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 considered 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, with multiple parts. Not all are equally difficult.
5. The exam lasts 60 minutes and there are 50 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. A reference sheet is attached as the last page of the examination.
9. Do not fail this city.
10. 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
1.1		13	2.1		12	3.1		5
‘ 1.2		2	2.2		10			
1.3		3						
1.4		5						
Total								**50**

1 Operating Systems, Processes, and Threads [23 marks total]

1.1 C Programming and System Calls [13 marks]

Professor Derek Wright needs your help. This year, the Department of Electrical & Computer Engineering needs to assemble a report on all of the ECE courses to submit for accreditation. On disk there is a binary and not-very-human-readable file called `courses.dat`. What Prof. Wright needs is a tool that can interpret this file. For a given course number (e.g., 254), the tool should print the number of accreditation units (AU) for that course. The first (and only) argument is the course number and the output is the number of AUs for that course. So a sample invocation would look like:

```
./ece_course_au 254
AUs: 5.5
```

The file `courses.dat` contains, in random order, the data for all ECE courses stored in the following format:

```
typedef struct {
  int number; /* integer, e.g. 254 */
  double lecture_hours;
  double tutorial_hours;
  double lab_hours;
  double credits;
  double au; /* Accreditation units (AUs) for this course */
  char mandatory; /* '0' for no, '1' for yes */
} ece_course;
```

Your program should open the file and read one `ece_course`-sized chunk at a time, interpret it as the `ece_course` structure, examine if the `number` field matches the provided value, and if so print the associated AUs to the console. If there are no matches for the provided course number in the file, then a message indicating an error should be printed as in the example and -1 should be returned:

```
./ece_course_au 999
No data found for course ECE 999.
```

Complete the code below to implement the desired behaviour. Be sure to close anything opened and deallocate anything that was allocated. Check the provided input for validity as well.

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









}
```

1.2 Fork and Pipe [2 marks]

In the in-class code example demonstrating `pipe()`, both the parent and child process call `close(fd[0])` and `close(fd[1])`, for a total of four calls to `close()`. Explain why both parent and child need to close both ends of the pipe.

1.3 RL-RTX OS [3 marks]

From your understanding of the RL-RTX OS, describe what occurs when a task is switched from the RUNNING state to another state that is not INACTIVE. Limit your answer to what was covered in Lab 1 (2 marks).

What is the purpose of the `MAGIC_WORD` in the stack of a task (1 mark)?

1.4 Not If I Cancel You First! [5 marks]

We might assume that something has become stuck if a thread takes too long to complete its task. One of the ways that we can deal with this is to have a "watchdog" thread. The watchdog thread will cancel the worker thread if it has taken too long. If the worker finishes in time, however, it cancels the watchdog thread.

Complete the code below so that the watchdog timer will wait for 30 seconds using the standard sleep function: `unsigned int sleep(unsigned int seconds)`. If the watchdog has not yet been cancelled, it should cancel the worker thread. If the worker thread completes `long_running_task` before being cancelled, it should cancel the watchdog instead. Assume that the worker is set up for asynchronous cancellation and the watchdog is set up for deferred cancellation.

```
void* worker( void* arg ) {
  pthread_t * watchdog_thread = (pthread_t*) arg;




  /* Might get stuck! */
  result_t * res = long_running_task( );




  pthread_exit( res );
}
```

```
void* watchdog( void* arg ) {
  pthread_t * worker_thread = (pthread_t*) arg;









  pthread_exit( NULL );
}
```

2 Concurrency and Synchronization [22 marks total]

2.1 Short Answer [12 marks]

Answer each of the following questions in max. 3 sentences, or complete the code as directed. 3 marks each.

1. Suppose there existed a function `int sem_check( sem_t * sem )` that told you the current value of the semaphore. Your co-worker proposes using this function to find out whether your thread would get blocked trying to enter a critical section. Is this a good idea? Explain why or why not.

2. Recall from lecture the *signalling* synchronization pattern that used a semaphore. The UNIX `signal` can be used as a method of interprocess communication. Can it be used to replace a semaphore in this pattern? Explain.

3. Complete the following code fragment to clean up allocated resources in case the task is cancelled before it gets to the end:

```
void foo( void* arg ) {



}

void* bar( void * argument ) {
  task* t = malloc( sizeof( task ) );



  /* Do something useful with task t - code not shown.
     A cancellation point exists in this block. */




  pthread_exit( t );
}
```

4. In the basic Readers-Writers solution, the one in which writers could starve, we discussed modifying the solution to prevent the possibility that a writer starves. Is it possible for readers to starve? Explain your answer.

2.2 Get a Pizza This, hosted by TV CHEF NAME [10 marks]

You are doing a co-op term at a television production company. A new show to be hosted by some famous TV chef personality is being pitched, and you're going to write a simulation of it. The show is about making pizza. A pizza requires three ingredients: dough, sauce, and cheese. All three ingredients are necessary to make a pizza (otherwise it does not meet the definition of a pizza).

Each contestant has an unlimited supply of one ingredient. Contestant A has an unlimited supply of dough, Contestant B has an unlimited supply of sauce, and Contestant C has an unlimited supply of cheese. Each contestant needs to get the two ingredients they do not have and then can make a pizza. They will continue to (try to) make pizza in a loop until time is up. At the beginning of the episode, the host places two different random ingredients out. Contestants can signal the host to ask for more ingredients, but they should not do so unless they actually need some. Each time the host is woken up (signalled), he again places two different random ingredients out. When an ingredient is placed on the table, the host signals the associated semaphore. For example, if the host puts out cheese and sauce, then the host signals both `cheese` and `sauce`.

Part 1 (3 marks). Someone else suggests the following (pseudocode) description of how the different contestants (threads) should behave. All semaphores start at 0, except for `host` which starts as 1 (so the host will run the first time). This does not work. Explain why, demonstrating a sequence of events that causes the problem.

Contestant A

