1. Recycled Problem(s):

(a) [6 points] Define the meaning of a race condition? Answer the question first and use an execution sequence to illustrate your answer. You will receive no credit if only an example is provided without an elaboration.

**Answer:** A race condition is a situation in which more than one processes or threads access a shared resource concurrently, and the result depends on the order of execution.

The following is a simple counter updating example discussed in class. The value of count may be 9, 10 or 11, depending on the order of execution of the machine instructions of count++ and count--.

```c
int count = 10;

Thread_1(...) Thread_2(...) {
    // do something // do something
    count++; count--;
}
```

The following execution sequence shows a race condition. Two threads run concurrently (condition 1). Both threads access the shared variable count at the same time (condition 2). Finally, the computation result depends on the order of execution of the SAVE instructions (condition 3). The execution sequence below shows the result being 9; however, switching the two SAVE instructions yields 11. Since all conditions are met, we have a race condition.

<table>
<thead>
<tr>
<th>Thread_1</th>
<th>Thread_2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>do something</td>
<td>do something</td>
<td>count = 10 initially</td>
</tr>
<tr>
<td>LOAD count</td>
<td></td>
<td>Thread_1 executes count++</td>
</tr>
<tr>
<td>ADD #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD count</td>
<td></td>
<td>Thread_2 executes count--</td>
</tr>
<tr>
<td>SUB #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAVE count</td>
<td></td>
<td>count is 11 in memory</td>
</tr>
<tr>
<td></td>
<td>SAVE count</td>
<td>Now, count is 9 in memory</td>
</tr>
</tbody>
</table>

Stating that “count++ followed by count--” or “count-- followed by count++” would produce different results and hence a race condition is incorrect, because the threads do not access the shared variable count at the same time (i.e., condition 2).

See p. 193 of our text and class notes.

2. Synchronization

(a) [10 points] The semaphore methods Wait() and Signal() must be atomic to ensure a correct implementation of mutual exclusion. Use an execution sequence to show that if Wait() is not atomic then mutual exclusion cannot be maintained. You must use an execution sequence to present your answer as we did in class. Otherwise, you risk low or zero point.
**Answer:** If `Wait()` is not atomic, its execution may be switched in the middle. If this happens, mutual exclusion will not be maintained. The following is a possible execution sequence, where `Count = 1` is the counter variable of the involved semaphore.

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Count</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD Count</td>
<td>1</td>
<td>A executes Count-- of Wait()</td>
<td></td>
</tr>
<tr>
<td>SUB #1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD Count</td>
<td>1</td>
<td>B executes Count-- of Wait()</td>
<td></td>
</tr>
<tr>
<td>SUB #1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAVE Count</td>
<td>0</td>
<td>B finishes Count--</td>
<td></td>
</tr>
<tr>
<td>if (Count &lt; 0)</td>
<td>0</td>
<td>A finishes Count--</td>
<td></td>
</tr>
<tr>
<td>if (Count &lt; 0)</td>
<td>0</td>
<td>It is false for A</td>
<td></td>
</tr>
<tr>
<td>Both A and B enter the critical section</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that this question asks you to demonstrate a violation of mutual exclusion. Consequently, you receive low grade if your demonstration is not a violation of mutual exclusion.

This problem was assigned as an exercise in class.

(b) [8 points] Enumerate and elaborate all major differences between a semaphore wait/signal and a condition variable wait/signal. Vague answers and/or inaccurate or missing elaboration receive no credit.

**Answer:** The following table has the details:

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>Condition Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used anywhere, but not in a monitor</td>
<td>Can only be used in monitors</td>
</tr>
<tr>
<td><code>wait()</code> does not always block its caller</td>
<td><code>wait()</code> always blocks its caller</td>
</tr>
<tr>
<td><code>signal()</code> increases the semaphore counter and may release a process</td>
<td><code>signal()</code> either releases a process, or the signal is lost as if it never occurs</td>
</tr>
<tr>
<td>If <code>signal()</code> releases a process, the caller and the released both continue</td>
<td>If <code>signal()</code> releases a process, either the caller or the released continues, but not both</td>
</tr>
</tbody>
</table>

This is part of the monitors slides discussed in class.

