12 **(8 marks)** **Threads and Scheduling.** Answer the following questions about threads.

12a Explain briefly (without giving any code) how threads can be used to simplify code that performs asynchronous operations such as communicating with an IO controller.

12b Explain briefly the difference between `uthread_yield` and `uthread_block`.

12c Is it possible for a thread to unblock itself? Explain your answer.

12d Consider a system in which there is at least one thread on the *ready queue* when a thread unblocks. What happens to the unblocked thread? Explain.

11 (10 marks) Threads.

11a Threads can be used to manage the asynchrony inherent in I/O-device access (e.g., reading from disk). Carefully explain how threads help.

11b Carefully describe in plain English the sequence of steps a user-level thread system such as *uthreads* follows to switch from one thread to another. Ensure that your answer explains the role of the *ready queue* and explains how the hardware switches from running one thread to the other.

11c What is the role of the *thread scheduler*?

11d Explain priority-based, round-robin scheduling.

11e Explain what else is needed to ensure that threads of equal priority get an equal share of the CPU?

```
int  c = 0;
int* v[2];

void bar (int* ap) {
  if (v[c] == NULL || *ap < *v[c]) {
    if (v[c] != NULL)
      dec(v[c]);
    v[c] = ap;
    inc(v[c]);
  }
  c = (c + 1) % 2;
}

void zot (int a) {
  if (v[c] != NULL && a < *v[c])
    *v[c] = a;
  c = (c + 1) % 2;
}
```

10 **(6 marks)** **Using Function Pointers.** Write a C procedure named `apply` that returns either the minimum or maximum value of a list of non-negative integers, as determined by one of its parameters, a function pointer. If the list is empty it should return -1. Assume the existence of procedures named `min(a,b)` and `max(a,b)` that compare two integers and returns the min or max integer. No other procedures are allowed and `apply` is not permitted to use an `if` statement. For example, the following statement should compute the maximum value in list `a` of length `n`.

```
int m = apply (max, a, n);
```

Write the procedure `apply()`:

```
int apply (int (*f)(int, int), int* a, int n) {
    if (n<=0)
        return -1;   // for 2016W2 this isn't necessary.
    else {
        int v = a[0];
        for (int i=1; i<n; i++)
            v = f(v, a[i]);
        return v;
    }
}
```

11 **(6 marks)** **IO Devices.** For each of the following, (a) explain what it is and (b) state whether the CPU or IO controller determines when it occurs (i.e., initiates it).

11a Programmed IO (PIO):

11b DMA:

11c Interrupts:

12 **(8 marks)** **Threads and Scheduling.** Answer the following questions about threads.

12a Explain briefly (without giving any code) how threads can be used to simplify code that performs asynchronous operations such as communicating with an IO controller.

12b Explain briefly the difference between `uthread_yield` and `uthread_block`.

12c Is it possible for a thread to unblock itself? Explain your answer.

12d Consider a system in which there is at least one thread on the *ready queue* when a thread unblocks. What happens to the unblocked thread? Explain.

13 **(9 marks)** **Synchronization** Consider the following C code that uses mutexes and condition variables. The code is considered to be is correct if both procedures complete successfully when they are called from concurrent threads

Insert a new stage just before the fetch stage (or after the execute stage). Check there to see if there is an interrupt pending and if so, jump to the address specified in the interrupt vector table. This will invoke the specific interrupt service routine for the given type of interrupt.

11 (10 marks) Threads.

11a Threads can be used to manage the asynchrony inherent in I/O-device access (e.g., reading from disk). Carefully explain how threads help.

Because threads execute asynchronously themselves, you can write code that looks synchronous but executes asynchronously. This is easier to read, write, and think about for programmers.

IO can take quite long (eg millions of CPU cycles), and in this time the CPU can be doing other things. Threads provide this flexibility because they can easily be blocked when they are waiting for something, and the CPU can go ahead and execute other threads.

11b Carefully describe in plain English the sequence of steps a user-level thread system such as *uthreads* follows to switch from one thread to another. Ensure that your answer explains the role of the *ready queue* and explains how the hardware switches from running one thread to the other.

A thread switch occurs in uthreads when a thread calls `uthread_block()` or `uthread_yield()`. This places the current thread on the ready queue and dequeues something else off the ready queue to begin execution.

The actual transfer of execution is achieved by pushing the register of the first thread onto its stack, saving its current stack pointer in its TCB and then switching the value of the stack pointer register to point to the target thread's stack. Then the CPU pops the registers of this new thread from its stack, and resumes execution.

11c What is the role of the *thread scheduler*?

The thread scheduler decides which thread should execute next.

11d Explain priority-based, round-robin scheduling.

Each thread has an assigned priority. When a thread finishes execution (or blocks/yields), the next thread chosen to execute is the thread with the highest priority.

11e Explain what else is needed to ensure that threads of equal priority get an equal share of the CPU?

In the above explanation of priority-based, round-robin scheduling *starvation* is possible because a single high priority thread could dominate all the CPU time if it never finishes execution. *Quantum preemption* would allow for all threads of equal priority to get an equal share of the CPU by periodically interrupting and switching threads.

12 (16 marks) Synchronization

12a Explain the difference between busy-waiting and blocking. Give one advantage of blocking.

Busy waiting occurs when the a thread waits for a lock to be free by actively polling the lock's value, spinning in a loop repeatedly reading it until it sees that it is available.

Blocking waiting occurs when a thread waits for a lock by sleeping so that other threads can use the CPU. It is then the responsibility of the thread that releases the lock to wakeup the waiting thread.