Instructions:

1. No aids are permitted except non-programmable calculators.
2. Turn off all communication devices.
3. There are three (3) questions, some with multiple parts. Not all are equally difficult.
4. The exam lasts 80 minutes and there are 70 marks.
5. 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.
6. 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		2	2a		5	3a		4
1b		3	2b		5	3b		4
1c		2	2c		10	3c		4
1d		5	2d		12			
1e		12	2e		2			
Total								**70**

Question 1: Processes and Threads [24 marks total]

1A: Cancellation of Threads [2 marks]

What is the difference between deferred and asynchronous cancellation of a thread?

1B: Fork Bomb Attack [3 marks]

In class we discussed a denial of service attack called the "Fork Bomb". Define this attack and two (2) strategies for preventing it (or at least limiting its impact).

1C: Kernel and User Level Threads [2 marks]

The discussion of kernel and user level threads contained three models of mapping user threads to kernel threads: one-to-one, many-to-one, and many-to-many. Why is there no one-to-many model?

1D: Task Stack Space [5 marks]

Assume you are using the ARM-RL RTX library on LPC1756 evaluation board as in the lab. The capacity of a task stack is 320 bytes (i.e., 320 bytes allocated to use as the stack space of the task). The stack usage of a task is defined as size (in bytes) of the items that pushed onto the stack, divided by the capacity of the stack.

(1) What is the lower bound of the stack usage if the task is in RUNNING state?

(2) What is the lower bound of the stack usage if the task is not in a RUNNING state?

1E: Inter-Process Communication with UNIX Pipes [12 marks]

Recall that in UNIX, one method for inter-process communication is the idea of a *pipe*, a unidirectional method of communication. In this question, one process will spawn another. The child process will read the data from a file, send it back to its parent using a pipe, and the parent will send it to a printer. For the child to do this, it will need to read the file data into a buffer, and then write that into the pipe. The parent will receive the data from the pipe into a buffer, and then call a function to print the data of that buffer.

To create a pipe: call `pipe( int fileDescriptors[] )` This call returns an integer. File descriptors are just integers (so the file descriptors variable is an integer array). The pipe call will fill in the correct values for `fileDescriptors[0]` (the read-end) and `fileDescriptors[1]` (the write-end) so you do not need to initialize them and this will automatically "open" these "files" as well.

To spawn a new process: `fork()`, which takes no parameters and returns a `pid_t` (an integer value). Remember, it creates an exact clone of the parent process.

To open a file: `open ( int fileDescriptor )`. A file must be opened before it can be read.

To close a file: `close( int fileDescriptor )`. It is good programming practice to close all files and pipe ends when finished with them.

To read from a file: `read( int fileDescriptor, void *destination, int numBytes )`. This is a blocking read: the calling function will be blocked until `numBytes` bytes have been read from the source.

To write to a file: `write( int fileDescriptor, void *source, int numBytes )`.

Assume there exists a function `void print( void* data )` that sends the data pointed to by the parameter to the printer. The implementation of this function is not shown.

Assume also there is a function `int fileLength( int fileDescriptor )` that takes the file descriptor of a file and returns its correct length, in bytes. Assume the file to print's descriptor is defined as a constant `PRINT_FILE`. Do not forget to open this file before reading it, and close it when done.

Complete the code below to get the desired behaviour. Remember that system calls like `fork` and `read` and `pipe` may fail and therefore your code should do error checking on their results. If an error is encountered, your `main` function should return `-1` immediately; otherwise it should return `0` to indicate normal exit. Remember to `free()` any memory you allocated with `malloc( int numBytes )`.

```
int fd[2];
pid_t pid;


int main( void ) {









}
```

Question 2: Concurrency and Synchronization [34 marks total]

2A: Mutual Exclusion with Lock Variables [5 marks]

The following code attempts to enforce mutual exclusion to protect a critical section by attempting to first lock, then verify. Notice that, as a boolean variable, zero represents false (unlocked, in this case), and any non-zero value represents true (locked):

```
while_locked:
   lock++;
   if (lock > 1) {   /* if > 1 it means that it was > 0 before we increased */
       lock--;     /* undo the increment that we just did and try again */
       goto while_locked;
   }
   /* critical section here */
   lock--;         /* when critical section is done, unlock */
```

Notice that the variable `lock` is shared between all threads requiring access to the critical section.

