Name: Nishanshi Shukla

Id: 121030

1. READY to SUSPEND: No more memory is available, so the READY process is temporarily swapped out of memory

READY to RUN: The process is allocated the CPU by the scheduler

READY to BLOCKED: Not possible

RUN to READY: Time allocated to the particular process expires

RUN to BLOCKED: I/O request of the process is fulfilled

RUN to SUSPEND: Not possible

BLOCKED to READY: The awaited event completes (e.g. I/O completion)

BLOCKE D to RUN: Not possible

SWAPPED to READY or SWAPPED to BLOCKED: When memory becomes available or the reason for swapping is no longer true

1. T = 22: P5, P8 - ready/running P1, P3, P7 - blocked for I/O

T = 37: P1, P3, P8 - ready/running P5 - blocked suspend (swapped out) P7 - blocked for I/O

T = 47: P1, P3, P5 - ready/running P7 - blocked for I/O P8 - exit

1. Possible outcomes:

<child pid>

0

2. 0

<child pid>

1. Mode Switch in Threads

Memory is shared by threads, so there's no need to exchange memory or data during thread creation or switching. Switching process requires OS to process more information.

Mode Switch in Process

Thread switching does not require kernel to get involved, which in turn saves time on switching user to kernel mode. Kernel is involved so more time is needed while switching mode

1. Since kernel is completely unconcerned about the ULTs, they can run on any OS without changing anything in the codes. Since all of the thread management structures are within the user address space of a single process, the process does not switch to the kernel mode to do thread management. Thus saving time and no mode switching needed.Scheduling can be application specific. So depending on the need the scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler.
2. When a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked. In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time. Therefore, only a single thread within a process can execute at a time.

1. Threads are only present at User level while they don't exist for the kernel. All the scheduling is done by the kernel and it treats a process as a unit assigning only one state to it at a time as to kernel the threads are not visible and only the process exists. Hence once one thread is blocked, the whole process is blocked and consequently all threads in that process are blocked.
2. Here the program spends most of its time waiting for I/O operation to complete. In a multithreaded program, one KLT can make the blocking system call, while the other KLTs can continue to run. On a uniprocessor machine, a process that would otherwise have to block for all these calls can continue to run its other threads.
3. No. Because the termination of process causes the process to terminate
4. Competing Process: Compete for resources and don't share/cooperate. This may lead to a race condition which OS needs to regulate. Cooperating Process: These processes may or may not be aware of each other but they share resources. They work jointly by sharing od resources by cooperating
5. Strong Semaphores: They strictly follow order (FIFO) to remove the processes from waiting queue. Weak semaphore: No such order is followed
6. Monitors are programming language construct that provides equivalent functionality to that of semaphores and is easier to control.
7. Blocking send, Blocking receive: Both sender and receiver are blocked until the message is delivered. Sometimes referred to as a rendezvous. Allows for tight synchronization between processes. Non-Blocking send, Blocking receive: Sender continues on but receiver is blocked until the requested message arrives. Most useful combination: Sends one or more messages to a variety of destinations as quickly as possible. Non-Blocking send, Non-Blocking receive: Neither party is required to wait .
8. No. Busy waiting can be more efficient if the expected wait time is shorter than the time it takes to pre-empt and re-schedule a thread.
9. Both codes prove out to be the same except that the semaphore code returns the value of number of processes waiting which is not given by the other code.