OPERATING SYSTEMS

HOME ASSIGNMENT 2

-Rujuta Shah

(121044)

1. RUN to READY can be caused by a time-quantum expiration

READY to NONRESIDENT occurs if memory is overcommitted, and a process is

temporarily swapped out of memory

READY to RUN occurs only if a process is allocated the CPU by the dispatcher

RUN to BLOCKED can occur if a process issues an I/O or other kernel request.

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

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>

or

<child pid>

0

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

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

* 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).
* 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.
* 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 .
* .
* The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
  + Simple Representation: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
  + Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
  + User-level threads does not require modification to operating systems.
  + Fast and Efficient: Thread switching is not much more expensive than a procedure call.

1. User-Level threads are not a perfect solution as with everything else, they are a trade off. Since,
   * User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.
   * User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, OS can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock. Solving this requires communication between between kernel and user-level thread manager.
   * There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
2. User process functions separately from Kernel processes. That is, thread structure of a process is not visible to the OS/kernel, which schedules on the basis of processes. 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.

1. 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
2. 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.

1. 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.
2. Yes. They function the same.