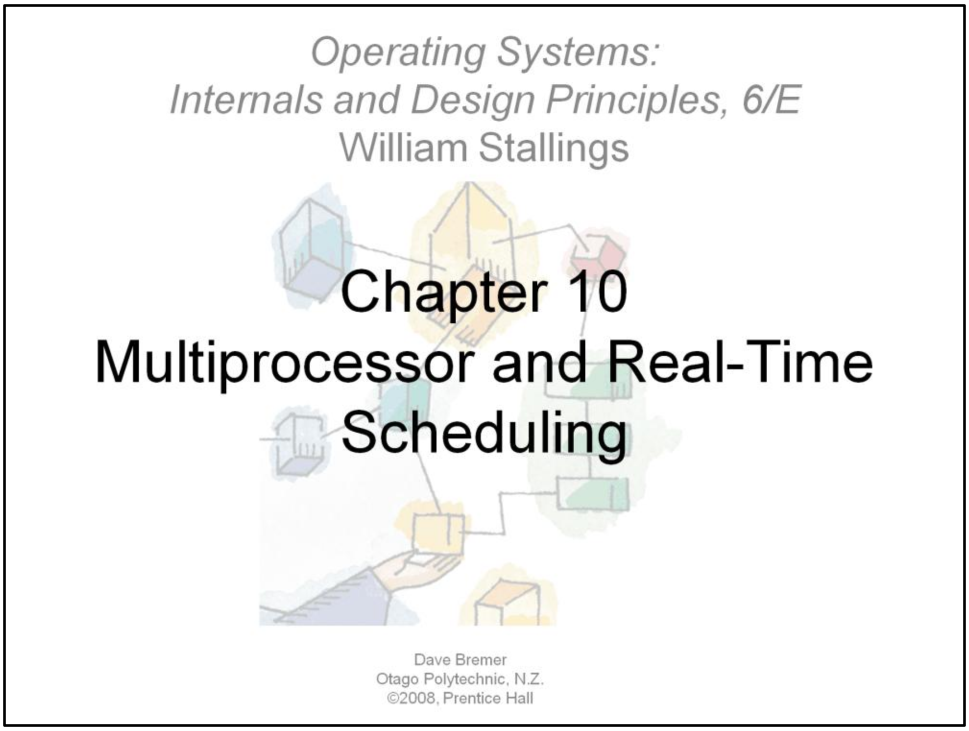
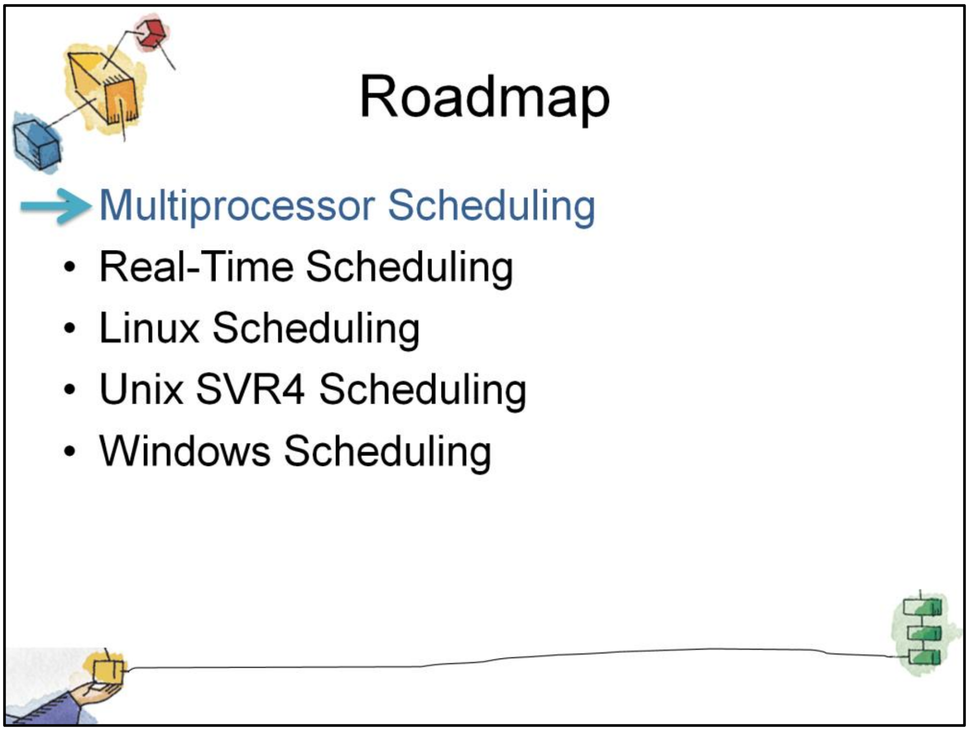


*Operating Systems:*
*Internals and Design Principles, 6/E*
William Stallings

# Chapter 10
# Multiprocessor and Real-Time 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

- Multiprocessor Scheduling
- Real-Time Scheduling
- Linux Scheduling
- Unix SVR4 Scheduling
- Windows Scheduling

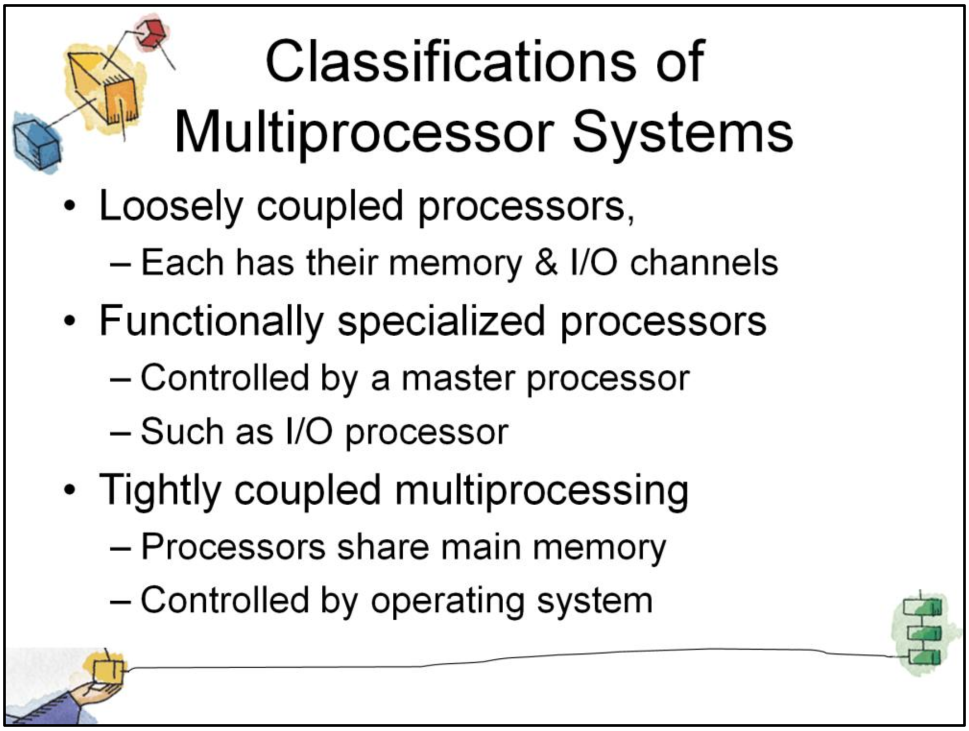
This chapter continues our survey of process and thread scheduling.

Firstly there is an examination of issues raised by the availability of more than one processor.

- We look at the scheduling of processes on a multiprocessor system.
- Then the somewhat different design considerations for multiprocessor thread scheduling are examined.

The second section of this chapter covers real-time scheduling.

- Beginning with a discussion of the characteristics of real-time processes and then looks at the nature of the scheduling process.
- Two approaches to real-time scheduling are examined:
  - deadline scheduling and
  - rate monotonic scheduling.

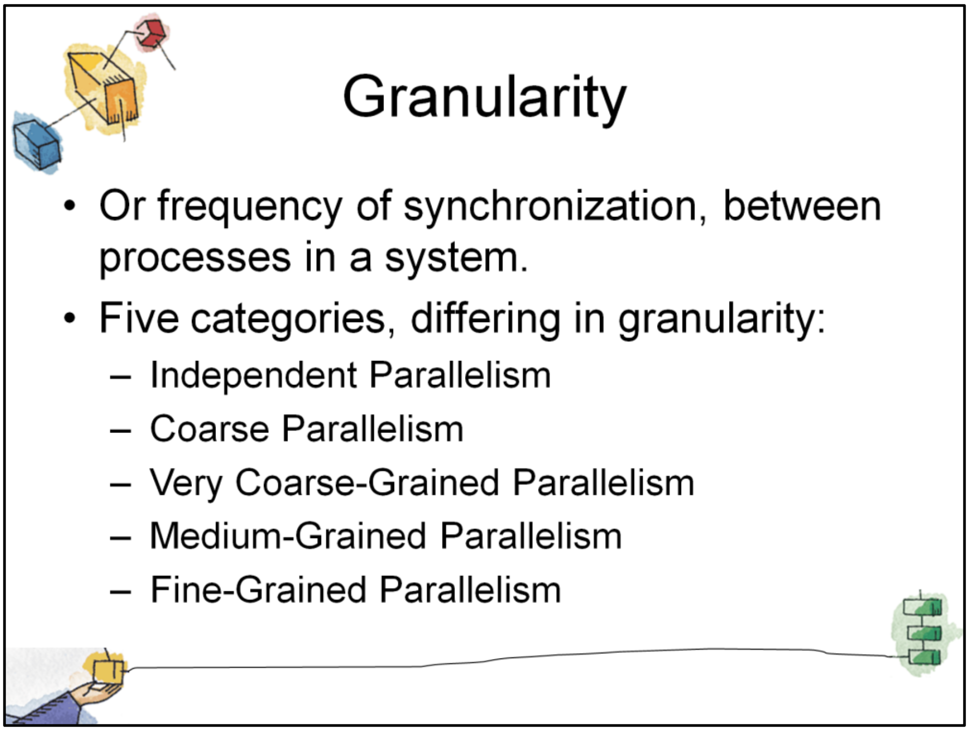


# Classifications of Multiprocessor Systems

- Loosely coupled processors,
  - Each has their memory & I/O channels
- Functionally specialized processors
  - Controlled by a master processor
  - Such as I/O processor
- Tightly coupled multiprocessing
  - Processors share main memory
  - Controlled by operating system

We can classify multiprocessor systems as follows:

- **Loosely coupled** or distributed multiprocessor, or cluster:
  - Consists of a collection of relatively autonomous systems, each processor having its own main memory and I/O channels.
- **Functionally specialized** processors:
  - An example is an I/O processor.
  - In this case, there is a master, general-purpose processor; specialized processors are controlled by the master processor and provide services to it.
- **Tightly coupled multiprocessing**:
  - Consists of a set of processors that share a common main memory and are under the integrated control of an operating system.

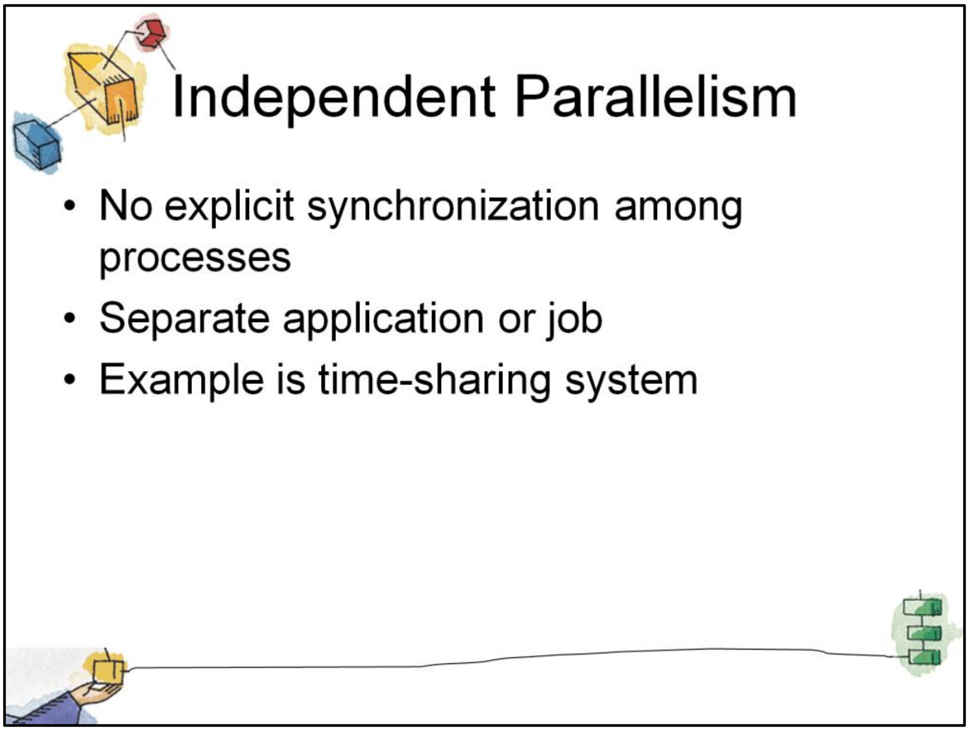


# Granularity

- Or frequency of synchronization, between processes in a system.
- Five categories, differing in granularity:
  - Independent Parallelism
  - Coarse Parallelism
  - Very Coarse-Grained Parallelism
  - Medium-Grained Parallelism
  - Fine-Grained Parallelism

A good way of characterizing multiprocessors and placing them in context with other architectures is to consider the synchronization granularity, or frequency of synchronization, between processes in a system.

We can distinguish five categories of parallelism that differ in the degree of granularity.



# Independent Parallelism

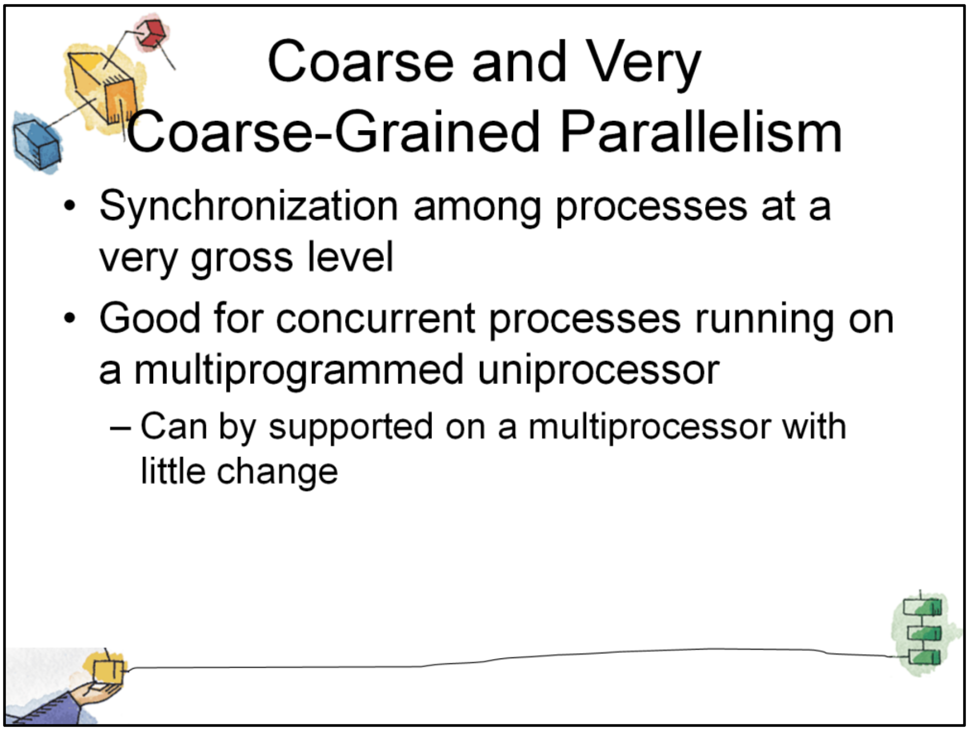
- No explicit synchronization among processes
- Separate application or job
- Example is time-sharing system

With independent parallelism, there is no explicit synchronization among processes.

Each represents a separate, independent application or job.

A typical use of this type of parallelism is in a time-sharing system.

- Each user is performing a particular application, such as word processing or using a spreadsheet.
- The multiprocessor provides the same service as a multiprogrammed uniprocessor.
- Because more than one processor is available, average response time to the users will be less.

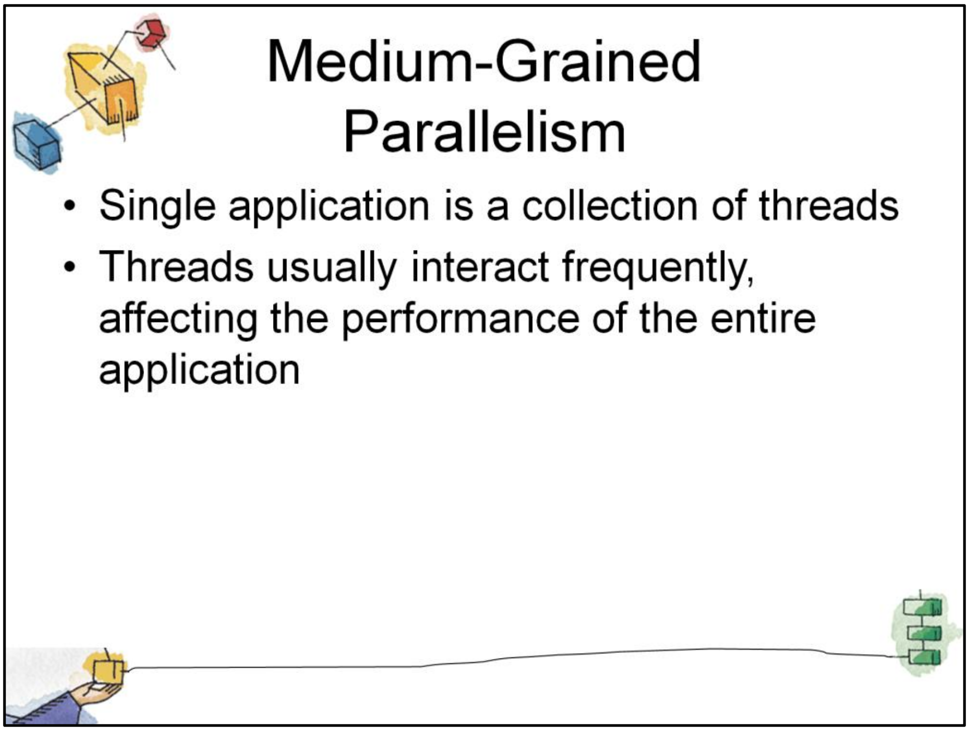


## Coarse and Very Coarse-Grained Parallelism

- Synchronization among processes at a very gross level
- Good for concurrent processes running on a multiprogrammed uniprocessor
  - Can by supported on a multiprocessor with little change

With coarse and very-coarse grained parallelism, there is synchronization among processes, but at a very gross level.

This kind of situation is easily handled as a set of concurrent processes running on a multiprogrammed uniprocessor and can be supported on a multiprocessor with little or no change to user software.



# Medium-Grained Parallelism

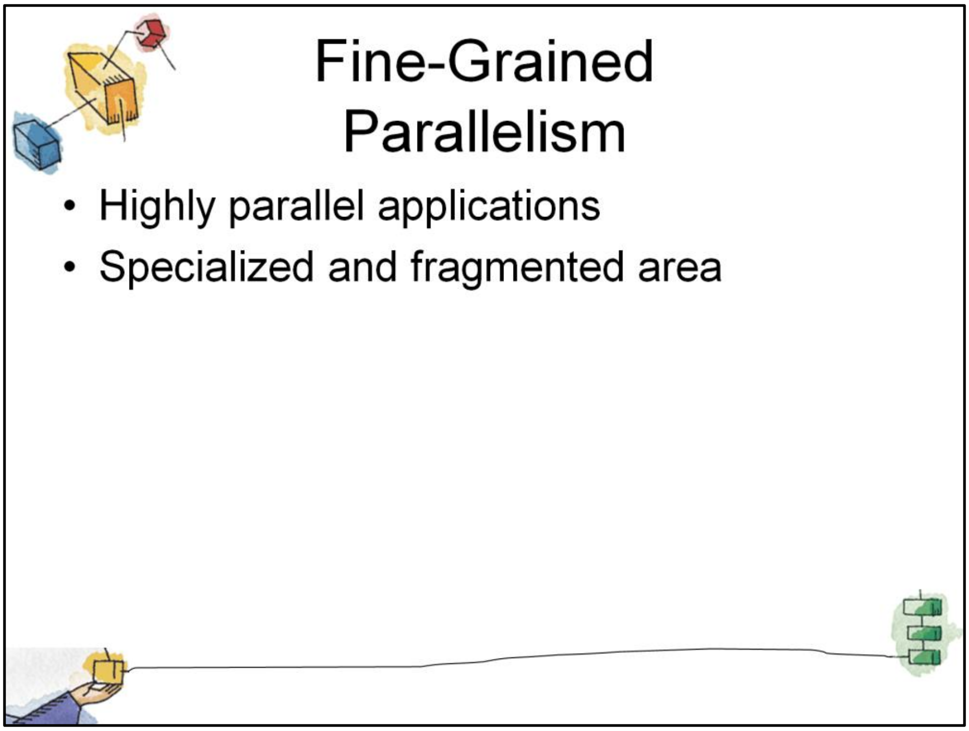
- Single application is a collection of threads
- Threads usually interact frequently, affecting the performance of the entire application

From chapt 4, we saw that a single application can be effectively implemented as a collection of threads within a single process.

In this case, the programmer must explicitly specify the potential parallelism of an application.

- Typically, there will need to be rather a high degree of coordination and interaction among the threads of an application, leading to a medium-grain level of synchronization.

Because the various threads of an application interact so frequently, scheduling decisions concerning one thread may affect the performance of the entire application.

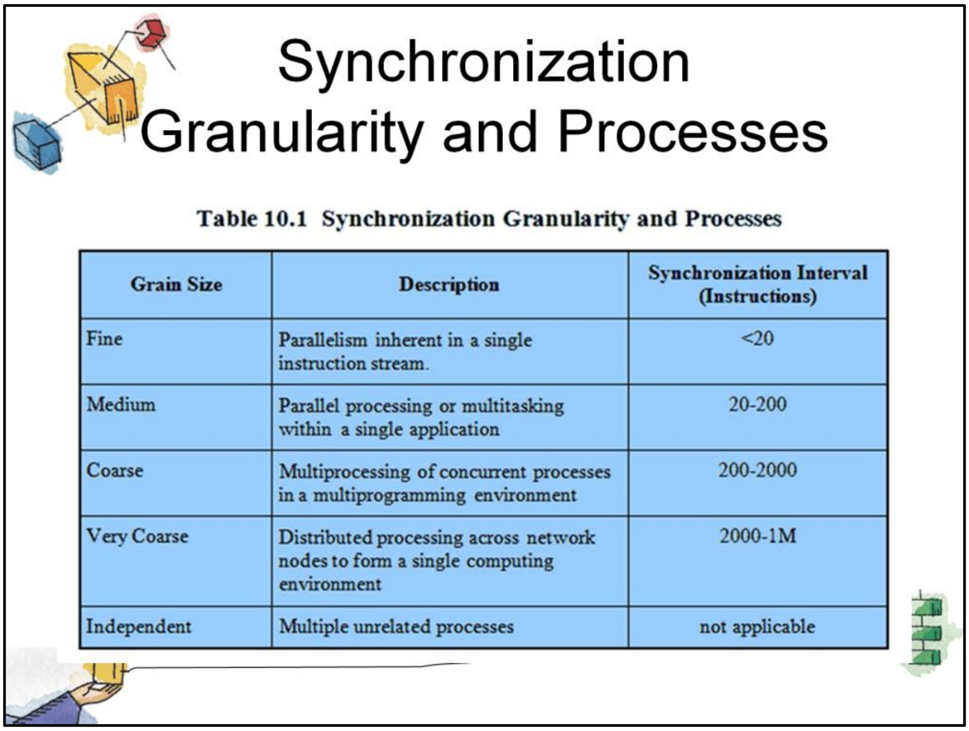


# Fine-Grained Parallelism

- Highly parallel applications
- Specialized and fragmented area

Fine-grained parallelism represents a much more complex use of parallelism than is found in the use of threads.

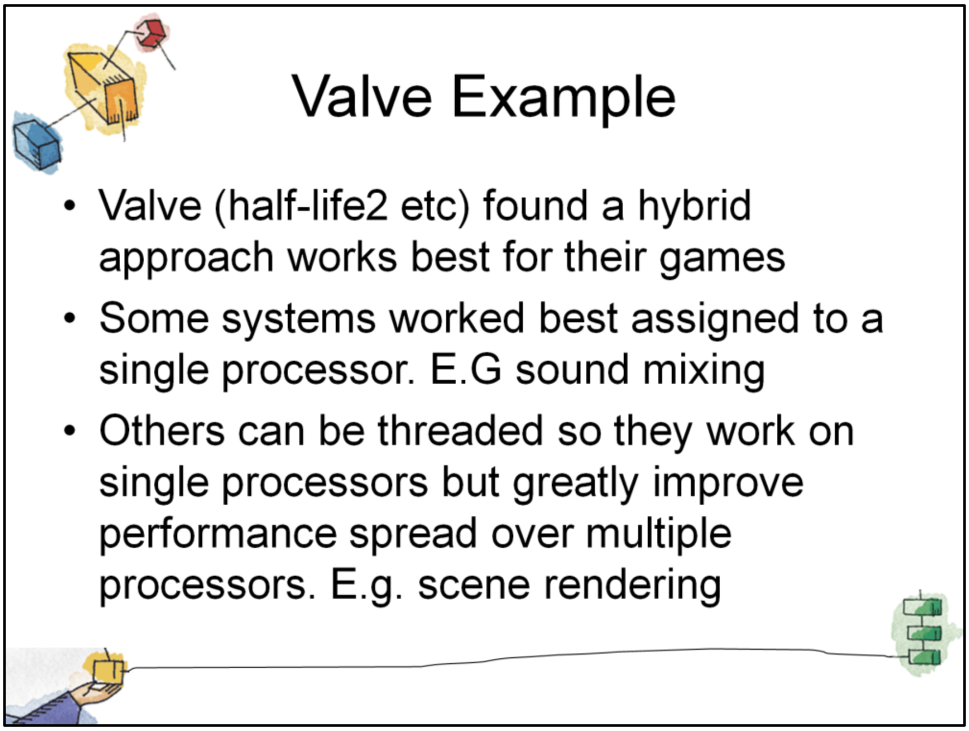
Although much work has been done on highly parallel applications, this is so far a specialized and fragmented area, with many different approaches.



