HW2

Written Exercise

4.2: Under which circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system.

4.4: Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

No. User level threads are interpreted by the OS as only a single process and will not schedule different threads of the process on separate processors. That strongly suggests that there is no performance benefit to be had in this situation.

4.13: Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

(a) The number of kernel threads allocated to the program is less than the number of processing cores.

When the number of kernel threads is less than the number of processors, then some of the processors would remain idle since the scheduler maps only kernel threads to processors and no user-level threads to processors. *Some of the processors would remain idle.

(b) The number of kernel threads allocated to the program is equal to the number of processing cores.

When the number of kernel threads is exactly equal to the number of processors, then it is possible that all of the processors might be utilized simultaneously. However, when a kernel-thread block inside the kernel (due to a page fault or while invoking system calls), the corresponding processor would remain idle. *All of the processors might be utilized simultaneously.

(c) The number of kernel threads allocated to the program is greater than the number of processing cores.

When the kernel threads are more than processors, a block kernel thread could be swapped out instead of another kernel thread that is ready to execute, this will increase the utilization of the multiprocessor system. *A blocked kernel thread could be swapped out instead of another kernel thread that is ready to execute.

5.6: A variation of the round-robin scheduler is the regressive round-robin scheduler.

- This scheduler assigns each process a time quantum and a priority.
- The initial value of a time quantum is 50 milliseconds
- However, every time a process has been allocated to the CPU and uses it entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted.
- (The time quantum for a process can be increased to a maximum of 100 milliseconds.)
- When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the same

What type of process (CPU-bound, I/O-bound) does the regressive round-robin scheduler favor? Explain the reasons.

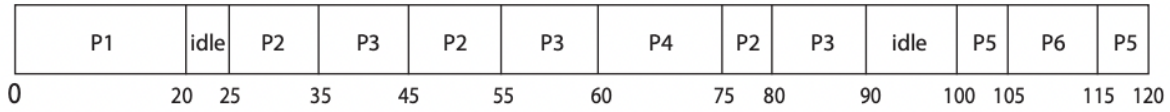
This scheduler would favor CPU-bound processes as they are rewarded with a longer time quantum as well as priority boost whenever they consume an entire time quantum. This scheduler does not penalize I/O-bound processes as they are likely to block for I/O before consuming their entire time quantum, but their priority remains the same.

5.8: The following processes are being scheduled using a preemptive round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an idle task (which consumes no CPU resources and is identified as Pidle). This task has priority 0 and is scheduled whenever the system has no other available processes to run.

The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

Thread	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

(a) Show the scheduling order of the processes using a Gantt chart.



(b) What is the turnaround time for each process?

P1: 20 - 0 = 20
P2: 80 - 25 = 55
P3: 90 - 30 = 60
P4: 75 - 60 = 15
P5: 120 - 100 = 20
P6: 115 - 105 = 10

(c) What is the waiting time for each process?

P1: 0
P2: 40
P3: 35
P4: 0
P5: 10
P6: 0

(d) What is the CPU utilization rate?

105 / 120 = 87.5%

5.10: Which of the following scheduling algorithms could result in starvation?

- First-come, first-served
- Shortest job first
- Round-robin
- Priority

Please explain the reasons.

FCFS = No starvation, everyone eventually gets its chance
Shortest job first = Longer processes will have more waiting time and eventually they will suffer starvations
Round Robin = Starvation doesn't really occur because everyone gets their chance after a quantum
Priority = lower priority process can experience starvation
*In conclusion, the shortest job first and the priority could result in starvation.

5.15: Explain the differences in how much the following scheduling algorithms discriminate in favor of short processes:

(a) FCFS

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

(b) RR

RR treats all jobs equally (giving them equal bursts of CPU time) so short jobs will be able to leave the system faster since they will finish first.

(c) Multilevel feedback queues

Multilevel feedback queues work similar to the RR algorithm. They discriminate favorably toward short jobs.

6.4: Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to monopolize the processor so the other process may never execute so there will be starvation.

6.10: The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

This would be very similar to the changes made in the description of the semaphore. Associated with each mutex lock would be a queue of waiting processes. When a process determines the lock is unavailable, they are placed into the queue. When a process releases the lock, it removes and awakens the first process from the list of waiting processes.

6.11: Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism – a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.

  The lock is held for a short duration - Use a spinlock. There will be much less overhead if the process waits 3-4 cycles then performing two context switches (first for removing a blocked thread and placing a new thread to run, second for reversing this process). Linux kernel tends to use spinlock for processes that should be busy waiting for a short duration of time.

- The lock is to be held for a long duration.

  The lock is to be held for a long duration - Use a mutex. Reasons given for spinlock in the previous answer do not hold here.

- A thread may be put to sleep while holding the lock.

  A thread may be put to sleep while holding the lock - if your code puts thread/process in critical region to sleep consider redesigning your critical section. This is bad code. However, it is possible that the thread/process which is in the critical section was put to sleep. This probability is greater the longer that thread/process is running. In this case, you should use a mutex. The thread which can't access the critical section will waste less cycles this way than if it was busy waiting.