

*Operating  
Systems:  
Internals  
and  
Design  
Principles*

# Chapter 10 Multiprocessor, Multicore and Real-Time Scheduling

Ninth Edition  
By William Stallings

# Classifications of Multiprocessor Systems

## 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

- There is a master, general-purpose processor;
- Specialized processors are controlled by the master processor and provide services to it

## Tightly coupled multiprocessor

- Consists of a set of processors that share a common main memory and are under the integrated control of an operating system



# Granularity

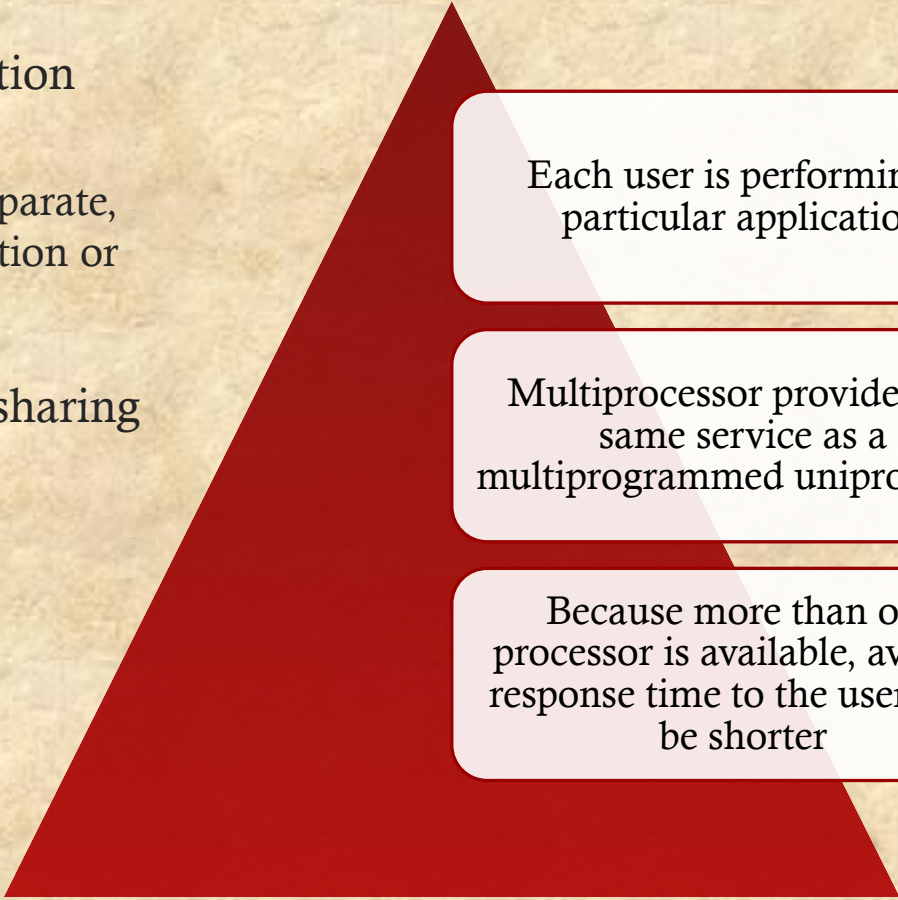
- A good way of characterizing multiprocessors and placing them in context with other architectures is to consider the synchronization granularity, or frequency of synchronization, between processes in a system.
- We will look at five distinct categories.
  - Fine
  - Medium
  - Coarse
  - Very Coarse
  - Independent

<b>Grain Size</b>	<b>Description</b>	<b>Synchronization Interval (Instructions)</b>
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

**Table 10.1 Synchronization Granularity and Processes**

# Independent Parallelism

- No explicit synchronization among processes
  - Each represents a separate, independent application or job
- Typical use is in a time-sharing system
- VMs



Each user is performing a particular application

Multiprocessor provides the same service as a multiprogrammed uniprocessor

Because more than one processor is available, average response time to the users will be shorter



# Coarse and Very Coarse-Grained Parallelism

- There is synchronization among processes, but at a very gross level
- Easily handled as a set of concurrent processes running on a multiprogrammed uniprocessor
- Can be supported on a multiprocessor with little or no change to user software
- Coarse grained: Distributed database queries
- Very coarse grained: Distributed rendering, Independent batch jobs.

# Medium-Grained Parallelism

- Single application can be effectively implemented as a collection of threads within a single process
  - Programmer must explicitly specify the potential parallelism of an application
  - There needs to be 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
- Whereas independent, very coarse, and coarse-grained parallelism can be supported on either a multiprogrammed uniprocessor or a multiprocessor with little or no impact on the scheduling function, we need to reexamine scheduling when dealing with the scheduling of threads



# Fine-Grained Parallelism

- Represents a much more complex use of parallelism than is found in the use of threads
- Is a specialized and fragmented area with many different approaches.
  - Vector Processing and SIMD Operations (Ex: AVX instructions)
  - Loop Unrolling in Parallel Processing
- Used in
  - Scientific computing
  - Graphics processing
  - Machine learning



<b>Grain Size</b>	<b>Description</b>	<b>Synchronization Interval (Instructions)</b>
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

**Table 10.1 Synchronization Granularity and Processes**

# Design Issues

Scheduling on a multiprocessor involves three interrelated issues:

