Agenda

- ✓ **Multiprocessor Scheduling**
  - ✓ Granularity
  - ✓ Design Issues
  - ✓ Process Scheduling
  - ✓ Thread Scheduling
- ■ **Real-Time Scheduling**
  - ✓ Background
  - ✓ Characteristics of Real-Time Operating Systems
  - ✓ Real-Time Scheduling
  - ■ Deadline Scheduling
  - ■ Rate Monotonic Scheduling
  - ■ Priority Inversion

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 basic 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
Starting deadline	• time task must begin
Completion deadline	• time 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 relative importance of the task
Subtask scheduler	• a task may be decomposed into a mandatory subtask and an optional subtask

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 **deadline** for collecting data from sensor A must be met every 20 ms, and that for B every 50 ms.

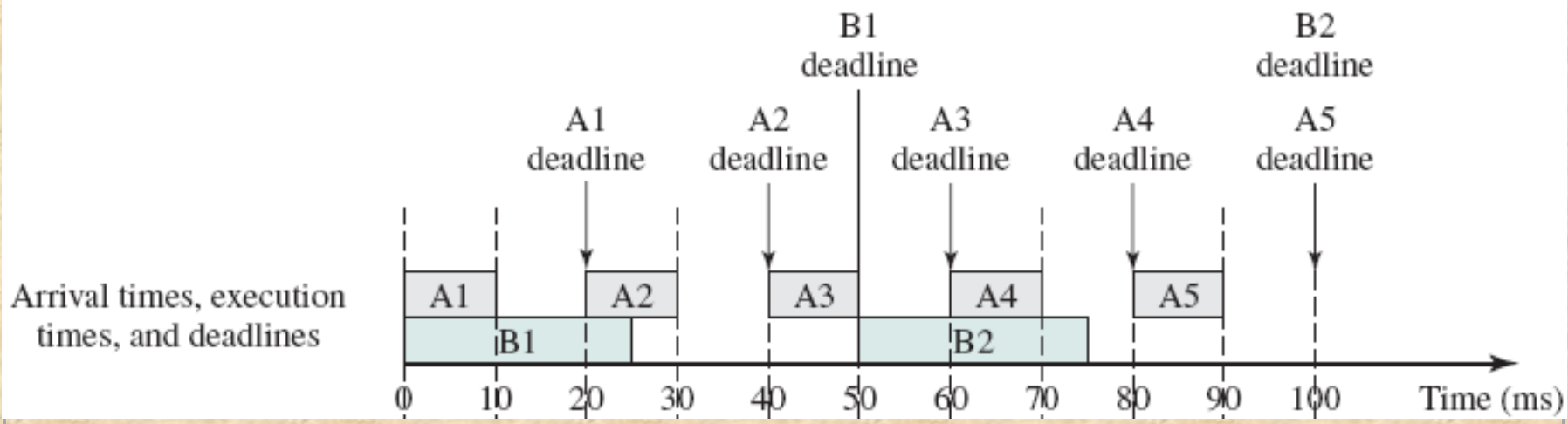
It takes 10 ms, including operating system overhead, to process each sample of data from A and 25 ms to process each sample of data from B.

Table 10.2 summarizes the execution profile of 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 10.5 compares three scheduling techniques using the execution profile of Table 10.2 .

The first row of Figure 10.5 repeats the information in Table 10.2 ; the remaining three rows illustrate three scheduling techniques.


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)

The computer is capable of making a scheduling decision every 10 ms. Suppose that, under these circumstances, we attempted to use a priority scheduling scheme.

The first two timing diagrams in Figure 10.5 show the result.

1. If **A has higher priority**, the first instance of task B is given only 20 ms of processing time, in two 10-ms chunks, by the time its deadline is reached, and thus fails.