3. Process Scheduling

(a) [8 points] What are preemptive and non-preemptive scheduling policies? Elaborate your answer.

**Answer:** With the non-preemptive scheduling policy, scheduling only occurs when a process enters the wait state or terminates. With the preemptive scheduling policy, scheduling also occurs when a process switches from running to ready due to an interrupt, and from waiting to ready (i.e., I/O completion).
(b) [8 points] What is priority inversion? How could it happen? How can it be overcome? Note that there are three questions here.

**Answer:** If a high-priority process needs to access a protected resource that is currently being held by a low-priority process, the high-priority process is blocked by the low-priority process until the low-priority process completes its access. Thus, a low priority process actually has a “higher” priority because it forces higher priority processes to wait! This is priority inversion.

To overcome this priority inversion problem, processes that are accessing the resource that the high-priority process needs inherit the high priority until they are done with the resource. As a result, this will speed up the low-priority processes. When the low priority processes finish, their priority reverts to its original value. This is the priority-inheritance protocol.

See p. 238 of our text.

(c) [20 points] Five processes A, B, C, D and E arrived in this order at the same time with the following CPU burst and priority values. A smaller value means a higher priority.

<table>
<thead>
<tr>
<th></th>
<th>CPU Burst</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Fill the entries of the following table with waiting time and average waiting time for each indicated scheduling policy and each process. Ignore context switching overhead.

<table>
<thead>
<tr>
<th>Scheduling Policy</th>
<th>Waiting Time</th>
<th>Average Waiting Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>First-Come-First-Served</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Non-Preemptive Shortest-Job First</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Round-Robin (time quantum=2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Answer:**

<table>
<thead>
<tr>
<th>Scheduling Policy</th>
<th>Waiting Time</th>
<th>Average Waiting Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>First-Come-First-Served</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Non-Preemptive Shortest-Job First</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Priority</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>Round-Robin (time quantum=2)</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>
The above diagram shows the execution pattern of the round-robin algorithm with time quantum 2, where dashed arrows indicate waiting periods.

See class notes for the details.

4. Problem Solving:

(a) [20 points] Design a class `Barrier` in C++, a constructor, and method `Barrier_wait()` that fulfill the following specification:

- The constructor `Barrier(int n)` takes a positive integer argument `n`, and initializes a private `int` variable in class `Barrier` to have the value of `n`.
- Method `Barrier_wait(void)` takes no argument. A thread that calls `Barrier_wait()` blocks if the number of threads being blocked is less than `n-1`, where `n` is the initialization value and will not change in the execution of the program. Then, the `n-th` calling thread releases all `n-1` blocked threads and all `n` threads continue. Note that the system has more than `n` threads. Suppose `n` is initialized to 3. The first two threads that call `Barrier_wait()` block. When the third thread calls `Barrier_wait()`, the two blocked threads are released, and all three threads continue. Note that your solution cannot assume `n` to be 3. Otherwise, you will receive zero point.

Use semaphores only to implement class `Barrier` and method `Barrier_wait()`. Otherwise, you will receive zero point. You may use type `Sem` for semaphore declaration and initialization (e.g., “Sem S = 0;”), `Wait(S)` on a semaphore `S`, and `Signal(S)` to signal semaphore `S`.

You should explain why your implementation is correct in some details. A vague discussion or no discussion receives no credit.

**Answer:** This is a simple variation of the readers-writers problem, because the last thread must activate/do something. Compare the task of the `n`-th thread with what the last reader should do, and you should be able to see the similarity.

It is obvious that we need a counter `count` to count the number of waiting threads. Initially, `count` should be 0. Based on the specification, we need two semaphores: `Mutex` for protecting the counter `count`, and `WaitingList` for blocking threads.

