Chapter 10
Multiprocessor and Real-Time Scheduling

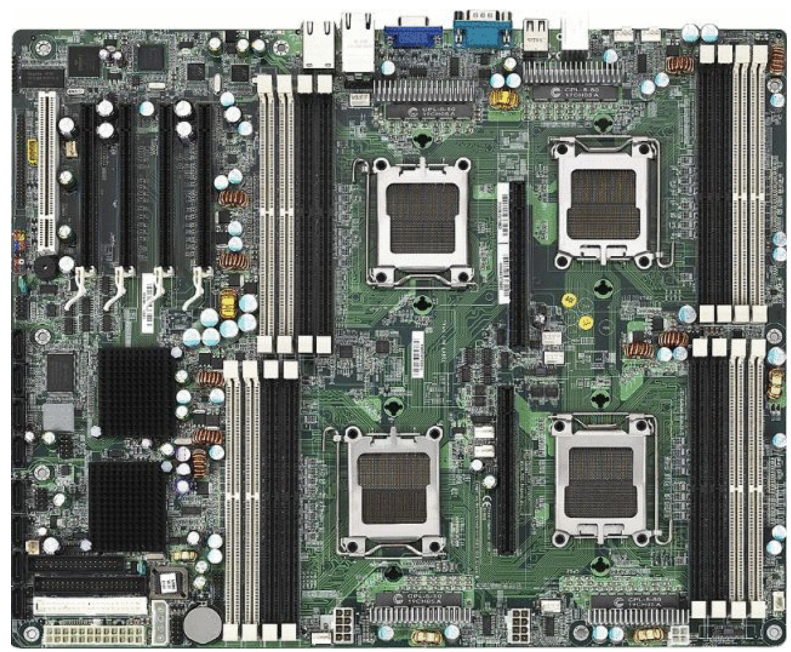
(based on original slides by Pearson)

Classifications of Multiprocessor Systems

- Loosely coupled or distributed multiprocessor, or cluster
  - Each processor has its own memory and I/O channels
  - Classical distributed system
- Functionally specialized processors
  - Such as I/O processor
  - Controlled by a master processor
  - FPGA cards

Classifications of Multiprocessor Systems

- Tightly coupled multiprocessing
  - Processors share main memory
  - Controlled by operating system
  - Multi-core, SMP, cell processor, FPGA in processor socket



Independent Parallelism

- Processes are separate applications
- → no synchronization among processes

- Example is time-sharing system
  - Run word processing, shell, mail, browser
  - Properties similar to the uniprocessor system

Types of Parallelism

- Coarse and Very Coarse-Grained Parallelism
  - Distributed processing across network nodes
  - Multiprocessing of concurrent processes in a multiprogramming environment
  - Little synchronization among processes
    - Good for concurrent processes running on a multiprogrammed uniprocessor
    - Can by supported on a multiprocessor with little change

Types of Parallelism

- Medium-Grained Parallelism
  - Parallel processing or multitasking within a single application
  - Threads usually interact frequently and share data
  - → requires synchronization

  - Scheduling decisions regarding one thread might affect the performance of the entire application.

Types of Parallelism

- Fine-Grained Parallelism
  - Parallelism inherent in a single instruction stream
  - Highly parallel applications

Thread Structure for Rendering Module

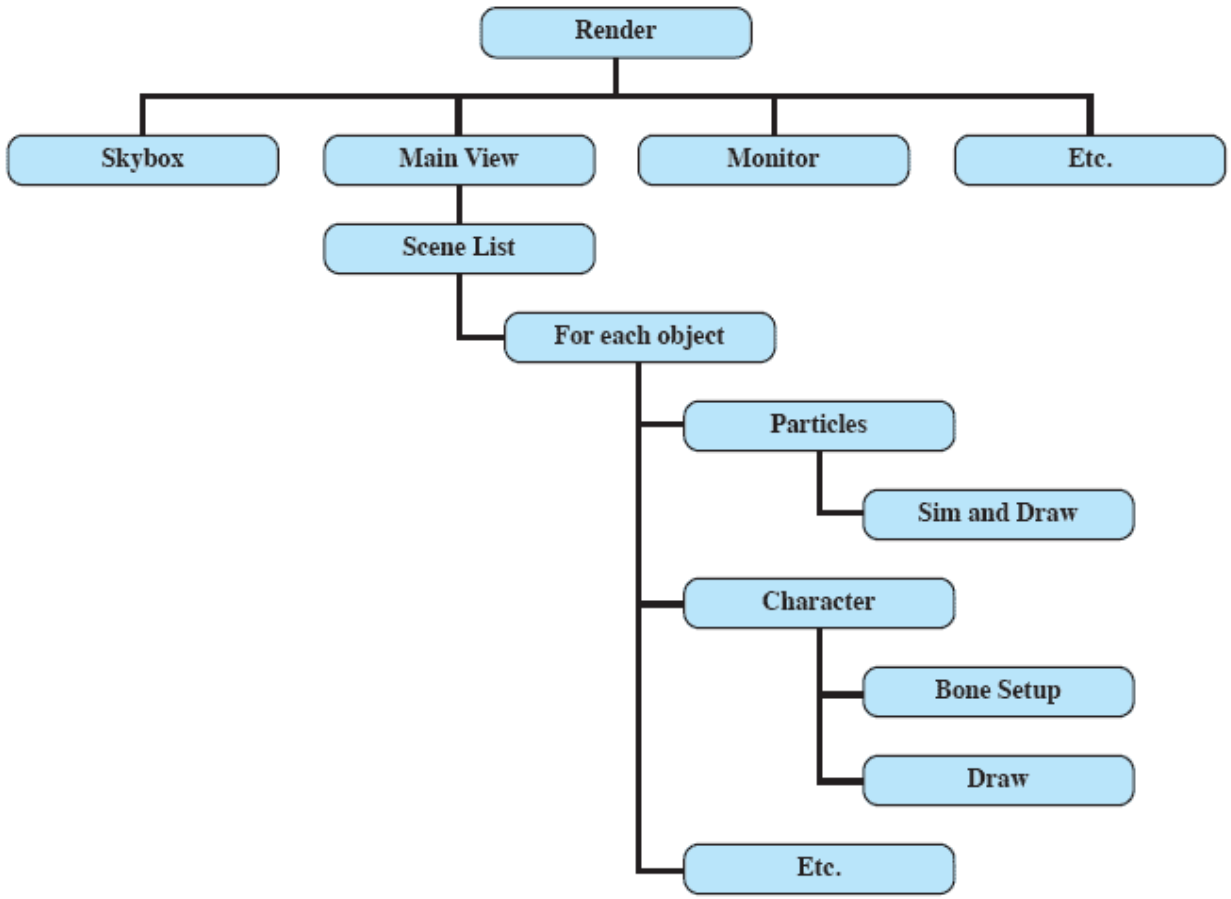


Figure 10.1 Hybrid Threading for Rendering Module

Scheduling Design Issues

- Scheduler needs to consider:
  - Assignment of processes to processors
  - Use of multiprogramming on individual processors
  - Actual dispatching of a process

Assignment of Processes to Processors

- Two architectural styles:
  1. Multiprocessor is uniform
  2. Multiprocessor is heterogeneous

- Assuming architectural style 1:
  a) Assign processes to a dedicated processor
  b) Migrate processes between processors

- Style 2 requires special software.

