鄭琳玲
108540050
資工三

Operating System - HW2

Chapter 4

(4.2)

when a kernel thread has a page fault, another kernel thread might be swapped in to make use of the interleaving time. A single threaded process, on the other hand, will be unable to conduct valueable work if a page failure occurs. As a result, in circumstances where a program may experience frequent page faults or must wait for the other system events, a multi threaded solution would outperform a single processor system.

(4.4)

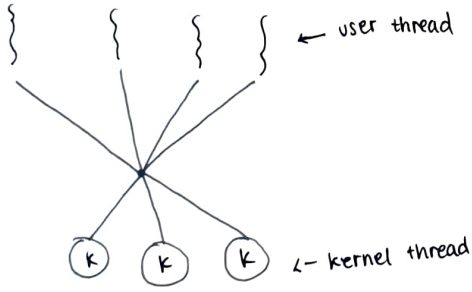
A multithreaded system with numerous user level threads can't utilise of the processors in a multiprocessor system at the same time. The operating system only sees a single process and will not schedule the process's several threads on distinct CPUs. As a result, there is no performance benefit to running several user level threads on a multiprocessor machine.

(4.13)

Kernel threads:

A kernel threads is an entity that handles the system scheduler, like processes & interruption operators. A kernel thread is running through the process, but any other thread in the system can be referenced. The programmer is without direct control of these threads except when you write kernel extensions or device drivers.

Threading model:



a) where the number of kernel threads assigned the program is below the processors, some processors are idle & system use isn't optimal.

b) If the number of kernel threads assigned to the program corresponds to the number of processors, there isn't no processor idle & system performance is optimized. But, please note that only when every thread is running on some processor & no thread is blocked is the performance optimal.

c) If the kernel threads are more than processors, a blocked kernel thread can be swapped to an executable kernel thread, increasing the use of system multiprocessor. Instead of another kernel thread that is ready to run, a blocked kernel thread can be swapped out.

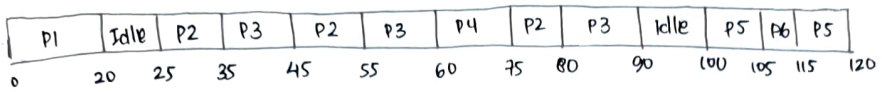
Chapter 5

(5.6)

The regressive round robin scheduler prefers the CPU bound processes. since CPU boond processes when uses its entire time quantum, they additionally gets 10 milliseconds as the time quantum as well as there priority gets boosted. The regressive will not prefers I/O-bound processes, since these processes can be blocked for I/O before consuming the full quota of time quantum, and their priority will not get effected, its mean priority will be the same as before.

(5.8)

a)



b) $P_1 = 20 - 0 = 20$

$P_2 = 80 - 25 = 55$

$P_3 = 90 - 30 = 60$

$P_4 = 75 - 60 = 15$

$P_5 = 120 - 100 = 20$

$P_6 = 115 - 105 = 10$

c) $P_1 = 0$

$P_2 = 40$

$P_3 = 35$

$P_4 = 0$

$P_5 = 10$

$P_6 = 0$

d) $\frac{105}{120} \times 100 = 87.5\ \%$

(5.10)

Starvation:

↳ a situation where a waiting process doesn't get allocated system resources because they are in use by other processes.

From the meaning stated above, shortest job first is a scheduling algorithm which like its names allocates resources to the process with the shortest job & thus causes starvation for other process. While priority scheduling gives system resources to the process which needs it most & causes starvation for the other processes which aren't on high priority list.

ANSWER: Shortest job first & priority.

(5.15)

FCFS

↳ discriminates against short jobs since any short jobs arriving after long jobs will have a longer waiting time.

RR

↳ treats all jobs equally by giving them both equal burst of CPU time, so short jobs will be able to leave faster since they will finish first.

Multilevel feedback queues

↳ the work is similar to RR algorithm, they discriminate favorably toward short jobs.

Chapter 6

(6.9)

Interrupts aren't sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled. There are no limitations on processes could be executing on other processors & therefore the process disabling interrupts can't guarantee mutually exclusive access to program state.

(6.10)

Semaphore is a special variable that can be accessed by two atomic system calls, known as p() & v() or wait() & signal(). which is a signaling system.

only a process or a task can aquire the mutex lock at a time since ownership is associated with a mutex lock & only the owner can release the mutex lock. It is a locking system.

Busy waiting problem is when a process is waiting to enter the critical section & testing the condition repeatedly; which only wasting the CPU cycle.

To solve the busy waiting problem, semaphore is able to make a queue & add coming process to the queue; so that it can go to its critical section. After the process is inserted to the queue, semaphore will block the process. When the semaphore value is incremented, or the resource is available, then the process will be waked by the semaphore & delered from the queue automatically.

(6.11)

Short duration

↳ use a spinlock, so that there will be much less overhead if the process waits 3-4 cycles then performing two context switches

Long duration

↳ use a mutex, so that given for spinlock in the previous answer can't hold in long duration.

Holding the lock.

↳ use mutex, so that the thread that can't access the critical session will waste less cycles than if it was busy waiting.