When a thread calls `Barrier_wait()`, it locks the counter, and checks to see if it is the `n`-th one. If it is not the `n`-th one, the thread releases the lock and waits on semaphore `WaitingList`. This portion is trivial. Note that the order of “releasing the lock” and “waiting on semaphore `WaitingList`” is important. Otherwise, a deadlock will occur. (Why?)

If the thread is the `n`-th one, it must release all waiting threads that were blocked on semaphore `WaitingList`. Since we know there are exactly `n-1` waiting threads, executing `n-1` signals to semaphore `WaitingList` will release them all. Then, the `n`-th thread resets the counter and releases the lock.
Based on this idea, the following is the `Barrier` class:

```cpp
class Barrier {
private:
    int Total; // total number of threads in a batch
    int count; // counter that counts blocked threads
    Semaphore WaitingList(0); // the waiting list
    Semaphore Mutex(1); // mutex lock that protects the counter
public:
    Barrier(int n) { Total = n; count = 0 }; // constructor
    Barrier_wait(); // the wait method
};

Barrier::Barrier_wait()
{
    int i;

    Mutex.Wait(); // lock the counter
    if (count == Total-1) { // if I am the n-th one
        for (i=0; i<Total-1; i++) // release all waiting threads
            WaitingList.Signal();
        count = 0; // reset counter
        Mutex.Signal(); // release the lock
    } else { // otherwise, I am not the last one
        count++; // one more waiting threads
        Mutex.Signal(); // release the mutex lock
        WaitingList.Wait(); // block myself
    }
}
```

The protection of the counter `count` must start at the very beginning and extend to the very end so that the blocked `n-1` threads can be released in a “single” batch. In other words, when the execution flow enters the “then” part of the `if`, all blocked `n-1` threads are released as a single group. Otherwise, we may have the following problems. **First**, we may release a newcomer rather than the threads that were waiting in the barrier prior to the release. **Second**, threads just released may come back (i.e., fast-runner) and be released again. In this case, the same thread is released twice and one of the originally blocked threads is not released. Hence, this violates the specification that the blocked `n – 1` threads must be released.

If the `Barrier_wait()` method is rewritten as the following to “increase efficiency,” we will have problems:
Barrier::Barrier_wait() // incorrect version
{
    int i;

    Mutex.Wait(); // lock the counter
    if (count == Total-1) { // I am the n-th one
        count = 0; // reset counter
        Mutex.Signal(); // release the lock
        for (i=1; i<=Total-1; i++) // release all waiting threads
            WaitingList.Signal();
    } // I am done
    else { // otherwise, I am not the last one
        count++; // one more waiting threads
        Mutex.Signal(); // release the mutex lock
        WaitingList.Wait(); // block myself
    }
}

With this version, a thread just released from semaphore WaitingList may come back and call Barrier_wait() again. This thread can immediately change the value of count and wait on WaitingList again while the original n-th thread is still in the process of releasing the blocked n-1 threads. Since we cannot make any assumption about the order used for releasing threads, it is possible that a fast running thread is released again and one of the originally blocked thread will be blocked and released the next run. Or, it may be blocked forever!

The following is a similar solution. In this solution the n-th thread signals n times so that it will release itself at the end. However, this “solution” does have the fast-runner problem. Suppose n is 2. The first thread blocks as usual. When the second comes, it signals WaitingList twice, releases the Mutex lock, and is switched out by a context switch. Since the first thread was released by one of the two signals, it may come back, go through all steps, and wait on semaphore WaitingList faster than the second thread does. Since WaitingList was signaled twice, this returning thread is not blocked and can pass through. As a result, the same thread is released twice and the releasing thread is blocked. Of course, this is terribly wrong!

