CSC 453 Operating Systems

Lecture 5 : Process Scheduling

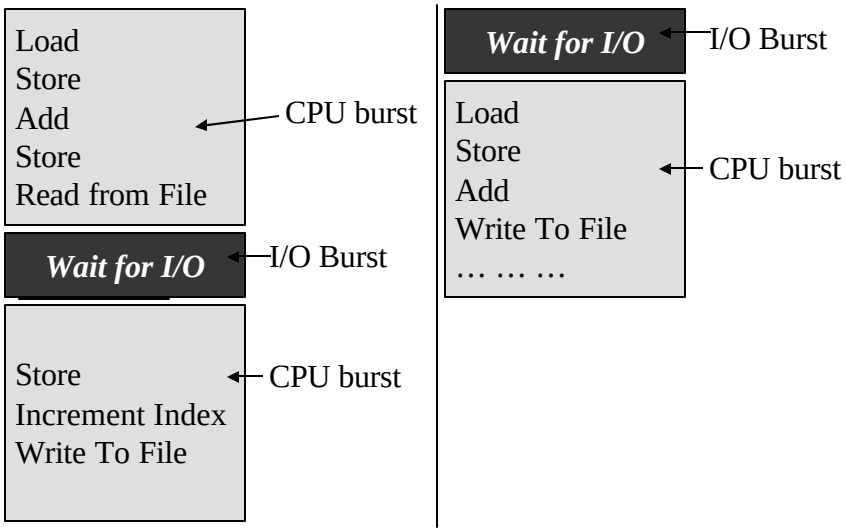
Concept of Multiprogramming

- Multiprogramming takes advantages of the fact that processes will spend a great deal of their time waiting for I/O operations to finish.
- While process #1 is waiting for I/O, the CPU will execute process #2.

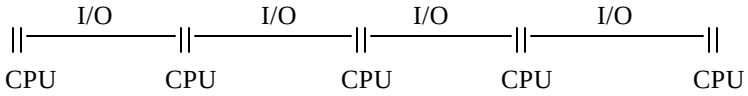
CPU-I/O Burst Cycle

- A program running on a computer has main different “bursts” of activities: bursts of CPU activity and bursts of Input/Output activity.
- Since this is cyclic in nature, it is called the ***CPU-I/O Burst Cycle***.

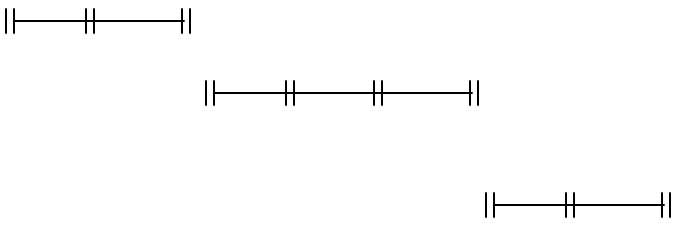
A Process Is a Series of CPU and I/O Bursts



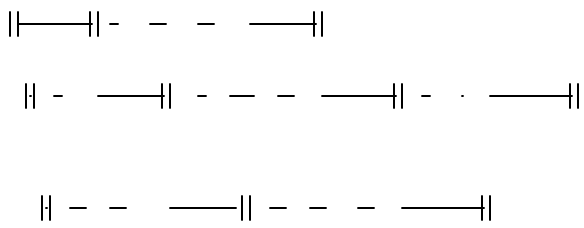
Process Spends Most Of Its Time Waiting For I/O



Three Processes Executing Without Multiprogramming



Three Processes Executing With Multiprogramming



Scheduling Criteria

- Performance criteria - What do we want from our scheduling algorithm?
- Utilization - As near 100% CPU time as possible.
- Throughput - Highest possible number of finished processes per unit time.
- Turnaround time - As low as possible for jobs from start to finish.
- Response time - For interactive systems, this is more important than turnaround time.
- Waiting Time - minimize time in ready queue and device queues.

Preemptive vs. Nonpreemptive Scheduling

Preemptive scheduling - processes using the CPU can be removed by the system.

Nonpreemptive scheduling - processes using the CPU cannot be removed by the CPU.

Starvation - A situation that arises when a process never gets to the CPU (or to perform an I/O operation, etc.)

