Q1. (1 mark each) True/False [5 minutes]

Indicate below, for each statement, whether it is (T)rue or (F)alse. Circle the correct answer.

T/F Whenever the processor is in kernel mode, interrupts should be disabled.

T/F One of the ways a context switch is triggered is when a process calls yield() voluntarily.

T/F User-level threads are not visible to the OS for scheduling purposes.

T/F A process will go back to the Running state immediately after receiving a SIGCONT signal.

T/F A process is placed in the blocked state when it fails to acquire a spinlock.

T/F In the round-robin scheduling algorithm, the quantum assigned to a process should be roughly equal to the context switch time.

Q2. (2 marks each) (Conceptual) [5 minutes]

Explain briefly the following concepts or terms, in the context of this course:

a) Spinlock

b) Scheduling quantum

c) System call

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.

Q4. (12 marks) Synchronization (Reasoning) [15 minutes]

Consider a bank that manages several customer accounts, but which allows only one operation for its customers: `transfer_amount(account *a1, account *a2, float amount)`, which transfers the given `amount` from account `a1` to account `a2`. The implementation of this function is given below:

```
typedef struct acct {
      float balance;
      pthread_mutex_t *lock;
} account;

void transfer_amount(account *a1, account *a2, float amount) {
      pthread_mutex_lock(a1->lock);
      pthread_mutex_lock(a2->lock);
      a1->balance -= amount;
      a2->balance += amount;
      pthread_mutex_unlock(a1->lock);
      pthread_mutex_unlock(a2->lock);
}
```

You can assume that all synchronization variables have been properly initialized and that memory has been allocated where necessary. Account balance can be negative too.

a) (2 marks) Provide a sequence of execution which leads to deadlock.

b) (4 marks) Consider that instead of using a lock per account like in the code above, we use only one global lock L. Consider that `transfer_amount` is modified to instead acquire this global lock L at the beginning of the function, and release it at the end.

i) Can deadlock occur in this case? If yes, give an example sequence of execution. If no, explain why not.

ii) Do you see some other drawback to using this approach?

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) {













}
```

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!