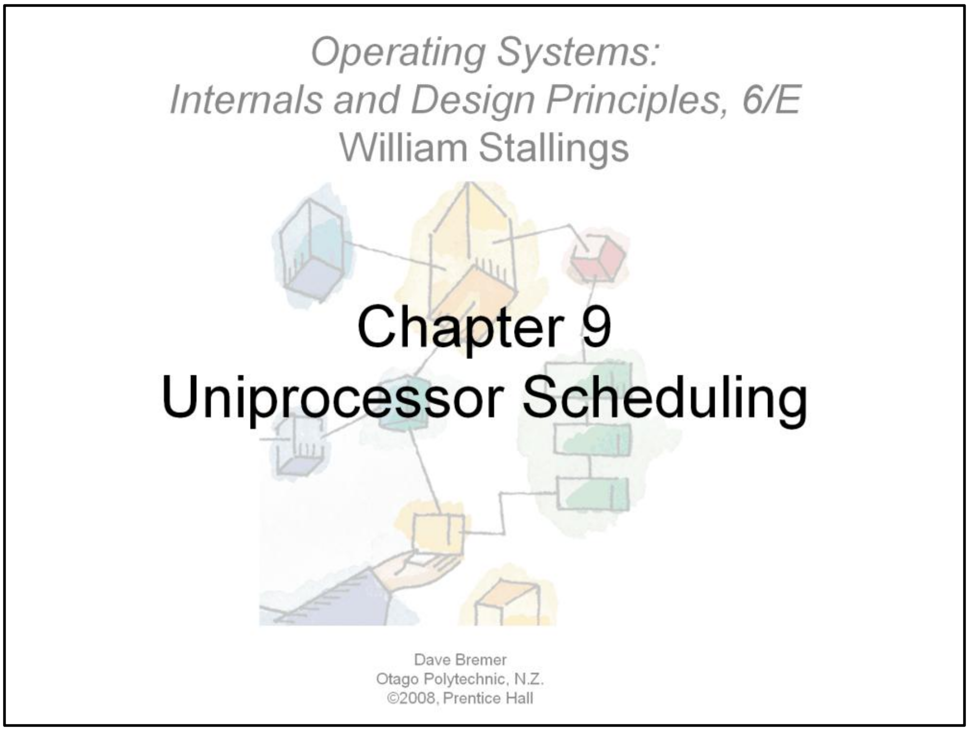
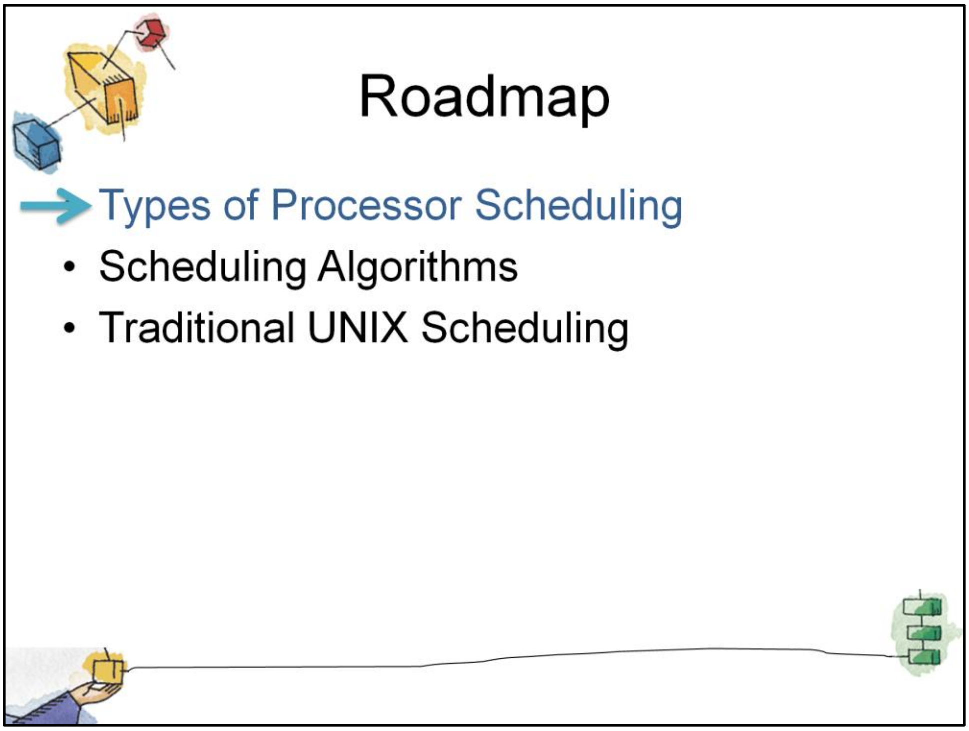


Operating Systems:
Internals and Design Principles, 6/E
William Stallings

Chapter 9
Uniprocessor Scheduling

Dave Bremer
Otago Polytechnic, N.Z.

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.



Roadmap

→ Types of Processor Scheduling
- Scheduling Algorithms
- Traditional UNIX Scheduling

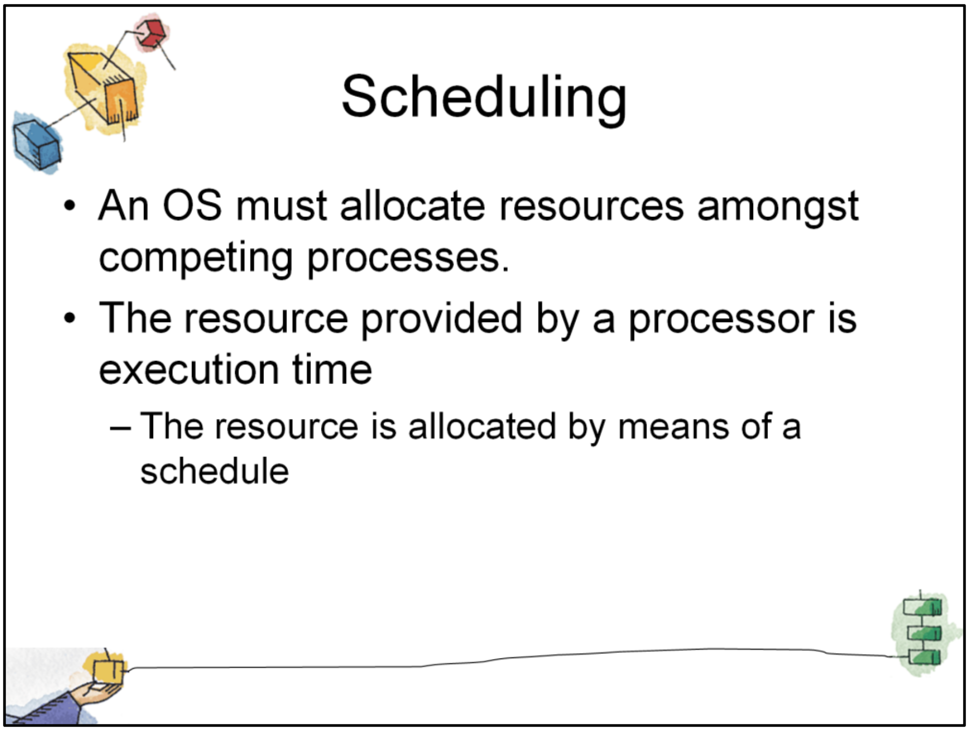
This chapter is concerned with scheduling on a system with a single processor.

It is possible to define and clarify many design issues related to scheduling in this context.

We examine the three types of processor scheduling:

- long term,
- medium term, and
- short term.

The bulk of the chapter focuses on short-term scheduling issues.



Scheduling

- An OS must allocate resources amongst competing processes.
- The resource provided by a processor is execution time
  - The resource is allocated by means of a schedule

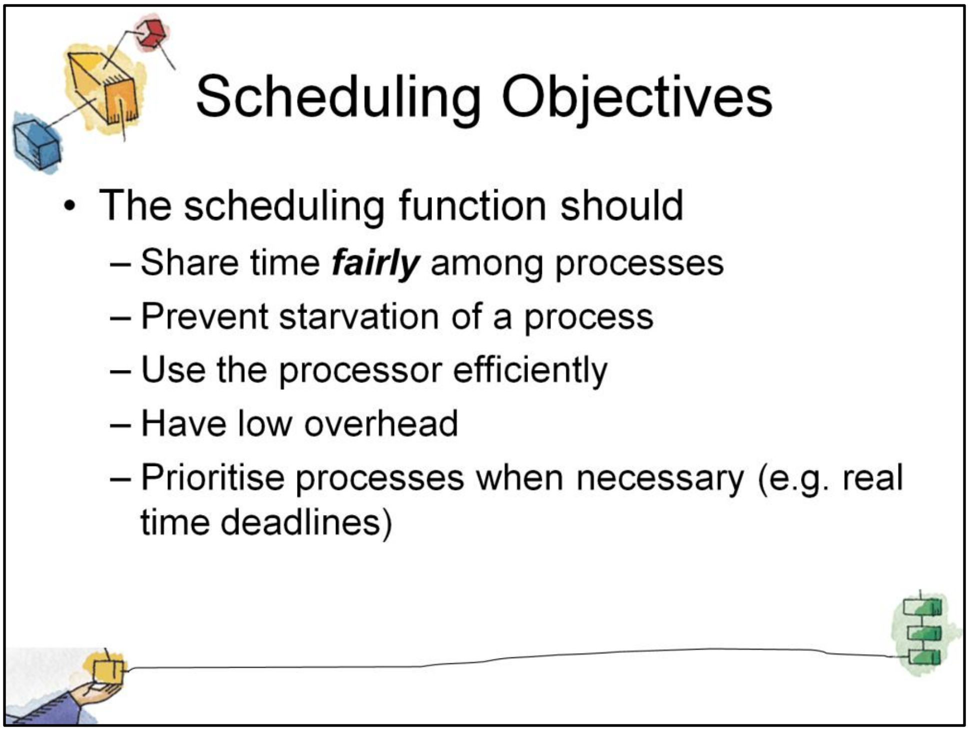
An operating system must allocate computer resources among the potentially competing requirements of multiple processes.

In the case of the processor, the resource to be allocated is execution time on the processor and the

means of allocation is scheduling.

Overall Aim of Scheduling

- The aim of processor scheduling is to assign processes to be executed by the processor over time,
  - in a way that meets system objectives, such as response time, throughput, and processor efficiency.

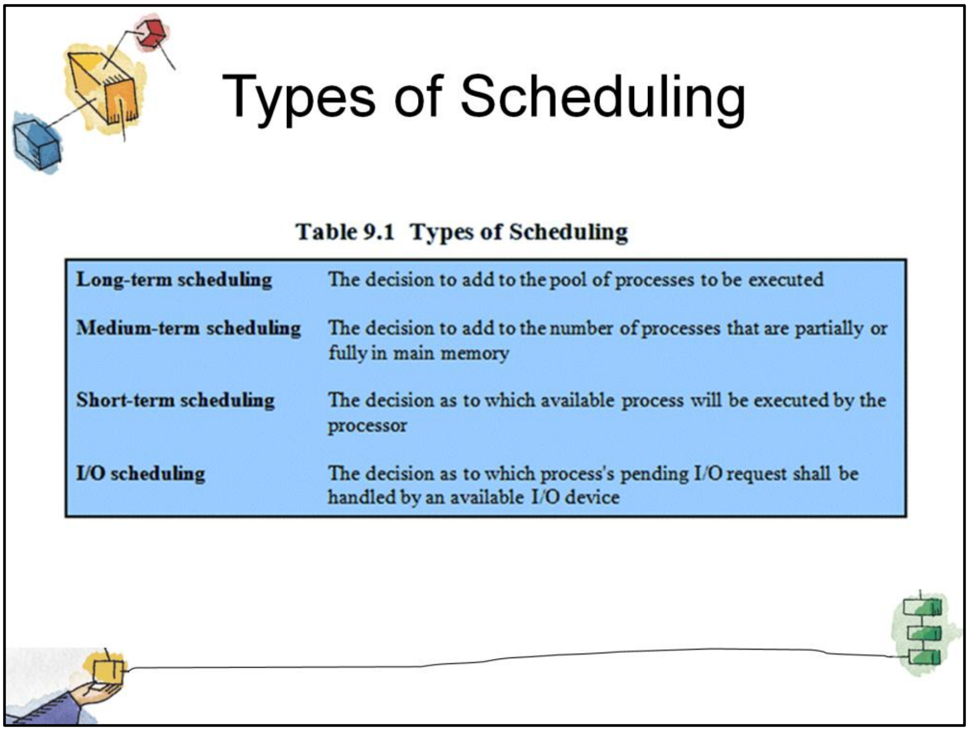


Scheduling Objectives

- The scheduling function should
  - Share time ***fairly*** among processes
  - Prevent starvation of a process
  - Use the processor efficiently
  - Have low overhead
  - Prioritise processes when necessary (e.g. real time deadlines)

The scheduling function must be designed to satisfy a number of objectives, including fairness, lack of starvation of any particular process, efficient use of processor time, and low overhead.

In addition, the scheduling function may need to take into account different levels of priority or real-time deadlines for the start or completion of certain processes.



Types of Scheduling

Table 9.1 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

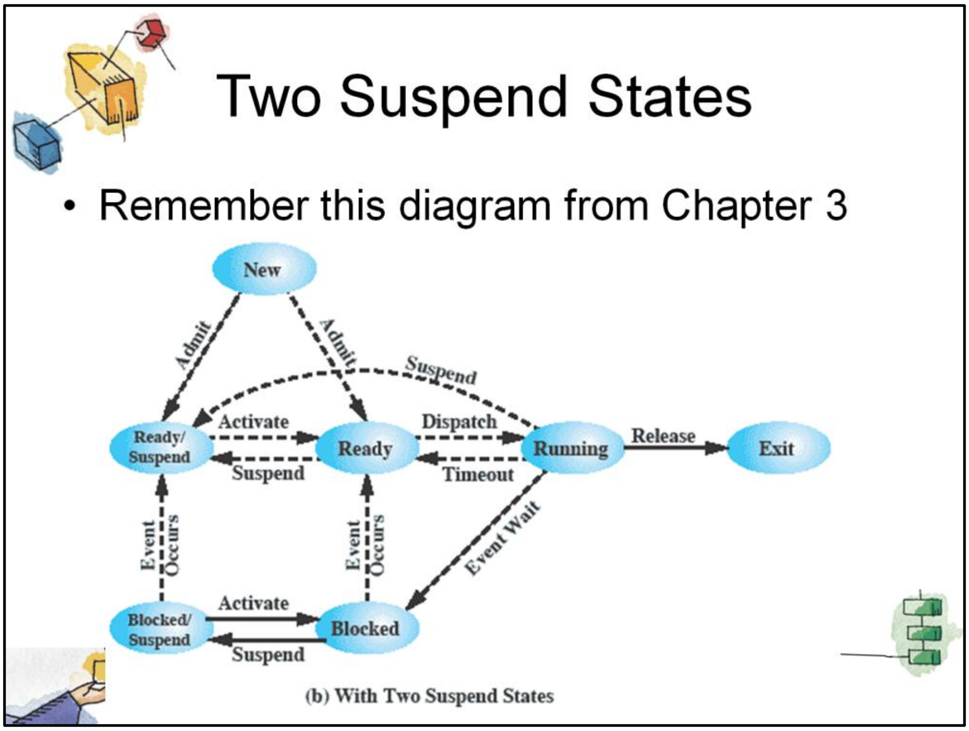
Long-term scheduling is performed when a new process is created.

- This is a decision whether to add a new process to the set of processes that are currently active.

Medium-term scheduling is a part of the swapping function.

- This is a decision whether to add a process to those that are at least partially in main memory and therefore available for execution.

Short-term scheduling is the actual decision of which ready process to execute next.



Two Suspend States

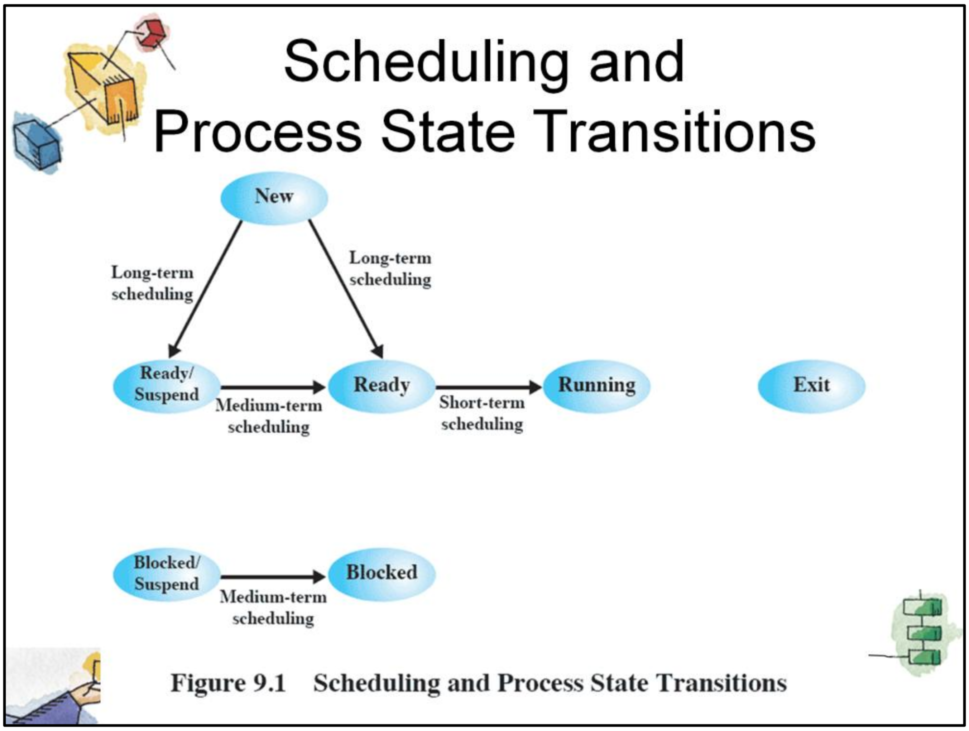
- Remember this diagram from Chapter 3

(b) With Two Suspend States

Figure 3.9b

Two suspend states allow all processes which are not actually running to be swapped.

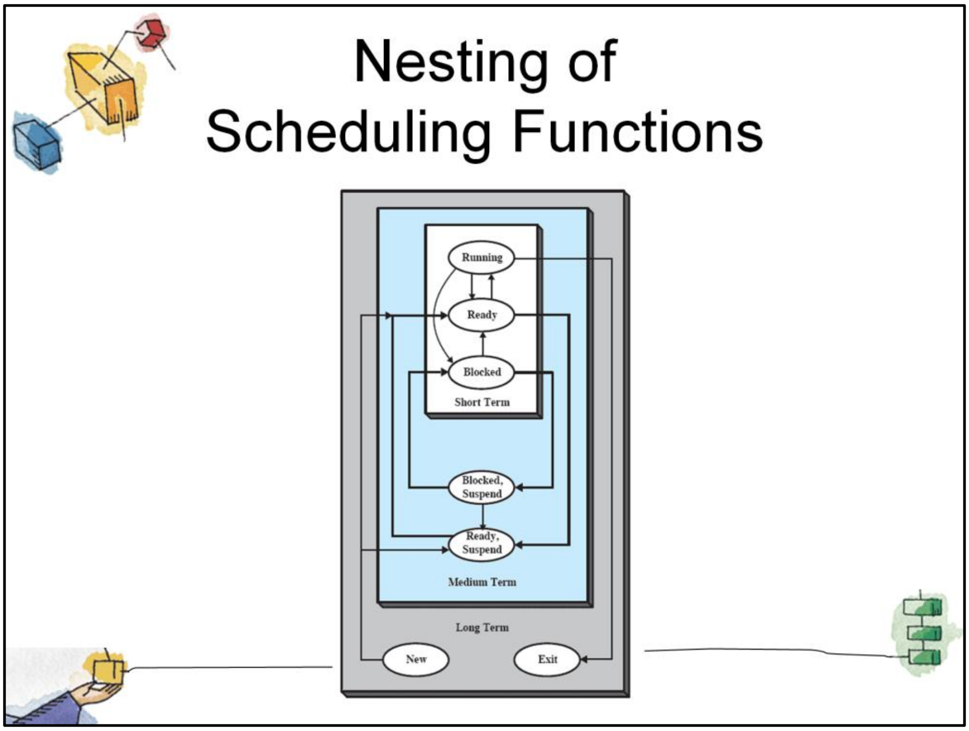
- **Ready:** The process is in main memory and available for execution.
- **Blocked:** The process is in main memory and awaiting an event.
- **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
- **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.



