/tmp/fifo", O_WRONLY);
    }
    mutex_unlock(&lock);
}
```

However, you get clever, and decide to re-write the code as follows:

```
void
MakeSurePipeIsOpen() {
    if (pipe == -1) {
        mutex_lock(&lock);
        if (pipe == -1) {
            pipe = open("/tmp/fifo", O_WRONLY);
        }
        mutex_unlock(&lock);
    }
}
```

Does this code still work correctly? If so, what advantage do we gain by using this implementation? If not, why doesn't it work?

2. Assume you are implementing a producer-consumer shared linked-list (which can be used by 1 or more producer threads to pass data to 1 or more consumer/worker threads). Note that a linked list is *unbounded*, so this problem is (potentially) a bit different than the situation with a bounded buffer (array) and organizing access between producer and consumer threads.

- How many condition variables will you need in order to implement this buffer properly, and why?
 - How is this different than a standard bounded buffer implementation?
-

3. Imagine a new synchronization primitive called the `WhatsItFor`. A `WhatsItFor` has an initial value (which is initialized by the user), and two related routines, `One()` and `Done()`, that work as follows:

- `One()` waits for the value of the `WhatsItFor` to be less than zero, and then increments the value by one.
- `Done()` decrements the `WhatsItFor` by one, and then wakes one waiting thread (if there is one).

- Both `One()` and `Done()` execute atomically.

Show how to use a `WhatsItFor` (specifically, `One()` and `Done()`) to build a simple lock around a critical section. Make sure to specify the initial value of the `WhatsItFor`.

4. Consider the following pseudocode segments P1 and P2, which would be executed by two different threads.

```
P1: {
    shared int x;
    x = 10;
    while (1) {
        x = x - 1;
        x = x + 1;
        if (x != 10)
            printf("x is %d",x)
    }
}

P2: {
    shared int x;
    x = 10;
    while ( 1 ) {
        x = x - 1;
        x = x + 1;
        if (x != 10)
            printf("x is %d",x)
    }
}
```

Note that the scheduler in a uniprocessor system would implement pseudo- parallel execution of these two concurrent processes by interleaving their instructions, without restriction on the order of the interleaving.

- Show a sequence (i.e., trace the sequence of interleavings of statements) such that the statement “x is 10” is printed.
- Show a sequence such that the statement “x is 8” is printed. You should remember that the increment/decrements at the source language level are not done atomically, i.e., the assembly language code:

```
LD      R0,X /* load R0 from memory location x */
INCR    R0 /* increment R0 */
STO     R0,X /* store the incremented value back in X */
implements the single C increment instruction (x = x + 1).
```

- Show where/how to add mutexes to the program in the preceding problem to insure that the `printf()` is never executed. Your solution should allow as much concurrency as possible.