Assignment of Processes to Processors

- Option 1.a: static assignment
  - Low system overhead, because decision is made once.
  - Permits group or gang scheduling
  - Processor can be idle while another has a backlog
  - Decisions are local
- Option 1.b: process migration
  - Potential high overhead due to migration
- Option 1.ab: dynamic load balancing
  - Have a static assignment, but migrate processes sometimes

Assignment of Processes to Processors

- Master/slave architecture
  - Key kernel functions always run on a particular processor (→ RTLinux design)
  - Master is responsible for scheduling
  - Slave sends service request to the master and waits for result
  - Advantage: simple design, similar to uniproc.
  - Disadvantages
    - Failure of master brings down whole system
    - Master can become a performance bottleneck

Assignment of Processes to Processors

- 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

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

Multiprogramming

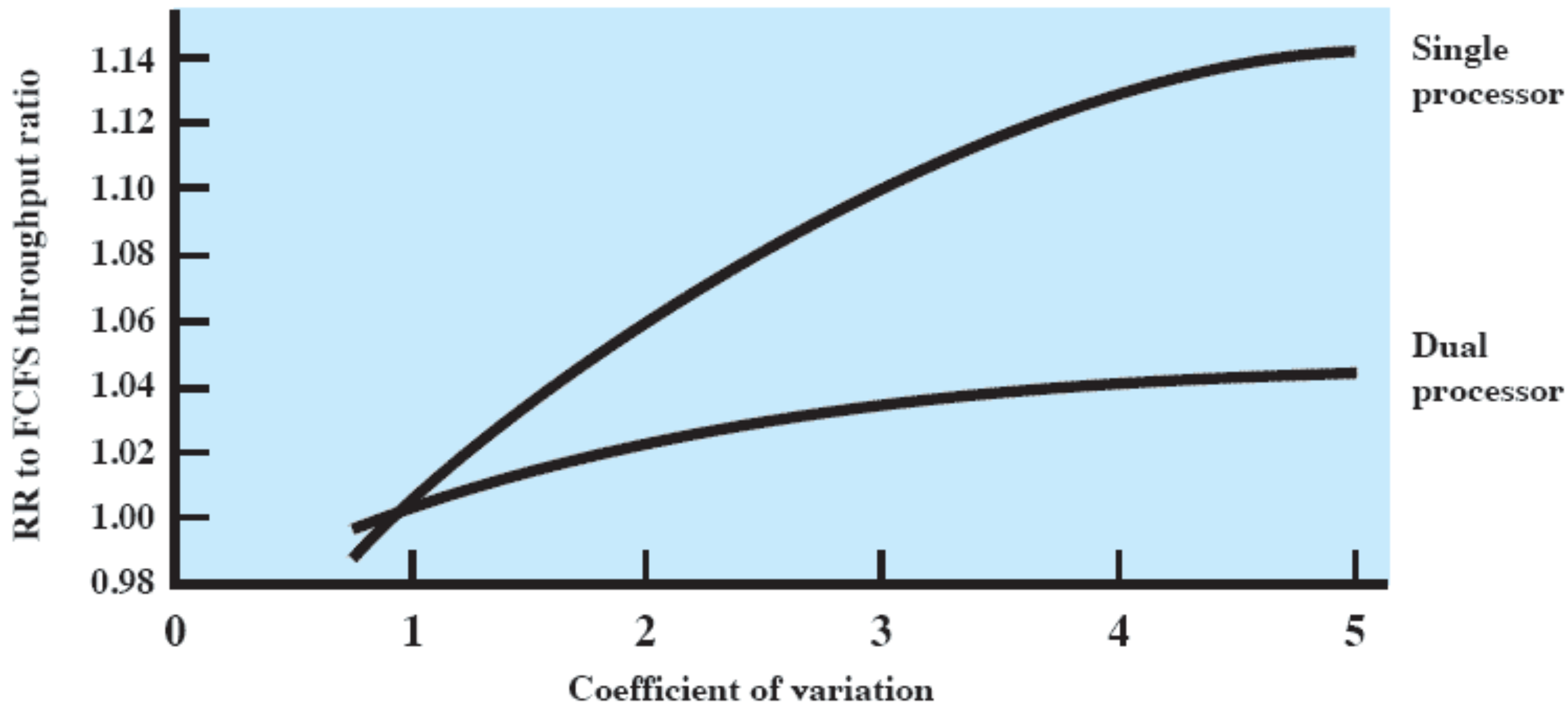
- Do we still need multiprogramming in a multiprocessor environment?
- Consider an 80 core machine
- No definitive answer. Advantages and disadvantages exist.
  – Low system overhead, less complexity
- Btw. in-application concurrency still necessary

Process Dispatching

- Do we still require sophisticated concepts to guide the scheduling decisions?
  - Complex scheduling algorithms
  - Priorities
  - Feedback queues
  - Compute metrics at run time

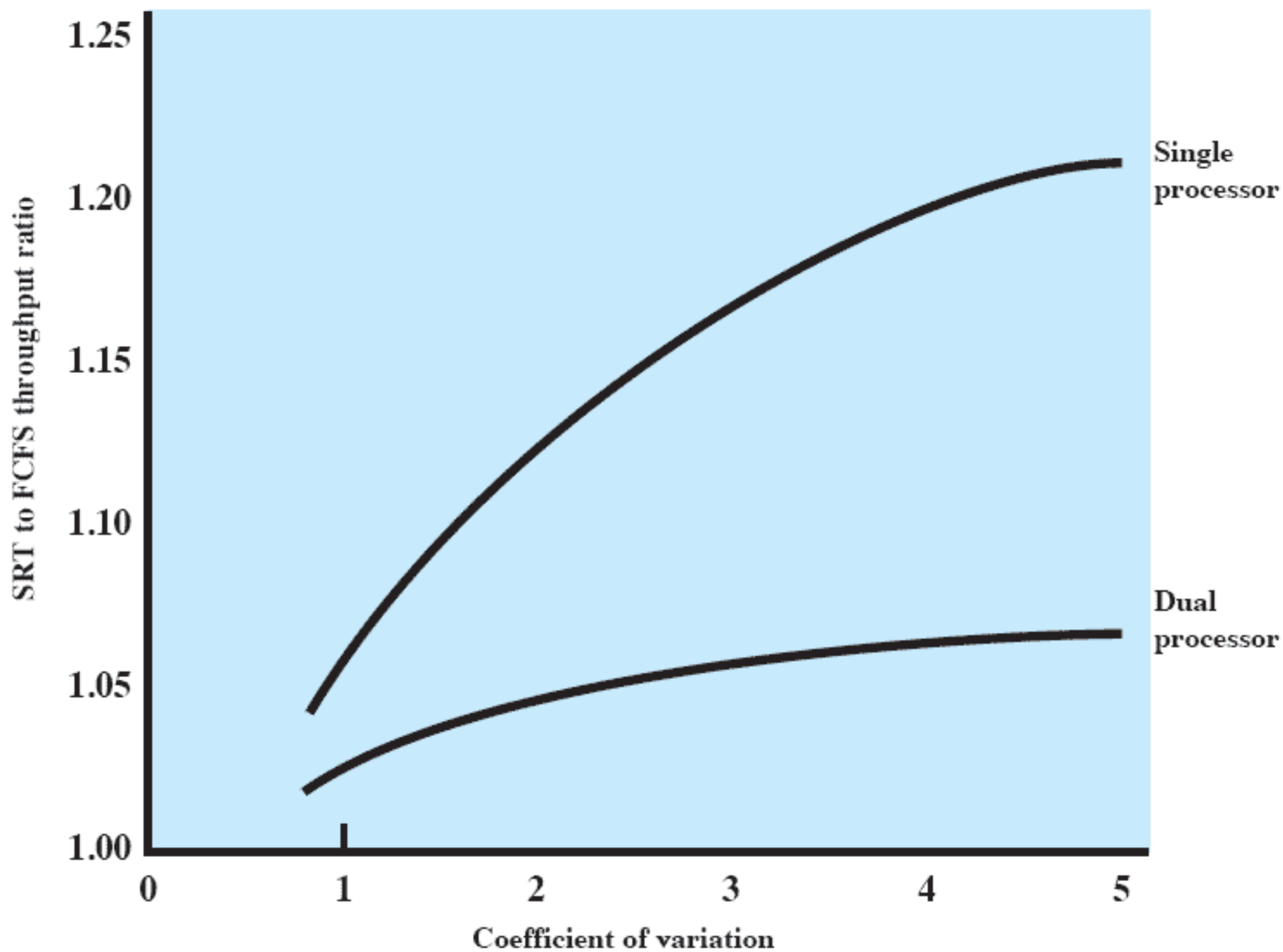
- Some are unnecessary and even counter productive → active area of research

