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

Correct Solution

a) False
b) True
c) True
d) False

(I am not too sure. But this may be due to the system setting up limited direct execution (baby proofing the CPU) before resuming the program)

e) False

(I am not too sure. But this may be because the lock spins using CPU cycle until spinlock is available for the process. Using this, the process may be in running state and not the blocked state)

f) False

Notes

- **User Mode**
  - Is restricted
  - Executing code has no ability to *directly* access hardware or reference memory [1]
  - Crashes are always recoverable [1]
  - Is where most of the code on our computer / applications are executed [3]
- **Kernel Mode**
  - Is previleged (non-restricted)
  - Executing code has complete and unrestricted access to the underlying hardware [3]

  - Is generally reserved for the lowest-level, most trusted functions of the operating system [1]
  - Is fatal to crash; it will halt the entire PC (i.e the blue screen of death) [3]

- **Interrupt**
  - Are signals sent to the CPU by external devices, normally I/O devices. [2]
  - Tells the CPU to stop its current activities and execute the appropriate part of the operating system (**Interrupt Handler**). [2]
  - Has three different types [2]

    1) **Hardware Interupts**
       * Are generated by hardware devices to signal that they need some attention from the OS.
       * May be due to receiving some data

         Examples

         · Keystrokes on the keyboard
         · Receiving data on the ethernet card

       * May be due to completing a task which the operating system previous requested

         Examples

         Transfering data between the hard drive and memory

    2) **Software Interupts**

       * Are generated by programs when a system call is requested

    3) **Traps**

       * Are generated by the CPU itself
       * Indicate that some error or condition occured for which assistance from the operating system is needed

- **Content Switch**
  - Is switching from running a user level process to the OS kernel and often to other user processes before the current process is resumed
  - Happens during a timer interrupt or system call
  - Saves the following states for a process during a context switch
    * Stack Pointer

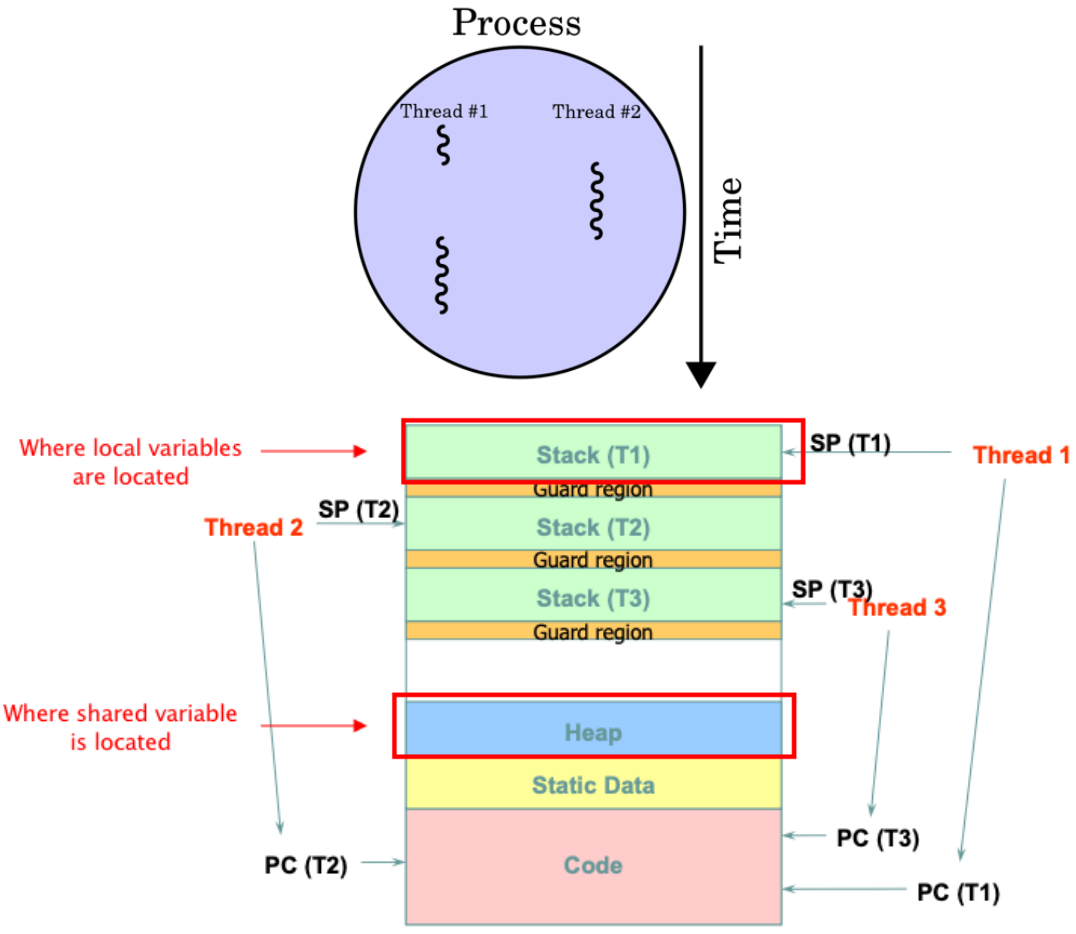
    - Program Counter
    - User Registers
    - Kernel State
  - May hinder performance

