# HOME ASSIGNMENT 2

|  |  |
| --- | --- |
| 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)  BLOCKED to RUN: Not possible  SWAPPED to READY or SWAPPED to BLOCKED: When memory becomes available or the reason for swapping is no longer true |
| 2 | 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 |
| 3 | Pid of the child process or 0 |
| 4 | Thread mode switch does not invoke kernel while process mode switching does. Hence thread mode switching is fast. |
| 5 | * 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. |
| 6 | * 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. |
| 7 | When a thread issues I/0 request the corresponding process under which the thread is running takes this request to kernel and kernel block it until the I/O operation is terminated. Due to this blocking of process, threads gets blocked. |
| 8 | 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. |
| 9 | No |
| 10 |  |
| 11 | Strong Semaphores: They strictly follow order (FIFO) to remove the processes from waiting queue  Weak semaphore: No such order is followed |
| 12 | Monitors are programming language construct that provides equivalent functionality to that of semaphores and is easier to control |
| 13 | 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  Example -- a service process that exists to provide a service or resource to other processes  Non-Blocking send, Non-Blocking receive:  Neither party is required to wait |
| 14 | 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. |
| 15 | 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. |

NOTE: Certain questions are directly referred from PPT of William Stalling