- The approach taken will depend on the degree of granularity of applications and on the number of processors available

Actual  
dispatching  
of a process

Use of  
multiprogramming on  
individual processors

Assignment of  
processes to  
processors

# Assignment of Processes to Processors

Assuming all processors are equal, it is simplest to treat processors as a pooled resource and assign processes to processors on demand

Static or dynamic needs to be determined

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

Advantage is that there may be less overhead in the scheduling function

Allows group or gang scheduling

- A disadvantage of static assignment 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
  - Another option is dynamic load balancing



# Assignment of Processes to Processors

- Both dynamic and static methods require some way of assigning a process to a processor
- Approaches:
  - Master/Slave
  - Peer

# Master/Slave Architecture

- Key kernel functions always run on a particular processor
- Master is responsible for scheduling
- Slave sends service request to the master
- Is simple and requires little enhancement to a uniprocessor multiprogramming operating system
- Conflict resolution is simplified because one processor has control of all memory and I/O resources

## Disadvantages:

- Failure of master brings down whole system
- Master can become a performance bottleneck

# Peer Architecture

- Kernel can execute on any processor
- Each processor does self-scheduling from the pool of available processes

## Complicates the operating system

- Operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue

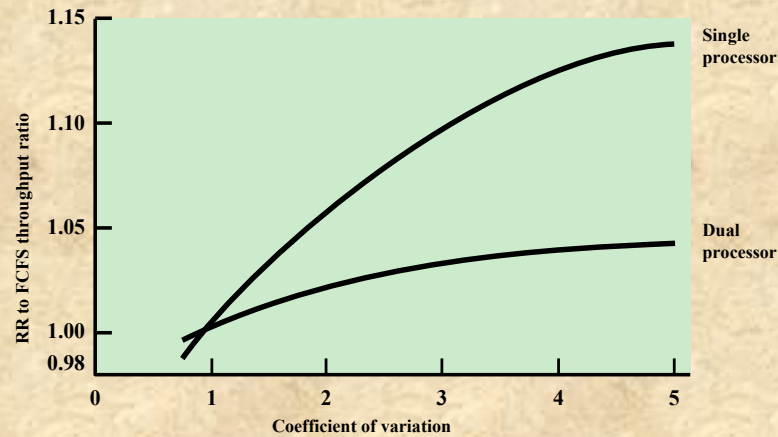


# Process Dispatching

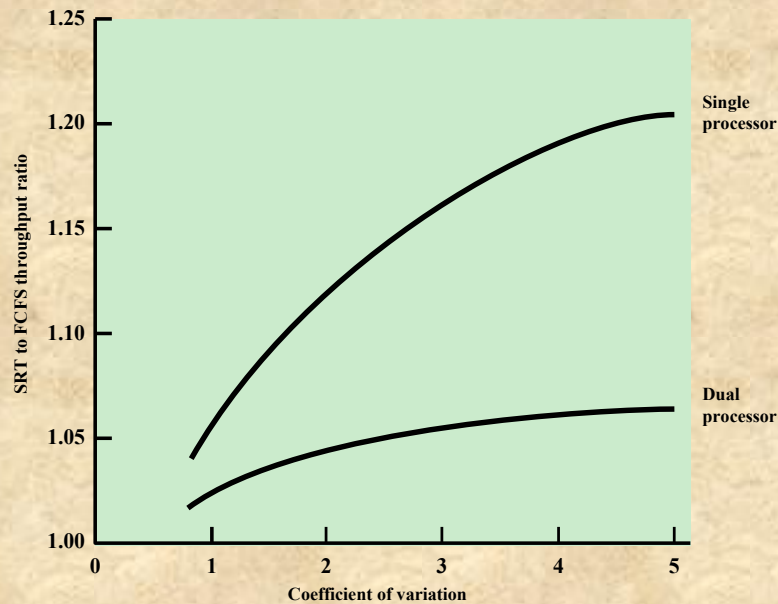
- The design issue related to multiprocessor scheduling is the actual selection of a process to run.
- On a multiprogrammed uniprocessor, the use of priorities or of sophisticated scheduling algorithms based on past usage may improve performance over a simple-minded first-come-first-served strategy.
- When we consider multiprocessors, these complexities may be unnecessary or even counterproductive, and a simpler approach may be more effective with less overhead.
- Recap on Uniprocessor scheduling algorithms: FCFS, Round Robin, SPN, SRT, HRRN, Feedback

# Process Scheduling

- In most traditional multiprocessor systems, processes are not dedicated to processors
- A single queue is used for all processors
  - If some sort of priority scheme is used, there are multiple queues based on priority, all feeding into the common pool of processors
- System is viewed as being a multi-server queuing architecture.
- On a multiprogrammed uniprocessor, the use of priorities or of sophisticated scheduling algorithms based on past usage may improve performance over a simple-minded first-come-first-served strategy.
- When we consider multiprocessors, these complexities may be unnecessary or even counterproductive, and a simpler approach may be more effective with less overhead.



(a) Comparison of RR and FCFS



(b) Comparison of SRT and FCFS

Figure 10.1 Comparison of Scheduling Performance for One and Two Processors



# Process Scheduling

- Conclusion: Using which scheduling algorithm does not quite matter in a multiprocessor system.