- **System Call**

  Example

  - `yield()`
    - Is a system call
    - Causes the calling thread to relinquish the CPU
    - Places the current thread at the end of the run queue
    - Schedules another thread to run

- **Thread**
  - Is a lightweight process that can be managed independently by a schdeduler [4]
  - Improves the application performance using parallelism. (e.g peach)



  - A thread is bound to a single process
  - A process can have multiple threads

  - Has two types
    - **User-level Threads:**
      - Are implemented by users and kernel is not aware of the existence of these threads
      - Are represented by a program counter(PC), stack, registers and a small process control block
      - Are small and much faster than kernel level threads
    - **Kernel-level Threads:**
      - Are handled by the operating system directly
      - Thread management is done by the kernel
      - Are slower than user-level threads

- **Process**
  - Is a program in execution
  - Is named by it's process ID or PID
  - Can be described by the following states at any point in time
    - Address Space
    - CPU Registers
    - Program Counter
    - Stack Pointer
    - I/O Information

    (wait. this is PCB)
  - Exists in one of many different **process states**, including
    1. Running
    2. Ready to Run
    3. Blocked

    - Different events (Getting Scheduled, descheduled, or waiting for I/O) transitions one of these states to the other

- **Signals**
  - Provides a way to communicate with the process
  - Can cause job to stop, continue, or terminate
  - Can be delivered to an application
    - Stops the application from whatever its doing
    - Runs Signal handler (some code in application to handle the signal)
    - When finished, the process resumes previous behavior

- **Spinlock**
  - Is the simplest lock to build
  - Uses a lock variable

  - 0 - (available/unlock/free)
  - 1 - (acquired/locked/held)
- Has two operations
  1. `acquire()`

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
boolean lock;

void acquire(boolean *lock) {
       while(test_and_set(lock));
}
```

  2. `release()`

```
void release(boolean *lock) {
       *lock = false;
}
```

- Allows a single thread to enter critical section at a time
- Spins using CPU cycles until the lock becomes available.
- May spin forever

- **Scheduling policies**
  - Are algorithms for allocating CPU resources to concurrent tasks deployed on (i.e., allocated to) a processor (i.e., computing resource) or a shared pool of processors [5]
  - Are sometimes called **Discipline**
  - Covers the following algorithms in textbook
    - **First In First Out**
    - **Shortest Job First**
    - **Shortest Time-to-completion First**
    - **Round Robin**
      - Runs job for a **time slice** or **quantum**
      - Each job gets equal share of CPU time

- Is clock-driven [6]
- Is starvation-free [7]
- Must have the length of a time slice (**quantum**) as multiple of timer-interrupt period

