1. No. if the access is read for both threads, then concurrency error will not occur.

2. `b)`, `c)` and `d)` are true

Correct solution

`c)` and `d)` are true

Notes

Question What does it mean when mutex is held by this thread?

Question What I do know is that `pthread_cond_wait` puts thread to sleep. My question here is, how come the mutex is not held when thread is in a blocked state/sleep?

3. a) Only `b)` causes starvation.

   b) Conditional variable is a queue that allows threads to be put themselves on to sleep (in blocked state) when thread it is not desired using `pthread_cond_wait` function.

   Since there are no threads inside `cv1`, there is nothing to awake using `pthread_cond_signal`.

   So, nothing will occur.

   c) System call is a subset of interrupt caused by user application to switch from user mode to kernel mode to perform previleged operations for the application.

   Interrupt is a signal sent by hardware (e.g keyboard, mouse, hard drive) or software.

   It tells the cpu to stop its activities and execute appropriate part of the operating system.

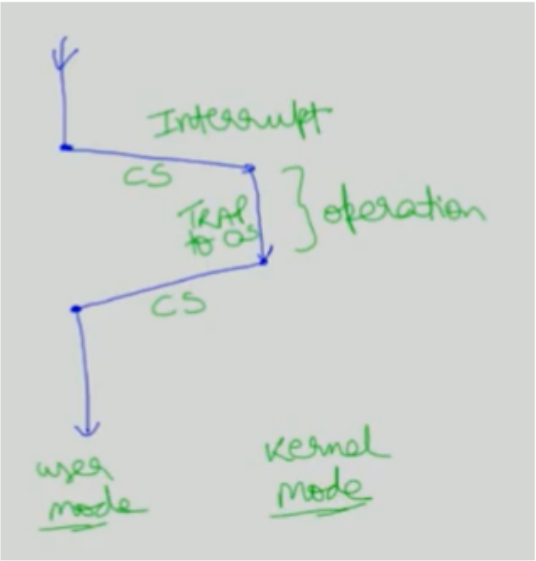
   Notes

   - I need to review how interrupt works. I had to look up the information.

   Question How does interrupt work?

   - **Interrupt**
     - Is a signal
     - there are two types of interrupts:
       - Hardware interrupt
         - Is signal generated by hardware (e.g RAM is full, Hard drive is full)
         - Is sent to operating system
       - Software interrupt
         - Is signal generated by software (e.g program crash, system call)

- Is sent to operating system
- May call trap instruction (esp. system call)



References

1. venkatesan ramachandran, What is an Interrupt?, link

d) No. This statment is false.

User level threads are generated in user-mode without kenerel being aware about it.

Notes

Question What is the difference between user-level thread and kernel-level thread?

Question Why is thread that is generated at user level using procedure call faster than kernel level thread?

Question What is procedure call? How does it work?

- **Procedure call**
  - works in user-mode only
  - doesn't require context switching
  - doesn't need help from OS/Kernel
  - no context-switching → faster

References

1. Tech Dose, System call vs Procedure call, link

e) System calls do not generate processes. `fork()` does.

With this reason the program `run_stuff` generates only 1 additional process.

Notes

Question What is a process? And how does process work?

Question How come system call doesn't generate process? And how come fork() generates process?

- **Process**
  - Is a running program
  - Has 3 states
    1. **Running:**
       - means a processs is running on a processor
       - means instructions are being executed
    2. **Ready:**
       - means a process is ready to run
       - means OS has chosen not to run the program at the given moment
    3. **Blocked:**
       - means a process has performed some kind of operation that makes it not ready to run until some event takes place

4.
```c
typedef struct acct {
    float balance;
    pthread_mutex_t lock;
    pthread_cond_t cond;
} account;

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

    // lock critical section during the transfer process
    pthread_mutex_lock(&a1->lock);
    pthread_mutex_lock(&a2->lock);
    // transfer amount
    a1->balance -= amount;
    a2->balance += amount;
    pthread_mutex_lock(&a1->lock);
    pthread_mutex_lock(&a2->lock);

    // lock the transferring user if the balance is negative
    if (a1->balance < 0) {
        pthread_cond_wait(&a1->cond, &a1->lock);
    }

}
```

Correct Solution

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

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

    pthread_mutex_lock(&a1->lock);
        a1->balance -= amount;

        while (a1->balance < 0) {
            pthread_cond_wait(&a1->cond, &a1->lock);
        }
    pthread_mutex_lock(&a1->lock);

    pthread_mutex_lock(&a2->lock);
        a2->balance += amount;

        if (a2->balance > 0) {
            pthread_cond_signal(&a2->cond);
        }
    pthread_mutex_lock(&a2->lock);
}
```

Notes

- Realized that I do not know how to create barriers to critical section.

Question When do we use the while loop like lock?

Question Does the use of if statement to put thread into sleep acceptable?

Question How can we construct safe barriers around critical section?

- **Locks**
  - Ensures that any critical section executes is a signle atomic operation
  - Guarentees that no more than single code can be active within the code
- `pthread_mutex_lock`
  - **Syntax:** `pthread_mutex_t VAR`
  - Is used to provide mutual exclusion between threads
  - If mutex is already locked, thread blocks until mutex is available
  - Must be proerly initialized before use

Static Way

`pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER`

Dynamic Way

```
int rc = pthread_mutex_init(&lock NULL);
assert(rc == 0);
```

5. a) It would favour I/O bound process. I/O bound process are mostly about waiting for the completion of input or output.

   And example of this is continuous typing on microsoft word.

   Usually the I/O-bound process lasts a short period of time.

   On the other hand, CPU-bound process involves the execution of algorithm that requires a huge computation time.

   An example of this is running a simulation.

   Because of this, CPU-bound process usually lasts a long period of time.

   Because of this, the processing algorithm would favour I/O bound process over CPU bound process.

   b) Yes.

   The processing algorithm favours algorithm with a short processing time in the past.

   Using the explanation given in part a), we can write I/O bound processes are favoured over CPU-bound process

   It follows from this information that if short I/O bound processes keep coming in, then CPU-bound process will never get a chance to run.

   So, we can conclude the scheduling algorithm would cause starvation to CPU-bound processes.

6. 1) a) False
      b) True
      c) True
      d) False
      e) False
      f) False

   2) a)
      b)
      c) Is a software interrupt sent by user application, so it traps into kernel mode, perform previliged operation, return to user mode from trap, and continue application with the returned result.