Comparison One and Two Processors



(a) Comparison of RR and FCFS

Comparison One and Two Processors



(b) Comparison of SRT and FCFS

Thread Scheduling

- Separate execution from resource ownership
- An application consists of multiple cooperating, concurrently-executing threads

- Uniprocessor:
  - Program structuring aid
  - Overlap I/O with processing
  - Low management overhead (compared to MP)
- Multiprocessor:
  - True parallelism

Thread Scheduling & Assignment Overview

- Load sharing
  - Processes are not assigned to a particular processor
  - Load sharing vs load balancing
- Gang scheduling
  - A set of related threads is scheduled to run on a set of processors at the same time

Thread Scheduling & Assignment Overview

- Dedicated processor assignment
  - Threads are assigned to a specific processor
- Dynamic scheduling
  - Number of threads can be altered during course of execution

Load Sharing

- Global queue, processor picks process
- Load is distributed evenly across the processors

- Advantages:
  - No processor is left idle while there are processes available.
  - No centralized scheduler required
  - Transparent for the developer

Disadvantages of Load Sharing

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

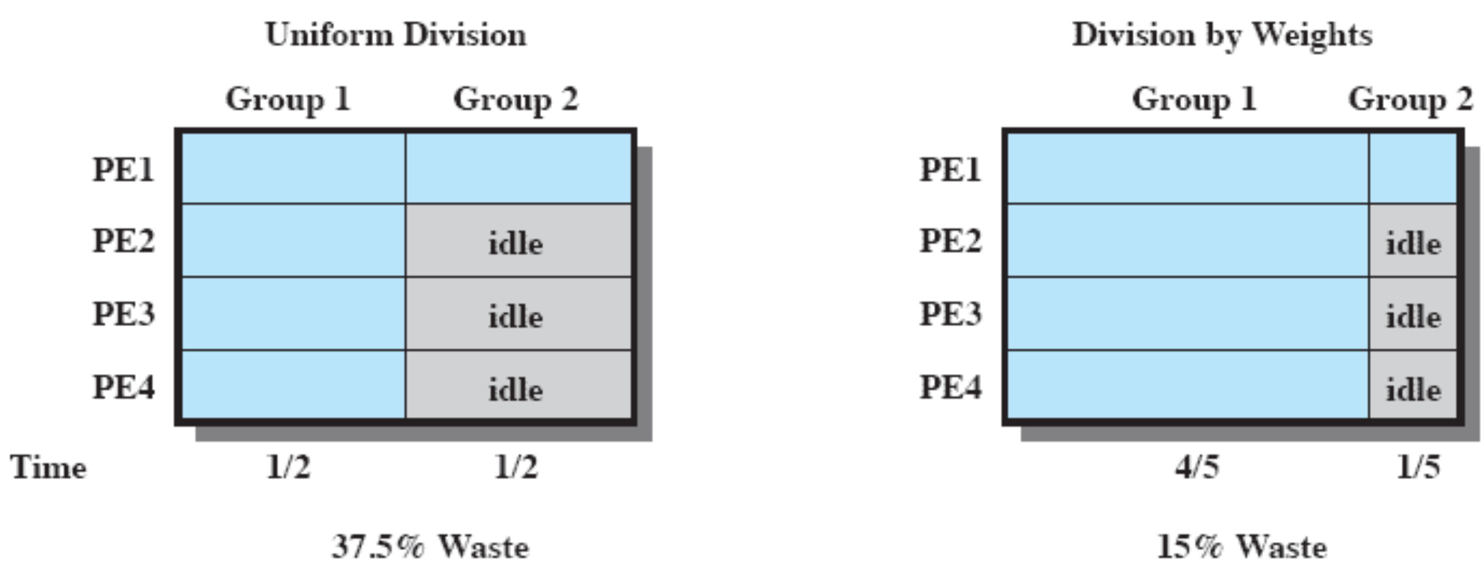
Gang Scheduling

- Threads often need to synchronize with each other
- → run multiple threads to together

- Simultaneous scheduling of threads that make up a single process
- Useful for applications where performance severely degrades when any part of the application is not running

Processor Allocation

- Gang scheduling can lead to inefficient use of the multiprocessor
  - Group independent threads into groups
  - Use weights in the scheduling algorithm



Dedicated Processor Assignment

- Dedicate thread groups to processors until application completes
- No multiprogramming !