# Thread Scheduling

- Thread execution is separated from the rest of the definition of a process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
- On a uniprocessor, threads can be used as a program structuring aid and to overlap I/O with processing
- In a multiprocessor system threads can be used to exploit true parallelism in an application
- Dramatic gains in performance are possible in multi-processor systems
- Small differences in thread management and scheduling can have an impact on applications that require significant interaction among threads

# Approaches to Thread Scheduling

Processes are not assigned to a particular processor

## *Load Sharing*

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

## *Gang Scheduling*

Four approaches for multiprocessor thread scheduling and processor assignment are:

Provides implicit scheduling defined by the assignment of threads to processors

## *Dedicated Processor Assignment*

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

## *Dynamic Scheduling*



# Load Sharing

- Simplest approach and the one that carries over most directly from a uniprocessor environment
- A global queue of ready threads is maintained, each processor when idle picks a thread from the queue.

## Advantages:

- Load is distributed evenly across the processors, assuring that no processor is idle while work is available to do
- No centralized scheduler required
- The global queue can be organized and accessed using any of the schemes discussed in Chapter 9

- Versions of load sharing:
  - First-come-first-served (FCFS)
  - Smallest number of threads first
  - Preemptive smallest number of threads first

# Disadvantages of Load Sharing

- Central queue occupies a region of memory that must be accessed in a manner that enforces mutual exclusion
  - Can lead to bottlenecks
- Preemptive threads are unlikely to resume execution on the same processor
  - Caching can become 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
  - The process switches involved may seriously compromise performance
- Despite the potential disadvantages, still one of the most commonly used strategies.



# Gang Scheduling

- Simultaneous scheduling of the threads that make up a single process

## Benefits:

- Synchronization blocking may be reduced, less process switching may be necessary, and performance will increase
  - Scheduling overhead may be reduced
- Useful for medium-grained to fine-grained parallel applications whose performance severely degrades when any part of the application is not running while other parts are ready to run
  - Also beneficial for any parallel application



# Gang Scheduling

- Idea:
  - Suppose we have  $N$  processors and  $M$  applications which has  $N$  or fewer threads.
  - Each application gets  $1/M$  of the available time on the  $N$  processors. (Uniform allocation)
  - Might not be ideal when you have processes with low number of threads.
  - We can give a weight to each process using the number of threads each process has.

		Processor			
		P1	P2	P3	P4
Time slot	0	A1	A2	A3	A4
	1	B1	idle	idle	idle
	2	A1	A2	A3	A4
	3	B1	idle	idle	idle
	4	A1	A2	A3	A4
		• • •			

(a) Uniform scheduling

		Processor			
		P1	P2	P3	P4
Time slot	0	A1	A2	A3	A4
	1	A1	A2	A3	A4
	2	A1	A2	A3	A4
	3	A1	A2	A3	A4
	4	B1	idle	idle	idle
		• • •			

(b) Weighted scheduling

Figure 10.2 Gang Scheduling

# Dedicated Processor Assignment

- When an application is scheduled, each of its threads is assigned to a processor that remains dedicated to that thread until the application runs to completion
- 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
- Defense of this strategy:
  - In a highly parallel system, with tens or hundreds of processors, processor utilization is no longer so important as a metric for effectiveness or performance
  - The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program



# Real-Time Systems

- Real-Time process or task
  - **A task that is executed in connection with some process or function or set of events external to the computer system and must meet one or more deadlines to interact effectively and correctly with the external environment.**
- Real-time system
  - An operating system that must schedule and manage real-time tasks.
- In a real-time system, some of the tasks are real-time tasks, and these have a certain degree of urgency to them.
- Tasks that are attempting to control or react to events that take place in the outside world.
- Usually possible to associate a deadline with a particular task, where the deadline specifies either a start time or a completion time.

# Real-Time Systems

- The operating system, and in particular the scheduler, is perhaps the most important component

## Examples:

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

- 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



# Basic Concepts of a real-time task

## Associated with a deadline

- Hard real-time tasks
- Soft real-time tasks

## Associated with periodicity

- Aperiodic tasks
- Periodic tasks



# Hard and Soft Real-Time Tasks

## Hard real-time task

- One that must meet its deadline
- Otherwise, it will cause unacceptable damage or a fatal error to the system
- Ex: Flight control systems, nuclear power plant controls systems

## 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
- Ex: Multimedia streaming, online gaming, stock trading

# Periodic and Aperiodic Tasks

## ■ Periodic tasks

- Requirement may be stated as:
  - Once per period  $T$
  - Exactly  $T$  units apart
- Ex: sensor data sampling, heartbeat monitoring

## ■ Aperiodic tasks

- Has a deadline by which it must finish or start
- May have a constraint on both start and finish time
- Ex: Emergency braking system, Intrusion detection systems

# Characteristics of Real Time Systems

Real-time operating systems have requirements in five general areas:

Determinism

Responsiveness

User control

Reliability

Fail-soft operation



# Determinism

- Concerned with how long an operating system delays before acknowledging an interrupt
- Operations are performed at fixed, predetermined times or within predetermined time intervals
  - When multiple processes are competing for resources and processor time, no system will be fully deterministic

The extent to which an operating system can deterministically satisfy requests depends on:

The speed with which it can respond to interrupts

Whether the system has sufficient capacity to handle all requests within the required time

# Determinism

- Multi-process system is in general non-deterministic.