# Synchronization Granularity and Processes

**Table 10.1 Synchronization Granularity and Processes**

| Grain Size | Description | Synchronization Interval (Instructions) |
|---|---|---|
| Fine | Parallelism inherent in a single instruction stream. | <20 |
| Medium | Parallel processing or multitasking within a single application | 20-200 |
| Coarse | Multiprocessing of concurrent processes in a multiprogramming environment | 200-2000 |
| Very Coarse | Distributed processing across network nodes to form a single computing environment | 2000-1M |
| Independent | Multiple unrelated processes | not applicable |



## Valve Example

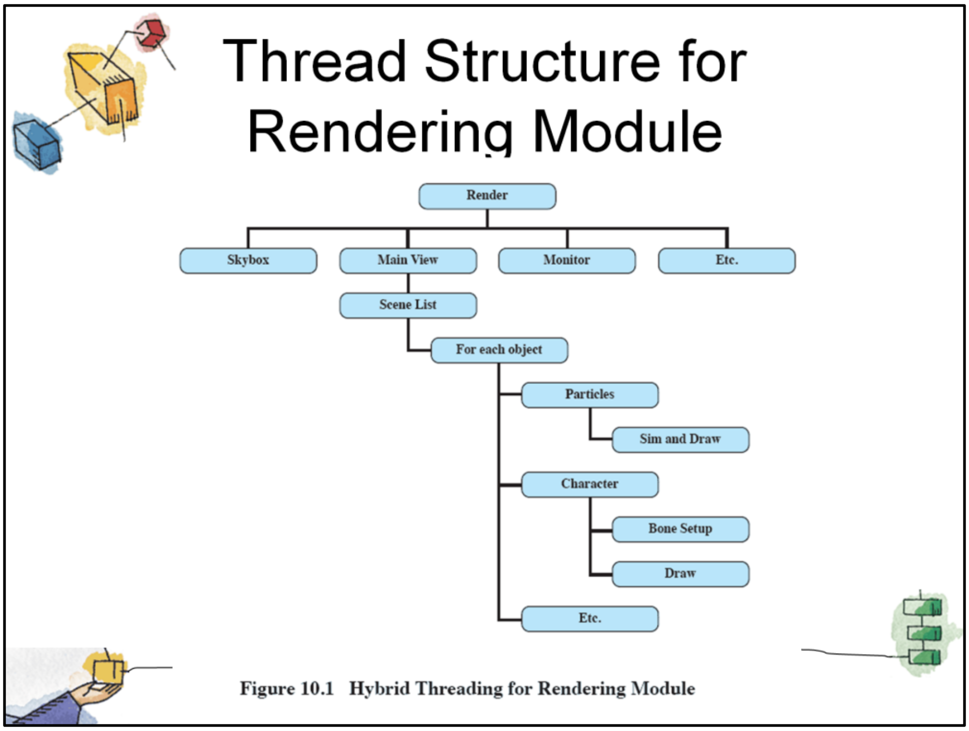
- Valve (half-life2 etc) found a hybrid approach works best for their games
- Some systems worked best assigned to a single processor. E.G sound mixing
- Others can be threaded so they work on single processors but greatly improve performance spread over multiple processors. E.g. scene rendering

Valve identified systems that operate very effectively being permanently assigned to a single processor.

- An example is sound mixing, which has little user interaction,
- is not constrained by the frame configuration of windows, and
- works on its own set of data.

Other modules can be organized into a number of threads so that the module can execute on a single processor but achieve greater performance as it is spread out over more and more processors.

- e.g. scene rendering,



# Thread Structure for Rendering Module

**Figure 10.1** Hybrid Threading for Rendering Module

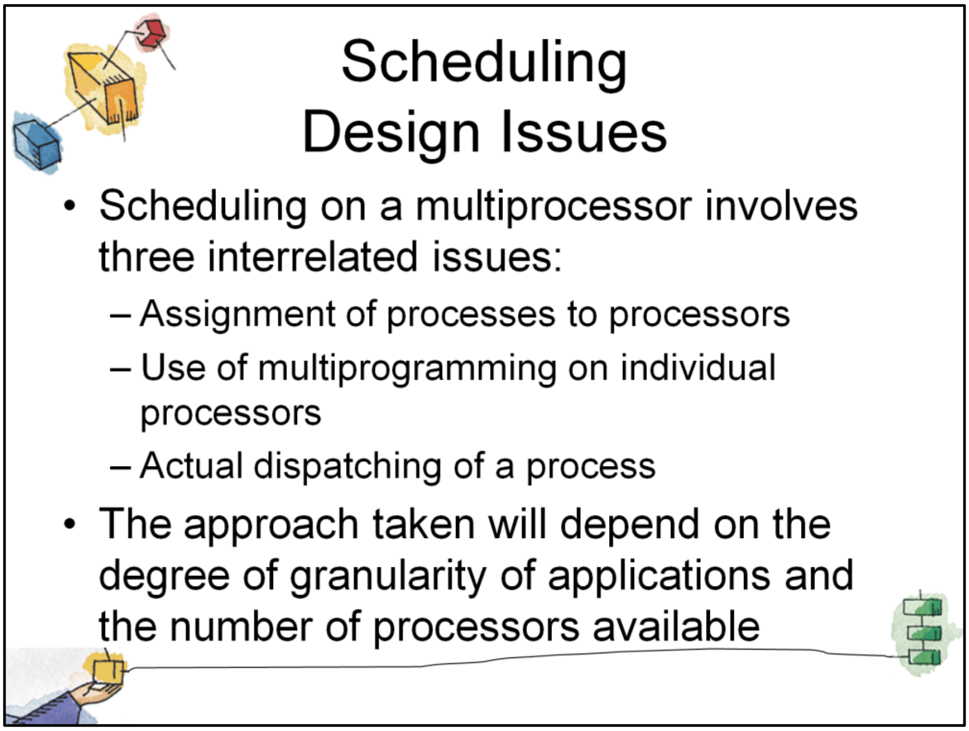
This figure illustrates the thread structure for the rendering module.

In this hierarchical structure, higher-level threads spawn lower-level threads as needed.

The rendering module relies on a critical part of the Source engine, the world list, a database representation of the visual elements in the game’s world.

The first task is to determine what are the areas of the world that need to be rendered.

- next is to determine what objects are in the scene as viewed from multiple angles.
- Then it has to work out the rendering of each object from multiple points of view – **VERY processor intensive**



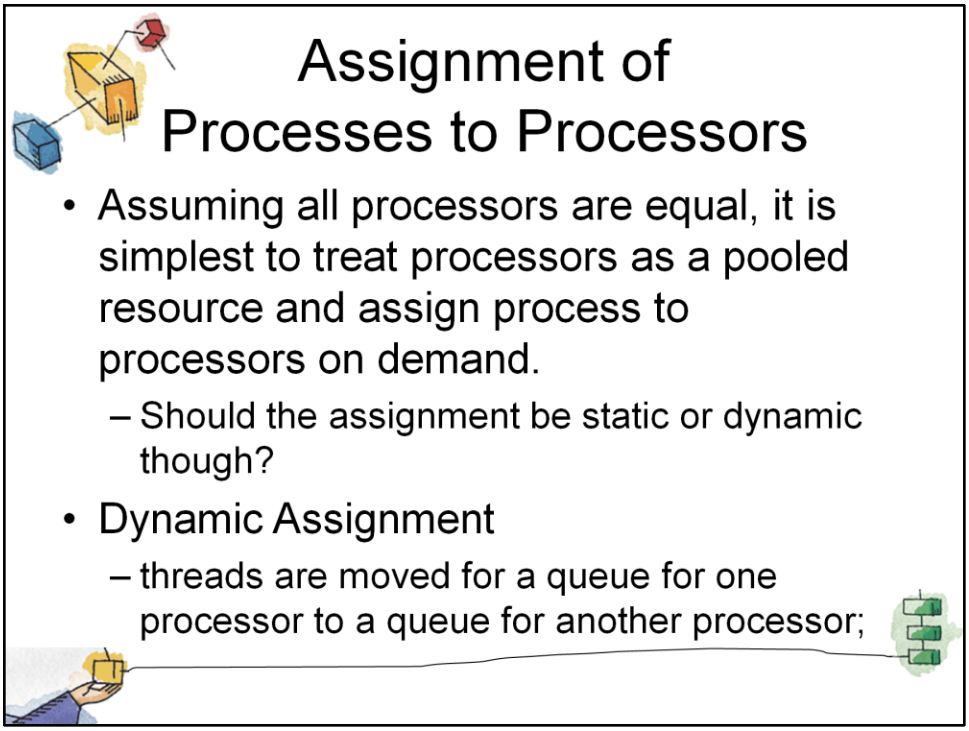
# Scheduling Design Issues

- Scheduling on a multiprocessor involves three interrelated issues:
  - Assignment of processes to processors
  - Use of multiprogramming on individual processors
  - Actual dispatching of a process
- The approach taken will depend on the degree of granularity of applications and the number of processors available

Scheduling on a multiprocessor involves three interrelated issues:

- The assignment of processes to processors
- The use of multiprogramming on individual processors
- The actual dispatching of a process

It is important to keep in mind that the approach taken will depend, in general, on the degree of granularity of the applications and on the number of processors available.



# Assignment of Processes to Processors

- Assuming all processors are equal, it is simplest to treat processors as a pooled resource and assign process to processors on demand.
  - Should the assignment be static or dynamic though?
- Dynamic Assignment
  - threads are moved for a queue for one processor to a queue for another processor;

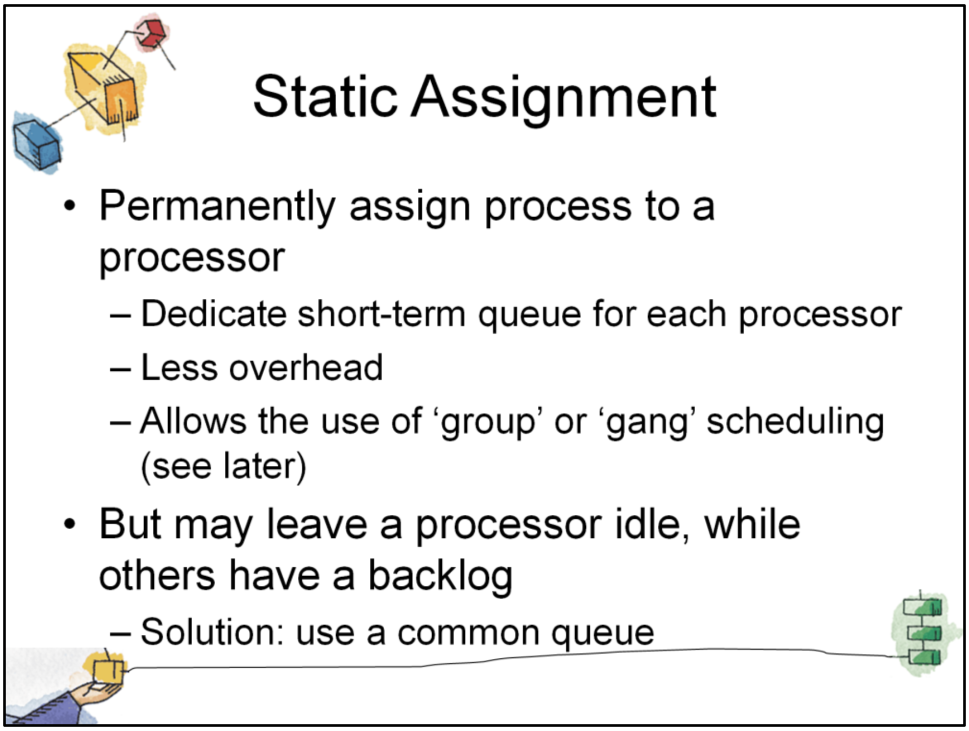
Assuming that the architecture of the multiprocessor is uniform, in the sense that no processor has a particular physical advantage with respect to access to main memory or to I/O devices,

- then the simplest scheduling approach is to treat the processors as a pooled resource
- and assign processes to processors on demand.

The question then arises as to whether the assignment should be static or dynamic.

With Dynamic Assignment, threads are moved for a queue for one processor to a queue for another processor;

Linux uses this approach.



# Static Assignment

- Permanently assign process to a processor
  - Dedicate short-term queue for each processor
  - Less overhead
  - Allows the use of ‘group’ or ‘gang’ scheduling (see later)
- But may leave a processor idle, while others have a backlog
  - Solution: use a common queue

If a process is permanently assigned to one processor from activation until its completion,

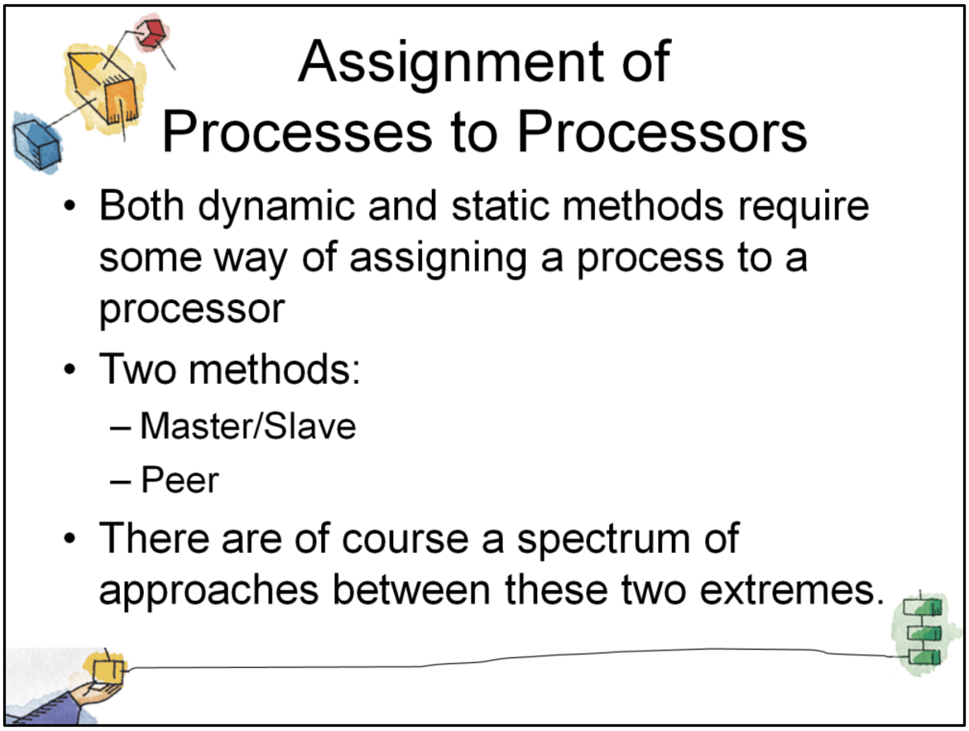
- then a dedicated short-term queue is maintained for each processor.

An advantage of this approach is that there may be less overhead in the scheduling function, because the processor assignment is made once and for all.

Also, the use of dedicated processors allows a strategy known as group or gang scheduling, as discussed later.

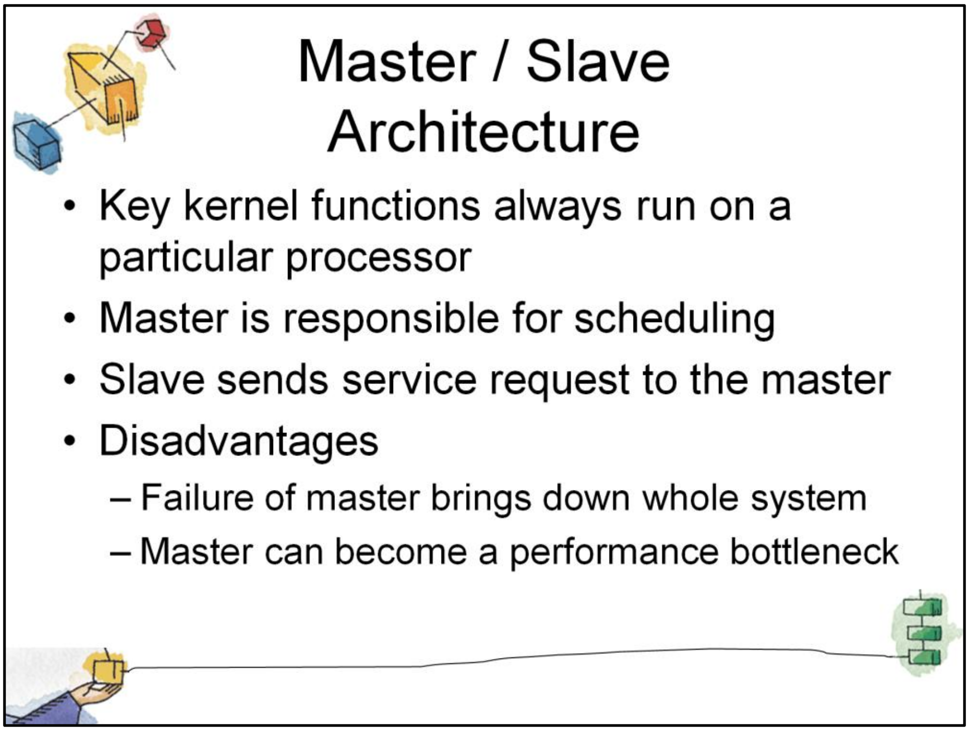
A disadvantage is that one processor can be idle, with an empty queue, while another processor has a backlog.

- To prevent this situation, a common queue can be used.
- All processes go into one global queue and are scheduled to any available processor.



# Assignment of Processes to Processors

- Both dynamic and static methods require some way of assigning a process to a processor
- Two methods:
  - Master/Slave
  - Peer
- There are of course a spectrum of approaches between these two extremes.



## Master / Slave Architecture

- Key kernel functions always run on a particular processor
- Master is responsible for scheduling
- Slave sends service request to the master
- Disadvantages
  - Failure of master brings down whole system
  - Master can become a performance bottleneck

Key kernel functions of the operating system always run on a particular processor.

- The other processors may only execute user programs.

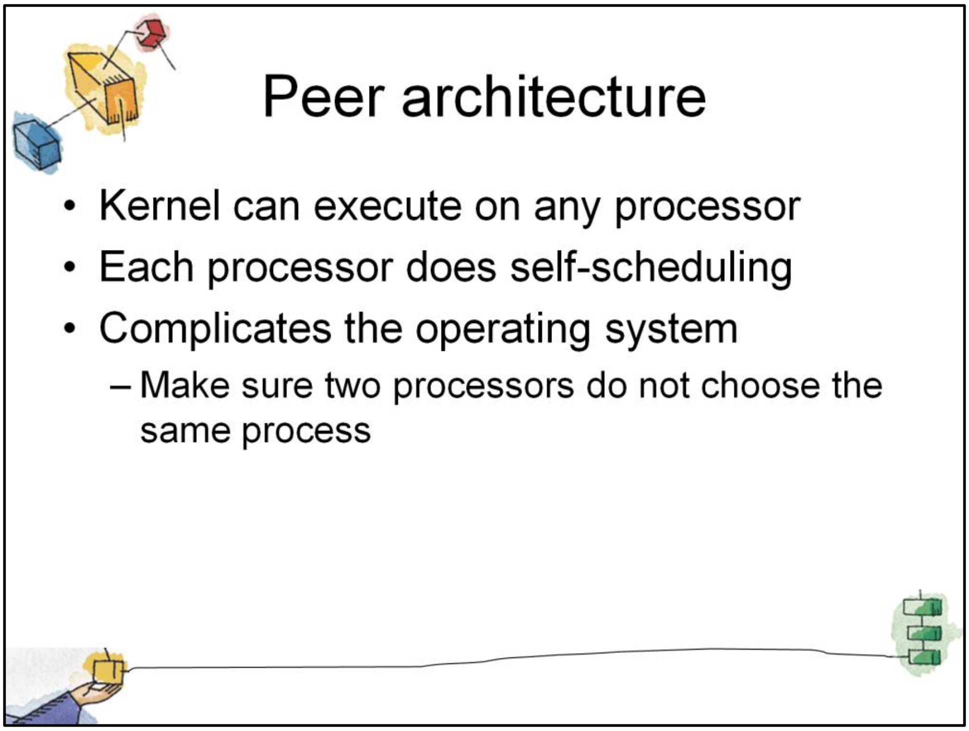
The master is responsible for scheduling jobs.

Once a process is active, if the slave needs service (e.g., an I/O call), it must send a request to the master and wait for the service to be performed.

There are two disadvantages to this approach:

(1) A failure of the master brings down the whole system,

(2) The master can become a performance bottleneck.



## Peer architecture

- Kernel can execute on any processor
- Each processor does self-scheduling
- Complicates the operating system
  - Make sure two processors do not choose the same process

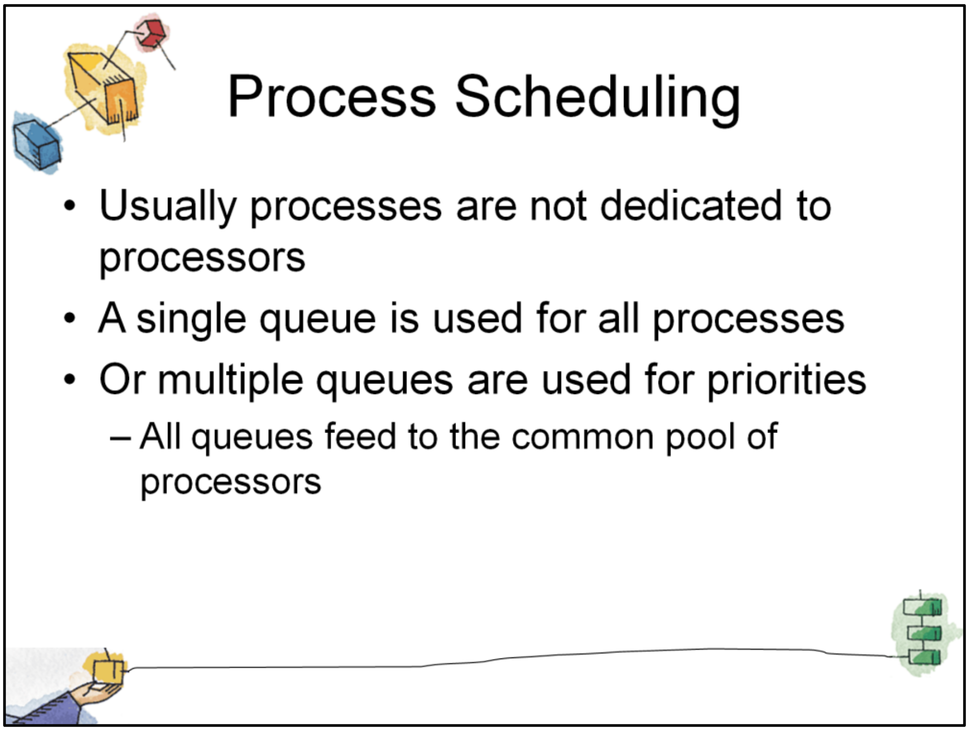
The kernel can execute on any processor,

- and each processor does self-scheduling from the pool of available processes.

This approach complicates the operating system.

The operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue.

Techniques must be employed to resolve and synchronize competing claims to resources.



# Process Scheduling

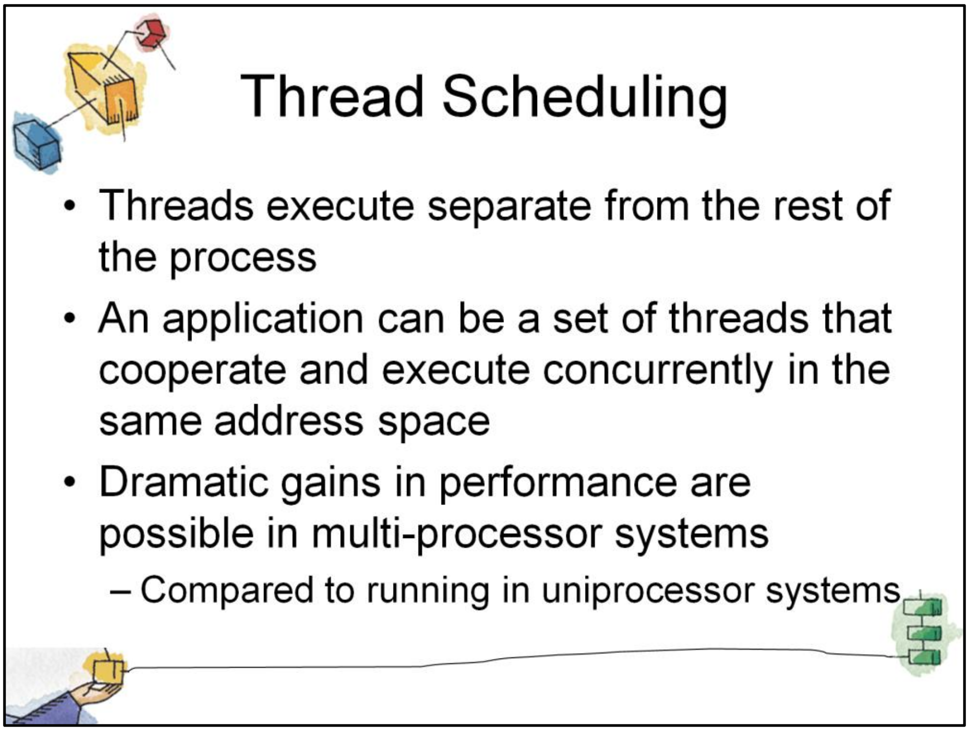
- Usually processes are not dedicated to processors
- A single queue is used for all processes
- Or multiple queues are used for priorities
  - All queues feed to the common pool of processors

Usually, processes are not dedicated to processors.

Instead, a single queue is used for all processors,

- or if some sort of priority scheme is used, there are multiple queues based on priority, all feeding into the common pool of processors.

We can view the system as being a multiserver queuing architecture.



# Thread Scheduling

- Threads execute separate from the rest of the process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
- Dramatic gains in performance are possible in multi-processor systems
  - Compared to running in uniprocessor systems

With threads, the concept of execution is separated from the rest of the definition of a process.