- Looks bad: extremely wasteful
- However:
  - Assume 1000 processor cores
  - Zero overhead

Application Speedup

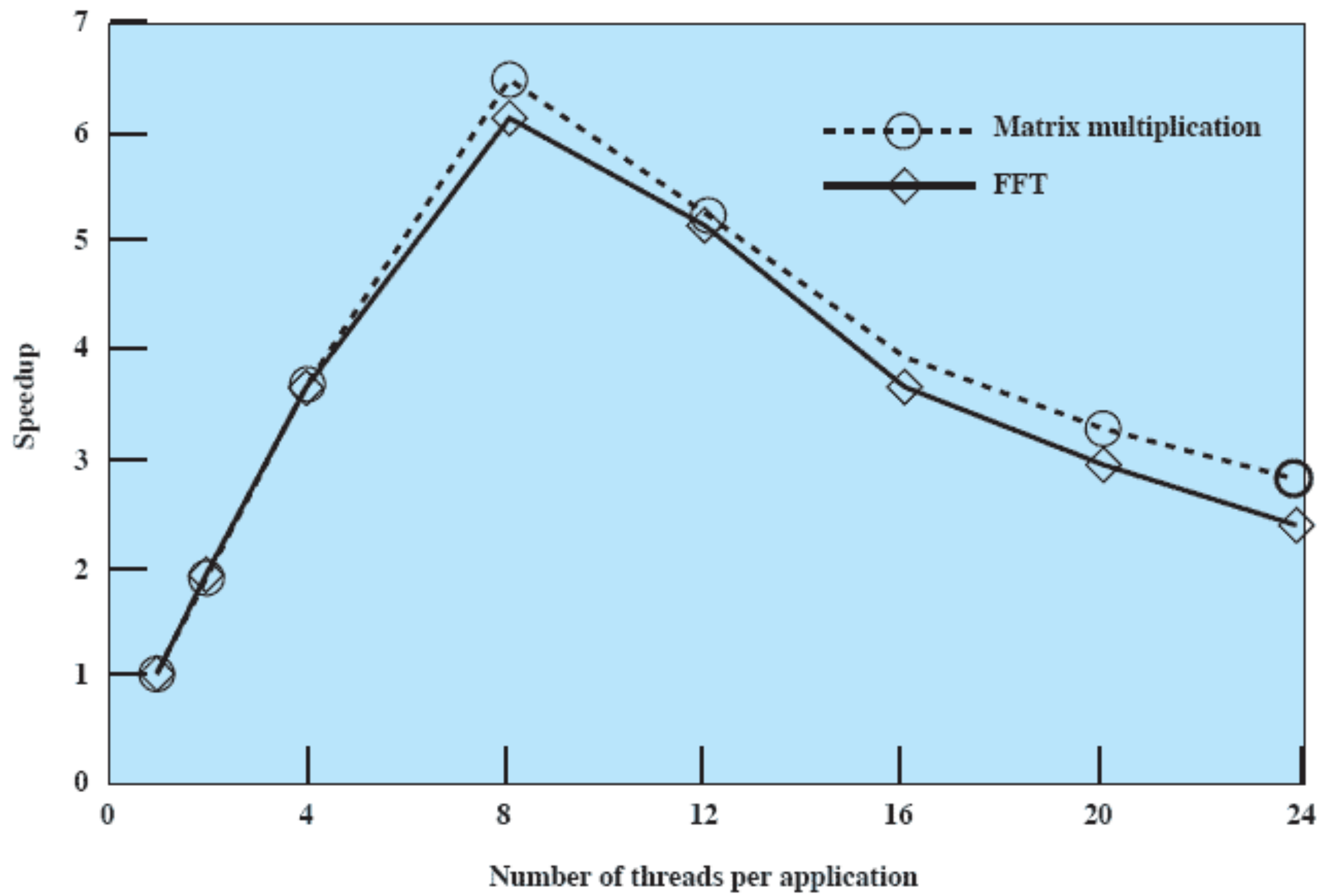


Figure 10.4 Application Speedup as a Function of Number of Threads

Dynamic Scheduling

- Number of threads in a process are altered dynamically by the application
- Requires a layer of indirection which maps computation tasks to threads

Real-Time Scheduling

- Correctness of the system depends on
  - Logic behavior
  - Timing

- **A correct value at the wrong time is a fault.**

- 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

Properties Of RT Systems

- Real-time is about producing the correct result at the right time.

Value	**Timing**	**Result**
Wrong	Too late	Failure
Wrong	On time	Failure
Correct	Too late	Failure
Correct	**On time**	**Ok**

- Temporal constraints are a way to specify, when the value is on time.

Soft Temporal Constraints

- A **soft real-time system** is one where the response time is normally specified as an average value. This time is normally dictated by the business or market.
- A single computation arriving late is not significant to the operation of the system, though many late arrivals might be.
- Ex: Airline reservation system - If a single computation is late, the system's response time may lag. However, the only consequence would be a frustrated potential passenger.

Hard Temporal Constraints

- A **hard real-time system** is one where the response time is specified as an absolute value. This time is normally dictated by the environment.

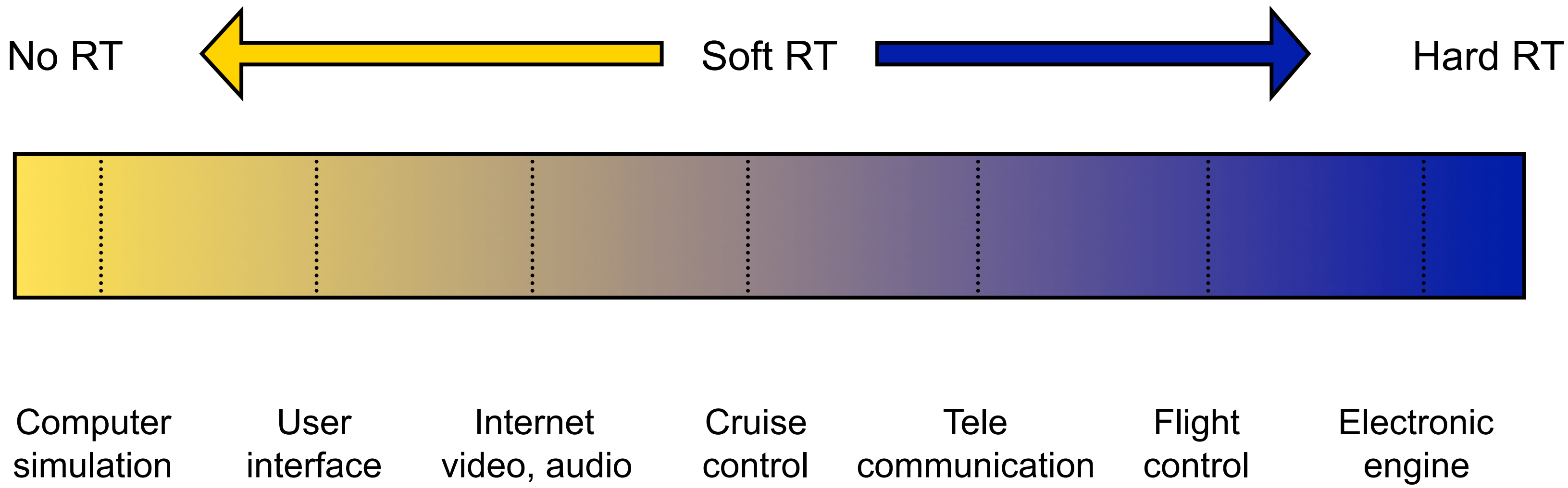
- A system is called a hard real-time if tasks always must finish execution before their deadlines or if message always can be delivered within a specified time interval.

- Hard real-time is often associated with safety critical applications. A failure (e.g. missing a deadline) in a safety-critical application can lead to loss of human life or severe economical damage.

Firm Temporal Constraints

- In a **firm real-time system** timing requirements are a combination of both hard and soft ones. Typically the computation will have a shorter soft requirement and a longer hard requirement.
- Ex: Ventilator – The system must ventilate a patient so many times within a given time period. But a few second delay in the initiation of the patient's breath is allowed, but not more.

