Operating System

Chapter 9. Multiprocessor and Real Time Scheduling

Lynn Choi

School of Electrical Engineering

Classification of Multiprocessors

- **Shared-memory multiprocessors**
  - Also called tightly coupled multiprocessors
  - Processors share a common shared main memory
  - Most of multicores and servers
- **Hybrid multiprocessors**
  - A master, general-purpose processor + special slave coprocessors such as DSP, graphic processors
- **Distributed-memory multiprocessors**
  - Also called multicomputers
  - A collection of processors, each with its own private memory and IO channels
  - Some manycores and supercomputers
- **Clusters or distributed systems**
  - A set of autonomous systems connected through networks/internets

Synchronization Granularity

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

Source: Pearson

Independent Parallelism

- **No explicit synchronization among processes**
  - Each represents a separate, independent application or job
  - Typical use is in a time-sharing system
  - Each user is performing a particular application
- **Multiprocessor provides the same service as a multiprogrammed uniprocessor**
  - Average response time will be reduced because more than one processor is available

Coarse and Very Coarse Grained Parallelism

- **Synchronization among processes, but at a very gross level**
- **Good for concurrent processes running on a multiprogrammed uniprocessor**
  - Can be supported on a multiprocessor with little or no change to user software

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**

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

Design Issues

- **Scheduling on a multiprocessor involves three interrelated issues:**
  - Assignment of processes to processors
  - The use of multiprogramming on individual processors
  - The actual selection of a process to run
    - The use of priorities or past usage may improve performance for uniprocessor but these complexities may be unnecessary or even counterproductive for multiprocessors
- **The approach taken will depend on the degree of granularity of applications and 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 processes to processors on demand**
- **Static assignment**
  - A process is permanently assigned to one processor from activation until its completion with a dedicated queue for each processor
    - Advantage: less scheduling overhead
    - Disadvantage : 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. In this case over the lifetime of a process, it may be executed on different processors at different times
- **Another option is dynamic load balancing**
  - Threads are moved from a queue for one processor to a queue for another processor. Linux uses this approach.

Master/Slave Architecture

- **Master processor**
  - Run key kernel functions
  - Responsible for scheduling
  - Has control of all memory and IO resources
- **Slave processors**
  - Run user programs
  - Send service request to the master for IO and system services
- **Advantage**
  - Simple and requires little enhancement to a uniprocessor multiprogramming operating system
- **Disadvantage**
  - Failure of master brings down the 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**
- **Complicate the operating system**
  - Operating system must ensure that two processors do not choose the same process and need to resolve and synchronize competing claims to resources

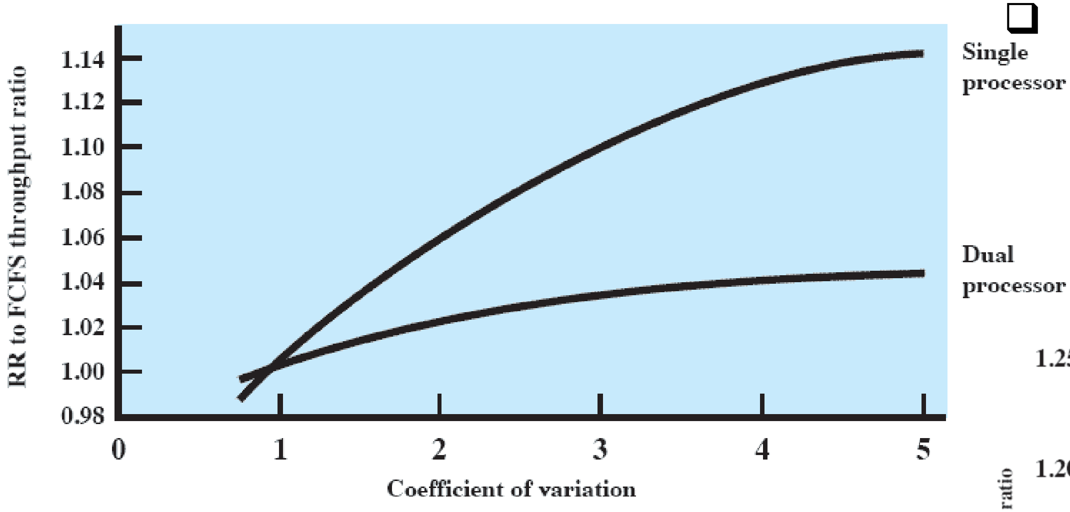
The Use of Multiprogramming on Individual Processors

- **When each process is statically assigned to a processor for the duration of lifetime, should that processor multiprogrammed?**
  - In the traditional multiprocessor with coarse-grained or independent synchronization granularity, it is clear that each individual processor should be able to switch among a number of processes to achieve high utilization.
  - However, for medium-grained applications running on a multiprocessor with many processors, the situation is not clear.
    - With many processors available, it is no longer paramount that every single processor be busy as much as possible
    - Rather, we are concerned to provide the best performance for the applications.
    - An application that consists of many threads may run poorly unless all of its threads are available to run simultaneously.

Process Scheduling

- **Usually 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
  - We can view the system as a multi-server queuing architecture

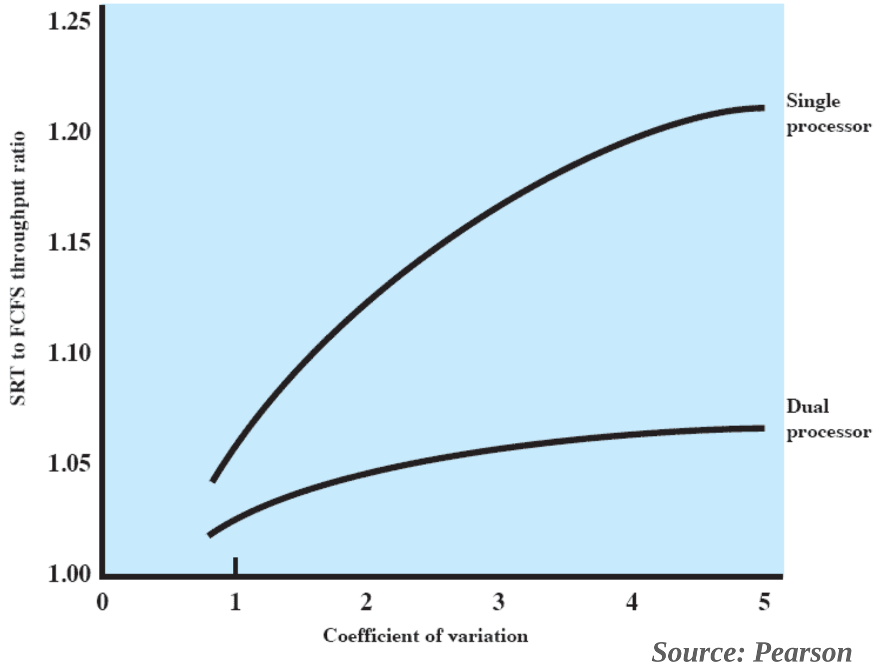
Comparison of Single and Dual Processors