```
void release(boolean *lock) {
        *lock = false;
}
```

* **Multi-level Feedback Queue**

References

1) Coding Horror, Understanding User and Kernel Mode, link
2) Kansas State University, Basics of How Operating Systems Work, link
3) Kansas State University, Glossary, link
4) Tutorials Point, User-level threads and Kernel-level threads, link
5) Science Direct, Scheduling Policy, link
6) Guru 99: What is CPU Scheduling?, link
7) Wikipedia: Round-robin Scheduling, link

2. a) Is a simple form of lock. It allows a single thread to enter ciritical section at a time. It spins using CPU cycles until lock becomes available. It uses variable `lock` with two values (0 for available, 1 for acquired) with operations `acquire()` and `release()`.

   b) Is a fraction of total turnaround time for a process. Is used by round-robin scheduling algorithm. A process under RR has equal parts of these. Furthermore, scheduling quantum assigned to a process to be multiples of timer-interrupt period

   Correct Solution

   In a preemptive scheduler, this is the time alloted to a process before context-switching to another process

   c) Is the programmatic way in which user requests for previleged service in operating system. On system call, the current process states (program counter, CPU register, kernal state) are saved, enters kernel mode, performs previleged operations such as reading the disk, executes return-from-trap instructions, and returns back to user mode and resumes the program with the attained result

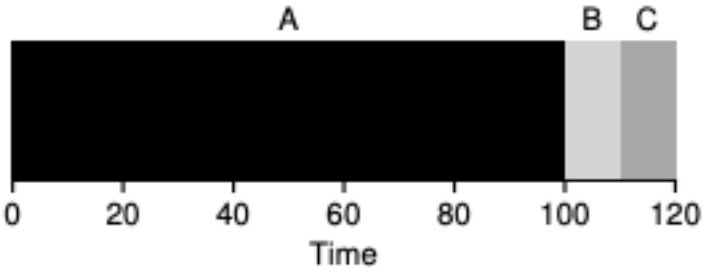
Notes

- **Response Time**
  - **Formula** $T_{response} = T_{firstrun} - T_{arrival}$
  - measures the interactive performance between users and the system
- **Turnaround Time**
  - **Formula** $T_{turnaround} = T_{completion} - T_{arrival}$
  - measures the amount of time taken to complete a process
- **System Call**
  - Is the programmatic way in which a computer program requests a previleged service from the kernel of the operating system
  - i.e. Reading from disk
