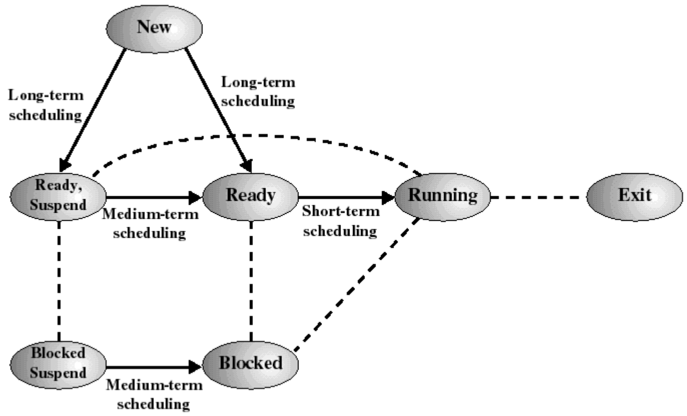
Uniprocessor Scheduling

Chapter 9

CPU Scheduling

- **We concentrate on the problem of scheduling the usage of a single processor among all the existing processes in the system**
- **The goal is to achieve**
  - High processor utilization
  - High throughput
    - number of processes completed per unit time
  - Low response time
    - time elapse from the submission of a request to the beginning of the response

Classification of Scheduling Activity



- **Long-term: which process to admit**
- **Medium-term: which process to swap in or out**
- **Short-term: which ready process to execute next**

Long-Term Scheduling

- **Determines which programs are admitted to the system for processing**
- **Controls the degree of multiprogramming**
- **If more processes are admitted**
  - less likely that all processes will be blocked
    - better CPU usage
  - each process has less fraction of the CPU
- **The long term scheduler may attempt to keep a mix of processor-bound and I/O-bound processes**

Medium-Term Scheduling

- **Swapping decisions based on the need to manage multiprogramming**
- **Done by memory management software and discussed intensively in chapter 8**
  - see resident set allocation and load control

Short-Term Scheduling

- **Determines which process is going to execute next (also called CPU scheduling)**
- **Is the subject of this chapter**
- **The short term scheduler is known as the dispatcher**
- **Is invoked on a event that may lead to choose another process for execution:**
  - clock interrupts
  - I/O interrupts
  - operating system calls and traps
  - signals

Short-Tem Scheduling Criteria

- **User-oriented**
  - Response Time: Elapsed time from the submission of a request to the beginning of response
  - Turnaround Time: Elapsed time from the submission of a process to its completion
- **System-oriented**
  - processor utilization
  - fairness
  - throughput: number of process completed per unit time

Priorities

- **Implemented by having multiple ready queues to represent each level of priority**
- **Scheduler will always choose a process of higher priority over one of lower priority**
- **Lower-priority may suffer starvation**
- **Then allow a process to change its priority based on its age or execution history**
- **Our first scheduling algorithms will not make use of priorities**
- **We will then present other algorithms that use dynamic priority mechanisms**

Characterization of Scheduling Policies

- **The selection function: determines which process in the ready queue is selected next for execution**
- **The decision mode: specifies the instants in time at which the selection function is exercised**
  - Nonpreemptive
    - Once a process is in the running state, it will continue until it terminates or blocks itself for I/O
  - Preemptive
    - Currently running process may be interrupted and moved to the Ready state by the OS
    - Allows for better service since any one process cannot monopolize the processor for very long

The CPU-I/O Cycle

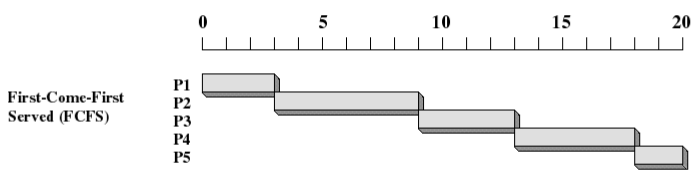
- **We observe that processes require alternate use of processor and I/O in a repetitive fashion**
- **Each cycle consist of a (usually relatively short)CPU burst followed by a (usually longer) I/O burst**
- **A process terminates on a CPU burst**
- **CPU-bound processes have longer CPU bursts than I/O-bound processes**