Source: Pearson

(a) Comparison of RR and FCFS

- **C_s measures the coefficient of variation.**
  - C_s is calculated as σ/T_s where σ is the standard deviation of the service time and T_s is the mean service time
  - $C_s = 0$ corresponds to the case where the service times of all processes are equal.
  - The larger the C_s, the more variation in the service time
  - Values of C_s of 5 or more are not unusual



Source: Pearson

(b) Comparison of SRT and FCFS

- **The specific scheduling discipline is much less important with dual processors than with one.**

Thread Scheduling

- **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 performance gains are possible in multiprocessor systems
  - For applications that require significant interaction among threads, small differences in thread management and scheduling can have a significant performance impact

Approaches to Thread Scheduling

- **Load sharing**
  - Processes are not assigned to a particular processor
  - A global queue of ready threads is maintained. Each processor, when idle, selects a thread from the queue
- **Gang scheduling**
  - A set of related threads is scheduled to run on a set of processors at the same time
- **Dedicated processor assignment**
  - The opposite of the load sharing approach
  - Dedicate a group of processors to an application for the duration of the application.
    - When an application is scheduled, each of its threads is assigned a processor that remains dedicated to that thread until completion
- **Dynamic scheduling**
  - The number of threads in a process can be altered during the course of execution

Load Sharing

- Simplest approach and carries over most directly from a uniprocessor environment
- Advantages
  - Load is distributed evenly across the processors
  - No centralized scheduler is required. When a processor is available, the scheduler can run on that processor to select the next thread
  - The global queue can be organized and accessed using any of the scheduling algorithms discussed in Chapter 9
- Versions of load sharing
  - First-come-first-served
    - When a job arrives, its threads are placed at the end of the shared queue. An idle processor selects the next ready thread, which it executes until completion or blocking.
  - Smallest number of threads first
    - The shared ready queue is organized as a priority queue with highest priority given to threads from jobs with smallest number of threads.
  - Preemptive smallest number of threads first

Load Sharing

- **Disadvantages**
  - 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
    - May seriously compromise performance
- **Despite its disadvantages, it is one of the most commonly used schemes**

Gang Scheduling

- **Simultaneous scheduling of the threads that make up a single process on a set of processors**
- **Advantages**
  - If closely related threads execute in parallel, synchronization blocking may be reduced, less thread switching, and performance will increase
  - Scheduling overhead may be reduced because a single decision affects a number of processors and threads at one time
- **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**

Example of Scheduling Groups with 4 and 1 Threads

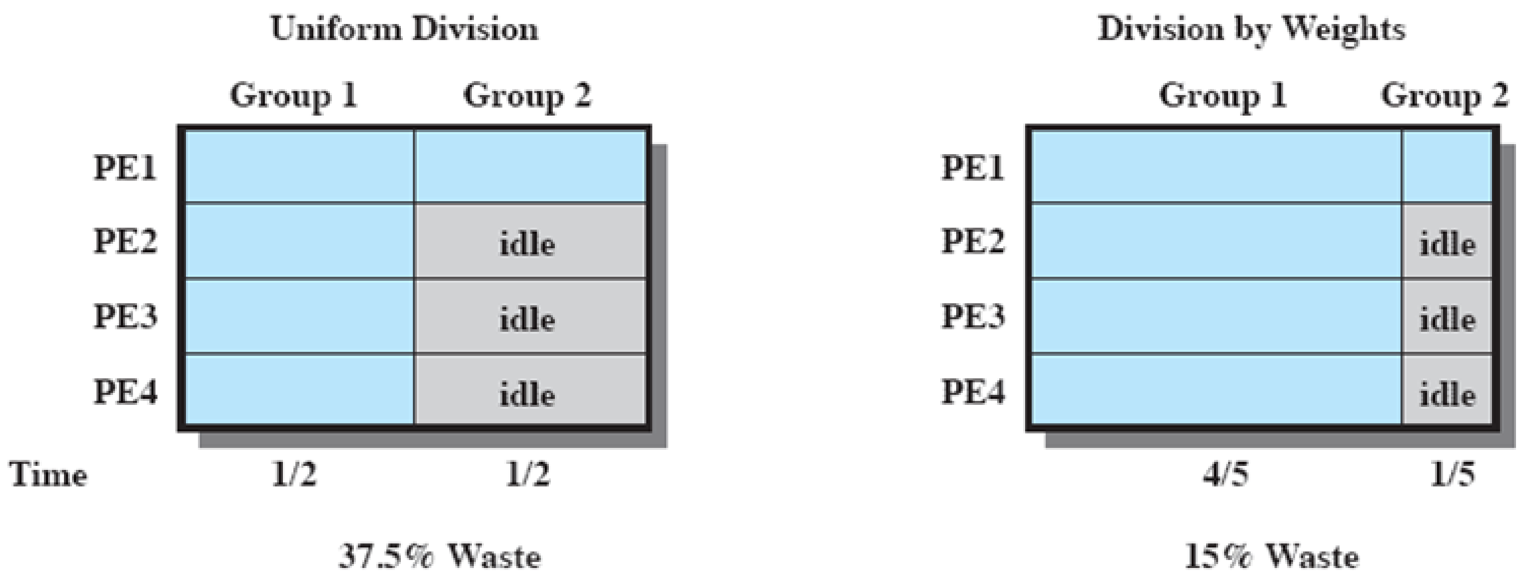


Figure 10.3 Example of Scheduling Groups with Four and One Threads [FEIT90b]

