## Aim of Scheduling

Assign processes to be executed by the processor(s)

Response time

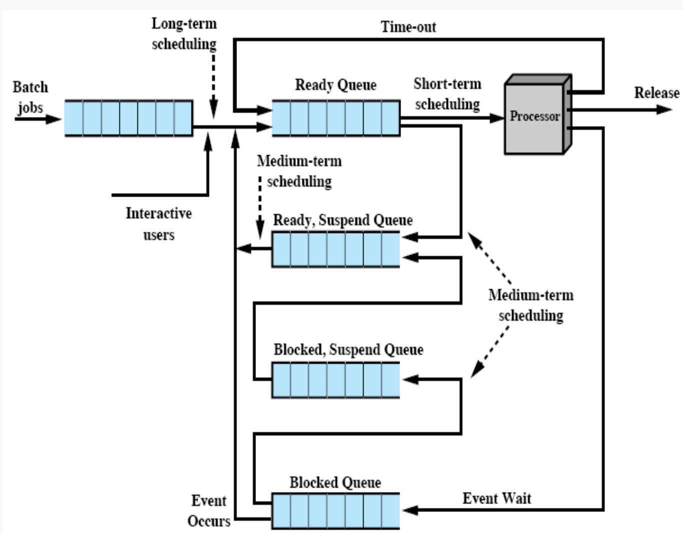
Throughput

Processor efficiency





## General Queuing Diagram



# Priorities

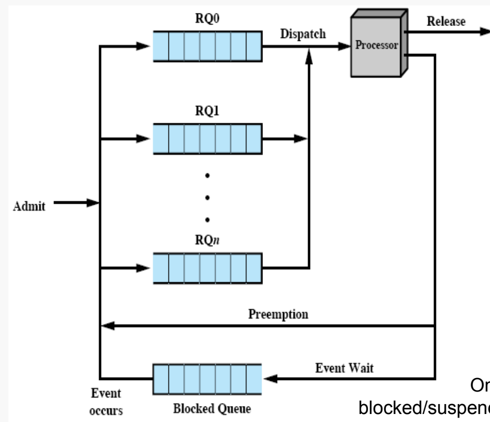


Scheduler will always choose a process of higher priority over one of lower priority

Have multiple ready queues to represent each level of priority

Lower-priority may suffer starvation

Allow a process to change its priority based on its age or execution history




Omits blocked/suspended queues for simplicity.

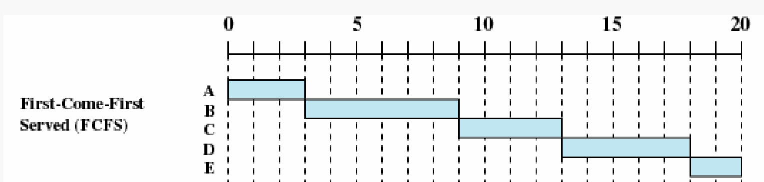



# First-Come-First-Served (FCFS)



Each process joins the Ready queue

When the current process ceases to execute, the oldest process in the Ready queue is selected




A short process may have to wait a very long time before it can execute

Favors CPU-bound processes

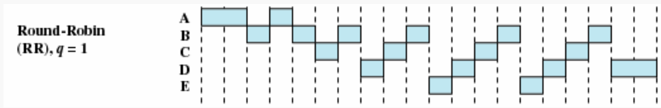
I/O processes have to wait until CPU-bound process completes

## Round-Robin



Uses preemption based on a clock

An amount of time is determined that allows each process to use the processor for that length of time




Clock interrupt is generated at periodic intervals

When an interrupt occurs, the currently running process is placed in the read queue

Next ready job is selected

Known as time slicing

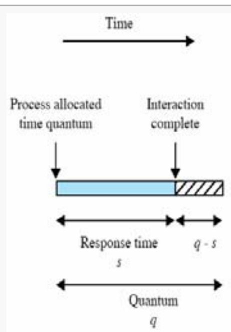
Principal design issue is the size of the quantum

Favors CPU-bound processes over I/O-bound processes



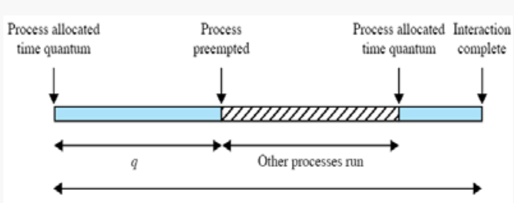
## Effect of Quantum Size






Too-short quantum leads to excessive percentage of time being spend handling process switching.