  - Steps
    1) Setup **trap tables** on boot
    2) Execute system call
    3) Save *Program Counter*, *CPU registers*, *kernal stack* (so process can resume after **return-from-trap** or **context switch**)
    4) Switch from **user mode** to **kernel mode**
    5) Perform previleged operations
    6) Finish and execute **return-from-trap** instruction
    7) Return from **kernel mode** to **user mode** and resume user program

3. a) `(b)` is the only scheduling algorithm that causes starvation

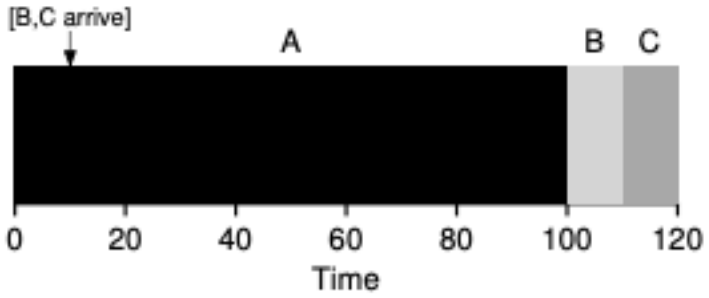
<u>Notes</u>

- **Starvation**
  - Is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time [1]
- **Convoy Effect**
  - Is the problem where number of relatively-short potential consumers of a resource get queued behind a heavy weight consumer

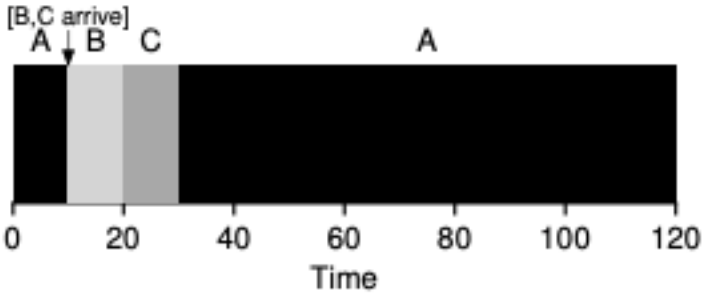


- **First In First Out**
  - Is the most basic scheduling algorithm
  - Is vulnerable to **convoy effect**

  - No **starvation** as long as every process eventually completes
- **Shortest Job First**
  - Improves average **turnaround time** given processes of uneven length
  - Is a general scheduling principle useful in situation where turnaround time per process matters
  - Is vulnerable to **convoy effect**



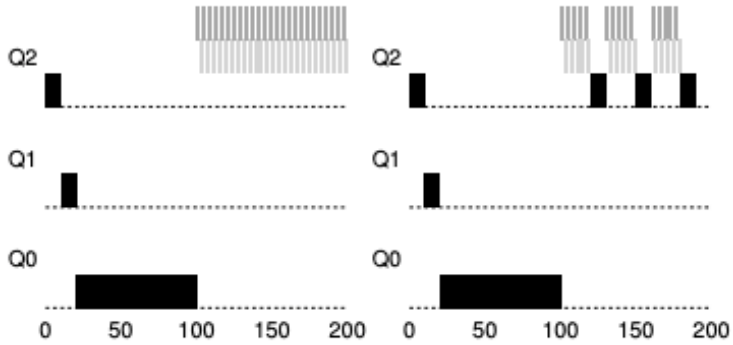
  - Is vulnerable to **starvation**
    - When only short-term jobs come in while a long term job is in queue
- **Shortest Time-to-completion First**
  - Addresses **convoy effect** in **Shortest Job First**
  - Determines which of the remaining+new jobs has least time left, and schedule accordingly at any time



  - Is vulnerable to **starvation**
    - When only short-term jobs come in while a long term job is in queue
- **Round Robin**
  - Has good **response time** but terrible **turnaround time**
  - Runs job for a **time slice** or **quantum**
  - Each job gets equal share of CPU time
  - Is clock-driven [6]
  - Is starvation-free [7]
  - Must have the length of a time slice (**quantum**) as multiple of timer-interrupt period

