CPSC 213, Winter 2013, Term 2 — Quiz 2 Sample Questions **Solution**

Date: March 2014; Instructor: Mike Feeley

1 **(10 marks)** Consider the following C code.

```
void x () { printf ("x"); }          void foo () {
void y () { printf ("y"); }            proc = z;
void z () { printf ("z"); }            proc ();
void (*proc)();                      }
```

1a Explain in a single, simple plain-English sentence what the user sees when `foo()` executes.

It prints ”z”.

1b Give commented SM213 assembly code that implements the statement `proc()`.

```
  ld $proc, r0   # r0 = &proc
  gpc $2, r6     # r6 = return address
  j *(r0)        # goto proc (pc <= m[r[0]])
```

2 **(7 marks)** A `switch` statement can be implemented using a jump table or a sequence of `if` statements.

2a Explain the benefit of using jump tables (just the benefit; not how they work).

The running time of the switch implement with jump tables is independent of the number of case arms, unlike the alternative.

2b Are jump tables always the best implementation choice for every switch statement? Explain carefully.

No. If case labels are sparse, the jump table becomes too large for this approach to be feasible.

2c Give the SM213 assembly-language implementation using jump tables of the following C code where `i` and `j` are global `int` variables.

```
switch (i) {
    case 10: j=3; break;
    case 12: j=4; break;
    default: j=5; break;
}
```

```
# code for switch statement
            ld $i, r0             # r0 = &i
            ld (r0), r0           # r0 = i
            ld $j, r1             # r1 = &j
            ld $-10, r2           # r2 = -10
            add r2, r0            # r0 = i-10
            bgt r0, L0            # goto l0 if i > 10
            beq r0, L0            # goto l0 if i == 10
            br default            # goto default if i < 10
L0:         ld $-2, r2            # r2 = -2
            add r0, r2            # r2 = ((i-10) - 2) = i-12
            bgt r2, default       # goto default if i > 12
            ld $jt, r2            # r2 = &jt
            j *(r2, r0, 4)        # goto jt [i-10]
case10:     ld $3, r0             # r0 = 3
            st r0, (r1)           # j = 3
            br L1                 # break
case12:     ld $4, r0             # r0 = 4
            st r0, (r1)           # j = 4
            br L1                 # break
default:    ld $5, r0             # r0 = 5
            st r0, (r1)           # j=5
L1:                               # continue...

# in global, static data part of memory
jt:         .long case10
            .long default
            .long case12
```

3 **(10 marks)** Consider a program that reads a block of data from a disk without using threads. Doing so will involve these four things: programmed IO, DMA, interrupts, and asynchronous programming. Carefully explain the role that each of these play by providing a step-by-step timeline of the disk read starting with the program's request for a block and ending with the program doing something with the block.

1. The program adds a pointer to a completion routine for this request the disk's pending-request queue.
2. The program then uses PIO to send its read request to the disk's IO controller.
3. The program must now do other things that do not use the value of the disk block being read. This code is executes asynchronously with the disk read operation.
4. In parallel with step 3, the disk controller processes the read to retrieve the block from the disk and to then use DMA to transfer its value into main memory and to send an interrupt to the CPU.
5. The interrupt causes the CPU to suspend the currently executing program and to jump to the disk's interrupt service routine, which calls the original read request's completion routine, thus completing the second half of the asynchronous disk read.

4 **(10 marks)** Consider a procedure `A` running on a **single-processor machine** that uses UThreads to run procedure `B` on a separate thread and then joins with that thread, as shown below:

```
void A () {
    uthread_t t = uthread_create (B,0);
    void*     r = uthread_join   (t);
}
```

Give a step-by-step, timeline description of how UThreads handles these two statements. This should be fairly high level with each step described by a single, simple sentence. Refer to the key uthread data structures and indicate when thread's are created, started, block, unblocked, or destroyed (if these things happen) and when the CPU switches from one thread to another.

1. Thread A calls uthread_create, which creates a thread-control-block and stack for the new thread and places the thread on the ready queue.
2. Thread A calls uthread_join, which sees that thread B has not completed and so blocks thread A. It does this by storing a pointer to A's TCB in the *joiner* field in B's TCB and then removing the thread on the head of the ready queue (i.e., thread B) and switching to that thread. Thread B starts by thread_switch calling the procedure B.
3. When the procedure B returns, it returns to thread_switch, which stores B's return value in B's TCB, moves the TCB's *joiner* thread (thread A) to the ready queue, and stops thread B by removing the thread on the head of the ready queue (thread A) and switching to it.
4. Thread A returns from thread_switch into A's call to uthread_join, which retrieves B's return value from its TCB, free's B's TCB and stack, and returns this value to A.