Quantum slightly larger than the time for a typical interaction or process function.




Quantum shorter than that means that most processes will take two or more quanta to finish, hurting interactive response time.

# Virtual Round-Robin Queuing Diagram

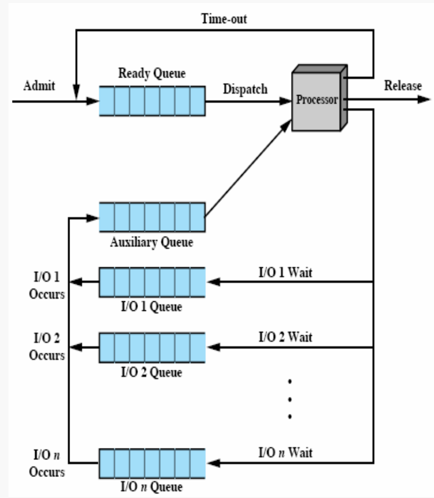


VRR attempts to improve relative handling of I/O- and CPU-bound processes.

When pending I/O operation is completed, the waiting process is moved to an auxiliary queue instead of the Ready queue.

On a process switch, the dispatcher will favor processes in the auxiliary queue over those in the Ready queue.

But, a process selected from the auxiliary queue will not receive a full quantum. Instead, it gets a quantum minus the time it has spent running since it was last selected from the main Ready queue.







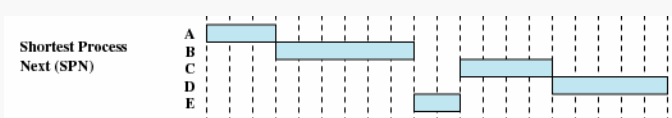
# Shortest Process Next



Batch policy

Process with shortest expected service time is selected next

Short process jumps ahead of longer processes




Predictability of longer processes is reduced

If estimated time for process not correct, the operating system may abort it

- history of batch jobs used to estimate their service times
- for interactive processes…

Possibility of starvation for longer processes

## Predicting Service Time

The service time for an interactive process can be predicted statistically.

The *burst time* for an interactive process is the processor execution time the process uses during one period in the Running state. A simple predictor for the next burst time would be given by:

$$S_{n+1} = \frac{1}{n}\sum_{k=1}^{n} T_k$$

However, we can rewrite this formula to avoid recalculating the sum:

$$S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n$$

Obviously, this is only an estimate, and assumes a very simple relationship between past and future behavior. In particular, it gives the same weight to recent and far-past burst times.

Intuitively we might expect that recent values would be better predictors, leavened with older values.






## Exponential Averaging

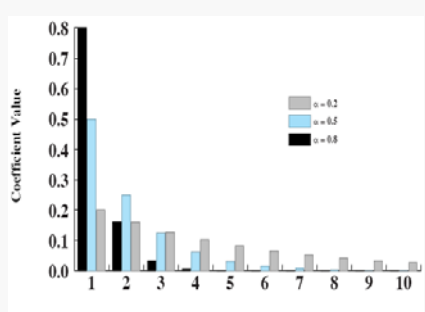
A (perhaps) better predictor can be obtained by using a weighted average:

