**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 (i.e., one thread calls `procX` and the other calls `procY`). Indicate whether or not the code is correct. If it is correct, describe why. If it is not correct, carefully describe the problem; you must be specific.

All variations share the following declarations. Note that initialization code is omitted for brevity; you should assume that all variables were initialized correctly.

```
int            flag = 0;
uthread_mutex  mx;
uthread_cond   condX, condY;
```

**13a**

```
void procX() {
  for (int i=0; i<100; i++) {
    uthread_cond_signal (condX);
    uthread_cond_wait (condY);
  }
}
void procY() {
  for (int i=0; i<100; i++) {
    uthread_cond_signal (condY);
    uthread_cond_wait (condX);
  }
}
```

Is this code correct? Justify your answer.

**13b**

```
void procX() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condX);
      uthread_cond_wait (condY);
    }
  uthread_mutex_unlock (mx);
}
void procY() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condY);
      uthread_cond_wait (condX);
    }
  uthread_mutex_unlock (mx);
}
```

Is this code correct? Justify your answer.

**13c**

```
void procX() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condX);
      if (i!=0 || flag==0) {
        flag = 1;
        uthread_cond_wait (condY);
      }
    }
  uthread_mutex_unlock (mx);
}
void procY() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condY);
      if (i!=0 || flag==0) {
        flag = 1;
        uthread_cond_wait (condX);
      }
    }
  uthread_mutex_unlock (mx);
}
```

Is this code correct? Justify your answer.

## 14 (6 marks) Semaphores.

Add semaphores to the code shown below to guarantee that the program always prints ”do...re...me”. Add code in the blank areas, if necessary. You can only use the `uthread_sem_t` API: `uthread_sem_create(int)`, `uthread_sem_wait(uthread_sem_t)`, and `uthread_sem_signal(uthread_sem_t)`.

```
#include <stdio.h>




void* doe(void* p) {



    printf("do...");


    return 0;
}

void* re(void* p) {




    printf("re...");


    return 0;
}

void* me(void* p) {



    printf("me");



    return 0;
}


int main (int argc, char** argv) {
    uthread_t  at,bt,ct;
    uthread_init(3);




    at = uthread_create (doe,NULL);
    bt = uthread_create (re,NULL);
    ct = uthread_create (me,NULL);
    uthread_join(at,0);
    uthread_join(bt,0);
    uthread_join(ct,0);



}
```

**15** **(6 marks)** **Deadlock.**
Consider these three procedures that are allowed to run concurrently in different threads.

```
void one() {
    uthread_mutex_lock   (a);
    uthread_mutex_lock   (b);
    ...
    uthread_mutex_unlock (b);
    uthread_mutex_unlock (a);
}
void two() {
    uthread_mutex_lock   (b);
    uthread_mutex_lock   (c);
    ...
    uthread_mutex_unlock (c);
    uthread_mutex_unlock (b);
}
void three() {
    uthread_mutex_lock   (c);
    uthread_mutex_lock   (a);
    ...
    uthread_mutex_unlock (a);
    uthread_mutex_unlock (c);
}
```

Can this program *deadlock*? Explain briefly.

## 12 (16 marks) Synchronization

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

**12b** Consider the following program in which `inc` and `dec` can run concurrently.

```
spinlock_t s;
int        c;
void dec() {
    int success = 0;
    while (success==0) {
        while (c==0) {}
        spinlock_lock (s);
        if (c>0) {
            c = c - 1;
            success = 1;
        }
        spinlock_unlock (s);
    }
}
void inc() {
    spinlock_lock (s);
    c = c + 1;
    spinlock_unlock (s);
}
```

Re-implement the program to eliminate all busy waiting using *monitors* and *condition variables*. You may make the changes in place above or re-write some or all of the code below.

**12c** Assume that monitors are implemented in such a way that a thread inside of a monitor is permitted to re-enter that monitor repeatedly without blocking (e.g., when `bar` calls `zot`, which calls `foo`, `foo` is permitted to enter monitor `x`). Indicate whether the following procedures could cause deadlock in multi-threaded program that contained them (and other procedures as well). Explain why or why not. If they could, say whether you could eliminate this deadlock by only adding additional monitors or additional monitor enter or exits (you may not remove monitors). If so, show how.

```
void foo () {
    monitor_enter (x);
    monitor_exit (x);
}
void bar () {
    monitor_enter (x);
    zot ();
    monitor_exit (x);
}
void zot () {
    monitor_enter (y);
    foo ();
    monitor_exit (y);
}
```

(i.e., one thread calls `procX` and the other calls `procY`). Indicate whether or not the code is correct. If it is correct, describe why. If it is not correct, carefully describe the problem; you must be specific.