# Responsiveness

- Together with determinism make up the response time to external events
  - Determinism deals with how long OS can delay before acknowledging an interrupt
  - Critical for real-time systems that must meet timing requirements imposed by individuals, devices, and data flows external to the system
- Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt

## Responsiveness includes:

- Amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR)
- Amount of time required to perform the ISR
- Effect of interrupt nesting



# Aspects of Responsiveness

- The amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR).
- The amount of time required to perform the ISR. This generally is dependent on the hardware platform.
- The effect of interrupt nesting. If an ISR can be interrupted by the arrival of another interrupt, then the service will be delayed.

# User Control

- Generally, much broader in a real-time operating system than in ordinary operating systems
- It is essential to allow the user fine-grained control over task priority
- User should be able to distinguish between hard and soft tasks and to specify relative priorities within each class
- May allow user to specify such characteristics as:

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

# Reliability

- More important for real-time systems than non-real time systems.
- How reliable is the system?
  - In a non-real time system, you can reboot the system if it fails.
- Real-time systems respond to and control events in real time so loss or degradation of performance may have catastrophic consequences such as:
  - Financial loss
  - Major equipment damage
  - Loss of life



# Fail-Soft Operation

- 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
- Important aspect is stability
  - A real-time system is stable if the system will meet the deadlines of its most critical, highest-priority tasks even if some less critical task deadlines are not always met
- The following features are common to most real-time OSs
  - A stricter use of priorities than in an ordinary OS, with preemptive scheduling that is designed to meet real-time requirements
  - Interrupt latency is bounded and relatively short. (the amount of time between when a device generates an interrupt and when that device is serviced)
  - More precise and predictable timing characteristics than general purpose OSs

# Common Features of a real-time system

- The following features are common to most real-time OSs
  - A stricter use of priorities than in an ordinary OS, with preemptive scheduling that is designed to meet real-time requirements
  - Interrupt latency is bounded and relatively short. **(the amount of time between when a device generates an interrupt and when that device is serviced)**
  - More precise and predictable timing characteristics than general purpose OSs



# Short term scheduler

- The heart of a real-time system is the short-term task scheduler.
- When designing such a scheduler, fairness and minimizing average response time are not paramount.
- What is important is that all hard real-time tasks complete (or start) by their deadline and that as many as possible soft real-time tasks also complete (or start) by their deadline.



# Real-time Scheduling policies

- Let's look at several real-time scheduling policies.

# Deadline Scheduling

- Real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible and emphasize rapid interrupt handling and task dispatching
- Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times
- Priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time

# Information Used for Deadline Scheduling

## Ready time

- Time task becomes ready for execution

## Resource requirements

- Resources required by the task while it is executing

## Starting deadline

- Time task must begin

## Priority

- Measures relative importance of the task

## Completion deadline

- Time task must be completed

## Subtask scheduler

- A task may be decomposed into a mandatory subtask and an optional subtask

## Processing time

- Time required to execute the task to completion



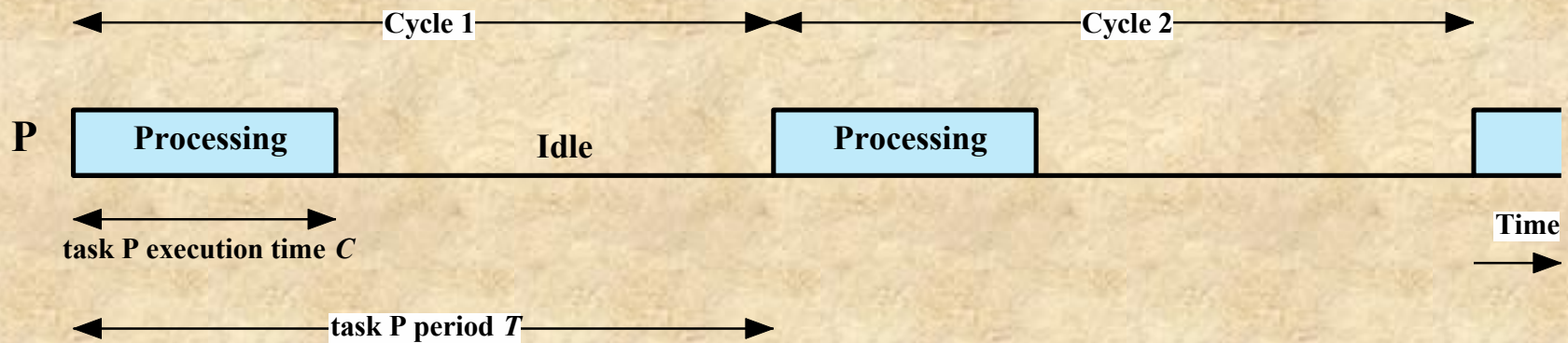
# Deadline Scheduling

- On either a uniprocessor or a multiprocessor, scheduling tasks with the earliest deadline gives us an optimal solution.