Does this technique work? If you answer yes, explain why it works; if you answer no, show an interleaving demonstrating a situation where it fails (i.e., a situation where two threads are in the critical section at a time).

2B: Semaphore with Mutex [5 marks]

Consider the following (partial) implementation of a semaphore based on an available mutex primitive:

```
struct semaphore {
    mutex lock;
    int counter;
    // ...
}

sem_init (struct semaphore * sem, int value) {
    mutex_init (sem->lock);
    sem->counter = value;
}

sem_wait (struct semaphore * sem) {
    // counter decrement is not atomic at the assembly level;
    mutex_lock (sem->lock);
    --counter;
    mutex_unlock (sem->lock);

    if (counter < 0) {
        set state of the calling task to BLOCKED
    }
}

sem_post (struct semaphore * sem) {
    mutex_lock (sem->lock);
    ++counter;
    mutex_unlock (sem->lock);

    if (there are tasks blocked on this semaphore) {
        pick "oldest" task and set its state to READY
    }
}
```

Does this work? If your answer is yes, explain why it works; if your answer is no, show an example of an interleaving of two or more tasks that illustrates the flaw.

2C: Writer Priority [10 marks]

Recall from lectures the *Readers-Writers Problem*. Any number of readers may be in the critical section simultaneously; only one writer may be in the critical section at a time, and when it is, no readers are allowed. The solution that prevented readers and writers from starving, as outlined in the lectures:

Writer

```
1. wait( turnstile )
2. wait( roomEmpty )
3. [write data]
4. signal( turnstile )
5. signal( roomEmpty )
```

Reader

```
 1. wait( turnstile )
 2. signal( turnstile )
 3. wait( mutex )
 4. readers++
 5. if readers == 1
 6.     wait( roomEmpty )
 7. end if
 8. signal( mutex )
 9. [read data]
10. wait( mutex )
11. readers--
12. if readers == 0
13.     signal( roomEmpty )
14. end if
15. signal( mutex )
```

Using the pseudocode format above, modify the solution so that writers have priority over readers. Giving writers priority may potentially cause readers to starve, but you may ignore this. Hint: you will probably want to break up the `roomEmpty` semaphore into something like `noReaders` and `noWriters`. A reader in the critical section should hold the `noReaders` semaphore and a writer should hold `noWriters` and `noReaders`.

Writer

Reader

2D: Hardware Mechanisms [12 marks]

For each of the following hardware mechanisms, discuss its relevance to concurrency and concurrency control. You should discuss both the relevance in terms of how the Operating System implements or controls concurrency, and in terms of how programs and programmers deal with concurrency.

1. Timer interrupts
2. Multicore architectures
3. Assembly level atomic instructions

2E: Memory Allocation [2 marks]

Why is calling the C memory allocation function `malloc()` inside a critical section is a bad idea?

Question 3: Deadlock [12 marks total]

3A: Deadlock Conditions [4 marks]

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

3B: Deadlock Recovery Victim Selection [4 marks]

Give two (2) reasons why deadlock recovery algorithms tend to choose "younger" processes over "older" ones.

3C: Two-Phase Locking [4 marks]

Recall the concept of *two-phase locking*: a thread or process attempts to acquire all resources, and if it does not get all of them, it releases any resources that it did acquire and tries again from the beginning. While this will prevent deadlock, it is possible to encounter the more difficult to detect problem of *livelock*: the processes are not, strictly speaking, deadlocked, because they are not blocked, but they are unable to make progress either. Explain, using an example, how this might happen.