Real-Time Spectrum



Characteristics

- Determinism
  - Operations are performed at fixed, predetermined times or within predetermined time intervals
  - Example: 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

Characteristics

- Responsiveness of tasks/jobs
  - How long, after acknowledgment, it takes the operating system to service the interrupt
  - Includes amount of time to begin execution of the interrupt
  - Includes the amount of time to perform the interrupt service routine
  - Includes interference from interrupt nesting

Characteristics

- User control
  - Usually user has little control (priorities, grouping), in an RTOS this is different
  - User specifies priority, importance, timing
  - Manually control memory management (locking pages, specifying resource demands)
  - Manually controlling I/O algorithms (What disk transfer algorithms to use)

Characteristics

- Reliability
  - Degradation of performance may have catastrophic consequences (safety-critical systems)
- Fail-soft operation
  - Ability of a system to fail in such a way as to preserve as much capability and data as possible
  - Graceful degradation
  - Compare to fail-hard, fail-safe, fail-silent

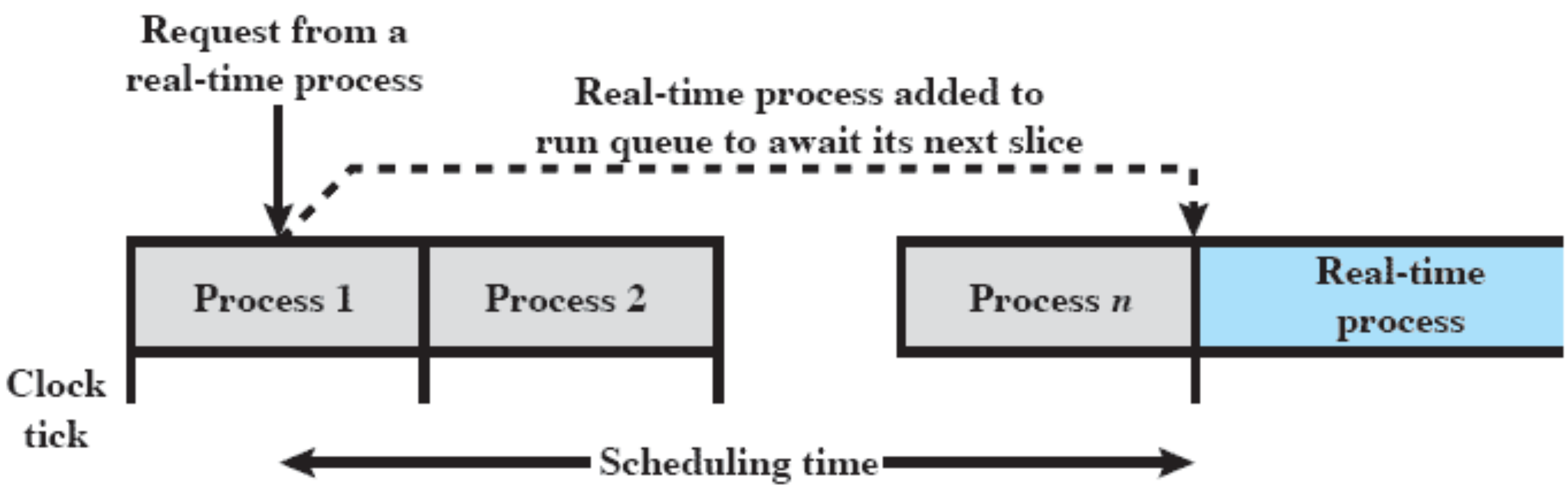
Features of Real-Time OS

- 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

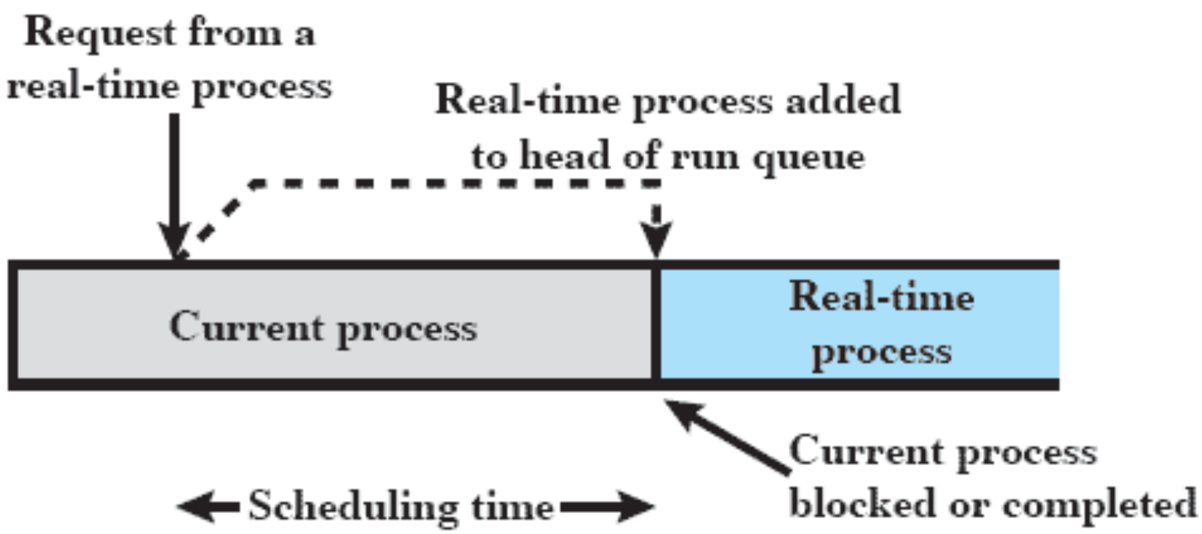
- Use of special sequential files that can accumulate data at a fast rate
- Preemptive scheduling based on priority
- Minimization of intervals during which interrupts are disabled
- Delay tasks for fixed amount of time
- Special alarms and timeouts

