Unit	Details		Lectures
I	**Operating System Overview:** Objectives and Functions, Evolution, Achievements, Modern Operating Systems, Fault tolerance, OS design considerations for multiprocessor and multicore, overview of different operating systems **Processes:** Process Description and Control.		**12**
II	**Threads, Concurrency:** Mutual Exclusion and Synchronization.		**12**
III	**Concurrency:** Deadlock and Starvation, **Memory:** Memory Management, Virtual Memory.		**12**
IV	**Scheduling:** Uniprocessor Scheduling, Multiprocessor and Real-Time Scheduling		**12**
V	**IO and File Management:** I/O Management and Disk Scheduling, File Management, **Operating System Security.**		**12**

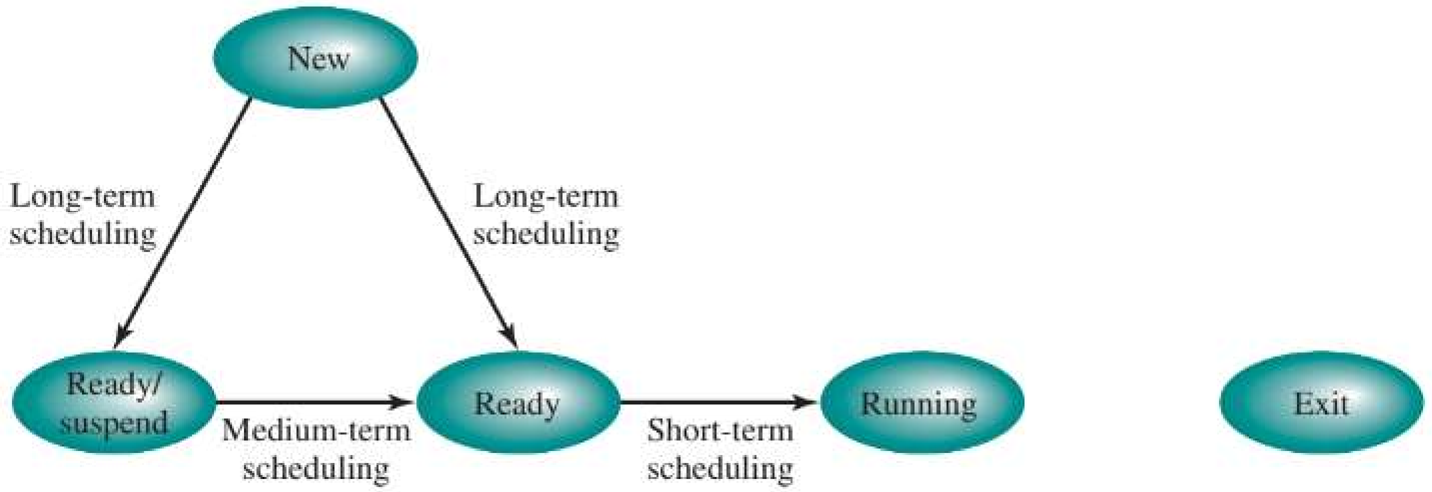
Scheduling: Uniprocessor Scheduling

In a multiprogramming system, multiple processes exist concurrently in main memory. Each process alternates between using a processor and waiting for some event to occur, such as the completion of an I/O operation. The processor or processors are kept busy by executing one process while the others wait.

The key to multiprogramming is scheduling. In fact, four types of scheduling are typically involved.

Table 9.1 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device


New
Long-term scheduling
Long-term scheduling
Ready/ suspend
Medium-term scheduling
Ready
Short-term scheduling
Running
Exit

TYPES OF PROCESSOR SCHEDULING

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput, and processor efficiency. In many systems, this scheduling activity is broken down into three separate functions: long-, medium-, and short-term scheduling

Long-term scheduling determines when new processes are admitted to the system.

Medium-term scheduling is part of the swapping function and determines when a program is brought partially or fully into main memory so that it may be executed.

Short-term scheduling determines which ready process will be executed next by the processor.

Long-term Scheduling

The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming. Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

