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 room, or beneath your table, such that they are inaccessible.
4. There are five (5) questions. 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
1a		10	2a		14	3a		10
1b		2	2b		5	3b		5
			2c		4			
Total								**50**

Question 1: Processes and Threads [12 marks total]

1A: Fork and Find [10 marks]

There is a linear array of integers and you wish to search the array to find the index of a specific value (any location of it will do). Linear searches are slow, but they are a parallelizable task. You would therefore like to break up this job into three (3) processes.

Assume there is a globally defined array of integers called `array` that is filled with appropriate values. Its length is defined in the global variable `array_length`. It may or may not contain the desired search value (defined as the global variable `search_value`). If either thread does find it, print a message to the console with `printf` indicating the index. Example: `printf("Found at %d\n", index);`

Each process should search its (approximately) one third of the array. If one process has a slightly larger section because the array does not divide evenly by 3, that is expected.

Complete the code below to perform a linear search of the `array` using two processes.

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

  return 0;
}
```

1B: Signals [2 marks]

Recall that in UNIX, processes as well as users can send signals to processes. Some signals, such as SIGHUP, can be "caught" and then handled or ignored. Other signals cannot. Explain (1) the value of allowing some signals can be caught and (2) a reason why some signals cannot be caught.

Question 2: Concurrency and Synchronization [23 marks total]

2A: Chemistry [14 marks]

There are two kinds of thread, `oxygen()` and `hydrogen()`. As you will recall from basic chemistry, water, H_2O, requires two hydrogen modules and one oxygen module. To assemble the desired molecule (water) a group rendezvous pattern is needed to make each thread wait until all ingredients are present in the correct amounts. As each thread passes the barrier, it should call the function `bond()` which makes the water. Your solution must function so that all threads for one molecule invoke `bond()` before any of the threads from the next molecule do.

To clarify, if an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads to arrive. If a hydrogen thread arrives a the barrier when no other threads are waiting, it has to wait for an oxygen thread and another hydrogen thread. It is not necessary for the threads to know what other threads they are matched with, as long as the correct elements are present in the correct proportions.

Complete the C code below to enforce the constraints as explained above. Assume the `oxygen` and `hydrogen` threads are created and started correctly and in the correct proportions. The code for the creation of those two types of threads is not shown for space reasons. To make things simpler, the reusable two-phase barrier discussed in lecture has been converted into C code and is available for your use.

You are not required to declare any additional variables or synchronization constructs but you may if you wish. Be sure to properly initialize and clean up anything you do use.

```
int oxygen;
int hydrogen;
pthread_mutex_t barrier_mutex;
sem_t turnstile;
int barrier_count;
int barrier_N;
pthread_mutex_t bond_mutex;
sem_t oxygen_queue;
sem_t hydrogen_queue;



void barrier_enter( ) {
  pthread_mutex_lock( &barrier_mutex );
  barrier_count++;
  if ( barrier_count == barrier_N ) {
    sem_post( &turnstile );
  }
  pthread_mutex_unlock( &barrier_mutex );
  sem_wait( &turnstile );
  sem_post( &turnstile );
}

void barrier_exit( ) {
  pthread_mutex_lock( &barrier_mutex );
  barrier_count--;
  if ( barrier_count == 0 ) {
    sem_wait( &turnstile );
  }
  pthread_mutex_unlock( &barrier_mutex );
}



void* oxygen( void* ignore ) {



}
```

```
int main( void ) {
  oxygen = 0;
  hydrogen = 0;
  barrier_count = 0;
  barrier_N = 3;

  pthread_mutex_init( &barrier_mutex, NULL );
  sem_init( &barrier_turnstile, 0, 0 );

  pthread_mutex_init( &bond_mutex, NULL );
  sem_init( &oxygen_queue, 0, 0 );
  sem_init( &hydrogen_queue, 0, 0 );

  /* Additional initializations go here */



  /* Creation of oxygen and hydrogen threads
   not shown for space reasons */

  pthread_mutex_destroy( &barrier_mutex );
  sem_destroy( &barrier_turnstile );
  pthread_mutex_destroy( &bond_mutex );
  sem_destroy( &oxygen_queue );
  sem_destroy( &hydrogen_queue );

  /* Additional cleanups go here */



  pthread_exit( 0 );
}


void* hydrogen( void* ignore ) {



}
```

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

The following code attempts to enforce mutual exclusion to protect a critical section by attempting to lock, then checking if it was successful. If `lock` equals zero, it means it is unlocked; any non-zero value means it is locked. Assume at program startup `lock` is initialized with a value of `0`.

```
while( 1 ) { /* Infinite Loop */
   lock++;
   if ( lock > 1 ) {
        /* lock was > 0 before we increased */
        /* undo the increment and try again */
       lock--;
   } else {
       break; /* Success! Can enter critical section */
   }
}
/* Critical Section */
lock--; /* 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).

2C: Rude Philosophers Problem [4 marks]

In the normal description of the dining philosophers problem, each of the philosophers attempts to pick up the chopsticks immediately adjacent to their seat. Does it solve the problem of deadlock if that restriction is removed and a philosopher may pick up any free chopstick, regardless of where it is on the table? Your answer should begin with yes or no, followed by your reasoning.

Question 3: Deadlock [15 marks total]

3A: Try, Try Again [10 marks]

In the following C code, functions `foo()` and `bar` run concurrently. Assume the mutexes `m1` and `m2` have been properly initialized.

```
void foo() {
    /* Get Ready */
    pthread_mutex_lock( &m1 );
    pthread_mutex_lock( &m2 );
    /* Critical Section */
    pthread_mutex_unlock( &m2 );
    pthread_mutex_unlock( &m1 );
    /* Clean Up */
}
```

```
void bar() {
    /* Get Ready */
    pthread_mutex_lock( &m2 );
    pthread_mutex_lock( &m1 );
    /* Critical Section */
    pthread_mutex_unlock( &m1 );
    pthread_mutex_unlock( &m2 );
    /* Clean Up */
}
```

The code as shown very obviously has a risk of deadlock. To solve this deadlock, change both `foo` and `bar` to use try-lock functionality (`pthread_mutex_trylock`) instead. Write your modified code below.

```
void foo() {









}
```

```
void bar() {









}
```

3B: Thread Vigilantism [5 marks]

Imagine you are writing an application that uses pthreads and runs on a UNIX system that has neither a deadlock detection nor a recovery process. Users have recently reported that if a particular XML message arrives then the system becomes deadlocked due to some problems in the parser. Given that the operating system is not able to do anything for you, explain how in your process you would (1) detect a (likely) deadlock and (2) what you can do about it if you detect one. Assume you cannot change the parser to avoid the problem altogether.

POSIX System Call Reference

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 );`

Some UNIX functions you may need:

```
pid_t fork( )
void wait( int* status )
void waitpid( pid_t pid, int status )
kill( pid_t pid, int signal )
```

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 **returnValue )
pthread_detach( pthread_t thread )
pthread_cancel( pthread_t thread )
pthread_testcancel( ) /* returns nonzero if cancelled */
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 )

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 )
```