Scheduling of Real-Time Process



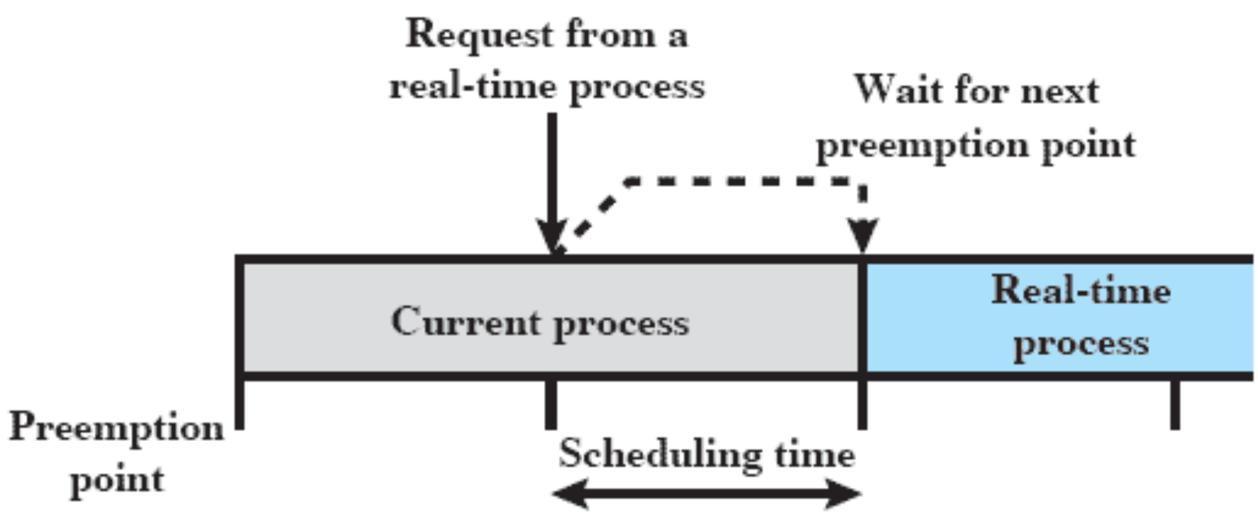
(a) Round-robin Preemptive Scheduler

Scheduling of Real-Time Process



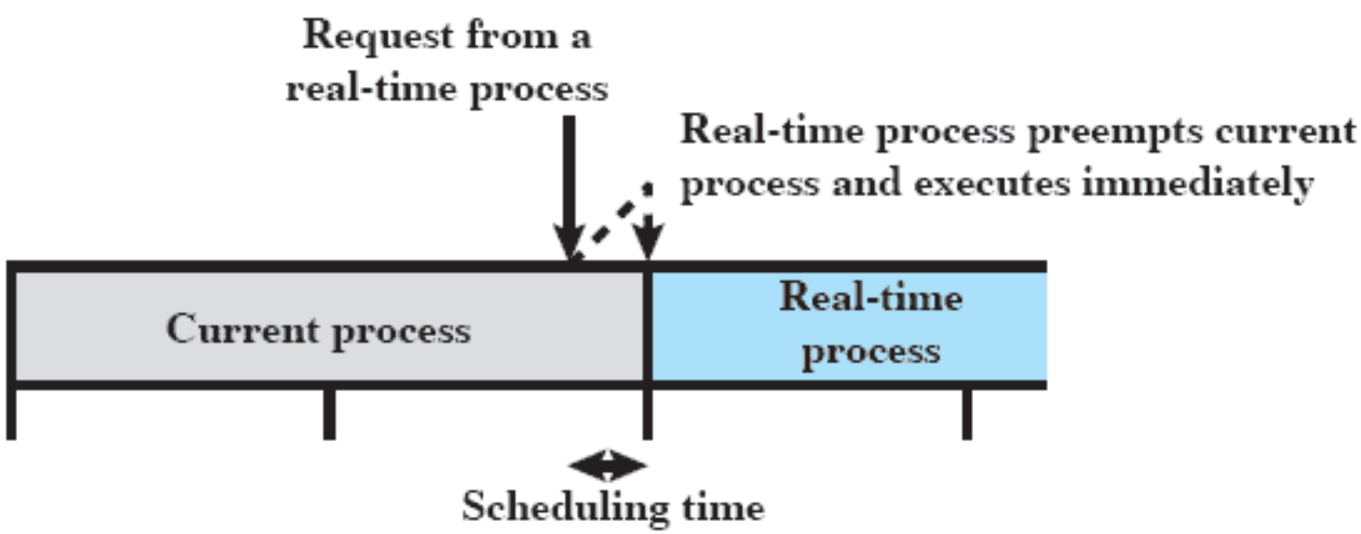
(b) Priority-Driven Nonpreemptive Scheduler

Scheduling of Real-Time Process



(c) Priority-Driven Preemptive Scheduler on Preemption Points

Scheduling of Real-Time Process



(d) Immediate Preemptive Scheduler

Real-Time Scheduling

- Static table-driven
  - Determines at run time when a task begins execution
- 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

Deadline Scheduling

- Real-time applications are not concerned with speed but with completing tasks

Deadline Scheduling

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

Two Periodic Tasks

T=(e,p)
A=(10,20)
B=(25,50)

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

Scheduling with Completion DL

T=(e,p)
A=(10,20)
B=(25,50)

