CO327 : Assignment 3

E/15/202

1. Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

CPU-bound processes take a long time to execute while the I/O-bound processes execute only for a short time compared to CPU-bound processes. Scheduler's responsibility is to schedule processes according to their time to execute, so that it can prevent convoy effects.
I/O-bound programs have the property of performing only a small amount of computation before performing I/O. Such programs typically do not use up their entire CPU quantum. CPU-bound programs, on the other hand, use their entire quantum without performing any blocking I/O operations. Consequently, one could make better use of the computer's resources by giving higher priority to I/O-bound programs and allow them to execute ahead of the CPU-bound programs.

2. Explain the difference between preemptive and non-preemptive scheduling.

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready state
4. Terminates

From the above 4 conditions,
a non-preemptive scheduling only occurs under conditions 1 and 4.
a preemptive scheduling occurs under any of the conditions.

3. Discuss how the following pairs of scheduling criteria conflict in certain settings.
(a) CPU utilization and response time
(b) Average turnaround time and maximum waiting time
(c) I/O device utilization and CPU utilization

a) CPU utilization is increased if the overheads associated with context switching are minimized. The context switching overhead could be lowered by performing context switches infrequently. This could however result in increasing the response time for processes.
b) Average turnaround time is minimized by executing the shortest task first. Such a scheduling process could however starve-long running tasks and thereby increase their waiting time.
c) CPU utilization is maximized by running long-running CPU bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

4. One technique for implementing lottery scheduling works by assigning processes lottery tickets, which are used for allocating CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the CPU. The BTV operating system implements lottery scheduling by holding a lottery 50 times each second, with each lottery winner getting 20 milliseconds of CPU time (20 milliseconds × 50 = 1 second). Describe how the BTV scheduler can ensure that higher-priority threads receive more attention from the CPU than lower priority threads.

We can assign more lottery tickets to threads with higher priorities. Number of lottery tickets to be assigned will be decided by the priority level of the thread.

5. 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 the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds are 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 or I/O-bound) does the regressive round-robin scheduler favor? Explain.

This type of scheduler would favor CPU-bound processes. Because CPU-bound processes take longer time compared to I/O-bound processes. For that this scheduler provides a large time quantum which will boost up if it uses its entire time quantum without any block. And this scheduler does not penalize the I/O-bound processes. Although an I/O-bound process blocks before using its entire time quantum the priority level would remain the same.

6. Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, <u>use non-preemptive scheduling</u>, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

(a) What is the average turnaround time for these processes with the FCFS scheduling algorithm?

P1	P2	P3

0 8 12 13

Turnaround time = The interval from the time of submission of a process to the time of completion
= periods spent (waiting in the ready queue + executing on the CPU + doing I/O)

Turnaround times of P1, P2, P3

P1 = 0 + 8 = 8

P2 = (8 – 0.4) + 4 = 11.6

P3 = (12 - 1) + 1 = 12

Average turnaround time = (8 + 11.6 + 12) / 3 = 10.53

(b) What is the average turnaround time for these processes with the SJF scheduling algorithm?

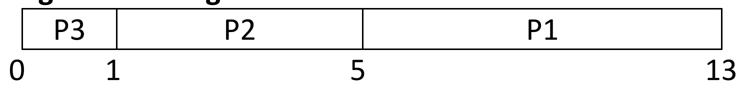
P1	P3	P2

0 8 9 13

Turnaround times of P1, P2, P3
P1 = 0 + 8 = 8
P2 = (9 – 0.4) + 4 = 12.6
P3 = (8 – 1) + 1 = 8

Average turnaround time = (8 + 12.6 + 8) / 3 = 9.53

(c) The SJF algorithm is supposed to improve performance, but notice that we chose to run process P1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is <u>left idle for the first 1 unit</u> and then SJF scheduling is used. Remember that processes <u>P1 and P2 are waiting during this idle time</u>, so their waiting time may increase. This algorithm could be called future knowledge Scheduling.