Barrier::Barrier_wait() // incorrect version
{
    int i;

    Mutex.Wait(); // lock the counter
    if (count == Total-1) { // I am the n-th one
        count = 0; // reset counter
        for (i=1; i<=Total; i++) // release all waiting threads
            WaitingList.Signal(); // including myself
    } // I am done
    else { // otherwise, I am not the last one
        count++; // one more waiting threads
    }
    Mutex.Signal(); // release the mutex lock
    WaitingList.Wait(); // block myself
}
(b) [20 points] Using ThreadMentor to design a Hoare monitor `Barrier` and method `Barrier_wait()` that fulfill the following specification:

- The constructor (i.e., initialization) of monitor `Barrier`, `Barrier(int n)`, takes a positive integer argument `n`, and initializes a private `int` variable of the monitor to have the value of `n`.

- Method `Barrier_wait(void)` takes no argument. A thread that calls `Barrier_wait()` is blocked if the number of threads being blocked, including this one, is less than `n`, where `n` is the initialization value and will not change in the execution of the program. Then, the `n`-th calling thread will release all `n-1` blocked threads. Suppose `n` is initialized to 3. The first two threads that call `Barrier_wait()` block. When the third thread calls `Barrier_wait()`, all three threads will continue. Note that your solution cannot assume `n` to be 3. Otherwise, you will receive zero point.

Use ThreadMentor syntax to write the monitor code. **You must elaborate and justify your solution. Otherwise, you will receive low or even no grade.** Hint: This problem looks easy; but, if you forget this is a Hoare monitor you could end up with an incorrect solution.

**Answer:** If you know the semantics of the Hoare-style monitor well, this is actually an extremely easy problem. With a Hoare-style monitor, the signaling process (or thread) yields the monitor to the released process (or thread) immediately. In other words, if there are waiting threads on a signaled condition variable, one of them will take over the monitor and run. *This is the key to the solution of this problem.*

The following is a possible solution. It uses a condition variable `block` to block those threads that have to wait, variable `count` to count the number of blocked threads, and variable `n` for the control purpose. Note that `count` is initialized to 0.

```cpp
class Barrier : public Monitor {
public:
    Barrier(int); // constructor
    Barrier_wait(void); // monitor procedure Access()
private:
    Condition block; // C.V. for blocking threads
    int n; // maximum number of threads
    int count, i; // working variables
};

Barrier::Barrier(int Max): Monitor(HOARE) // constructor
{
    n = Max;
    count = 0;
}
```

The following is the `Barrier_wait()` method. If the current `count` is less than `n - 1`, the calling thread increases the value of `count`, indicating one more waiting thread arrived, and waits. The key is the way of using `Signal()`. If the current `count` is `n - 1`, the newcomer should take the responsibility of releasing the `n - 1` previously blocked threads. In the else part, only one call to `Signal()` is made. This `Signal()` releases one blocked thread, which will reduce the `count` and `Signal()` the condition variable `block`. In this way, we start a “chain of reaction” and each released thread releases another thread. This is usually referred to cascade release or cascade signal.
void Barrier::Barrier_wait(void)
{
    MonitorBegin(); // enter monitor
    if (count < n-1) { // if not full count
        count++; // increase the counter
        block.wait(); // wait to be released
        count--; // reduce count
        block.signal(); // release one more blocked
    }
    else
        block.signal(); // initiate cascade release
    MonitorEnd(); // exit monitor
}

There are a few important notes about this solution:

- For the sake of simplicity, assume that the $n$-th thread $T_n$ releases thread $T_{n-1}$ that was blocked earlier, thread $T_{n-1}$ releases thread $T_{n-2}$, ..., and thread $T_2$ releases $T_1$.

- When thread $T_i$ releases $T_{i-1}$, $T_i$ yields the monitor to thread $T_{i-1}$, and $T_i$ and $T_{i-1}$ become inactive and running, respectively, in the monitor. Consequently, starting from $T_n$’s $\text{Signal}()$ call to the release of thread $T_1$, the monitor is non-empty, and no other threads can enter the monitor. Therefore, all $n$ threads, $T_n$ included, are released in the same “batch” and no other threads can have any influence on this “chain of action.”