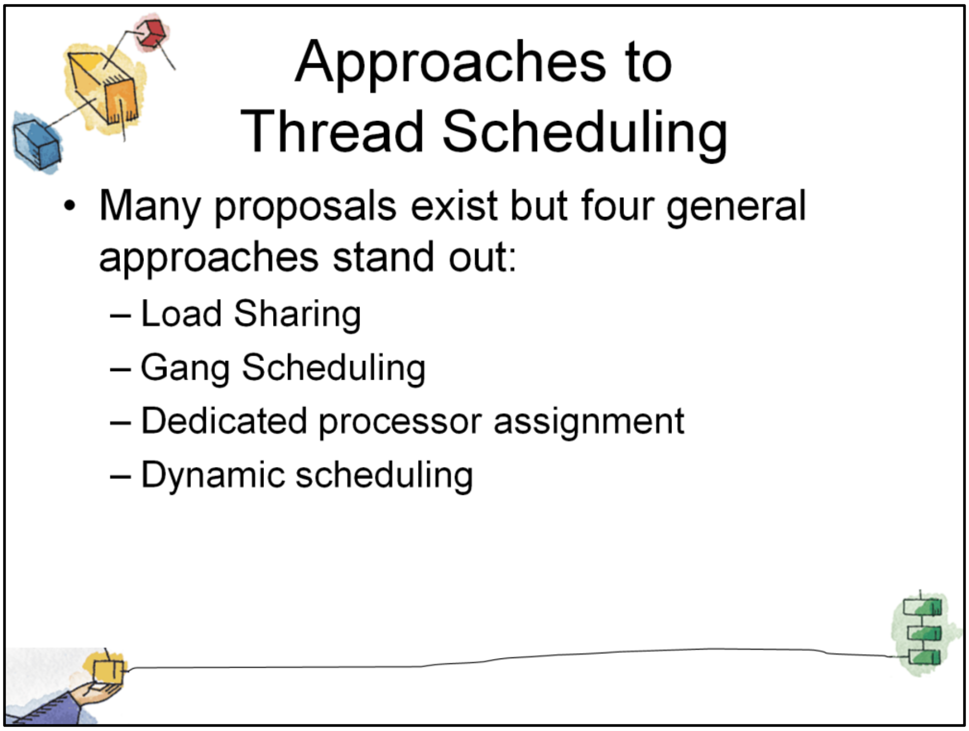
- An application can be implemented as a set of threads, which cooperate and execute concurrently in the same address space.

Threads on a uniprocessor system can be used as a program structuring aid and to overlap I/O with processing.

However, the full power of threads becomes evident in a multiprocessor system.

Threads can be used to exploit true parallelism in an application.

If the various threads of an application are simultaneously run on separate processors, dramatic gains in performance are possible.



# Approaches to Thread Scheduling

- Many proposals exist but four general approaches stand out:
  - Load Sharing
  - Gang Scheduling
  - Dedicated processor assignment
  - Dynamic scheduling

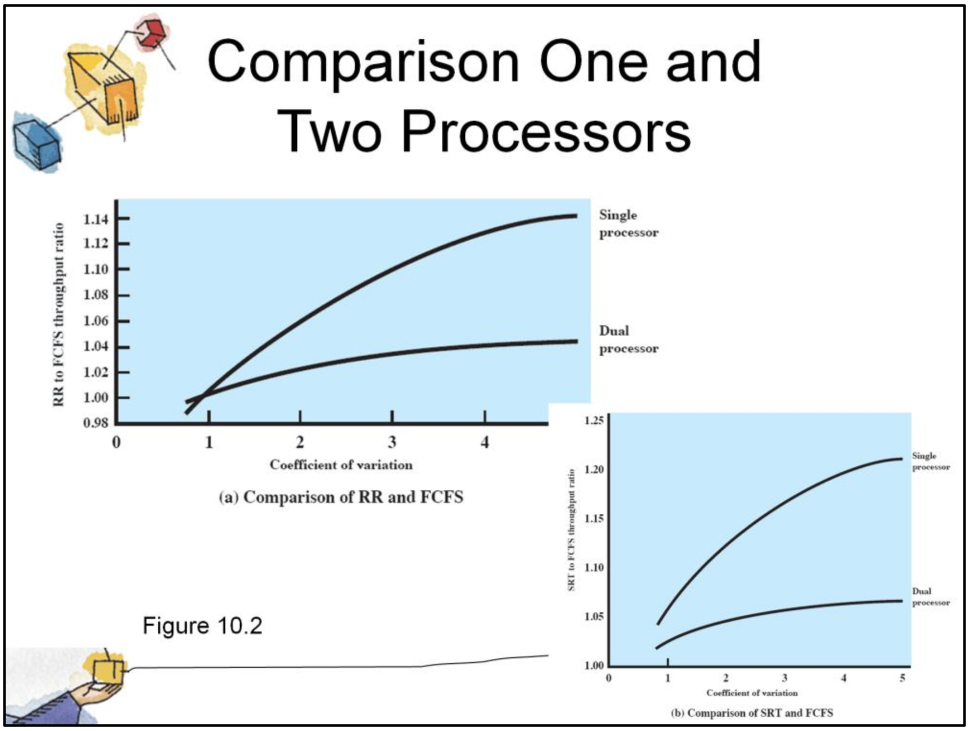
Among the many proposals for multiprocessor thread scheduling and processor assignment, four general approaches stand out:

•Load Sharing

•Gang Scheduling

•Dedicated processor assignment

•Dynamic scheduling



# Comparison One and Two Processors

(a) Comparison of RR and FCFS

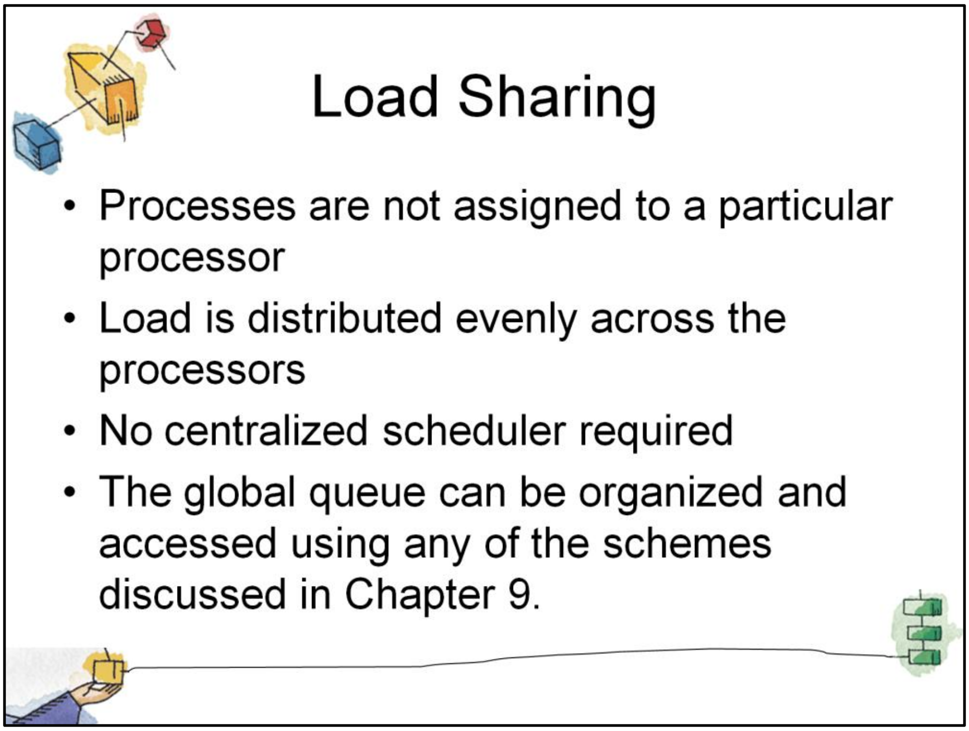
(b) Comparison of SRT and FCFS

Figure 10.2

Figure 10.2a compares round-robin throughput to FCFS throughput as a function of Coefficient Variation.

- Note that the difference in scheduling algorithms is much smaller in the dual-processor case.
- With two processors, a single process with long service time is much less disruptive in the FCFS case; other processes can use the other processor.

Similar results are shown in Figure 10.2b.



## Load Sharing

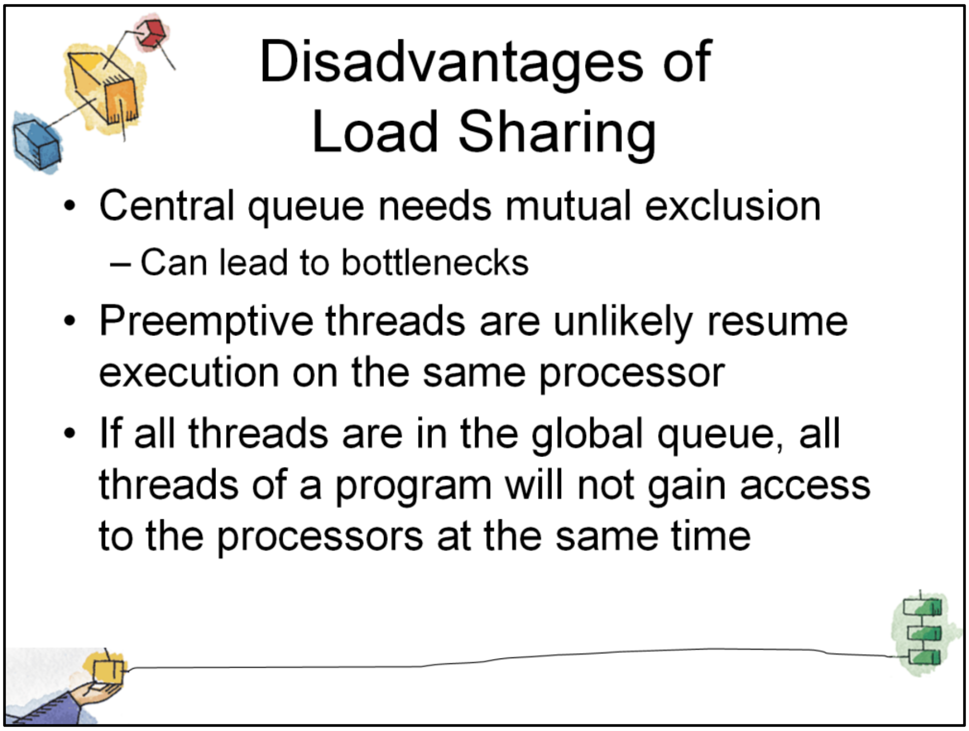
- Processes are not assigned to a particular processor
- Load is distributed evenly across the processors
- No centralized scheduler required
- The global queue can be organized and accessed using any of the schemes discussed in Chapter 9.

**Load Sharing:**

- Processes are not assigned to a particular processor.
- A global queue of ready threads is maintained, and each processor, when idle, selects a thread from the queue.
- The term load sharing is used to distinguish this strategy from *load-balancing* schemes in which work is allocated on a more permanent basis

**Advantages:**

- The load is distributed evenly across the processors, assuring that no processor is idle while work is available to do.
- No centralized scheduler is required; when a processor is available, the scheduling routine of the operating system is run on that processor to select the next thread.
- The global queue can be organized and accessed using any of the schemes discussed in Chapter 9.



## Disadvantages of Load Sharing

- Central queue needs mutual exclusion
  - Can lead to bottlenecks
- Preemptive threads are unlikely resume execution on the same processor
- If all threads are in the global queue, all threads of a program will not gain access to the processors at the same time

The central queue occupies a region of memory that must be accessed in a manner that enforces mutual exclusion.

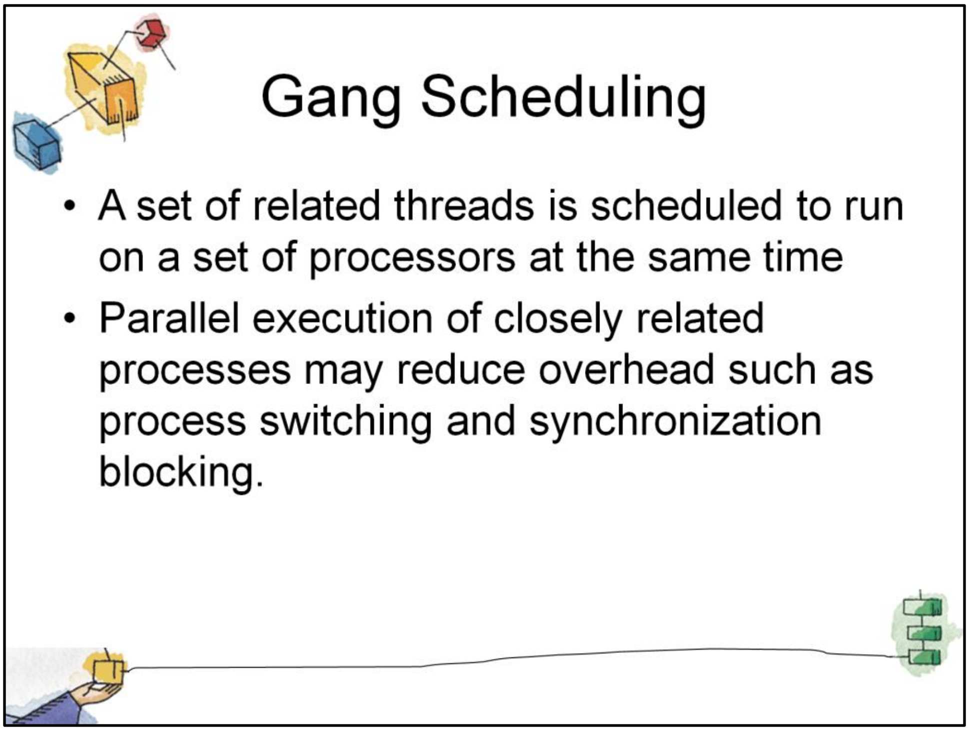
- Thus, it may become a bottleneck if many processors look for work at the same time.
- When there is only a small number of processors, this is unlikely to be a noticeable problem. But when the multiprocessor consists of dozens or even hundreds of processors, the potential for bottleneck is real.

Preempted threads are unlikely to resume execution on the same processor.

- If each processor is equipped with a local cache, caching becomes less efficient.

If all threads are treated as a common pool of threads, it is unlikely that all of the threads of a program will gain access to processors at the same time.

- If a high degree of coordination is required between the threads of a program, the process switches involved may seriously compromise performance.



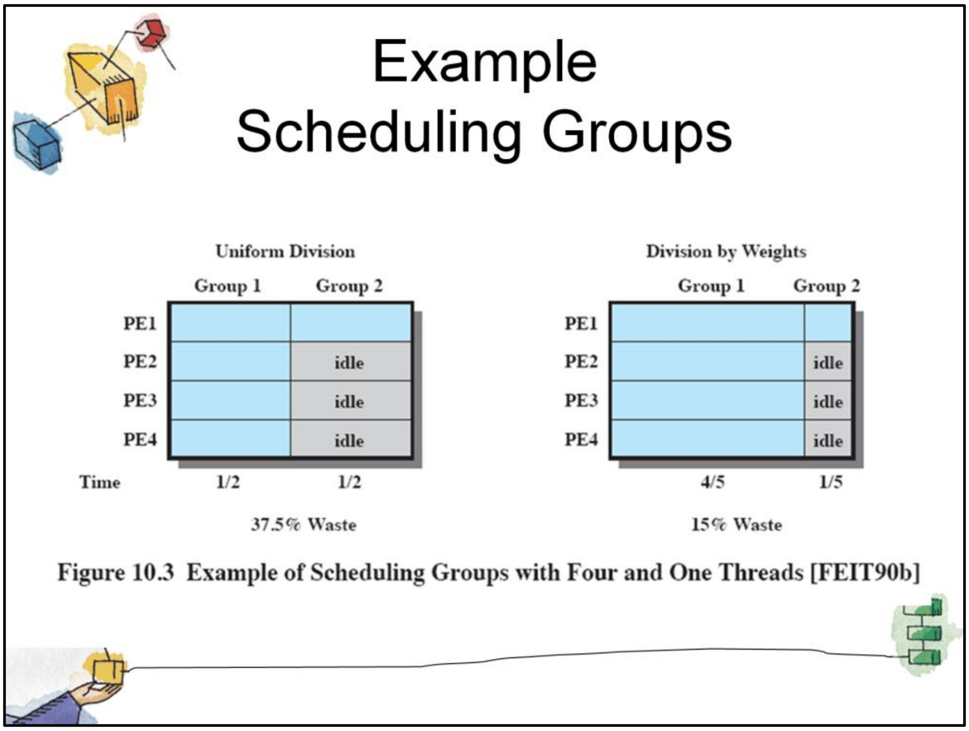
# Gang Scheduling

- A set of related threads is scheduled to run on a set of processors at the same time
- Parallel execution of closely related processes may reduce overhead such as process switching and synchronization blocking.

A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.

If closely related processes execute in parallel, synchronization blocking may be reduced, less process switching may be necessary, and performance will increase.

Scheduling overhead may be reduced because a single decision affects a number of processors and processes at one time.



## Example Scheduling Groups

Uniform Division

| | Group 1 | Group 2 |
|---|---|---|
| PE1 | | |
| PE2 | | idle |
| PE3 | | idle |
| PE4 | | idle |
| Time | 1/2 | 1/2 |

37.5% Waste

Division by Weights

| | Group 1 | Group 2 |
|---|---|---|
| PE1 | | |
| PE2 | | idle |
| PE3 | | idle |
| PE4 | | idle |
| Time | 4/5 | 1/5 |

15% Waste

**Figure 10.3 Example of Scheduling Groups with Four and One Threads [FEIT90b]**

The use of gang scheduling creates a requirement for processor allocation.

Consider an example in which there are two ap plications,

- one with four threads and one with one thread.

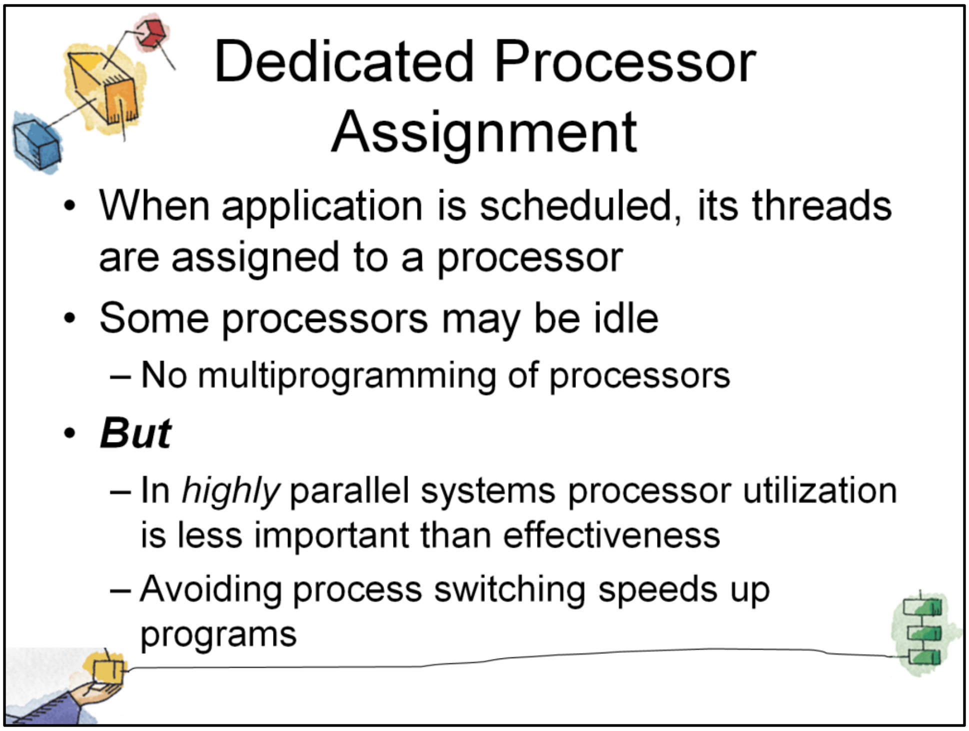
Using uniform time allocation wastes 37.5% of the processing resource,

- because when the single-thread application runs, three processors are left idle.

If there are several one-thread applications, these could all be fit together to increase processor utilization.

If that option is not available, an alternative to uniform scheduling is scheduling that is weighted by the number of threads.

- Thus, the four-thread application could be given four-fifths of the time and the one-thread application given only one-fifth of the time, reducing the processor waste to 15%.



## Dedicated Processor Assignment

- When application is scheduled, its threads are assigned to a processor
- Some processors may be idle
  - No multiprogramming of processors
- ***But***
  - In *highly* parallel systems processor utilization is less important than effectiveness
  - Avoiding process switching speeds up programs

Each program is allocated a number of processors equal to the number of threads in the program, for the duration of the program execution.

- When the program terminates, the processors return to the general pool for possible allocation to another program.

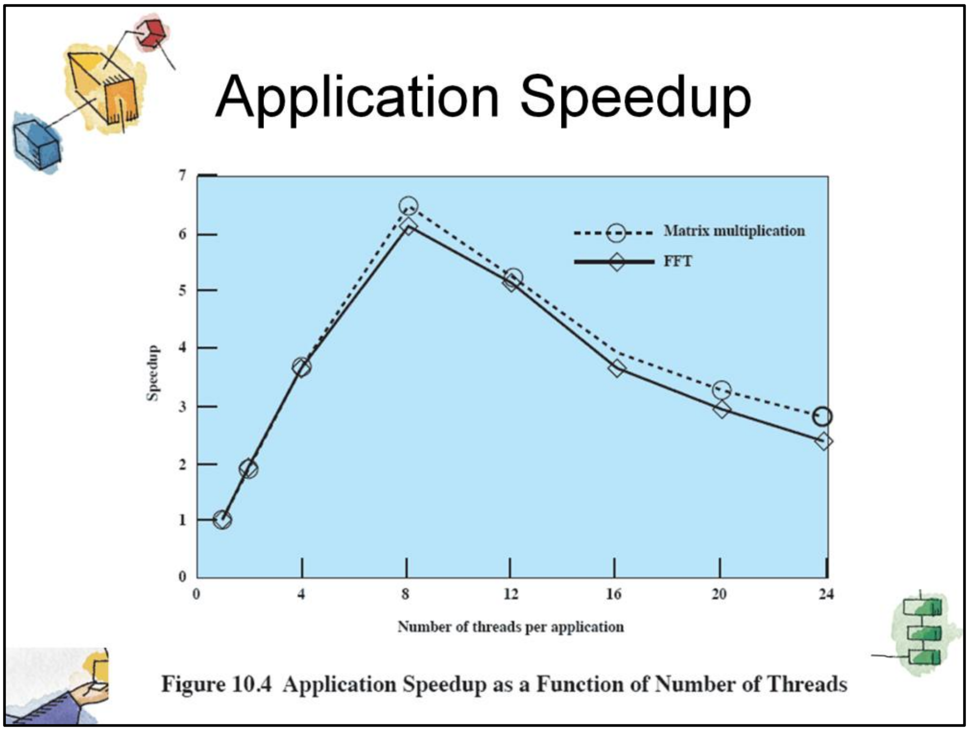
This approach would appear to be extremely wasteful of processor time.

- If a thread of an application is blocked waiting for I/O or for synchronization with another thread, then that thread's processor remains idle: there is no multiprogramming of processors.

***But***

1) In a highly parallel system, with tens or hundreds of processors, each of which represents a small fraction of the cost of the system, processor utilization is no longer so important as a metric for effectiveness or performance.

2) The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program.



# Application Speedup

Figure 10.4 Application Speedup as a Function of Number of Threads

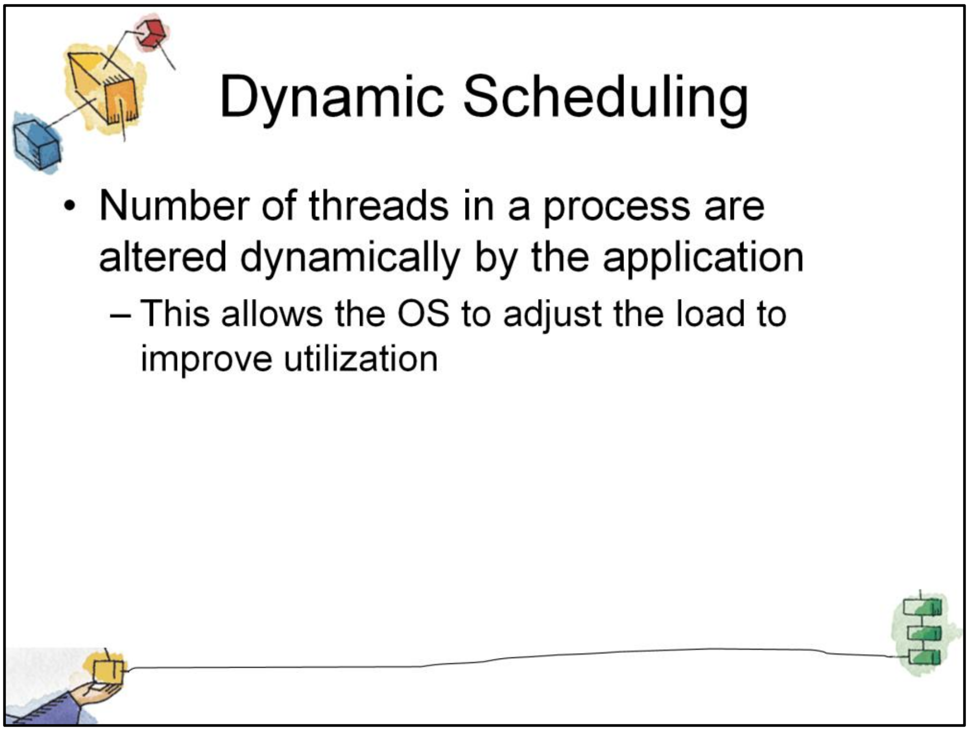
Figure 10.4 shows the speedup for the applications as the number of threads executing the tasks in each application is varied from 1 to 24.

We see that when both applications are started simultaneously with 24 threads each, the speedup obtained, compared to using a single thread for each application, is 2.8 for matrix multiplication and 2.4 for FFT.

The figure shows that the performance of both applications worsens considerably when the number of threads in each application exceeds 8 and thus the total number of processes in the system exceeds the number of processors.

Furthermore, the larger the number of threads the worse the performance gets, because there is a greater frequency of thread preemption and rescheduling.

- This excessive preemption results in inefficiency from many sources, including time spent waiting for a suspended thread to leave a critical section, time wasted in process switching, and inefficient cache behavior.