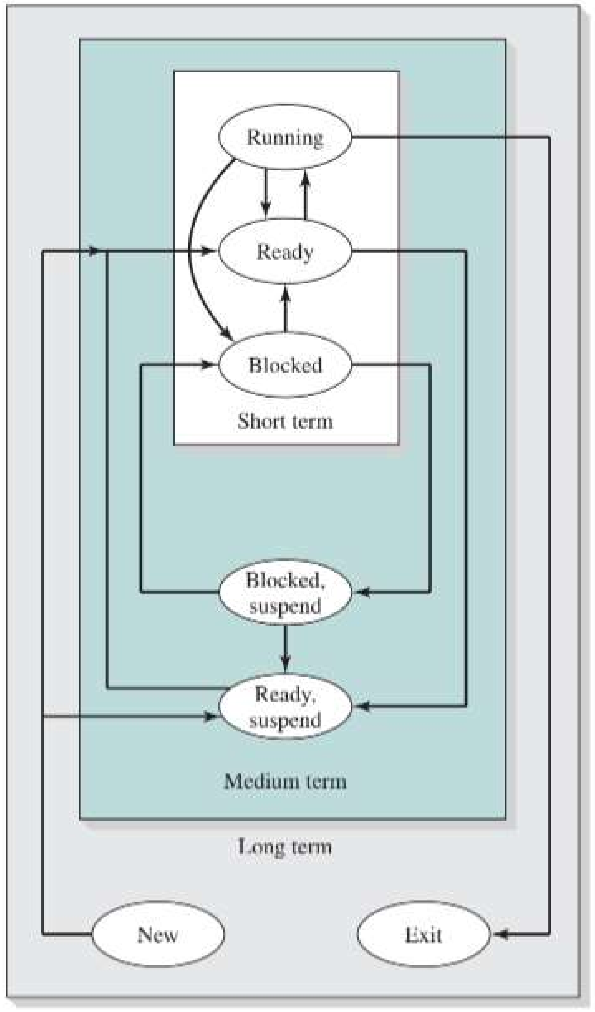


Figure 9.2 Levels of Scheduling

Long-term Scheduling

There are **two** decisions involved. **The scheduler must decide when the OS can take on one or more additional processes. And the scheduler must decide which job or jobs to accept and turn into processes.**

The more processes that are created, the smaller is the percentage of time that each process can be executed . Thus, the long-term scheduler may limit the degree of multiprogramming to provide satisfactory service. Each time a job terminates, the scheduler may decide to add one or more new jobs.

Medium-Term Scheduling

Medium-term scheduling is part of the swapping function. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

Short-Term Scheduling

In terms of frequency of execution, the long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process and which one to take. The medium-term scheduler is executed somewhat more frequently to make a swapping decision.

The short-term scheduler, also known as the **dispatcher**, executes most frequently and makes the fine-grained decision of which process to execute next.

The short-term scheduler is invoked whenever an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favor of another. Examples of such events include:

- Clock interrupts
- I/O interrupts
- Operating system calls
- Signals (e.g., semaphores)

SCHEDULING ALGORITHMS

Short-Term Scheduling Criteria

The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior. Generally, a set of criteria is established against which various scheduling policies may be evaluated.

The commonly used criteria can be categorized along two dimensions. First, we can make a distinction between user-oriented and system-oriented criteria.

User-oriented criteria relate to the behavior of the system as perceived by the individual user or process. An example is response time in an interactive system. **Response time is the elapsed time between the submission of a request until the response begins to appear as output.** This quantity is visible to the user and is naturally of interest to the user.
We would like a scheduling policy that provides “good” service to various users. In the case of response time, a threshold may be defined, say two seconds. Then a goal of the scheduling mechanism should be to maximize the number of users who experience an average response time of two seconds or less.

Other criteria are system oriented. That is, the focus is on effective and efficient utilization of the processor. An example is **throughput**, **which is the rate at which processes are completed**. This is certainly a worthwhile measure of system performance and one that we would like to maximize. However, it focuses on system performance rather than service provided to the user. Thus, throughput is of concern to a system administrator but not to the user population.

Scheduling Criteria

Response time

For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user’s point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.

Scheduling Criteria

Turnaround time This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.

Throughput

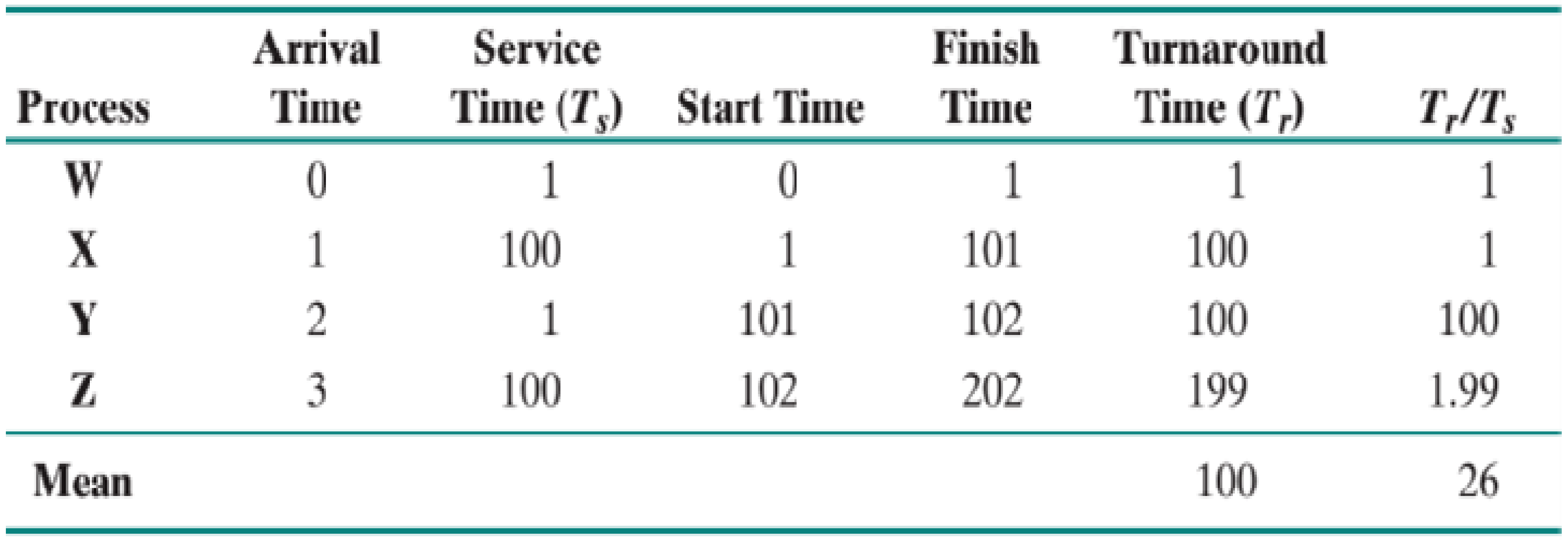
The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.