# Table 10.3

## 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
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•



**Figure 10.7 Periodic Task Timing Diagram**



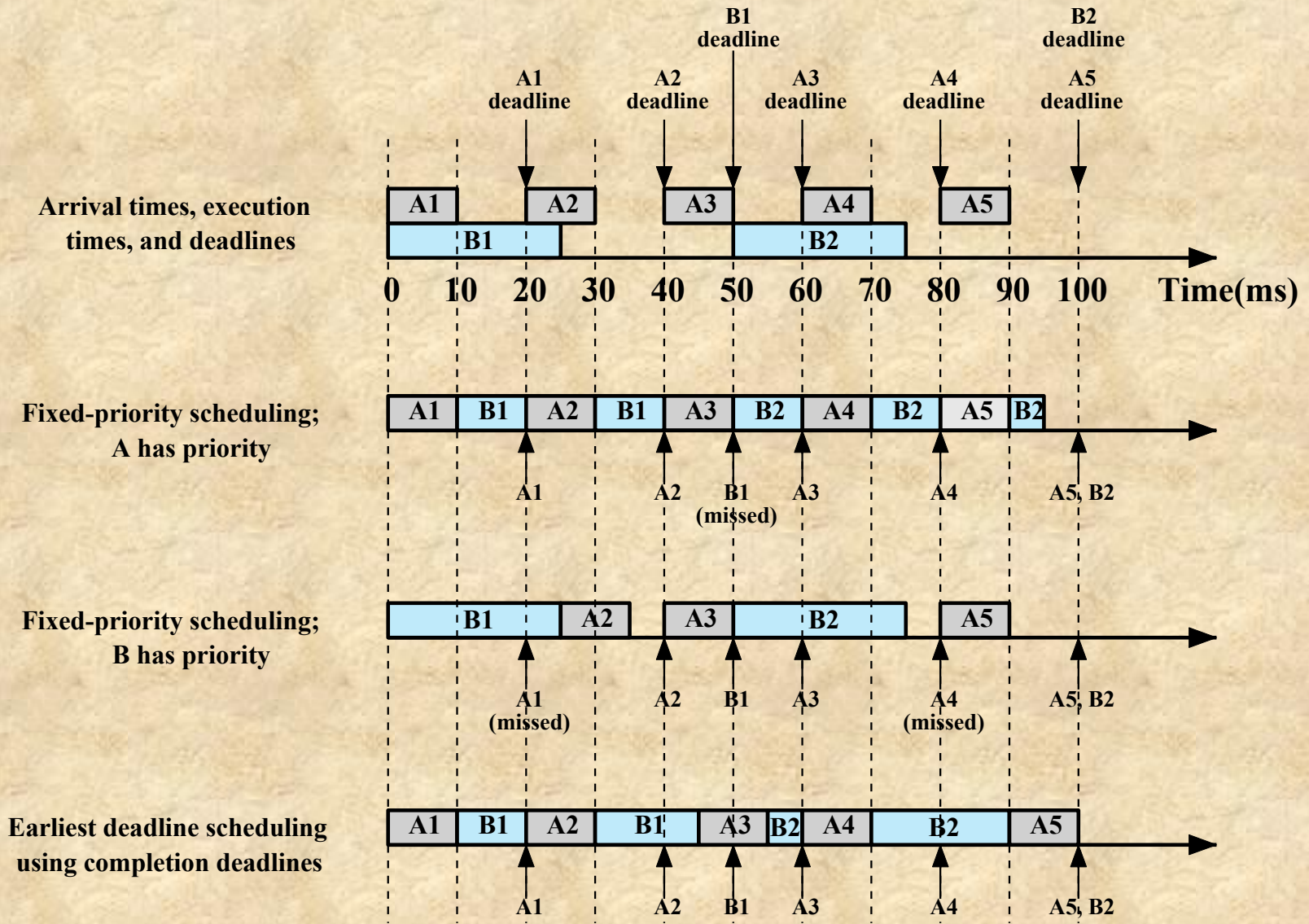
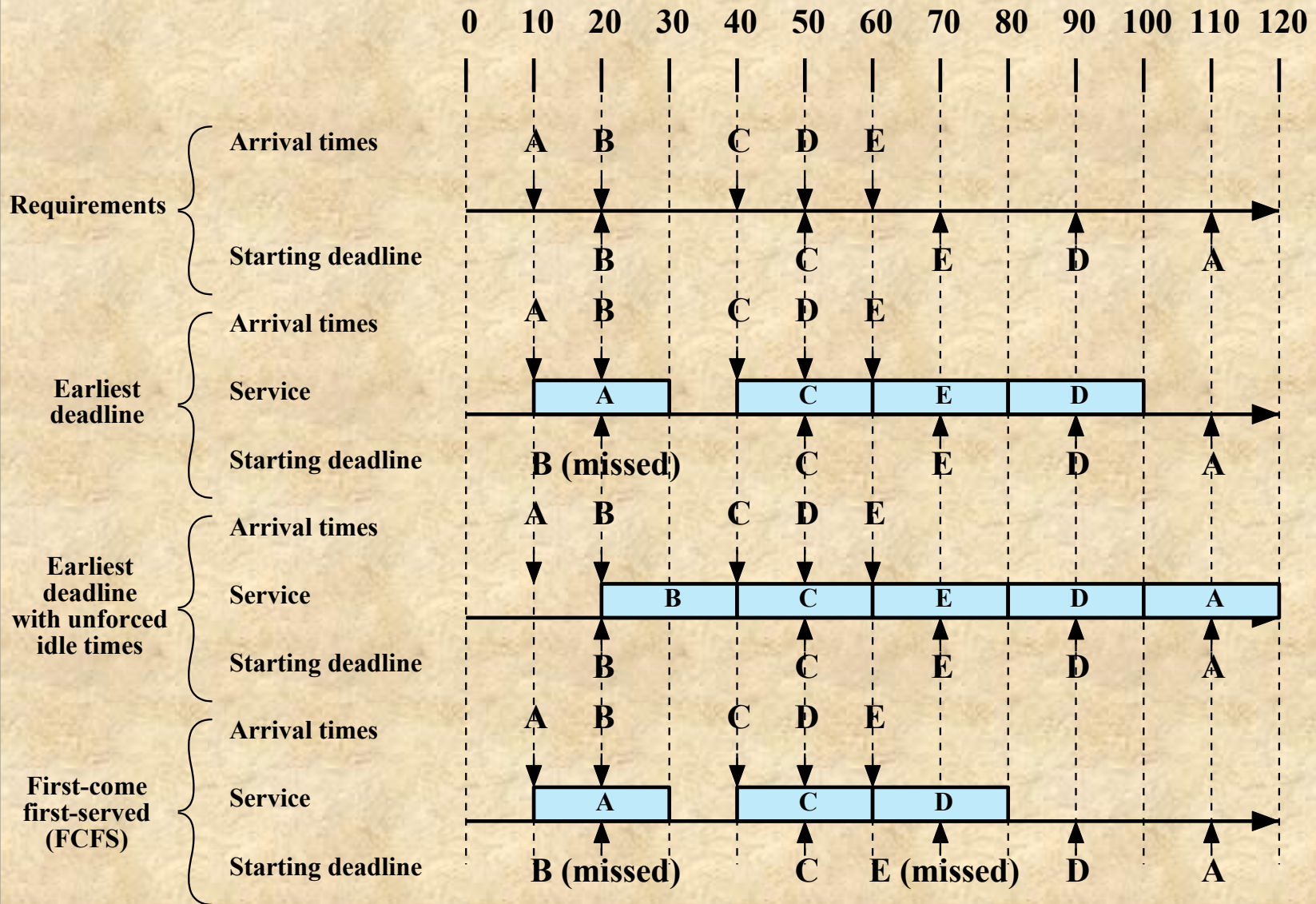


Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.3)

## Table 10.4

### 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



**Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines**

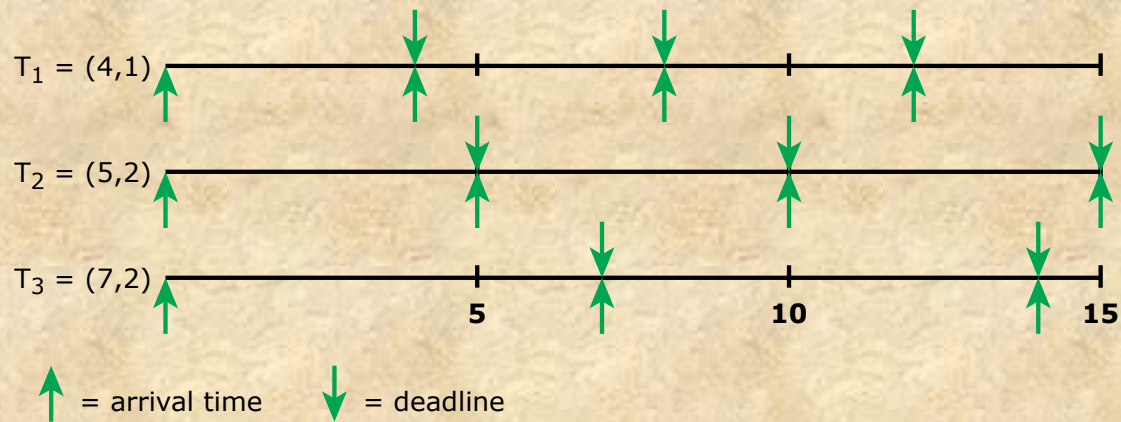


# Rate Monotone scheduling (RMS)

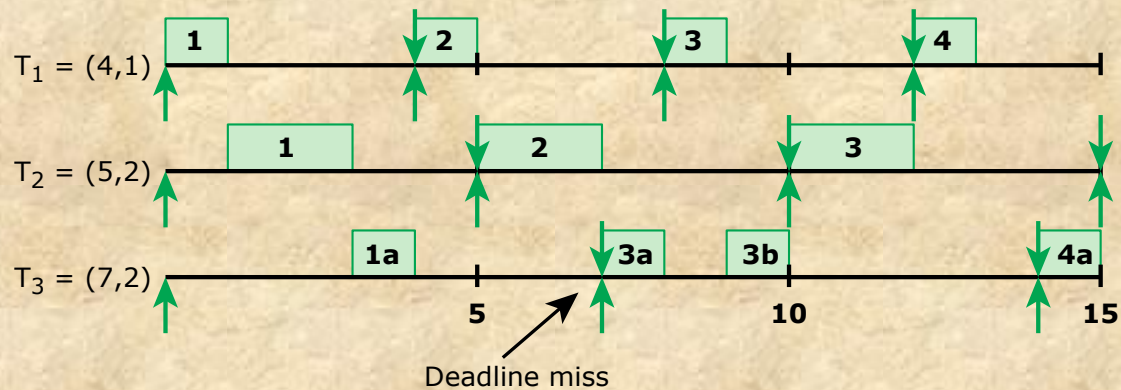
- This is a scheduling algorithm for **periodic tasks**.
- We use  $T$  to denote the task period. (The amount of time between arrival of the next instance of the task).
- The task's rate (in hertz) is the inverse of its period.
  - Basically, it means how many tasks comes to the system per time unit.
- $C$  is used to denote task's computation time.
- $C \leq T$  (otherwise we cannot finish the task before the next tasks comes in.)
- $U$  is used to denote the resource utilization  $U = \frac{C}{T}$
- Typically, end of the period is the deadline for periodic tasks. (not always)