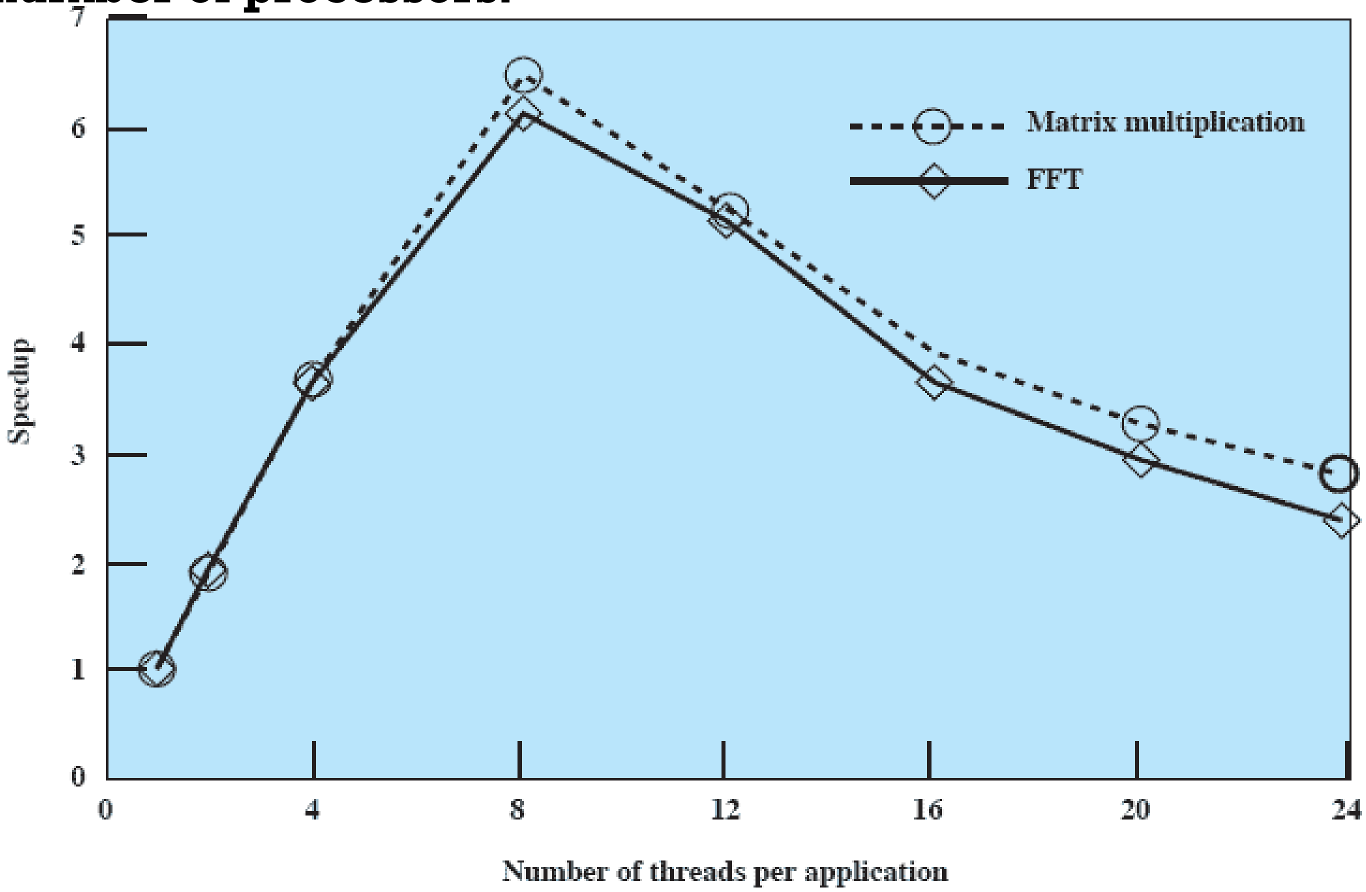
Source: Pearson

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

Application Speedup as a Function of Number of Threads

- The performance worsens considerably when the number of threads in each application exceeds 8 and thus the total number of threads exceeds the number of processors.



Source: Pearson

Figure 10.4 Application Speedup as a Function of Number of Threads

Dynamic Scheduling

- **For some applications it is possible to provide language and system tools that permit the number of threads in the process to be altered dynamically**
  - This would allow the operating system to adjust the load to improve utilization
- **Both the operating system and the application are involved in making scheduling decisions**
  - When a job requests one or more processors,
    - If there are idle processors, assign them to satisfy the request
    - Otherwise, if the job is a new arrival, allocate it a single processor by taking one away from any job currently allocated more than one processor
    - If any of the request cannot be satisfied, it remains outstanding until either a processor become available
    - Upon release of one or more processors, scan the current queue of unsatisfied requests an assign a single processor to each job in the list
- **This approach is superior to gang scheduling or dedicated processor assignment for applications that can take advantage of it, but the overhead may negate the performance advantage.**

Real-Time Systems

- **Examples of real-time systems**
  - 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**
  - Real-time tasks attempt to control or react to events that take place in the outside world
  - Because these events occur in “real time” and tasks must be able to keep up with the events in time

Hard and Soft Real Time Tasks

- **A hard real time task**
  - 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 and Aperiodic Tasks

- **An aperiodic task**
  - Has a deadline by which it must finish or start
  - May have a constraint on both start and finish time
- **A periodic task**
  - Has a requirement that may be stated as:
    - Once per period *T*, or
    - Exactly *T* units apart

Characteristics of Real Time Systems

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

Determinism

- **When multiple processes are competing for resources and processor time, no system will be fully deterministic**
- **One useful measure of the ability of OS to function deterministically is the maximum delay from the arrival of a high-priority interrupt to when the service begins**
- **The extent to which an operating system can deterministically satisfy requests depends on:**
  - The speed with which it can respond to interrupts
    - In non-real-time OS, this delay may be in the range of tens to hundreds of milliseconds
    - In real-time OS, this delay may have an upper bound from a few microseconds to a millisecond
  - Whether the system has sufficient capacity to handle all requests within the required time