Our running example to discuss various scheduling policies

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Service time = total processor time needed in one (CPU-I/O) cycle
Jobs with long service time are CPU-bound jobs
and are referred to as "long jobs"

First Come First Served (FCFS)

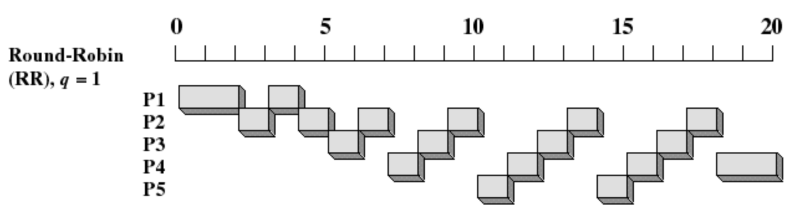


- **Selection function: the process that has been waiting the longest in the ready queue (hence, FCFS)**
- **Decision mode: nonpreemptive**
  - a process run until it blocks itself

FCFS drawbacks

- **A process that does not perform any I/O will monopolize the processor**
- **Favors CPU-bound processes**
  - I/O-bound processes have to wait until CPU-bound process completes
  - They may have to wait even when their I/O are completed (poor device utilization)
  - we could have kept the I/O devices busy by giving a bit more priority to I/O bound processes

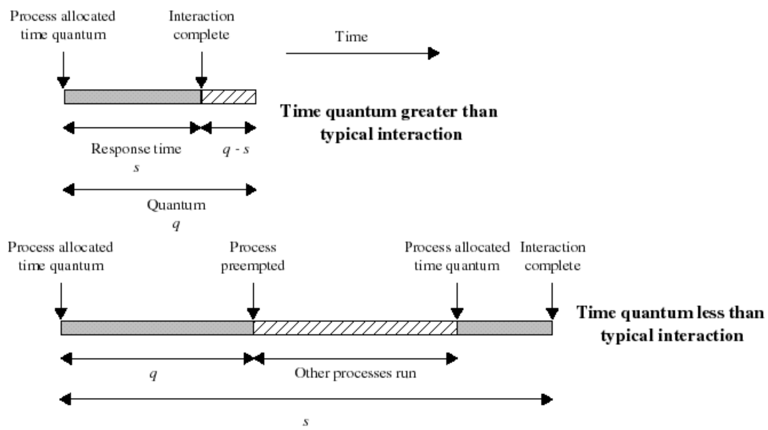
Round-Robin



- **Selection function: same as FCFS**
- **Decision mode: preemptive**
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - then a clock interrupt occurs and the running process is put on the ready queue

Time Quantum for Round Robin

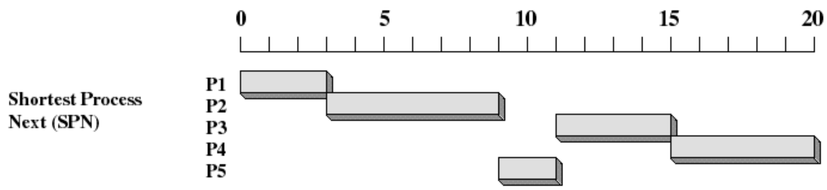
- **must be substantially larger than the time required to handle the clock interrupt and dispatching**
- **should be larger then the typical interaction (but not much more to avoid penalizing I/O bound processes)**



Round Robin: critique

- **Still favors CPU-bound processes**
  - A I/O bound process uses the CPU for a time less than the time quantum and then is blocked waiting for I/O
  - A CPU-bound process run for all its time slice and is put back into the ready queue (thus getting in front of blocked processes)
- **A solution: virtual round robin**
  - When a I/O has completed, the blocked process is moved to an auxiliary queue which gets preference over the main ready queue
  - A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running since it was selected from the ready queue

Shortest Process Next (SPN)



- Selection function: the process with the shortest expected CPU burst time
- Decision mode: nonpreemptive
- I/O bound processes will be picked first
- We need to estimate the required processing time (CPU burst time) for each process