FIRST-COME-FIRST-SERVED

The simplest scheduling policy is first-come-first-served (FCFS), also known as first-in-first-out (FIFO) or a strict queueing scheme. As each process becomes ready, it joins the ready queue. When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running.

FCFS performs much better for long processes than short ones.

Consider the following example:



Process	Arrival Time	Service Time (T_s)	Start Time	Finish Time	Turnaround Time (T_r)	T_r/T_s
W	0	1	0	1	1	1
X	1	100	1	101	100	1
Y	2	1	101	102	100	100
Z	3	100	102	202	199	1.99
Mean					100	26

ROUND ROBIN

A straightforward way to reduce the penalty that short jobs suffer with FCFS is to use preemption based on a clock. The simplest such policy is round robin. A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis.

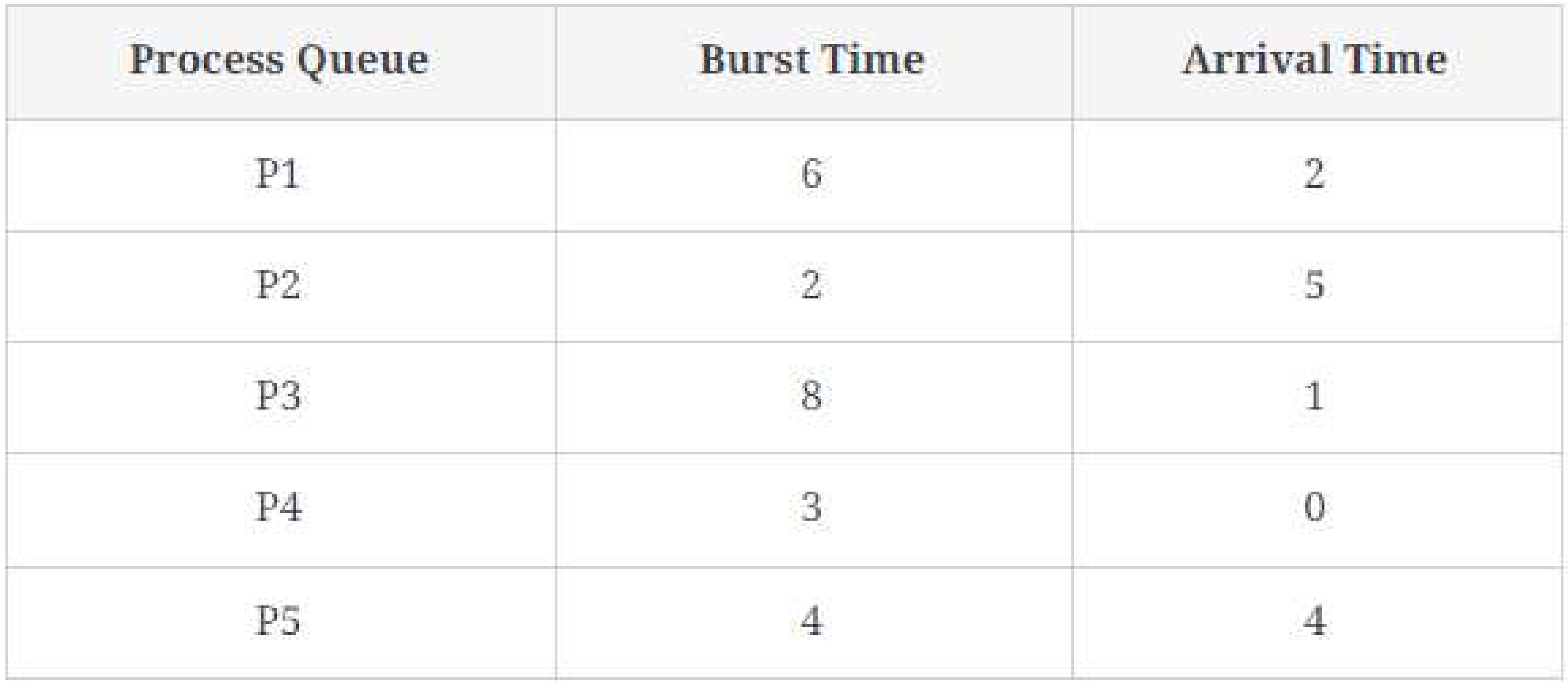
This technique is also known as **time slicing** , because each process is given a slice of time before being preempted.

With round robin, the principal **design issue** is the length of the time quantum, or slice, to be used. If the quantum is very short, then short processes will move through the system relatively quickly. On the other hand, there is processing overhead involved in handling the clock interrupt and performing the scheduling and dispatching function.