Responsiveness

- **Determinism is concerned with how long an OS delays before acknowledging an interrupt while responsiveness is concerned with how long it takes an OS to service the interrupt after the acknowledgment**
  - The amount of time required to initially handle the interrupt and begin execution of the ISR. This includes context switching.
  - The amount of time required to perform ISR
  - The effect of interrupt nesting. If an ISR can be interrupted by another interrupt, the service will be delayed.
- **Determinism and responsiveness make up the response time to external events**
  - Critical for real-time systems that must meet timing requirements imposed by individuals, devices, and data flows external to the system

User Control

- **In a typical non-real-time OS, the user has no control over the scheduling, or only provide broad guidance such as grouping users into more than one priority class.**
- **In a real-time OS, it is essential to allow the user fine-grained control over task priority.**
  - The user distinguish between hard and soft real-time tasks.
  - The user specify relative priorities within each priority 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 each priority class have

Reliability

- **More important for real-time systems than non-real time systems**
  - In a non-real-time system, a failure may result in a reduced level of service. And, it can be often solved by rebooting the system.
- **However, in real-time systems, loss or degradation of performance may have catastrophic consequences such as:**
  - Financial loss
  - Major equipment damage
  - Loss of life

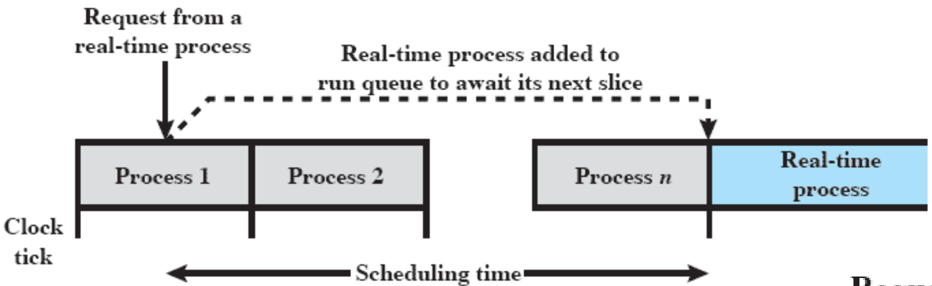
Fail-Soft Operation

- **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

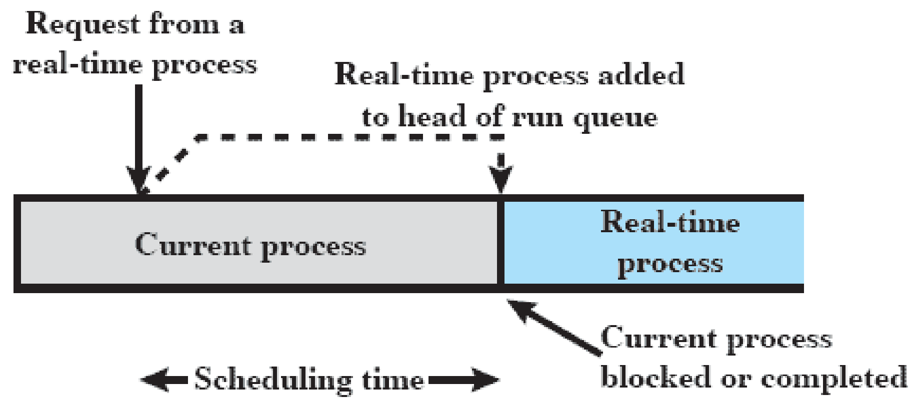
Real-Time OS Charatertistics

- **To meet the requirements, real-time OS has the following features in general**
  - Fast process/thread switching
  - Small size
  - Ability to respond to external events quickly
  - Preemptive scheduling based on priority
  - Minimize intervals during which interrupts are disabled
  - Short-term scheduler optimized for real-time tasks
    - Fairness and minimizing average response time is not important
    - What is important is that all hard real-time tasks must complete (or start) by their deadline an soft real-time tasks must also complete by their deadline as much as possible

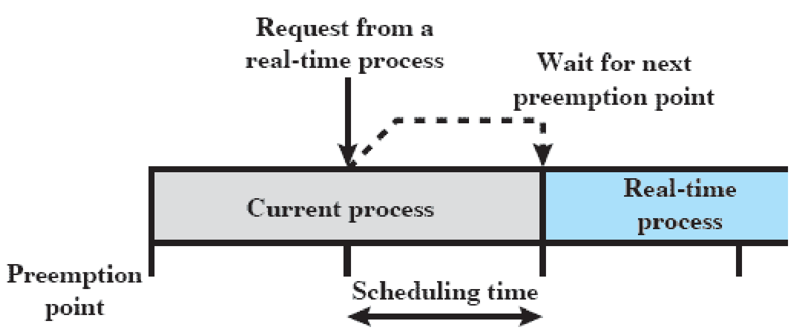
Real Time Scheduling of Processes



(a) Round-robin Preemptive Scheduler



(b) Priority-Driven Nonpreemptive Scheduler



(c) Priority-Driven Preemptive Scheduler on Preemption Points

(d) Immediate preemptive scheduler!

Source: Pearson

Real-Time Scheduling

- **Real-time scheduling is one of the most active areas of research in computer science**
- **Scheduling approaches depend on**
  - Whether a system performs schedulability analysis
    - If it does, whether it is done statically or dynamically
  - Whether the result of the analysis itself produces a schedule according to which tasks are dispatched at run time

Classes of Real-Time Scheduling Algorithms

Static table-driven approaches

- Perform a static analysis of feasible schedules of dispatching
- Result is a schedule that determines when a task must begin execution
- Applicable to periodic tasks whose arrival time, execution time, deadlines are predictable

Static priority-driven preemptive approaches

- A static analysis is performed but no schedule is drawn up
- Analysis is used to assign priorities to tasks so that a traditional priority-driven preemptive scheduler can be used
- Common in most non-real-time systems

Dynamic planning-based approaches

- Feasibility is determined at run time rather than offline prior to the start of execution
- One result of the analysis is a schedule that is used to decide when to dispatch this task