# Rate Monotone scheduling (RMS)

- Idea is very simple.
- RMS assigns priorities to tasks based on their periods ( $T$ ).
- Highest priority task is the one with the shortest period.
- When more than one task is available for execution, the one with the shortest period is serviced first.
- Preemptive scheduling algorithm.
- Let's look at an example.



(a) Arrival times and deadlines for task  $T_i = (P_i, C_i)$ ;  
 $P_i$  = period,  $C_i$  = processing time



(b) Scheduling results

**Figure 10.8 Rate Monotonic Scheduling Example**



# Rate Monotone scheduling (RMS)

- $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n}$  is the total processor utilization.
- $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \leq 1$ , means, you can always use a perfect scheduling algorithm to schedule them without missing deadline.
- For RMS, it has been shown that upper bound of the total processor utilization is  $n \cdot (2^{\frac{1}{n}} - 1)$ .
- Therefore, if  $\sum_{i=1}^n \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$  for the given periodic tasks, then RMS does not miss deadlines.

# Rate Monotone scheduling (RMS)

- Earliest deadline scheduling is better than RMS, in fact using EDF it is possible to achieve better CPU utilization.
- Nevertheless, RMS is widely used in industrial applications.
  - The performance difference is small in practice.
  - It can handle a mixture of hard real-time tasks and soft real-time tasks.
  - It is stable. (maintains predictable, consistent performance as the system load increases.)
    - When a system cannot meet all deadlines because of overload or transient errors, the deadlines of essential tasks need to be guaranteed provided that this subset of tasks is schedulable.



# Priority Inversion

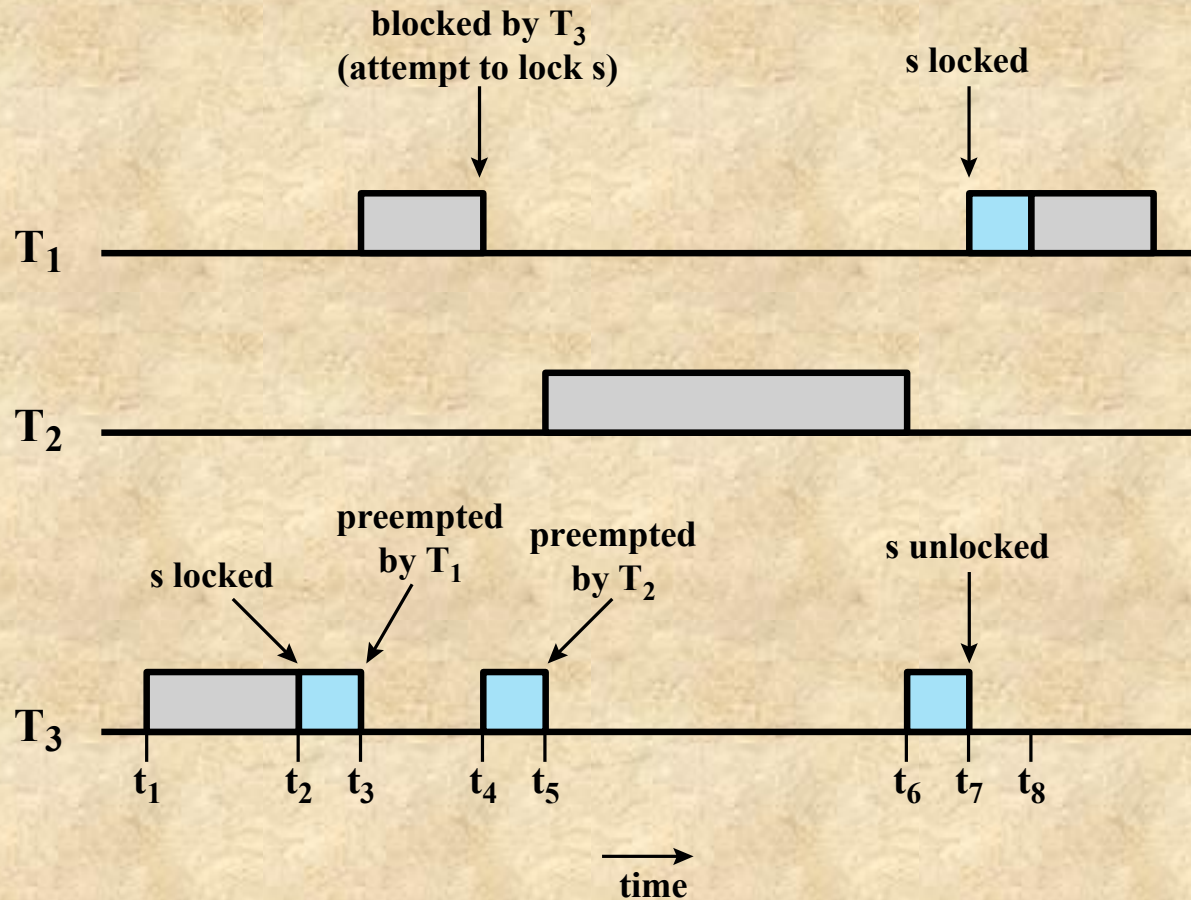
- Can occur in any priority-based preemptive scheduling scheme
- Particularly relevant in the context of real-time scheduling
- Best-known instance involved the Mars Pathfinder mission
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task