Scheduling and Process State Transitions

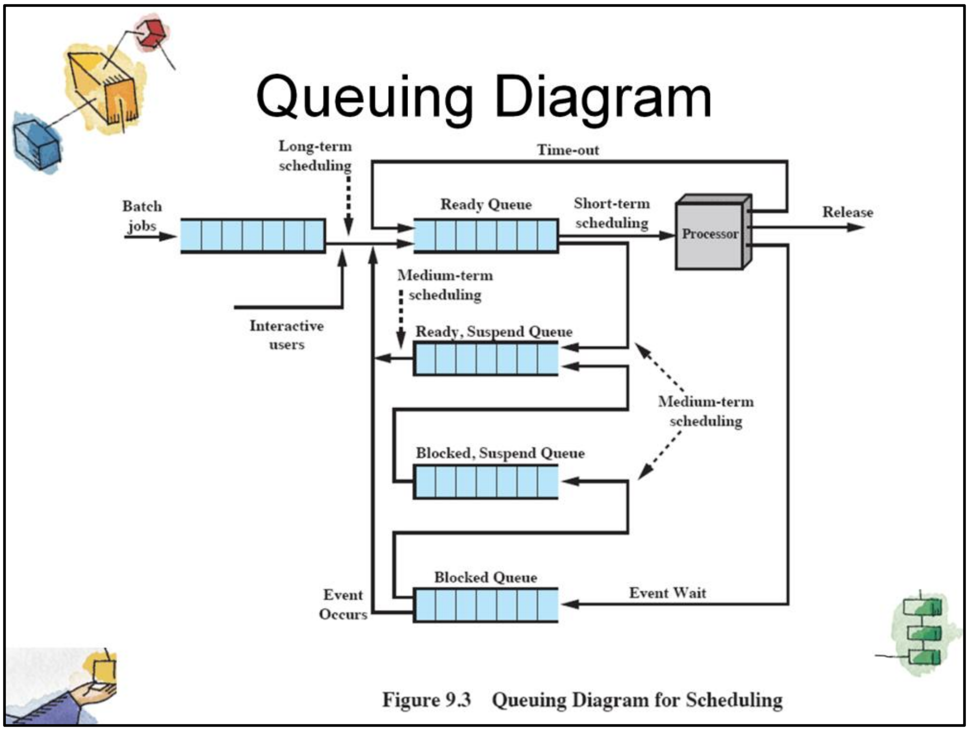
Figure 9.1 Scheduling and Process State Transitions

This figure relates the scheduling functions to the process state transition diagram shown in chapter 3 (fig 3.9b)



Nesting of Scheduling Functions

This figure reorganizes the state transition diagram to suggest the nesting of scheduling functions.



Queuing Diagram

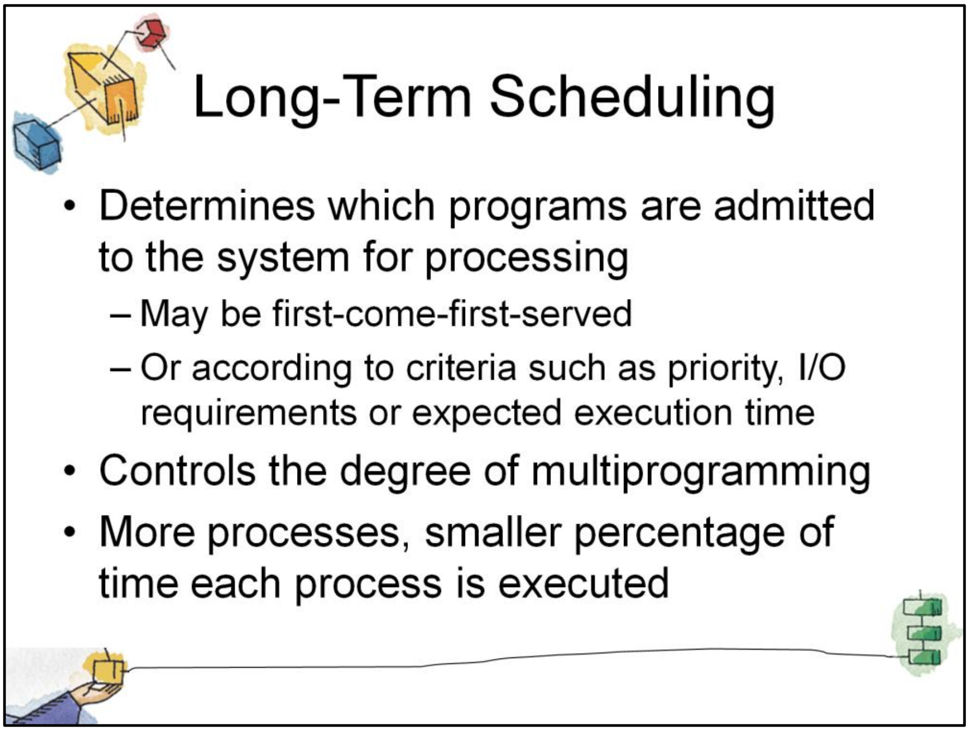
Figure 9.3 Queuing Diagram for Scheduling

This figure shows that scheduling affects the performance of the system because it determines which processes will wait and which will progress.

The figure shows the queues involved in the state transitions of a process.

- For simplicity, new processes are shown going directly to the Ready state, whereas the earlier figures (9.1 and 9.2) show the option of either the Ready state or the Ready/Suspend state.

Fundamentally, scheduling is a matter of managing queues to minimize queuing delay and to optimize performance in a queuing environment.



Long-Term Scheduling

- Determines which programs are admitted to the system for processing
  - May be first-come-first-served
  - Or according to criteria such as priority, I/O requirements or expected execution time
- Controls the degree of multiprogramming
- More processes, smaller percentage of time each process is executed

The long-term scheduler determines which programs are admitted to the system for processing.

- Thus, it controls the degree of multiprogramming.

In terms of frequency of execution, the long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process and which one to take.

Once admitted, a **job** or user program becomes a process and is added to the queue for the short-term scheduler.

- Or (as on some systems) a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

The decision as to which job to admit next can be on a simple first-come-first-served basis,

- or it can be a tool to manage system performance.

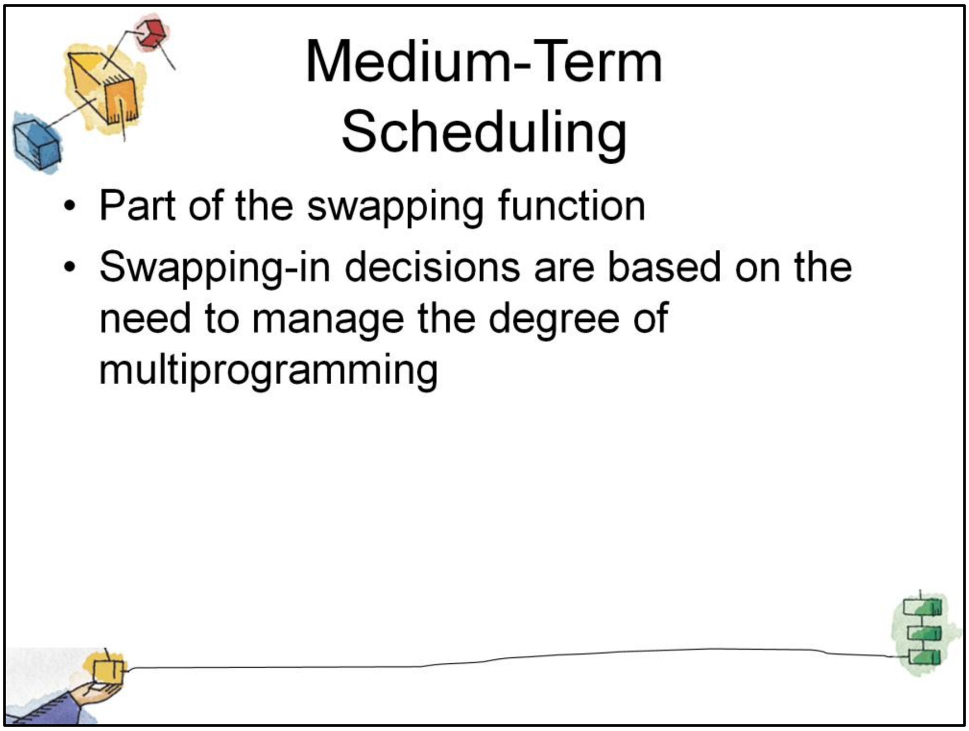
The criteria used may include

- priority,
- expected execution time, and
- I/O requirements.

The decision as to when to create a new process is generally driven by the desired degree of multiprogramming.

The more processes that are created, the smaller is the percentage of time that each process can be executed

- i.e. more processes are competing for the same amount of processor time.



Medium-Term Scheduling

- Part of the swapping function
- Swapping-in decisions are based on the need to manage the degree of multiprogramming

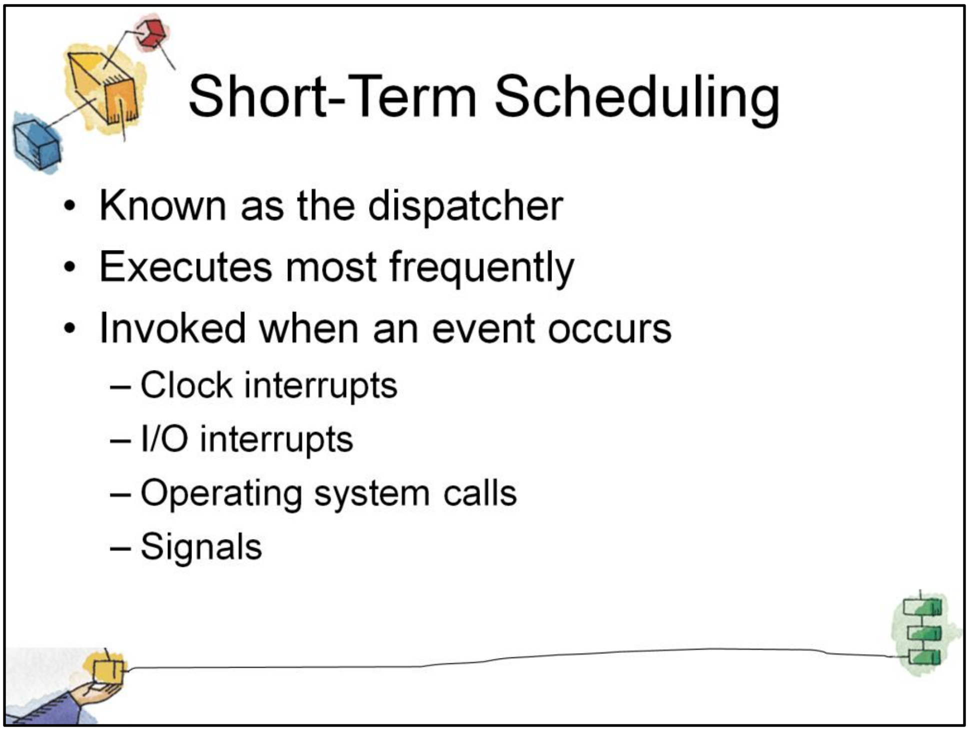
The medium-term scheduler is executed somewhat more frequently.

Medium-term scheduling is part of the swapping function.

Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming.

On a system that does not use virtual memory, memory management is also an issue.

- Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.



Short-Term Scheduling

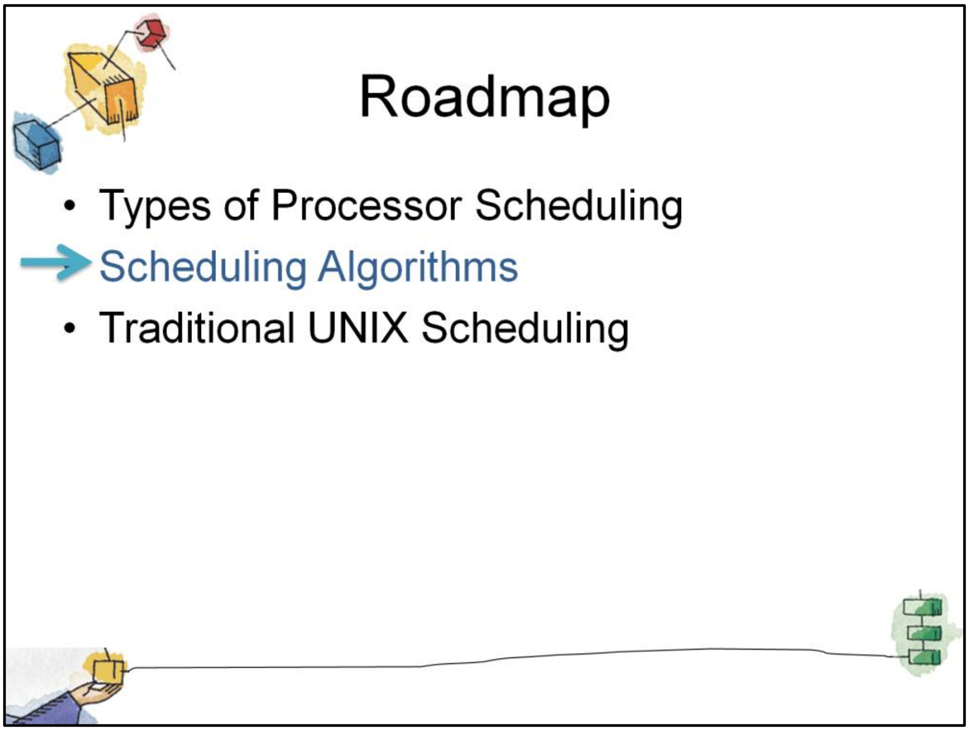
- Known as the dispatcher
- Executes most frequently
- Invoked when an event occurs
  - Clock interrupts
  - I/O interrupts
  - Operating system calls
  - Signals

Also known as the dispatcher, executes most frequently and makes the fine-grained decision of which process to execute next.

The short-term scheduler is invoked whenever an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favor of another.

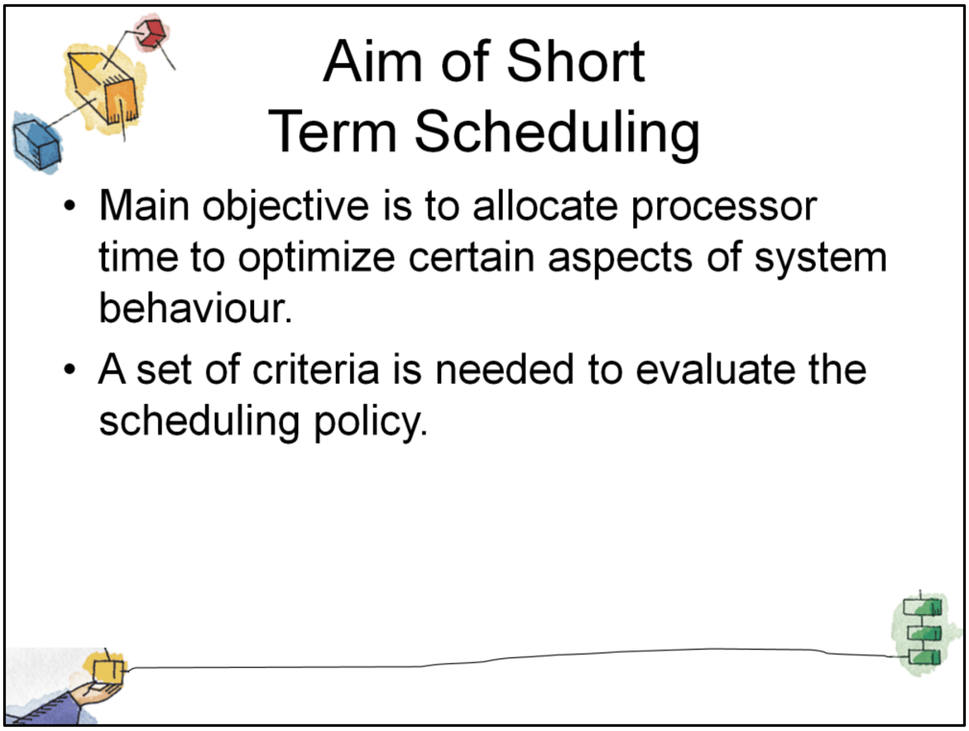
Examples of such events include

- Clock interrupts
- I/O interrupts
- Operating system calls
- Signals (e.g., semaphores)



Roadmap

- Types of Processor Scheduling
- → Scheduling Algorithms
- Traditional UNIX Scheduling

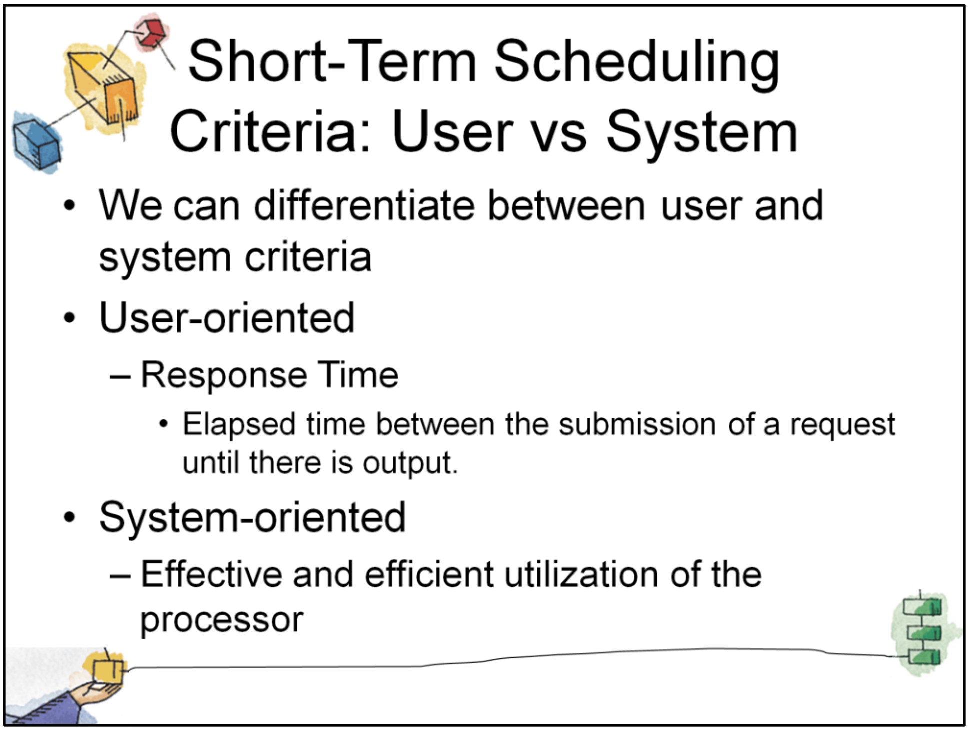


Aim of Short Term Scheduling

- Main objective is to allocate processor time to optimize certain aspects of system behaviour.
- A set of criteria is needed to evaluate the scheduling policy.

The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior.