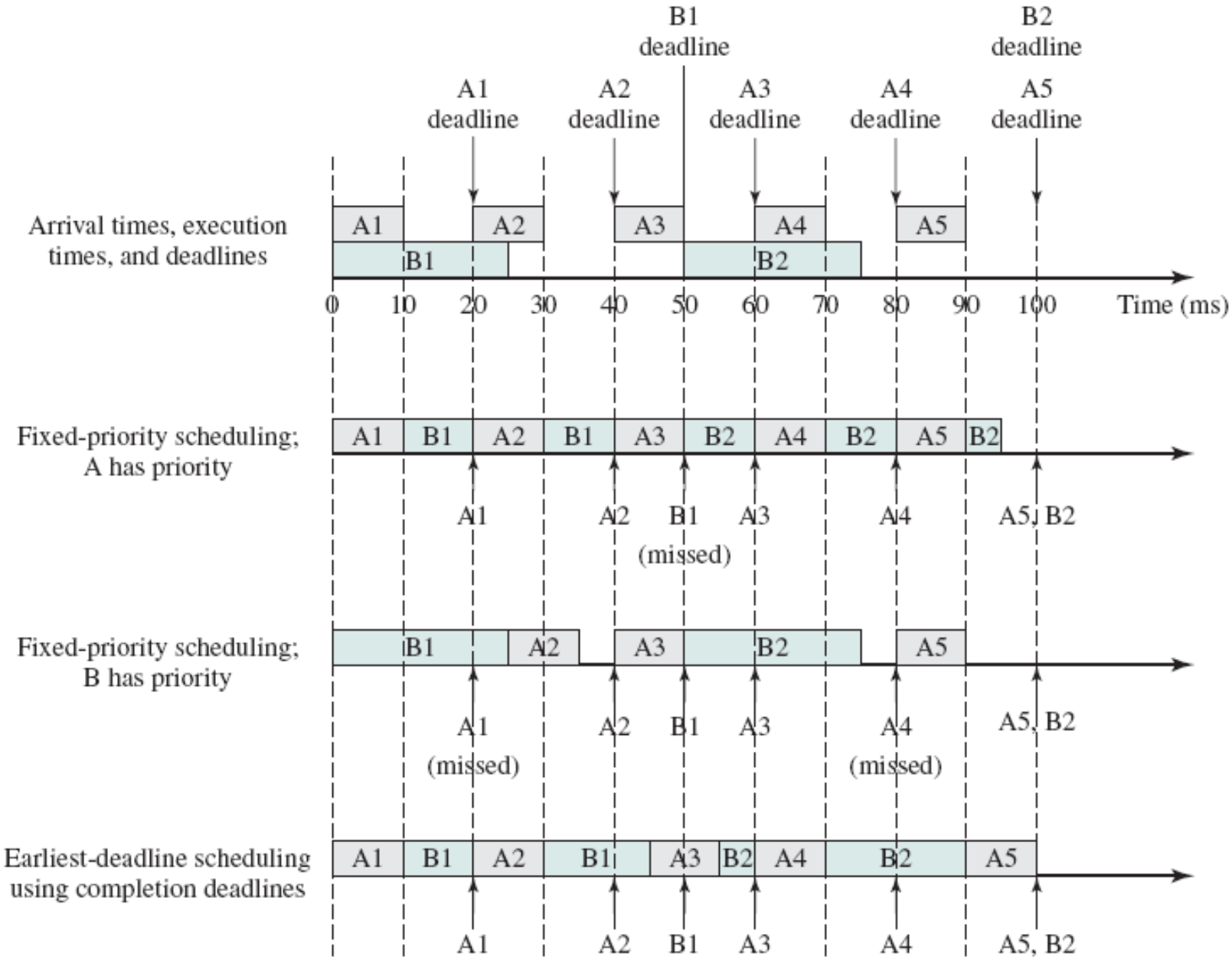


Figure 10.5 **Scheduling of Periodic Real-Time Tasks with Completion Deadlines (Based on Table 10.2)**

2. If B is given higher priority, then A will miss its first deadline.

3. The final timing diagram shows the use of earliest-deadline scheduling.

At time *t = 0 , both A1 and B1 arrive.*

Because A1 has the earliest deadline, it is scheduled first.

When A1 completes, B1 is given the processor.

At *t = 20 , A2 arrives. Because A2 has an earlier deadline than B1, B1 is interrupted* so that A2 can execute to completion.

Then B1 is resumed at *t = 30 .*

At t = 40 , A3 arrives.

However, B1 has an earlier ending deadline and is allowed to execute to completion at *t = 45 .*

A3 is then given the processor and finishes at t = 55 .

In this example, by scheduling to give priority at any preemption point to the task with the nearest deadline, all system requirements can be met.

Because the tasks are **periodic** and **predictable**, a static table-driven scheduling approach is used.