Dynamic best effort approaches

- No feasibility analysis is performed
- System tries to meet all deadlines and aborts any started process whose deadline is missed
- Used by many commercial real-time systems

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

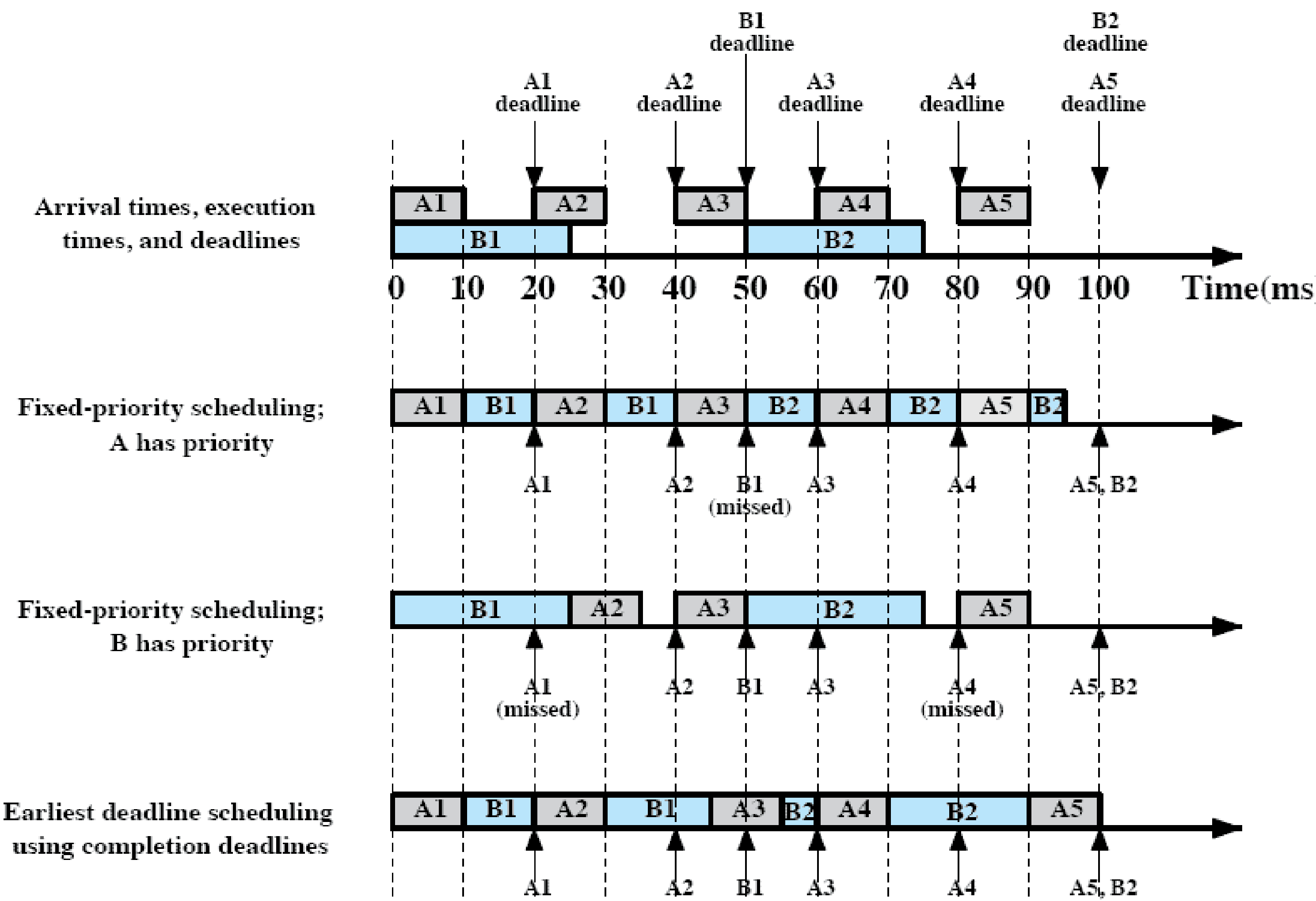
- **Ready time**
  - Time at which task becomes ready for execution
- **Starting deadline**
  - Time by which task must begin
- **Completion deadline**
  - Time by which task must be completed
- **Processing time**
  - Time required to execute the task to completion
- **Resource requirements**
  - Resources required by the task while it is executing
- **Priority**
  - Measures the relative importance of the task
- **Subtask structure**
  - A task may be decomposed into a mandatory subtask and an optional subtask. Only the mandatory subtask possesses a hard deadline

Execution Profile of Two 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
• • •	• • •	• • •	• • •