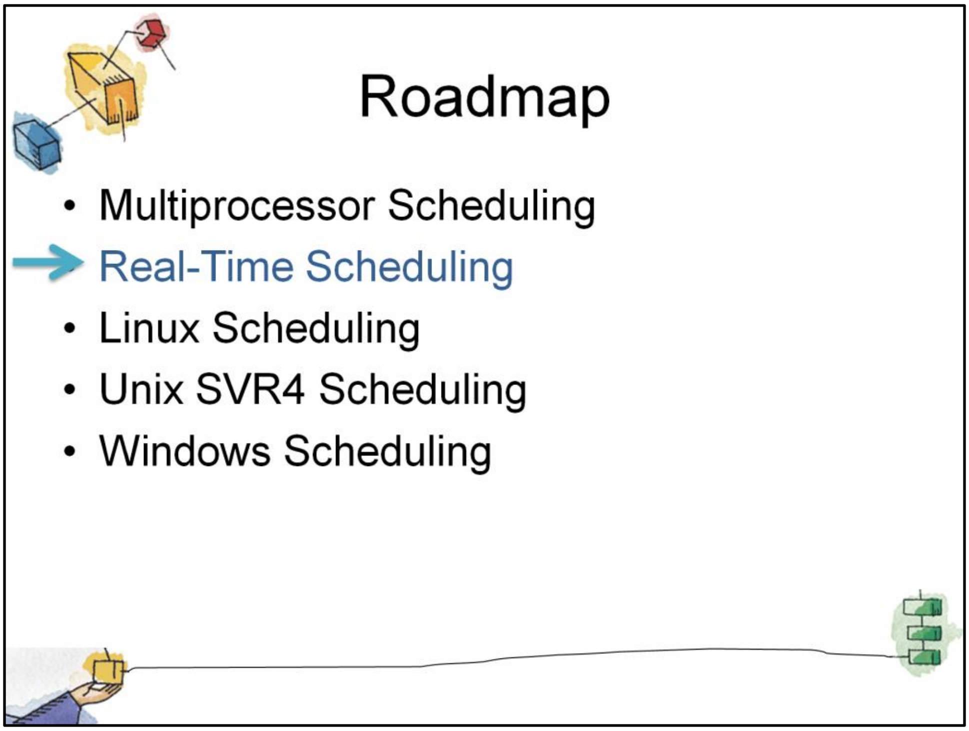


# Dynamic Scheduling

- Number of threads in a process are altered dynamically by the application
  - This allows the OS to adjust the load to improve utilization

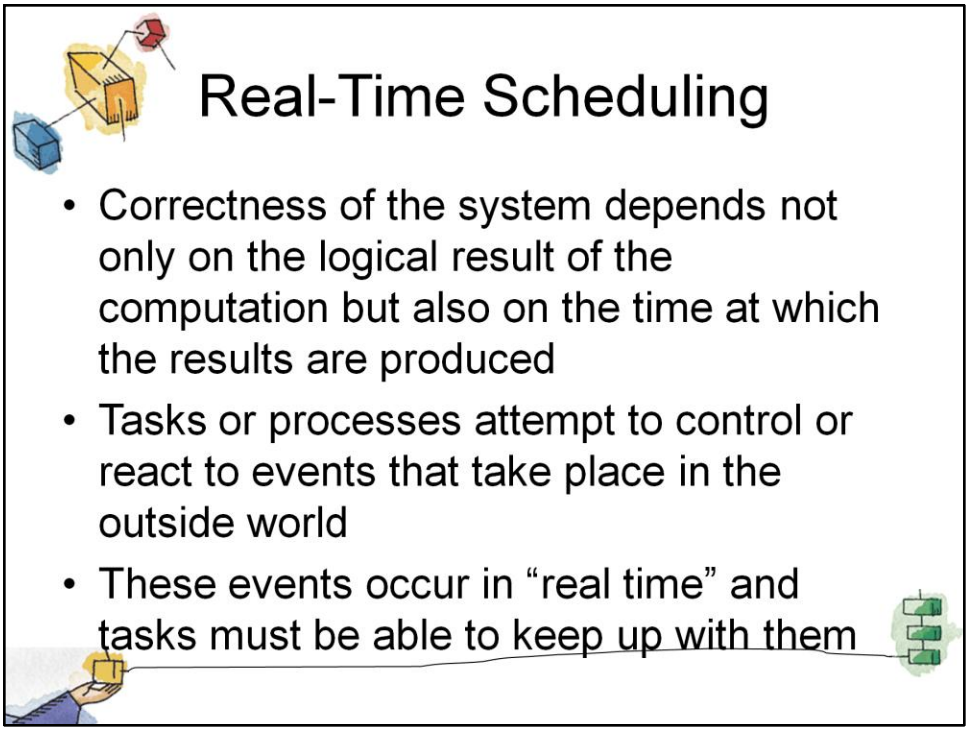
The number of threads in a process can be altered during the course of execution

- allowing the operating system to adjust the load to improve utilization.



# Roadmap

- Multiprocessor Scheduling
- → Real-Time Scheduling
- Linux Scheduling
- Unix SVR4 Scheduling
- Windows Scheduling



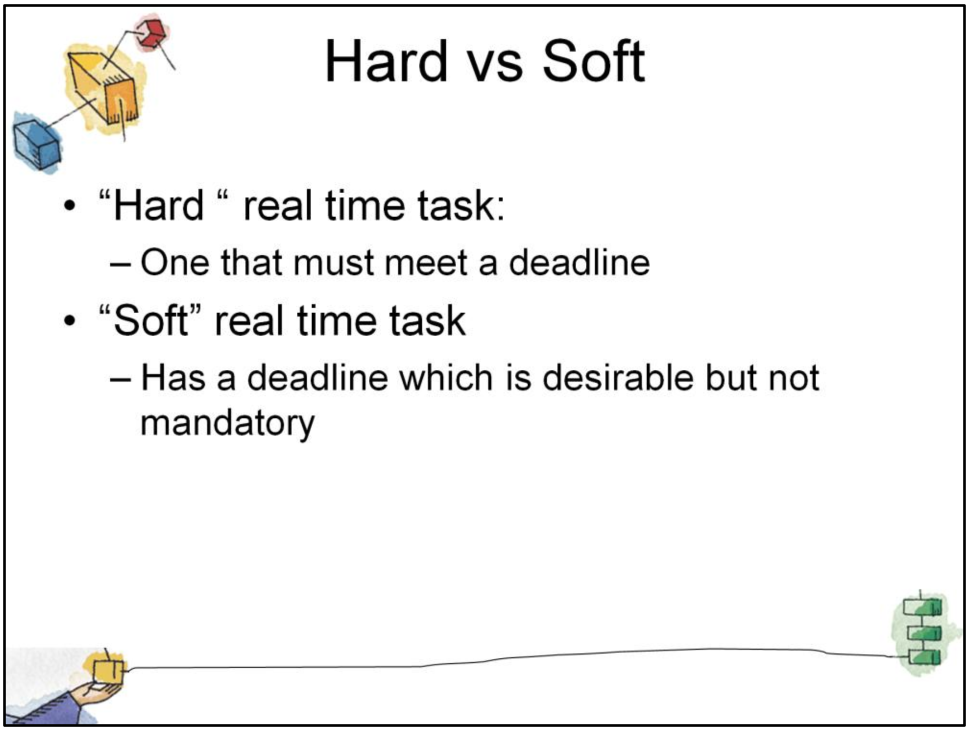
# Real-Time Scheduling

- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced
- Tasks or processes attempt to control or react to events that take place in the outside world
- These events occur in “real time” and tasks must be able to keep up with them

Real-time computing may be defined as that type of computing in which the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

A real-time system is defined by what is meant by a real-time process, or task.

In general, in a real-time system, some of the tasks are real-time tasks, and these have a certain degree of urgency to them.



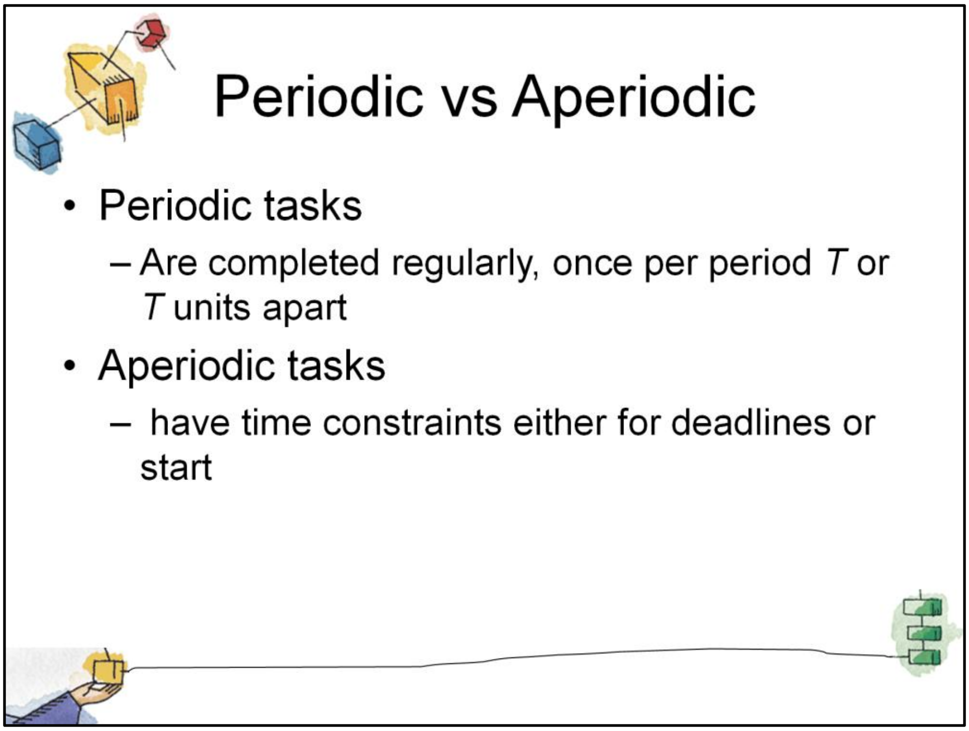
# Hard vs Soft

- “Hard “ real time task:
  - One that must meet a deadline
- “Soft” real time task
  - Has a deadline which is desirable but not mandatory

A **hard real-time** task is one that must meet its deadline;

- otherwise it will cause unacceptable damage or a fatal error to the system.

A **soft real-time task** has an associated deadline that is desirable but not mandatory; it still makes sense to schedule and complete the task even if it has passed its deadline.

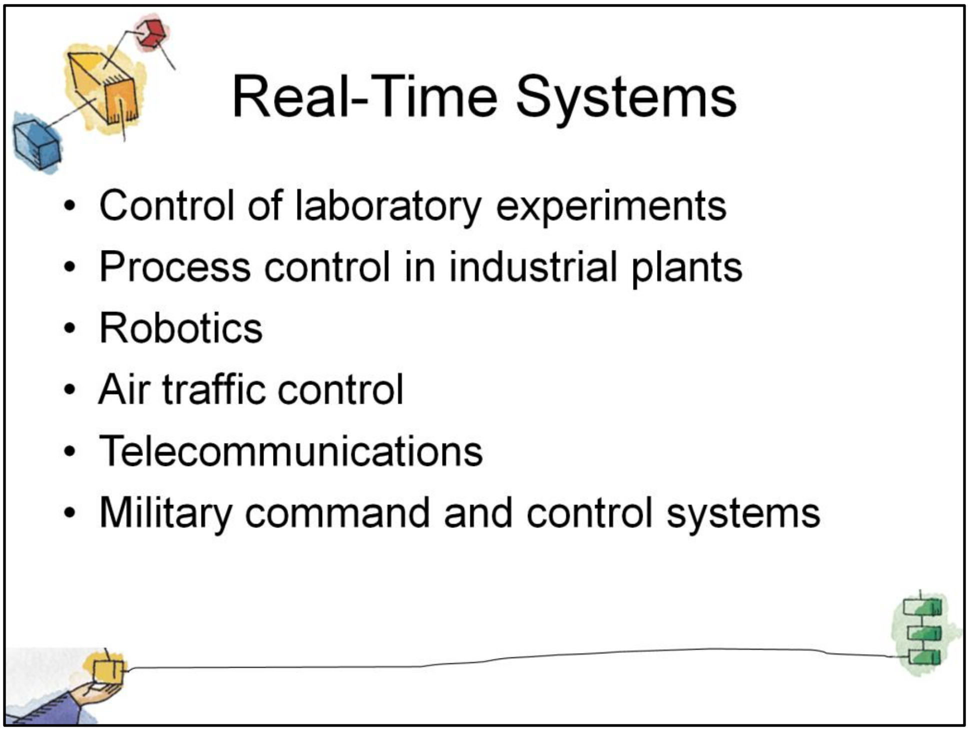


# Periodic vs Aperiodic

- Periodic tasks
  - Are completed regularly, once per period *T* or *T* units apart
- Aperiodic tasks
  - have time constraints either for deadlines or start

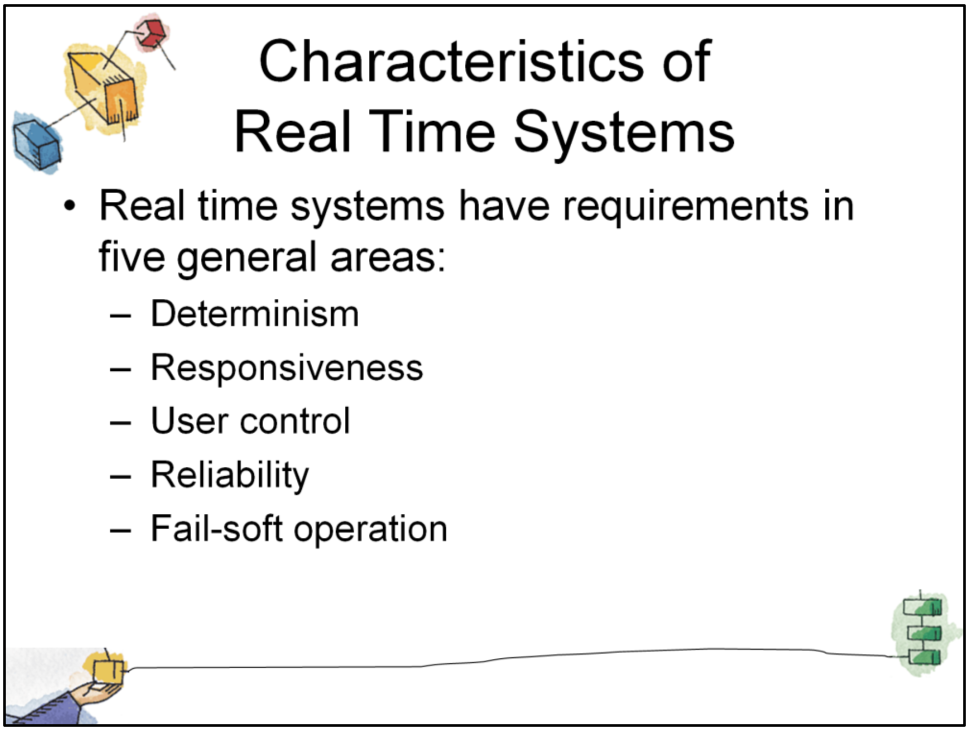
A **periodic task**, the requirement may be stated as “once per period T” or “exactly T units apart.”

An **aperiodic** task has a deadline by which it must finish or start, or it may have a constraint on both start and finish time.



# Real-Time Systems

- Control of laboratory experiments
- Process control in industrial plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems

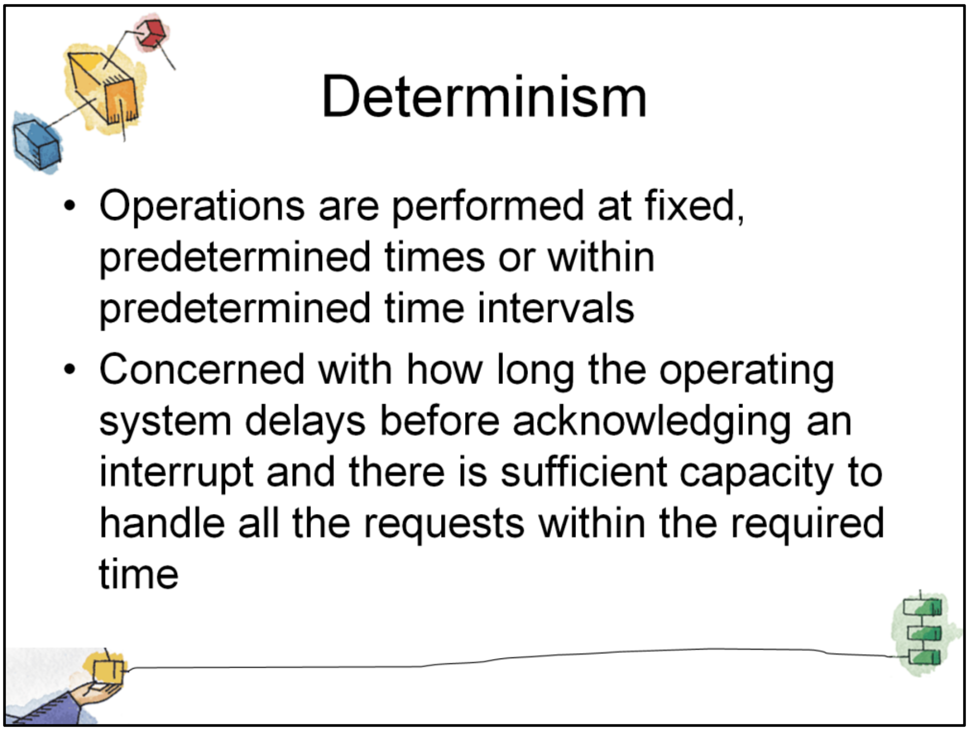


## Characteristics of Real Time Systems

- Real time systems have requirements in five general areas:
  - Determinism
  - Responsiveness
  - User control
  - Reliability
  - Fail-soft operation

Real-time operating systems can be characterized as having unique requirements in five general areas:

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation



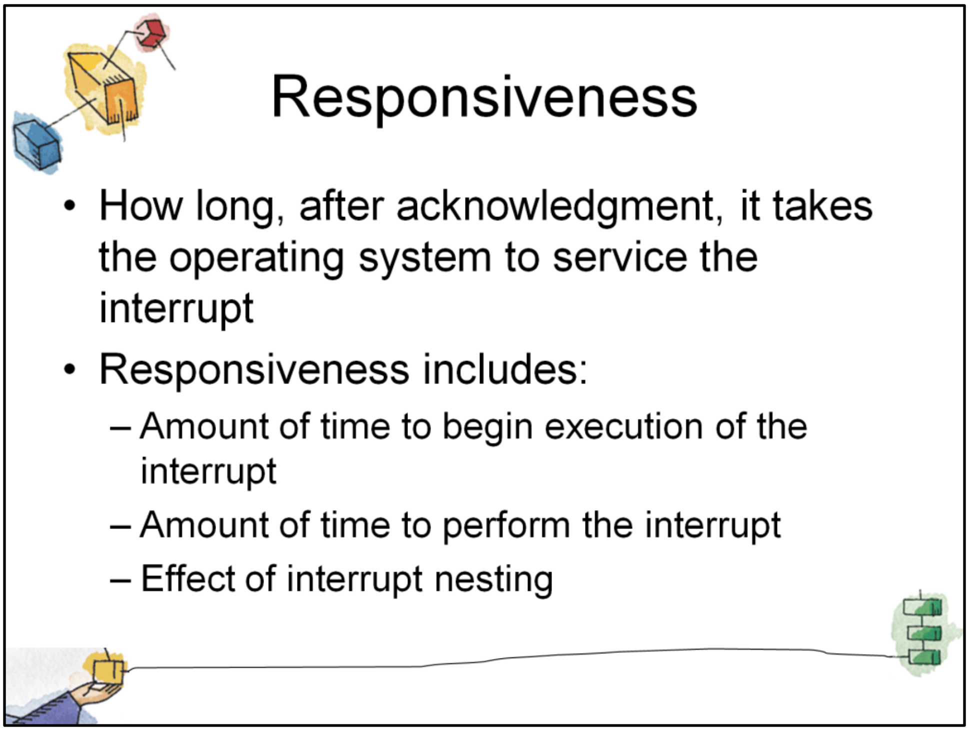
# Determinism

- Operations are performed at fixed, predetermined times or within predetermined time intervals
- Concerned with how long the operating system delays before acknowledging an interrupt and there is sufficient capacity to handle all the requests within the required time

An operating system is deterministic to the extent that it performs operations at fixed, predetermined times or within predetermined time intervals.

One useful measure of the ability of an operating system to function deterministically is the maximum delay from the arrival of a high-priority device interrupt to when servicing begins.

- In non-real-time operating systems, this delay may be in the range of tens to hundreds of milliseconds,
- While in real-time operating systems that delay may have an upper bound of anywhere from a few microseconds to a millisecond.



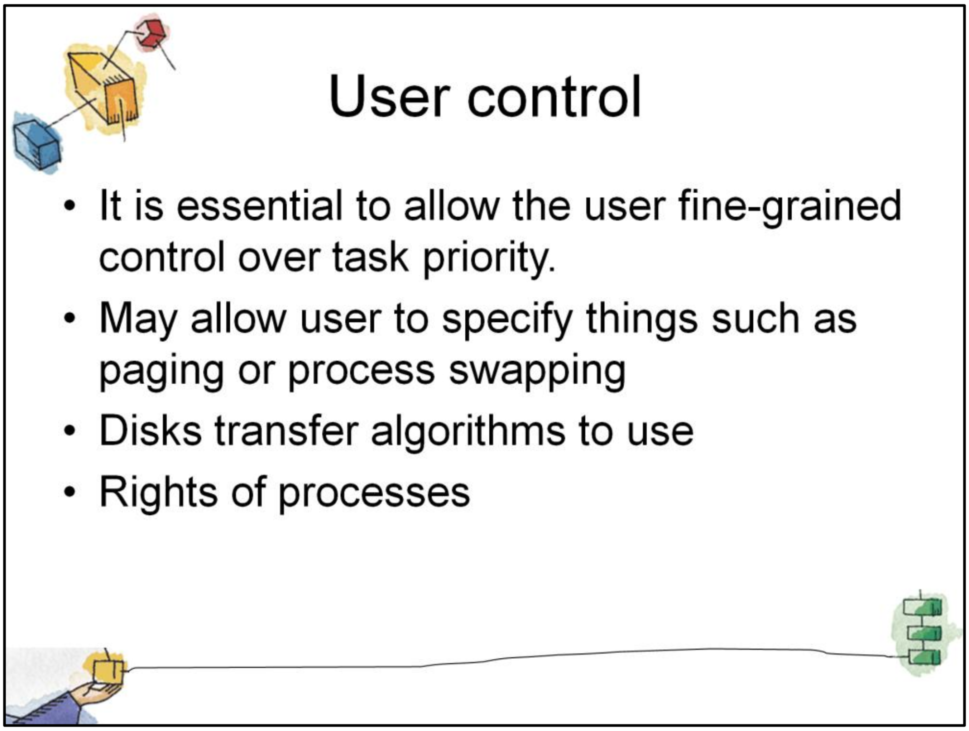
# Responsiveness

- How long, after acknowledgment, it takes the operating system to service the interrupt
- Responsiveness includes:
  - Amount of time to begin execution of the interrupt
  - Amount of time to perform the interrupt
  - Effect of interrupt nesting

Responsiveness is concerned with how long, after acknowledgment, it takes an operating system to service the interrupt.

Aspects of responsiveness include:

1. The amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR).
2. The amount of time required to perform the ISR.
3. The effect of interrupt nesting.



## User control

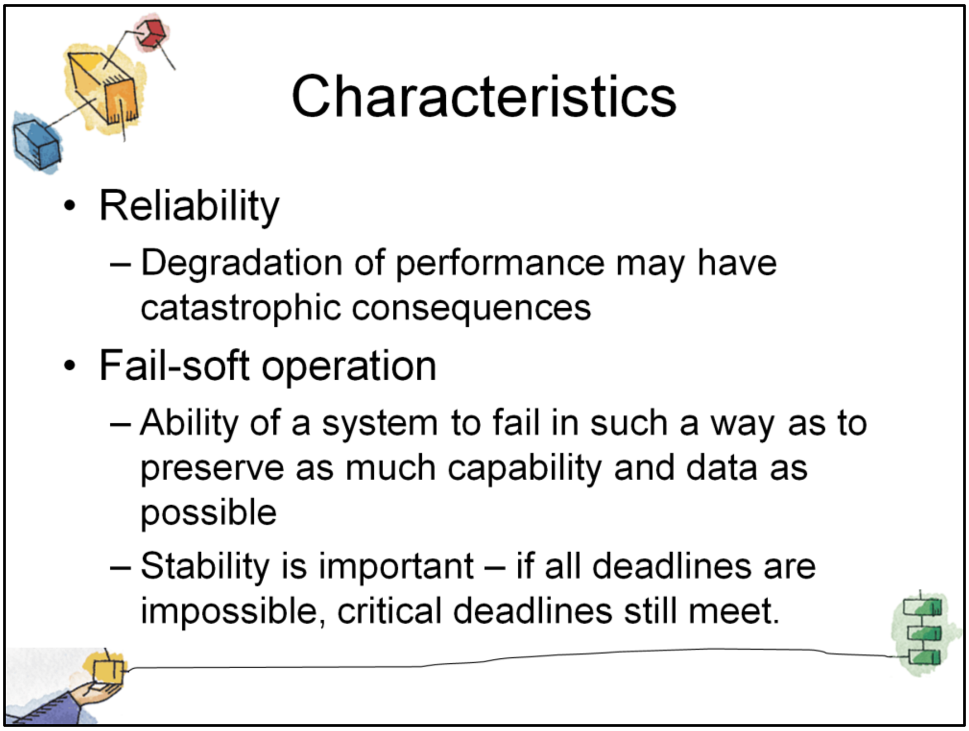
- It is essential to allow the user fine-grained control over task priority.
- May allow user to specify things such as paging or process swapping
- Disks transfer algorithms to use
- Rights of processes

In a real-time system it is essential to allow the user fine-grained control over task priority.

The user should be able to distinguish between hard and soft tasks and to specify relative priorities within each class.

A real-time system may also allow the user to specify such characteristics as

- the use of paging or process swapping,
- what processes must always be resident in main memory,
- what disk transfer algorithms are to be used,
- what rights the processes in various priority bands have, and so on.



## Characteristics

- Reliability
  - Degradation of performance may have catastrophic consequences
- Fail-soft operation
  - Ability of a system to fail in such a way as to preserve as much capability and data as possible
  - Stability is important – if all deadlines are impossible, critical deadlines still meet.

