Instructions:

1. No aids are permitted except non-programmable calculators with no persistent memory.
2. Turn off all communication devices. Communication devices must be stored with your personal items for the duration of the exam. Taking a communication device to a washroom break during this examination is not allowed and will be interpreted as an academic offence.
3. Place all bags at the front or side of the examination room, or beneath your table, so they are inaccessible.
4. There are three (3) questions, some with multiple parts. Not all are equally difficult.
5. The exam lasts 80 minutes and there are 70 marks.
6. Verify that your name and student ID number is on the cover page and that your examination code appears on the bottom of each page of the examination booklet.
7. If you feel like you need to ask a question, know that the most likely answer is "Read the Question". No questions are permitted. If you find that a question requires clarification, proceed by clearly stating any reasonable assumptions necessary to complete the question. If your assumptions are reasonable, they will be taken into account during grading.
8. Do not fail this city.
9. After reading and understanding the instructions, sign your name in the space provided below.

Signature

Marking Scheme (For Examiner Use Only):

Question	Mark	Weight	Question	Mark	Weight	Question	Mark	Weight
1a		15	2a		15	3a		8
1b		4	2b		12	3b		4
1c		5				3c		3
1d		3						
Total								**70**

Question 1: Processes and Threads [28 marks total]

1A. UNIX Fork [15 marks]

In this question, there is a task that can be split into parts 'A' and 'B'. You may assume there are implementations for the functions `int execute_A()` and `int execute_B()` that work correctly (and are not shown). If either of the execution functions returns a non-zero value, this indicates an error.

In this question, complete the code below to achieve the following implementation. Use `fork()` to create a child process. The child process should call function `execute_B()` and return the result to the parent. The parent process should call `execute_A()` and collect its result. The parent should then collect the result of the child using `wait()` and then produce the console output described in the next paragraph. If no errors occurred, `main` should return 0; otherwise it should return -1.

If an error occurs, it should be reported to the console including the error number (e.g., "Error 7 Occurred."). If more than one error occurs, report both errors. If both functions return zero, it means all is well and the program should print "Completed." to the console.

Recall that `fork()` returns a `pid_t` (which is effectively an integer) and may fail (return -1 from `main` if that happens). The system call `wait( int* return_value )` is used to collect a result from a child. Recall also that `%d` is used in a call to `printf` to indicate a value that will be replaced with a number and `\n` produces a new line.

```
int main( int argc, char** argv ) {










}
```

1B: Thread Cancellation [4 marks]

In the pthread standard the `pthread_cancel` function behaves differently depending on the attributes of the thread being cancelled. There are two (2) different cancellation modes; list both and briefly describe them.

1C: Five State Model [5 marks]

Although processes use the seven-state model, for threads the somewhat simpler five-state model is adequate. List the five states; it is not necessary to explain what each of them means.

1D: Amdahl's Law [3 marks]

Recall Amdahl's Law:

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

...where S is the sequential fraction of the program and N is the number of processors.

Suppose we have a task that takes 120 s in total when run on single CPU system. Measurements have shown the sequential part S of the program takes 24 s and that the work of splitting up the tasks and recombining the results takes negligible time.

Compute, using Amdahl's Law:

1. The speedup if the task is moved to a machine with 4 CPUs.
2. The maximum possible speedup (in a world where infinite CPUs are available).
3. How many CPUs are necessary (round up if your answer is fractional) to achieve a speedup of 3.5.

Question 2: Concurrency and Synchronization [27 marks total]

2A: The Barbershop Problem [15 marks]

Consider the "Barbershop Problem", originally proposed by Dijkstra. A variant of this appears in the textbook. A barbershop is a place where customers get their hair cut. A barbershop consists of a waiting area with n chairs, and a barber chair.

If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

Customer threads should call `get_hair_cut()` when it is their turn. If the shop is full, the customer should `return` (exit/leave). The barber thread will call `cut_hair`. The barber can cut only one person's hair at a time, so there should be exactly one thread calling `get_hair_cut()` concurrently. You can assume that external forces cause customers to appear and the barber to keep working (so you do not need to write any loops). Assume `n` is initialized to an appropriate value.

Complete, using pseudocode as in the lectures, the barber and customer sections below, as well as the initialization code necessary for this to work.

Hint: you will need two binary semaphores and a mutex as well as a way of keeping track of how many customers are present. Remember to initialize them correctly.

Indicate how you will name and initialize your variables, semaphores, and mutexes here:

Add your pseudocode for the barber and customer below:

Customer **Barber**

2B: Synchronization Patterns [12 marks]

In this question, there is a task that can be split into parts 'A' and 'B'. The code below has already been partially completed so that two pthreads are created and started. Each thread does preparation on its part and then runs its part. The functions to prepare are `prepare_a()` and `prepare_b()`; the ones to run are `run_a()` and `run_b()`. The implementation of these functions is not shown but you may assume they exist and are implemented correctly.

However, the code as written does not produce the desired output. For correct output, both `prepare_a()` and `prepare_b()` must be completed before either of the run functions may begin. This is the classical "rendezvous" synchronization pattern.

In this question, you will use semaphores to arrange the rendezvous so that the output will be correct. As shown in class, the POSIX semaphore is a `struct` typdef-ed as `sem_t`. The associated functions are:

```
int sem_init( sem_t* semaphore, int shared, int initial_value);
int sem_destroy( sem_t* semaphore )
int sem_wait( sem_t* semaphore )
int sem_post( sem_t* semaphore )
```

Use `0` as the value of `shared` when initializing a semaphore. Remember also that "post" is the same as "signal".

```
/* Global Variables Go Here */




int main( int argc, char** argv ) {




    pthread_t thread_a;
    pthread_t thread_b;
    pthread_create(&thread_a, NULL, execute_a, NULL);
    pthread_detach( thread_a );
    pthread_create(&thread_b,  NULL, execute_b, NULL);
    pthread_detach( thread_b );




    pthread_exit(0);
}

void *execute_a( void* ignore ) {

    prepare_a();




    run_a();


    pthread_exit(0);
}

void *execute_b( void* ignore ) {

    prepare_b();




    run_b();


    pthread_exit(0);
}
```

Question 3: Deadlock [15 marks total]

3A: Deadlock Recovery: Victim Selection [8 marks]

In class we discussed seven (7) criteria that could be considered in choosing a "victim" process. In the system in question, the deadlock resolution strategy is "murder" (selectively killing a process involved in a deadlock). List four (4) of the criteria we discussed in class and explain how those criteria would be used in making a decision.

3B: Deadlock Conditions [4 marks]

What are the four (4) conditions necessary for a deadlock to occur?

3C: There is no try (?) [3 marks]

Can the pthread function `pthread_mutex_trylock` be used instead of `pthread_mutex_lock` to prevent a deadlock? Your answer should begin with yes or no, followed by your reasoning.