Generally, a set of criteria is established against which various scheduling policies may be evaluated.



Short-Term Scheduling Criteria: User vs System

- We can differentiate between user and system criteria
- User-oriented
  - Response Time
    - Elapsed time between the submission of a request until there is output.
- System-oriented
  - Effective and efficient utilization of the processor

The commonly used criteria can be categorized along two dimensions.

We can make a distinction between user-oriented and system-oriented criteria.

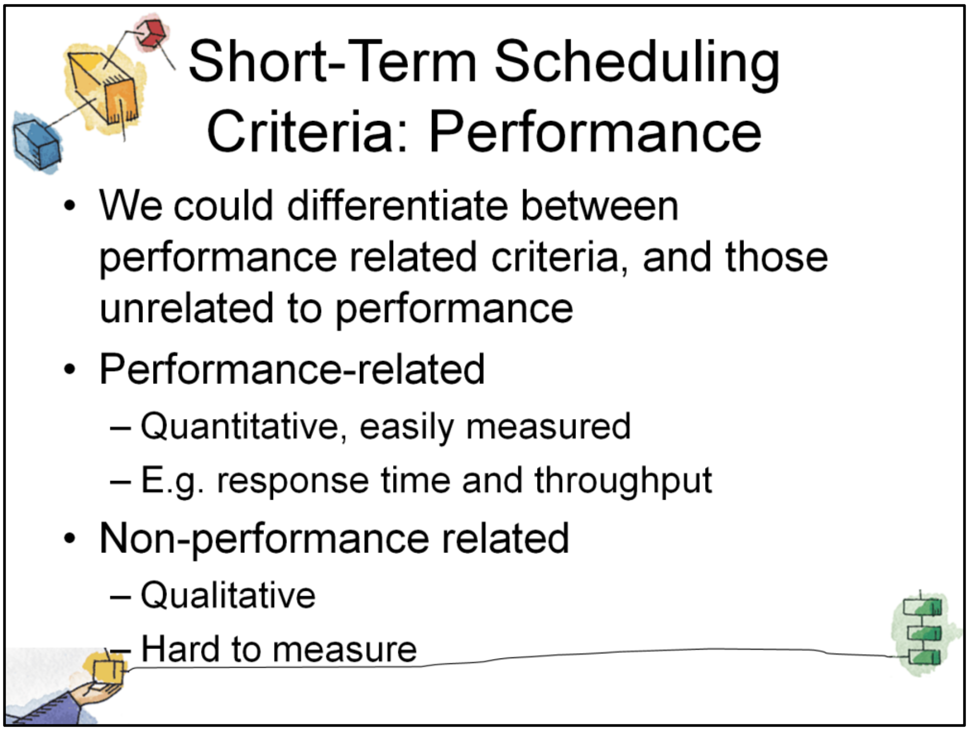
User oriented criteria relate to the behavior of the system as perceived by the individual user or process.

- •E.g. response time in an interactive system which is the elapsed time between the submission of a request until the response begins to appear as output.
- • We would like a scheduling policy that provides "good" service to various users. A threshold may be defined, say 2 seconds. Then a goal of the scheduling mechanism should be to maximize the number of users who experience an average response time of 2 seconds or less.

System oriented criteria is the focused on effective and efficient utilization of the processor.

- • An example is throughput, i.e. the rate at which processes are completed. Thus, throughput is of concern to a system administrator but not to the user population.

On a single-user system, it probably is not important to achieve high processor utilization or high throughput as long as the responsiveness of the system to user applications is acceptable.



Short-Term Scheduling Criteria: Performance

- We could differentiate between performance related criteria, and those unrelated to performance
- Performance-related
  - Quantitative, easily measured
  - E.g. response time and throughput
- Non-performance related
  - Qualitative
  - Hard to measure

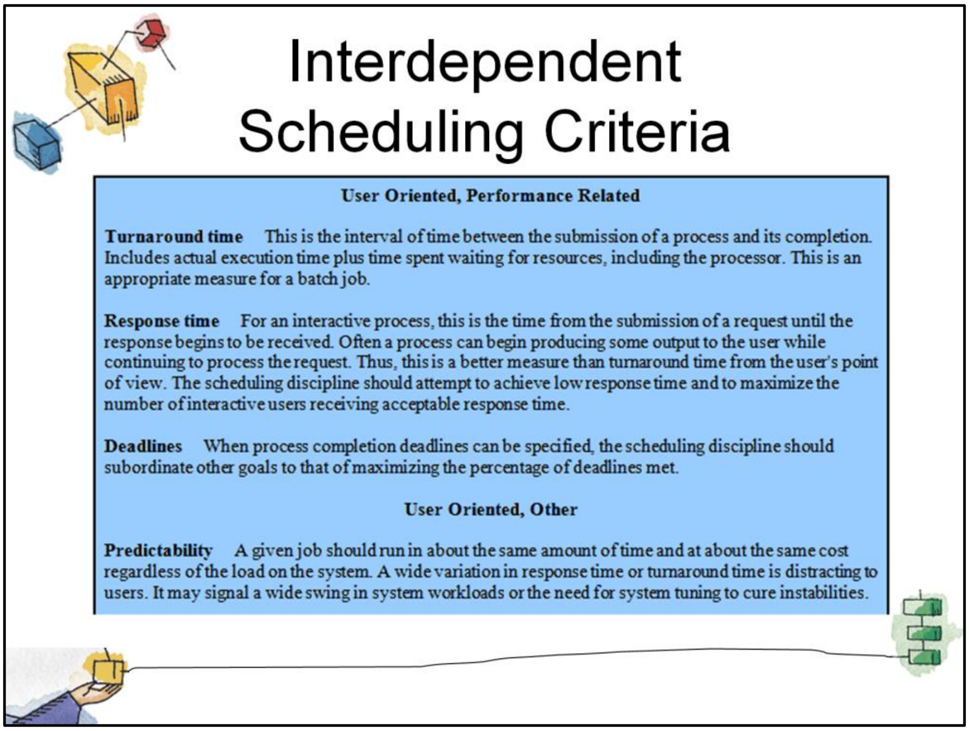
Another dimension along which criteria can be classified is those that are performance related and those that are not directly performance related.

Performance-related criteria are quantitative and generally can be readily measured.

- e.g. include response time and throughput.

Criteria that are not performance related are either qualitative in nature or do not lend themselves readily to measurement and analysis.

- e.g. predictability. We would like for the service provided to users to exhibit the same characteristics over time, independent of other work being performed by the system.
- To some extent, this criterion can be measured, by calculating variances as a function of workload. However, this is not nearly as straightforward as measuring throughput or response time as a function of workload.



Interdependent Scheduling Criteria

User Oriented, Performance Related

Turnaround time This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.

Response time For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.

Deadlines When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.

User Oriented, Other

Predictability A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.

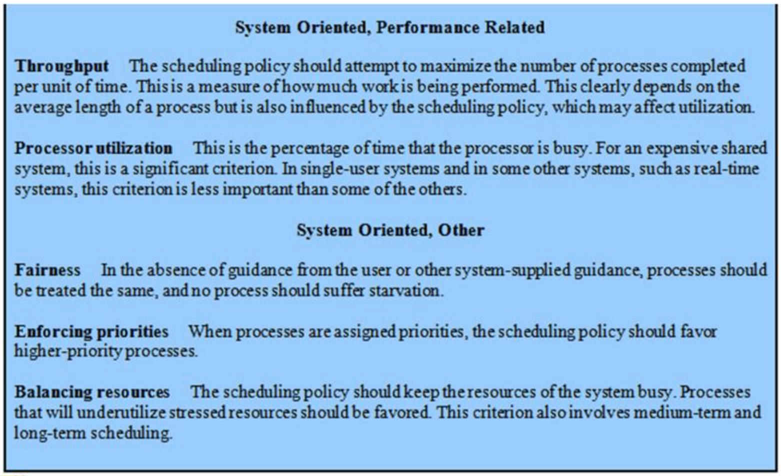
This table summarizes key scheduling criteria.

These are interdependent, and it is impossible to optimize all of them simultaneously.

- Thus, the design of a scheduling policy involves compromising among competing requirements;
- The relative weights given the various requirements will depend on the nature and intended use of the system.

In most interactive operating systems, whether single user or time shared, adequate response time is the critical requirement.

Interdependent Scheduling Criteria cont.



System Oriented, Performance Related

Throughput The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.

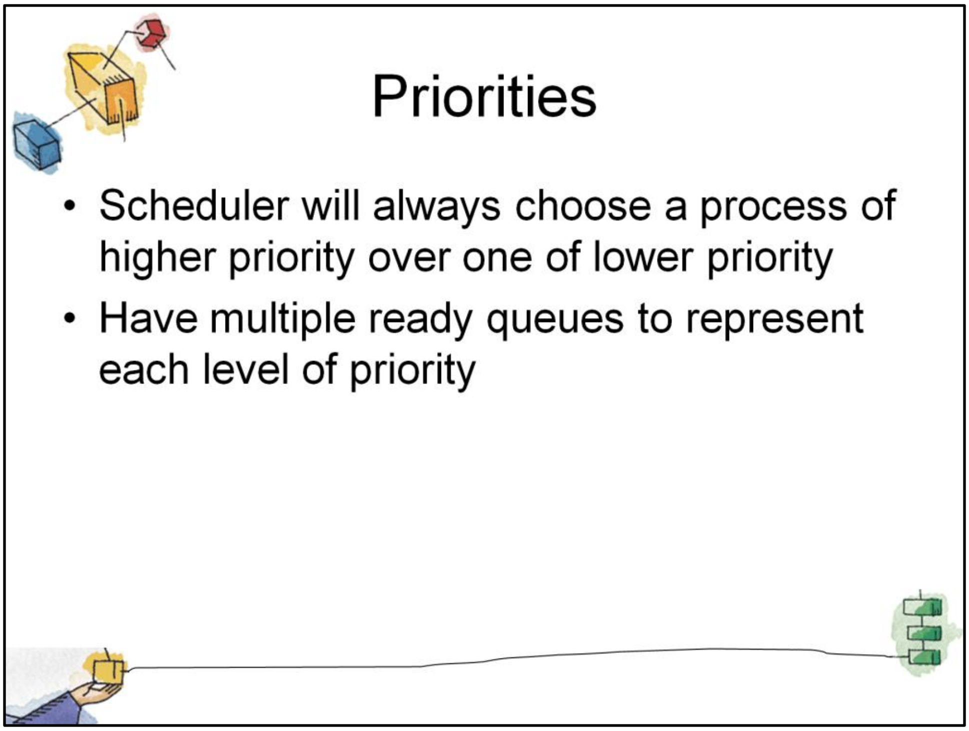
Processor utilization This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.

System Oriented, Other

Fairness In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.

Enforcing priorities When processes are assigned priorities, the scheduling policy should favor higher-priority processes.

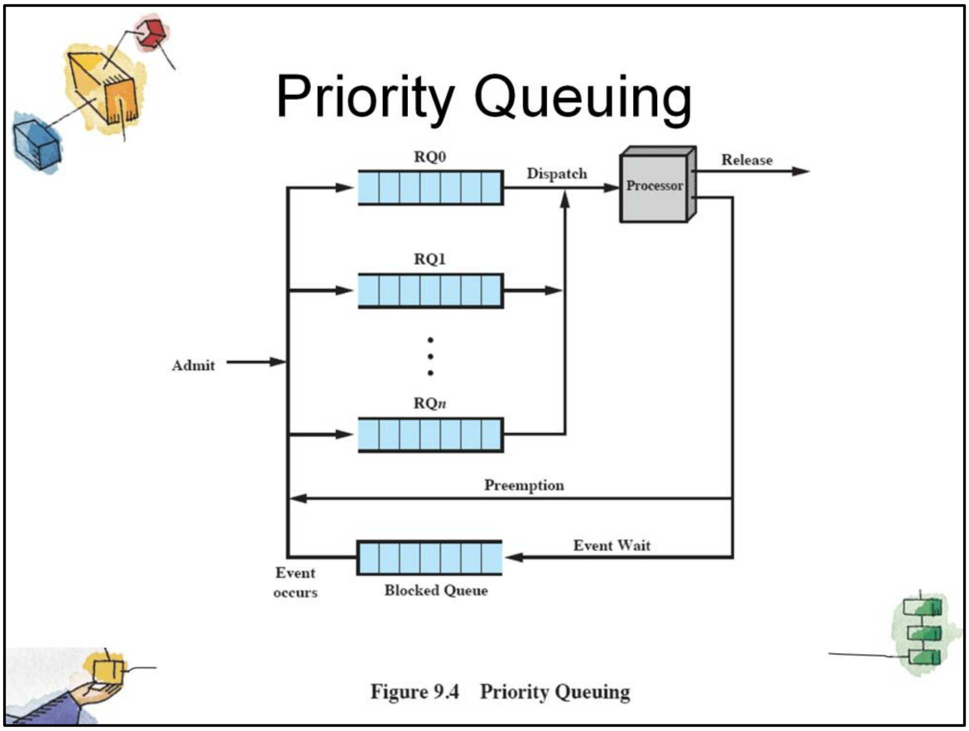
Balancing resources The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.



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

In many systems, each process is assigned a priority and the scheduler will always choose a process of higher priority over one of lower priority.



Priority Queuing

Figure 9.4 Priority Queuing

This figure illustrates the use of priorities. This figure is simplified and ignores the blocked and suspend queues of fig3.8a (chapter 3)

Instead of a single ready queue, we provide a set of queues, in descending order of priority:

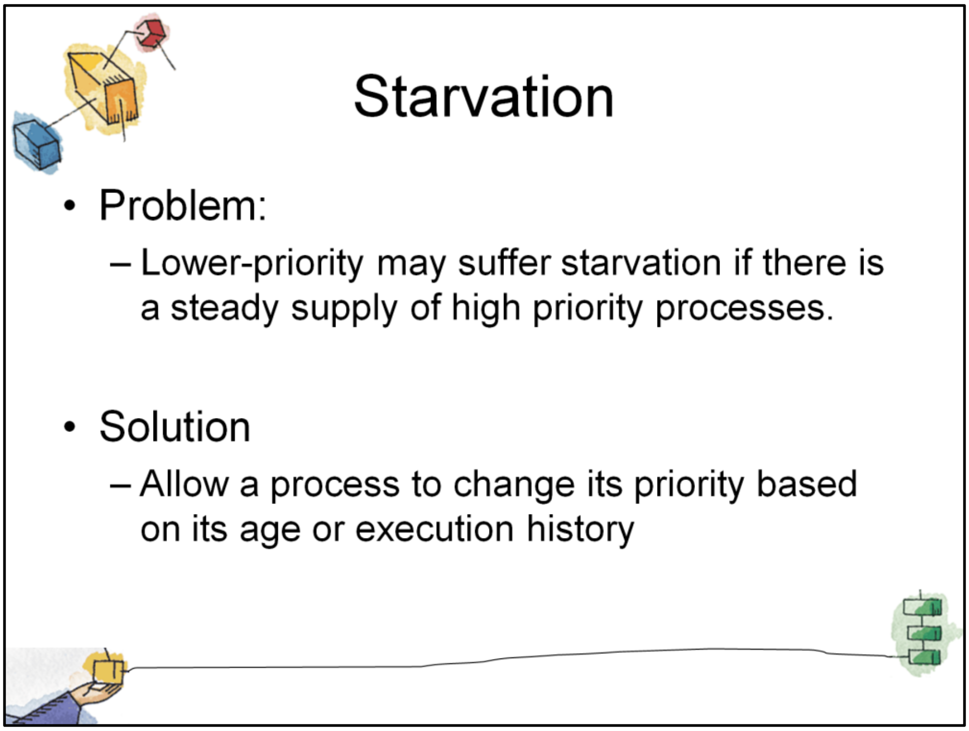
- RQ0, RQ1, . . . RQn, with priority[RQ*i*] > priority[RQ*j*] for i<j.

NB: In UNIX and many other systems, larger priority values represent lower priority processes;

- unless other-wise stated we follow that convention.
- Some systems, such as Windows, use the opposite convention: a higher number means a higher priority.

When a scheduling selection is to be made, the scheduler will start at the highest-priority ready queue (RQ0).

- If there are one or more processes in the queue, a process is selected using some scheduling policy.
- If RQ0 is empty, then RQ1 is examined, and so on.

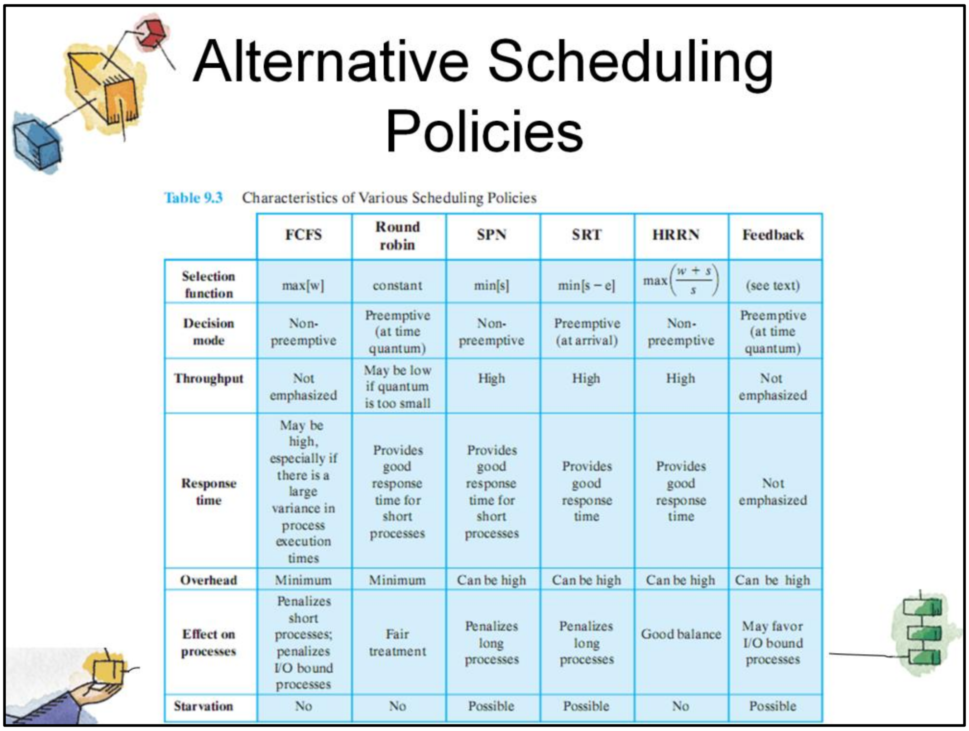


Starvation

- Problem:
  - Lower-priority may suffer starvation if there is a steady supply of high priority processes.
- Solution
  - Allow a process to change its priority based on its age or execution history

One problem with a pure priority scheduling scheme is that lower-priority processes may suffer starvation.

- This will happen if there is always a steady supply of higher-priority ready processes.
- If this behaviour is not desirable, the priority of a process can change with its age or execution history.