Scheduling Algorithms

- Scheduling Algorithms include:
  - *First-Come-First-Served*
  - *Shortest Job First*
  - *Round Robin*
  - *Priority*
  - *Guaranteed*
  - *Lottery*
  - *Real-Time*

First-Come-First-Served

- First-Come-First-Served Algorithm is the simplest CPU scheduling.
- Whichever process requests the CPU first gets it first.
- It is implemented using a standard FIFO single queue.
- Waiting time can be long and it depends heavily on the order in which processes request CPU time:

An Example of First-Come-First-Served

Imagine three processes with the following burst times:

Process	Burst time (in msec)
P_1	24
P_2	3
P_3	4

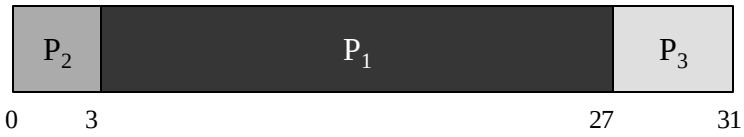
Scenario #1 For FCFS Scheduling

Processing time = (24 + 28 + 31) / 3 = 83/3 = 27.7 msec

Scenario #2 For FCFS Scheduling

Waiting time = (3 + 7 + 31) / 3 = 41/3 = 13.7 msec

Scenario #3 For FCFS Scheduling



Waiting time = (3 + 27 + 31) / 3 = 61/3 = 20.3 msec

Shortest Job First

- Most appropriately called ***Shortest Next CPU Burst First*** because it bases the order upon an approximation of how long what the next CPU burst will be.
- This can be proven to be the optimal scheduling algorithm with the shortest average processing (and waiting) time.
- The SJF algorithm can be preemptive or non-preemptive, with the preemptive SJF algorithm more properly being called ***shortest- remaining-time-first*** scheduling.

Gantt Chart for Shortest Job First Example

Waiting time = (3 + 7 + 31) / 3 = 41/3 = 13.7 msec

CPU Burst Length

- The real difficulty is that we are trying to predict how long the next CPU burst will be and this cannot be done with any real accuracy for short-term CPU scheduling.
- The next burst is usually predicted using the exponential average of previously CPU bursts:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau$$

t_n = length of the nth CPU burst.

τ_n = the past history of the CPU bursts.

$0 <= \alpha <= 1$

Predicting The Next CPU Burst Length

- We can substitute for τ_n and expand get the exponential average:

  $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha\, t_{n-1} + \ldots$

  $+ (1-\alpha)^j\,\alpha\, t_{n-j} + \ldots (1-\alpha)^{n+1}\,\alpha\, \tau_0.$

- More recent terms all have more weight than earlier term in the calculation.

Priority Scheduling

- Priority scheduling involves a priority assigned to each process, which is scheduled in accordance with its priority.
- Processes with equal priority are scheduled on a FCFS basis.
- A *SJF* algorithm is a special case of a priority scheduling algorithm with *priority(p)* being proportional to 1/p.

Priority Levels

- There is no general agreement on whether 0 is the highest or lowest priority (priority numbers are assumed to be positive).
  - UNIX uses 0 as the highest priority
  - IBM's MVS uses it as the default (lowest) priority.

Setting Priorities

- Priorities can be set:
  - ***internally*** (by some measurable quantity or quantities such as time limits, memory requirements, number of open files, I/O burst-to-CPU burst ratio, etc.)
  - or
  - ***externally*** (by system policy, such as process importance, type or availability of funds, sponsoring department, etc.)

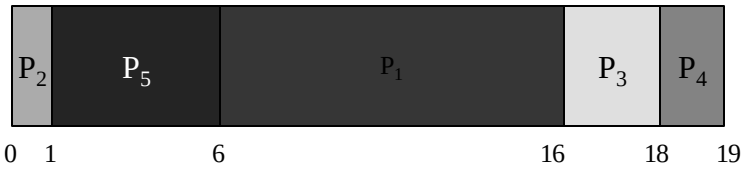
Starvation

- ***Starvation*** is a major problem of priority scheduling algorithms.
- On a busy system, a low-priority process may sit for extremely long periods of time.
- A solution to the problem is *aging*, where we increment the priority (make it a higher priority) for every 1-15 minutes of waiting.