Turnaround times of P1, P2, P3
P1 = (1+5) + 8 = 14
P2 = (1 – 0.4 + 1) + 4 = 5.6
P3 = 0 + 1 = 1

Average turnaround time = (14 + 5.6 + 1) / 3 = 6.87

7. What advantage is there in having different time-quantum sizes at different levels of a multilevel queuing system?
Processes that need more frequent servicing, for instance, interactive processes such as editors, can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing, and thus making more efficient use of the computer.

8. Most scheduling algorithms maintain a run queue, which lists processes eligible to run on a processor. On multicore systems, there are two general options:
(1) each processing core has its own run queue, or
(2) a single run queue is shared by all processing cores.
What are the advantages and disadvantages of each of these approaches?
The primary advantage of each processing core having its own run queue is that there is no contention over a single run queue when the scheduler is running concurrently on 2 or more processors. When a scheduling decision must be made for a processing core, the scheduler only need to look no further than its private run queue.

A disadvantage of a single run queue is that it must be protected with locks to prevent a race condition and a processing core may be available to run a thread, yet it must first acquire the lock to retrieve the thread from the single queue. However, load balancing would likely not be an issue with a single run queue, whereas when each processing core has its own run queue, there must be some sort of load balancing between the different run queues.

9. Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α. When it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.
(a) What is the algorithm that results from $\beta > \alpha > 0$?
(b) What is the algorithm that results from $\alpha < \beta < 0$?

a) FCFS
b) LIFO

10. Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.
Both interrupt and dispatch latency needs to be minimized in order to ensure that real-time tasks receive immediate attention. Furthermore, sometimes interrupts are disabled when kernel data structures are being modified, so the interrupt does not get serviced immediately. For hard real-time systems, the time-period for which interrupts are disabled must be bounded in order to guarantee the desired quality of service.

11. Write a short note on the current status and the historical evolution of the Linux Scheduler.
Linux scheduler has been gone through some big improvements since kernel version 2.4. There were a lot of complaints about the interactivity of the scheduler in kernel 2.4. During this version, the scheduler was implemented with one running queue for all available processors. At every scheduling, this queue was locked and every task on this queue got its time slice update. This implementation caused poor performance in all aspects. The scheduler algorithm and supporting code went through a large rewrite early in the 2.5 kernel development series. The new scheduler was arisen to achieve O(1) run-time regardless number of runnable tasks in the system. To achieve this, each processor has its own running queue. This helps a lot in reducing lock contention. The priority array was introduced which used active array and expired array to keep track running tasks in the system. The O(1) running time is primarily drawn from this new data structure. The scheduler puts all expired processes into expired array. When there is no active process available in active array, it swaps active array with expired array, which makes active array becomes expired array and expired array becomes active array. There were some twists made into this scheduler to optimize further by putting expired task back to active array instead of expired array in some cases. O(1) scheduler uses a heuristic calculation to update dynamic priority of tasks based on their interactivity (I/O bound versus CPU bound) The industry was happy with this new scheduler until Con Kolivas introduced his new scheduler named Rotating Staircase Deadline (RSDL) and then later Staircase Deadline (SD). His new schedulers proved the fact that fair scheduling among processes can be achieved without any complex computation. His scheduler was designed to run in O(n) but its performance exceeded the currentO(1) scheduler.
The result achieved from SD scheduler surprised all kernel developers and designers. The fair scheduling approach in SD scheduler encouraged Igno Molnar to re-implement the new Linux scheduler named Completely Fair Scheduler (CFS). CFS scheduler was a big improvement over the existing scheduler not only in its performance and interactivity but

also in simplifying the scheduling logic and putting more modularized code into the scheduler. CFS scheduler was merged into mainline version 2.6.23. Since then, there have been some minor improvements made to CFS scheduler in some areas such as optimization, load balancing and group scheduling feature.