Reliability is typically far more important for real-time systems than non-real-time systems

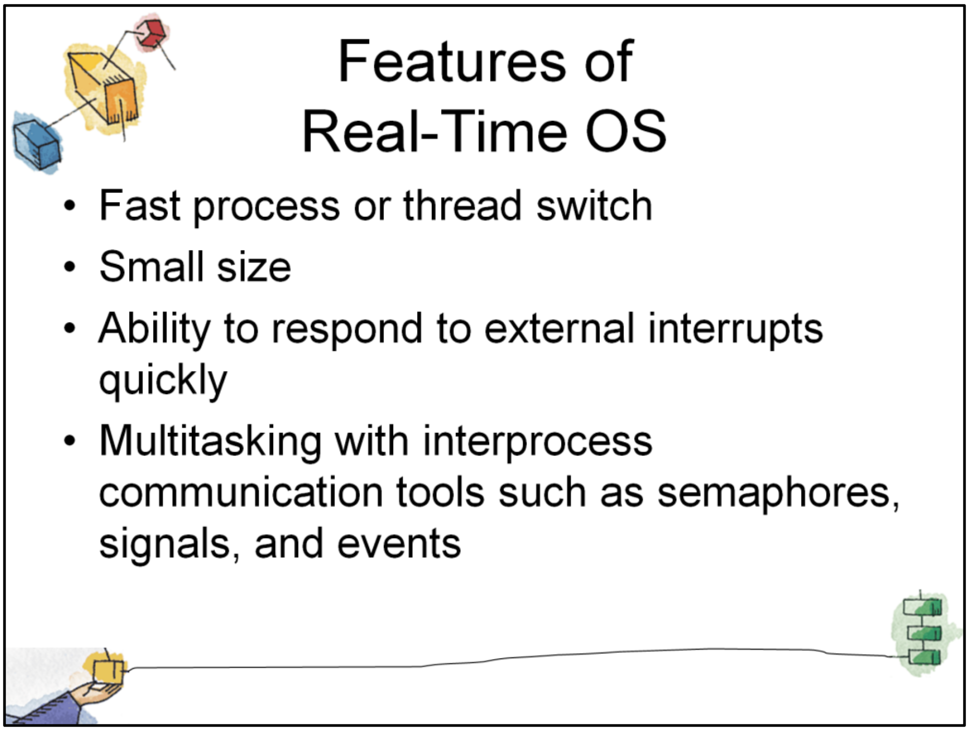
A real-time system is responding to and controlling events in real time.

- Loss or degradation of performance may have catastrophic consequences, ranging from financial loss to major equipment damage and even loss of life.

**Fail-soft operation** is a characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible.

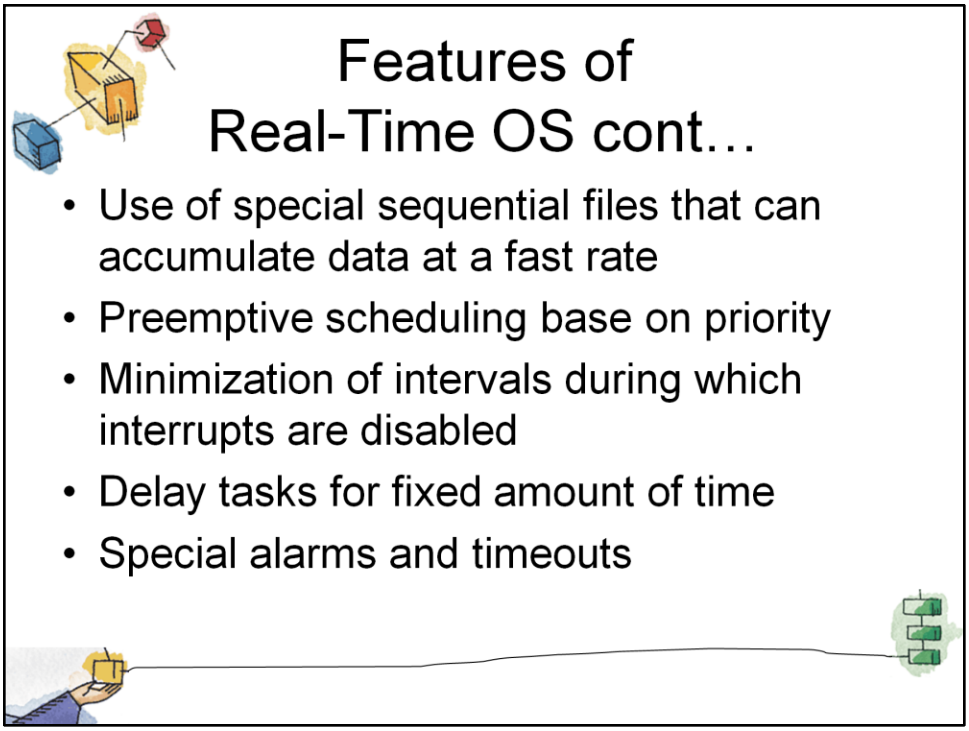
An important aspect of fail-soft operation is referred to as stability.

- A real-time system is stable if, in cases where it is impossible to meet all task deadlines, the system will meet the deadlines of its most critical, highest-priority tasks, even if some less critical task deadlines are not always met.



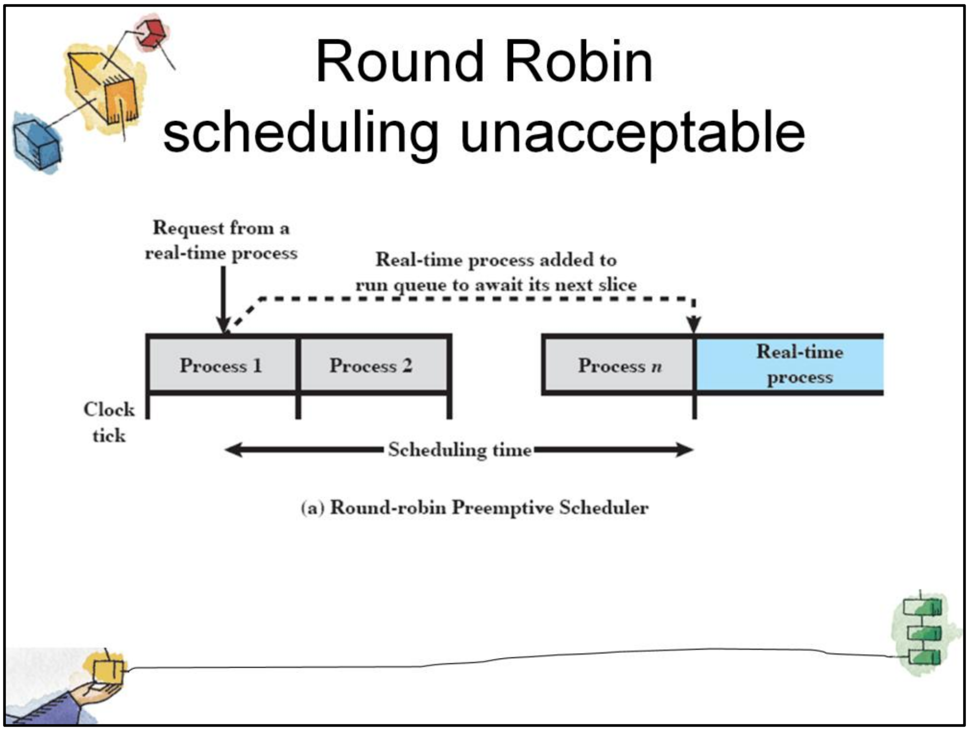
# Features of Real-Time OS

- Fast process or thread switch
- Small size
- Ability to respond to external interrupts quickly
- Multitasking with interprocess communication tools such as semaphores, signals, and events



# Features of Real-Time OS cont…

- Use of special sequential files that can accumulate data at a fast rate
- Preemptive scheduling base on priority
- Minimization of intervals during which interrupts are disabled
- Delay tasks for fixed amount of time
- Special alarms and timeouts



# Round Robin scheduling unacceptable

Request from a real-time process

Real-time process added to run queue to await its next slice

Process 1

Process 2

Process *n*

Real-time process

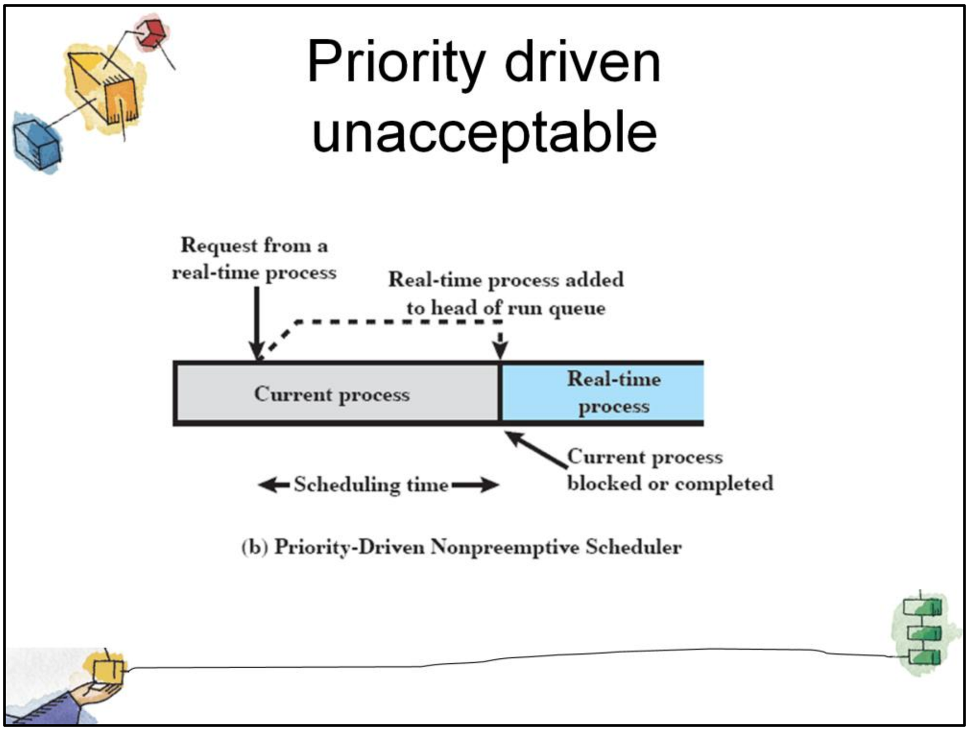
Clock tick

Scheduling time

(a) Round-robin Preemptive Scheduler

In a preemptive scheduler that uses simple round-robin scheduling, a real-time task would be added to the ready queue to await its next time slice, as illustrated in Figure 10.5a.

In this case, the scheduling time will generally be unacceptable for real-time applications.



# Priority driven unacceptable

Request from a real-time process
Real-time process added to head of run queue
Current process
Real-time process
Current process blocked or completed
Scheduling time

(b) Priority-Driven Nonpreemptive Scheduler

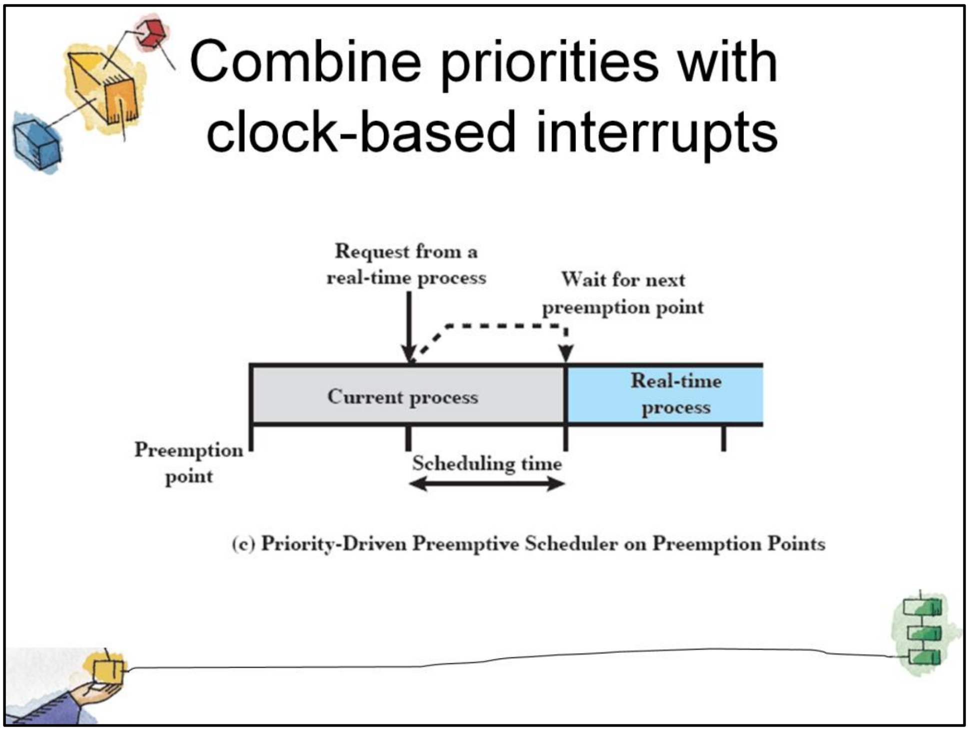
In a nonpreemptive scheduler, we could use a priority scheduling mechanism, giving real-time tasks higher priority.

In this case, a real-time task that is ready would be scheduled as soon as the current process blocks or runs to com-

pletion (Figure 10.5b).

This could lead to a delay of several seconds if a slow, low-priority task were executing at a critical time.

Again, this approach is not acceptable.



## Combine priorities with clock-based interrupts

(c) Priority-Driven Preemptive Scheduler on Preemption Points

A more promising approach is to combine priorities with clock-based interrupts.

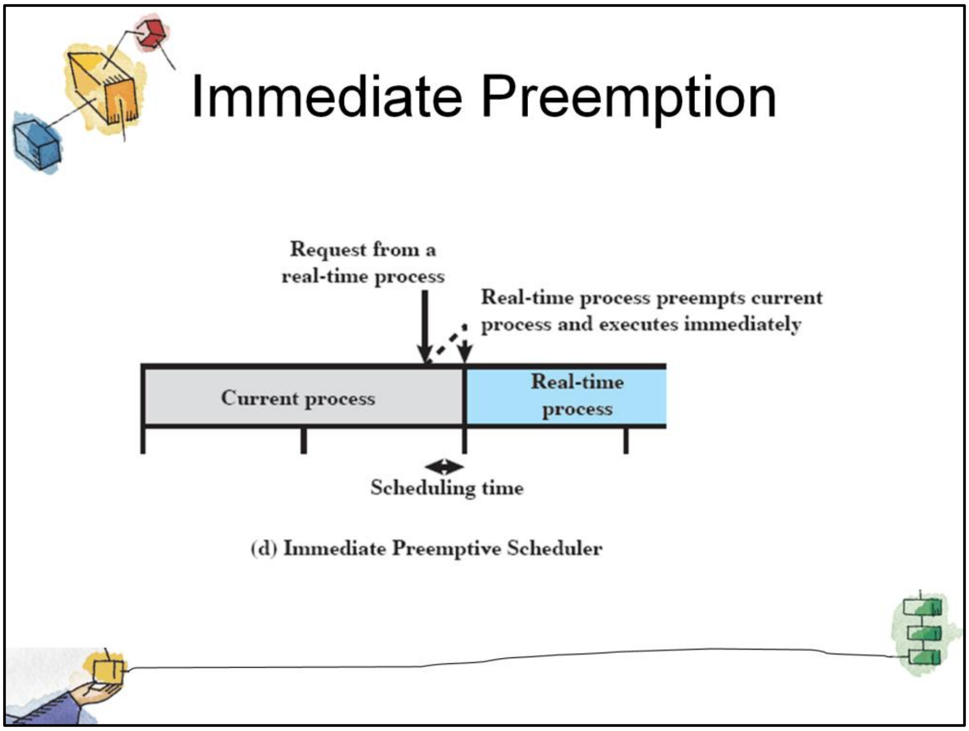
Preemption points occur at regular intervals.

When a preemption point occurs, the currently running task is preempted if a higher-priority task is waiting.

This would include the preemption of tasks that are part of the operating system kernel.

Such a delay may be on the order of several milliseconds.

While this last approach may be adequate for some real-time applications, it will not suffice for more demanding applications.



# Immediate Preemption

Request from a real-time process

Real-time process preempts current process and executes immediately

Current process

Real-time process

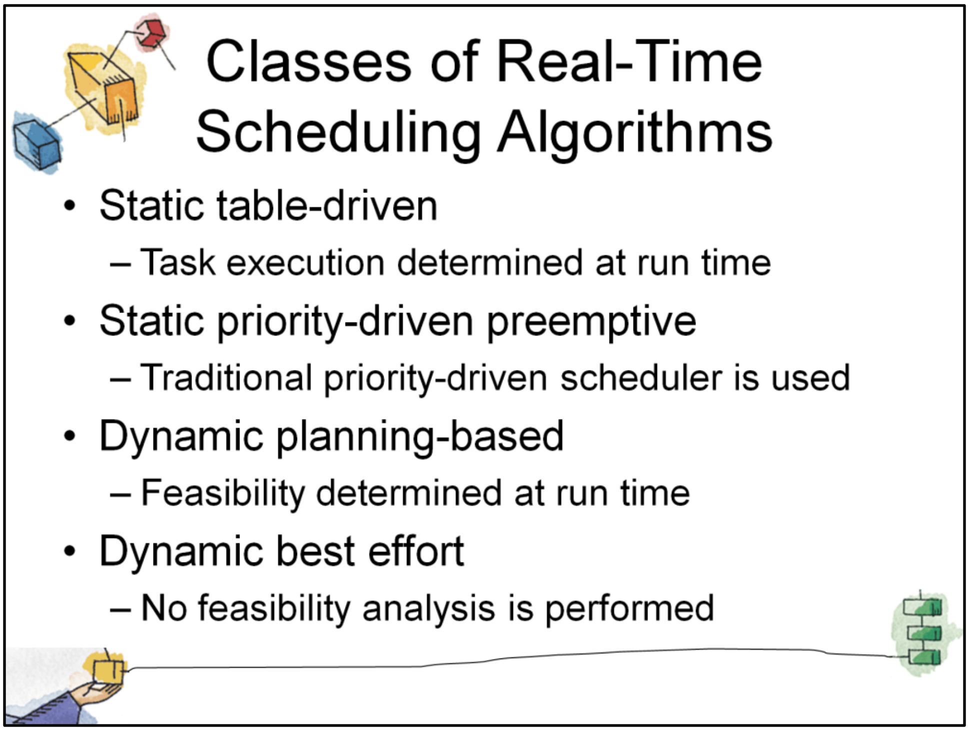
Scheduling time

(d) Immediate Preemptive Scheduler

For more demanding applications, the approach that has been taken is sometimes referred to as **immediate preemption**.

In this case, the operating system responds to an interrupt almost immediately, unless the system is in a critical-code lockout section.

Scheduling delays for a real-time task can then be reduced to 100 "s or less.



## Classes of Real-Time Scheduling Algorithms

- Static table-driven
  - Task execution determined at run time
- Static priority-driven preemptive
  - Traditional priority-driven scheduler is used
- Dynamic planning-based
  - Feasibility determined at run time
- Dynamic best effort
  - No feasibility analysis is performed

**Static table-driven approaches**:

- These perform a static analysis of feasible schedules of dispatching.
- The result of the analysis is a schedule that determines, at run time, when a task must begin execution.

**Static priority-driven preemptive approaches:**

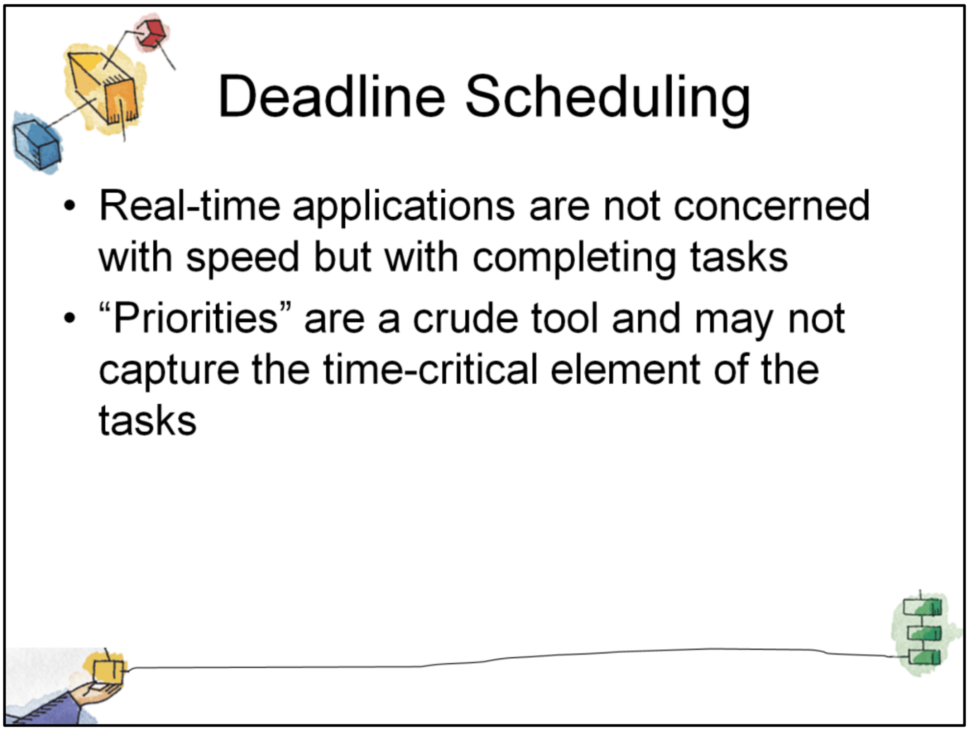
- Again, a static analysis is performed, but no schedule is drawn up.
- The analysis is used to assign priorities to tasks, so that a traditional priority-driven preemptive scheduler can be used.

**Dynamic planning-based approaches:**

- Feasibility is determined at run time (dynamically) rather than offline prior to the start of execution statically).
- An arriving task is accepted for execution only if it is feasible to meet its time constraints.
- One of the results of the feasibility analysis is a schedule or plan that is used to decide when to dispatch this task.

**Dynamic best effort approaches:**

- No feasibility analysis is performed.
- The system tries to meet all deadlines and aborts any started process whose deadline is missed.

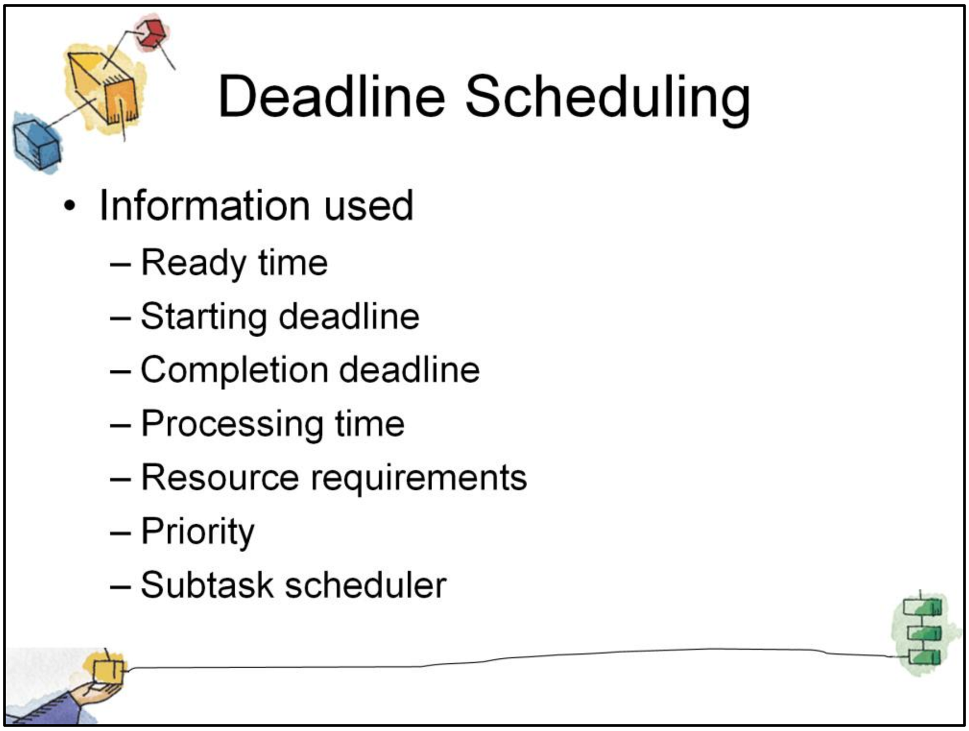


# Deadline Scheduling

- Real-time applications are not concerned with speed but with completing tasks
- “Priorities” are a crude tool and may not capture the time-critical element of the tasks

Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times, neither too early nor too late, despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults.

Therefore, priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time.



# Deadline Scheduling

- Information used
  - Ready time
  - Starting deadline
  - Completion deadline
  - Processing time
  - Resource requirements
  - Priority
  - Subtask scheduler

**Ready time:**

- Time at which task becomes ready for execution.

**Starting deadline:**

- Time by which a task ***must*** begin.

**Completion deadline:**

- Time by which task ***must*** be completed.

**Processing time:**

- Time required to execute the task to completion.

**Resource requirements:**

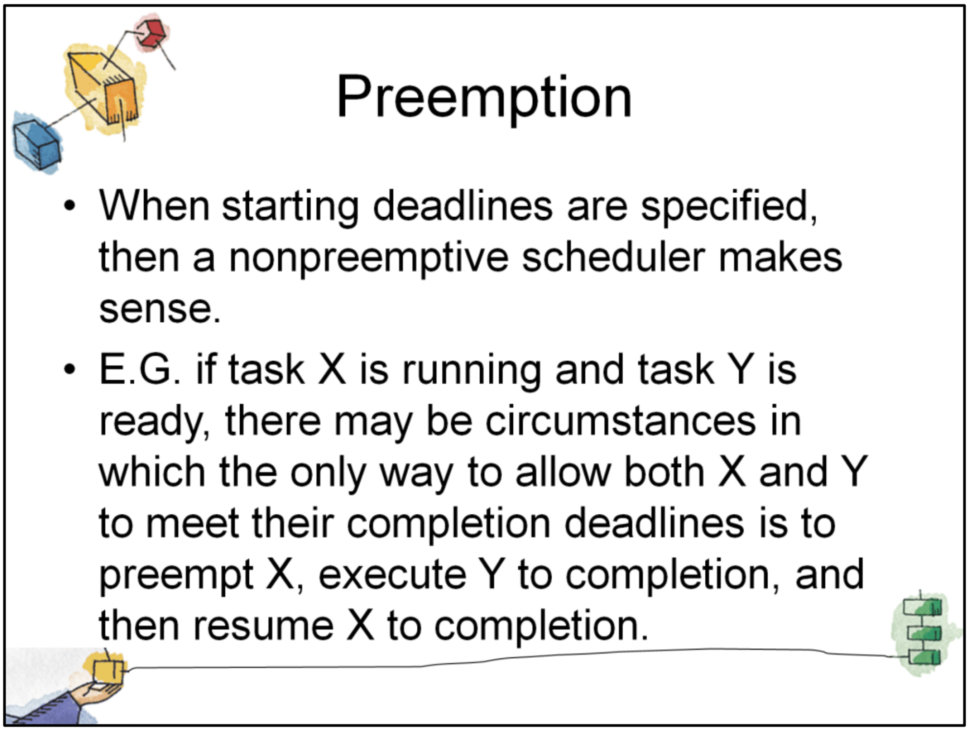
- Set of resources (other than the processor) required by the task while it is executing.