SHORTEST PROCESS NEXT

Another approach to reducing the bias in favor of long processes inherent in FCFS is the shortest process next (SPN) policy. This is
a nonpreemptive policy in which the process with the shortest expected processing time is selected next. Thus, a short process will jump to the head of the queue past longer jobs.

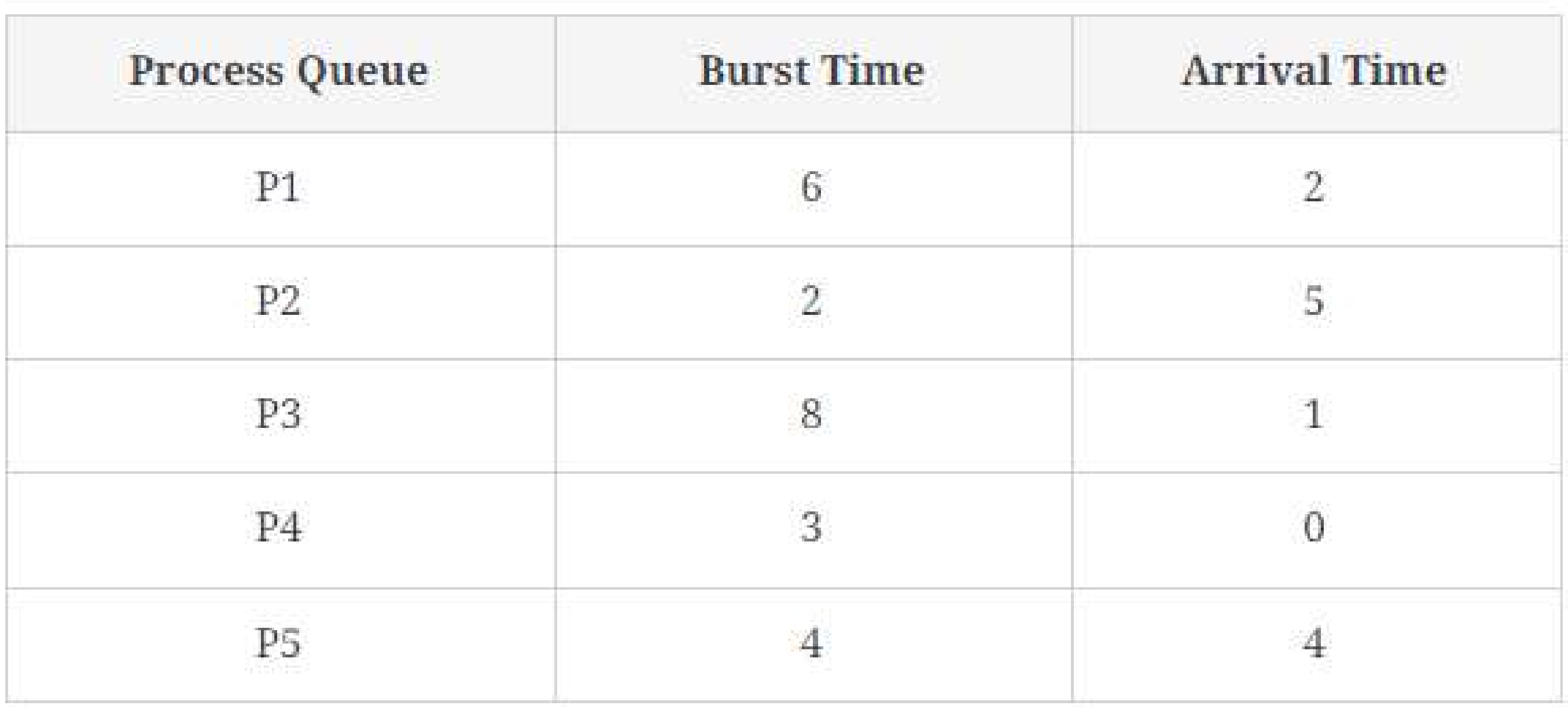
Example



Process Queue	Burst Time	Arrival Time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

SHORTEST REMAINING TIME The shortest remaining time (SRT) policy is a preemptive version of SPN. In this case, the scheduler always chooses the process that has the shortest expected remaining processing time. When a new process joins the ready queue, it may in fact have a shorter remaining time than the currently running process. Accordingly, the scheduler may preempt the current process when a new process becomes ready. As with SPN, the scheduler must have an estimate of processing time to perform the selection function, and there is a risk of starvation of longer processes.

Example



Process Queue	Burst Time	Arrival Time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

HIGHEST RESPONSE RATIO NEXT

Consider the following ratio:

$$R = \frac{w + s}{s}$$

where

R response ratio

w time spent waiting for the processor

s expected service time

Note that the minimum value of R is 1.0, which occurs when a process first enters the system.

Thus, our scheduling rule becomes the following: when the current process completes or is blocked, choose the ready process with the greatest value of R .

This approach is attractive because it accounts for the age of the process. While shorter jobs are favored (a smaller denominator yields a larger ratio), waiting without service increases the ratio so that a longer process will eventually get past competing shorter jobs.

As with SRT and SPN, the expected service time must be estimated to use highest response ratio next (HRRN).