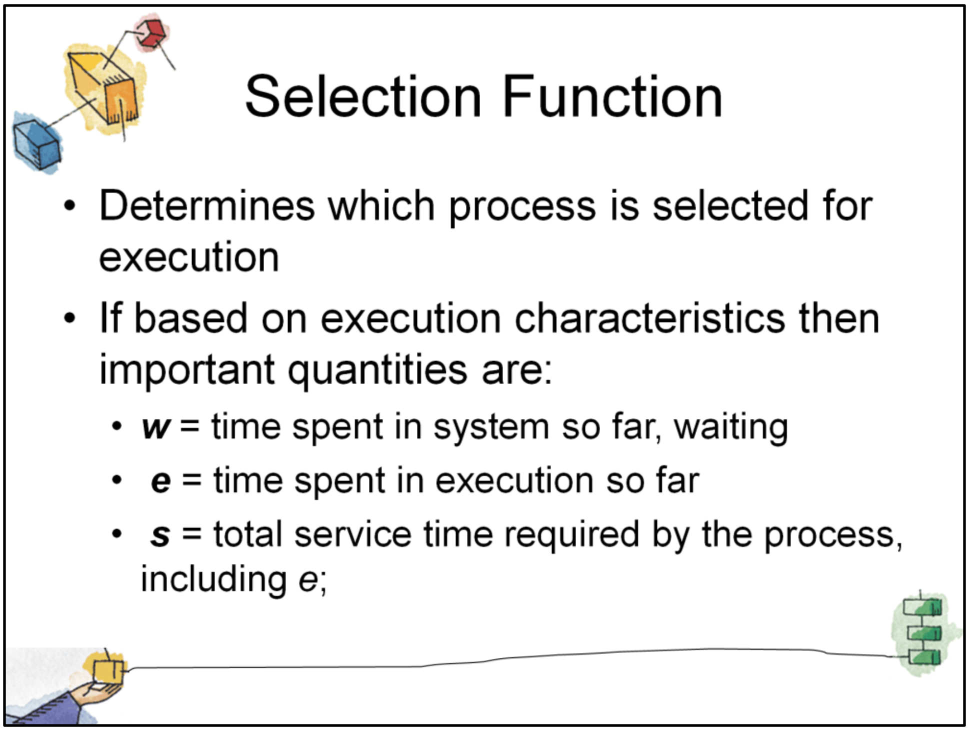
Alternative Scheduling Policies

Table 9.3 Characteristics of Various Scheduling Policies

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

This table may be difficult to read depending on the display – refer to table 9.3 in textbook

Animation on-click enlarges the image



Selection Function

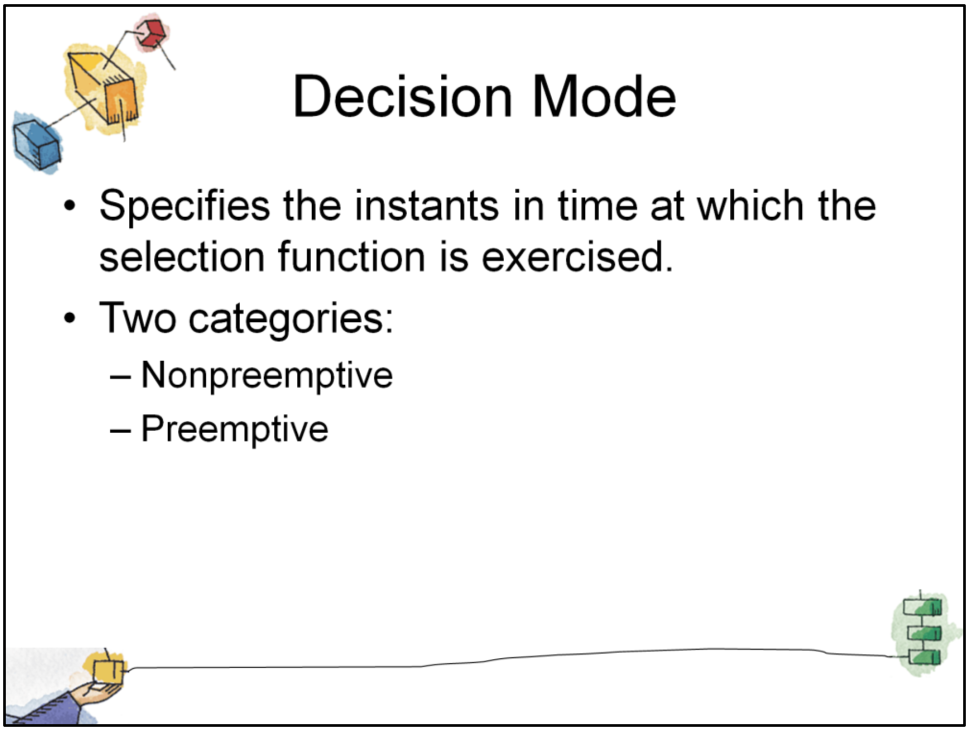
- Determines which process is selected for execution
- If based on execution characteristics then important quantities are:
  - ***w*** = time spent in system so far, waiting
  - ***e*** = time spent in execution so far
  - ***s*** = total service time required by the process, including *e*;

Selection function determines which process, among ready processes, is selected next for execution.

- May be based on priority, resource requirements, or the execution characteristics of the process.

In the latter case, three quantities are significant:

- ***w*** = time spent in system so far, waiting
- ***e*** = time spent in execution so far
- ***s*** = total service time required by the process, including *e*;

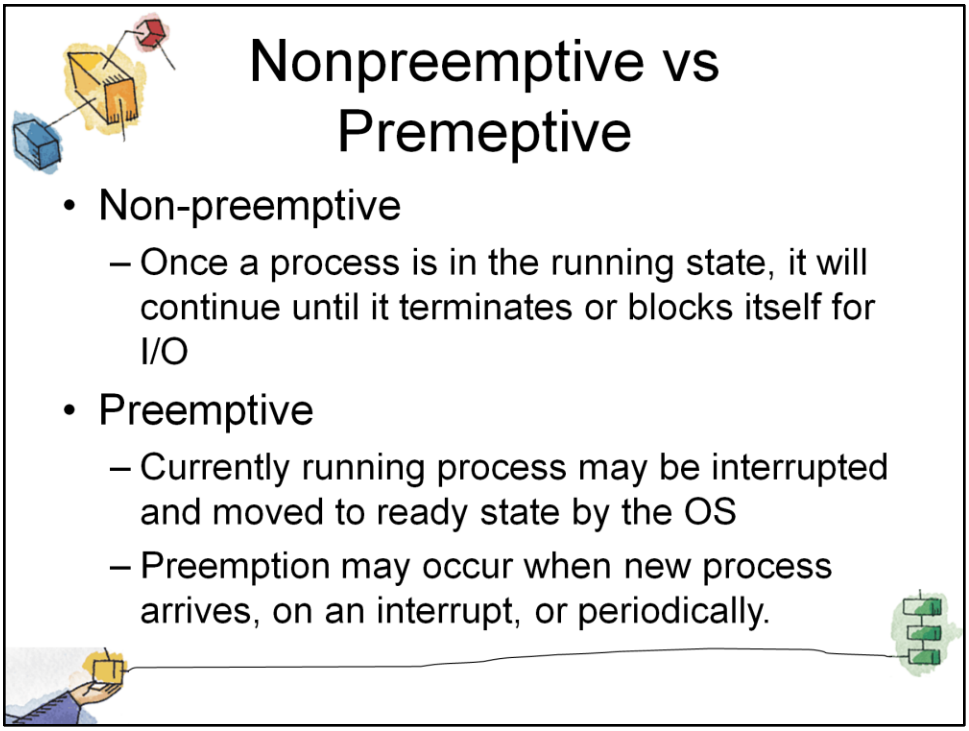


Decision Mode

- Specifies the instants in time at which the selection function is exercised.
- Two categories:
  - Nonpreemptive
  - Preemptive

Decision mode specifies the instants in time at which the selection function is exercised.

There are two general categories: non-preemptive and preemptive



Nonpreemptive vs Preemptive

- Non-preemptive
  - 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 ready state by the OS
  - Preemption may occur when new process arrives, on an interrupt, or periodically.

Non-preemptive

In this case, once a process is in the Running state, it continues to execute until

(a) it terminates or

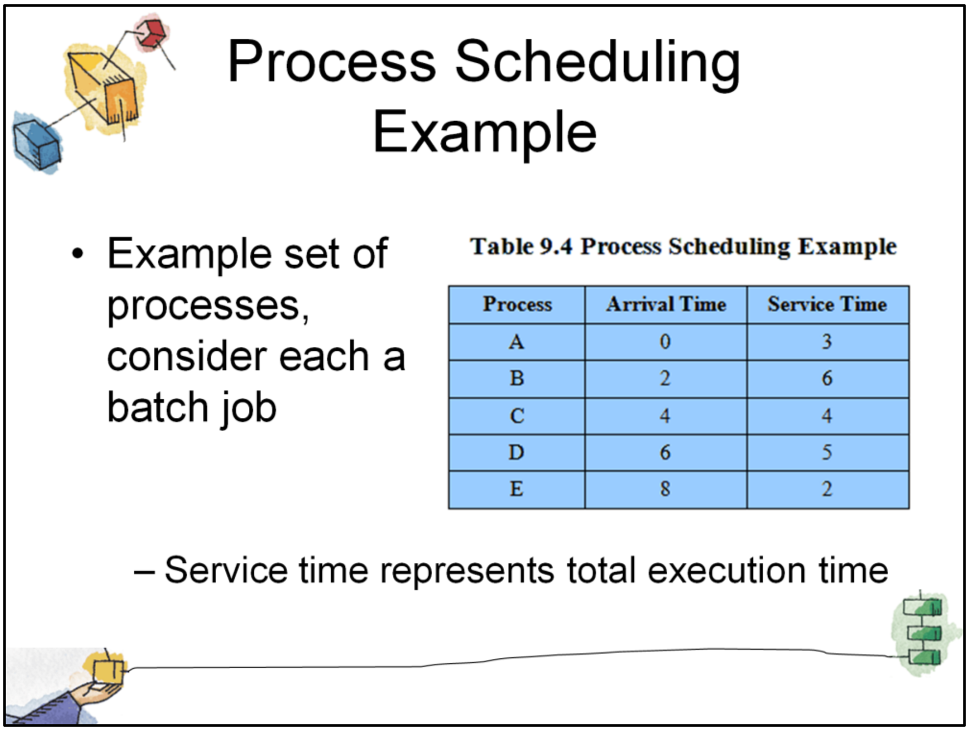
(b) it blocks itself to wait for I/O or to request some operating system service.

Preemptive:

The currently running process may be interrupted and moved to the Ready state by the operating system.

The decision to preempt may be performed

- when a new process arrives;
- when an interrupt occurs that places a blocked process in the Ready state;
- or periodically, based on a clock interrupt.



Process Scheduling Example

- Example set of processes, consider each a batch job

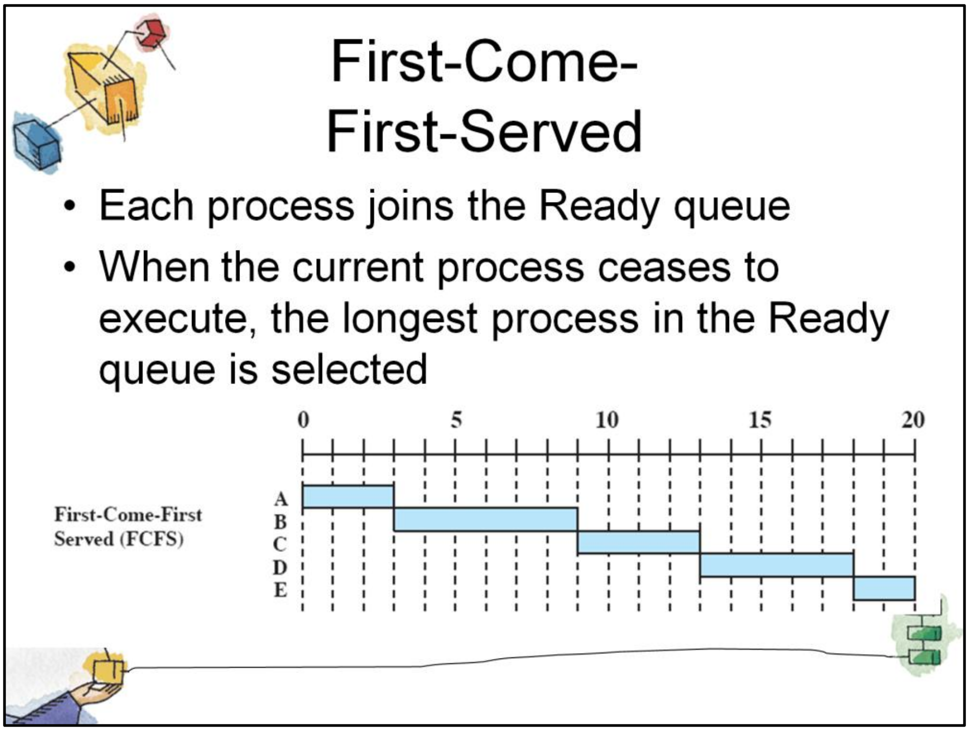
Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

  - Service time represents total execution time

We can think of these as batch jobs, with the service time being the total execution time required.

Movie button links to animation of *Process Scheduling Algorithms* at http://gaia.ecs.csus.edu/%7ezhangd/oscal/pscheduling.html



First-Come-First-Served

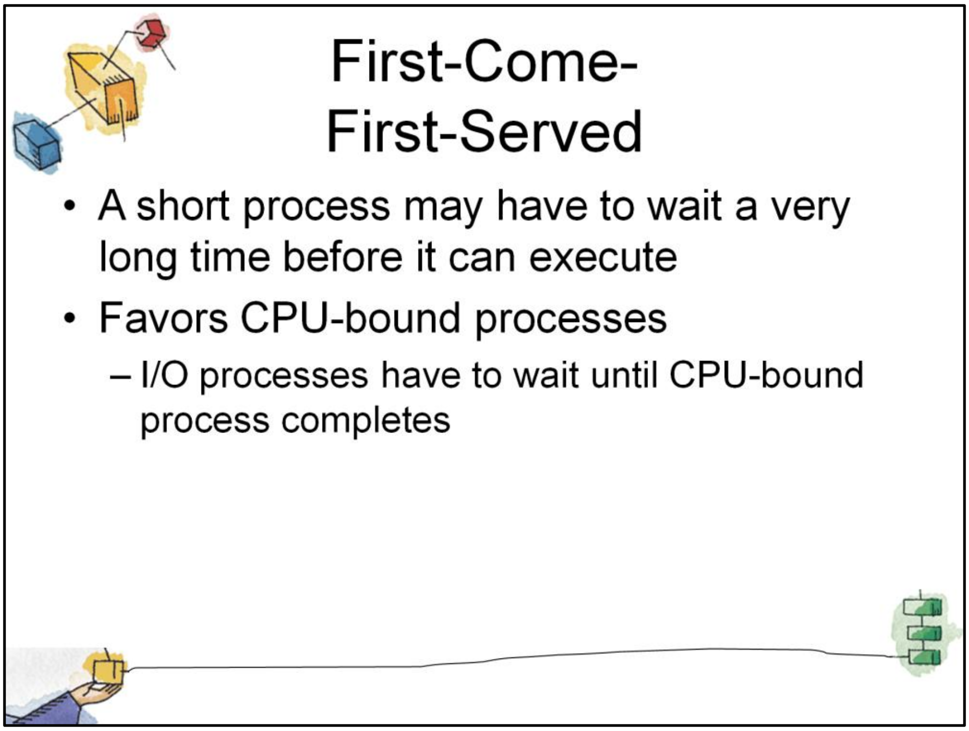
- Each process joins the Ready queue
- When the current process ceases to execute, the longest process in the Ready queue is selected

The simplest scheduling policy is first-come-first-served (FCFS),

- aka. first-in-first-out (FIFO) or a strict queuing scheme.

As each process becomes ready, it joins the ready queue.

When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running.



First-Come-First-Served

- 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

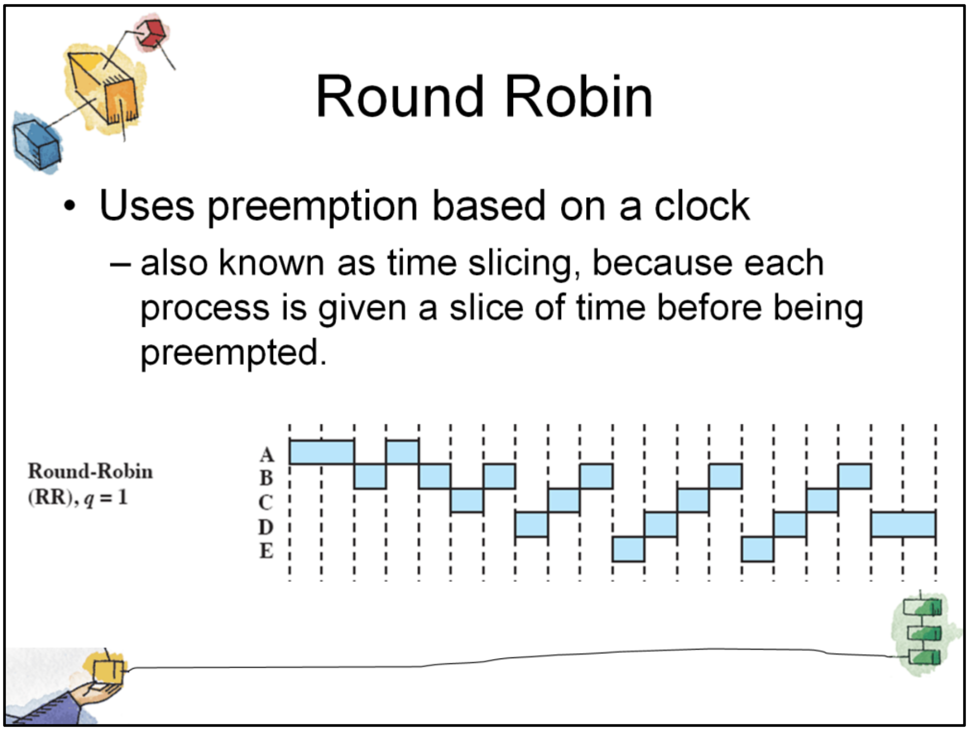
FCFS performs much better for long processes than short ones.

Another difficulty with FCFS is that it tends to favor processor-bound processes over I/O-bound processes.

FCFS is not an attractive alternative on its own for a uniprocessor system.

However, it is often combined with a priority scheme to provide an effective scheduler.

> Thus, the scheduler may maintain a number of queues, one for each priority level, and dispatch within each queue on a first-come-first-served basis.



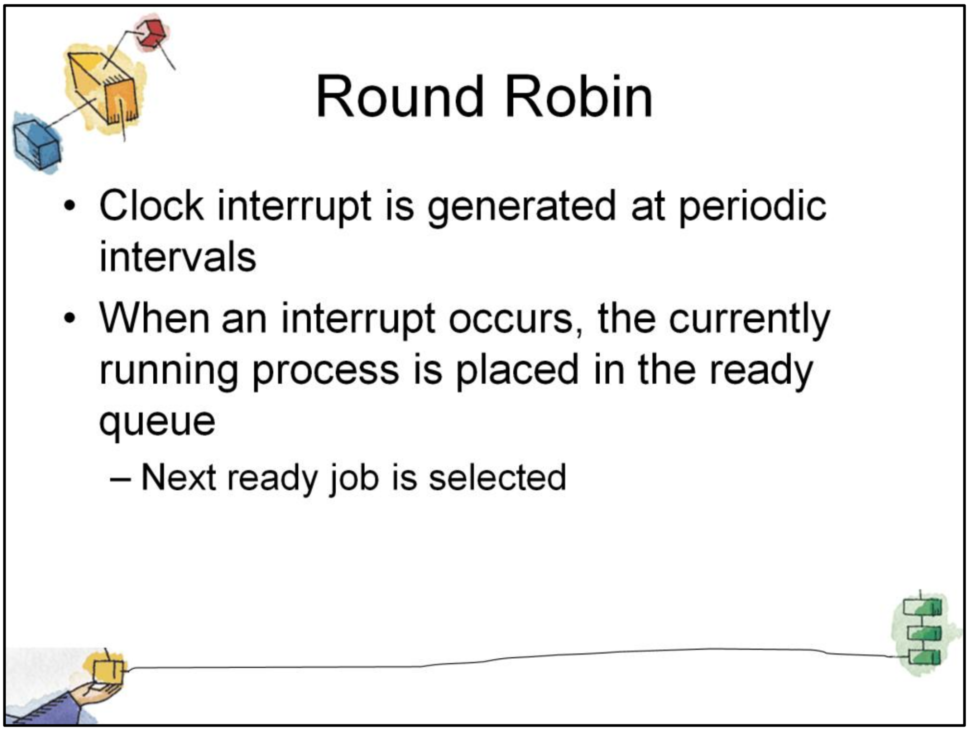
Round Robin

- Uses preemption based on a clock
  - also known as time slicing, because each process is given a slice of time before being preempted.