**Priority**:

- Relative importance of the task.
- Hard real-time tasks may have an “absolute” priority, with the system failing if a deadline is missed.
- If the system is to continue to run no matter what, then both hard and soft real-time tasks may be assigned relative priorities as a guide to the scheduler.

**Subtask structure:**

- A task may be decomposed into a mandatory subtask and an optional subtask.
- Only the mandatory subtask possesses a hard deadline.



## Preemption

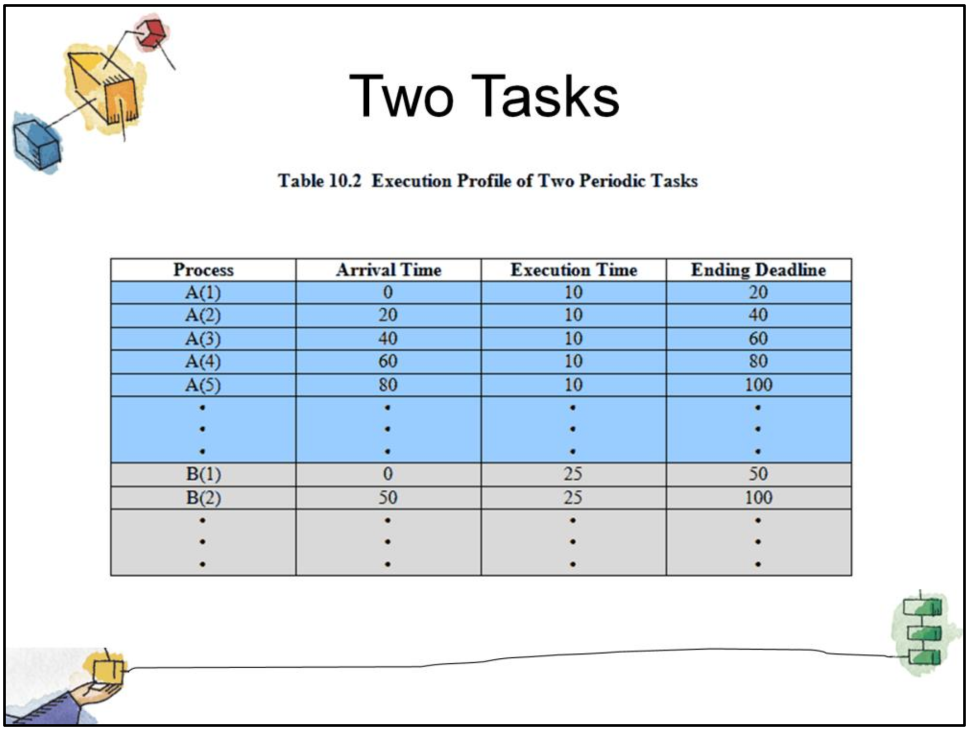
- When starting deadlines are specified, then a nonpreemptive scheduler makes sense.
- E.G. if task X is running and task Y is ready, there may be circumstances in which the only way to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, and then resume X to completion.

The other critical design issue is that of preemption.

When starting deadlines are specified, then a nonpreemptive scheduler makes sense.

•In this case, it would be the responsibility of the real-time task to block itself after completing the mandatory or critical portion of its execution, allowing other real-time starting deadlines to be satisfied.

For example, if task X is running and task Y is ready, there may be circumstances in which the only way to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, and then resume X to completion.



# Two Tasks

Table 10.2 Execution Profile of Two Periodic Tasks

| Process | Arrival Time | Execution Time | Ending Deadline |
|---|---|---|---|
| A(1) | 0 | 10 | 20 |
| A(2) | 20 | 10 | 40 |
| A(3) | 40 | 10 | 60 |
| A(4) | 60 | 10 | 80 |
| A(5) | 80 | 10 | 100 |
| •<br>•<br>• | •<br>•<br>• | •<br>•<br>• | •<br>•<br>• |
| B(1) | 0 | 25 | 50 |
| B(2) | 50 | 25 | 100 |
| •<br>•<br>• | •<br>•<br>• | •<br>•<br>• | •<br>•<br>• |

Movie icon links to animation of Periodic with Completion Deadline:
http://gaia.ecs.csus.edu/%7ezhangd/oscal/pdeadlineschedulingperiodic.html

As an example of scheduling periodic tasks with completion deadlines, consider a system that collects and processes data from two sensors,A and B.

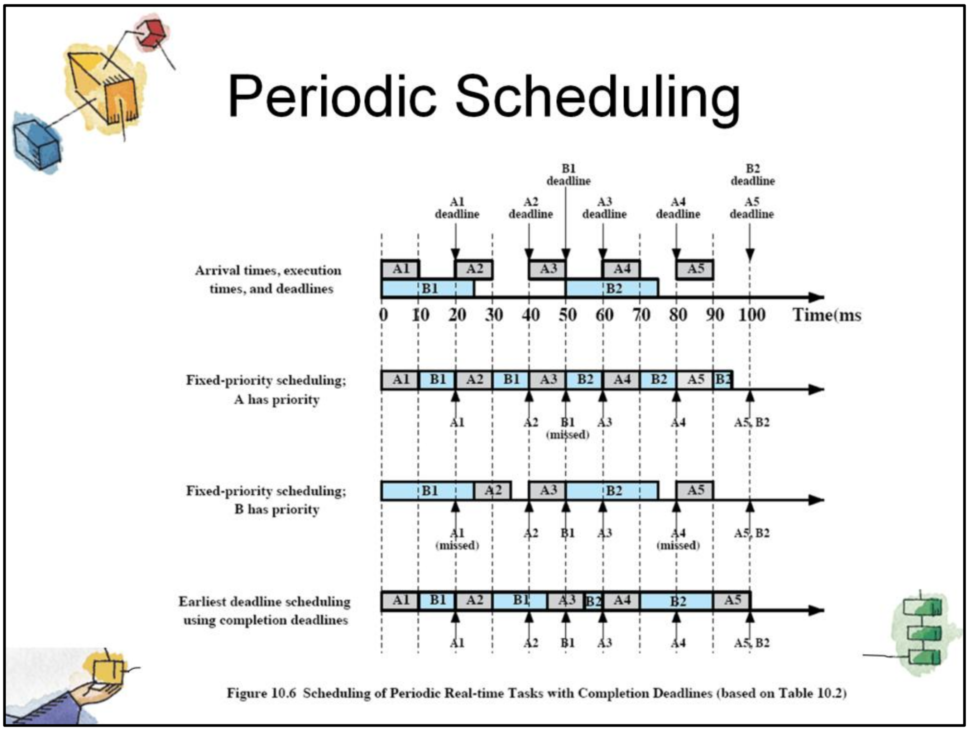
The dead-line for collecting data from

- sensor A must be met every 20 ms,
- for B every 50 ms.

It takes 10 ms, including operating system overhead, to process each sample of data from A

- and 25 ms to process each sample of data from B.

Table 10.2 summarizes the execution profile of the two tasks.



# Periodic Scheduling

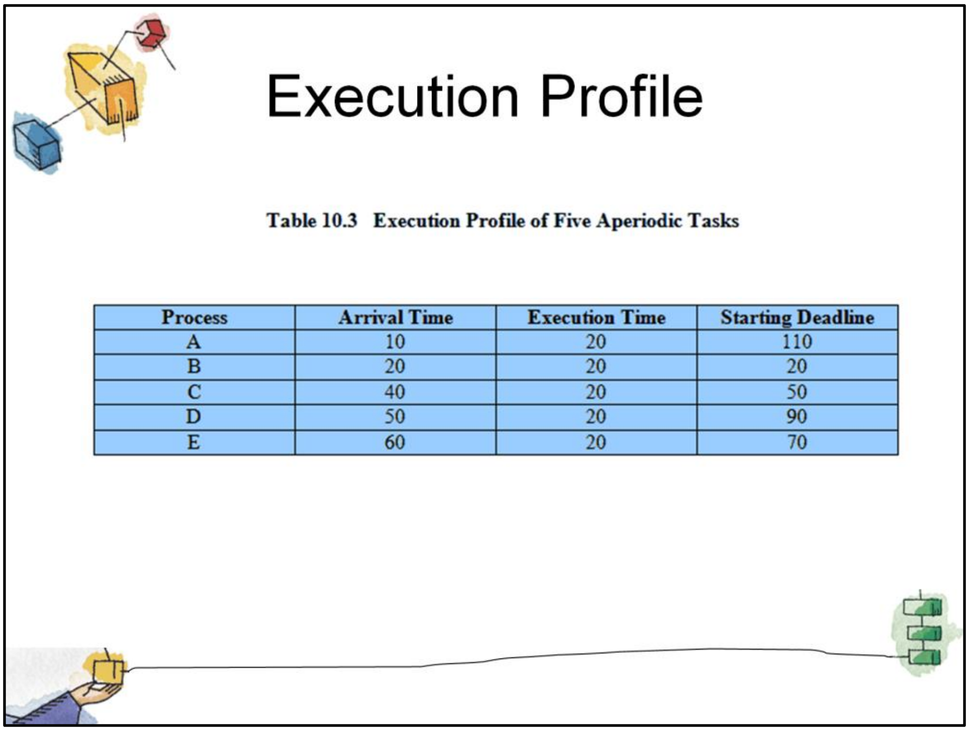
B1 deadline
B2 deadline
A1 deadline
A2 deadline
A3 deadline
A4 deadline
A5 deadline
Arrival times, execution times, and deadlines
A1
A2
A3
A4
A5
B1
B2
0
10
20
30
40
50
60
70
80
90
100
Time(ms
Fixed-priority scheduling; A has priority
A1 B1 A2 B1 A3 B2 A4 B2 A5 B2
A1
A2
B1 (missed)
A3
A4
A5, B2
Fixed-priority scheduling; B has priority
B1 A2 A3 B2 A5
A1 (missed)
A2
B1
A3
A4 (missed)
A5, B2
Earliest deadline scheduling using completion deadlines
A1 B1 A2 B1 A3 B2 A4 B2 A5
A1
A2
B1
A3
A4
A5, B2

Figure 10.6 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

Figure 10.6 compares three scheduling techniques using the execution profile of Table 10.2. (previous slide)

The first row of Figure 10.6 repeats the information in Table 10.2;

- the remaining three rows illustrate three scheduling techniques.

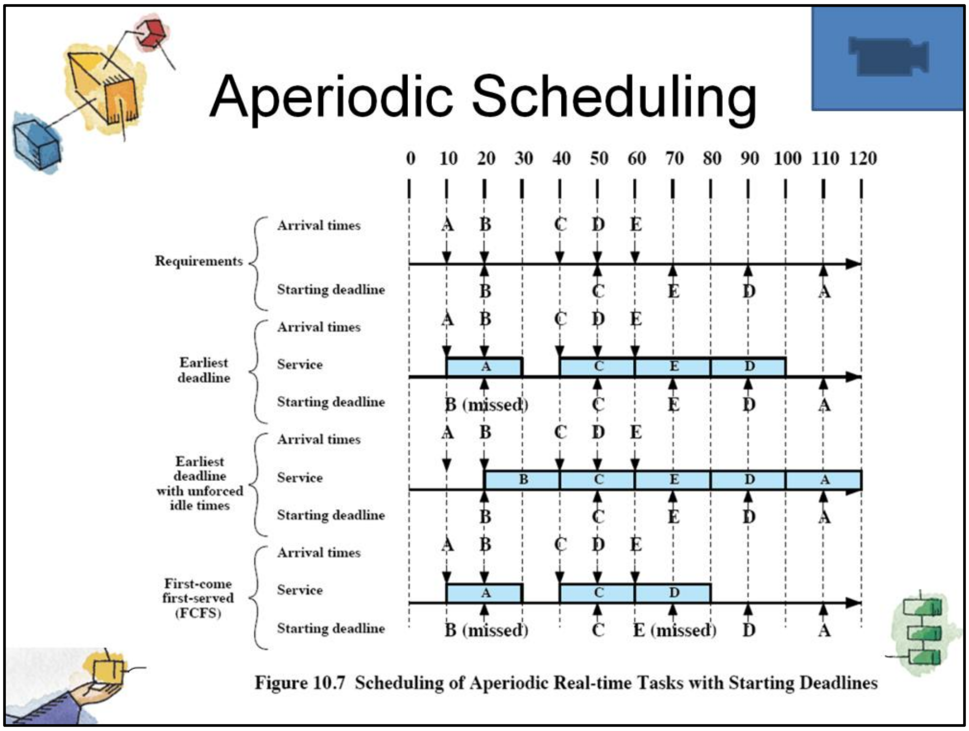


# Execution Profile

Table 10.3 Execution Profile of Five Aperiodic Tasks

| Process | Arrival Time | Execution Time | Starting Deadline |
|---|---|---|---|
| A | 10 | 20 | 110 |
| B | 20 | 20 | 20 |
| C | 40 | 20 | 50 |
| D | 50 | 20 | 90 |
| E | 60 | 20 | 70 |

Table 10.3 summarizes the execution profile of the five tasks with starting deadlines.

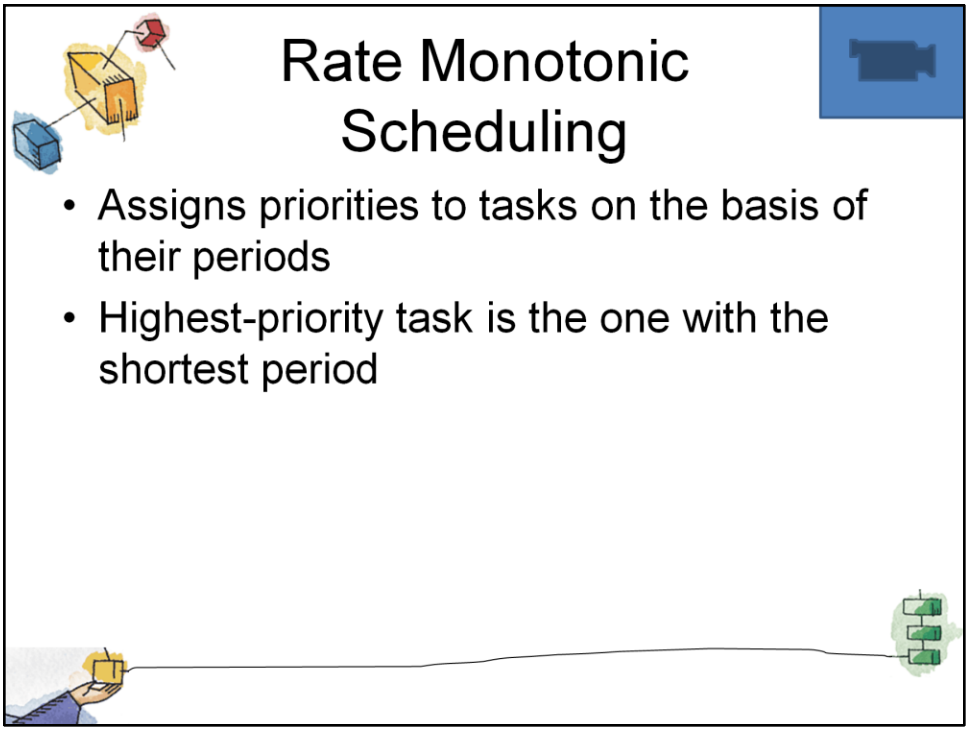


# Aperiodic Scheduling

Figure 10.7 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

Movie icon links to animation of Aperiodic with Starting Deadline:
http://gaia.ecs.csus.edu/%7ezhangd/oscal/pdeadlinescheduling.html

The top part of Figure 10.7 shows the arrival times and starting deadlines for an example consisting of five tasks each of which has an execution time of 20 ms.



## Rate Monotonic Scheduling

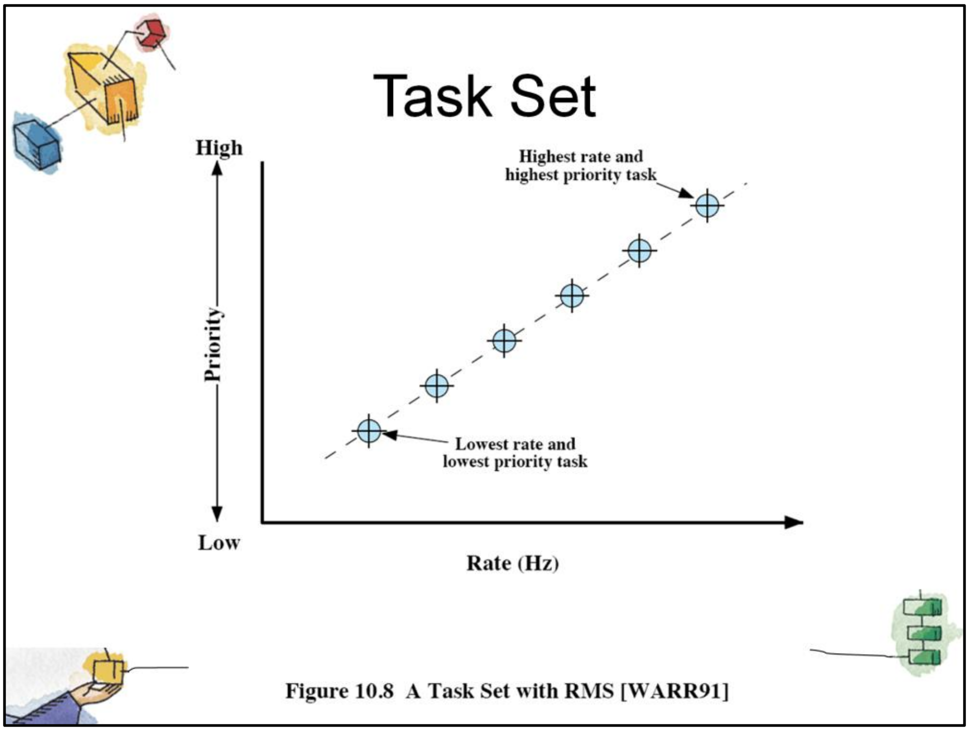
- Assigns priorities to tasks on the basis of their periods
- Highest-priority task is the one with the shortest period

Movie icon links to animation of Rate Monotonic Scheduling:
http://gaia.ecs.csus.edu/%7ezhangd/oscal/pschedulingrms.html

The highest-priority task is the one with the shortest period,

- the second highest-priority task is the one with the second shortest period, and so on.

When more than one task is available for execution, the one with the shortest period is serviced first.

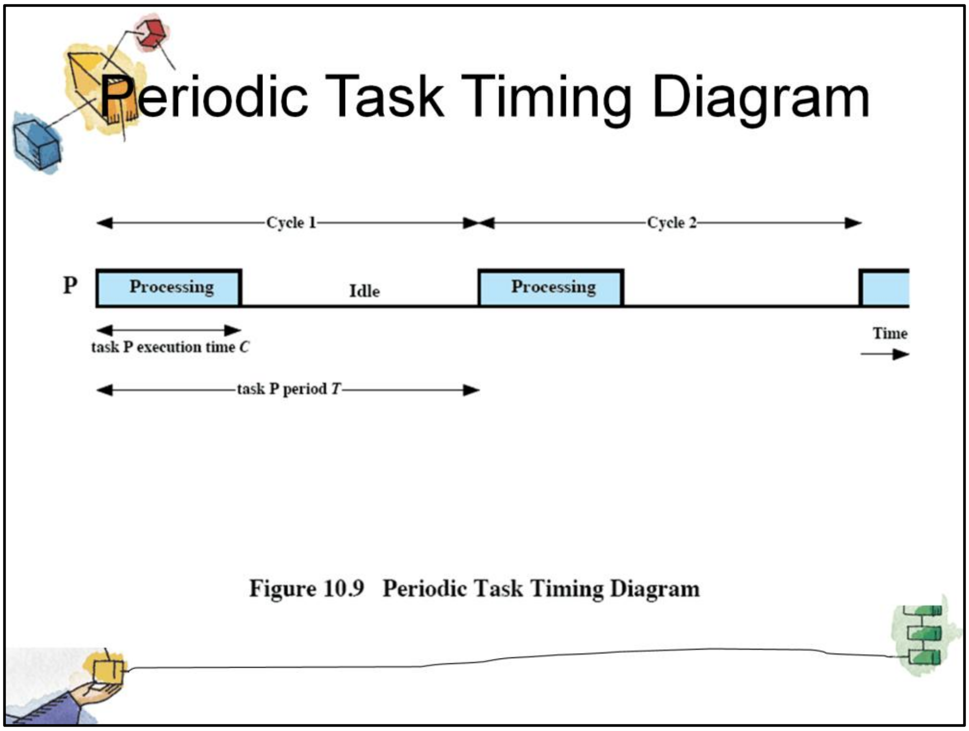


## Task Set

Figure 10.8 A Task Set with RMS [WARR91]

If we plot the priority of tasks as a function of their rate, the result is a monotonically increasing function

hence the name, **rate monotonic scheduling.**



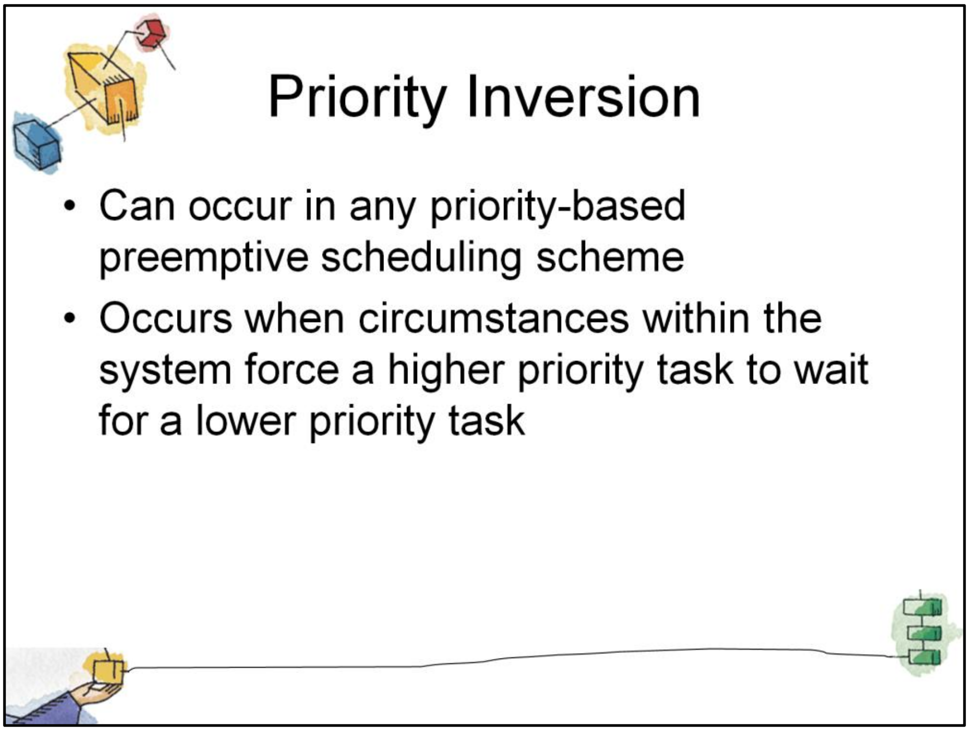
# Periodic Task Timing Diagram

Figure 10.9 Periodic Task Timing Diagram

This figure illustrates the relevant parameters for periodic tasks.

The task’s period, *T*, is the amount of time between the arrival of one instance of the task and the arrival of the next instance of the task.

A task’s rate (in Hertz) is simply the inverse of its period (in seconds).



# Priority Inversion

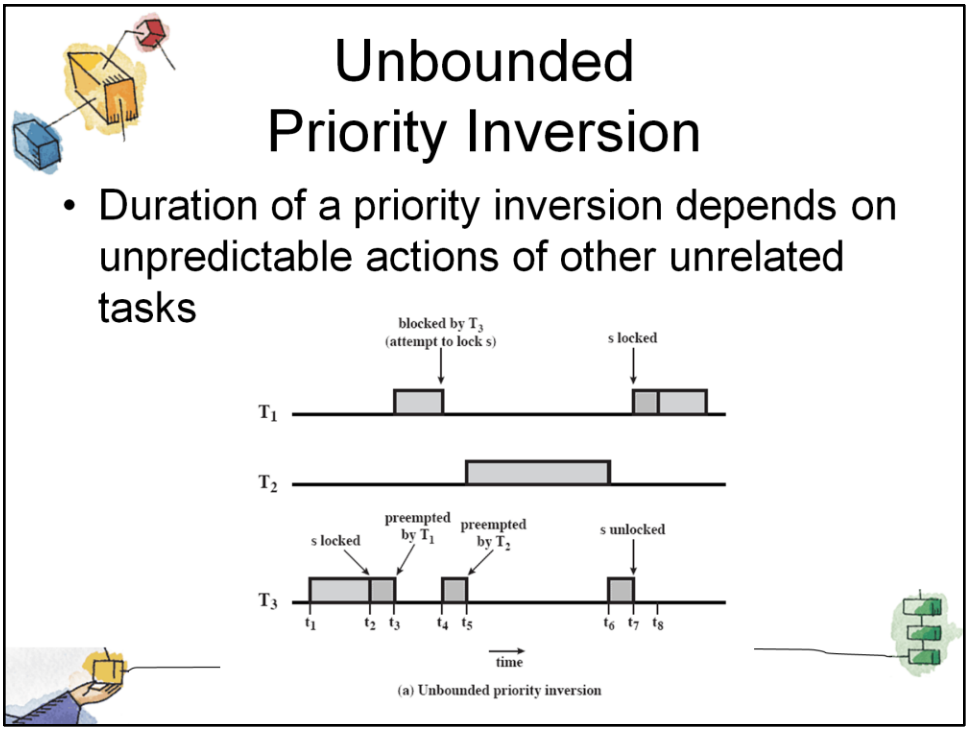
- Can occur in any priority-based preemptive scheduling scheme
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task

In any priority scheduling scheme, the system should always be executing the task with the highest priority.

**Priority inversion** occurs when circumstances within the system force a higher-priority task to wait for a lower-priority task.

A simple example of priority inversion occurs if a lower-priority task has locked a resource and a higher-priority task attempts to lock that same resource.

- The higher-priority task will be put in a blocked state until the resource is available.
- If the lower-priority task soon finishes with the resource and releases it, the higher-priority task may quickly resume and it is possible that no real-time constraints are violated.