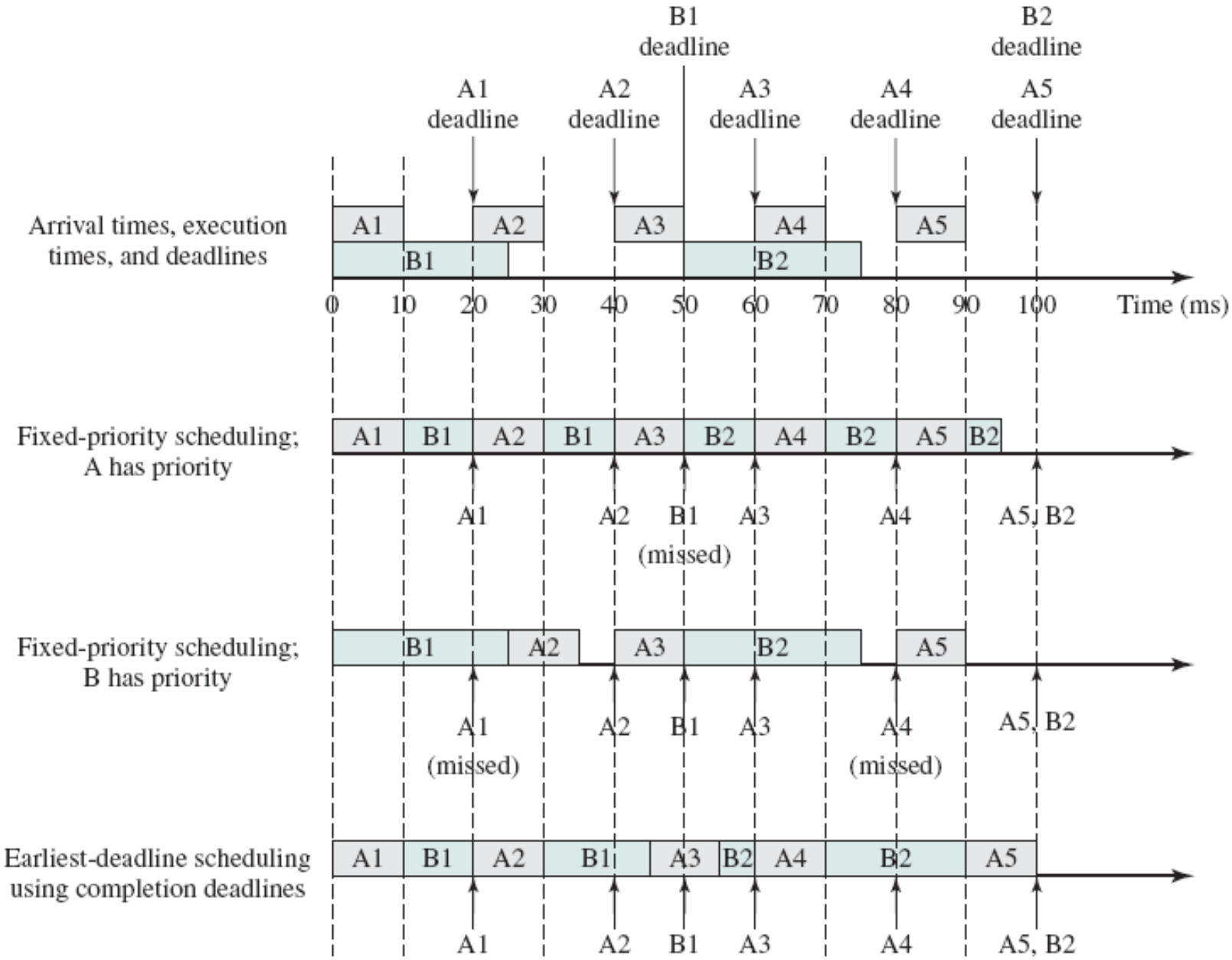


Figure 10.5 **Scheduling of Periodic Real-Time Tasks with Completion Deadlines (Based on Table 10.2)**

Now consider a scheme for dealing with **aperiodic** tasks with starting deadlines. The top part of Figure 10.6 shows the arrival times **and starting deadlines** for an **example** consisting of five tasks each of which has an execution time of 20 ms. Table 10.3 summarizes the execution profile of the five 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