```
wait( sauce )
get_sauce()
wait( cheese )
get_cheese()
make_pizza( )
signal( host )
```

Contestant B

```
wait( dough )
get_dough()
wait( cheese )
get_cheese()
make_pizza( )
signal( host )
```

Contestant C

```
wait( sauce )
get_sauce()
wait( dough )
get_dough()
make_pizza( )
signal( host )
```

Part 2. (1 mark) Imagine now that each contestant gets a helper. The job of the helper is to, well, help their contestant to make pizza by figuring out whose turn it is. For this there are boolean variables `dough_present`, `sauce_present`, and `cheese_present` that are all initialized to `false`. They are protected by a semaphore (called `mutex`). The helpers update that variable, and based on the information available, signal which contestant should come up to the table and take ingredients. Each contestant now has a semaphore (such as `contestantA` for contestant A) which the helpers will signal on. Contestants are still responsible for telling the host to put out more ingredients. See the psuedocode on the next page.

Helper 1

```
wait( sauce )
wait( mutex )
if dough_present
  dough_present = false;
  signal( contestantC )
else if cheese_present
  cheese_present = false;
  signal( contestantA )
else
  sauce_present = true;
end if
signal( mutex )
```

Helper 2

```
wait( dough )
wait( mutex )
if sauce_present
  sauce_present = false;
  signal( contestantC )
else if cheese_present
  cheese_present = false;
  signal( contestantB )
else
  dough_present = true;
end if
signal( mutex )
```

Helper 3

```
wait( cheese )
wait( mutex )
if dough_present
  dough_present = false;
  signal( contestantB )
else if sauce_present
  sauce_present = false;
  signal( contestantA )
else
  cheese_present = true;
end if
signal( mutex )
```

Indicate the Initial Values for the following semaphores.
`contestantA`, `contestantB`, `contestantC`:
`mutex`:

Part 3 (6 marks) Complete the psuedocode for the contestants below so that they cooperate with the helpers to successfully make delicious, delicious pizza.

Contestant A **Contestant B** **Contestant C**

3 Deadlock [5 marks total]

3.1 Rewriting the Program [5 marks]

Although we can't make deadlock categorically impossible, when we have control of the source code of a program, we can make some modifications to our program to make it so that deadlock cannot happen in that program. For each of the strategies below, name which of the four essential elements of deadlock is eliminated, and explain why it works.

1. Use one mutex for everything (no other synchronization constructs).

2. Use more than one mutex in the program, but a thread may hold only one at a time.

3. Use more than one mutex in the program, but threads are required to acquire the mutexes in a specified order.

4. Make each thread work on its own local values (such as the `pthread5.c` example from lecture) instead of shared data.

5. Use trylock functionality for mutex locking.

Reference Sheet

Assume always the C99 standard (e.g., you can declare an integer in the same line as the for statement).

Memory is allocated in C with `malloc()` and to get the size of memory you want to allocate, there is `sizeof`, normally used in conjunction with `malloc`. Example: `int* p = malloc( sizeof( int ) );`

Memory is deallocated using `free`. Example: `free( p );`

An argument can be converted to an integer using the function `int atoi( char* arg )`.

Printing is done using `printf` with formatting. `%d` prints integers; `%lu` prints unsigned longs; `%f` prints double-precision floating point numbers. A newline is created with `\n`.

Some UNIX functions you may need:

```
pid_t fork( )
pid_t wait( int* status )
pid_t waitpid( pid_t pid, int status, int options ) /* 0 for options OK */
int kill( pid_t pid, int signal ) /* returns 0 returned if signal sent, -1 if an error */
void signal( int signal_number, void (*handler)(int signal) );


int open(const char *filename, int flags);  /* Returns a file descriptor if successful, -1 on error */
ssize_t read(int file_descriptor, void *buffer, size_t num_bytes); /* Returns number of bytes read */
ssize_t write(int file_descriptor, const void *buffer, size_t num_bytes); /* Returns number of bytes written */
int rename(const char *old_filename, const char *new_filename); /* Returns 0 on success */
int close(int file_descriptor);
```

When opening a file the following flags may be used for the `flags` parameter (and can be combined with bitwise OR, the `|` operator):

Value	Meaning
`O_RDONLY`	Open the file read-only
`O_WRONLY`	Open the file write-only
`O_RDWR`	Open the file for both reading and writing
`O_APPEND`	Append information to the end of the file
`O_TRUNC`	Initially clear all data from the file
`O_CREAT`	Create the file
`O_EXCL`	If used with `O_CREAT`, the caller MUST create the file; if the file exists it will fail

For your convenience, a quick table of the various pthread and semaphore functions we have discussed:

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument )
pthread_join( pthread_t thread, void **return_value )
pthread_detach( pthread_t thread )
pthread_cancel( pthread_t thread )
pthread_testcancel( ) /* If the thread is cancelled, this function does not return (thread terminated) */
pthread_exit( void *value )
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */

sem_init( sem_t* semaphore, int shared, int initial_value); /* 0 for shared OK */
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```