FEEDBACK

If we have no indication of the relative length of various processes, then none of SPN, SRT, and HRRN can be used. Another way of establishing a preference for shorter jobs is to penalize jobs that have been running longer. In other words, **if we cannot focus on the time remaining to execute, let us focus on the time spent in execution so far.**

The way to do this is as follows. Scheduling is done on a **preemptive** (at time quantum) basis, and a dynamic priority mechanism is used. When a process first enters the system, it is placed in RQ0. After its first preemption, when it returns to the Ready state, it is placed in RQ1. Each subsequent time that it is preempted, it is demoted to the next lower-priority queue. A short process will complete quickly, without migrating very far down the hierarchy of ready queues. A longer process will gradually drift downward. Thus, newer, shorter processes are favored over older, longer processes. Within each queue, except the lowest-priority
queue, a simple FCFS mechanism is used.

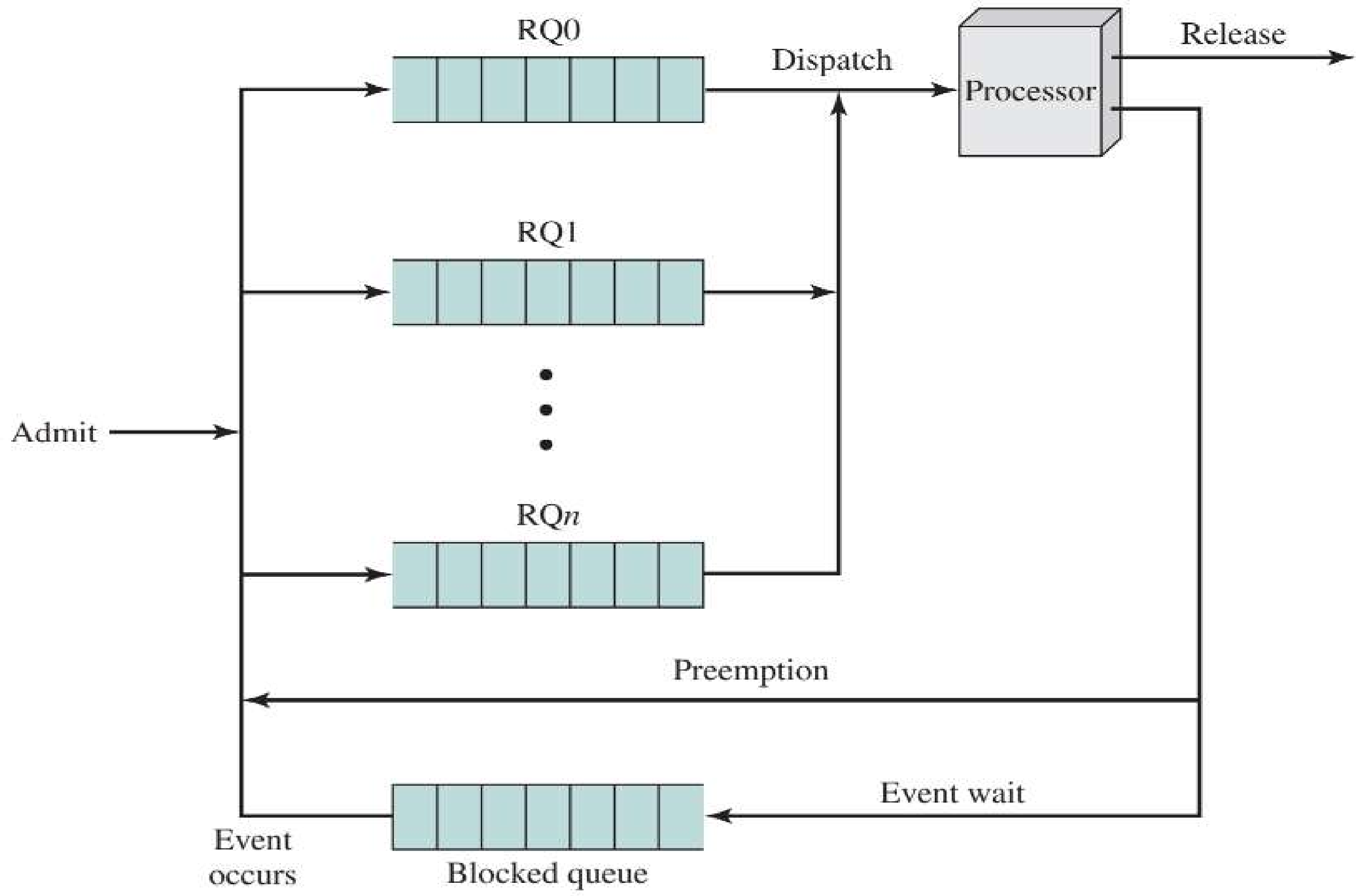


Figure 9.4 Priority Queueing

Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution. Thus, this queue is treated in round-robin fashion.

This approach is known as **multilevel feedback** , meaning that the OS allocates the processor to a process and, when the process blocks or is preempted, feeds it back into one of several priority queues.