Source: Pearson

Scheduling of Periodic Tasks with Completion Deadlines



Source: Pearson

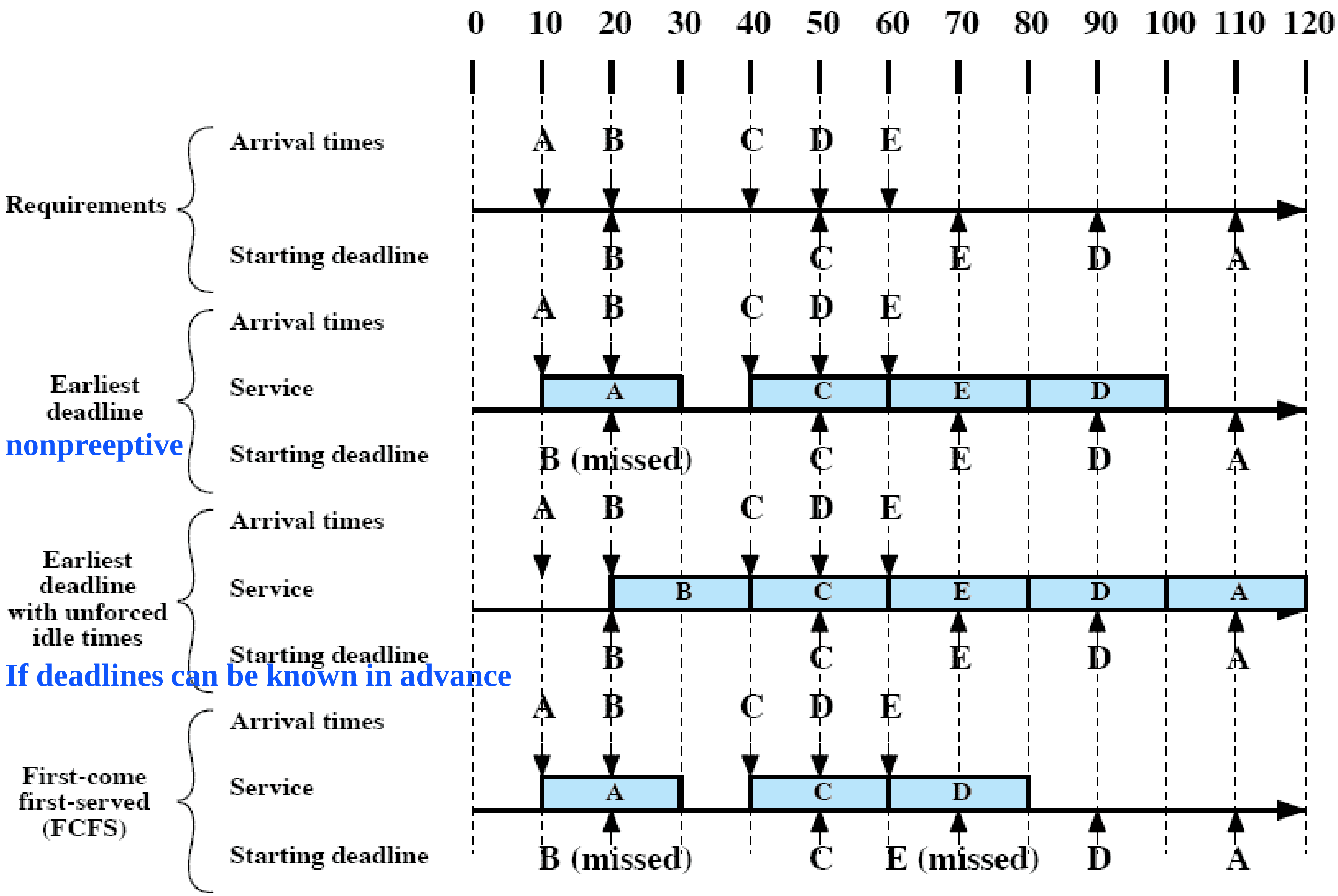
Figure 10.6 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

Execution Profile of 5 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

Source: Pearson

Scheduling of Aperiodic Tasks with Starting Deadlines



Source: Pearson

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

Rate Monotonic Scheduling

- **RMS**
  - Popular scheduling algorithm for periodic tasks
  - The highest priority is given to a process with the shortest period
  - The 2^{nd} highest priority is given to a process with the second shortest period
  - The priority is a monotonically increasing function of their rate

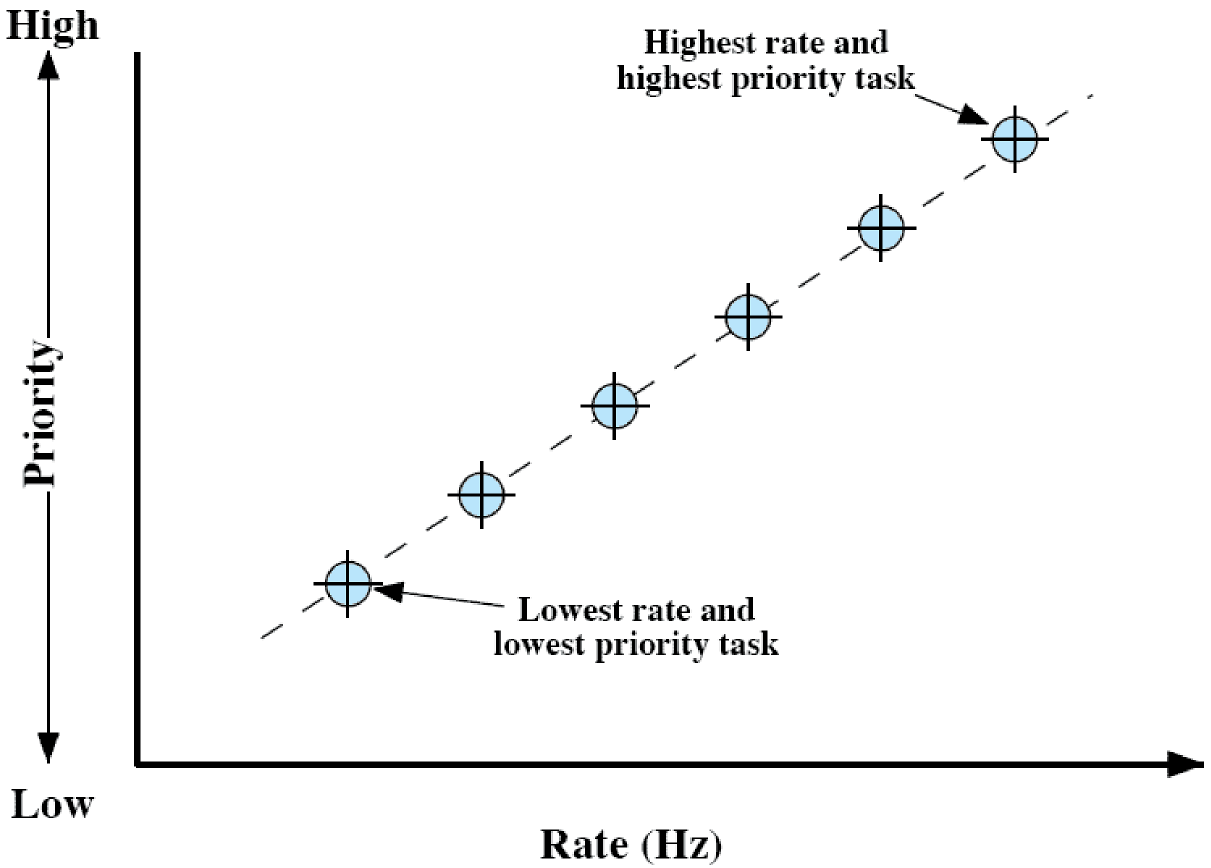
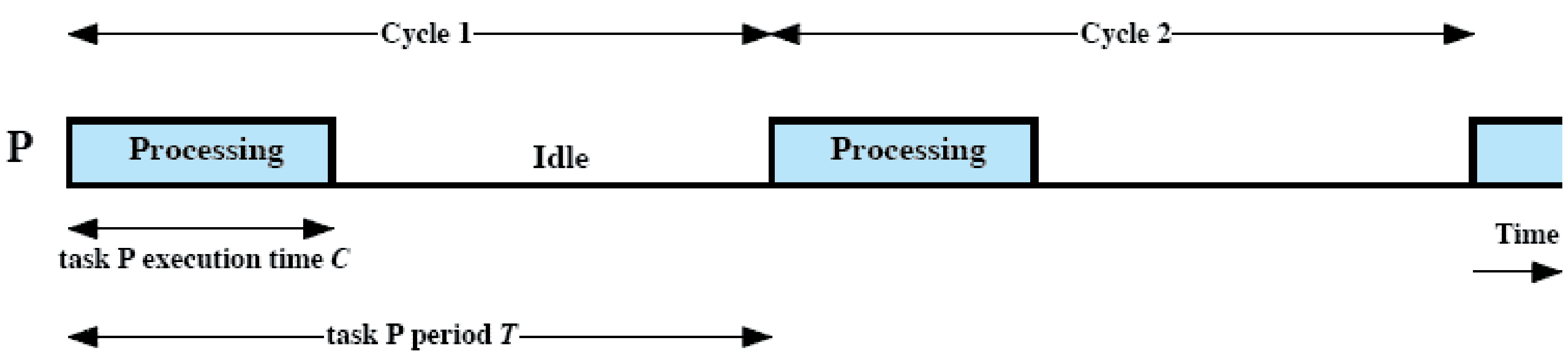