Scenario For Priority Scheduling

Process	**Burst Time**	**Priority**
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Gantt Chart For Priority Scheduling Scenario



Processing time = (1 + 6 + 16 + 18 + 19) / 5 = 8 msec

Round Robin Scheduling

- Round-robin scheduling is designed for time-sharing system.
- It is similar to the FCFS scheduling, but preemptive is added to switch between processes.
- A time quantum is typically 10 to 100 milliseconds.
- The ready queue is implemented in FIFO manner..

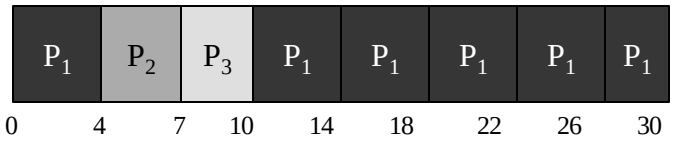
Round-Robin Scheduling and Preemption

- If a process needs less than a time quantum, it releases the CPU voluntarily.
- If a process needs more than a time quantum, it is preempted from the CPU and placed at the back of the ready queue.

Scenario For Round-Robin Scheduling

Process	**Burst Time**
P_1	24
P_2	3
P_3	3

Gantt Chart For Round Robin Scheduling Scenario



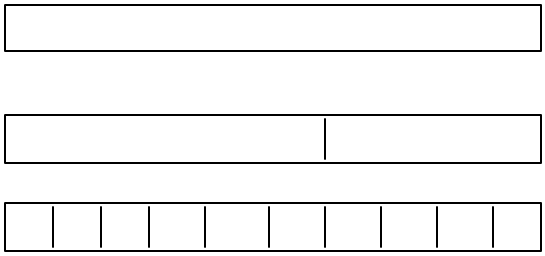
Processing time = (30 + 7 + 10) / 3 = 15.7 msec

Time Quanta

- The performance of round robin scheduling is extremely dependent upon the size of the time quantum in use.
- If the time quantum is large (approaching infinity), it approaches a FCFS algorithm.
- If the time is quantum is small, it appears (in theory at least) that each user has his/her own processor.

Time Quanta and Context Switches

- We need the time quantum to be large with respect to the context-switch time (time it takes to switch processes) because each of these can effectively slow the processor.



quantum	context switches
10	0
6	1
1	9

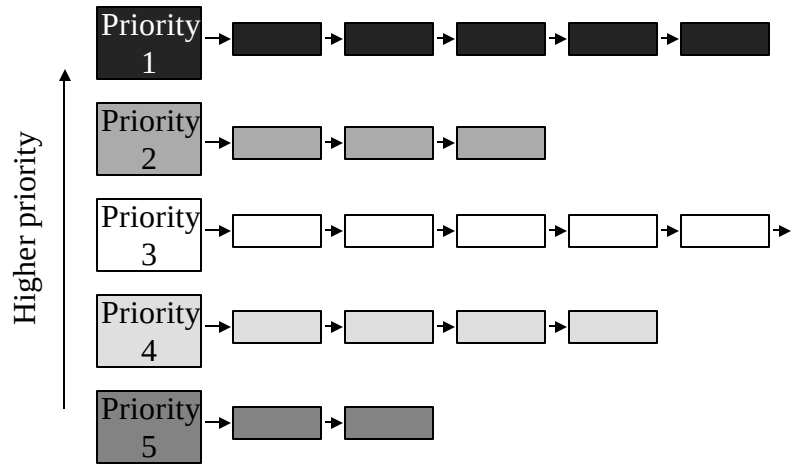
Guaranteed Scheduling

- If fairness is an important concern, and there are n users on a uniprocessor system, each user should be able to get 1/n of the system's time.
- To make good on this promise to provide each user with 1/n of the CPU time, we keep track of how much CPU each user has gotten over a time frame and calculate the ratio of actual CPU time to entitled CPU time.
- A ratio of 0.5 means that a process should have gotten half as much CPU time; a ratio of 2.0 means that the process has gotten twice as much as it should have gotten.
- Such a scheduling algorithm runs the process with the poorest ratio until it catches up to and passes its nearest competitor.