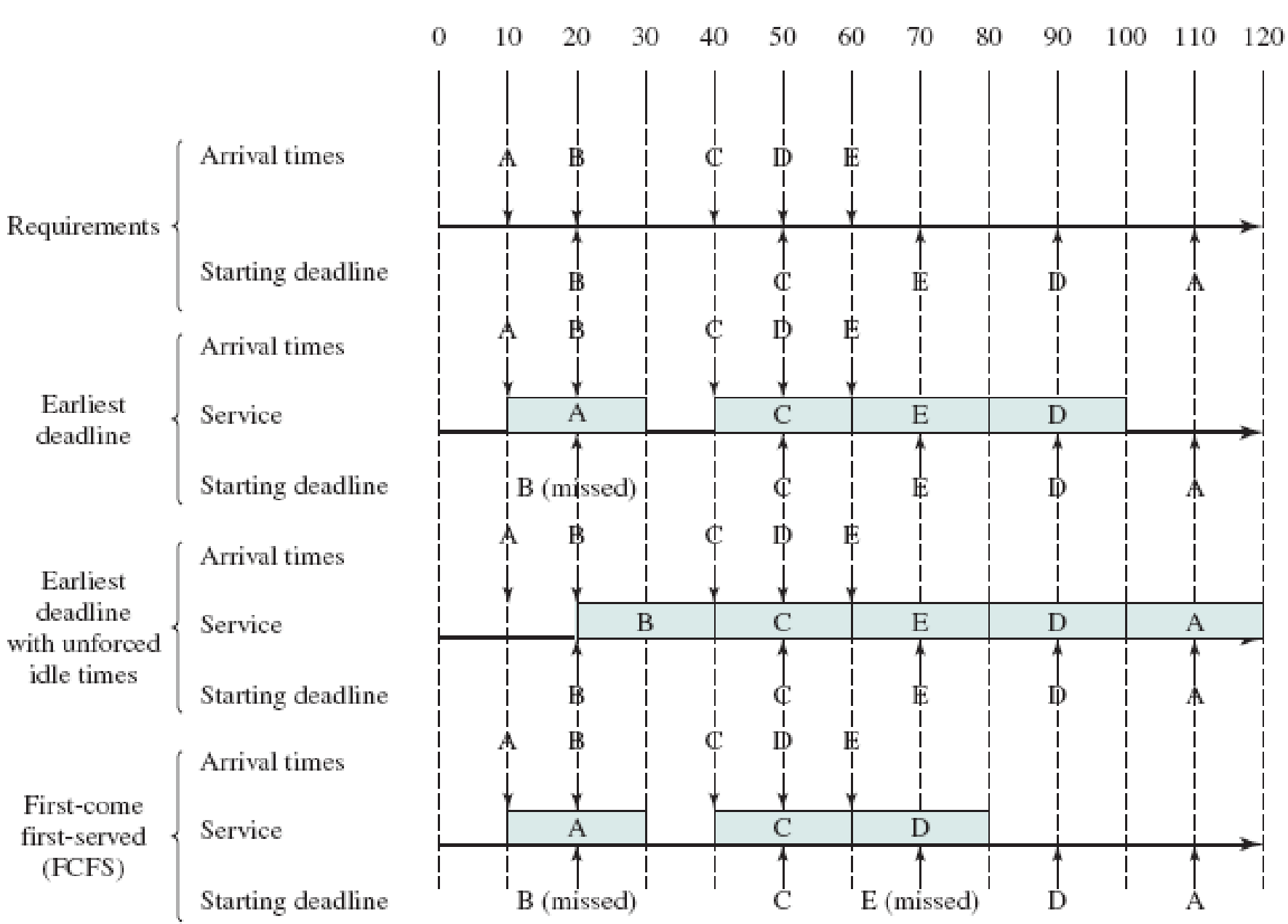


Figure 10.6 Scheduling of Aperiodic Real-Time Tasks with Starting Deadlines

When this approach (earliest deadline) is used in the example of Figure 10.6 , note that **although task B requires immediate service**, the service is denied.
This is the risk in dealing with aperiodic tasks, especially with starting deadlines.

A refinement of the policy will improve performance if deadlines can be known in advance of the time that a task is ready.

This policy, referred to as earliest deadline with unforced idle times, operates as follows: **Always schedule the eligible task with the earliest deadline and let that task run to completion.**

Rate Monotonic Scheduling

- One of the more promising methods of resolving multitask scheduling conflicts for periodic tasks is rate monotonic scheduling (RMS).

- RMS assigns priorities to tasks on the basis of their periods.

For RMS, 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.

If we plot the priority of tasks as a function of their rate, the result is a **monotonically** increasing function (Figure 10.7); hence the name, rate monotonic scheduling.