- Note that the $\text{Signal}()$ from $T_1$, the last released thread, is ignored because there is no waiting thread on condition variable $\text{block}$. Additionally, even though $T_1$’s $\text{Signal}()$ yields the monitor to a new thread not in this group, it does not matter because all $n$ threads have been released properly, and will exit the monitor once they become running from inactive.

- From the above discussion, the value of $\text{count}$ is maintained properly. In other words, the value of $\text{count}$ increases by one when a new thread successfully enters the monitor via the call to $\text{Barrier}\_\text{wait}()$ until the value of $\text{count}$ becomes $n - 1$. Then, the next thread’s $\text{Signal}()$ initiates cascade release, one at a time. Each released thread decreases the value of $\text{count}$ by 1 before it yields to the next thread with $\text{Signal}()$. Since there cannot be any new threads in the monitor other than those involved in this chain, the value of $\text{count}$ goes down to 0 steadily.

- In this way, thread $T_n$ successfully releases the $n - 1$ threads that called $\text{Barrier}\_\text{wait}()$ before $T_n$ does.

Some added a number of statements, possibly with a new condition variable, to the beginning of $\text{Barrier}\_\text{wait}()$ to prevent “intruders” from entering when releasing threads. This is not necessary if your code can take advantages of the mutual exclusion property of a monitor. Additionally, once threads can get into $\text{Barrier}\_\text{wait}()$, it is difficult to guarantee the generation of a correct batch of $n$ threads.

Many used a $\text{for}$ loop to release threads from condition variable $\text{block}$. A typical way is shown below. This is a risky move. Consider the following scenario. Suppose $T_n$ is the $n$-th thread calling $\text{Barrier}\_\text{wait}()$ and starts releasing $n - 1$ threads blocked on condition variable $\text{block}$. When $T_n$ signals, it yields the monitor to a released thread, say $T_1$. Then, $T_1$ reduces the value of $\text{count}$ by 1 and leaves the monitor. Now, the monitor is empty with $T_n$ being inactive. What if $T_1$ comes back again (i.e., fast-runner) and calls $\text{Barrier}\_\text{wait}()$? Since we should not assume who would get the monitor between $T_n$ and the newcomer $T_1$, it may well be $T_1$. If this
is the case, $T_1$ increases the counter $count$ and blocks itself. Suppose $T_n$ is picked to run (i.e., from inactive to running). $T_n$’s second signal may cause $T_1$ to be released again because one cannot assume any policy for releasing threads from a condition variable. As a result, we have an incorrect “batch” as $T_1$ is released twice!

```c
void Barrier::Barrier_wait(void)
{
    MonitorBegin(); // enter monitor
    if (count < n-1) { // if not full count
        count++; // increase the counter
        block.wait(); // wait to be released
    }
    else
    {
        for (int i = 1; i < n; i++)
            block.signal(); // release n-1 blocked threads
    }
    MonitorEnd(); // exit monitor
}
```

Some may suggest the following similar but a bit subtle solution. This solution is more “efficient” because it resets $count$ rather than decreasing it by one every time a thread is released. If a fast runner comes back, it finds out $count$ is still $n-1$ (since the releasing thread is inactive in the monitor and not able to reset $count$), and executes the for loop. As a result, we have two releasing threads!

```c
void Barrier::Barrier_wait(void)
{
    MonitorBegin(); // enter monitor
    if (count < n-1) { // if not full count
        count++; // increase the counter
        block.wait(); // wait to be released
    }
    else
    {
        for (int i = 1; i < n; i++)
            block.signal(); // release n-1 blocked threads
        count = 0; // reset counter
    }
    MonitorEnd(); // exit monitor
}
```

What if the else part is replaced as follows? A fast-runner will come back and wait on condition variable $block$. It may be released in this batch! Again, this is an incorrect solution.

```c
else {
    count = 0; // reset counter
    for (int i = 1; i < n; i++)
        block.signal(); // release n-1 blocked threads
}
```