

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 privileged 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**

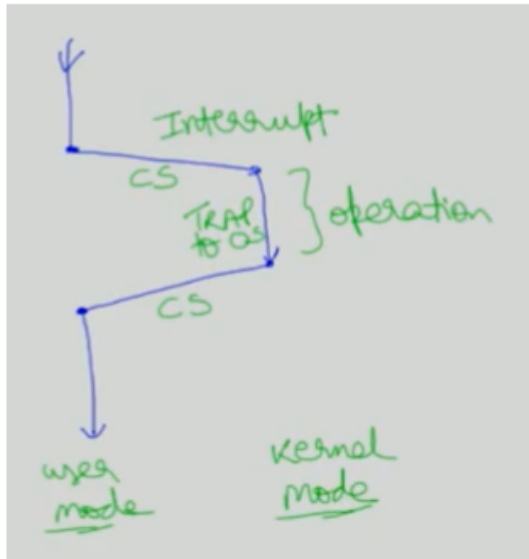
- I need to review how interrupt works. I had to look up the information.

**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 statement is false.

User level threads are generated in user-mode without kernel 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 process 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

```

41  typedef struct acct {
2      float balance;
3      pthread_mutex_t lock;
4      pthread_cond_t cond;
5  } account;
6
7  void transfer_amount(account *a1, account *a2, float amount) {
8
9      // lock critical section during the transfer process
10     pthread_mutex_lock(&a1->lock);
11     pthread_mutex_lock(&a2->lock);
12     // transfer amount
13     a1->balance -= amount;
14     a2->balance += amount;
15     pthread_mutex_lock(&a1->lock);
16     pthread_mutex_lock(&a2->lock);
17
18     // lock the transferring user if the balance is negative
19     if (a1->balance < 0) {
20         pthread_cond_wait(&a1->cond, &a1->lock);
21     }
22
23 }
```

**Correct Solution**

```

1  typedef struct acct {
2      float balance;
3      pthread_mutex_t lock;
4      pthread_cond_t cond;
5  } account;
6
7  void transfer_amount(account *a1, account *a2, float amount) {
8
9      pthread_mutex_lock(&a1->lock);
10     a1->balance -= amount;
11
12     while (a1->balance < 0) {
13         pthread_cond_wait(&a1->cond, &a1->lock);
14     }
15     pthread_mutex_lock(&a1->lock);
16
17     pthread_mutex_lock(&a2->lock);
18     a2->balance += amount;
19
20     if (a2->balance > 0) {
21         pthread_cond_signal(&a2->cond);
22     }
23     pthread_mutex_lock(&a2->lock);
24 }

```

**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 single atomic operation
  - Guarantees 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 properly 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) Is one of simplest lock to build. It allows only one thread to enter at a time. The variable lock has two values 0 (free/available), 1 (in use/locked). Spinlock uses two operations. One is `acquire()` and another is `release()`. `acquire()` uses while loop and `test-and-lock` function to ensure atomic operation in critical section. A thread is in locked state until lock is freed.

- b) Is the amount of time where processes in the same queue runs until it repeats. The time slice or scheduling quantum must be a multiple of the time of timer-interrupt.
- c) Is a software interrupt sent by user application, so it traps into kernel mode, perform privileged operation, return to user mode from trap, and continue application with the returned result.

### Notes

- I feel weak about scheduling quantum

**Question** What is scheduling quantum?

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

### Correct Solution

- a) **False**
- b) True
- c) False
- d) True
- e) False
- f) False

### Notes

- I feel weak about PCB and context switch

**Question** What is response for changing from user mode to kernel mode on system call interrupt?

**Question** What is program counter?

**Question** What is stack pointer?

**Question** What is user register?

**Question** What is kernel state?

- 2) a) Is a function that is a part of spin lock. It is used as a conditional statement in while loop to make sure only one thread can enter the critical at a time. Test and lock returns false if lock variable in spin lock is freed (with value of 0).

**Correct Solution**

Is an atomic instruction. It is used to implement synchronization algorithm such as spinlock.

**Notes**

- I feel weak about test-and-set
- b) Is a type of scheduling algorithm where one process is favoured and executed first over the others. Some examples are MLFQ and SJF scheduling algorithm.

**Correct Solution**

Is a type of scheduling algorithm where a process can be interrupted by an OS. Once the time slice of a process is used, the next process is scheduled into running state.

**Notes**

- I don't feel confident about preemptive scheduling and non-preemptive scheduling

**Question** What is preemptive scheduling?

8. a) c) is the only scheduling algorithm that minimizes average wait time.
- b) A conditional variable is a queue where an undesired thread can be added and put to sleep (by switching to blocked state) by calling the function `pthread_cond_wait`.

When state is changed, one or more threads in conditional variable can be awakened.

A thread in conditional variable is put to sleep using the function `pthread_cond_wait`

And a thread in conditional variable is awakened using the function `pthread_cond_signal`

Semaphore is a signal. It uses integer variable shared between multiple threads.

Semaphore has two types: counting semaphore and binary semaphore.

Counting semaphore has count variable where `count = 0` means resource is not available.

Binary semaphore has count variable with two states 0 - locked / not available  
1 - available/free.

A situation where a conditional variable is favored over the other variable is during banking transaction. Here, we use conditional variable to put a particular thread to sleep if condition is not satisfied (i.e balance remaining is negative).

**Notes**

- I feel weak about conditional variable and semaphores

**Question** What is conditional variable?

**Question** What is semaphore?

- I feel that conditional variable is used to control specific thread where as semaphore is used to control a group of thread (like a traffic)

c)