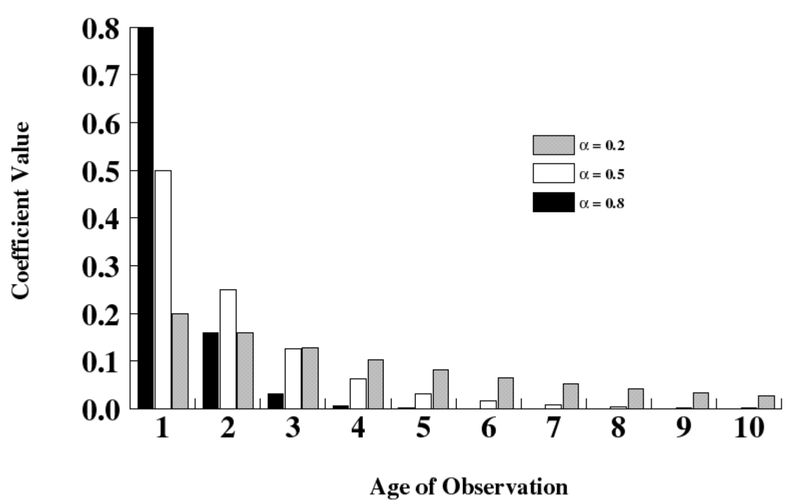
Estimating the required CPU burst

- Let T[i] be the execution time for the ith instance of this process: the actual duration of the ith CPU burst of this process
- Let S[i] be the predicted value for the ith CPU burst of this process. The simplest choice is:
  - $S[n+1] = (1/n) \Sigma_{\{i=1 \text{ to } n\}} T[i]$
- To avoid recalculating the entire sum we can rewrite this as:
  - $S[n+1] = (1/n) T[n] + ((n-1)/n) S[n]$
- But this convex combination gives equal weight to each instance

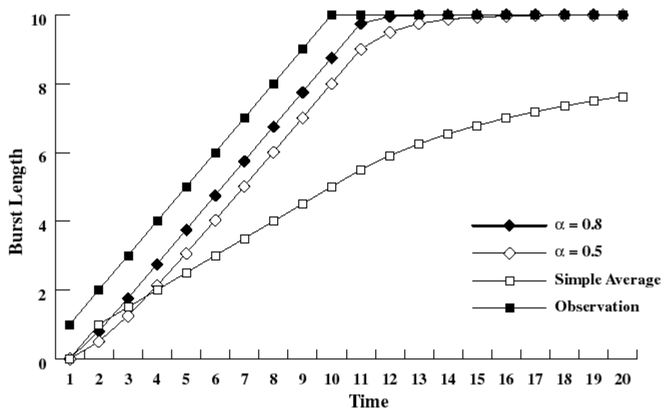
Estimating the required CPU burst

- **But recent instances are more likely to reflect future behavior**
- **A common technique for that is to use exponential averaging**
  - S[n+1] = α T[n] + (1-α) S[n] ; 0 < α < 1
  - more weight is put on recent instances whenever α > 1/n
- **By expanding this eqn, we see that weights of past instances are decreasing exponentially**
  - S[n+1] = αT[n] + (1-α)αT[n-1] + ... (1-α)^{i}αT[n-i] + ... + (1-α)^{n}S[1]
  - predicted value of 1st instance S[1] is not calculated; usually set to 0 to give priority to to new

Exponentially Decreasing Coefficients



Exponentially Decreasing Coefficients



- Here S[1] = 0 to give high priority to new processes
- Exponential averaging tracks changes in process behavior much faster than simple averaging

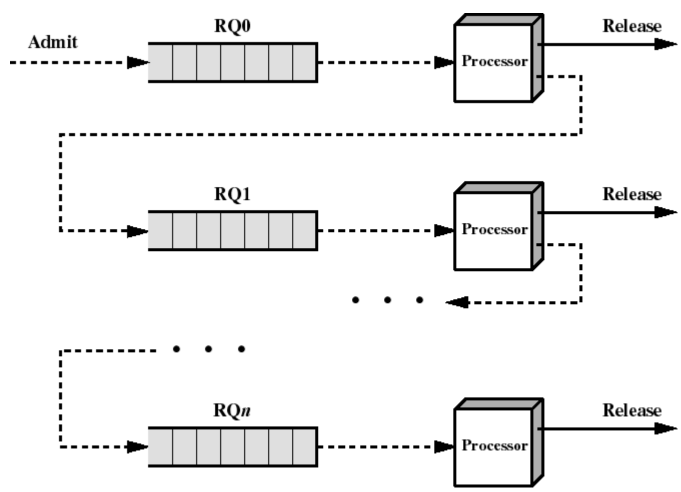
Shortest Process Next: critique

- **Possibility of starvation for longer processes as long as there is a steady supply of shorter processes**
- **Lack of preemption is not suited in a time sharing environment**
  - CPU bound process gets lower priority (as it should) but a process doing no I/O could still monopolize the CPU if he is the first one to enter the system
- **SPN implicitly incorporates priorities: shortest jobs are given preferences**
- **The next (preemptive) algorithm penalizes directly longer jobs**

