Question 1

Part (c) [1 MARK]

If two threads access a shared variable at the same time, there will be a concurrency error.

Question 2

Question 3. [2 MARKS]

Consider the following excerpt from the solution to the reader/writer problem using condition variables:

```
// assume all variable are correctly initialized
// assume writing will only ever hold the values 0 or 1

pthread_mutex_lock(&mutex);
while(writing) {
    pthread_cond_wait(&turn, &mutex);
}
readcount++;
pthread_mutex_unlock(&mutex);
```

Which of the following statements are true?

- ☐ `writing` must be 0 when `pthread_cond_wait` returns.
- ☐ The mutex is held by this thread while it is blocked in `pthread_cond_wait`
- ☐ The mutex is held by this thread whenever it checks the value of `writing`
- ☐ The mutex is held by this thread when the while loop terminates.

Question 3

Q3. (10 marks) (Conceptual + Reasoning) [15 minutes] Answer the following short questions.

a) [1 marks] Which of the following scheduling algorithms may cause starvation? Circle all that apply.
a. FCFS
b. SJF
c. Round-Robin
d. MLFQ (final revision)

b) [2 marks] What happens if a thread executes a pthread_cond_signal on a condition variable *cv1*, if no other thread is currently waiting on *cv1*? Explain in detail.

c) [2 marks] What's the difference between an interrupt and a system call?

d) [2 marks] Kernel-level threads are typically faster than user-level threads. Do you agree with this statement? Explain your rationale in detail.

e) [3 marks] A program called "run_stuff" uses 1 fork() system call and 2 exec() system calls. The exec system calls execute the following commands "ls -l" and "cat /proc/cpuinfo", respectively. Depending on how these system calls are issued, how many processes are likely to be spawned by the "run_stuff" program (other than the main program itself)? You must explain your rationale to get marks for this question.

Question 4

c) (6 marks) The bank hires you to change the `transfer_amount` implementation so that the amount can be transferred only if there is at least that amount in the account. If the account were to be left with a negative balance as a result of the transfer, then the operation waits until the account gets sufficient money (from some other transfer).
You may modify the account structure as you see fit, and add in it any other synchronization variables you may need (no need to worry about initializing them, as long as it is clear what your intent is). Your solution should ensure *no deadlocks* occur. You may *NOT declare any global variables for synchronization purposes*.
Note: efficiency does matter (particularly in terms of achievable parallelism)!

```
typedef struct acct {

     float balance;



} account;


void transfer_amount(account *a1, account *a2, float amount) {















}
```

Question 5

Q5. (6 marks) Scheduling (Reasoning) [10 minutes]
Consider a **process scheduling** algorithm which prioritizes processes that have used the least processor time in the recent past. Answer the following questions and explain *in detail* your rationale.

a) Will this algorithm favour either CPU-bound processes or I/O-bound processes?
b) Can this algorithm permanently starve either CPU-bound or I/O-bound processes?

Note: Do not feel compelled to use this entire page, but be specific and elaborate on your thought process!