```
void release(boolean *lock) {
        *lock = false;
}
```

- **Multi-level Feedback Queue**
  - Is the most well known approaches to shceduling
  - Optimizes **turnaround time**, and minimizes **response time**
  - Observees the execution of a job and priortizes accordingly without prior knowledge
  - Rules
    - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't)
    - **Rule 2:** If Priority(A) = Priority(B). A & B run in round-robin fashion using the time slice (quantum length) of the given queue
    - **Rule 3:** When a job enters the system, it is placed at the highest priority(the top most queue)
    - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (it moves down on queue)
    - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

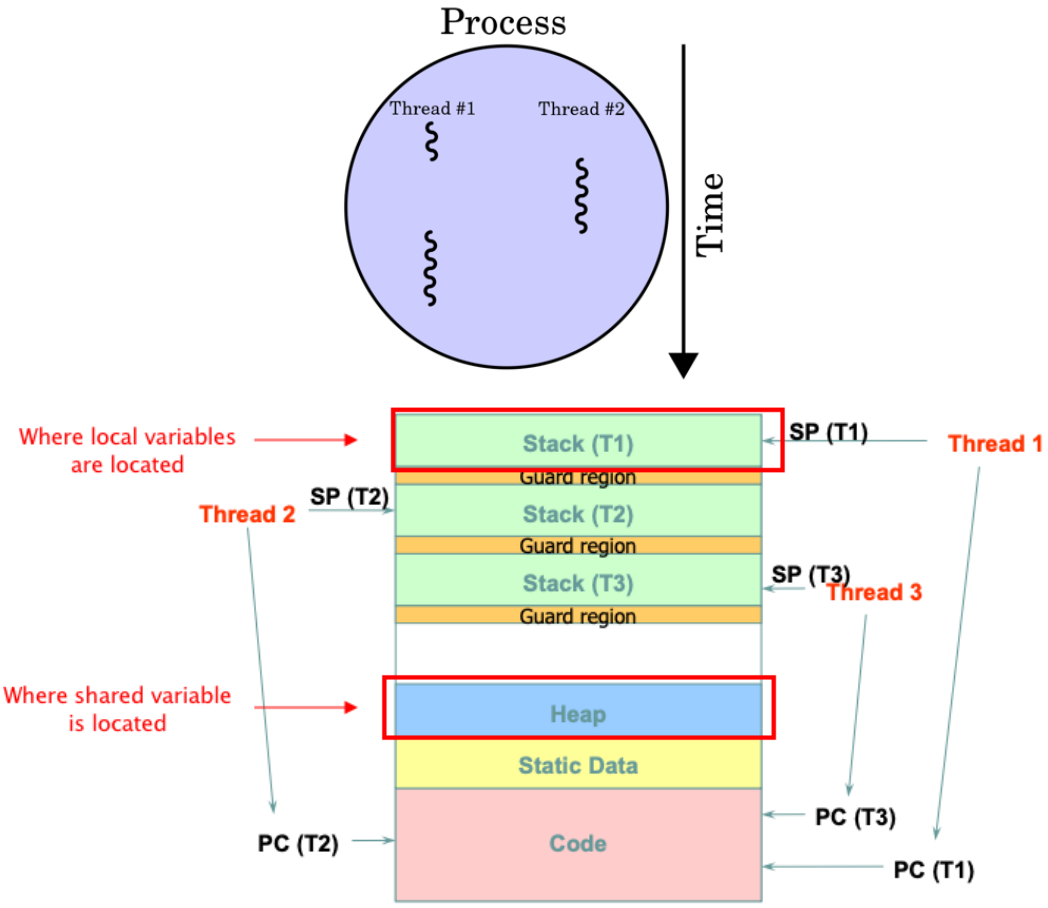


b) Nothing. `pthread_cond_signal` unblocks at least one of the thread that are blocked on the specified condition variable `cond`. Since there are no other threads waiting on `cv1`, there are no threads to awake.

Notes

- **Critical Section**
  - Is a piece of code that accesses a *shared* resource, usually a variable or data structure
- **Thread**
  - Is a lightweight process that can be managed independently by a schdeduler [4]

- Improves the application performance using parallelism. (e.g peach)



  - A thread is bound to a single process
  - A process can have multiple threads
  - Has two types
    - **User-level Threads:**
      - Are implemented by users and kernel is not aware of the existence of these threads
      - Are represented by a program counter(PC), stack, registers and a small process control block
      - Are small and much faster than kernel level threads
    - **Kernel-level Threads:**
      - Are handled by the operating system directly
      - Thread management is done by the kernel
      - Are slower than user-level threads

- **Thread API**
  - `pthread_cond_wait`
    - Puts the calling thread to sleep (a blocked state)
    - Waits for some other thread to signal it

Example

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

Puts calling thread cond to sleep

- `pthread_cond_signal`
  - Is used to unblocks at least one of the threads that are blocked on the specified condition variable cond

Example

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Wakes a thread that's been put to sleep on cond variable

c)