Multilevel Feedback Scheduling

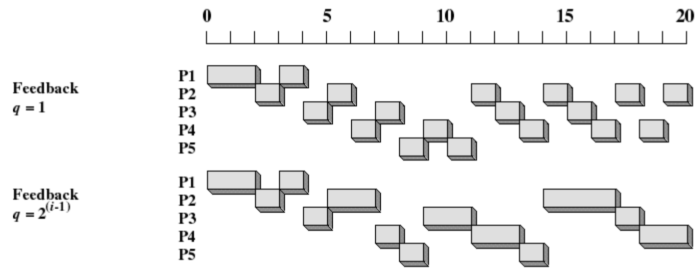
- **Preemptive scheduling with dynamic priorities**
- **Several ready to execute queues with decreasing priorities:**
  - P(RQ0) > P(RQ1) > ... > P(RQn)
- **New process are placed in RQ0**
- **When they reach the time quantum, they are placed in RQ1. If they reach it again, they are place in RQ2... until they reach RQn**
- **I/O-bound processes will stay in higher priority queues. CPU-bound jobs will drift downward.**
- **Dispatcher chooses a process for execution in RQi only if RQi-1 to RQ0 are empty**
- **Hence long jobs may starve**

Multiple Feedback Queues



- **FCFS is used in each queue except for lowest priority queue where Round Robin is used**

Time Quantum for feedback Scheduling



- **With a fixed quantum time, the turnaround time of longer processes can stretch out alarmingly**
- **To compensate we can increase the time quantum according to the depth of the queue**
  - Ex: time quantum of RQi = 2^{i-1}
- **Longer processes may still suffer starvation. Possible fix: promote a process to higher priority after some time**

Algorithm Comparison

- **Which one is best?**
- **The answer depends on:**
  - on the system workload (extremely variable)
  - hardware support for the dispatcher
  - relative weighting of performance criteria (response time, CPU utilization, throughput...)
  - The evaluation method used (each has its limitations...)
- **Hence the answer depends on too many factors to give any...**

Fair Share Scheduling

- **In a multiuser system, each user can own several processes**
- **Users belong to groups and each group should have its fair share of the CPU**
- **This is the philosophy of fair share scheduling**
- **Ex: if there are 4 equally important departments (groups) and one department has more processes than the others, degradation of response time should be more pronounced for that department**

The Fair Share Scheduler (FSS)

- **Has been implemented on some Unix OS**
- **Processes are divided into groups**
- **Group k has a fraction Wk of the CPU**
- **The priority Pj[i] of process j (belonging to group k) at time interval i is given by:**
  - $P_j[i] = B_j + (1/2)\, CPU_j[i-1] + GCPU_k[i-1]/(4W_k)$
    - A high value means a low priority
    - Process with highest priority is executed next
    - B_j = base priority of process j
    - $CPU_j[i]$ = Exponentially weighted average of processor usage by process j in time interval i
    - $GCPU_k[i]$ = Exponentially weighted average processor usage by group k in time interval i

The Fair Share Scheduler (FSS)

- **The exponentially weighted averages use $\alpha = 1/2$:**
  - CPUj[i] = (1/2) Uj[i-1] + (1/2) CPUj[i-1]
  - GCPUk[i] = (1/2) GUk[i-1] + (1/2) GCPUk[i-1]
  - where
    - Uj[i] = processor usage by process j in interval i
    - GUk[i] = processor usage by group k in interval i
- **Recall that**
  - Pj[i] = Bj + (1/2) CPUj[i-1] + GCPUk[i-1]/(4Wk)
- **The priority decreases as the process and group use the processor**
- **With more weight Wk, group usage decreases less the priority**