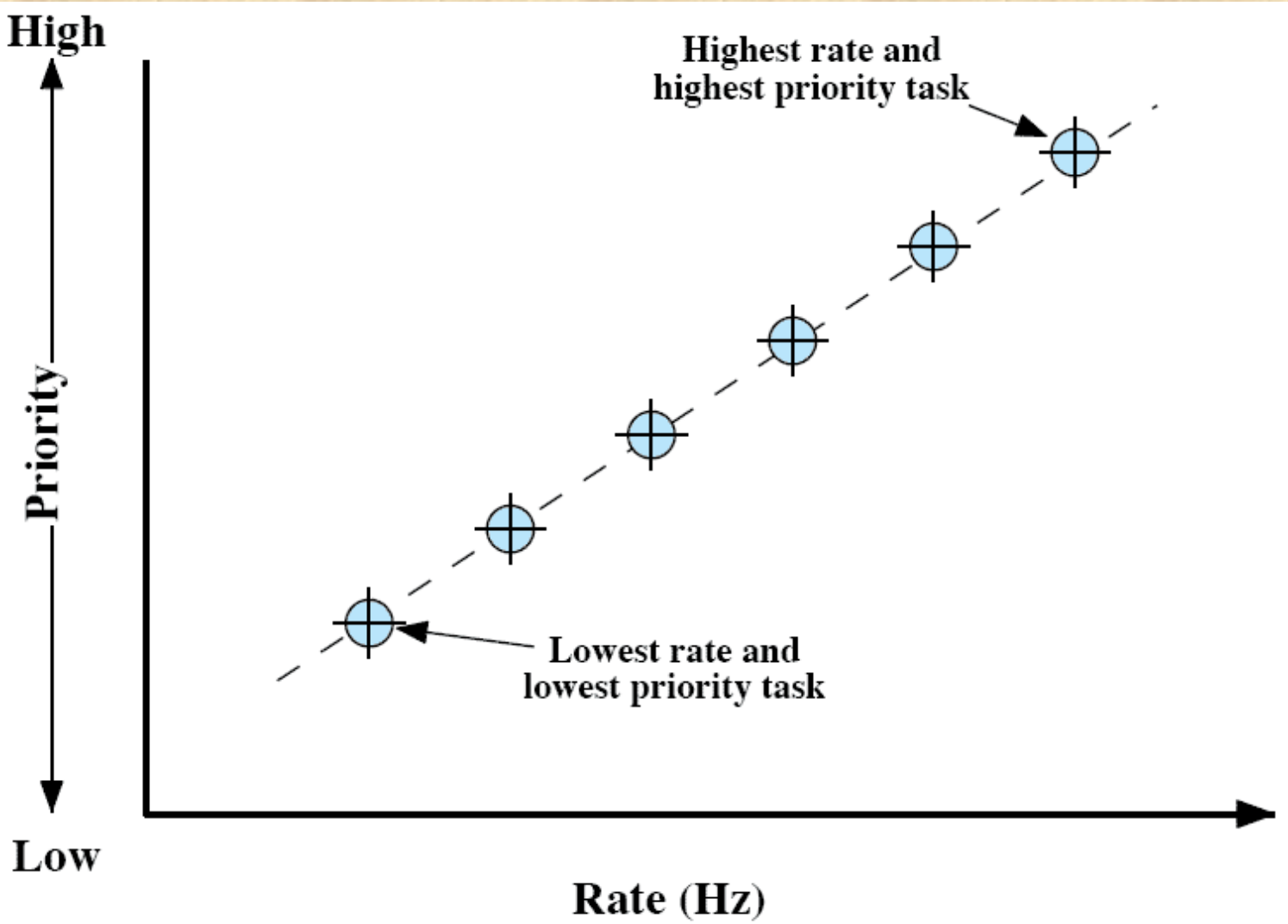


Figure 10.7

Figure 10.8 A Task Set with RMS [WARR91]

Periodic Task Timing Diagram

Figure 10.8 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).

Periodic Task Timing Diagram

For example, a task with a period of 50 ms occurs at a rate of 20 Hz. Typically, the end of a task's period is also the task's hard deadline, although some tasks may have earlier deadlines.

The execution (or computation) time, ***C****, is the amount of processing time required for each occurrence of* the task.

It should be clear that in a **uniprocessor** system, the execution time must be no greater than the period (must have ***C less than or equal T*** *).*

Periodic Task Timing Diagram

If a periodic task is always run to completion, that is, if no instance of the task is ever denied service because of insufficient resources, then the utilization of the processor by this task is

U = C/T

For example, if a task has a period of 80 ms and an execution time of 55 ms, its processor utilization is 55/80 = 0.6875 .