$$S_{n+1} = \alpha T_n + (1-\alpha)S_n$$

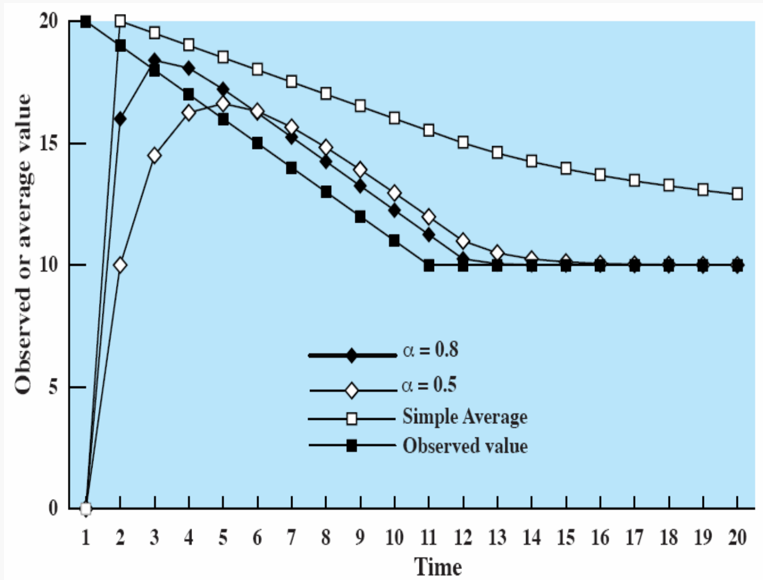
where $\alpha$ is a constant weighting factor such that $0 < \alpha < 1$. If we expand this formula recursively, we obtain

$$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + (1-\alpha)^2 \alpha T_{n-2} + \cdots + (1-\alpha)^n T_1$$

The coefficients of the successive terms are decreasing, and so the older terms are given less weight:



## Example





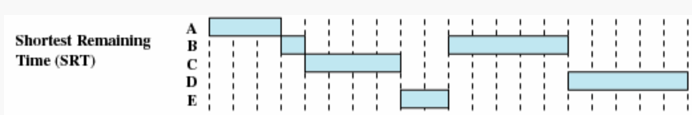




## Shortest Remaining Time

Preemptive (multiprogramming) version of shortest process next policy

As with SPN, must estimate processing time



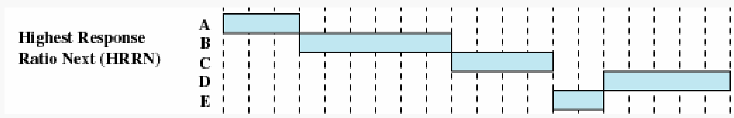
# Highest Response Ratio Next (HRRN)

Define the response ratio for a process by

$$R = \frac{w+s}{s}$$

where $w$ is the time the process has spent waiting for the processor and $s$ is the expected service time for the process.

Choose next process with the greatest ratio









# Scheduling via Feedback

If we can't know the time remaining, penalize jobs that have been running longer:

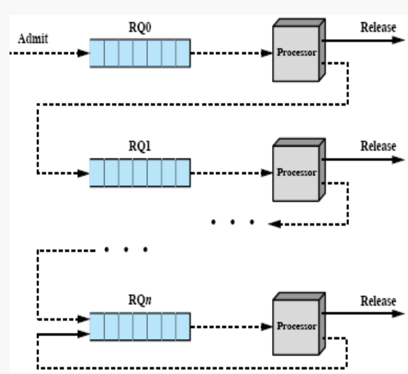
- preemptive scheduling via quantum timer
- dynamic priorities
- process starts in level 0 Ready queue
- after each preemption, process is demoted to next level queue of Ready state
- process just cycles once it reaches the lowest level

Rewards shorter processes because they only compete with other processes that haven't been running very long.

Problem: long processes may starve.