Performance Comparison

Clearly, the performance of various scheduling policies is a critical factor in the choice of a scheduling policy. However, it is impossible to make definitive comparisons because relative performance will depend on a variety of factors, **including the probability distribution of service times of the various processes, the efficiency of the scheduling and context switching mechanisms, and the nature of the I/O demand and the performance of the I/O subsystem.**

MULTIPROCESSOR SCHEDULING

When a computer system contains more than a single processor, several new issues are introduced into the design of the scheduling function. We begin with a brief overview of multiprocessors and then look at the rather different considerations when **scheduling** is done at the **process level and at the thread level.**

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 multiprocessor:** Consists of a set of processors that share a common main memory and are under the integrated control of an operating system.

Design Issues

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** In looking at these three issues, 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.

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.

ASSIGNMENT OF PROCESSES TO PROCESSORS

If we assume 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**.

Static assignment

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.

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. All processes go into one global queue and are scheduled to any available processor. Thus, over the life of a process, the process may be executed on different processors at different times. This is called as **dynamic assignment.**

In a tightly coupled shared-memory architecture, the context information for all processes will be available to all processors, and therefore the cost of scheduling a process will be independent of the identity of the processor on which it is scheduled.
Yet another option is **dynamic load balancing,** in which threads are moved for a queue for one processor to a queue for another processor; Linux uses this approach.

Regardless of whether processes are dedicated to processors, some means is needed to assign processes to processors.

Two approaches have been used: master/slave and peer.

With a master/slave architecture, **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. This approach is quite 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.

There are **two disadvantages** to this approach:
(1) A failure of the master brings down the whole system, and
(2) the master can become a performance bottleneck.

In a peer architecture, **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.

THE USE OF MULTIPROGRAMMING ON INDIVIDUAL PROCESSORS

When each process is statically assigned to a processor for the duration of its lifetime, a new question arises: Should that processor be multiprogrammed?

The reader’s first reaction may be to wonder why the question needs to be asked; it would appear particularly wasteful to tie up a processor with a single process when that process may frequently be blocked waiting for I/O or because of concurrency/synchronization considerations.

In the traditional multiprocessor, which is dealing 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 and therefore better performance.
However, for medium-grained applications running on a multiprocessor with many processors, the situation is less clear. When many processors are 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, on average, for the applications.
An application that consists of a number of threads may run poorly unless all of its threads are available to run simultaneously.

PROCESS DISPATCHING The final design issue related to multiprocessor scheduling is the actual selection of a process to run.
We have seen that, 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.
In the case of thread scheduling, new issues come into play that may be more important than priorities or execution histories.

Thread Scheduling

On a **uniprocessor, threads can be used as a program structuring aid and to overlap I/O with processing. Because of the minimal penalty in doing a thread switch compared to a process switch, these benefits are realized with little cost.**

However, the full power of threads becomes evident in a multiprocessor system. In this environment, 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. However, it can be shown that for applications that require significant interaction among threads (medium-grain parallelism), small differences in thread management and scheduling can have a significant performance impact

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

1. **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.

LOAD SHARING Load sharing is perhaps the simplest approach and the one that carries over most directly from a uniprocessor environment. It has several **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, including priority-based schemes.

There are several **disadvantages** of load sharing:

• 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. However, 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.

disadvantages cont…

• 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.

Despite the potential disadvantages, load sharing is one of the most commonly used schemes in current multiprocessors.

2. Gang scheduling: A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.
The concept of scheduling a set of processes simultaneously on a set of processors refers to the concept as group scheduling or gang scheduling

benefits:

• 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.

3. **Dedicated processor assignment:** This is the opposite of the load-sharing approach and provides implicit scheduling defined by the assignment of threads to processors. Each program, for the duration of its execution, is allocated a number of processors equal to the number of threads in the program.When the program terminates, the processors return to the general pool for possible allocation to another program.

4. **Dynamic scheduling:** The number of threads in a process can be altered during the course of execution.

REAL-TIME SCHEDULING

Real-time computing is becoming an increasingly important discipline. The operating system, and in particular the scheduler, is perhaps the most important component of a real-time system. **Examples of current applications of real-time systems** include control of laboratory experiments, process control in industrial plants, robotics, air traffic control, telecommunications, and military command and control systems. Next-generation systems will include the autonomous land rover, controllers of robots with elastic joints, systems found in intelligent manufacturing, the space station, and undersea exploration.

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 **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.

Characteristics of Real-Time Operating Systems

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

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

An operating system is **deterministic** to the extent that it performs operations 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. In a real-time operating system, process requests for service are dictated by external events and timings. The extent to which an operating system can deterministically satisfy requests depends first on the speed with which it can respond to interrupts and, second, on whether the system has sufficient capacity to handle all requests within the required time.

A related but distinct characteristic is **responsiveness**. Responsiveness is concerned with how long, after acknowledgment, it takes an operating system to service the interrupt.

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.

Reliability: 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.

To meet the foregoing requirements, real-time operating systems typically include the following **features** :

- Fast process or thread switch
- Small size (with its associated minimal functionality)
- Ability to respond to external interrupts quickly
- Multitasking with interprocess communication tools such as semaphores, signals, and events
- 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
- Primitives to delay tasks for a fixed amount of time and to pause/resume tasks
- Special alarms and timeouts

The **heart** of a real-time system is the **short-term task scheduler**. In designing such a scheduler, **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.**

Most contemporary real-time operating systems are unable to deal directly with deadlines. Instead, they are designed to be as responsive as possible to real-time tasks so that, when a deadline approaches, a task can be quickly scheduled. From this point of view, real-time applications typically require deterministic response

times in the several-millisecond to submillisecond span under a broad set of conditions; leading-edge applications—in simulators for military aircraft, for example often have constraints in the range of 10–100 µs

The **heart** of a real-time system is the **short-term task scheduler**. In designing such a scheduler, **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.**

Most contemporary real-time operating systems are unable to deal directly with deadlines. Instead, they are designed to be as responsive as possible to real-time tasks so that, when a deadline approaches, a task can be quickly scheduled. From this point of view, real-time applications typically require deterministic response

times in the several-millisecond to submillisecond span under a broad set of conditions; leading-edge applications—in simulators for military aircraft, for example often have constraints in the range of 10–100 µs

Real-Time Scheduling

Real-time scheduling is one of the most active areas of research in computer science.

Deadline Scheduling

Most contemporary real-time operating systems are designed with the **objective of starting real-time tasks as rapidly as possible, and hence emphasize rapid interrupt handling and task dispatching.** In fact, this is not a particularly useful metric in evaluating real-time operating systems.

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.** It follows that priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time.

There have been a number of proposals for more powerful and appropriate approaches to real-time task scheduling. All of these are based on **having additional information about each task.** In its most general form, the following information about each task might be used:

• **Ready time:** Time at which task becomes ready for execution. In the case of a repetitive or periodic task, this is actually a sequence of times that is known in advance. In the case of an aperiodic task, this time may be known in advance, or the operating system may only be aware when the task is actually ready.

- **Starting deadline:** Time by which a task must begin.

- **Completion deadline:** Time by which a task must be completed. The typical real-time application will either have starting deadlines or completion deadlines, **but not both**.

- **Processing time:** Time required to execute the task to completion. In some cases, this is supplied. In others, the operating system measures an exponential average. For still other scheduling systems, this information is not used.

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

- **Priority:** Measures 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.

There are several dimensions to the real-time scheduling function when dead-lines are taken into account:
which task to schedule next, and what sort of preemption is allowed. It can be shown, for a given preemption strategy and using either starting or completion deadlines, **that a policy of scheduling the task with the earliest deadline minimizes the fraction of tasks that miss their deadlines .** This conclusion holds for both single-processor and multiprocessor configurations.

The other critical design issue is that of **preemption**.

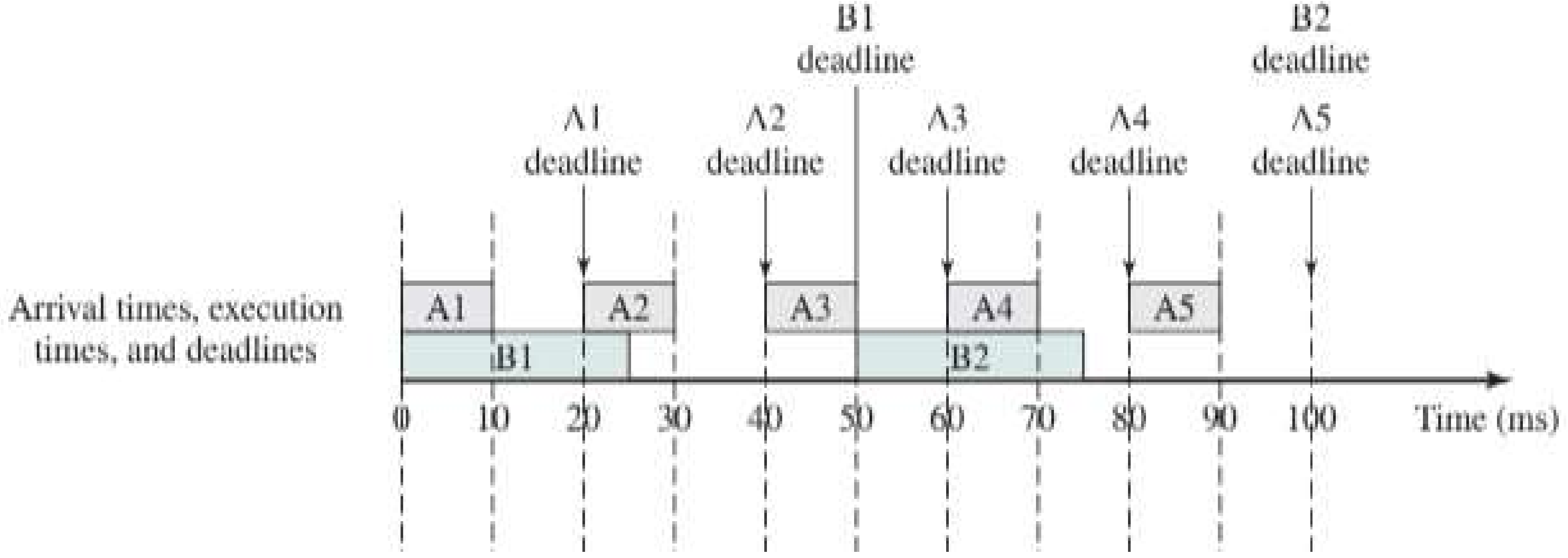
When starting deadlines are specified, then a non-preemptive scheduler makes sense. In this case, it would be the responsibility of the real-time task to block itself after completing the mandatory or critical portion of its execution, allowing other real-time starting deadlines to be satisfied.

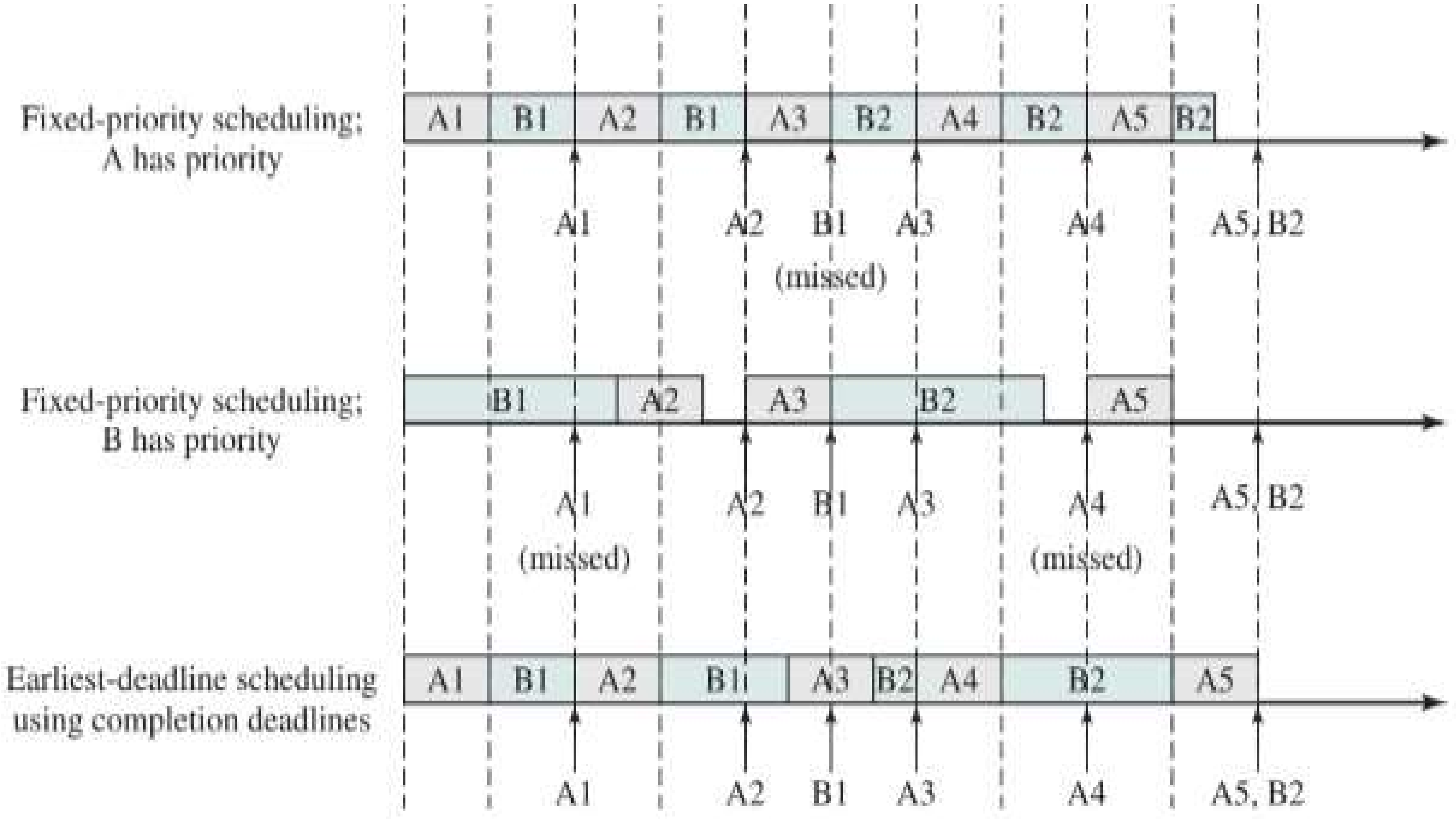
For a system with completion deadlines, a preemptive strategy is most appropriate. For example, if task X is running and task Y is ready, there may be circumstances in which the only way to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, and then resume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

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 summarizes the execution profile of the two tasks, Figure compares three scheduling techniques using the execution profile of Table .
The first row of Figure 10.5 repeats the information in Table ; the remaining three rows illustrate three scheduling techniques.

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
•	•	•	•
•	•	•	•
•	•	•	•


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

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 shows the result. 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. If B is given higher priority, then A will miss its first deadline.

The final timing diagram shows the use of earliest-deadline scheduling. At time t = 0 , both A1 and B1 arrive. Because A1 has the

Solve the following (4 marks each)

Q.1 What is memory management? Explain requirements of memory management.
Q.2 Explain Memory partitioning with its two basic techniques.
Q.3 Write a short note on the following.
Paging
Segmentation
Security Issues in Memory
Virtual Memory Technique

Real-Time Scheduling

In a survey of real-time scheduling algorithms,various scheduling approaches depend on

(1) whether a system performs schedulability analysis,

(2) if it does, whether it is done statically or dynamically, and

(3) whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run time.

Based on these considerations, following were the classes of algorithms:

- **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. Rather, 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.**