A straightforward way to reduce the penalty that short jobs suffer with FCFS is to use preemption based on a clock.

- The simplest such policy is round robin.

Also known as time slicing, because each process is given a slice of time before being preempted.

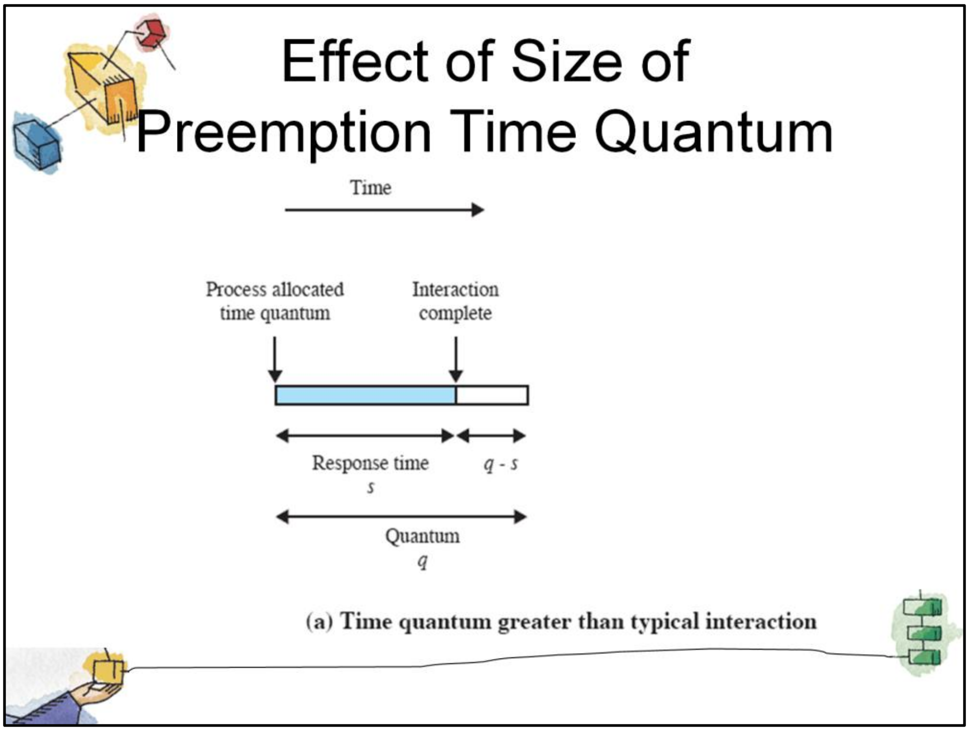


Round Robin

- Clock interrupt is generated at periodic intervals
- When an interrupt occurs, the currently running process is placed in the ready queue
  - Next ready job is selected

A clock interrupt is generated at periodic intervals.

- When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis.



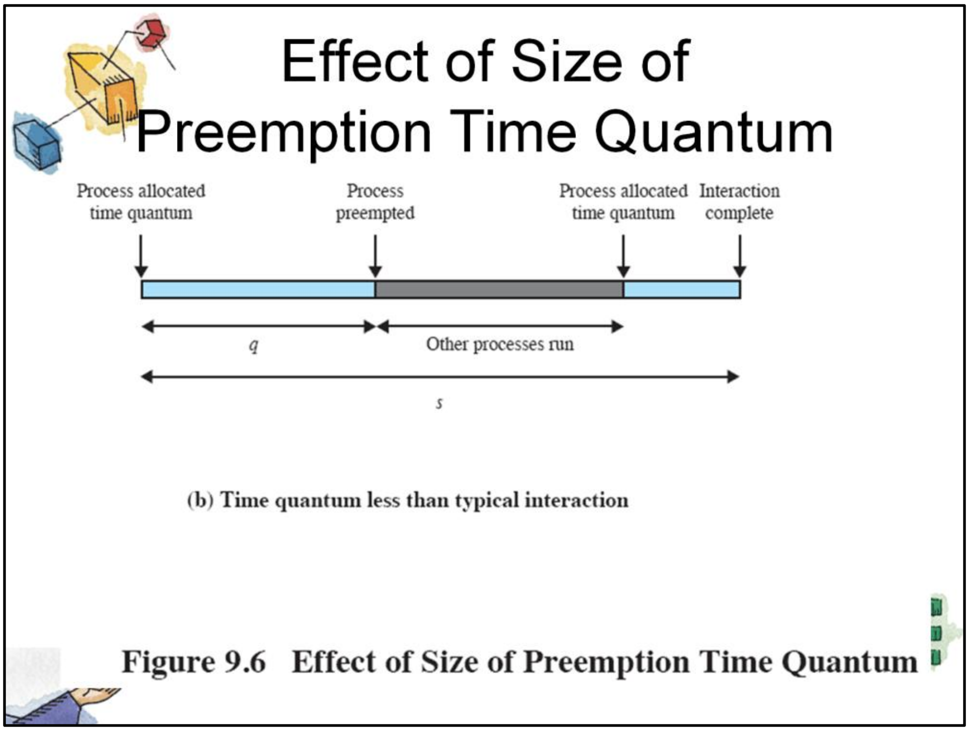
Effect of Size of Preemption Time Quantum

(a) Time quantum greater than typical interaction

The principal design issue is the length of the time quantum, or slice, to be used.

If the quantum is very short, then short processes will move through the system relatively quickly.

- BUT there is processing over-head involved in handling the clock interrupt and performing the scheduling and dispatching function.
- Thus, very short time quanta should be avoided.



Effect of Size of Preemption Time Quantum

Process allocated time quantum
Process preempted
Process allocated time quantum
Interaction complete
q
Other processes run
s

(b) Time quantum less than typical interaction

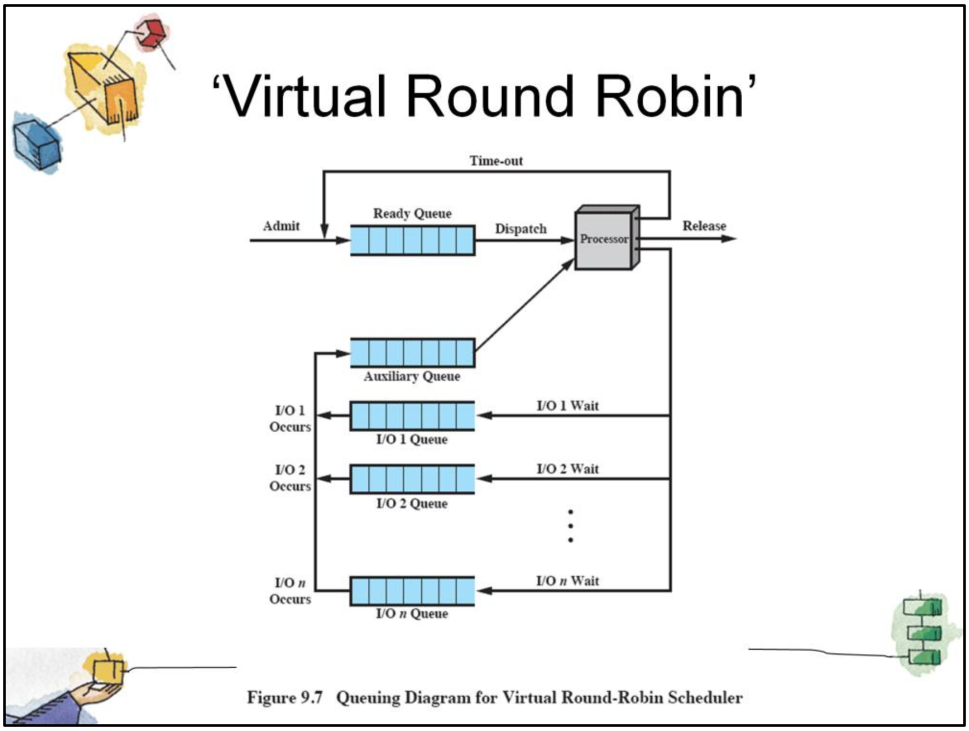
Figure 9.6 Effect of Size of Preemption Time Quantum

One useful guide is that the time quantum should be slightly greater than the time required for a typical interaction or process function.

- If it is less, then most processes will require at least two time quanta.

Processor-bound processes tend to receive an unfair portion of processor time, which results

- in poor performance for I/O-bound processes,
- inefficient use of I/O devices,
- and an increase in the variance of response time.



‘Virtual Round Robin’

Figure 9.7 Queuing Diagram for Virtual Round-Robin Scheduler

As per usual:

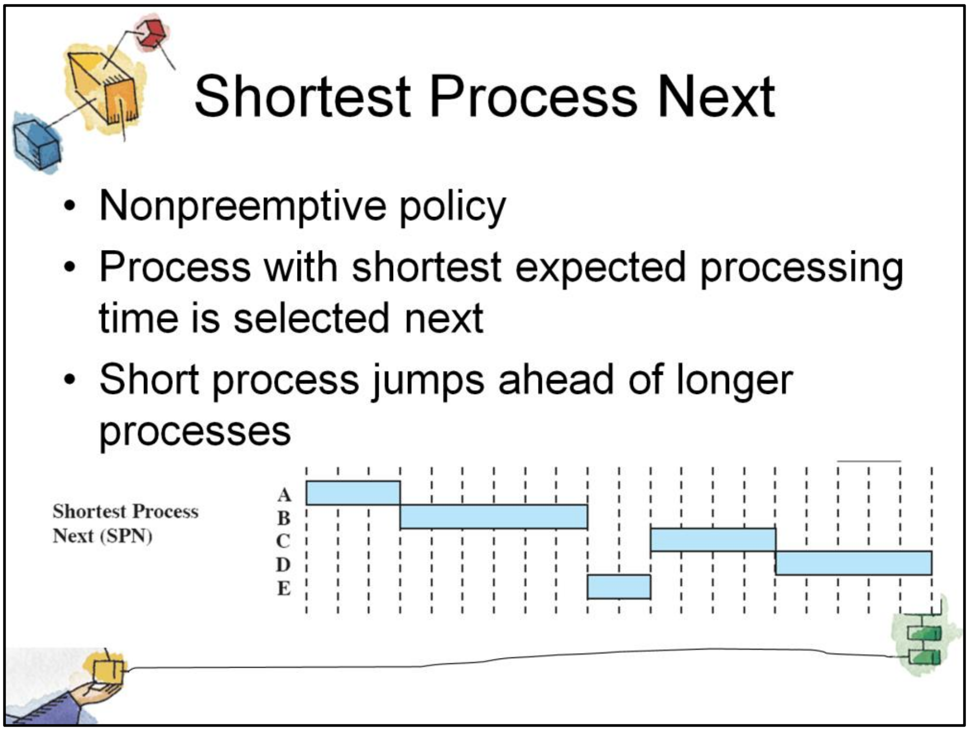
- New processes arrive and join the ready queue, which is managed on an FCFS basis.
- When a running process times out, it is returned to the ready queue.
- When a process is blocked for I/O, it joins an I/O queue.

The new feature is an FCFS auxiliary queue to which processes are moved after being released from an I/O block.

When a dispatching decision is to be made, processes in the auxiliary queue get preference over those in the main ready queue.

When a process is dispatched from the auxiliary queue, it runs no longer than a time equal to the basic time quantum minus the total time spent running since it was last selected from the main ready queue.

Performance studies indicate that this approach is indeed superior to round robin in terms of fairness.



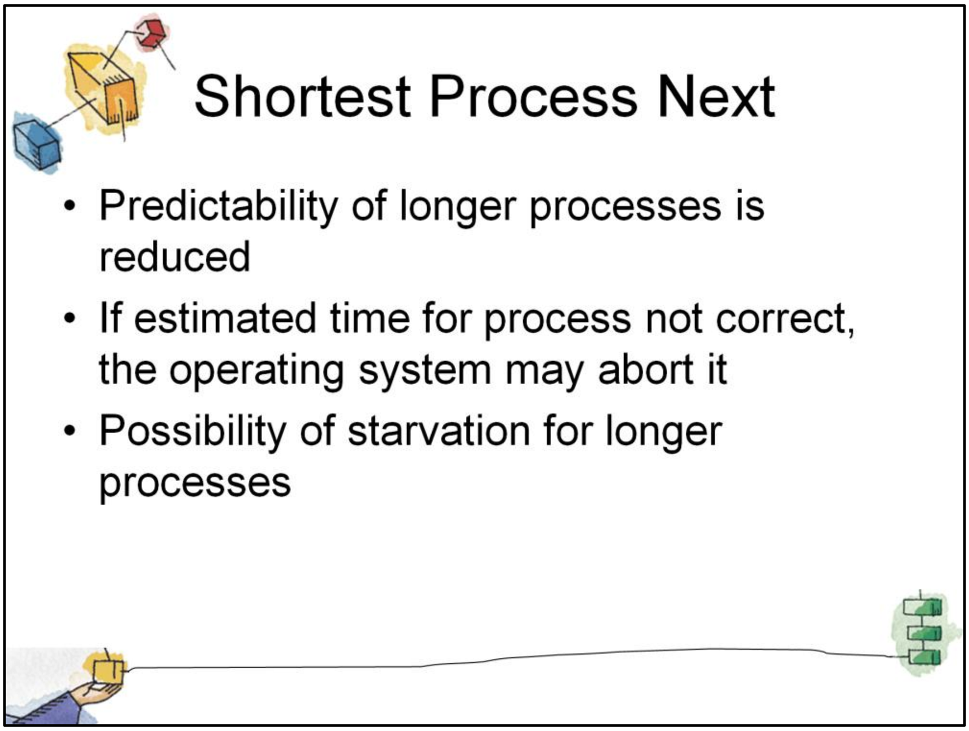
Shortest Process Next

- Nonpreemptive policy
- Process with shortest expected processing time is selected next
- Short process jumps ahead of longer processes

Shortest Process Next (SPN)
A
B
C
D
E

A non-preemptive policy in which the process with the shortest expected processing time is selected next.

- Thus a short process will jump to the head of the queue past longer jobs.



Shortest Process Next

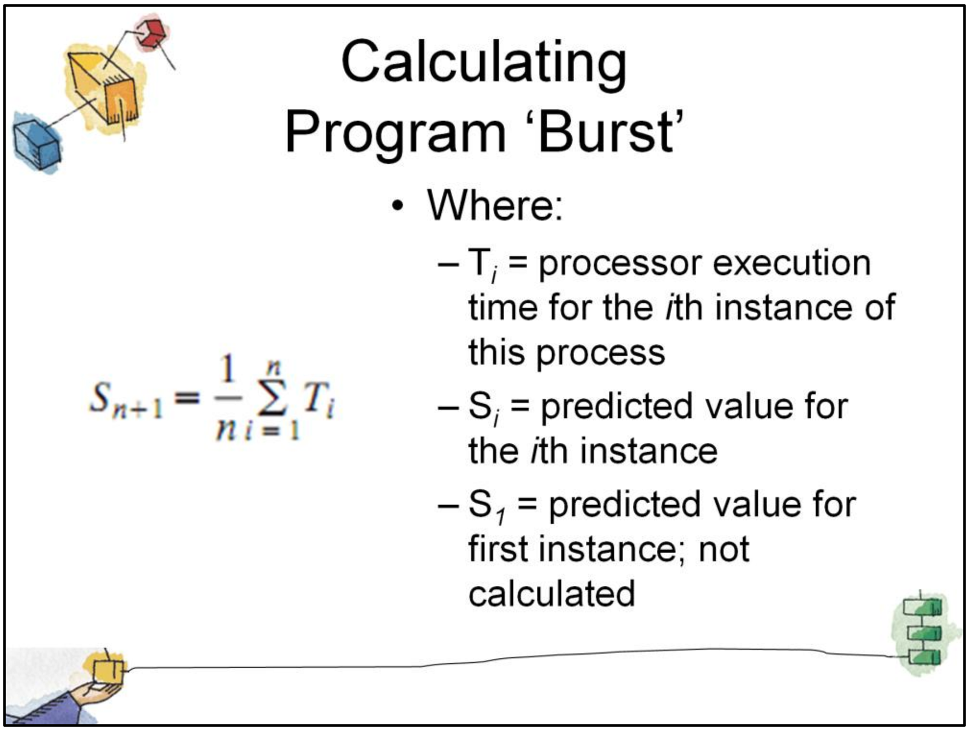
- Predictability of longer processes is reduced
- If estimated time for process not correct, the operating system may abort it
- Possibility of starvation for longer processes

Overall performance is significantly improved in terms of response time.

- However, the variability of response times is increased, especially for longer processes, and thus predictability is reduced.

One difficulty with the SPN policy is the need to know or at least estimate the required processing time of each process.

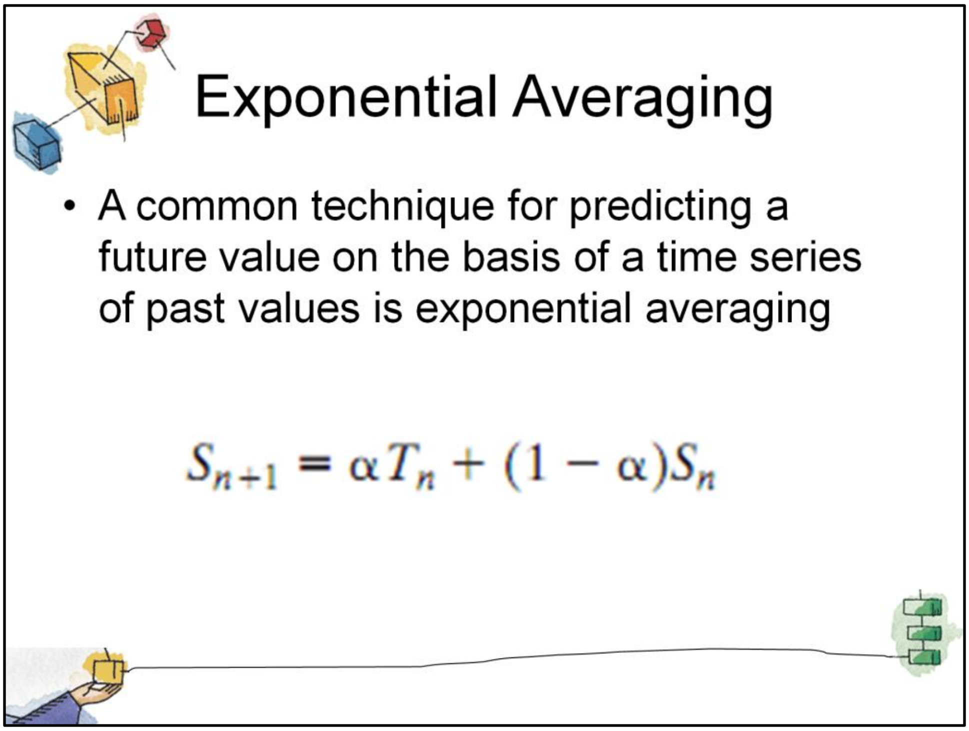
- For batch jobs, the system may require the programmer to estimate the value and supply it to the operating system.
- If the programmer’s estimate is substantially under the actual running time, the system may abort the job.



