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		2	2.1		1.5	3		9
‘ 1.2		3	2.2		8.5			
1.3		5	2.3		6			
1.4		6	2.4		9			
Total								**50**

1 Operating Systems, Processes, and Threads [16 marks total]

1.1 Resource Management [2 marks]

The Operating System is responsible for management of many system resources. For each of the following resources, give a concrete example of what could happen if the Operating System did not manage the resource:

1. Memory
2. Printer (I/O Device)
3. Files
4. CPU

1.2 Interprocess Communication with a Pipe [3 marks]

In the example in the lecture notes of the `pipe( int fd[] )` system call, the call to `pipe` takes place before the call to `fork()`. Explain why the call to `pipe` needs to be before `fork`.

1.3 ARM RL-RTX Stacks [5 marks]

This question relates to Lab 1, and a task running on the ARM RL-RTX Operating System. From a task's `OS_TCB`, use the variables `stack`, `tsk_stack`, and `os_stackinfo` to draw a block representation of a task's stack and the relevant memory addresses for the start, top, and limit of the stack. Please outline sections of the stack that hold specific information for the WAITING/READY and RUNNING state. Include placement of the `MAGIC_WORD`. Assume `p_TCB->priv_stack=0`.

WAITING/READY State

RUNNING State

1.4 Signal Handling in UNIX [6 marks]

You are working on software that makes use of multiple threads (pthreads), interacts with files on the local file system, and communicates over the network. Your supervisor asks you to look into modifying the program so it can handle the `SIGINT` signal (typically generated by a user at the console using the keyboard command `Ctrl-C`).

Describe what happens currently, when the signal is not handled (1 mark):

What is the (likely) motivation for wanting to handle this signal (1 mark)?

Write a brief description (in point form) of what steps you will likely take when handling the signal (2 marks).

You are also asked if you can handle `SIGKILL` in the same way. You know the answer is no; explain why (1 mark).

A team member observes the program sometimes terminates with a segmentation fault. A segmentation fault is really a signal, `SIGSEGV`. He asks if you can tell the program to ignore `SIGSEGV`. Ignoring a signal stops the program from crashing and restarts the instruction that caused the signal. Explain why this is a bad idea (1 mark).

2 Concurrency and Synchronization [25 marks total]

2.1 Synchronization Patterns [1.5 marks]

Give a real-life example of when:

1. The *Signalling* synchronization pattern is used:

2. The *Rendezvous* synchronization pattern is used:

3. The *Mutual Exclusion* synchronization pattern is used:

2.2 Semaphore Implementation [8.5 marks]

You are writing code for a very simple embedded system where the OS does not provide semaphores, but you can use the pthread library and therefore the `pthread_mutex_t` as a mutex. Your task will be to implement the semaphore for this system. In this question, however, you will just write pseudo-code.

The internal definition of the semaphore you are planning to implement is:

```
struct threadinfo {
  int thread_id;
  struct threadinfo * next;
};
```

```
typedef struct {
  int value;
  pthread_mutex_t mutex;
  struct thread_info * queue;
} semaphore;
```

Write pseudocode to show step-by-step what each function named below will do:

```
void sem_init( semaphore * s, int initial_value )
```

```
void sem_destroy( semaphore * s )
```

```
void sem_wait( semaphore * s )
```

```
void sem_signal( semaphore * s )
```

2.3 Producer-Consumer Starvation & Deadlock [6 marks]

Consider the Producer-Consumer problem as in lecture, where both produce and consumer run in infinite loops:

Producer
```
1. [produce item]
2. wait( spaces )
3. [add item to buffer]
4. signal( items )
```

Consumer
```
1. wait( items )
2. [remove item from buffer]
3. signal( spaces )
4. [consume item]
```

The buffer is shared and has capacity C. Can either the producer or the consumer starve in this scenario? Justify your answer for the producer and consumer separately (4 marks).

Can they become deadlocked? Justify your answer to this part as well (2 marks).