All variations share the following declarations. Note that initialization code is omitted for brevity; you should assume that all variables were initialized correctly.

```
int             flag = 0;
uthread_mutex   mx;
uthread_cond    condX, condY;
```

**13a**

```
void procX() {
  for (int i=0; i<100; i++) {
    uthread_cond_signal (condX);
    uthread_cond_wait (condY);
  }
}
void procY() {
  for (int i=0; i<100; i++) {
    uthread_cond_signal (condY);
    uthread_cond_wait (condX);
  }
}
```

Is this code correct? Justify your answer.

Incorrect, condition variables without doing first locking the mutux. This causes an assertion error. Also accepted, deadlock because both processes could signal, where both signals are missed, and then both wait.

**13b**

```
void procX() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condX);
      uthread_cond_wait (condY);
    }
  uthread_mutex_unlock (mx);
}
void procY() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condY);
      uthread_cond_wait (condX);
    }
  uthread_mutex_unlock (mx);
}
```

Is this code correct? Justify your answer.

Incorrect, the first process to execute sends a signal that is missed by the second process to start. As a result the last wait() in the second process to start will not be matched and the process will not terminate (deadlock).

**13c**
```
void procX() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condX);
      if (i!=0 || flag==0) {
        flag = 1;
        uthread_cond_wait (condY);
      }
    }
  uthread_mutex_unlock (mx);
}
void procY() {
  uthread_mutex_lock (mx);
    for (int i=0; i<100; i++) {
      uthread_cond_signal (condY);
      if (i!=0 || flag==0) {
        flag = 1;
        uthread_cond_wait (condX);
      }
    }
  uthread_mutex_unlock (mx);
}
```

Is this code correct? Justify your answer.

Correct, fixes the problem with the previous example. Now, because of the flag the second process does one less wait, corresponding to the missed signal. Both finish.

**14** **(6 marks)** **Semaphores.**

**14a** Add semaphores to the code shown below to guarantee that the program always prints ”do...re...me”. Add code in the blank areas, if necessary. You can only use the `uthread_sem_t` API: `uthread_sem_create(int)`, `uthread_sem_wait(uthread_sem_t)`, and `uthread_sem_signal(uthread_sem_t)`.

```
#include <stdio.h>



void* doe(void* p) {


    printf("do...");


    return 0;
}

void* re(void* p) {



    printf("re...");


    return 0;
}

void* me(void* p) {


    printf("men");



    return 0;
}


int main (int argc, char** argv) {
    uthread_t  at,bt,ct;
    uthread_init(3);




    at = uthread_create (doe,NULL);
    bt = uthread_create (re,NULL);
    ct = uthread_create (me,NULL);
    uthread_join(at,0);
    uthread_join(bt,0);
    uthread_join(ct,0);



}
```

```
 //TODO
 uthread_sem_t  lockab, lockbc;

void* doe(void* p) {
    // TODO
    printf("do...");
    uthread_sem_signal(lockab);
    return NULL;
}

void* re(void* p) {
    // TODO
    uthread_sem_wait(lockab);
    printf("re...");
    uthread_sem_signal(lockbc);
    return NULL;
}

void* me(void* p) {
    // TODO
    uthread_sem_wait(lockbc);
    printf("me...");
    return NULL;
}

int main (int argc, char** argv) {
    uthread_t  at,bt,ct;
    uthread_init(3);
    // TODO
    lockab = uthread_sem_create (0);
    lockbc = uthread_sem_create (0);
     at = uthread_create (doe,NULL);
     bt = uthread_create (re,NULL);
     ct = uthread_create (me,NULL);
    uthread_join(at, 0);
    uthread_join(bt,0);
    uthread_join(ct,0);
    printf("n");
   TODO
    uthread_sem_destroy(lockab);
    uthread_sem_destroy(lockbc);
}
```

**15** **(6 marks)** **Deadlock.**

**15a** Consider these three procedures that are allowed to run concurrently in different threads.

```
void one() {
    uthread_mutex_lock   (a);
    uthread_mutex_lock   (b);
    ...
    uthread_mutex_unlock (b);
    uthread_mutex_unlock (a);
}
void two() {
    uthread_mutex_lock   (b);
    uthread_mutex_lock   (c);
    ...
    uthread_mutex_unlock (c);
    uthread_mutex_unlock (b);
}
void three() {
    uthread_mutex_lock   (c);
    uthread_mutex_lock   (a);
    ...
    uthread_mutex_unlock (a);
    uthread_mutex_unlock (c);
}
```

Can this program *deadlock*? Explain briefly.

