

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

## Notes

### • User Mode

- Is restricted
- Executing code has no ability to *directly* access hardware or reference memory<sup>[1]</sup>
- Crashes are always recoverable<sup>[1]</sup>
- Is where most of the code on our computer / applications are executed<sup>[3]</sup>

### • Kernel Mode

- Is privileged (non-restricted)
- Executing code has complete and unrestricted access to the underlying hardware<sup>[3]</sup>
- Is generally reserved for the lowest-level, most trusted functions of the operating system<sup>[1]</sup>
- Is fatal to crash; it will halt the entire PC (i.e the blue screen of death)<sup>[3]</sup>

### • Interrupt

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

#### 1) **Hardware Interrupts**

- \* 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 previously requested

### Examples

Transferring data between the hard drive and memory

## 2) **Software Interrupts**

- \* Are generated by programs when a system call is requested

## 3) **Traps**

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

### • **Context 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

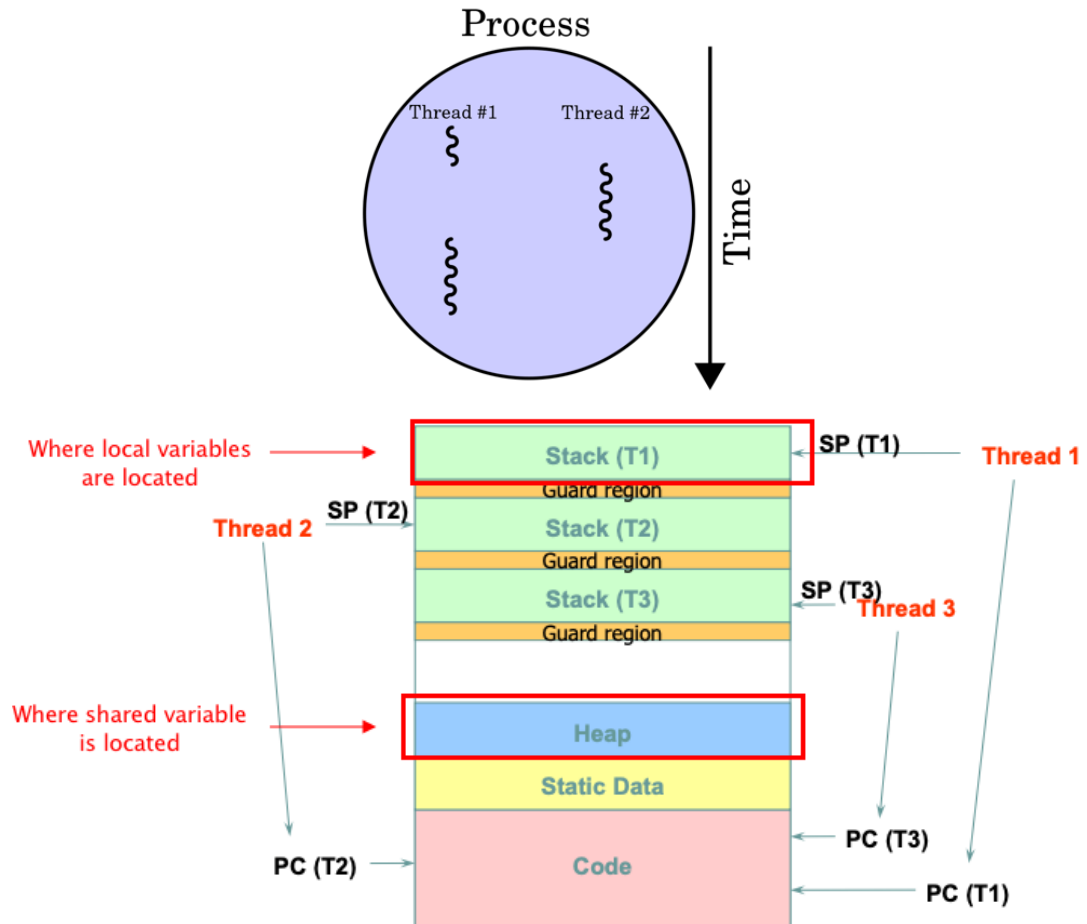
### • **System Call**

#### 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 scheduler <sup>[4]</sup>
- 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](#)