Periodic Task Timing Diagram

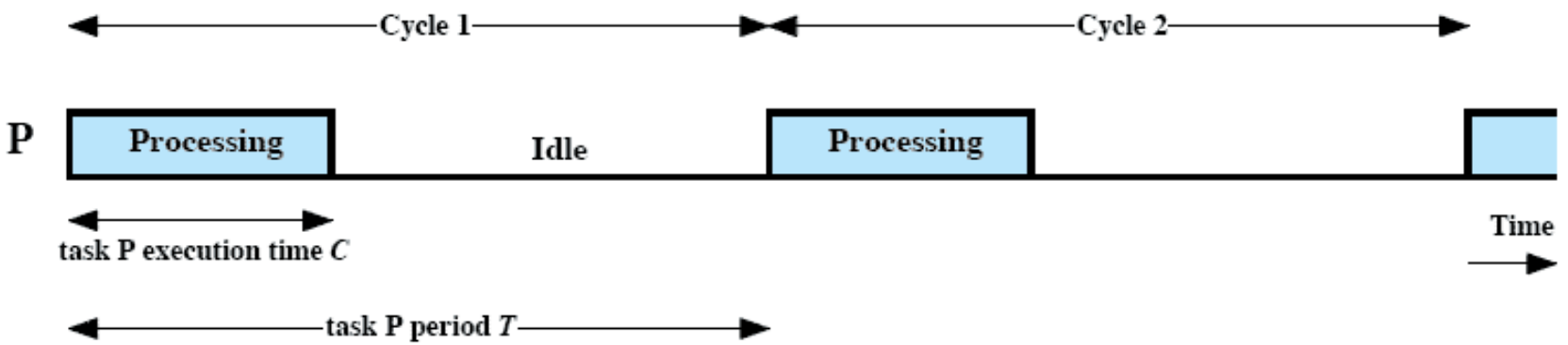


Figure 10.8 **Periodic Task Timing Diagram**

One measure of the effectiveness of a periodic scheduling algorithm is **whether or not it guarantee**s that all hard deadlines are met.

Suppose that we have ***n*** *tasks,* each with a fixed period and execution time.

Then for it to be possible to meet all deadlines, the following inequality must hold:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 1$$

The sum of the processor utilizations of the individual tasks cannot exceed a value of 1, which corresponds to total utilization of the processor.

Equation (10.1) provides a bound on the number of tasks that a perfect scheduling algorithm can successfully schedule.

For any particular algorithm, the bound may be lower. For RMS, it can be shown that the following inequality holds:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

Table 10.4 gives some values for this upper bound. As the number of tasks increases, the scheduling bound converges to *ln 2 ~ 0.693*

As an example, consider the case of **three** periodic tasks, where *Ui = Ci/Ti :*

- **Task P 1 : *C1 = 20 ; T1 = 100 ; U1 = 0.2***
- **Task P 2 : *C2 = 40 ; T2 = 150 ; U2 = 0.267***
- **Task P 3 : *C3 = 100 ; T3 = 350 ; U3 = 0.286***

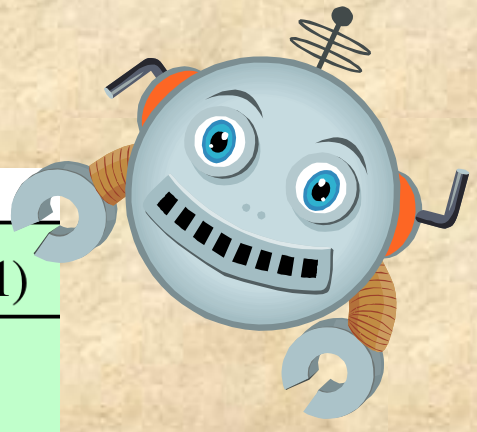
The total utilization of these three tasks is $0.2 + 0.267 + 0.286 = 0.753$. The upper bound for the schedulability of these three tasks using RMS is

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \leq n(2^{1/3} - 1) = 0.779$$

Value of the RMS Upper Bound

Table 10.4

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



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


JPL

The best-known instance of priority inversion involved the Mars Pathfinder mission.

This rover robot landed on Mars on July 4, 1997 and began gathering and transmitting voluminous data back to Earth.

But a few days into the mission, the **lander software began experiencing total system resets**, each resulting **in losses of data.**

After much effort by the Jet Propulsion Laboratory (JPL) team that built the Pathfinder, the problem was traced to **priority inversion** [JONE97].

The Pathfinder software included the following three tasks, in decreasing order of priority:

T 1 : Periodically checks the health of the spacecraft systems and software

T 2 : Processes image data

T 3 : Performs an occasional test on equipment status

- After T1 executes, it reinitializes a timer to its maximum value.
- If this timer ever expires, it is assumed that the integrity of the lander software has somehow been compromised.
- The processor is halted, all devices are reset, the software is completely reloaded, the spacecraft systems are tested, and the system starts over.
- This recovery sequence does not complete until the next day.
- T 1 and T 3 **share a common data structure**, protected by a binary semaphore s .
- Figure 10.9a shows the sequence that caused the priority inversion:

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.