Calculating Program 'Burst'

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i$$

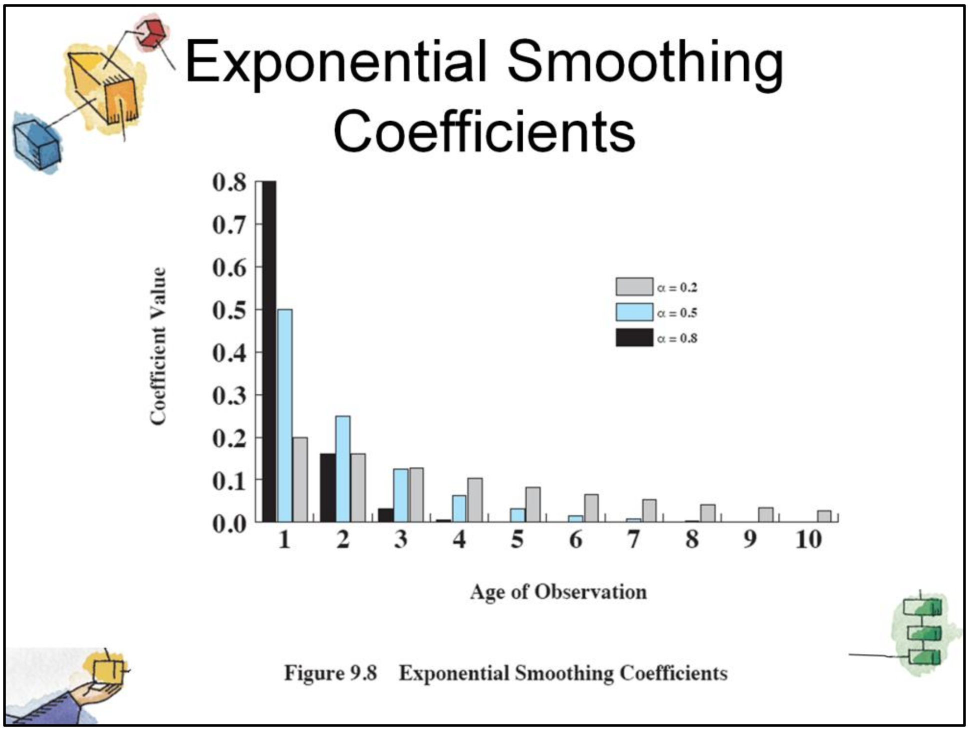
- Where:
  - T_i = processor execution time for the *i*th instance of this process
  - S_i = predicted value for the *i*th instance
  - S_1 = predicted value for first instance; not calculated



Exponential Averaging

- A common technique for predicting a future value on the basis of a time series of past values is exponential averaging

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



Exponential Smoothing Coefficients

Figure 9.8 Exponential Smoothing Coefficients

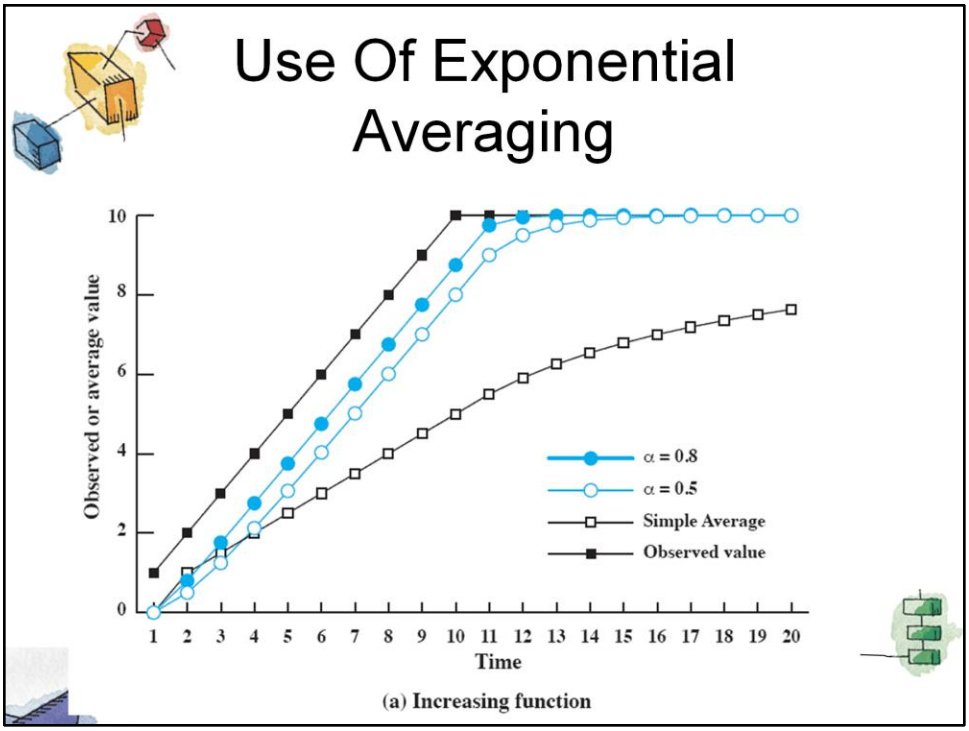
The size of the coefficient as a function of its position in the expansion is shown here.

The larger the value of α, the greater the weight given to the more recent observations.

> For α = 0.8, virtually all of the weight is given to the four most recent observations, whereas for α = 0.2, the averaging is effectively spread out over the eight or so most recent observations.

The advantage of using a value of α close to 1 is that the average will quickly reflect a rapid change in the observed quantity.

The disadvantage is that if there is a brief surge in the value of the observed quantity and it then settles back to some average value, the use of a large value of α will result in jerky changes in the average.

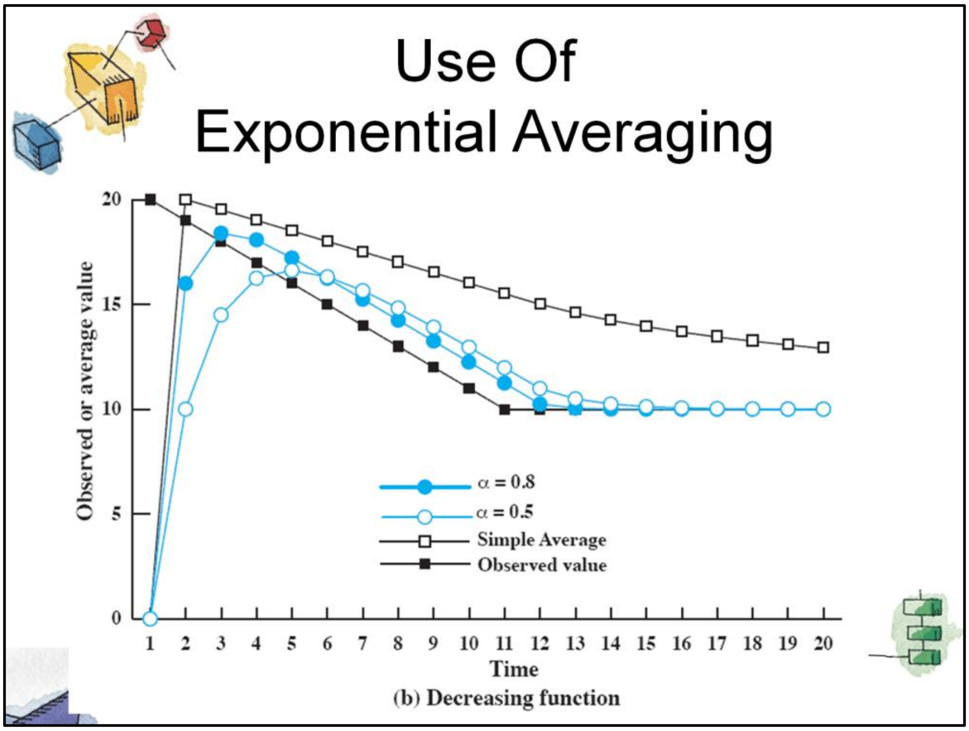


Use Of Exponential Averaging

(a) Increasing function

Figure 9.9 compares simple averaging with exponential averaging (for two different values of α).

Here (Figure 9.9a), the observed value begins at 1, grows gradually to a value of 10, and then stays there.



Use Of Exponential Averaging

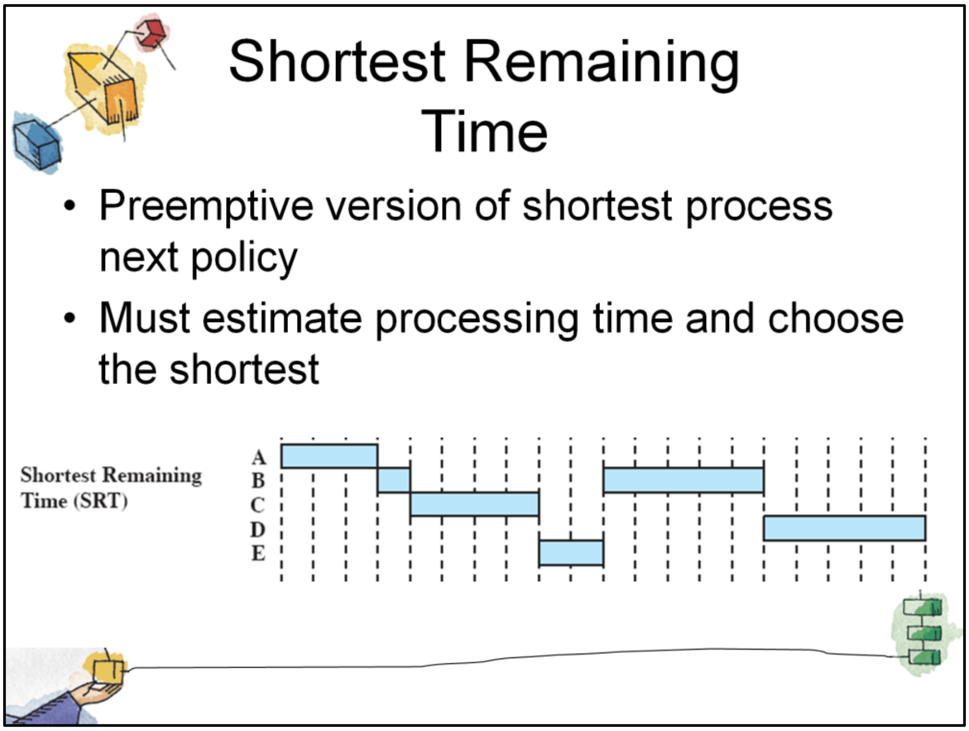
(b) Decreasing function

While here (Figure 9.9b), the observed value begins at 20, declines gradually to 10, and then stays there.

In both cases, we start out with an estimate of $S_1 = 0$.

- This gives greater priority to new processes.

Note that exponential averaging tracks changes in process behavior faster than does simple averaging and that the larger value of α results in a more rapid reaction to the change in the observed value.



Shortest Remaining Time

- Preemptive version of shortest process next policy
- Must estimate processing time and choose the shortest

Shortest Remaining Time (SRT)
A
B
C
D
E

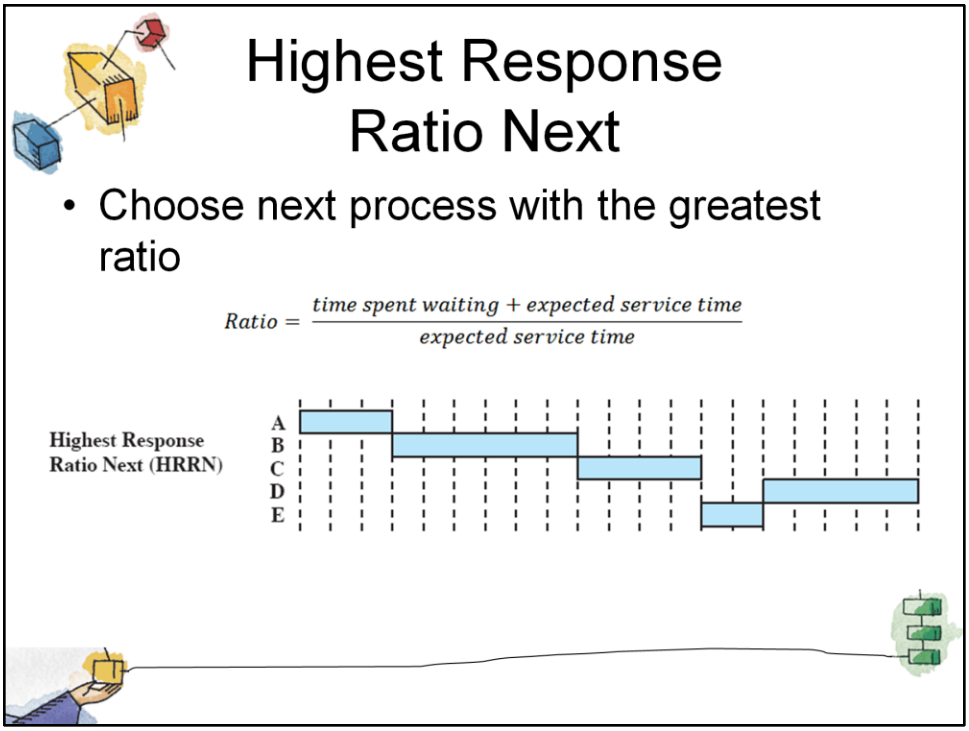
A pre-emptive version of SPN.

In this case, the scheduler always chooses the process that has the shortest expected remaining processing time.

SRT does not have the bias in favor of long processes found in FCFS.

- Unlike round robin, no additional interrupts are generated, reducing overhead.
- On the other hand, elapsed service times must be recorded, contributing to overhead.

SRT should also give superior turnaround time performance to SPN, because a short job is given immediate preference to a running longer job.



Highest Response Ratio Next

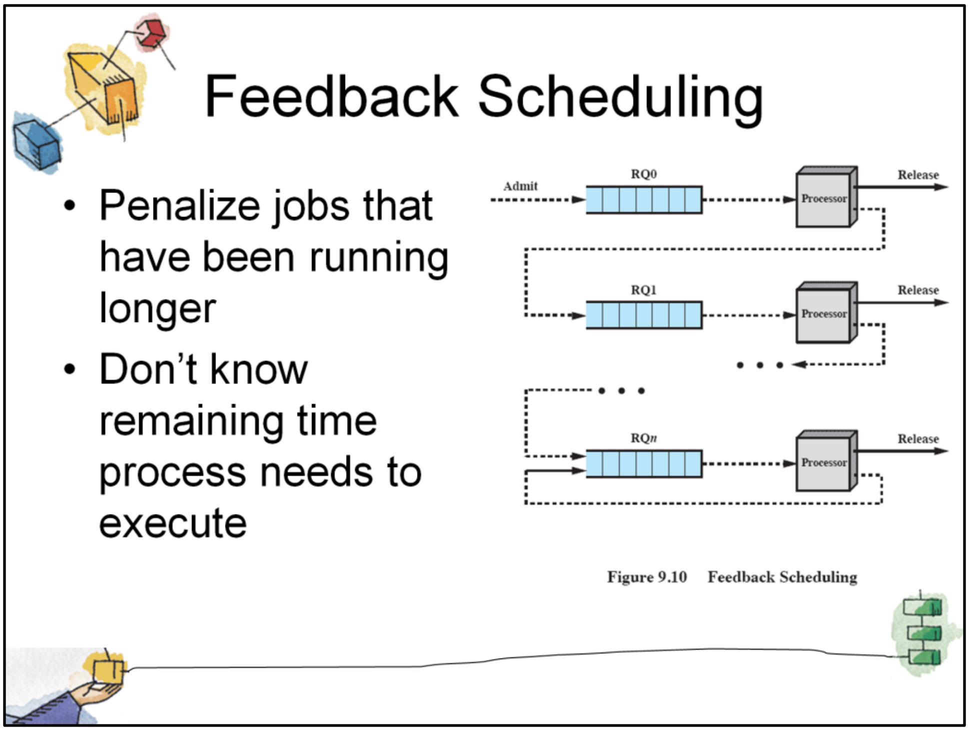
- Choose next process with the greatest ratio

$$Ratio = \frac{time\ spent\ waiting + expected\ service\ time}{expected\ service\ time}$$

Highest Response
Ratio Next (HRRN)
A
B
C
D
E

A smaller denominator yields a larger ratio so that shorter jobs are favored, but aging without service increases the ratio so that a longer process will eventually get past competing shorter jobs.

As with SRT and SPN, the expected service time must be estimated to use highest response ratio next (HRRN).



Feedback Scheduling

- Penalize jobs that have been running longer
- Don’t know remaining time process needs to execute

Figure 9.10 Feedback Scheduling

If we have no indication of the relative length of various processes, then none of SPN, SRT, and HRRN can be used.

Another way of establishing a preference for shorter jobs is to penalize jobs that have been running longer.

- IE if we cannot focus on the time remaining to execute, let’s focus on the time spent in execution so far.

Scheduling is done on a preemptive (at time quantum) basis, and a dynamic priority mechanism is used.

When a process first enters the system, it is placed in RQ0 (refer to Figure 9.4).

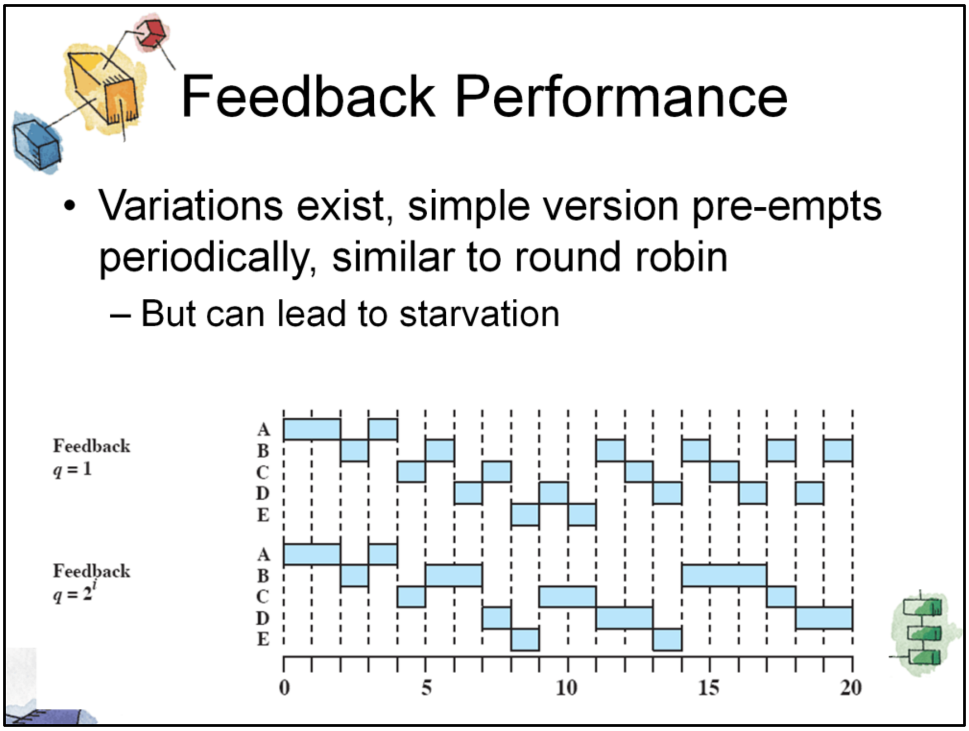
- After its first preemption, when it returns to the Ready state, it is placed in RQ1.
- Each subsequent time that it is preempted, it is demoted to the next lower-priority queue.

A short process will complete quickly, without migrating very far down the hierarchy of ready queues.

- A longer process will gradually drift downward.
- Thus, newer, shorter processes are favored over older, longer processes.

Within each queue, except the lowest-priority queue, a simple FCFS mechanism is used.

- Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution.
- Thus, this queue is treated in round-robin fashion.



Feedback Performance

- Variations exist, simple version pre-empts periodically, similar to round robin
  - But can lead to starvation

A number of variations of this scheme exist.

A simple version is to perform preemption in the same fashion as for round robin: at periodic intervals.