One solution: give processes in lower levels longer quanta

Another: gradually promote processes upward after a certain interval in lowest level



## Summary of Scheduling Policies

| | Selection Function | Decision Mode | Throughput | Response Time | Overhead | Effect on Processes | Starvation |
|---|---|---|---|---|---|---|---|
| **FCFS** | max[$w$] | Nonpreemptive | Not emphasized | May be high, especially if there is a large variance in process execution times | Minimum | Penalizes short processes; penalizes I/O bound processes | No |
| **Round Robin** | constant | Preemptive (at time quantum) | May be low if quantum is too small | Provides good response time for short processes | Minimum | Fair treatment | No |
| **SPN** | min[$s$] | Nonpreemptive | High | Provides good response time for short processes | Can be high | Penalizes long processes | Possible |
| **SRT** | min[$s - e$] | Preemptive (at arrival) | High | Provides good response time | Can be high | Penalizes long processes | Possible |
| **HRRN** | $\max\left(\frac{w+s}{s}\right)$ | Nonpreemptive | High | Provides good response time | Can be high | Good balance | No |
| **Feedback** | (see text) | Preemptive (at time quantum) | Not emphasized | Not emphasized | Can be high | May favor I/O bound processes | Possible |

$w$ = time spent waiting
$e$ = time spent in execution so far
$s$ = total service time required by the process, including $e$





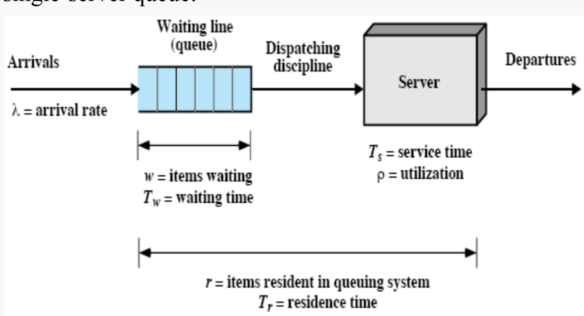
## Simple Queuing Analysis

How can we make projections regarding the performance of a system?

If we have performance information relating to existing load, and some basis for estimating future load, we have several options.

We will consider how to develop an analytical model based on queuing theory.

First consider the single-server queue:



## Queuing Theory Notation

$\lambda$ *arrival rate*; mean number of arrivals per second

$T_s$ *mean service time*; counts time served but excludes wait time

$\rho$ *utilization*; fraction of time server is busy

$w$ mean number of items waiting to be served

$T_w$ *mean waiting time*; excludes items with wait time 0 and items that must wait

$r$ mean number of items resident in system (waiting and being served)

$T_r$ *mean residence time*; time an items spends in the system waiting and being served






## Assumptions and Observations

We assume:

- queue has infinite capacity
- no item is ever lost from the system
- all items eventually finish
- infinite population of items that will eventually arrive
- when the server becomes free, dispatcher uses some well-defined policy to select next item for service

Note:

- as the arrival rate increases, the utilization also increases
- as utilization approaches 1, the queue becomes congested with waiting items
- the arrival rate equals the departure rate over the long term
- the maximum arrival rate that can be handled by the system is $\lambda_{MAX} = 1/T_s$

## Case Study


Assume a two-priority system in which:

- priority 1 items are processed before priority 2 items
- FIFO dispatching is used for items of equal priority
- no item is interrupted while being served
- items arrive according to a Poisson distribution

Then we have a few basic formulas:

$$\lambda = \lambda_1 + \lambda_2$$

$$\rho_1 = \lambda_1 T_{s1}; \rho_2 = \lambda_2 T_{s2}$$

$$\rho = \rho_1 + \rho_2$$

$$T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$$

$$T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$$

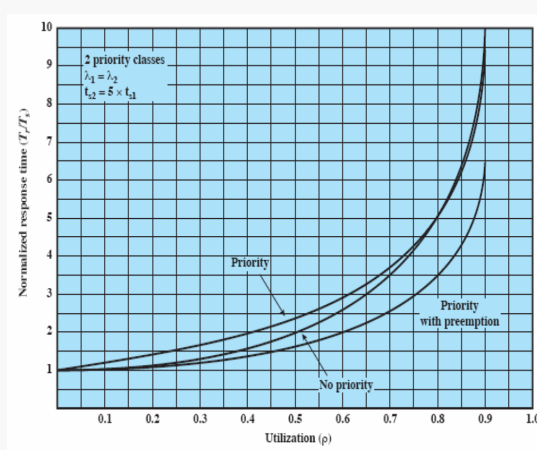



## Normalized Response Times


Suppose we have a system that uses a two-priority scheme where priorities are assigned based on service time.