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.

**12b** Consider the following program in which `inc` and `dec` can run concurrently.

```
spinlock_t s;
int        c;
void dec() {
    int success = 0;
    while (success==0) {
        while (c==0) {}
        spinlock_lock (s);
        if (c>0) {
            c = c - 1;
            success = 1;
        }
        spinlock_unlock (s);
    }
}
void inc() {
    spinlock_lock (s);
    c = c + 1;
    spinlock_unlock (s);
}
```

Re-implement the program to eliminate all busy waiting using *monitors* and *condition variables*. You may make the changes in place above or re-write some or all of the code below.

```
monitor m = new_monitor();
cond_variable gtzero = new cond_variable(m);
int c;

void dec() {
    monitor_enter(m);
    while (c == 0)
        cond_wait(gtzero);
    c = c - 1;
    monitor_exit(m);
}

void inc() {
    monitor_enter(m);
    c = c + 1;
    cond_signal(gtzero);
    monitor_exit(m);
}
```

If you are familiar with how the producer/consumer problem is implemented with condition variables, this question should be straightforward. If you're not, **go study that right now**. You should be able to easily identify the pattern of spinning on a value until it meets a condition, then acquiring a lock to verify that condition actually holds. Also look into how spinlocks are implemented, they use quite similar logic to acquire the shared lock (by spinning on the lock's value before attempting the atomic exchange).

I use pseudo code here to represent monitor/condition variable operations (recall that a monitor is basically a blocking lock). You could have instead used uthread function calls like `uthread_mutex_lock` and `uthread_cond_wait` if you wanted to. Pay attention to how the condition variable is initialized - don't forget you must associate it with a mutex!

**12c** Assume that monitors are implemented in such a way that a thread inside of a monitor is permitted to re-enter that monitor repeatedly without blocking (e.g., when `bar` calls `zot`, which calls `foo`, `foo` is permitted to enter monitor `x`). Indicate whether the following procedures could cause deadlock in multi-threaded program that contained them (and other procedures as well). Explain why or why not. If they could, say whether you could eliminate this deadlock by only adding additional monitors or additional monitor enter or exits (you may not remove monitors). If so, show how.

```
void foo () {
    monitor_enter (x);
    monitor_exit (x);
}
void bar () {
    monitor_enter (x);
    zot ();
    monitor_exit (x);
}
void zot () {
    monitor_enter (y);
    foo ();
    monitor_exit (y);
}
```

Consider the case where Thread 1 is executing `bar()` and Thread 2 is executing `zot()`. Let's say Thread 1 acquires `x` at the same time that Thread 2 acquires `y`. Now when Thread 1 calls `zot()` and tries to acquire `y`, it will block (because Thread 2 holds it). Similarly, when Thread 2 proceeds to call `foo()` and attempt to acquire `x`, it blocks because Thread 1 holds it. So a deadlock is possible.

Since the concurrent execution of `bar()` and `zot()` causes a deadlock, we can avoid the deadlock by preventing them from executing concurrently:

```
void foo () {
    monitor_enter (x);
    monitor_exit (x);
}
void bar () {
    monitor_enter (z);
    monitor_enter (x);
    zot ();
    monitor_exit (x);
    monitor_exit (z);
}
void zot () {
    monitor_enter (z);
    monitor_enter (y);
    foo ();
    monitor_exit (y);
    monitor_exit (z);
}
```

Doing well on this question requires you to be very comfortable with imagining the execution of threads. You know that a deadlock occurs when two threads are blocked in such a way that neither can release the resource that the other is waiting for. Here we only have two locks/resources - `x` and `y`.

Knowing that, you should start working backwards from a deadlock. One thread will be blocked waiting for `x` and the other `y`. There is only one place where a thread can be waiting for `y` - at `zot()`. So one of the threads must be blocked on that call for a deadlock to occur *and that thread must be holding* `x` already. This implies that the thread must have started by calling something that acquired `x` before calling `zot()`; ie `bar()`. In the answer we call this 'Thread 1'.

Now all that we have to do is come up with a second thread that needs to be blocked on `x` while holding `y`, and we will have a deadlock. Using similar logic as above, we can figure out that this happens when `foo()` is called by `zot()`.