Our example shows this (Figure 9.5 and Table 9.5) for a quantum of one time unit.

Note that in this case, the behavior is similar to round robin with a time quantum of 1.

But the turnaround time of longer processes can stretch out alarmingly.

- it is possible for starvation to occur if new jobs are entering the system frequently.

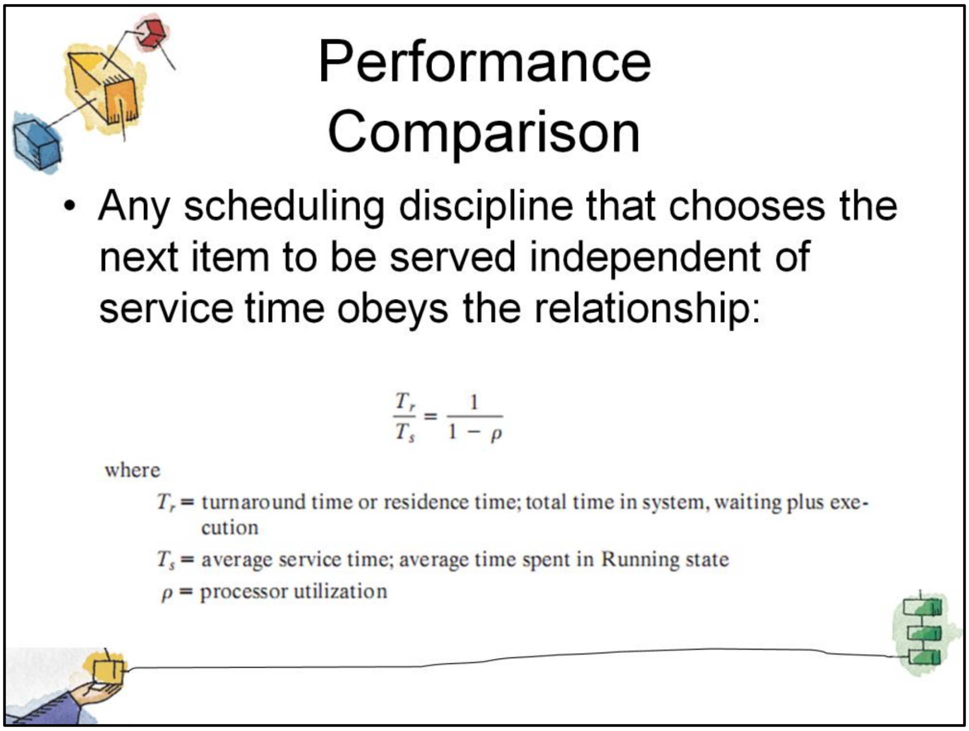
To compensate for this, we can vary the preemption times according to the queue:

- A process scheduled from RQ0 is allowed to execute for one time unit and then is preempted;
- a process scheduled from RQ1 is allowed to execute two time units, and so on.

In general, a process scheduled from RQ*i* is allowed to execute 2*i* time units before preemption.

- Even with the allowance for greater time allocation at lower priority, a longer process may still suffer starvation.

A possible remedy is to promote a process to a higher-priority queue after it spends a certain amount of time waiting for service in its current queue.



Performance Comparison

- Any scheduling discipline that chooses the next item to be served independent of service time obeys the relationship:

$$\frac{T_r}{T_s} = \frac{1}{1 - \rho}$$

where

T_r = turnaround time or residence time; total time in system, waiting plus execution

T_s = average service time; average time spent in Running state

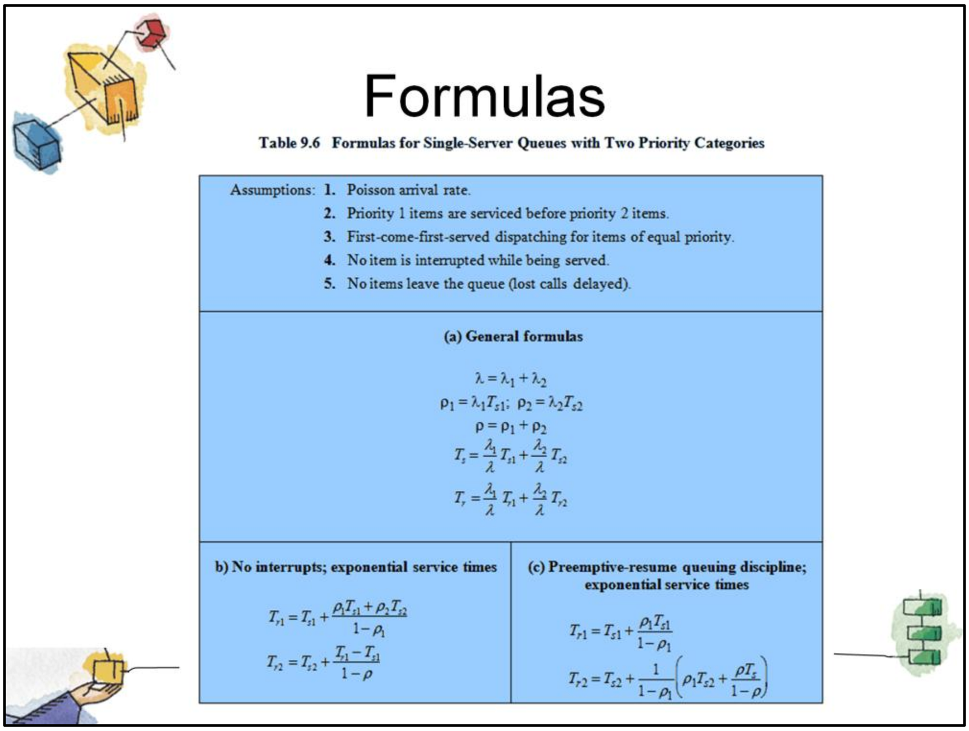
ρ = processor utilization

The performance of various scheduling policies is a critical factor in the choice of a scheduling policy.

But, it is impossible to make definitive comparisons because relative performance will depend on a variety of factors,

- But it is possible to make some general conclusions

For any scheduling discipline that chooses the next item to be served independent of service time obeys the above relationship



Formulas

Table 9.6 Formulas for Single-Server Queues with Two Priority Categories

Assumptions:
1. Poisson arrival rate.
2. Priority 1 items are serviced before priority 2 items.
3. First-come-first-served dispatching for items of equal priority.
4. No item is interrupted while being served.
5. No items leave the queue (lost calls delayed).

(a) General formulas

$$\lambda = \lambda_1 + \lambda_2$$
$$\rho_1 = \lambda_1 T_{s1}; \ \rho_2 = \lambda_2 T_{s2}$$
$$\rho = \rho_1 + \rho_2$$
$$T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$$
$$T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$$

b) No interrupts; exponential service times	(c) Preemptive-resume queuing discipline; exponential service times
$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 - \rho_1}$ $T_{r2} = T_{s2} + \frac{T_{r1} - T_{s1}}{1 - \rho}$	$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1}}{1 - \rho_1}$ $T_{r2} = T_{s2} + \frac{1}{1 - \rho_1}\left(\rho_1 T_{s2} + \frac{\rho T_s}{1 - \rho}\right)$

If scheduling is done on the basis of priority and if processes are assigned to a priority class on the basis of service time, then differences do emerge.

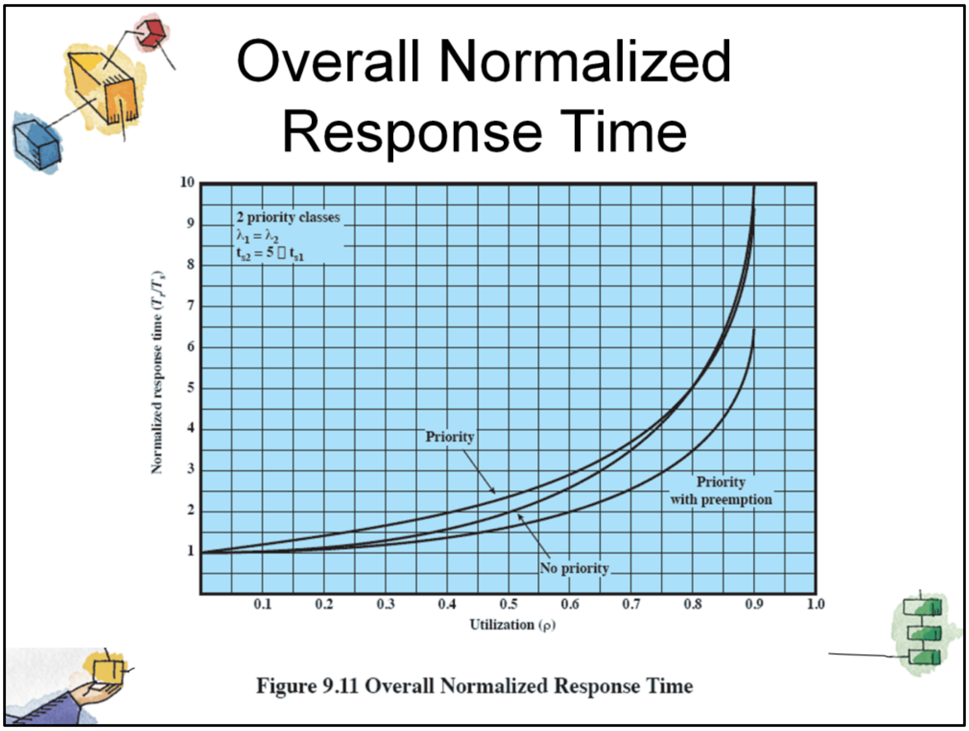
This slide shows the formulas that result when we assume two priority classes, with different service times for each class.

λ refers to the arrival rate.

These results can be generalized to any number of priority classes.

Note: the formulas differ for nonpreemptive versus preemptive scheduling.

In the preemptive , it is assumed that a lower-priority process is immediately interrupted when a higher-priority process becomes ready.



Overall Normalized Response Time

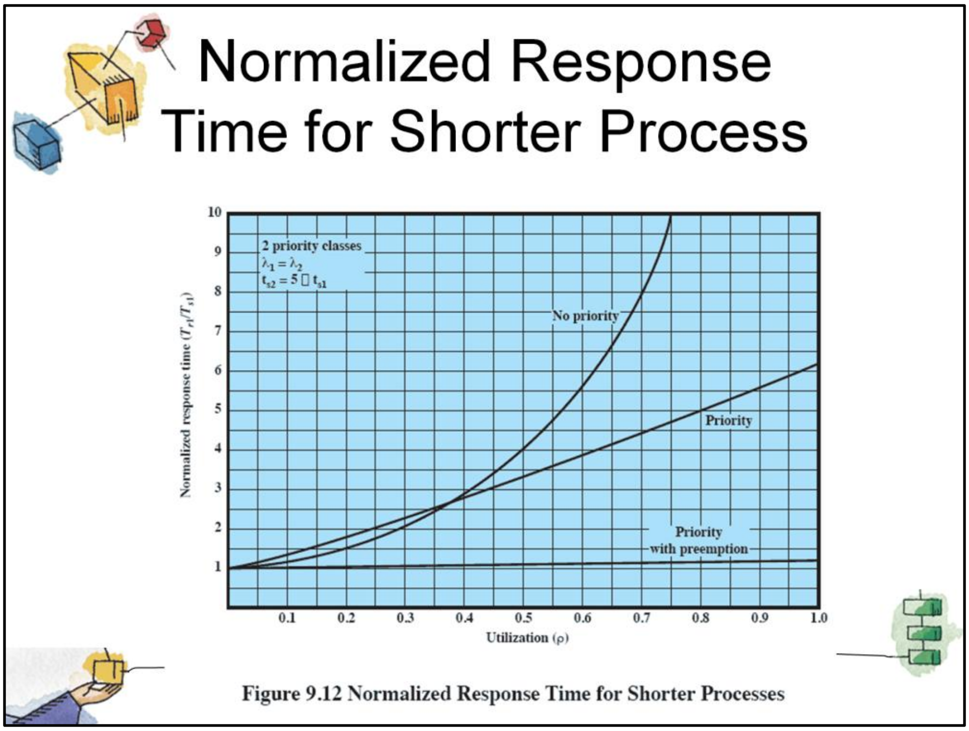
Figure 9.11 Overall Normalized Response Time

This figure shows the overall result.

By giving preference to shorter jobs, the average normalized turnaround time is improved at higher levels of utilization.

The improvement is greatest with the use of preemption.

Notice, however, that overall performance is not much affected.



Normalized Response Time for Shorter Process

Figure 9.12 Normalized Response Time for Shorter Processes

Significant differences emerge when we consider the two priority classes separately.

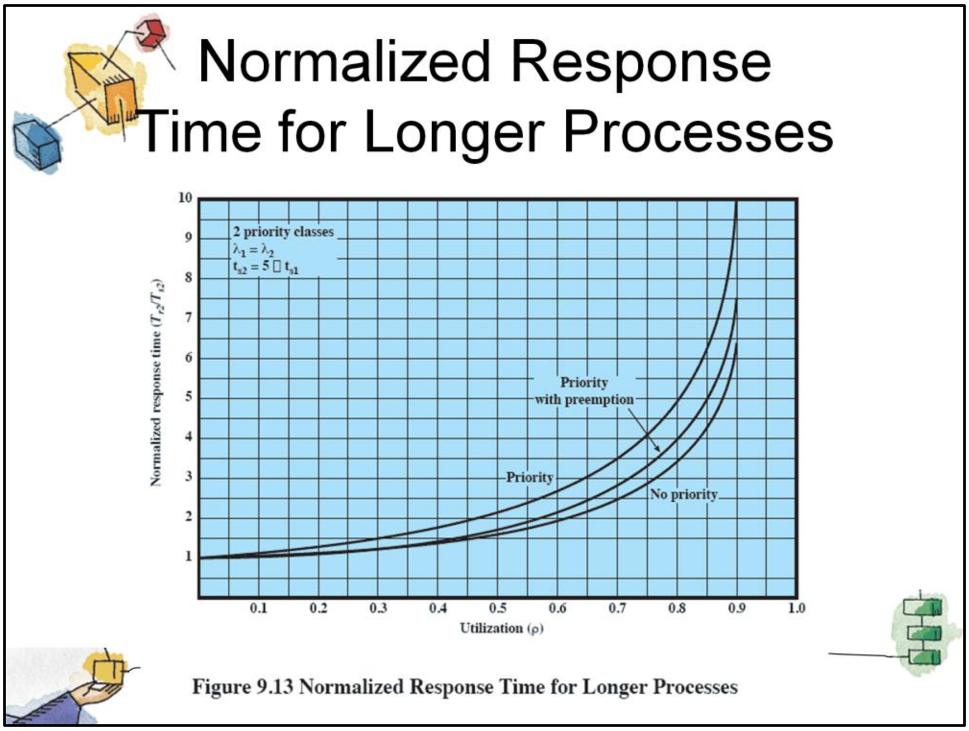
This figure shows the results for the higher-priority, shorter processes.

For comparison, the upper line on the graph assumes that priorities are not used but that we are simply looking at the relative performance of half of all processes that have the shorter processing time.

- The other two lines assume that these processes are assigned a higher priority.

When the system is run using priority scheduling without preemption, the improvements are significant.

- They are even more significant when preemption is used.

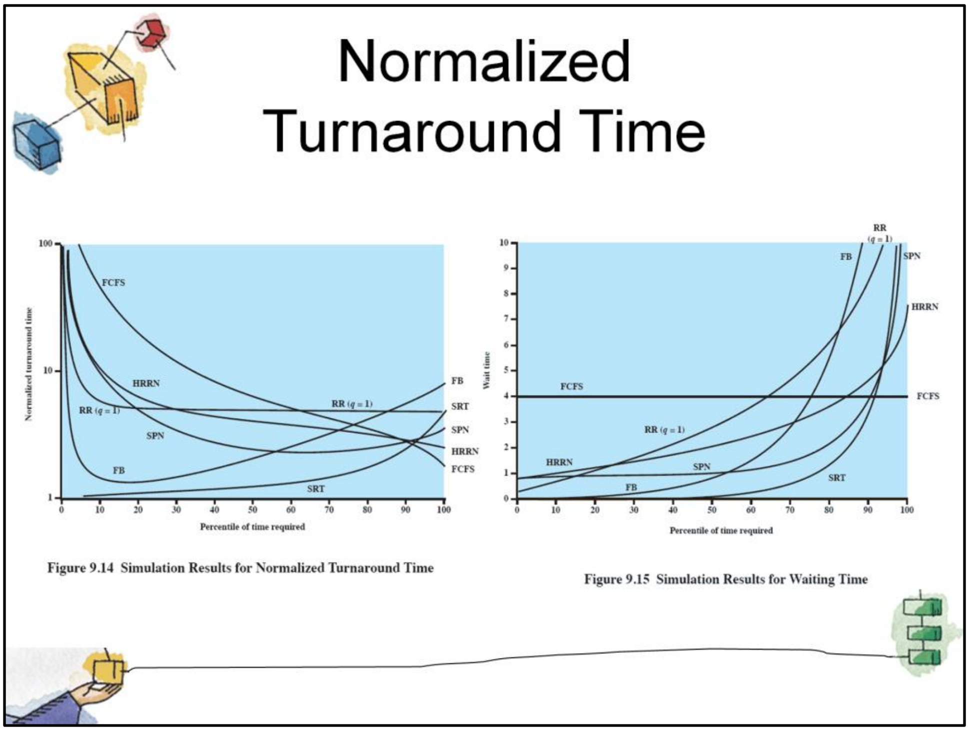


Normalized Response Time for Longer Processes

Figure 9.13 Normalized Response Time for Longer Processes

This shows the same analysis for the lower-priority, longer processes.

As expected, such processes suffer a performance degradation under priority scheduling.



Normalized Turnaround Time

Figure 9.14 Simulation Results for Normalized Turnaround Time

Figure 9.15 Simulation Results for Waiting Time

Some of the difficulties of analytic modelling are over-come by using discrete-event simulation, which allows a wide range of policies to be
modelled.

The disadvantage of simulation is that the results for a given “run” only apply to that particular collection of processes under that particular set of assumptions. Nevertheless, useful insights can be gained.

This figure shows the result of one study. Figure 9.14 shows the normalized turnaround time, and Figure 9.15 shows the average waiting time.

Looking at the turnaround time, we can see that the performance of FCFS is very unfavorable,

- with one-third of the processes having a normalized turnaround time greater than 10 times the service time (these were the shortest)
- On the other hand, the absolute waiting time is uniform, as is to be expected because scheduling is independent of service time.

The figures show round robin using a quantum of one time unit.