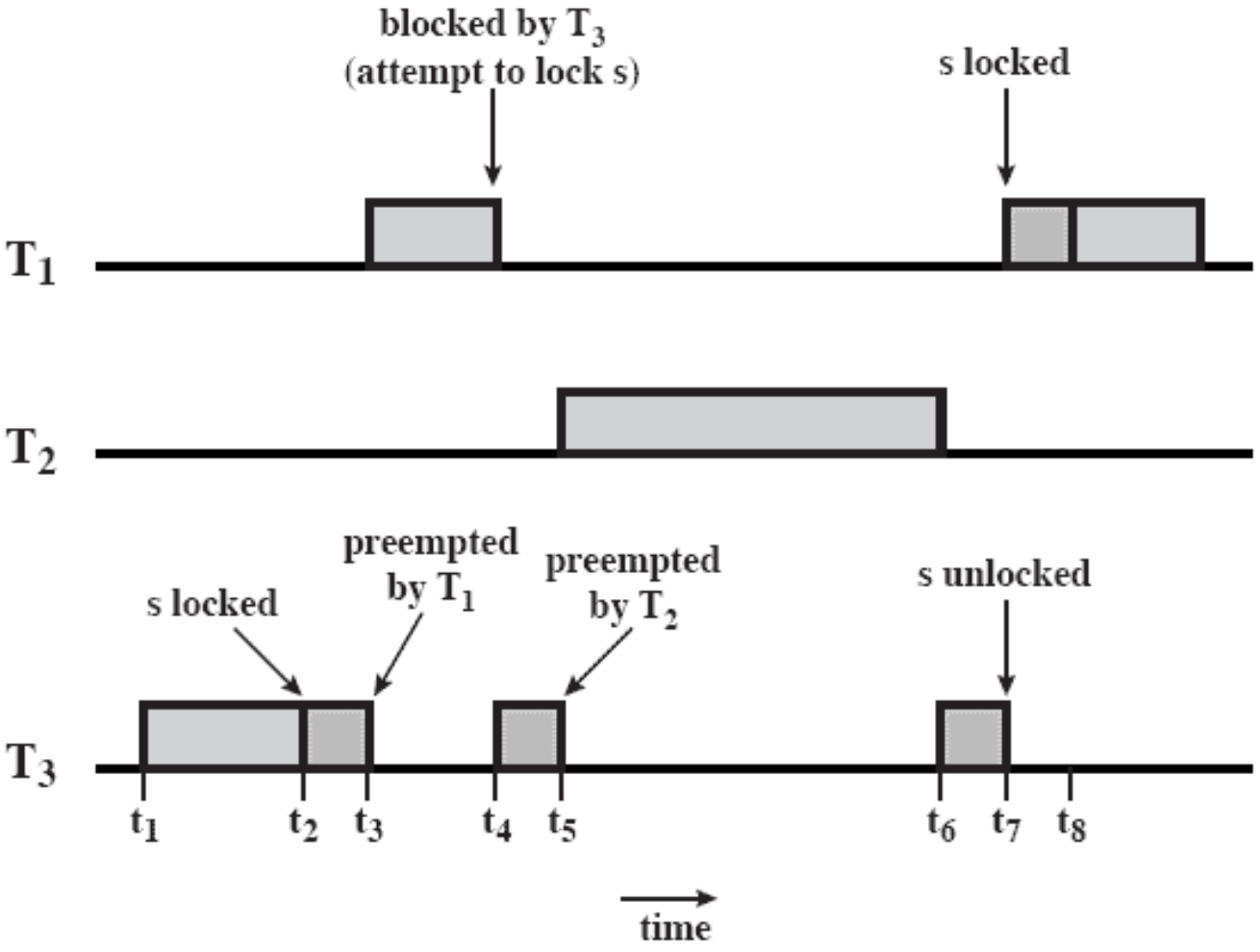
t 4 : T 1 attempts to enter its critical section but is blocked because the semaphore is locked by T 3 ; T 3 resumes execution in its critical section.

t 5 : T 2 , which has a higher priority than T 3 , preempts T 3 and begins executing.

t 6 : T 2 is suspended for some reason unrelated to T 1 and T 3 ; T 3 resumes.

t 7 : T 3 leaves its critical section and unlocks the semaphore. T1preempts T 3, locks the semaphore and enters its critical section.

Unbounded Priority Inversion



(a) Unbounded priority inversion

In practical systems, two alternative approaches are used to avoid unbounded priority inversion: priority inheritance protocol and priority ceiling protocol.

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.

Figure 10.9b shows that priority inheritance resolves the problem of unbounded priority inversion illustrated in Figure 10.9a .

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.