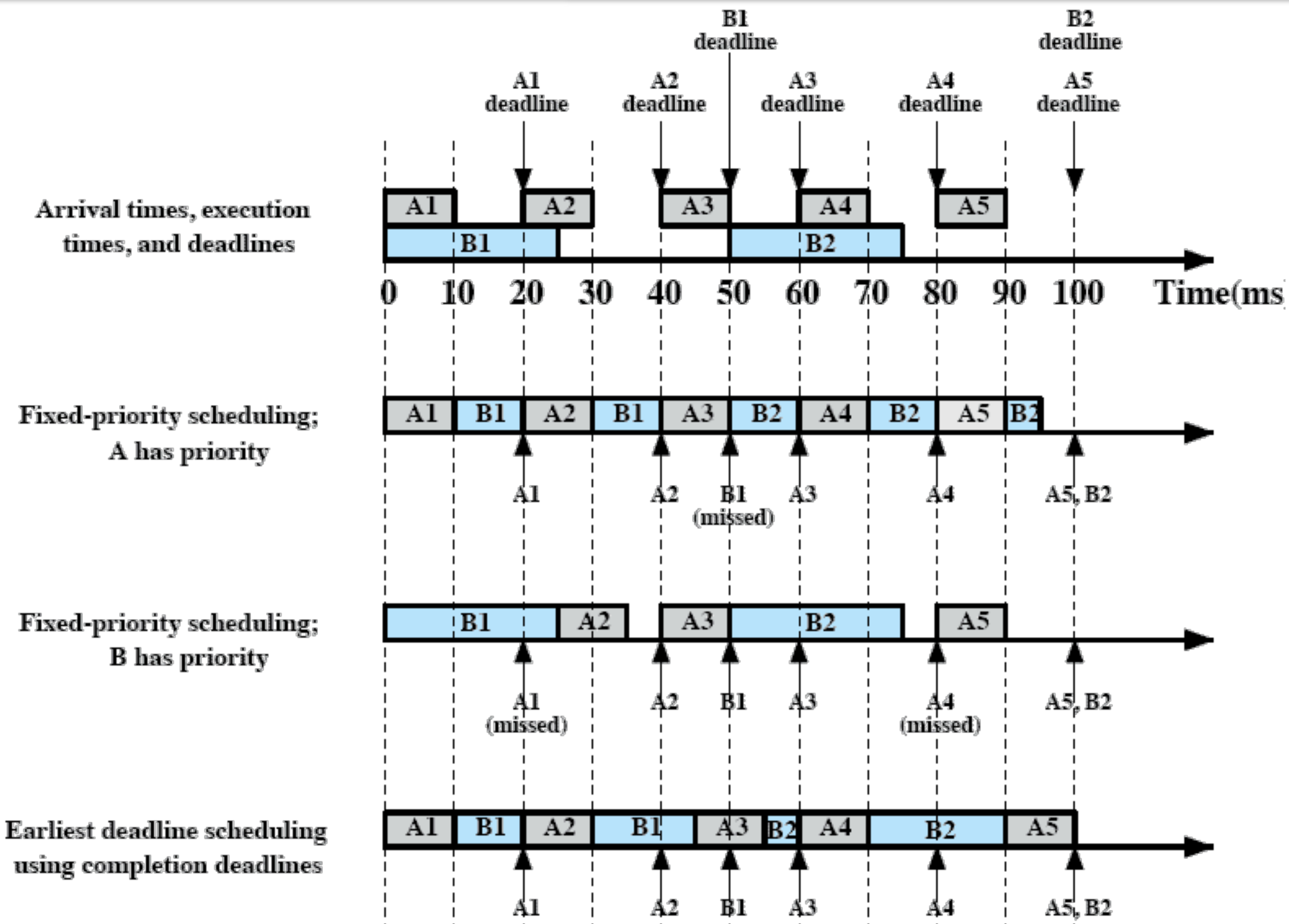


Figure 10.6 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

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

Scheduling with Starting DL

No preemption

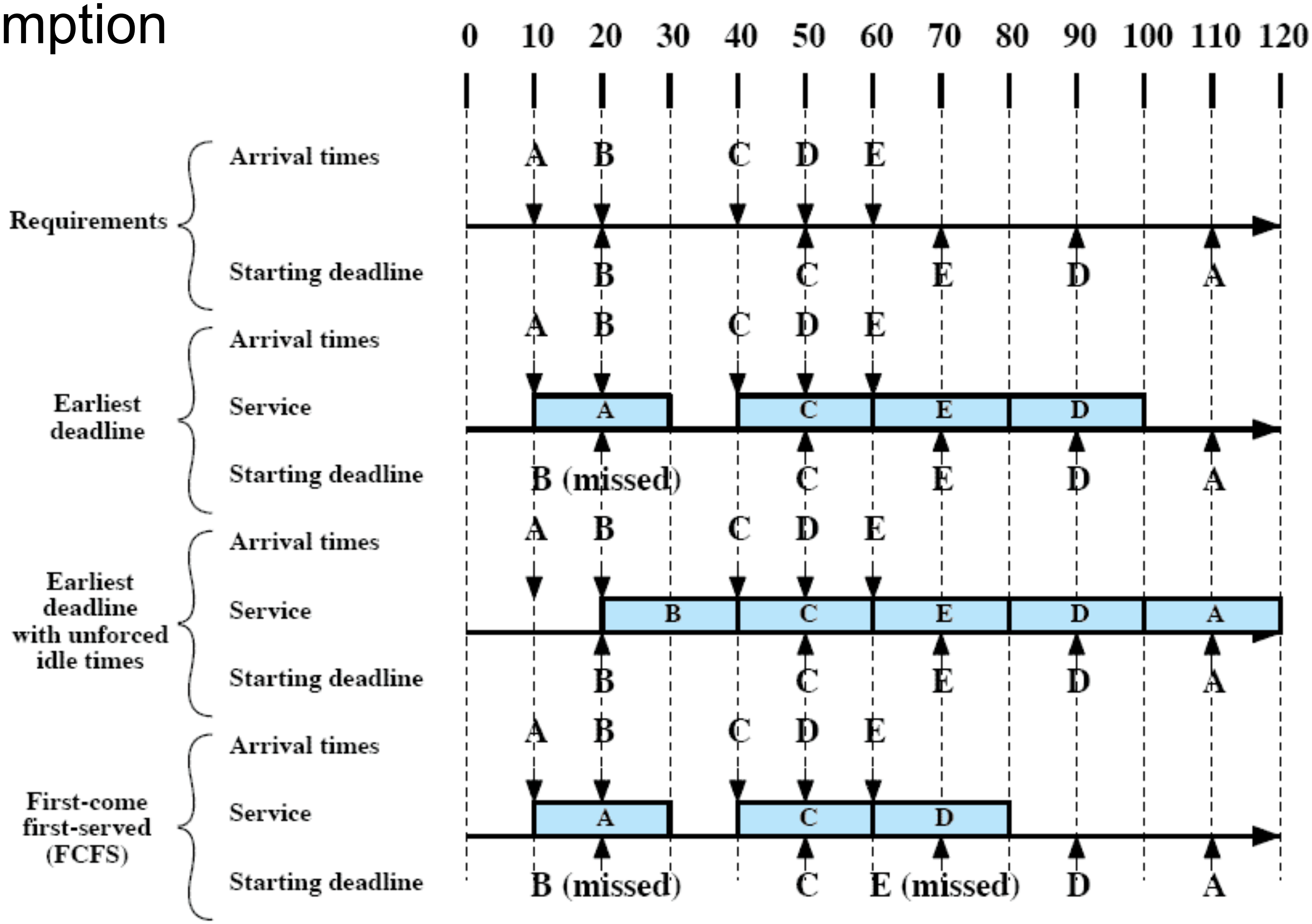


Figure 10.7 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

Rate Monotonic Scheduling

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

- Lower bound on schedulable utilization = 0.693

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

Task Set

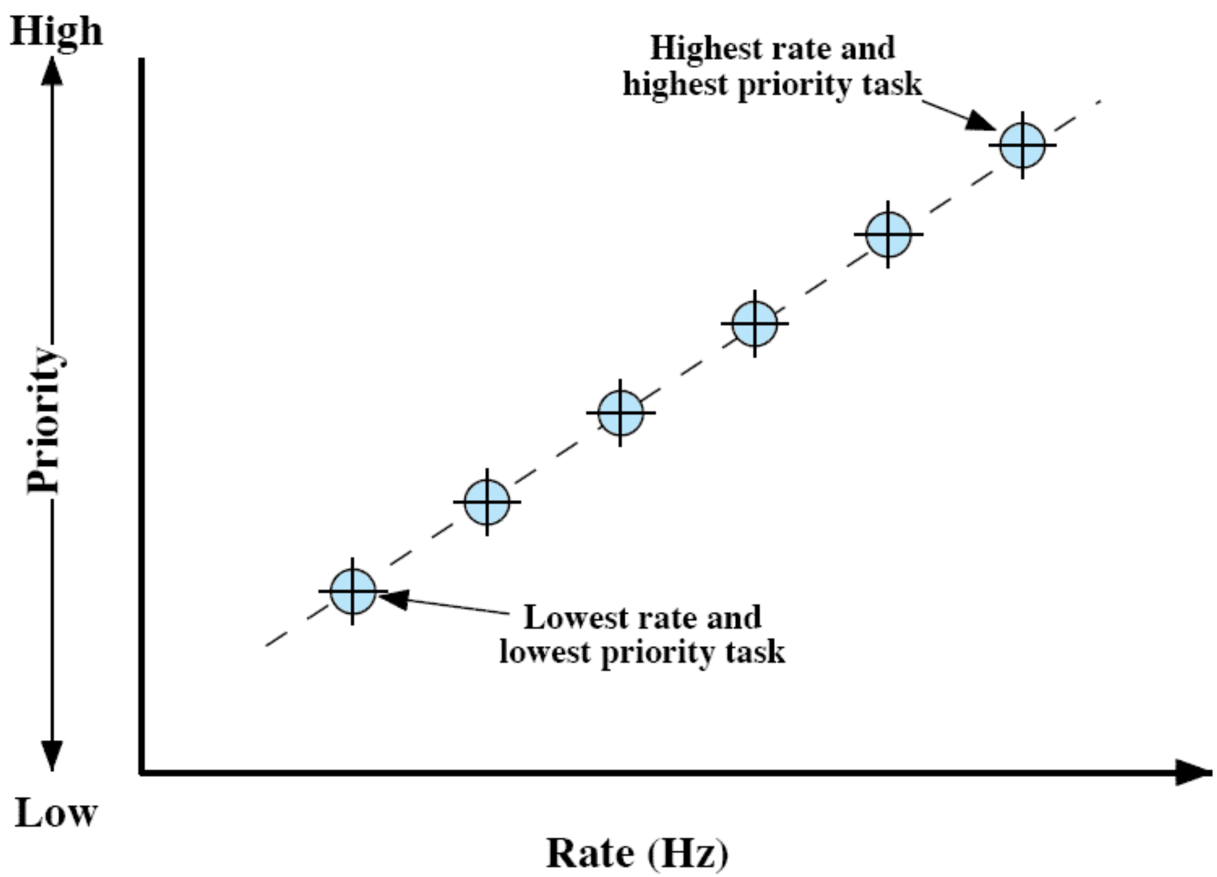


Figure 10.8 A Task Set with RMS [WARR91]

Earliest Deadline First

- Always execute the task with the earliest deadline.

- Optimal (can schedule a CPU utilization of 1), if the system supports preemption