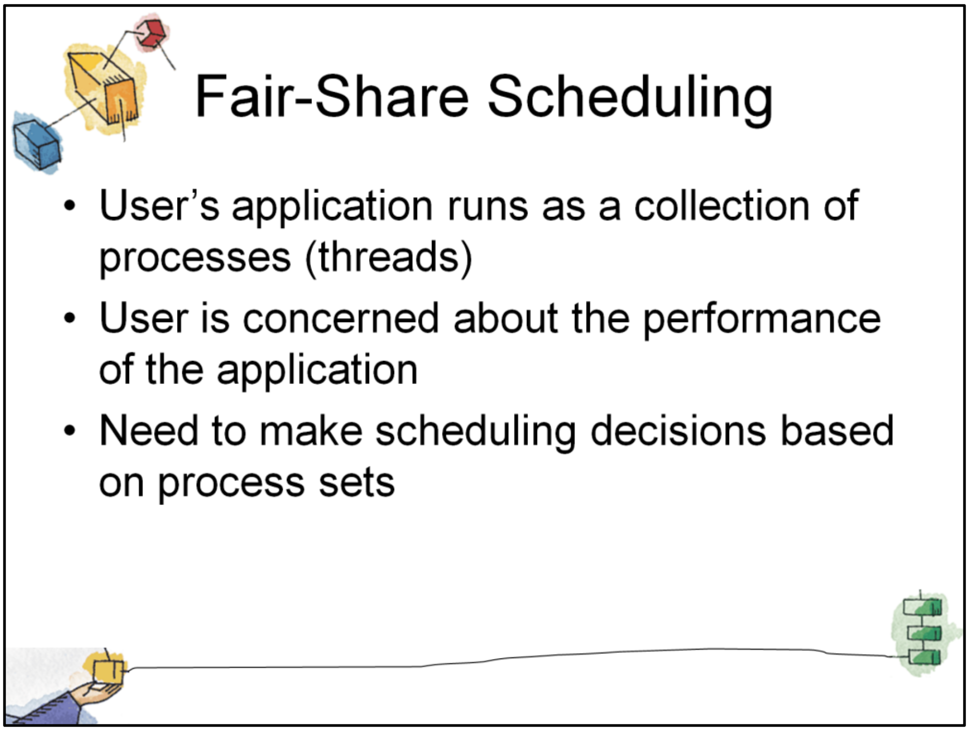
- Round robin yields a normalized turnaround time of about 5 for all processes, treating all fairly, except for the shortest processes, which execute in less than one quantum,
- Shortest process next performs better than round robin, except for the shortest processes.
- Shortest remaining time, the preemptive version of SPN, performs better than SPN except for the longest 7% of all processes.

We have seen that, among nonpreemptive policies, FCFS favors long processes and SPN favors short ones.

Highest response ratio next is intended to be a compromise between these two effects, and this is indeed confirmed in the figures.

Finally, the figure shows feedback scheduling with fixed, uniform quanta in each priority queue.

- As expected, FB performs quite well for short processes.



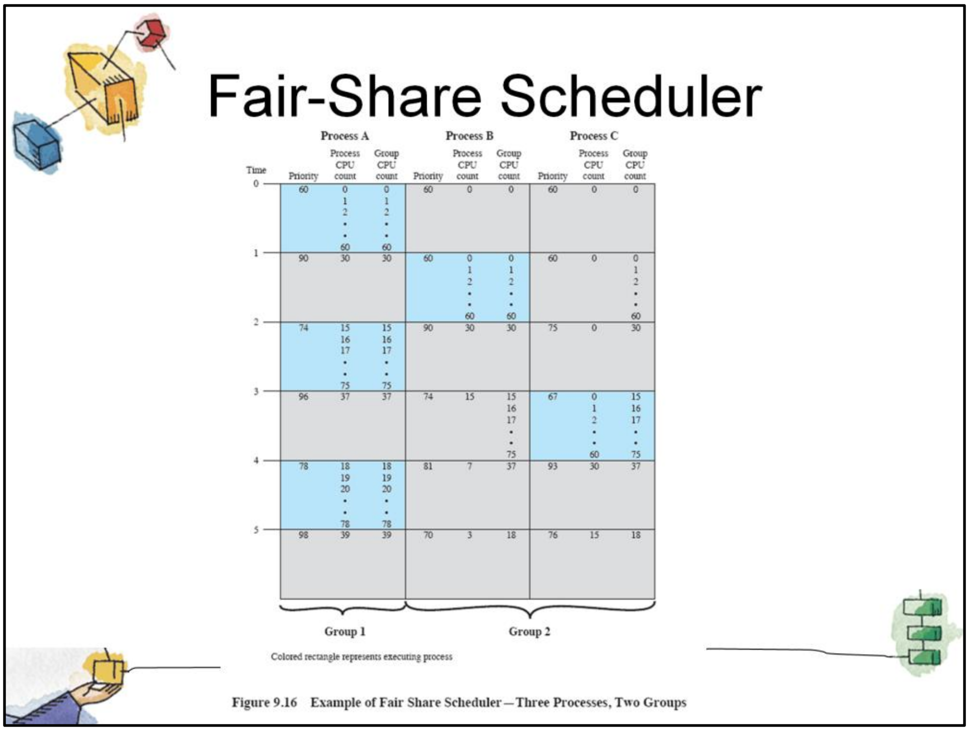
Fair-Share Scheduling

- 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

From the user's point of view, the concern is not how a particular process performs but rather how his or her set of processes, which constitute a single application, performs.

It would be attractive to make scheduling decisions on the basis of these process sets.

- This approach is generally known as fair-share scheduling.
- The concept can be extended to groups of users, even if each user is represented by a single process.



Fair-Share Scheduler

Time	Process A Priority	Process A Process CPU count	Process A Group CPU count	Process B Priority	Process B Process CPU count	Process B Group CPU count	Process C Priority	Process C Process CPU count	Process C Group CPU count
0	60	0 1 2 • • 60	0 1 2 • • 60	60	0	0	60	0	0
1	90	30	30	60	0 1 2 • • 60	0 1 2 • • 60	60	0	0 1 2 • • 60
2	74	15 16 17 • • 75	15 16 17 • • 75	90	30	30	75	0	30
3	96	37	37	74	15	15 16 17 • • 75	67	0 1 2 • • 60	15 16 17 • • 75
4	78	18 19 20 • • 78	18 19 20 • • 78	81	7	37	93	30	37
5	98	39	39	70	3	18	76	15	18

Group 1 (Process A); Group 2 (Process B, Process C)

Colored rectangle represents executing process

Figure 9.16 Example of Fair Share Scheduler—Three Processes, Two Groups

This figure is an example in which process A is in one group and process B and process C are in a second group,

- with each group having a weighting of 0.5.

Assume that all processes are processor bound and are usually ready to run.

All processes have a base priority of 60.

Processor utilization is measured as follows:

- The processor is interrupted 60 times per second;
- during each interrupt, the processor usage field of the currently running process is incremented, as is the corresponding group processor field.
- Once per second, priorities are recalculated.

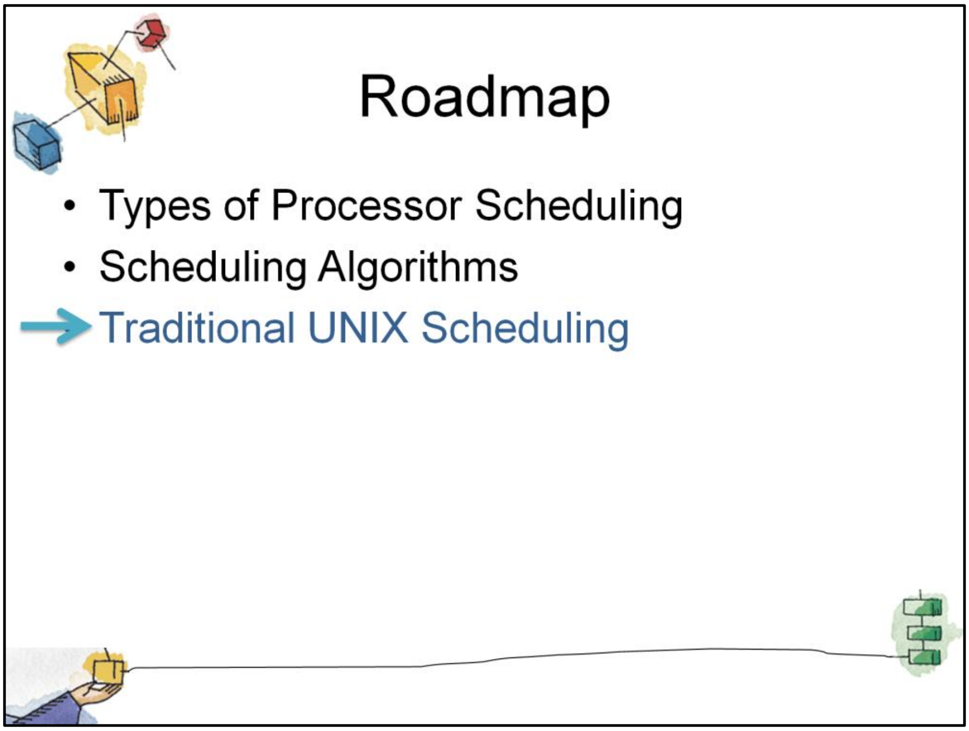
In the figure, process A is scheduled first.

- At the end of one second, it is pre-empted.
- Processes B and C now have the higher priority, and process B is scheduled.

At the end of the second time unit, process A has the highest priority.

Note that the pattern repeats:

- the kernel schedules the processes in order:A, B,A, C,A, B, and so on.
- Thus, 50% of the processor is allocated to process A, which constitutes one group, and 50% to processes B and C, which constitute another group.



Roadmap

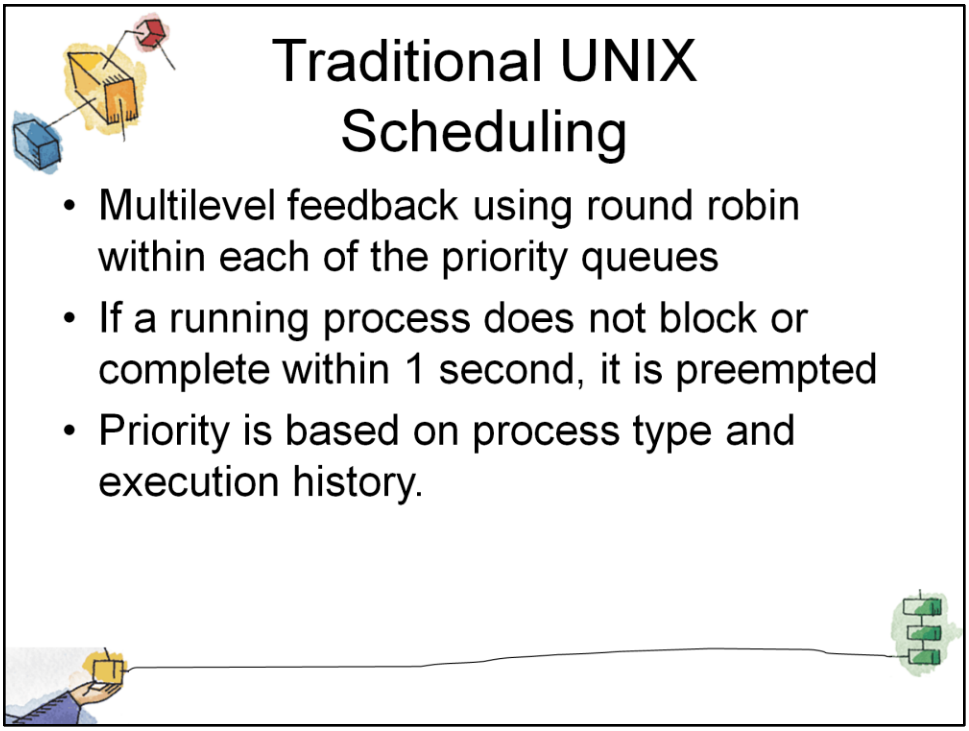
- Types of Processor Scheduling
- Scheduling Algorithms
- → Traditional UNIX Scheduling

In this section we examine traditional UNIX scheduling, which is used in both SVR3 and 4.3 BSD UNIX.

- Primarily targeted at the time-sharing interactive environment.

The scheduling algorithm is designed to provide good response time for interactive users while ensuring that low-priority background jobs do not starve.

- Although this algorithm has been replaced in modern UNIX systems, the approach because is representative of practical time-sharing scheduling algorithms.



Traditional UNIX Scheduling

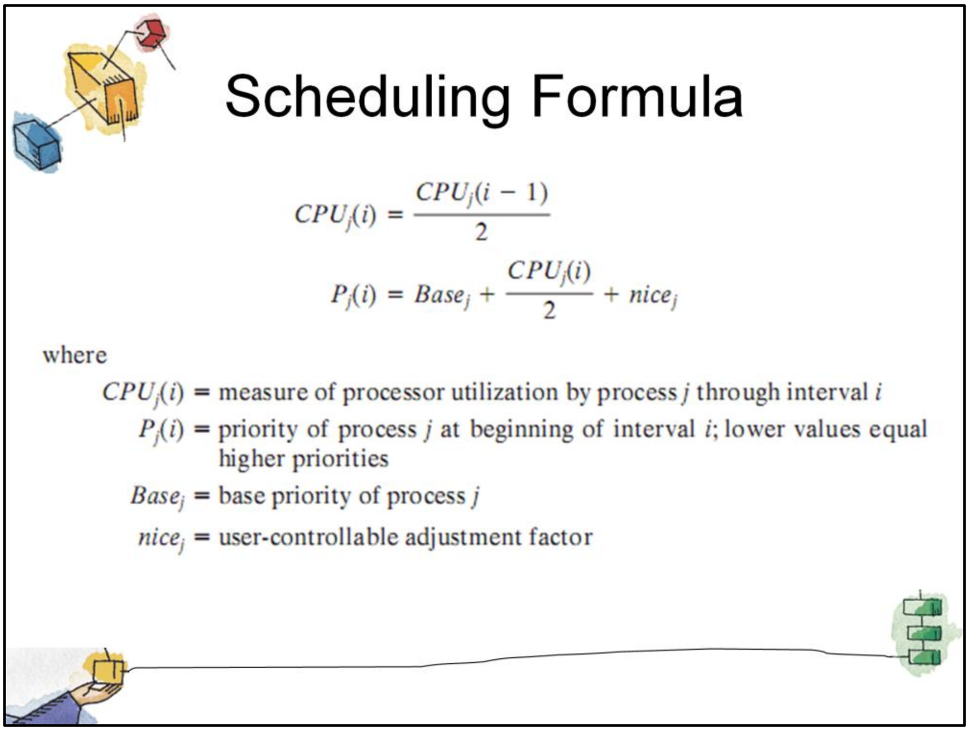
- Multilevel feedback using round robin within each of the priority queues
- If a running process does not block or complete within 1 second, it is preempted
- Priority is based on process type and execution history.

The traditional UNIX scheduler employs multilevel feedback using round robin within each of the priority queues.

The system makes use of 1-second preemption.

- ie if a running process does not block or complete within 1 second, it is preempted.

Priority is based on process type and execution history.



Scheduling Formula

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

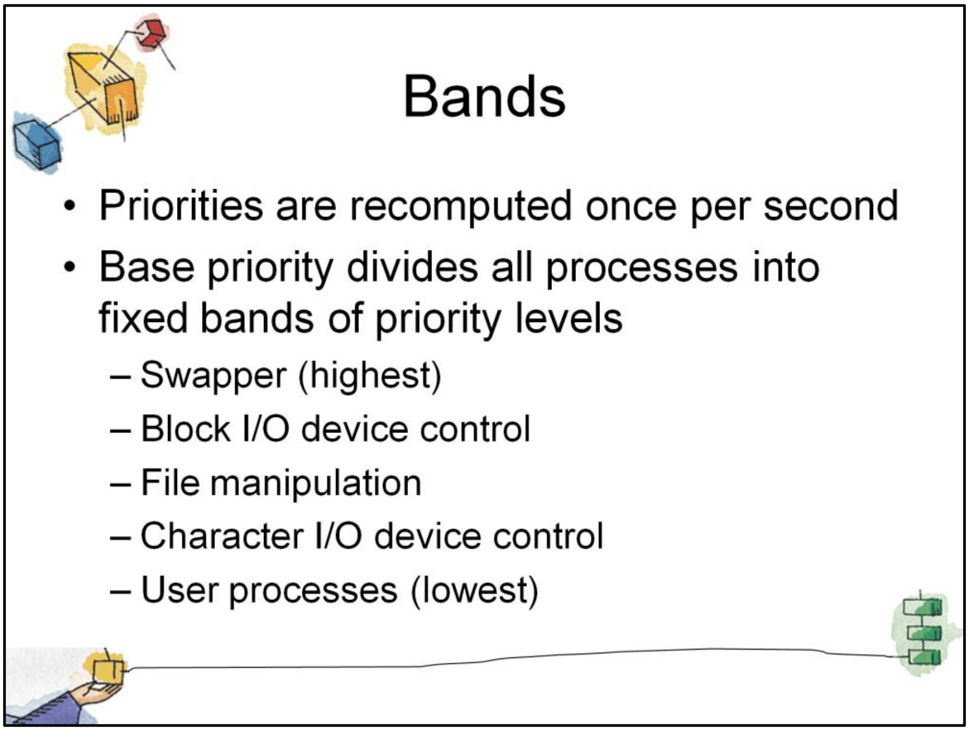
where

$CPU_j(i)$ = measure of processor utilization by process j through interval i

$P_j(i)$ = priority of process j at beginning of interval i; lower values equal higher priorities

$Base_j$ = base priority of process j

$nice_j$ = user-controllable adjustment factor



Bands

- Priorities are recomputed once per second
- Base priority divides all processes into fixed bands of priority levels
  - Swapper (highest)
  - Block I/O device control
  - File manipulation
  - Character I/O device control
  - User processes (lowest)

The priority of each process is recomputed once per second, at which time a new scheduling decision is made.

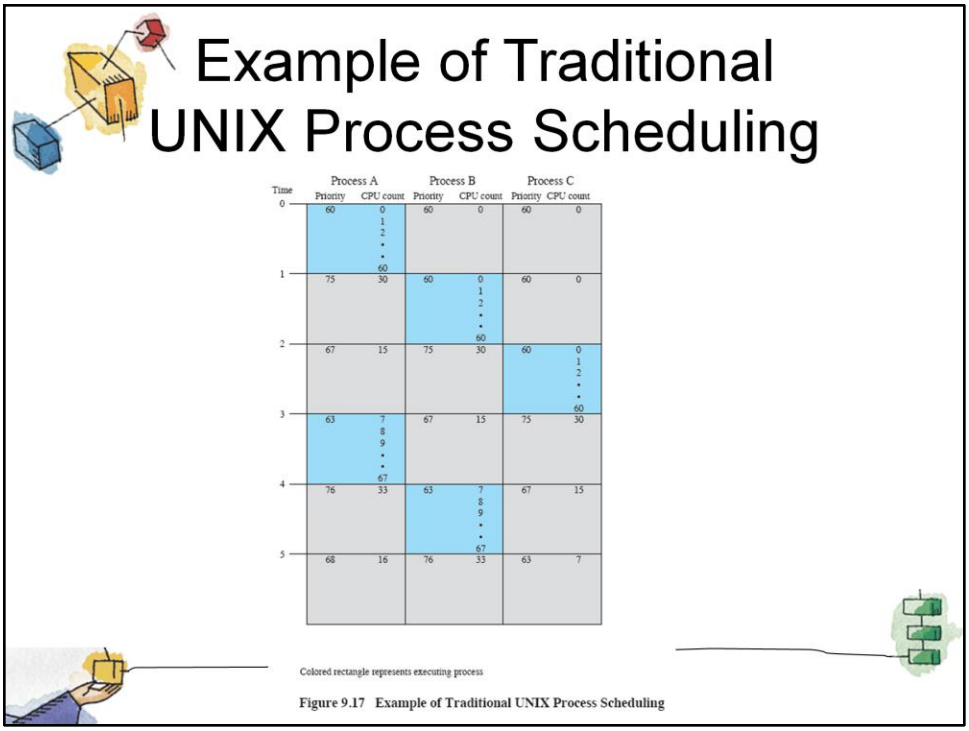
- The base priority is to divides all processes into fixed bands of priority levels.

The CPU and nice components are restricted to prevent a process from migrating out of its assigned band (assigned by the base priority level).

These bands are used to optimize access to block devices (e.g., disk) and to allow the operating system to respond quickly to system calls.

In decreasing order of priority, the bands are

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



Example of Traditional UNIX Process Scheduling

	Process A		Process B		Process C	
Time	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

Colored rectangle represents executing process

Figure 9.17 Example of Traditional UNIX Process Scheduling

This figure shows an example of process scheduling

Processes A, B, and C are created at the same time with base priorities of 60 (ignoring the nice value).

The clock interrupts the system 60 times per second and increments a counter for the running process.

The example assumes that none of the processes block themselves and that no other processes are ready to run.