2.4 Harry Potter and the Return of the pthread House Elves [9 marks]

At Hogwarts School of Witchcraft and Wizardry, undesirable tasks are done by House Elves, magical creatures who apparently are not covered under any sort of employment standards or workers rights legislation. They work as a team under the direction of Dobby, who is a free elf. Dobby creates a list of items for the elves to do, and the elves do them. This cycle never ends, unfortunately for all involved: Dobby and the elves must keep working.

When the list is empty, Dobby posts 20 tasks to do by calling `post_tasks()`. After that, Dobby sleeps (is blocked). Worker elves take tasks using the function `take_task()` (which cannot be safely run in parallel) and then do the work by calling `do_work()` (which can be done in parallel). If the list of tasks is empty, an elf cannot call `take_task()` and must instead wake up Dobby, who will then post the next 20 tasks before going to sleep again; meanwhile the elf is blocked until the next group of tasks is ready. Note that Dobby and the elves are responsible for managing the count of tasks available.

Consider the following code implementation of the above behaviour. It, unfortunately, contains errors that need to be corrected. For three (3) of the errors, identify the problem, explain some possible effects of this problem, and explain how it should be corrected (including what line(s) should be modified).

```
1  int* tasks;
2  pthread_mutex_t mutex;
3  sem_t empty_list;
4  sem_t full_list;
5
6  void init() {
7    tasks = malloc( sizeof( int ) );
8    pthread_mutex_init( &mutex, NULL );
9    sem_init( &empty_list, 0, 0 );
10   sem_init( &full_list, 0, 0 );
11 }
12
13 void cleanup() {
14   free( tasks );
15   pthread_mutex_destroy( &mutex );
16   sem_destroy( &empty_list );
17   sem_destroy( &full_list );
18 }
19 void* dobby( void * ignore ) {
20   while ( 1 ) {
21     sem_wait( &empty_list );
22     post_tasks();
23     (*tasks) += 20;
24     sem_wait( &full_list );
25   }
26 }
27
28 void* house_elf( void * ignore ) {
29   while( 1 ) {
30     pthread_mutex_lock( &mutex );
31     if ( *tasks == 0 ) {
32       sem_post( &empty_list );
33       sem_wait( &full_list );
34     }
35     take_task();
36     pthread_mutex_unlock( &mutex );
37     (*tasks)--;
38     do_work();
39   }
40 }
```

3 Deadlock [9 marks]

You are writing a function called `find_and_resolve_deadlock()`, to resolve a deadlock, once one has been detected. The provided helper function `struct proc * find_deadlock()` returns a pointer to a linked list of type `struct proc` (see definition below). If the pointer is `NULL` then there is no deadlock at the current time. Otherwise, the pointer points to the head of an unsorted linked list containing all the processes involved in the deadlock. Deallocation of the linked list is the responsibility of the caller of `find_deadlock()`.

```
struct proc {
  pid_t pid; /* pid_t is really an int and can be treated as int */
  struct proc* next; /* Is NULL if this is the last item of the list */
};
```

Your system's chosen deadlock strategy is to terminate the youngest process (that is, the process with the highest process id, `pid`). If a deadlock is detected, terminate the chosen process (using signal 9) and check if the deadlock is resolved by running the `find_deadlock()` function again. It may be necessary to kill more than one process to fully resolve the deadlock. Once the deadlock is resolved, the `find_and_resolve_deadlock()` function is done.

Complete the `find_and_resolve_deadlock()` function below to implement the desired behaviour. Remember to deallocate all allocated resources.

```
struct proc * find_deadlock() {
  /* Implementation not shown (but described above) */
}

void find_and_resolve_deadlock() {

}
```

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 kill( pid_t pid, int signal ) /* returns 0 returned if signal sent, -1 if an error */

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 count); /* Returns number of bytes read */
ssize_t write(int file_descriptor, const void *buffer, size_t count); /* 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 **returnValue )
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 )

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 )
```