RECITATION: 104
NUMBER OF HOURS TO COMPLETE: 6 hours

1. *(10 pt) Describe and explain each of the following (use examples):*

   (a) *The concept of interleaving instructions of two threads*

   Interleaving instructions of two threads is a process by which the threads act as if they're running in parallel but are actually simply switching between each other via a context switch. This has the potential to cause a race condition if proper preventative measures aren't taken.

   (b) *Each of the requirements for deadlock (give examples). Explain why the first three are necessary but not sufficient for deadlock.*

   i. **Mutual exclusion** - this is the case when a resource is in a "non shareable mode." In other words, the resource is set so that only one process is allowed or able to access said resource at a time. This is an issue for situations when multiple threads may require access to a resource simultaneously. One thread is forced to sit idle until the resource is freed by the other thread currently accessing it.

   ii. **Hold and wait** - this is the case when a process is allocated resources that it needs before all of the resources necessary are available. If that process can't relinquish those resources while holding, then resources are being unused by the idle process while it waits for the missing resources. This can also be an issue if two processes require resources that are being held by the other and neither can continue unless the other relinquishes the resource.

   iii. **No preemption** - this is the case when the availability of the resource is entirely dependent on the process currently using it. This becomes an issue because the operating system itself no longer has control of the availability and can't force a process to relinquish a resource if the OS decides to run another process at the given time.

   iv. **Circular wait** - this is the case when a process is currently holding resource that was allocated to it, but requires other resources that are simultaneously being held by other processes. This becomes an issue when processes require resources being held by other processes that also require resources being held by others and so on, circling around when the final process (if you imagine it like a chain of processes) requires a resource being held by the first process in a chain. This means that no processes can continue because they're all waiting for the next process in the chain to release its resources first.

2. *(10 pt) Compare and contrast the following (make sure to define each of the items):+*

   (a) *Mode switch vs process switch*

   A process switch is essentially a method of switching between one running process to another via a scheduler. In doing so, the entire state of the first process is saved as well as any memory it was using, and then the state of the second process is restored and the process continues from it left off. A mode switch is used for system calls which allows the process to switch between user mode and kernel mode. The primary purpose of the mode switch is to only allow the specific processes that require kernel access to run in a state in which the kernel can be altered/directly referenced in order to protect the kernel from unwanted changes.

   (b) *Semaphore vs conditional variable*

   Semaphores save the state of the threads. They are aware of which process is next in the list and whether or not the limit of processes with access to a resource has been reached. When one process is done, then the semaphore signals that the next process can begin using the resource since the limit is no longer maxed. A condition variable on the other hand is simply a signal to threads as to whether a specific event has or hasn't occurred yet. A thread that causes an event would signal a condition variable while a thread waiting for an event waits for said signal. Unlike semaphores, condition variables aren't aware of other process running and simply react to their own signals.

3. *(20 pt) Given the code below answer the following questions (explain your reasoning):*

```
int temp;
void swap(int *y, int *z) {
    int local;
    local = temp;
    temp = *y;
    *y = *z;
    *z = temp;
    temp = local;
}
```

   (a) *Is the function thread safe?*

   This function is not thread safe. There is no scheduling to protect the values, so there's nothing stopping a thread from entering and changing the value of *y (for example) and breaking the logic of the function. This is something that could easily occur with a context switch.

(b) *Is the function reentrant?*

Yes, the code is reentrant. There are no semaphores or scheduling that could lock access to resources, so there wouldn't be multiple process trying to modify/share the same bits of memory.

4. *(30 pt) Barrier Synchronization: A* ***barrier*** *is a tool for synchronizing the activity of a number of threads. When a thread reaches a* ***barrier point****, it cannot proceed until other threads have reached this point as well. When the last thread reaches the barrier point, all threads are released and can resume concurrent execution. You have a number of threads processing data and each piece of data takes a different amount of time to process. When a thread has completed its processing, it will ask for more work. However, the data requires that 4 workers all start together on the next 4 pieces of data. This means we want to gather 4 threads before having them begin processing. Please provide a solution (pseudocode), explain how it works, show that it is starvation free and deadlock free.*

```
count = 0 //global
threads[] //array of threads

barrier()
  threads.append() //append array with current thread
  mutex lock //protects count from being affected by other threads
    count++
  mutex unlock //remove the lock
  if count != 4 //not all threads have reached barrier point
    wait() //wait until signaled
    release() //force release resources allowing preemption
  else //all 4 threads at barrier
    for i = 4:1, i--
      signal(threads[i].pop()) //restart ith thread and pop from array
      mutex lock
      count-- //decrement count back down to zero for next barrier point
      mutex unlock
```

The line that releases the resources taken up by the current thread actively prevents deadlock and starvation by force releasing them.

5. *(30 pt) CU wants to show off how politically correct it is by applying the U.S. Supreme Courts Separate but equal is inherently unequal doctrine to gender, ending its long-standing practice of gender-segregated bathrooms on campus. However, as a concession to tradition, it decrees that when a woman is in the bathroom, other women may enter, but no men, and vice versa. Also, due to fire code, at most $N(N > 1)$ individuals may use the bathroom at any time. Your task is to write two functions:* ***man_use_bathroom()*** *and* ***woman_use_bathroom()****. Provide a monitor-based solution that manages access to the bathroom. Your solution should be fair, starvation free and deadlock free.*

```
//initialize global variables:
int men,women,num;

man_use_bathroom()
    men++ //increment the number of men waiting in line
    if state == w or state == full //if women or bathroom is full
        wait()
    else if state == empty or state == mens //empty or men inside
        num++ //increment number of people in bathroom
        men-- //no longer waiting in line
        if num == N
            state = full //bathroom is now full
        else
            state = mens //bathroom is now men only

        //call function to use bathroom

        num-- //leaving bathroom
        if num == 0
            state = empty
            if women > 0 //women waiting
                women.signal //signal them to enter
        else
            state = mens //in case state was still at full
            men.signal //signal next man
```

```
women_use_bathroom()
    women++ //increment the number of women waiting in line
    if state == m or state == full //if men or bathroom is full
        wait()
    else if state == empty or state == womens //empty or women inside
        num++ //increment number of people in bathroom
        women-- //no longer waiting in line
        if num == N
            state = full //bathroom is now full
        else
            state = womens //bathroom is now women only

        //call function to use bathroom

        num-- //leaving bathroom
        if num == 0
            state = empty
            if men > 0 //men waiting
                men.signal //signal them to enter
        else
            state = womens //in case state was still at full
            women.signal //signal next woman
```