On my honor, I have not given, nor received, nor witnessed any unauthorized assistance on this work.

Print name and sign: ______________________________

Question:	1	2	3	4	Total
Points:	5	8	5	12	30
Score:					

1. (5 points) Could the following scenario lead to deadlock? If it could, draw a diagram showing the deadlock.

   Thread A will try to grab first lock 1 and then lock 2. Thread B will try to grab first lock 2 and then lock 3. Thread C will try to grab first lock 1 and then lock 3.

   Solution: No. The sequence in which the locks are acquired is ordered. We can't generate any circular dependencies!

2. (8 points) In order for deadlock to occur, four conditions (listed below) must be met. Give a brief explanation of why each is needed for deadlock to occur.

   (a) Mutual exclusion

   Solution: Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock). If threads can share locks, we don't have deadlock (and in fact have another shared resource).

   (b) Hold-and-wait

   Solution: Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire). If a thread yields a lock after a certain amount of time, we could prevent deadlock (although we still may encounter performance issues)

   (c) No preemption

   Solution: Resources (e.g., locks) cannot be forcibly removed from threads that are holding them. If locks could be forcible removed (by another thread, OS, etc) then we could resolve deadlock by taking a lock from one thread and giving it to another.

   (d) Circular wait

   Solution: There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

3. (5 points) Explain the difference between a **mutex** and **semaphore**.

Solution: Both are synchronization primitives which enable programmers to write atomic code. However, a semaphore is more flexible. A semaphore can be made to act like either a lock or a condition variable. The initial value of a semaphore determines its behavior. If we set it's initial value to 1, it acts just like a mutex. Given that a semaphore is more flexible, it can be used in a variety of ways which locks cannot. In particular, it can be used to enforce ordering or allow multiple threads in a critical section.

4. Dr. Summet is working on her implementation of locks for some multi-threaded code. Here's what she's got so far (minus header and `include` code):

```
int lock = 0;   //0 for unlocked, 1 for locked
int shared = 0; //shared variable

void* incrementer(void* args){
  for(int i = 0; i < 100; i++){
    //check lock
    while(lock > 0) {}  //spin until unlocked

    lock = 1; //set lock
    shared++; //increment
    lock = 0; //unlock
  }
  return NULL;
}

int main(int argc, char * argv[]){
    int num_threads;
    // Declare an array of threads
    pthread_t threads[num_threads];

    // Create those threads!
    long i;
    for (i = 0; i < num_threads; i++) {
        pthread_create(&threads[i], NULL, print_thread_id, (void *) i);
    }
    // Use pthread_join to make the main thread pause
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Shared value: %d; should be 200\n", shared);
    return 0;
}
```

However, no matter how many times she runs her code, she's still getting weird results:

```
    Shared value: 198; should be 200
    Shared value: 208; should be 200
```

(a) (5 points) Explain why Dr. Summet's solution is not atomic and thus doesn't work.

Solution: Dr. Summet is trying to use high level language integers like locks. Checking the int variable `lock` and updating a variable takes multiple machine instructions (e.g. `ld` to read the value and store it into a register, an `add` to change the register's value, `str` to write it back to memory). A context switch to the other thread can happen at any point in that series leading to a race condition. Semaphores or locks are implemented with different machine instructions which make them atomic in a way that regular variables can never be.

(b) (3 points) You try to explain to Dr. Summet how she should fix her code. Should she use a semaphore or a mutex? Why?

Solution: Haha! Trick question. Either is fine here. Semaphores can be initialized with a starting value of 1 which makes them work just like a lock/mutex. Arguably she should use a lock/mutex here because the simpler solution is usually better, but she could make her code work with either.

(c) (4 points) Correct her code on the previous page using the synchronization primitive you selected in the previous part. You do not need to have syntactically correct code, but you should demonstrate you understand how to guarantee atomicity.

Solution: For full credit your answer should include:
changing of `int lock` to the correct datatype
initializing the mutex/semaphore somewhere in main before the function is called
calling lock/unlock or post/wait in the incrementer function