Figure 10.8 A Task Set with RMS [WARR91]

Source: Pearson

Periodic Task Timing Diagram



For example, a task with a period of 50ms occurs at a rate of 20Hz!

Figure 10.9 Periodic Task Timing Diagram

Source: Pearson

RMS Analysis

- **Given**
  - C = execution time
  - T = period
  - U = C/T = CPU utilization
    - The execution time must be no greater than the period
- **The following inequality holds**
  - $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 1$
  - Provide a bound on the number of tasks that a scheduling algorithm can schedule
  - For any particular algorithm, the bound may be lower
- **For RMS, it can be shown that the following inequality holds**
  - $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$

Value of the RMS Upper Bound

n	$n(2^{1/n}-1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
•	•
•	•
•	•
∞	$\ln 2 \approx 0.693$

Source: Pearson

Priority Inversion

- **Can occur in any priority-based preemptive scheduling scheme**
  - Best-known instance was the Mars Pathfinder mission. The robot gathered and transmitted voluminous data back to Earth. But a few days into the mission, the software began experiencing system resets, each resulting in losses of data. The problem was traced to priority inversion.
- **Priority inversion occurs when circumstances within the system force a higher priority task to wait for a lower priority task**
  - A simple example occurs if a lower-priority task has locked a resource and a higher priority task attempts to lock the same resource. The higher priority task will block until the resource is available.
- **A more serious condition is referred as an *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. The pathfinder software was a good example.

Pathfinder Example

- **Pathfinder software included the following 3 tasks in decreasing order of priority**
  - T1: periodically check the health of the spacecraft system and software
  - T2: process image data
  - T3: perform an occasional test on equipment status
- **After T1 executes, it reinitializes a timer to a maximum**
- **If this timer ever expires,**
  - It is assumed that the system integrity has been compromised
  - The CPU is halted, all devices are reset, and the software is reloaded, and the spacecraft system starts over.
  - This recovery sequence does not complete until the next day.
- **T1 and T3 share a common data structure protected by a binary semaphore**

Unbounded Priority Inversion

t1: T3 begin executing.

t2: T3 locks semaphore s

t3: T1 (higher priority) preempt T3 and begin executing.

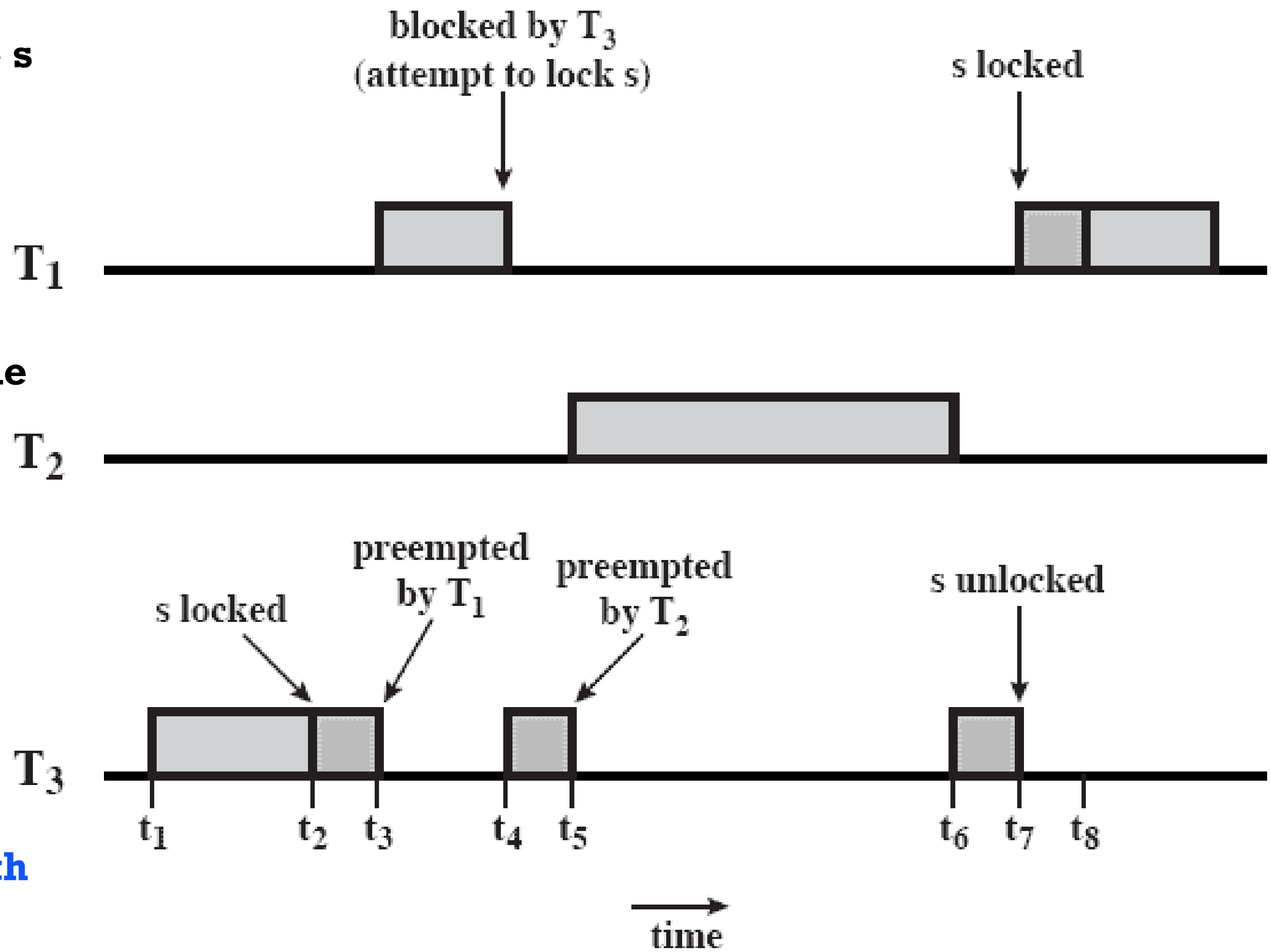
t4: T1 is blocked since the semaphore s is locked by T3. T3 resume execution.