## Unbounded Priority Inversion

- 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



# Unbounded Priority Inversion

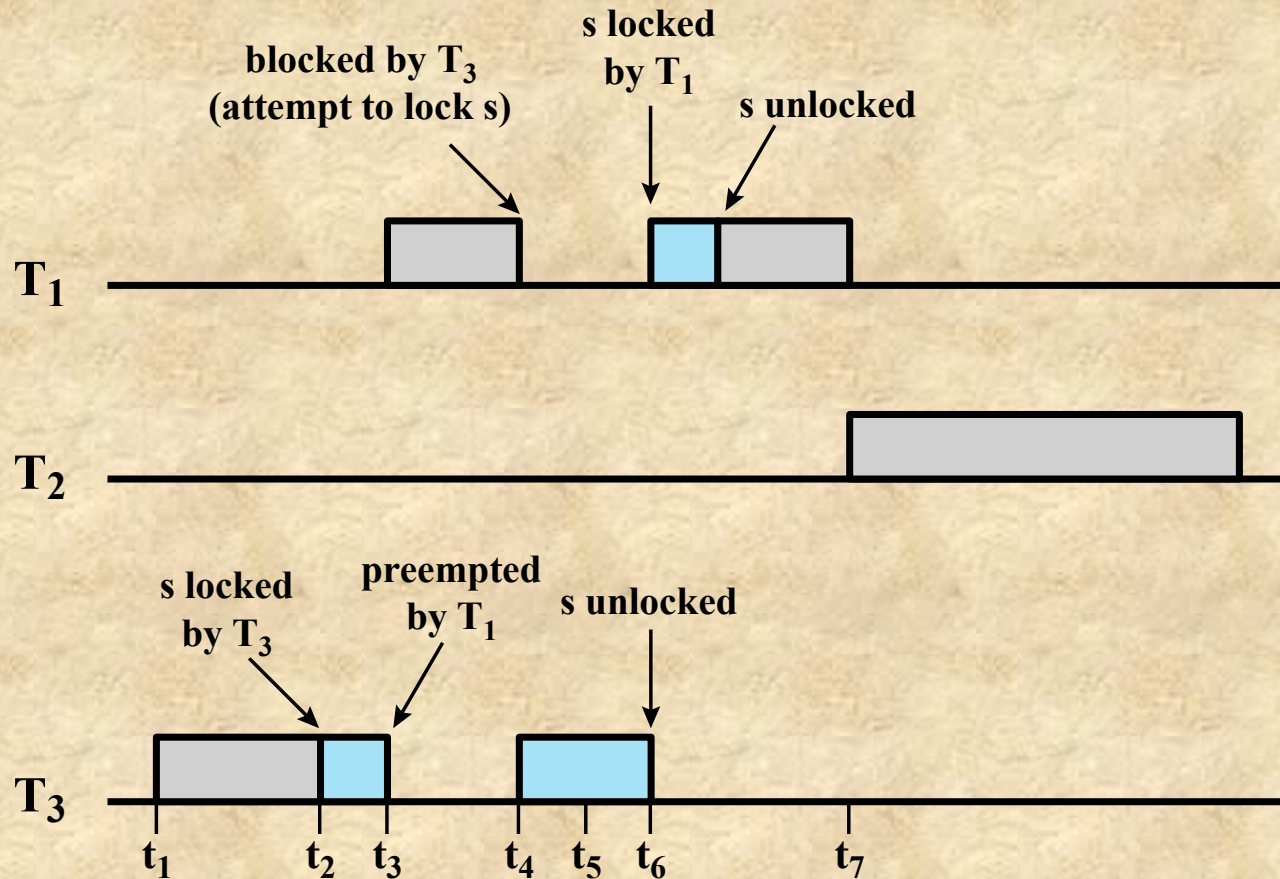


(a) Unbounded priority inversion

# Solution for Priority Inversion ?

- 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
- 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.

# Priority Inheritance



(b) Use of priority inheritance



# Priority Ceiling

- In the **priority ceiling** approach, a priority is associated with each resource.
- The priority assigned to a resource is one level higher than the priority of its highest-priority user.
- The scheduler then dynamically assigns this priority to any task that accesses the resource.
- Once the task finishes with the resource, its priority returns to normal.