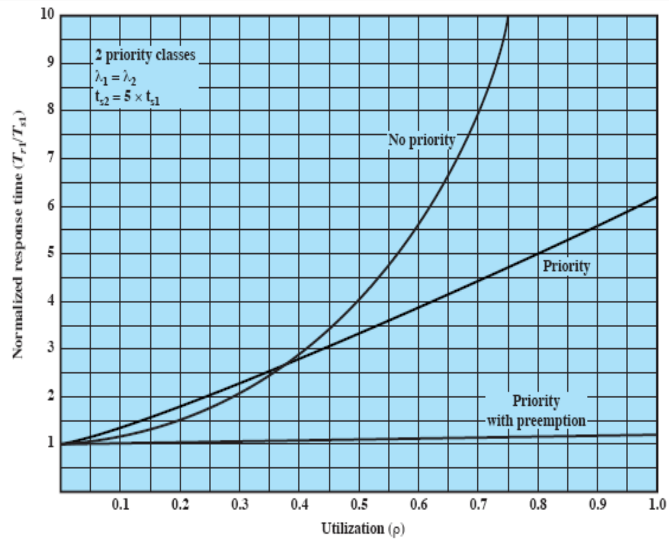
With equal arrival rates for short and long processes, and long processes taking 5 times longer than short ones, we have (theoretically):

Overall, using priorities this way makes the most improvement in the normalized response times if scheduling also uses preemption, but there's not much difference.



## Effect on Shorter Processes

The effect on shorter processes is, however, fairly dramatic, especially if preemptive scheduling is used:



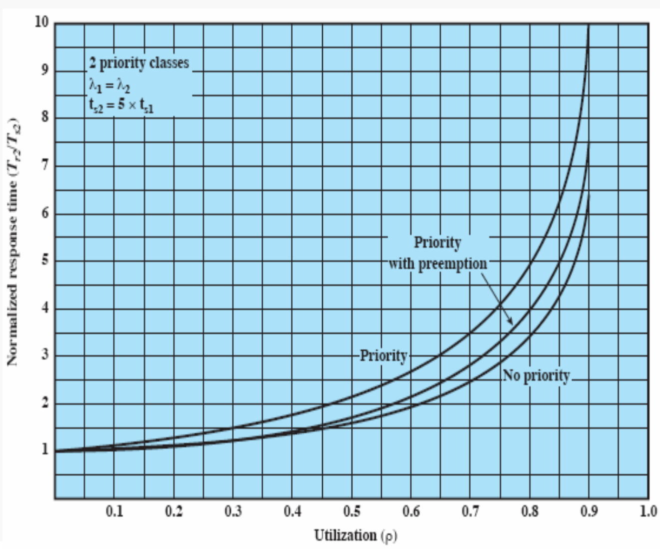






## Effect on Longer Processes

The effect on longer processes is, as you should expect, slight, but their performance is degraded:

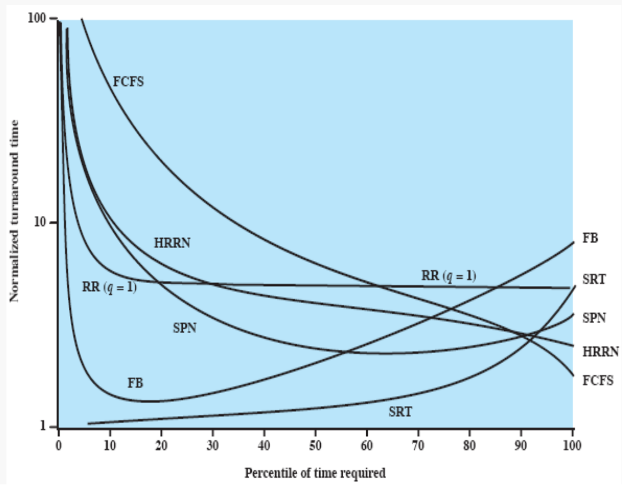


## Simulated Turnaround Time

Theoretical analysis can be supplemented by discrete-event simulations.