t5: T2 preempt T3

t6: T2 is suspended for some reason and T3 resumes

t7: T3 unlocks s and T1 preempts T3

=> T1 must wait for both T2 and T3 to complete and fails to reset the timer before it expires!

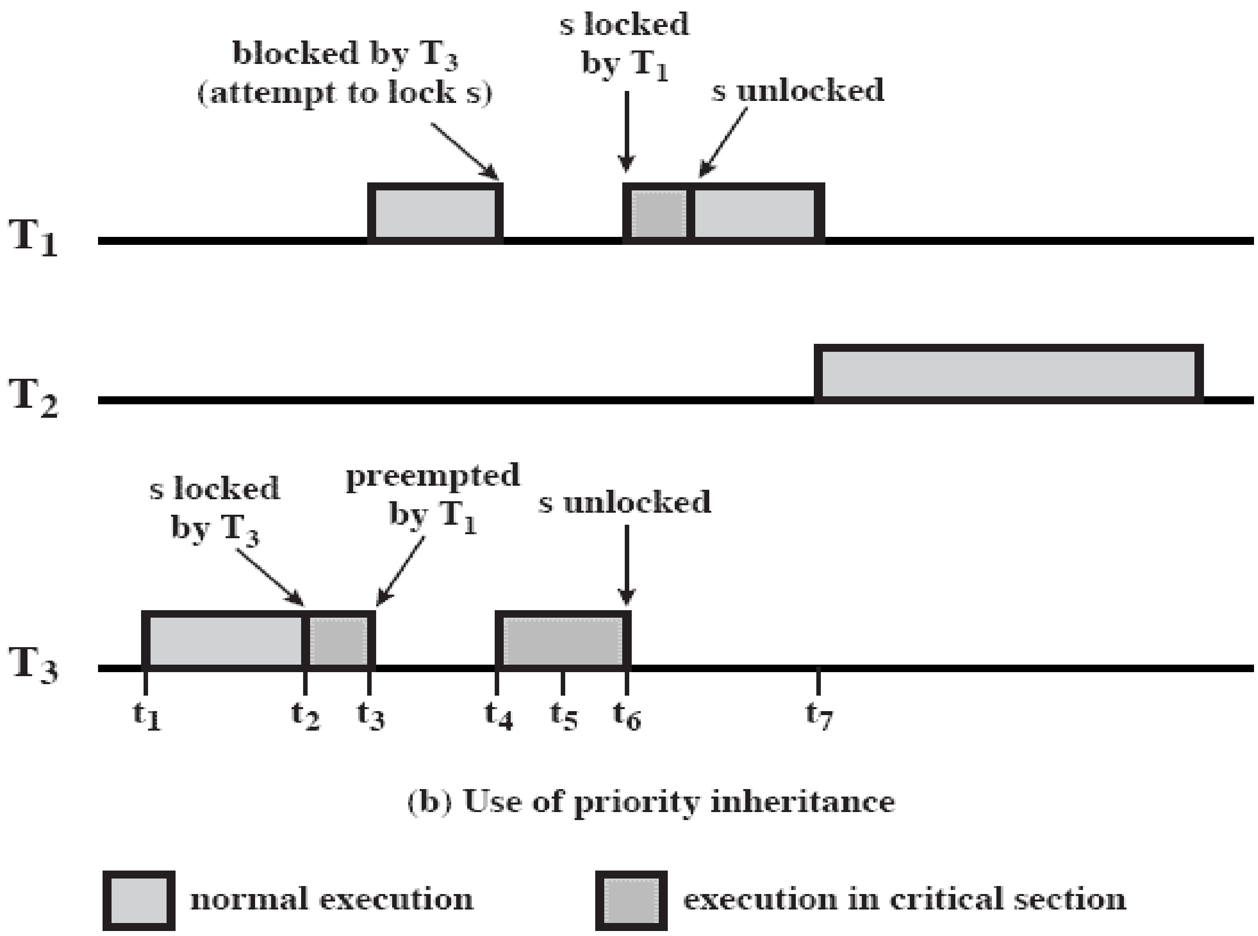


(a) Unbounded priority inversion

Source: Pearson

Priority Inheritance

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



(b) Use of priority inheritance

Source: Pearson

Homework 9

- Exercise 10.1
- Exercise 10.2
- Exercise 10.6
- Exercise 10.9
- Exercise 10.15
- Exercise 10.16