EDF vs. RM

… Or why does industry prefer RM?

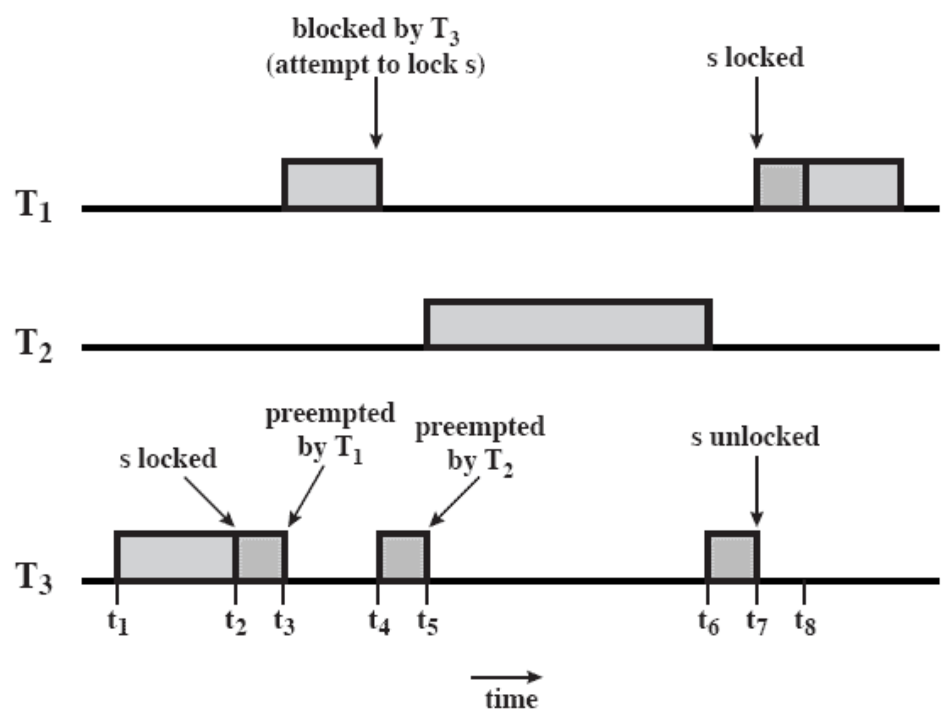
- 30% gain doesn’t make much a difference
  - Processor speeds double
  - Good designs have a safety margin anyways
- Only parts are time critical
  - Use the ~30% for soft real-time tasks
- RM is more predictable in overload situations.
- Often you don’t have deadlines.

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

Unbounded Priority Inversion

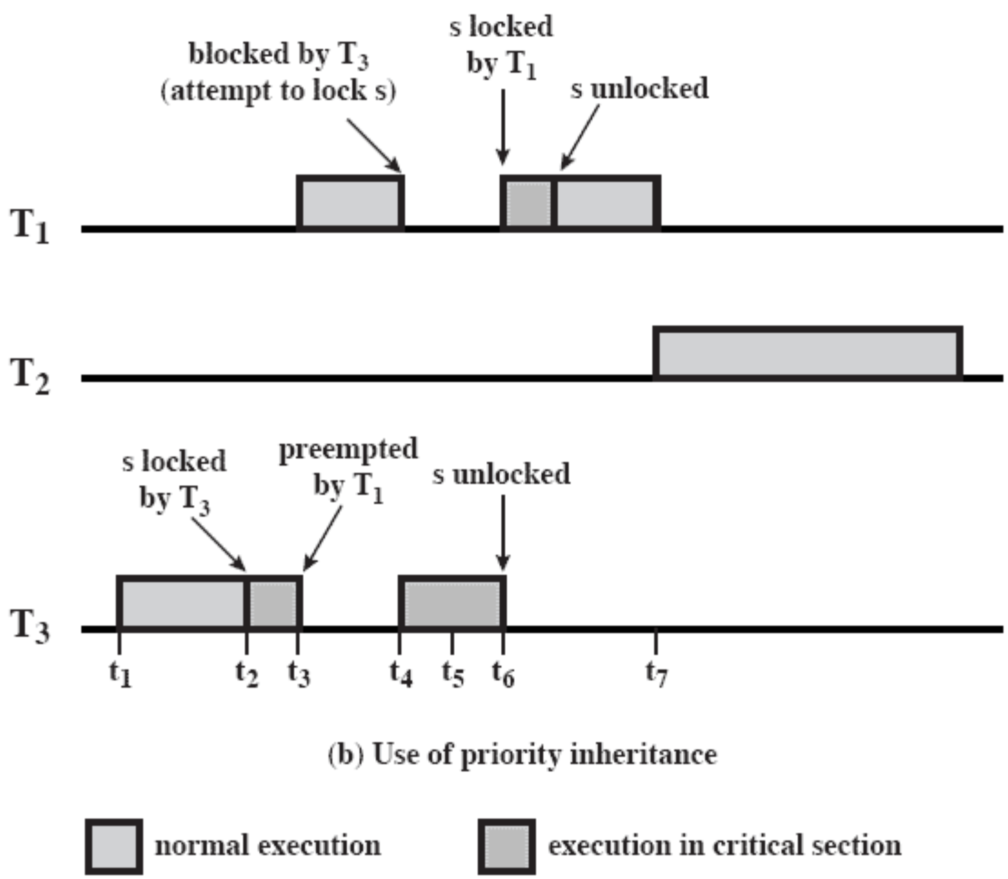
- Duration of a priority inversion depends on unpredictable actions of other unrelated tasks



(a) Unbounded priority inversion

Priority Inheritance

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



(b) Use of priority inheritance