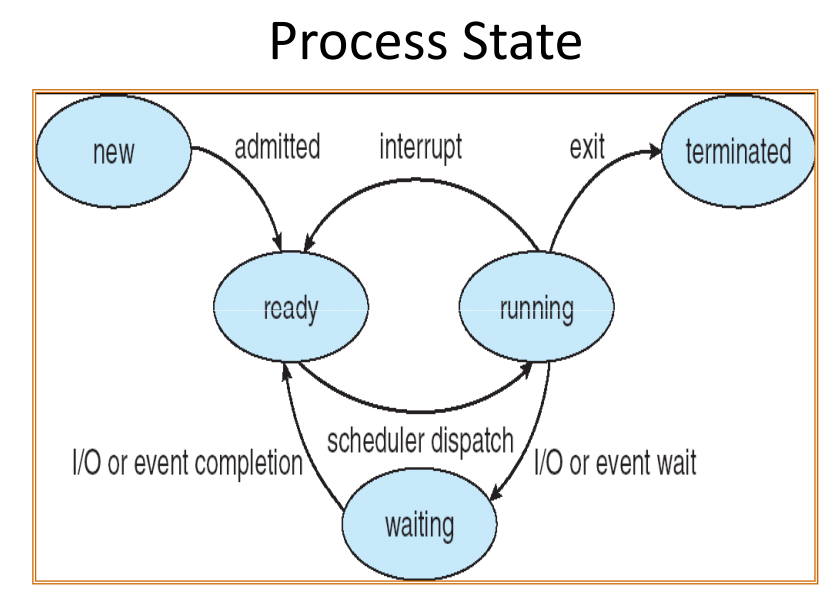
2017

1. d



1. d

answer a is correct but it depends on the scheduler used, preemptive or not, lacking the information, answer a is false

3) b

This can occur when a running process is blocked due to one of the following

reasons:

* Process need to perform I/O operation
* Waiting for some event to occur such as a non busy waiting semaphore

There for we can conclude that the process is put to blocked state when it is waiting for a long operation to finish but its a bit ambigue.

4) a

A transition from Running state to Ready state can happen only in case the OS uses a preemptive scheduler. Some examples of preemptive scheduling algorithms are

* Round Robin Scheduling Algorithm
* Preemptive Priority Based Scheduling Algorithm
* Shortest Remaining Job First Algorithm

In each of the above algorithms the scheduler can select another process to RUN, while some process is already running on the Processor. In such a situation the Running process is said to be "PREEMPTED" and moved from Running state to Ready queue

5) d

by elimination but normally the process passes from block to ready when the I/O operation or event for which the process was waiting gets completed.