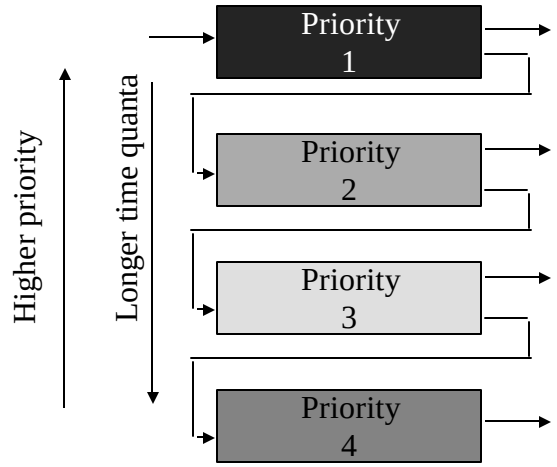
Lottery Scheduling

- Every process is given, in effect, tickets for a lottery, where the prize is the next time slice (or some other system resource).
- Applied to CPU scheduling, there may be 50 lottery drawings each second, with each winner getting 20 msec of CPU time.
- Important processes can get extra CPU time by being given extra “tickets” for the drawings.
- Cooperating processes can exchange tickets if they wish..

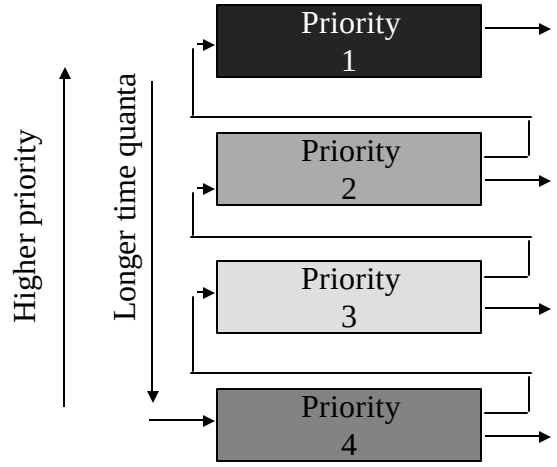
Multilevel Queue Scheduling



Multilevel Feedback Queues: Shifting to Lower Priorities



Multilevel Feedback Queues: Shifting to Higher Priorities



Multiple Processor Scheduling

- Process scheduling on a multiprocessor system is more complex.
- It is easier to schedule ***homogeneous*** multiprocessor systems than ***heterogeneous systems.***
- Identical processors can do ***load sharing*** with ***separate ready queues*** or a ***common ready queue***.

Symmetric vs. Asymmetric Multiprocessing

- In symmetric multiprocessing, all the processors are considered peers and any one of them can handle any sort of task.
- In asymmetric multiprocessing, there is a hierarchy among the processors and one of them may handle the task of scheduling processes for the others.

What is Real-time Scheduling?

- A real-time system is one in which time plays a crucial role.
- An example is a CD player which must read and then translate the bits into music within a tight time frame.
- Real-time systems can be ***hard real time*** (where absolute deadlines must always be met) or ***soft real time*** (where an occasional deadline can be missed).

Real-time Scheduling

- Real-time behavior is achieved by by dividing the program into a number of processes, each of which have known behavior.
- Real-time systems react to events which can be ***periodic*** (happening at regular intervals) or ***aperiodic*** (not happening at regular intervals).
- If there are m periodic events and event i occurs with a period P_i and requires C_i seconds of CPU time, then the load can only be handled if

$$\sum_{i} C_i/P_i <= 1$$

  Such a system is ***schedulable***.

Algorithm Evaluation

- There are several ways in which we can evaluate the scheduling algorithms:
  - Deterministic Modeling
  - ***Queueing Models***
  - Simulation
- Our goal is to see if they help us meet the performance criteria that were discussed earlier.

Deterministic Modeling

- We will assume a predetermined set of data.
- Given that data, we will determine how the scheduling algorithm will perform.
- Deterministic Modeling is easy to understand and implement but it only tells us about the data sets that we use.

Simulations

- We use random numbers to give us a large set of data that should be representative of real-life processing scenarios.
- The distributions can be defined either empirically or mathematically (e.g., a Poisson distribution).
- Such simulations can be expensive and more informative.