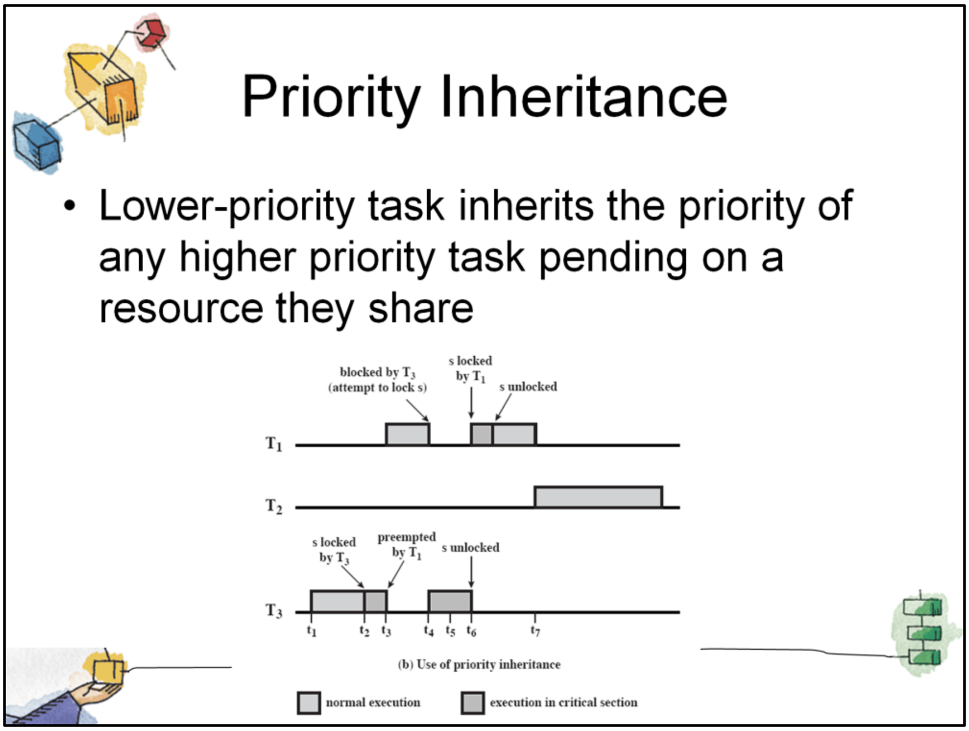


# Unbounded Priority Inversion

- Duration of a priority inversion depends on unpredictable actions of other unrelated tasks

(a) Unbounded priority inversion

A more serious condition is referred to as an unbounded priority inversion, in which the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks as well.



# Priority Inheritance

- Lower-priority task inherits the priority of any higher priority task pending on a resource they share

(b) Use of priority inheritance

The basic idea of priority inheritance is that a lower-priority task inherits the priority of any higher-priority task pending on a resource they share.

This priority change takes place as soon as the higher-priority task blocks on the resource;

- it should end when the resource is released by the lower-priority task.

This figure shows that priority inheritance resolves the problem of unbounded priority inversion illustrated in Figure 10.10a.

The relevant sequence of events is as follows:

$t_1$: $T_3$ begins executing.

$t_2$: $T_3$ locks semaphore s and enters its critical section.

$t_3$: $T_1$, which has a higher priority than $T_3$, preempts $T_3$ and begins executing.

$t_4$: $T_1$ attempts to enter its critical section but is blocked because the semaphore is locked by $T_3$.

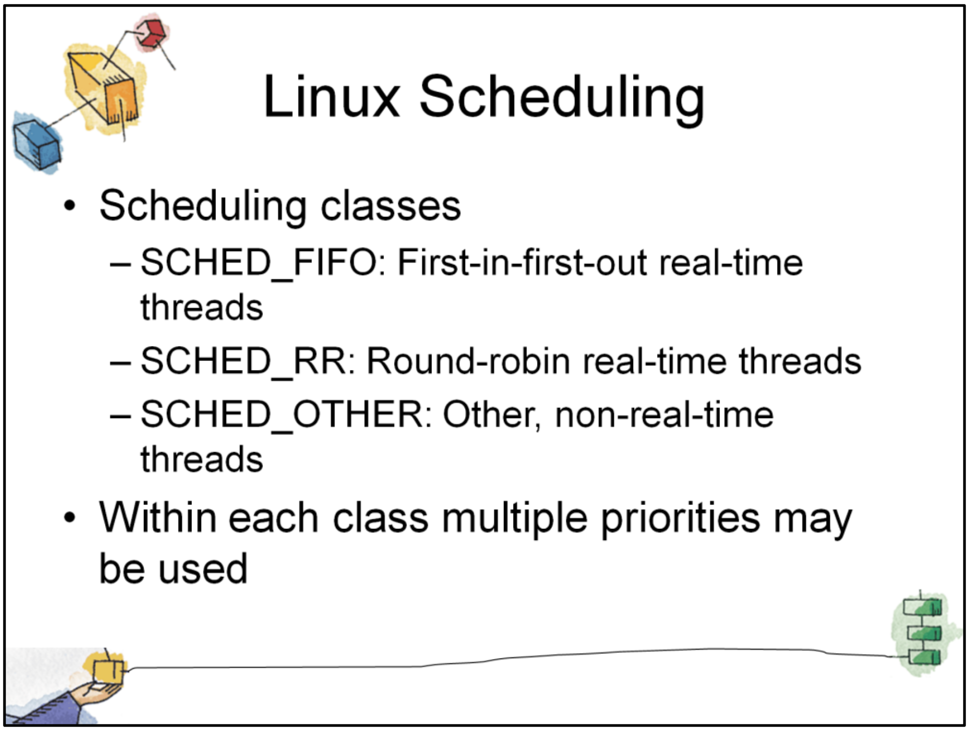
- $T_3$ is immediately and temporarily assigned the same priority as $T_1$.
- $T_3$ resumes execution in its critical section.

$t_5$: $T_2$ is ready to execute but, because $T_3$ now has a higher priority,$T_2$ is unable to preempt $T_3$.

$t_6$: $T_3$ leaves its critical section and unlocks the semaphore: its priority level is downgraded to its previous default level.

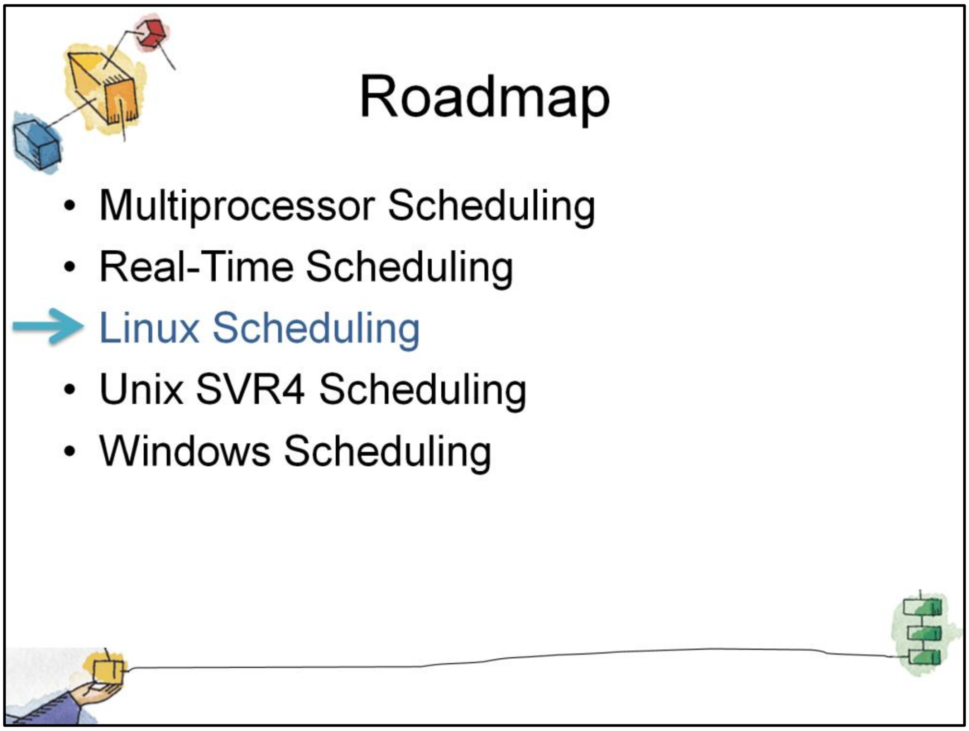
- $T_1$ preempts $T_3$, locks the semaphore, and enters its critical section.

$t_7$: $T_1$ is suspended for some reason unrelated to $T_2$, and $T_2$ begins executing.



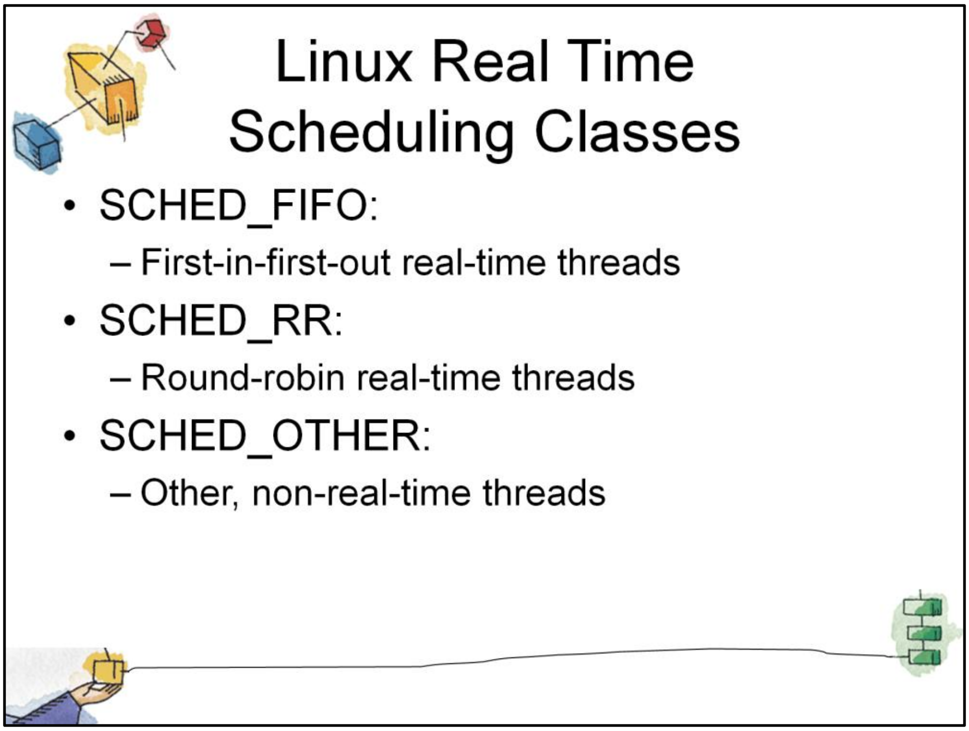
# Linux Scheduling

- Scheduling classes
  - SCHED_FIFO: First-in-first-out real-time threads
  - SCHED_RR: Round-robin real-time threads
  - SCHED_OTHER: Other, non-real-time threads
- Within each class multiple priorities may be used



# Roadmap

- Multiprocessor Scheduling
- Real-Time Scheduling
- → Linux Scheduling
- Unix SVR4 Scheduling
- Windows Scheduling



# Linux Real Time Scheduling Classes

- SCHED_FIFO:
  - First-in-first-out real-time threads
- SCHED_RR:
  - Round-robin real-time threads
- SCHED_OTHER:
  - Other, non-real-time threads

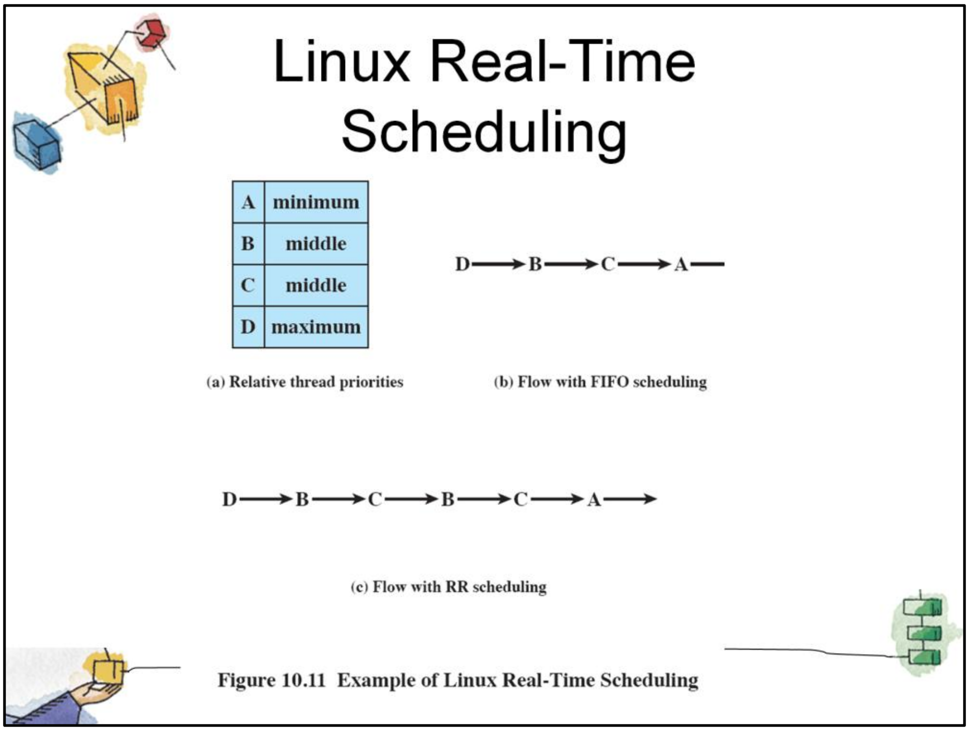
The three Linux scheduling classes are

- SCHED_FIFO: First-in-first-out real-time threads
- SCHED_RR: Round-robin real-time threads
- SCHED_OTHER: Other, non-real-time threads

Within each class, multiple priorities may be used, with priorities in the real-time classes higher than the priorities for the SCHED_OTHER class.

The default values are as follows:

- Real-time priority classes range from 0 to 99 inclusively,
- and SCHED_OTHER classes range from 100 to 139.
- A lower number equals a higher priority.



# Linux Real-Time Scheduling

| | |
|---|---|
| A | minimum |
| B | middle |
| C | middle |
| D | maximum |

(a) Relative thread priorities

D ⟶ B ⟶ C ⟶ A —

(b) Flow with FIFO scheduling

D ⟶ B ⟶ C ⟶ B ⟶ C ⟶ A ⟶

(c) Flow with RR scheduling

Figure 10.11 Example of Linux Real-Time Scheduling

Figure 10.11 is an example that illustrates the distinction between FIFO and RR scheduling.

Assume a process has four threads with three relative priorities assigned as shown in Figure 10.11a.

- Assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher-priority thread is awakened while a thread is executing.

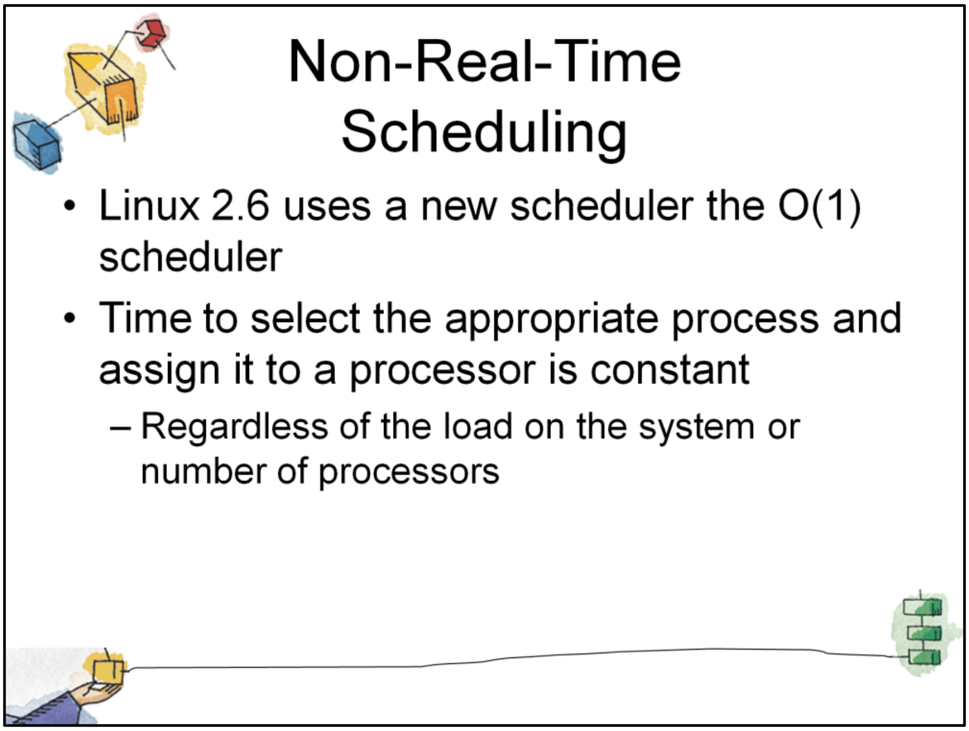
Figure 10.11b shows a flow in which all of the threads are in the SCHED_FIFO class.

- Thread D executes until it waits or terminates.
- Next, although threads B and C have the same priority, thread B starts because it has been waiting longer than thread C.
- Thread B executes until it waits or terminates, then thread C executes until it waits or terminates.
- Finally, thread A executes.

Figure 10.11c shows a sample flow if all of the threads are in the SCHED_RR class.

- Thread D executes until it waits or terminates.
- Next, threads B and C are time sliced, because they both have the same priority.
- Finally, thread A executes.

The final scheduling class is SCHED_OTHER. A thread in this class can only execute if there are no real-time threads ready to execute.

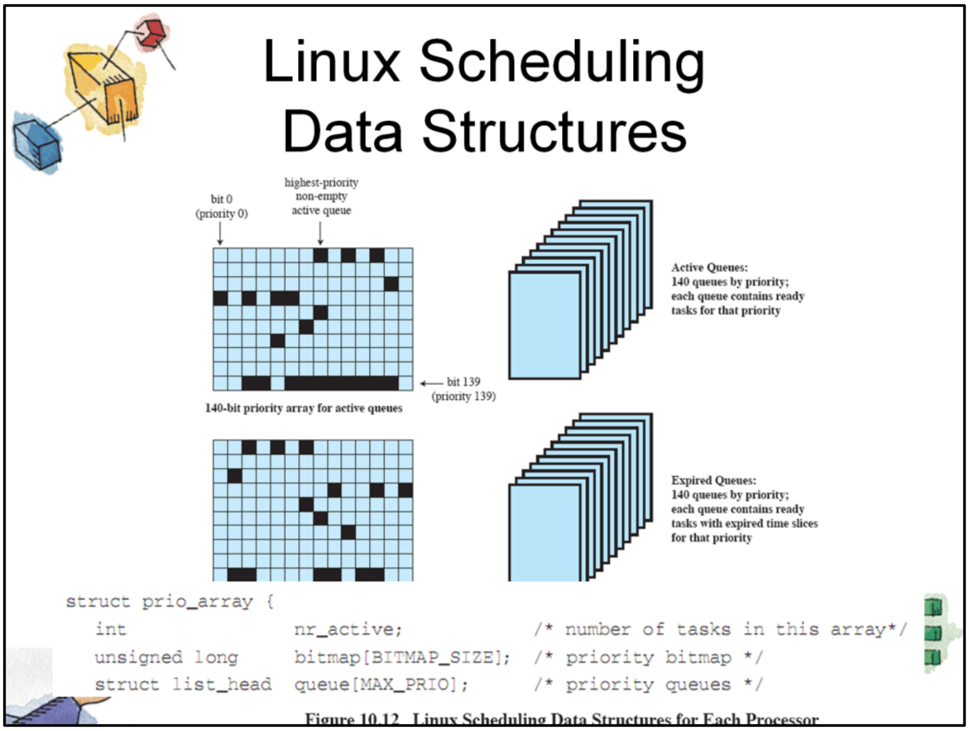


## Non-Real-Time Scheduling

- Linux 2.6 uses a new scheduler the O(1) scheduler
- Time to select the appropriate process and assign it to a processor is constant
  - Regardless of the load on the system or number of processors

Linux 2.6 uses a completely new priority scheduler known as the O(1) scheduler.

The scheduler is designed so that the time to select the appropriate process and assign it to a processor is constant, regardless of the load on the system or the number of processors.



# Linux Scheduling Data Structures

```
struct prio_array {
   int               nr_active;          /* number of tasks in this array*/
   unsigned long     bitmap[BITMAP_SIZE]; /* priority bitmap */
   struct list_head  queue[MAX_PRIO];    /* priority queues */
```

Figure 10.12 Linux Scheduling Data Structures for Each Processor

The kernel maintains two scheduling data structure for each processor in the system

A separate queue is maintained for each priority level.

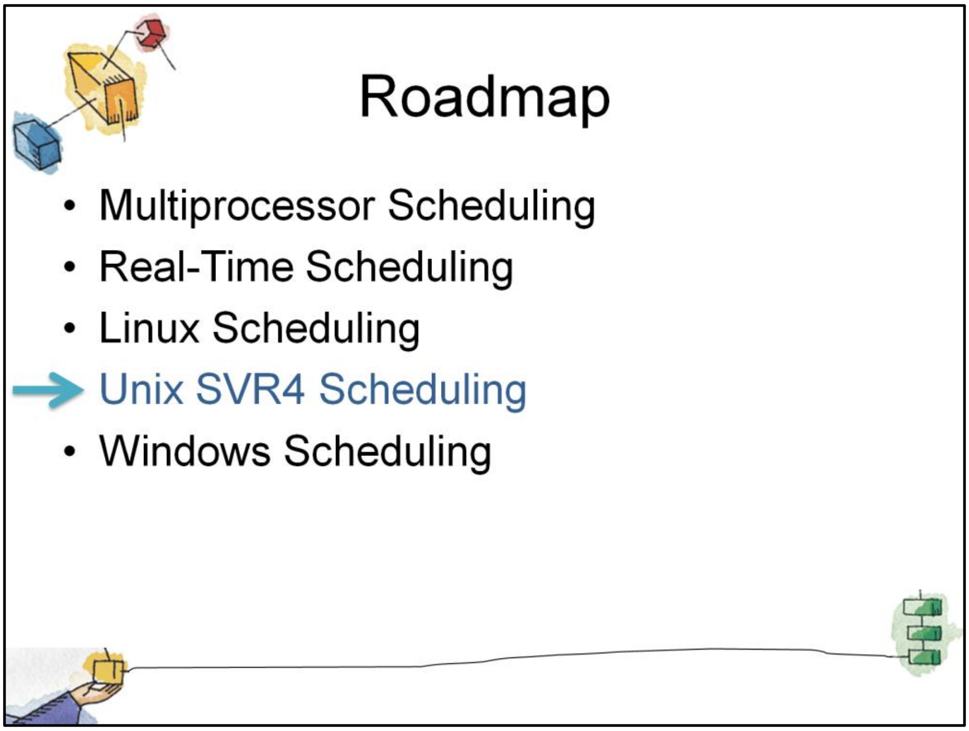
- The total number of queues in the structure is MAX_PRIO,which has a default value of 140.

The structure also includes a bitmap array of sufficient size to provide one bit per priority level.

- With 140 priority levels and 32-bit words, BITMAP_SIZE has a value of 5.
- This creates a bitmap of 160 bits, of which 20 bits are ignored.
- The bitmap indicates which queues are not empty.

Finally, nr_active indicates the total number of tasks present on all queues.

- Two structures are maintained: an active queues structure and an expired queues structure.

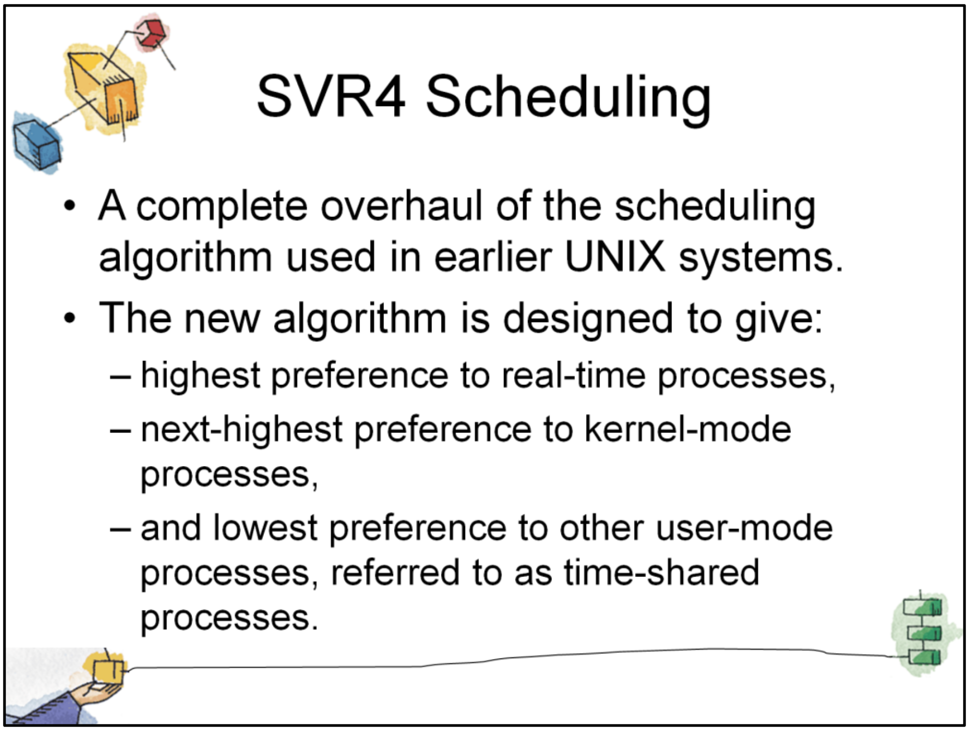


# Roadmap

- Multiprocessor Scheduling
- Real-Time Scheduling
- Linux Scheduling

→ Unix SVR4 Scheduling

- Windows Scheduling

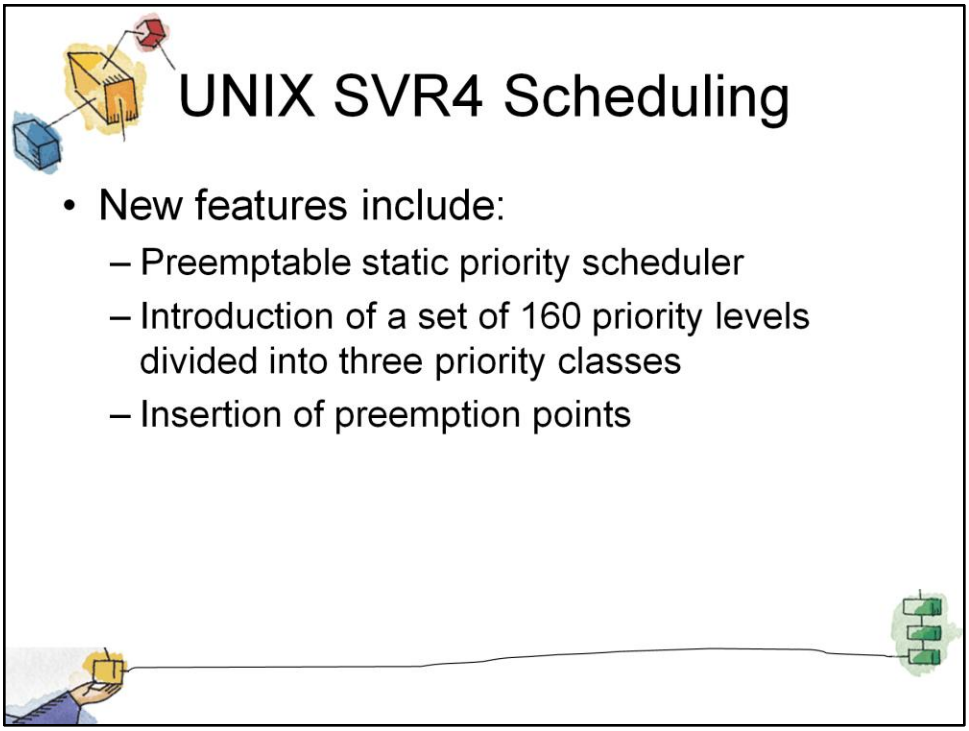


# SVR4 Scheduling

- A complete overhaul of the scheduling algorithm used in earlier UNIX systems.
- The new algorithm is designed to give:
  - highest preference to real-time processes,
  - next-highest preference to kernel-mode processes,
  - and lowest preference to other user-mode processes, referred to as time-shared processes.