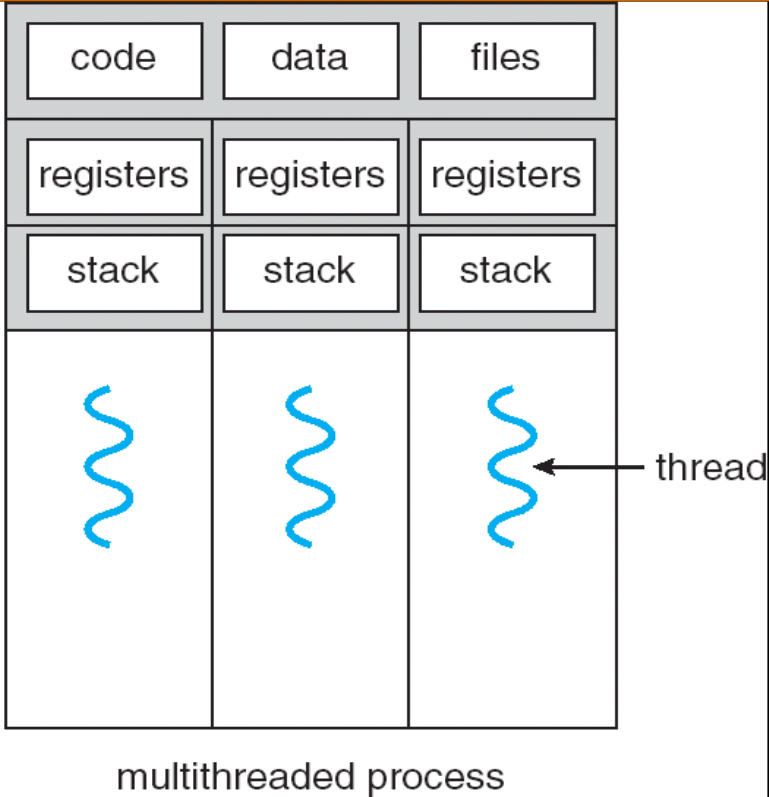
**In this case why is not a ?**

**“For some unknown reason” -> trop vague=**

6) b

there is only one file descriptor table per process, and it's shared among all the threads.

**Plus, each thread has its own stack and registry, so it is not shared amongst them.**



7) b

The inode contains the following pieces of information:

* Mode/permission (protection)
* Owner ID
* Group ID
* Size of file
* Number of hard links to the file
* Time last accessed
* Time last modified
* Time inode last modified

8) a



9) a

because turnaround time = EndTime - Arrival Time ⇒ For T3 : 16 - 1 = 15

10) d

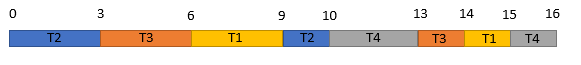
Waiting time = Turnaround Time - Running Time = 15 - 4 = 11

11) a



doesn’t change because time quantum >= exec time(max)

12) b



13) c

because turnaround time = EndTime - Arrival Time ⇒ For T3 : 14 - 1 = 13

14) d

A process runs on the CPU until it is [context switched](http://osr507doc.sco.com/en/PERFORM/PERFORM_Glossary.html#glP_context_switch). This happens when one of the following occurs:

* The process exits.
* The process uses up its time slice.
* The process requires another resource that is not currently available or needs to wait for I/O to complete.
* A resource has become available for a sleeping process. If there is a higher priority process ready to run, the kernel will run this instead (the current process is [preempted](http://osr507doc.sco.com/en/PERFORM/PERFORM_Glossary.html#glP_preemption)).
* The process relinquishes the CPU using a semaphore or similar system call.

15) c

Task Synchronization. In a multitasking environment, you sometimes need to synchronize the order of operations between two or more tasks. For example, some tasks may need the results of another task before executing. Semaphores can be used to synchronize tasks.

16) c

outcome 45 : the three processes access the variable in “the same time” there for they don’t take in consideration the changes made by the other processes.

outcome 46: Only two processes access the variable in “the same time” there for they don’t take in consideration the changes made by the other process.

outcome 47: None of the processes access the variable in “the same time” there for they take in consideration the changes made by the other processes.

17) a

The semaphore is set to value 0 this mean that no one can get in the semaphore UNLESS it receives a signal(sem\_post or a release in this case) which will increment to 1 the value of the semaphore and there for let one process get in but unfortunately it doesn’t get any signal and that why all the process are stuck and the value doesn’t increment

18) c => 66,67,68,69

This time the semaphore is set to 2, this means that two processes can get in the semaphore at the same time. This implies that two processes can access the variable at the same time, this is why we get differents outcomes:

outcome 66: possible, imagine that the first one that gets in the semaphore, stays in the semaphore for a long time (because there is a sleep of 10s for example). The other three processes have time to go in the semaphore one by one and make their changes to the variable “i” but the last one to makes its incrementation is process 1 and at the time he got in the semaphore, the variable was equal to 65 and therefore he increments it to 66 and replace the value of the variable incremented by the other processes(68) to 66.

outcome 67: possible, the first two processes go in the semaphore and access in “the same time” the variable and make their changes without taking in consideration the changes made by the other process and both of them finish in “the same time” allowing the two other process in the semaphores to access the variable in “the same time” and there for not taking in consideration the incrementation made by the other process in the semaphore. This is why the result is 67.

outcome 68:possible, two process arrived in the semaphore, either the first two or the last two.

outcome 69: possible, they all arrive at different time in the semaphore and there for taking in consideration the changes made by all the other processes.

19) c

We use (8,8), because we use addresses over **16 bits**. In the other two responses, we’re not using 16 bits, so by elimination it is c.

**-> pas sur**

20) c