Neither is entirely trustworthy, but the combination may justify confidence…

Simulating 50,000 processes with an arrival rate $\lambda = 0.8$ and an average service time $T_S = 1$, and a few other assumptions described in Stallings, we might obtain:

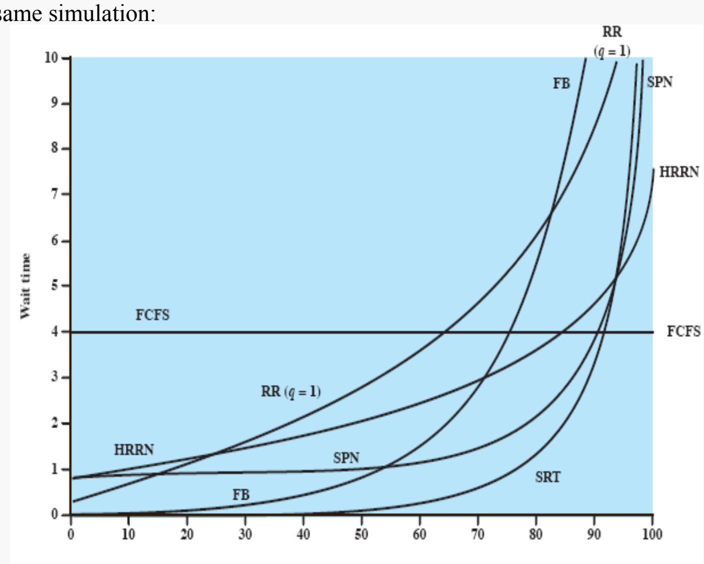




## Simulated Waiting Time

From the same simulation:



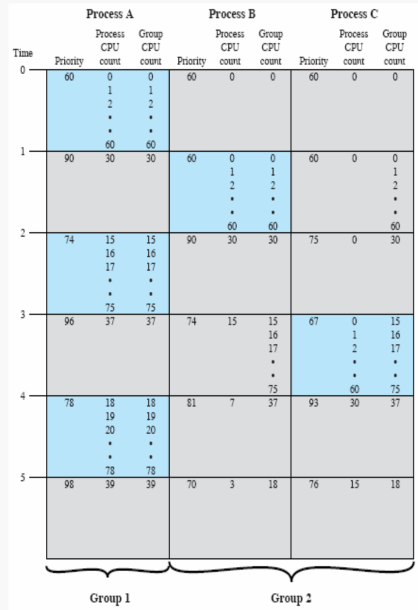
## Fair-Share Scheduling



User's application runs as a collection of processes (threads)

User is concerned about the performance of the application

Need to make scheduling decisions based on process sets






## Traditional UNIX Scheduling



Multilevel feedback using round robin within each of the priority queues

If a running process does not block or complete within 1 second, it is preempted

Priorities are recomputed once per second

Base priority divides all processes into fixed bands of priority levels

# Bands

Decreasing order of priority

- Swapper
- Block I/O device control
- File manipulation
- Character I/O device control
- User processes

The use of execution history favors I/O-bound processes and penalizes CPU-bound processes, which should improve overall efficiency.

Coupled with round-robin preemption, this achieves good general performance.

| Time | Process A | | Process B | | Process C | |
|---|---|---|---|---|---|---|
| | Priority | CPU Count | Priority | CPU Count | Priority | CPU Count |
| 0 | 60 | 0 1 2 • • 60 | 60 | 0 | 60 | 0 |
| 1 | 75 | 30 | 60 | 0 1 2 • • 60 | 60 | 0 |
| 2 | 67 | 15 | 75 | 30 | 60 | 0 1 2 • • 60 |
| 3 | 63 | 7 8 9 • • 67 | 67 | 15 | 75 | 30 |
| 4 | 76 | 33 | 63 | 7 8 9 • • 67 | 67 | 15 |
| 5 | 68 | 16 | 76 | 33 | 63 | 7 |