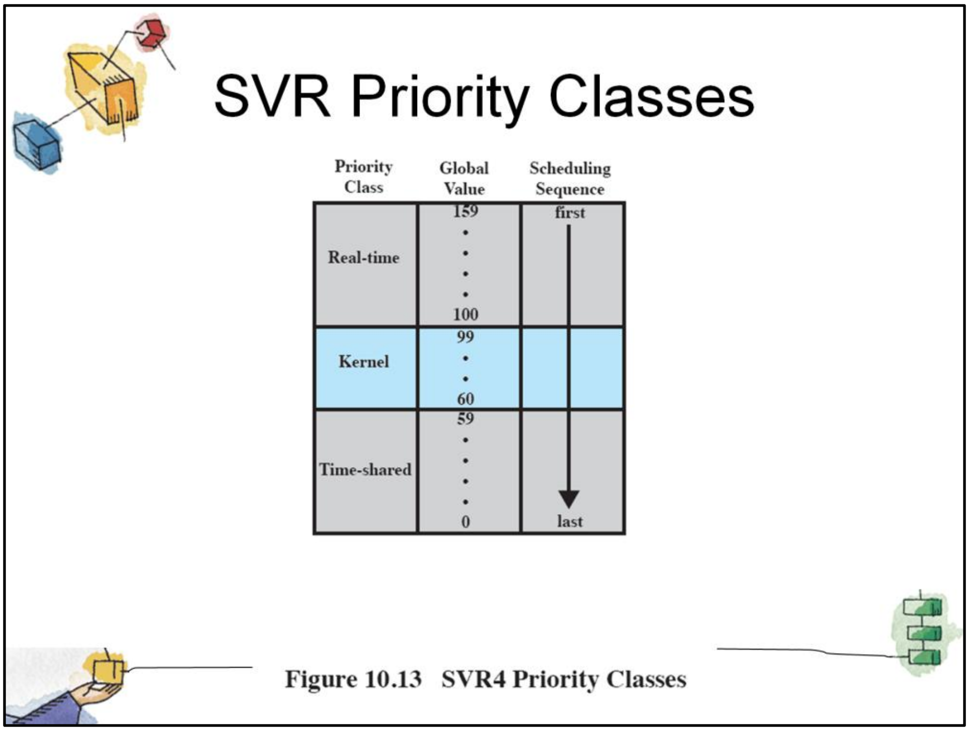
The scheduling algorithm used in UNIX SVR4 is a complete overhaul of the scheduling algorithm used in earlier UNIX systems.

The new algorithm is designed to give highest preference to real-time processes, next-highest preference to kernel-mode processes, and lowest preference to other user-mode processes, referred to as time-shared processes.



# UNIX SVR4 Scheduling

- New features include:
  - Preemptable static priority scheduler
  - Introduction of a set of 160 priority levels divided into three priority classes
  - Insertion of preemption points



# SVR Priority Classes

| Priority Class | Global Value | Scheduling Sequence |
|---|---|---|
| Real-time | 159<br>•<br>•<br>•<br>•<br>100 | first |
| Kernel | 99<br>•<br>•<br>60 | |
| Time-shared | 59<br>•<br>•<br>•<br>•<br>0 | last |

Figure 10.13 SVR4 Priority Classes

Figure 10.13 illustrates the 160 priority levels defined in SVR4.

Each process is defined to belong to one of three priority classes and is assigned a priority level within that class.

The classes are

**Real time** (159–100):

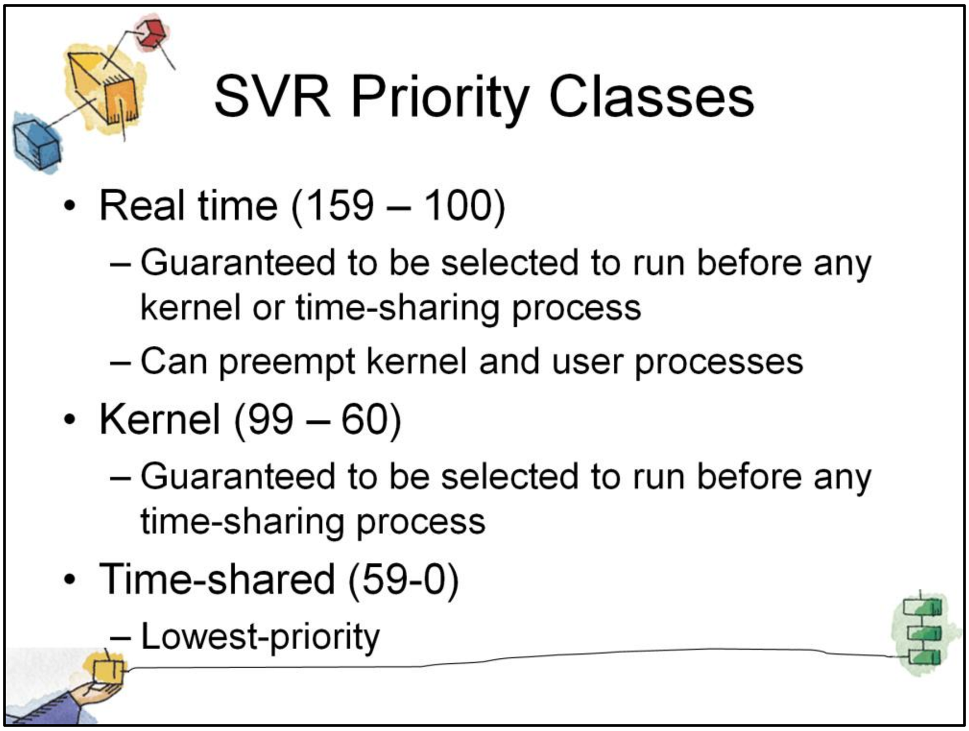
- Processes at these priority levels are guaranteed to be selected to run before any kernel or time-sharing process.
- In addition, real-time processes can make use of preemption points to preempt kernel processes and user processes.

**Kernel** (99–60):

- Processes at these priority levels are guaranteed to be selected to run before any time-sharing process but must defer to real-time processes.

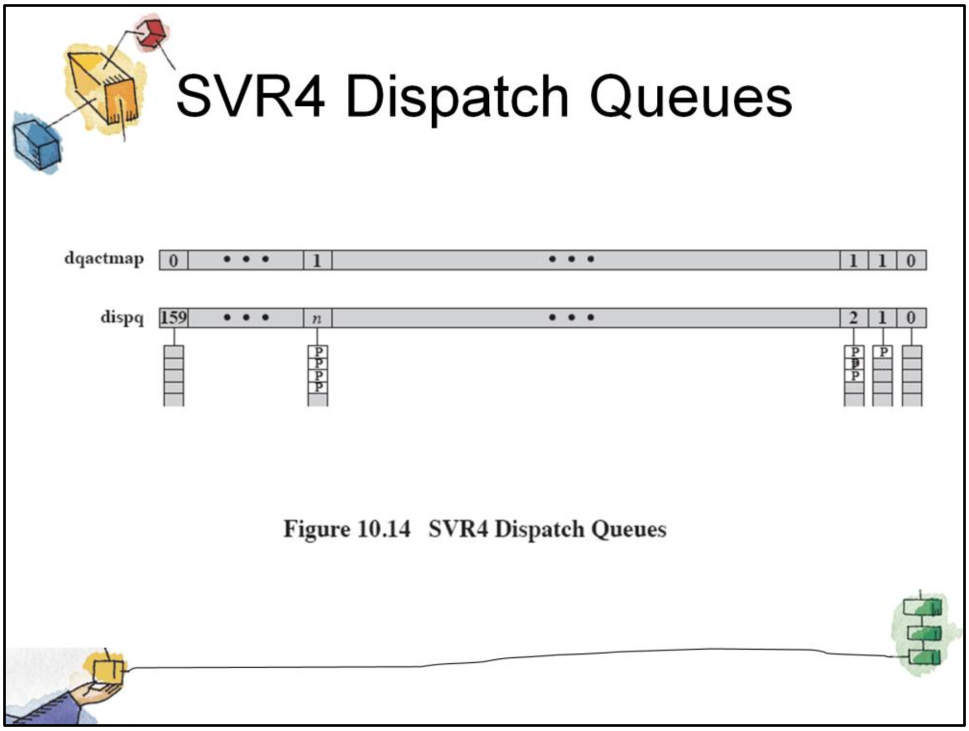
**Time-shared** (59–0):

The lowest-priority processes, intended for user applications other than real-time applications.



# SVR Priority Classes

- Real time (159 – 100)
  - Guaranteed to be selected to run before any kernel or time-sharing process
  - Can preempt kernel and user processes
- Kernel (99 – 60)
  - Guaranteed to be selected to run before any time-sharing process
- Time-shared (59-0)
  - Lowest-priority



## SVR4 Dispatch Queues

Figure 10.14 SVR4 Dispatch Queues

Figure 10.14 indicates how scheduling is implemented in SVR4.

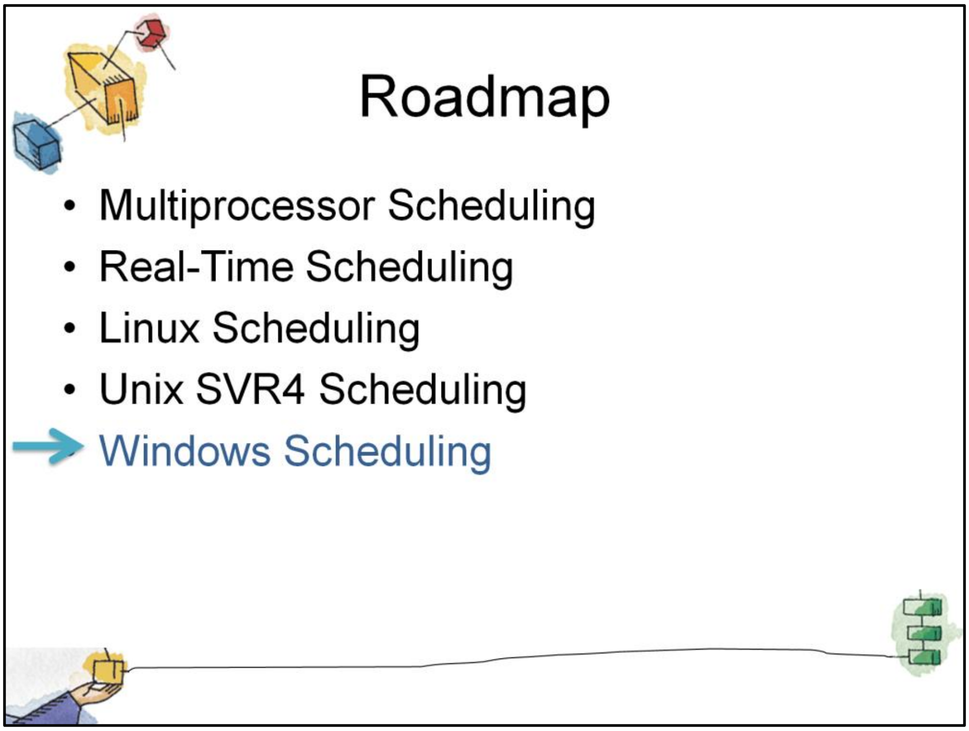
A dispatch queue is associated with each priority level, and processes at a given priority level are executed in round-robin fashion.

A bit-map vector, dqactmap, contains one bit for each priority level;

- the bit is set to one for any priority level with a nonempty queue.
- Whenever a running process leaves the Running state, due to a block, times-slice expiration, or preemption, the dispatcher checks dqactmap and dispatches a ready process from the highest-priority nonempty queue.
- In addition, whenever a defined preemption point is reached, the kernel checks a flag called kprunrun.
- If set, this indicates that at least one real-time process is in the Ready state, and the kernel preempts the current process if it is of lower priority than the highest-priority real-time ready process.

Within the time-sharing class, the priority of a process is variable.

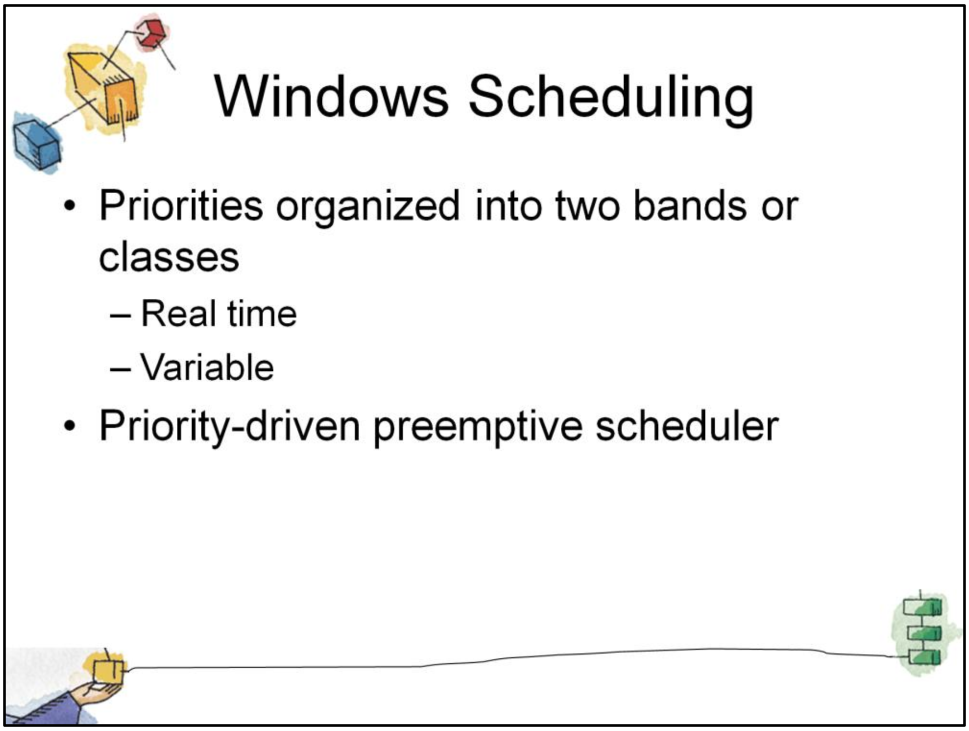
- The scheduler reduces the priority of a process each time it uses up a time quantum, and it raises its priority if it blocks on an event or resource.
- The time quantum allocated to a time-sharing process depends on its priority, ranging from 100 ms for priority 0 to 10 ms for priority 59.
- Each real-time process has a fixed priority and a fixed time quantum.



# Roadmap

- Multiprocessor Scheduling
- Real-Time Scheduling
- Linux Scheduling
- Unix SVR4 Scheduling

→ Windows Scheduling



## Windows Scheduling

- Priorities organized into two bands or classes
  - Real time
  - Variable
- Priority-driven preemptive scheduler

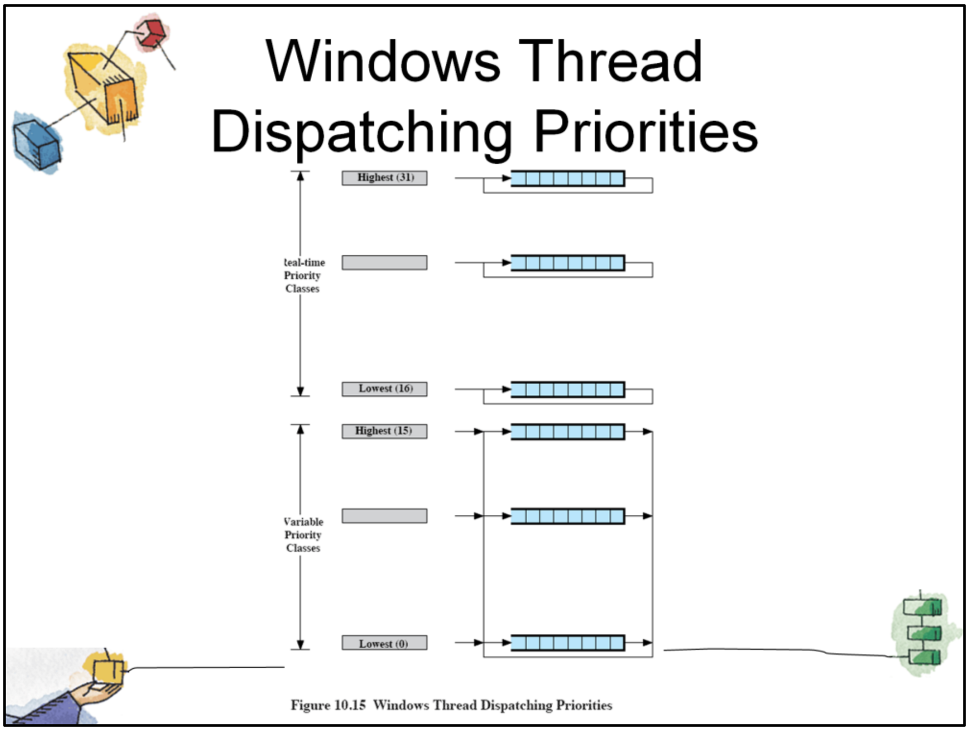
Priorities in Windows are organized into two bands, or classes:

- real time and variable.
- Each of these bands consists of 16 priority levels.

Threads requiring immediate attention are in the real-time class, which includes functions such as communications and real-time tasks.

Overall, because Windows makes use of a priority-driven preemptive scheduler, threads with real-time priorities have precedence over other threads.

On a uniprocessor, when a thread becomes ready whose priority is higher than the currently executing thread, the lower-priority thread is preempted and the processor given to the higher-priority thread.



## Windows Thread Dispatching Priorities

Figure 10.15 Windows Thread Dispatching Priorities

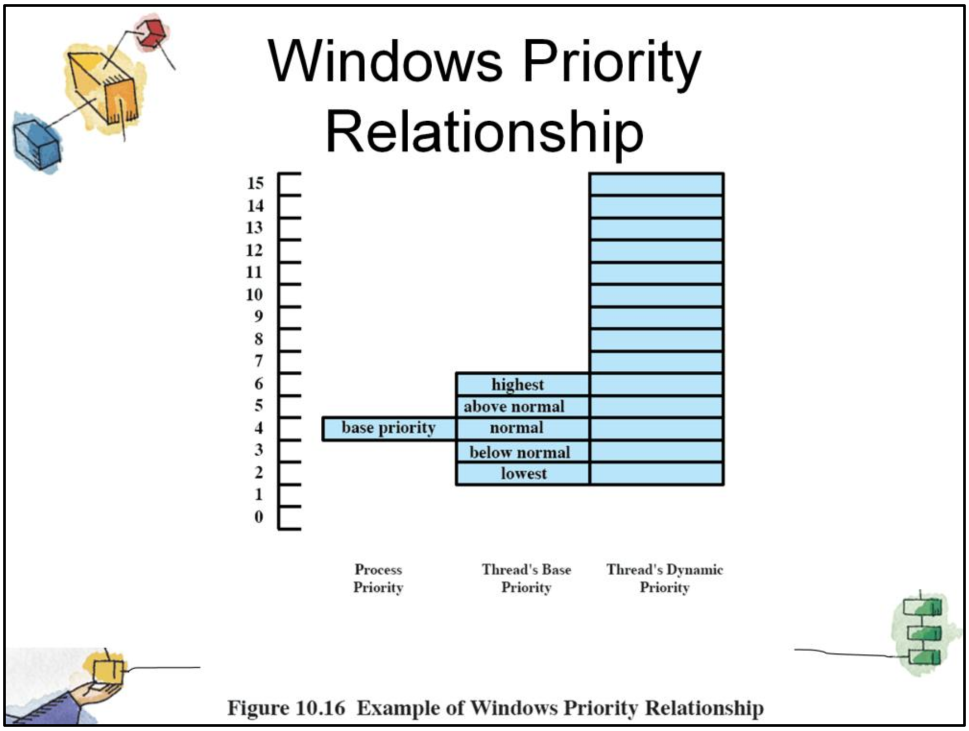
Priorities are handled somewhat differently in the two classes

In the **real-time** priority class, all threads have a fixed priority that never changes.

- All of the active threads at a given priority level are in a round-robin queue.

In the **variable** priority class, a thread's priority begins at some initial assigned value and then may be temporarily boosted (raised) during the thread's lifetime.

- There is a FIFO queue at each priority level;
- a thread will change queues among the variable priority classes as its own priority changes.
- However, a thread at priority level 15 or below is never boosted to level 16 or any other level in the real-time class.



## Windows Priority Relationship

Figure 10.16 Example of Windows Priority Relationship

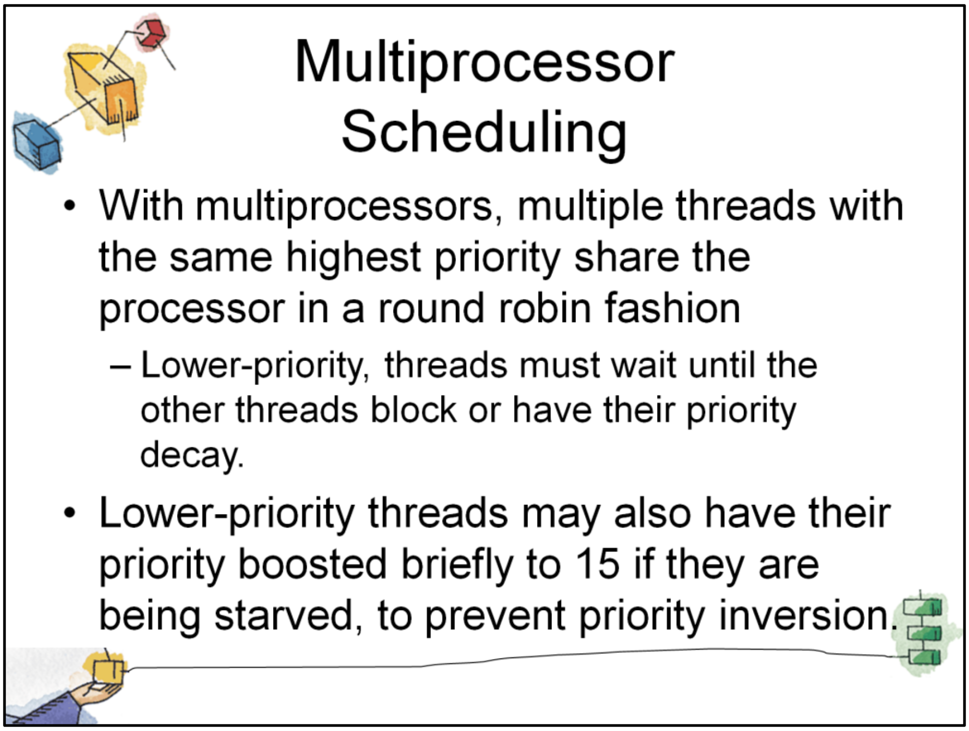
Once a thread in the variable priority class has been activated, its actual priority, referred to as the thread's current priority, may fluctuate within given boundaries.

- The current priority may never fall below the thread's base priority and it may never exceed 15.

Figure 10.16 gives an example.

The process object has a base priority attribute of 4.

- Each thread object associated with this process object must have an initial priority of between 2 and 6.
- Suppose the base priority for thread is 4.
- Then the current priority for that thread may fluctuate in the range from 4 through 15 depending on what boosts it has been given.
- If a thread is interrupted to wait on an I/O event, the Windows Kernel boosts its priority.
- If a boosted thread is interrupted because it has used up its current time quantum, the Kernel lowers its priority.
- Thus, processor-bound threads tend toward lower priorities and I/O-bound threads tend toward higher priorities. In the case of I/O-bound threads, the Kernel boosts the priority more for interactive waits (e.g., wait on keyboard or display) than for other types of I/O (e.g., disk I/O).
- Thus, interactive threads tend to have the highest priorities within the variable priority class.



## Multiprocessor Scheduling

- With multiprocessors, multiple threads with the same highest priority share the processor in a round robin fashion
  - Lower-priority, threads must wait until the other threads block or have their priority decay.
- Lower-priority threads may also have their priority boosted briefly to 15 if they are being starved, to prevent priority inversion.

When Windows is run on a single processor, the highest-priority thread is always active unless it is waiting on an event.

If there is more than one thread that has the same highest priority, then the processor is shared, round robin, among all the threads at that priority level.

In a multiprocessor system with N processors, the Kernel tries to give the N processors to the N highest priority threads that are ready to run.

The remaining, lower-priority, threads must wait until the other threads block or have their priority decay.

Lower-priority threads may also have their priority boosted to 15 for a very short time if they are being starved, solely to correct instances of priority inversion.