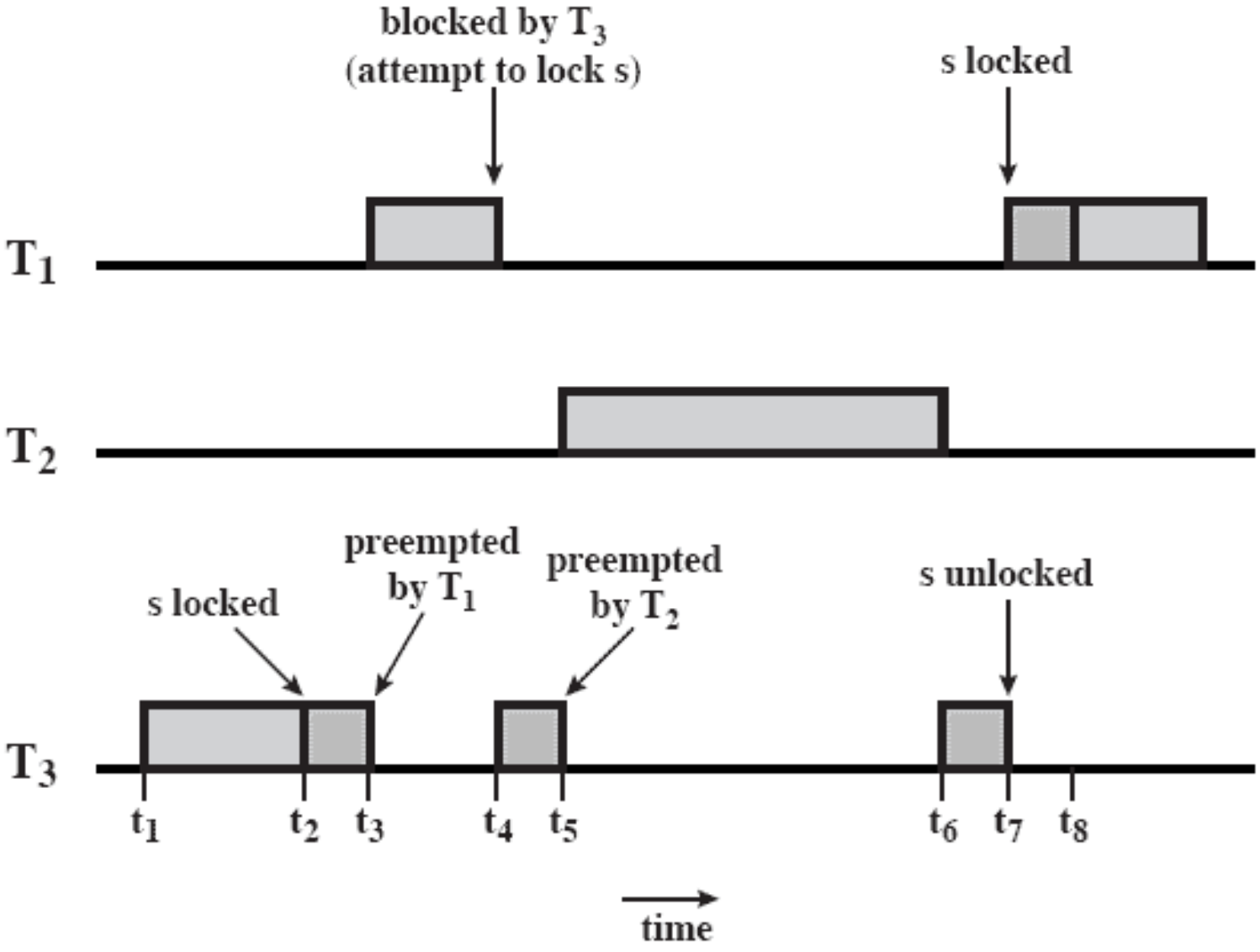
t_4: T_1 attempts to enter its critical section but is blocked because the semaphore is locked by T_3. T_3 is immediately and temporarily assigned the same priority as T_1. T_3 resumes execution in its critical section.

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.

Priority Inheritance



(a) Unbounded priority inversion

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.

Look for

- **Linux Scheduling**
- **Linux Virtual Machine Process Scheduling**

Sheet 2

Review Questions

10.1 List and briefly define five different categories of synchronization granularity.
10.2 List and briefly define four techniques for thread scheduling.
10.3 List and briefly define three versions of load sharing.
10.4 What is the difference between hard and soft real-time tasks?
10.5 What is the difference between periodic and aperiodic real-time tasks?
10.6 List and briefly define five general areas of requirements for a real-time operating system.
10.7 List and briefly define four classes of real-time scheduling algorithms.
10.8 What items of information about a task might be useful in real-time scheduling?

Problems

10-1, 10-2, 10-7