OS-HOME ASSIGNMENT 2

1. RUN to READY : This can happen when time out of currently running process happens and it is kept in ready queue.

READY to RUN : This event occurs when the currently process in run state blocks for an event wait or time out occurs. Hence the process from ready queue goes for execution.

RUN to BLOCKED : This event can occur if a process makes a system call.

BLOCKED to READY : This event occurs if the awaited event completes (perhaps I/O completion)

READY to NONRESIDENT :This event occurs if memory is overcommitted, and a process is temporarily swapped out of memory.

BLOCKED to NONRESIDENT - same as READY to NONRESIDENT.

1. At time 22-

P1: Blocked for I/O

P3: Blocked for I/O

P5: Ready Running

P7: Blocked for I/O

P8: Ready Running

At time 37-

P1: Ready Running

P3: Ready Running

P5: Blocked for I/O

P7: Blocked for I/O

P8: Ready Running

At time 47-

P1: Ready Running

P3: Ready Running

P5: Ready Suspended

P7: Blocked for I/O

P8: Exit

1. 0

<child pid>

/

<child pid>

0

1. The reasons why a mode switch between threads is better than a process switch is-

* The control blocks for processes are larger than for threads (hold more state information), so the amount of information to move during the thread switching is less than for process context switching.
* The major reason is that the memory management is much simpler for threads than for processes. Threads share their memory so during mode switching, memory information does not have to be exchanged/changed, pages and page tables do not have to be switched, etc. This makes the thread context switch much cheaper than for processes. In case of processes the memory pieces (pages) need to be exchanged, etc. (Will talk about the details in few weeks).
* Threads do not have to worry about accounting, etc, so do not have to fill out all the information about accounting and other process specific information in their thread control block, so keeping the thread control block consistent is much faster .
* Threads share files, so when mode switch happens in threads, these information stay the same and threads do not have to worry about it and that makes the mode switch much faster.

1. 1. Thread switching does not require kernel mode privileges because all of the thread management data structures are within the user address space of a single process. Therefore, the process does not switch to the kernel mode to do thread management. This saves the overhead of two mode switches (user to kernel; kernel back to user). 2. Scheduling can be application specific. One application may benefit most from a simple round-robin scheduling algorithm, while another might benefit from a priority-based scheduling algorithm. The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler. 3. ULTs can run on any operating system. No changes are required to the underlying kernel to support ULTs. The threads library is a set of application-level utilities shared by all applications.
2. In a typical operating system, many system calls are blocking. Thus, when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked. 2. 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.
3. Because, with ULTs, the thread structure of a process is not visible to the operating system, which only schedules on the basis of processes.
4. The issue here is that a machine spends a considerable amount of its waking hours waiting for I/O to complete. In a multithreaded program, one KLT can make the blocking system call, while the other KLTs can continue to run. On uniprocessors, a process that would otherwise have to block for all these calls can continue to run its other threads.
5. Yes.If a process exists then all the threads of that process will also stop running.

|  |  |
| --- | --- |
| Competing Process | Cooperating Process |
| Competing process is the process which does its work independent of any other process present. | Cooperating process is the one which does its work in accordance with the other present processes. |
| This process would compete for the resources. | This process would share the resources with some other process and at times even complete a task together with other processes. |
| There is a careful isolation done among all the processes. | The processes are made to communicate and share with each other. |

11.

|  |  |
| --- | --- |
| Strong Semaphore | Weak Semaphore |
| It specifies the order in which the processes should be removed from the waiting queue. | It does not specify the order from which the process should be removed from the waiting queue. |
| Mostly used by all the Operating System | Rarely used by any operating system |

1. Monitor is a synchronization construct that allows the threads to wait for some event to occur and assure mutual exclusion between them. It is helpful for multiprogramming. With the help of monitors only one thread will be executed at a time.
2. **Send blocking :**   Either the sending process is blocked until the message is received, or it is not

**Receive blocking :** there are two possibilities:

If a message has been sent and is available, the message is received and the receiver continues execution.

If there is no message waiting, then either (a) the process is blocked until a message arrives, or (b) the process continuers to execute abandoning the receive attempt.

14. False. Busy waiting can be more efficient if the expected wait time is shorter than the time it takes to preempt and re-schedule a thread. This is common on multiprocessors.

15. Yes. They function the same.