**OS Home Assignment-2**

* **Raj Thakkar (121041)**

1. RUN to READY can be caused if the time-slice expires

READY to NONRESIDENT when a process is temporarily swapped out of memory if its overcommitted

READY to RUN occurs only if a process is activated to run on the CPU by the dispatcher

RUN to BLOCKED can occur if a process issues an I/O or other kernel request and has to wait till the event occurs.

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

BLOCKED to NONRESIDENT - when a process is temporarily swapped out of memory if its overcommitted.

|  |  |  |
| --- | --- | --- |
| **At t=22** | **At t=37** | **At t=47** |
| P1: Blocked for I/O | P1: Ready Running | P1: Ready Running |
| P3: Blocked for I/O | P3: Ready Running | P3: Ready Running |
| P5: Ready Running | P5: Blocked for I/O | P5: Ready Suspended |
| P7: Blocked for I/O | P7: Blocked for I/O | P7: Blocked for I/O |
| P8: Ready Running | P8: Ready Running | P8: Exit |

1. 0

<child pid>

or

<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.
* 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 resources, 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. Advantages of ULT over KLT:

**1.  User level** 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 OS. No changes are required to the underlying kernel to support ULTs. The threads library is a set of application-level functions shared by all applications.

1. Two disadvantages:

**1.**In a typical OS, many system calls are blocking. As a result, 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. In effect, we have application-level multiprogramming within a single process.

1. The ULT thread structure is invisible to the kernel. The kernel continues to schedule the process as a unit and assigns a single execution state (Ready, Running, Blocked, etc.) to that process. Hence once one thread is blocked, the whole process is blocked and consequently all threads in that process are blocked.
2. One to one mapping between ULT-KLT that allows one or more threads within a process to issue blocking system calls while others continue to run because KLT in multi-threaded program enables at least one thread to issue a block system call independently without influencing other threads to continue with their execution.

However, in single threaded counterparts of multi-threaded program, computer generally spends a significant amount of time waiting for I/O operation to be complete

1. 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. **Blocking send, blocking receive:**Both the sender and receiver are blocked until the message is delivered.

**Nonblocking send, blocking receive:**Although the sender may continue on, the receiver is blocked until the requested message arrives. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives. An example is a server process that facilitates a service or resource to other processes.

**Nonblocking send, nonblocking receive:**Neither sender nor receiver is required to wait.

1. **False.** Although Busy waiting wastes instruction cycles, they 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.
2. **Yes.** They can be substituted. The negative value gives the count of waiting processes. Apart from that the working remains same.