CPSC 213

SOLUTIONS TO PRACTICE QUESTION SET 2

Question 1

a. There is an infinite loop because baz (address stored in r3) is set to 10 and thus r0 is never equal to 0, so the line of code at 'below' is never reached.

b. A second thread that also writes to the shared variable baz could be called at any time during the 'above' loop (or even after executing line 4 before executing line 5 to begin the loop), and sets baz to 0, then when control is returned to the first thread, the loop would end and the variable baz would get set to 42.

Question 2

a. Yes. It is possible that the scheduler scheduled threads A, B, C in that order and before letting main executing `printf ("D").`
b. Yes, same logic as above.
c. Yes, in this case the scheduler decided to continue with main before running the other threads.
d. Yes, same logic as above.
e. No. The join commands mean that the main program will not exit until after all of the child threads finish, so there will be four characters printed out before the program ends.

Question 3

a. No. D is always printed before any other letter, since that print statement occurs before any other threads are started.

b. No. The ordering of GHIJ is fixed, since those are called sequentially in the main thread. J is always printed last.

c. Yes. Order of C vs B vs A is not predictable.

d. No. D always first, and I must print before J

e. (blank) No. join commands mean program will not exit until after all of the child threads finish, so ten characters printed always.

f. Yes. The A/B/C might print immediately after the new threads are created, before the main thread gets to the join call.

g. Yes. The GHIJ can be interleaved with ABC.

h. No. Because of the order of the join calls, A must print before H, B before I, and C before J.

Question 4

RUNNING to BLOCKED:

- Attempting to enter a lock that is currently in the HELD state.
- Waiting on a condition variable will cause a running thread to block.

BLOCKED to RUNNING:

- Unlocking the lock on which another thread is waiting to get will awaken that waiting thread.
- Signaling (or notifying) a condition variable on which a thread is waiting will awaken that thread.
- Broadcasting (or notifying all) on a condition variable on which one or more threads are waiting will awaken all of these threads.
- Awoken threads are set to the RUNNABLE state. For a thread to transition to the RUNNING state it must then be scheduled (i.e., taken from the ready queue and switched with the thread that is currently running).

Question 5

a. Spinlocks use busy waiting. Monitors use blocking waiting.

b. Yes. Spinlocks have lower overhead than monitors (in fact, monitors are implemented using spinlocks) and so if the critical section is small and contention for the spinlock is low, then it is better to use a spinlock than a monitor.

c. Yes. Spinlocks waste CPU resources while waiting and so if the critical section is long or contention for the monitor is high, then it is better to use a monitor.

Question 6

B is preferred because bitwise operations execute much faster in hardware than division.

Question 7

```
int balance;
pthread_mutex_t mut;
pthread_cond_t cv;

pthread_mutex_init( &mut, NULL);
pthread_cond_init( &cv, NULL);

void deposit (int amt) {
     pthread_mutex_lock(&mut);
     balance += amt;
     pthread_cond_broadcast(&cv);
     pthread_ mutex_unlock(&mut);
}

void withdraw (int amt) {
     pthread_mutex_lock(&mut);
     while (balance <= amt)
          pthread_cond_wait(&cv, &mut);
     balance -= amt;
     pthread_ mutex_unlock(&mut);
}
```

Question 8

Polling I/O controllers is a bad idea since the cost of checking whether they are ready could be very high and cannot be performed during each cycle. Then there is also a problem of how frequently the polling should be.

Polling interrupts is safe because it requires only a check of a dedicated register. Checking a register values is fast enough to do at the start of every clock cycle, and still leave time for the fetch/execute part.

Question 9

The TCB does not need any other register values besides the stack pointer, because they are all saved on the stack for that thread. It does need the stack pointer because there is no other way to know where in the stack those registers are stored.

Question 10

a) This is deadlock free. A deadlock can occur only when each one of the two threads helds one of the locks and is waiting for the other lock. In this case, it is not possible for each thread to hold one lock. The thread that will acquire the lock s will also acquire the lock t.

b) Same as (a). One thread will hold both locks before it releases them. Therefore there is no possibility for deadlock.

c) No deadlock in this case. Each thread releases the lock that holds before it tries to acquire another lock. No matter how the threads are scheduled, each thread will always be able to release the locks that it holds.

d) Deadlock is possible. Here is a schedule that leads to deadlock: