COP 4600 – Summer 2013

Introduction To Operating Systems

Real-time Processor Scheduling

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4600/sum2013

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Real-Time Systems

- Real-time computing is becoming an increasingly important discipline.
- The OS, and in particular the scheduler, is perhaps the most important component of a real-time system.
- 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.

Examples of Real-Time Systems

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

Real-Time Systems

- To define a real-time system, we need to define what is meant by a real-time process or task.
- In general, in a real-time system, some of the tasks are real-time tasks, and these have a certain degree of urgency associated with them.
- Such tasks are attempting to control or react to events that take place in the outside world
- Because these events occur in “real time”, a real-time task must be able to keep up with the events with which it is concerned.

Real-Time Systems

- Usually, a deadline is associated with a particular task.
- Deadlines may represent either a start time or a completion time.
- Real-time tasks are typically classified as either hard or soft.
- 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 its deadline has passed.

Real-Time Systems

- Another characteristic of real-time tasks is whether they are periodic or aperiodic.
- An aperiodic task has a deadline by which it must start or finish, or it may have a constraint on **both** the start and finish time.
- For a periodic task, the deadline(s) may be stated a fashion such as *"once per period T"* or *"exactly T units of time apart."*

Characteristics of Real-Time Operating Systems

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

1. Determinism
2. Responsiveness
3. User control
4. Reliability
5. Fail-soft operation

Characteristics of Real-Time Operating Systems

Determinism

- An OS 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 OS, process requests for service are dictated by external events and times.
- The extent to which an OS 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.

Characteristics of Real-Time Operating Systems

Determinism (cont.)

- One useful measure of the ability of an OS to function deterministically is the maximum delay from the arrival of a high priority device interrupt to when servicing the interrupt begins.
- In non-real-time OS, this delay may be in the range of tens to hundreds of milliseconds, while in a real-time OS this delay may have an upper bound of anywhere from a few microseconds to a millisecond.

Characteristics of Real-Time Operating Systems

Responsiveness

- Determinism is concerned with how long an OS delays before acknowledging an interrupt. Responsiveness is concerned with how long, after the acknowledgement, it takes an OS to service the interrupt.
- Aspects of responsiveness include:
  - The amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR). If execution of the ISR requires a process switch, then the delay will be longer than if the ISR can be executed within the context of the current process.
  - The amount of time to perform the ISR. Generally, this is dependent on the hardware platform.
  - The effect of interrupt nesting. If an ISR can be interrupted by the arrival of another interrupt, then the service will be delayed.

Characteristics of Real-Time Operating Systems

- Determinism and responsiveness together make up the response time to external events.
- Response time requirements are critical for real-time systems, because such systems must meet timing requirements imposed by individuals, devices, and data flows external to the system.

Characteristics of Real-Time Operating Systems

User control

- User control is generally much broader in real-time OS than in ordinary OS.
- In a typical non-real-time OS, the user either has no control over the scheduling function of the OS or can only provide broad guidance, such as grouping users into more than one priority class.
- In a real-time system, however, it is essential to allow the user fine-grained control over task priority.
- In a real-time system the user can:
  - Specify priority
  - Distinguish between hard and soft tasks
  - Specify paging and/or process swapping
  - Specify which processes must always reside in main memory
  - Specify which disk algorithms to use
  - Specify the rights of processes in the various priority groups

Characteristics of Real-Time Operating Systems

Reliability

- Reliability is typically far more important for real-time systems than non-real-time systems.
- A transient failure in a non-real-time system may be solved by simply rebooting the system. A processor failure in a multiprocessor non-real-time system may result in a reduced level of service until the failed processor is repaired or replaced.
- 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.

Characteristics of Real-Time Operating Systems

Fail-soft operation

- 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
- An important aspect of fail-soft operation is referred to as stability.
- A real-time system is stable if, in cases where it is impossible to meet all task deadlines, the system will meet the deadlines of its most critical, highest-priority tasks, even if some less critical task deadlines are not always met.

Characteristics of Real-Time Operating Systems

Fail-soft operation (cont.)

- To meet these requirements, real-time OS typically include the following features:
  - Fast process or thread switch
  - Small size (with corresponding minimal functionality)
  - Ability to respond to external interrupts quickly
  - Multitasking with interprocess communication tools such as semaphores, signals, and events
  - Use of 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 time-outs

Real-Time Operating Systems

- The heart of a real-time system is the short-term task scheduler.
- In designing such a scheduler, fairness and minimizing average response time are not of supreme importance.
- What is important is that all hard real-time tasks meet their deadlines and that as many soft real-time tasks as possible also meet their deadlines.
- Most contemporary real-time OS 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 range under a broad set of conditions. Leading edge applications, such as in simulators for military aircraft, often have constraints in the range of 10 -100 μs.

Real-Time Operating Systems

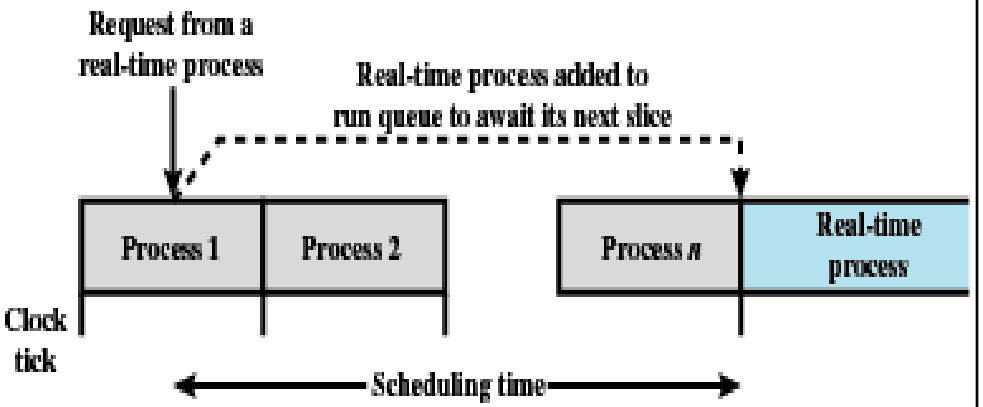
- The figure on page 19 illustrates the spectrum of scheduling protocol possibilities described below.
- Figure (a) represents a simple round-robin protocol, where a real-time task would be added to the ready queue to await its next time slice. The scheduling time is generally unacceptable for real-time applications.
- Figure (b) represents a priority-driven nonpreemptive scheduler in which real-time tasks are given high priority. In this case, a real-time task that is ready would be scheduled when the current process blocks or runs to completion. This could lead to a delay of several seconds if a slow, low-priority task were executing at a critical time. Again, this would be unacceptable for real-time applications.

Real-Time Operating Systems

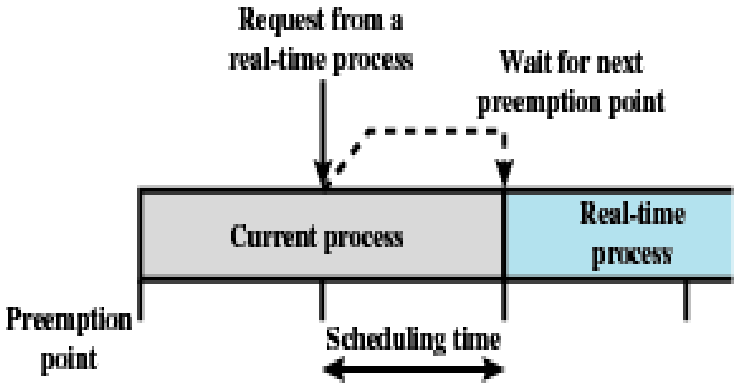
- Figure (c) represents a more promising approach that combines priorities with clock-based interrupts. In this case, preemption occurs at regular intervals. When a preemption point occurs, the currently running task is preempted if a higher-priority task is waiting. This would include the preemption of tasks that are part of the OS kernel. Such a delay may be on the order of several milliseconds.

- While the approach illustrated in Figure (c) may be adequate for some real-time applications, it will not suffice for more demanding applications. In those cases, an approach that has been successfully applied is referred to as immediate preemption. This technique is illustrated in Figure (d). In this case, the OS responds to an interrupt almost immediately, unless the system is in a critical-code lockout section. In this fashion scheduling delays can be reduced to 100 μs or less.

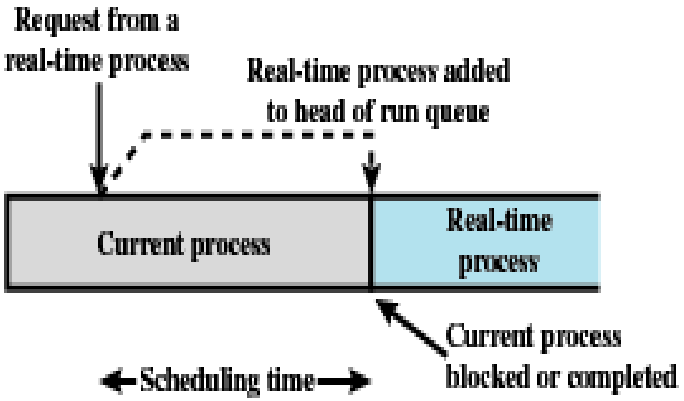
Scheduling of a Real-Time Process



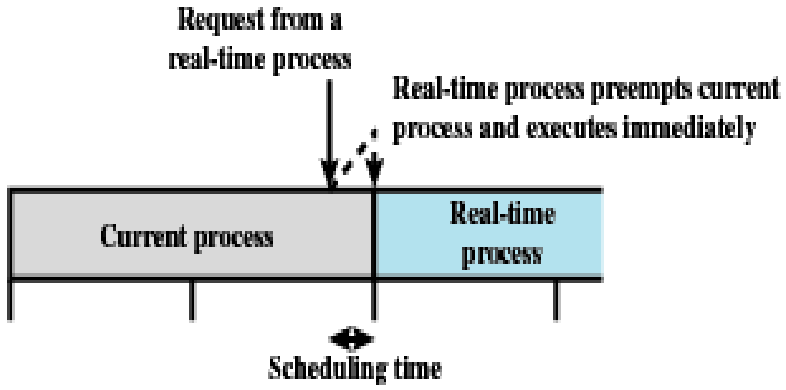
(a) Round-robin Preemptive Scheduler



(c) Priority-Driven Preemptive Scheduler on Preemption Points



(b) Priority-Driven Nonpreemptive Scheduler



(d) Immediate Preemptive Scheduler

Real-Time Scheduling

- Real-time scheduling approaches depend on:

  1. Whether the system performs schedulability analysis.
  2. If schedulability analysis is performed is it done statically or dynamically.
  3. Do the results of the schedulability analysis itself produce a schedule that can be used to dispatch tasks a run-time.

- Based on these considerations, the following four classes of algorithms have been developed:

Real-Time Scheduling

1. Static table-driven approaches
   - These techniques 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.
2. Static priority-driven preemptive approaches
   - Again, static analysis is performed, but no schedule is produced. Rather, the analysis is used to assign priorities to tasks, so that a traditional priority-driven scheduler can be used.
3. 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.
4. 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.

1. Static Table-Driven Scheduling

- Static table-driven scheduling is applicable to tasks that are periodic.
- Input to the analysis consists of the periodic arrival time, execution time, periodic ending deadline, and relative priority of each task.
- The scheduler attempts to develop a schedule that enables it to meet the requirements of all periodic tasks.
- This is a predictable approach but one that is inflexible, because any change to any task requirements requires that the schedule be redone.
- Earliest-Deadline-First (EDF) or other periodic deadline techniques (we'll see them shortly) are typical of this category of scheduling algorithms.

2. Static Priority-Driven Preemptive Scheduling

- Static priority-driven scheduling makes use of the priority-driven preemptive scheduling mechanism which is common to most non-real-time multiprogrammed systems.
- In a non-real-time system, a variety of factors might be used to determine priority. For example, in a time-sharing system, the priority of a process changes depending on whether it is CPU-bound or I/O bound.
- In a real-time system, priority assignment is related to the time constraints associated with each task.
- One example of this approach is the rate monotonic algorithm (we'll see this shortly), which assigns static priorities to tasks based on the length of their periods.

3. Dynamic Planning-Based Scheduling

- With dynamic planning-based scheduling, after a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously scheduled tasks as well as the new arrival.
- If the new arrival can be scheduled in such a way that its deadlines are satisfied and that no currently scheduled task misses a deadline, then the schedule is revised to accommodate the new task.
- If the new arrival cannot be scheduled in any manner that will either satisfy its deadlines or will cause a currently scheduled task to miss a deadline, then the new task is rejected and not admitted to the system.

4. Dynamic Best Effort Scheduling

- Dynamic best effort scheduling is the approach used by many real-time systems that are currently commercially available.
- When a task arrives, the system assigns a priority based on the characteristics of the task.
- Some form of deadline scheduling, such as EDF, is typically used.
- Typically, the tasks are aperiodic and so no static scheduling analysis is possible.
- With this type of scheduling, until a deadline arrives or until the task completes, we do not know whether a timing constraint will be met. This is a major disadvantage of this form of scheduling. Its advantage is that it is easy to implement.

Deadline Scheduling

- Most contemporary real-time OS designed with the objective of starting real-time tasks as rapidly as possible, and hence emphasize rapid interrupt handling and task dispatching.
- Unfortunately, being able to start a real-time task rapidly is not a particularly useful metric in evaluating real-time OS.
- Real-time applications are not generally concerned with sheer speed but rather with completing (or starting) tasks at the most valuable time, neither too early nor too late, despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults.
- Priorities provide a crude tool, but do not capture the requirement of completion (or initiation) at the most valuable time.

Deadline Scheduling (cont.)

- There have been a number of proposals for more powerful and appropriate approaches to real-time task scheduling. Virtually 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: The time at which a 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 OS may only be aware when the task is actually ready.
  - Starting deadline: The time by which a task must begin.
  - Completion deadline: The time by which a task must be completed. The typical real-time application will either have starting deadlines or completion deadline, but not both.

Deadline Scheduling (cont.)

- Processing time: The time required to execute the task to completion. In some cases, this is supplied. In others, the OS measures an exponential average. Some systems do not utilize this information at all.
- Resource requirements: The set of resources (other than the processor) required by the task while it is executing.
- Priority: This measures the 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.

Deadline Scheduling (cont.)

- There are several dimensions to the real-time scheduling function when deadlines 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 holds true for both uniprocessor and multiprocessor environments.

Deadline Scheduling (cont.)

- The other critical design issue is that of preemption.
  - When only starting deadlines are specified, then a nonpreemptive 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 section of its execution, allowing other real-time starting deadlines to be satisfied. This would fit the pattern of Figure (b) on page 19.
  - When completion deadlines are used, a preemptive strategy is the most appropriate. This situation is modeled by Figures (c) and (d) on page 19. 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.

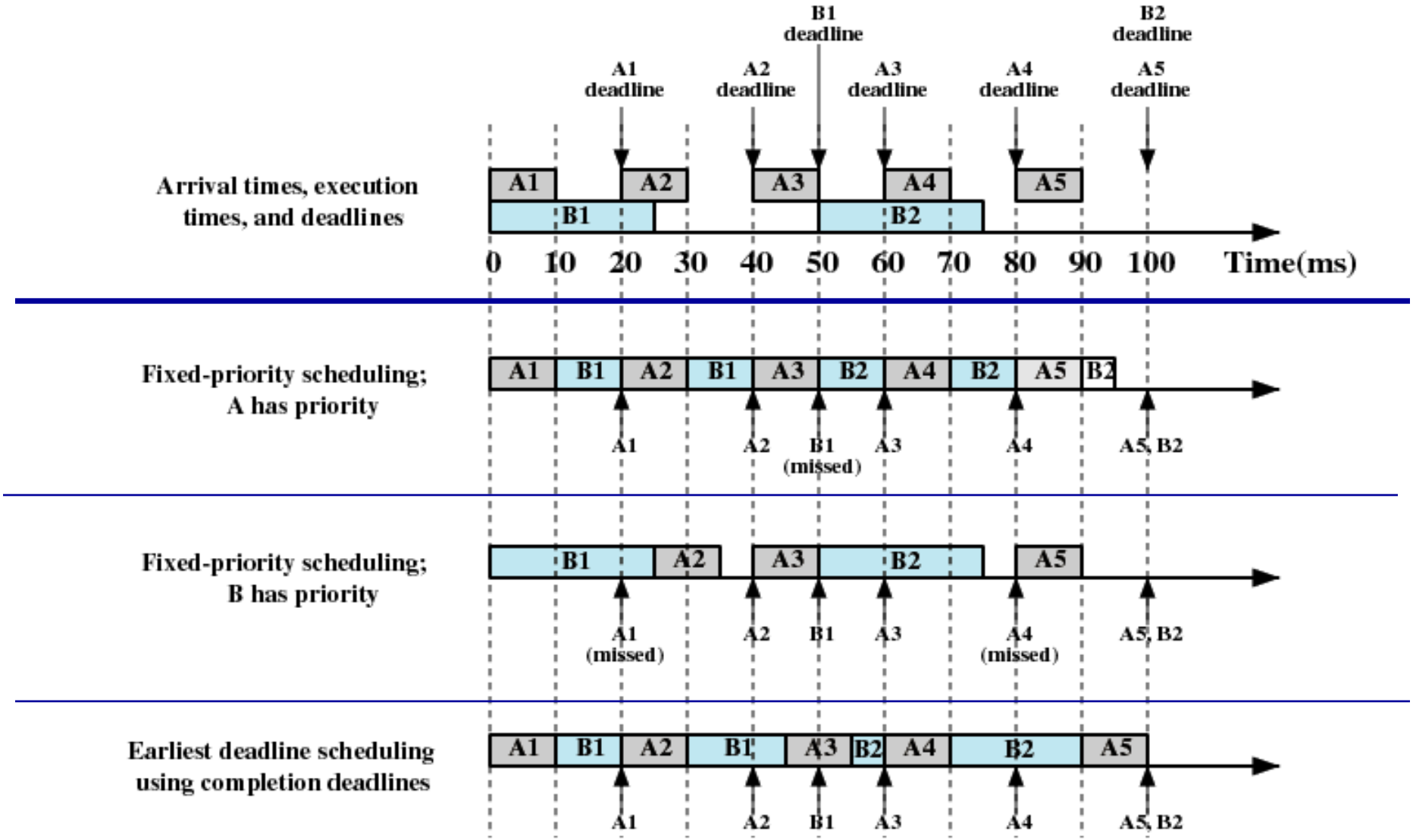
Deadline Scheduling (cont.)

- 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 msec, and that for B every 50 msec. It takes 10 msec, including OS overhead, to process each sample of data from A and 25 msec to process each sample of data from B. Further suppose that the computer is capable of making a scheduling decision every 10 msec.
- The table on page 32 summarizes the execution profile of the two tasks.
- The figure on page 33 compares three scheduling techniques using the execution profile from page 32. The first two schedules use a priority based scheme. The final schedule uses earliest deadline scheduling.

Example Scheduling Two Periodic Tasks

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 of Periodic Real-time Tasks with Completion Deadlines

Explanation of Example of Two Periodic Tasks

- As shown in the first schedule, if A has higher priority, the first task of B is only given 20 msec of processing time, in two 10 msec chunks, by the time its deadline arrives, and thus task B fails.
- The second schedule assumes that B has the higher priority, then A will miss its first deadline as it has had no time allocation at all before the deadline arrives.
- The final schedule uses the earliest deadline scheduling scheme. 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 time 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 resumes at time t=30. At time t=40, A3 arrives. However, B1 has an earlier deadline than A3 and is allowed to complete execution at time t=45. A3 is then given the processor and finishes at time t=55. Thus, all deadlines are met.
- This example works because the tasks are periodic and predictable allowing a static table-driven scheduling approach to be developed.

Example of Two Aperiodic Tasks with Starting Deadlines

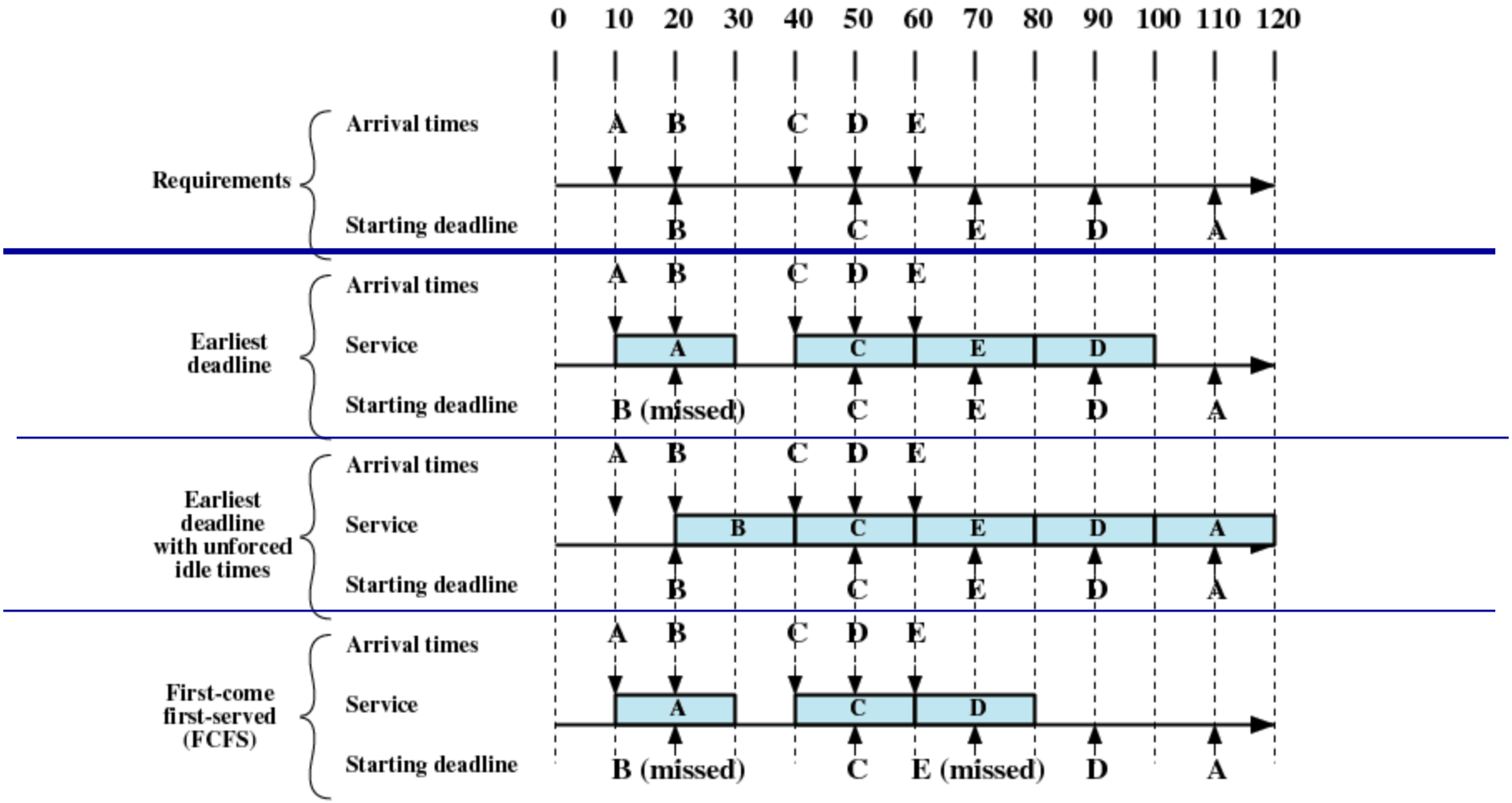
- Now let's consider a scheduling scheme for dealing with aperiodic tasks with starting deadlines.
- The table on page 37 illustrates the arrival times and starting deadlines for a set of five tasks each of which has an execution time of 20 msec.
- The diagram on page 38 illustrates three different scheduling schemes for these periodic tasks.
- A straight forward way to schedule such tasks is to always schedule the ready task with the earliest deadline and let that task run to completion.
- This is the first approach illustrated in the diagram on page 38. In this example, note that although task B requires immediate service, the service is denied.

Example of Two Aperiodic Tasks with Starting Deadlines

- The risk in dealing with aperiodic tasks, especially with starting deadlines, is that a starting deadline can be missed when a task arrives and the CPU is already allocated to an earlier arriving task.
- A refinement of this technique will improve performance (achieve a higher number of non-failures among real-time tasks) if deadlines can be known in advance of the time that a task is ready. This policy is known as earliest deadline with unforced idle times.
- This technique operates as follows: always schedule the eligible task with the earliest deadline and let that task run to completion. An eligible task may not be ready, and this may result in the processor remaining idle even though there are ready tasks.
- This technique is illustrated as the second case on page 38. Notice in the example that task A is not scheduled even though it is the only ready task. Even though processor utilization is not maximum, all deadlines are met.
- FCFS is shown for comparison purposes only, note two deadlines are missed.

Example of Two Aperiodic Tasks with Starting Deadlines

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 of Aperiodic Real-time Tasks with Starting Deadlines

Rate Monotonic Scheduling

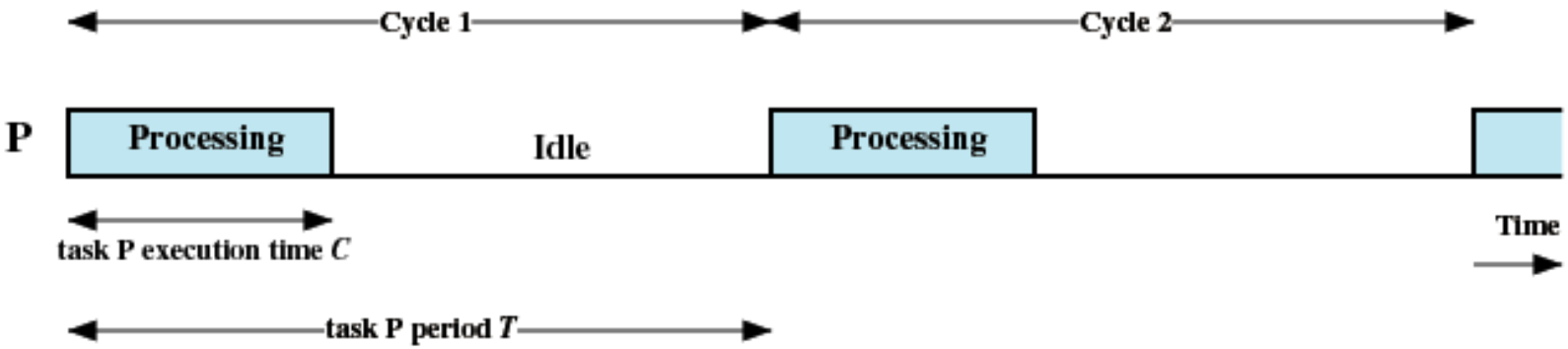
- One of the more promising methods of resolving multitask scheduling conflicts for periodic real-time tasks is rate monotonic scheduling (RMS).
- RMS assigns priorities to tasks on the basis of their periods.
- The diagram on page 41 illustrates the relevant parameters for periodic tasks.
- The task's 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. The task's rate (in Hertz) is simply the inverse of its period (in seconds).
  - For example, a task with a period of 50 msec occurs at a rate of 20 Hz.

$$\frac{1}{50\times10^{-3}} = \frac{1}{0.05} = 20$$

Rate Monotonic Scheduling (cont.)

- Typically, the end of a task's period is also the task's hard deadline, although some tasks may have earlier deadlines.
- The execution time C, is the amount of processing time required for each occurrence of the task.
  - In a uniprocessor system, this implies that the execution time must be no greater than the period, i.e., $C \leq T$.
- If a periodic task is always to 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 msec and an execution time of 55 msec, its processor utilization is 55/80 = 0.6875 = 68.75%.

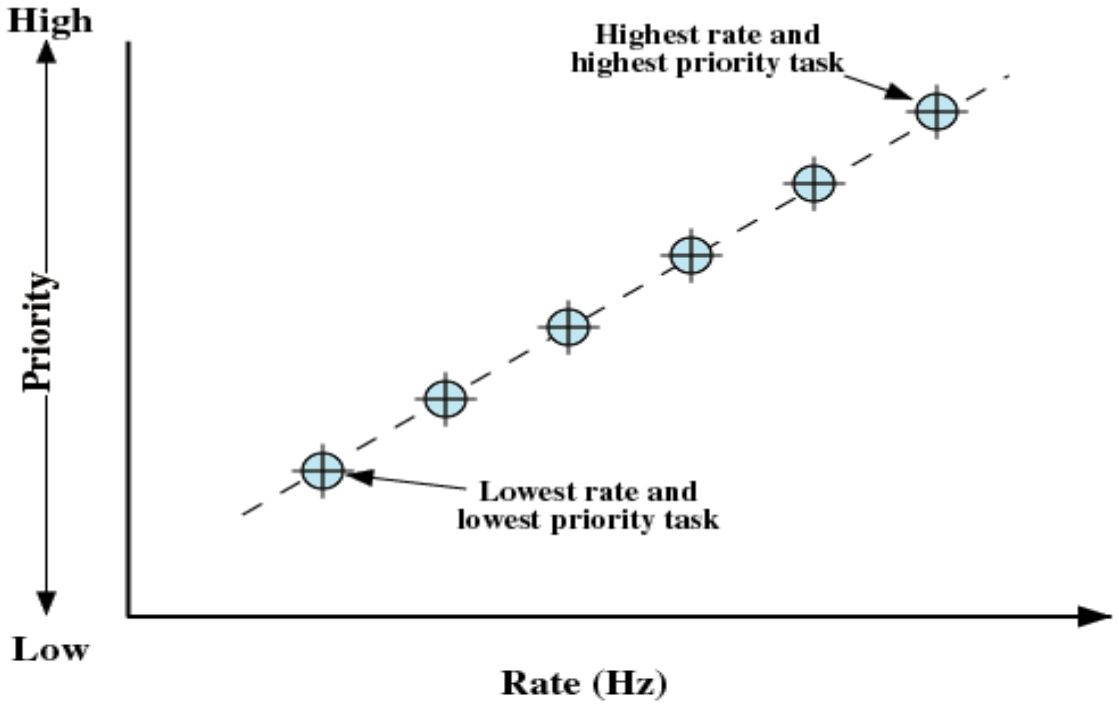
Periodic Task Timing Diagram



Periodic Task Timing Diagram

Rate Monotonic Scheduling (cont.)

- 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 periods, and so on.
- When more than one task is available for execution, the one with the shortest period is serviced first.
- Plotting the priority of tasks as a function of their rate, the result is a monotonically increasing function; hence the name rate monotonic scheduling. This is illustrated on the next page.



A Task Set with RMS

Evaluation of Periodic Scheduling Algorithms

- One measure of the effectiveness of a periodic scheduling protocol is whether or not it guarantees that all hard deadlines are met.
- Suppose that we have *n* tasks, each with a fixed period and execution time.
- 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 the total utilization of the processor.

Evaluation of Periodic Scheduling Algorithms

- The equation on the previous page 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.
- It has been shown for RMS that the following inequality holds:

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

- Using this inequality, we can determine the upper bounds for RMS. This is illustrated in the table on the following page.

Evaluation of Periodic Scheduling Algorithms

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 ≈ 0.693

Value of the RMS upper bound

Example: Consider three periodic tasks.

Task P_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$

Task P_2: $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$

Task P_3: $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$

Total utilization = 0.2 + 0.267 + 0.286 = 0.753

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 3\left(2^{1/3} - 1\right) = 0.779$$

Since the total utilization required for the three tasks is less than the upper bound for RMS (0.753 < 0.779), we know that if RMS is used, all tasks will be successfully scheduled.

Evaluation of Periodic Scheduling Algorithms

- Interestingly, it can also be shown that the upper bound of the equation on page 44 also holds for earliest deadline scheduling.
- Thus, it is possible to achieve greater overall processor utilization and therefore accommodate more periodic tasks with earliest deadline scheduling than with RMS.
- Nevertheless, RMS has been widely adopted for use in industrial applications. Some of the reasons for this are:

1. The performance difference is small in practice. The upper bound given by the equation on page 45 is a conservative one and, in practice, utilization as high as 90% is often achieved.
2. Most hard real-time systems also have soft real-time components that can execute at lower priority levels to absorb the processor time that is not utilized with RMS scheduling of hard real-time tasks.
3. Stability is easier to achieve with RMS. When a system cannot meet all deadlines because of overload or transient errors, the deadlines of essential tasks need to be guaranteed provided that this subset of tasks is schedulable. In a static priority assignment approach, one only needs to ensure that essential tasks have relatively high priorities. This can be done in RMS by structuring essential tasks to have short periods or by modifying the RMS priorities to account for essential tasks. With earliest deadline scheduling, a periodic task's priority changes from one period to another. This makes it more difficult to ensure that essential tasks meet their deadlines.

Priority Inversion

- Priority inversion is a phenomenon that can occur in any priority-based preemptive scheduling scheme but is particularly relevant in the context of real-time scheduling.
- The best known instance of priority inversion involved the Mars Pathfinder mission.
  - The rover robot landed on Mars on July 4, 1997 and began gathering and transmitting data back to Earth. A few days into the mission, the lander software began experiencing total system resets, each resulting in the loss of data. After much effort by the JPL team that built the Pathfinder, the problem was traced to priority inversion.
- In any priority scheduling scheme, the system should always be executing the task with the highest priority.
- Priority inversion occurs when circumstances within the system force a higher priority task to wait for a lower priority task.

Priority Inversion (cont.)

- A simple example of priority inversion occurs if a lower priority task has locked a resource (such as a device or synchronization construct) and a higher-priority task attempts to lock that same resource. The higher priority resource becomes blocked until the resource becomes available.
- If the lower-priority task finishes with the resource quickly and releases it, the higher-priority task may quickly resume and it might be possible that no real-time constraints are violated. However, the opposite may also be true, in which case the lower-priority task controls the resource for too long to allow the higher-priority task to meet its deadline.

Priority Inversion (cont.)

- An even more serious condition is referred to as an unbounded priority inversion, in which 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 as well.
- The priority inversion experienced in the Pathfinder software was unbounded and serves as a good example of the phenomenon.

Priority Inversion In The Mars Pathfinder

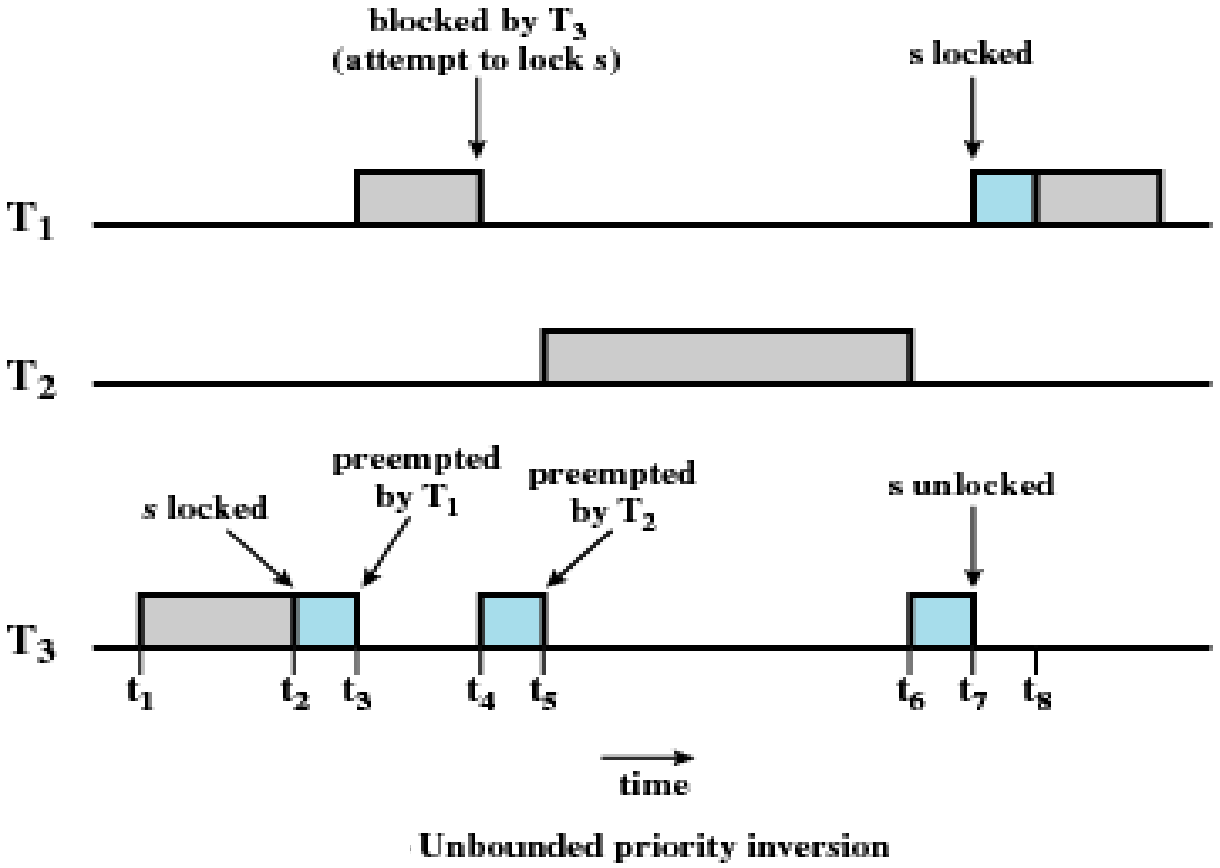
- The Pathfinder software included the following three tasks in decreasing order of priority:
  - T_1: Periodically check the health of the spacecraft systems and software.
  - T_2: Process image data.
  - T_3: Perform an occasional test on equipment status.
- After T_1 executes, it reinitializes a time 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.
- The 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*.
- The illustration on the next page shows the sequence of events that caused the priority inversion.

Priority Inversion In The Mars Pathfinder

- The set of events that caused the priority inversion is:
  - t_1: T_3 begins executing
  - t_2: T_3 locks semaphore *s* and enters its critical section.
  - t_3: T_1 which has 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 higher priority than T_3, preempts T_3 and begins execution.
  - t_6: T_2 is suspended for some reason unrelated to T_1 and T_2, and T_3 resumes.
  - t_7: T_3 leaves its critical section and unlocks the semaphore. T_1 preempts T_3, locks the semaphore, and enters its critical section.
- In this set of circumstances, T_1 must wait for both T_3 and T_2 to complete and fails to reset the timer before it expires.
- The illustration on the next page shows the sequence of events that caused the priority inversion.

Priority Inversion In The Mars Pathfinder

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



Unbounded priority inversion

Priority Inheritance

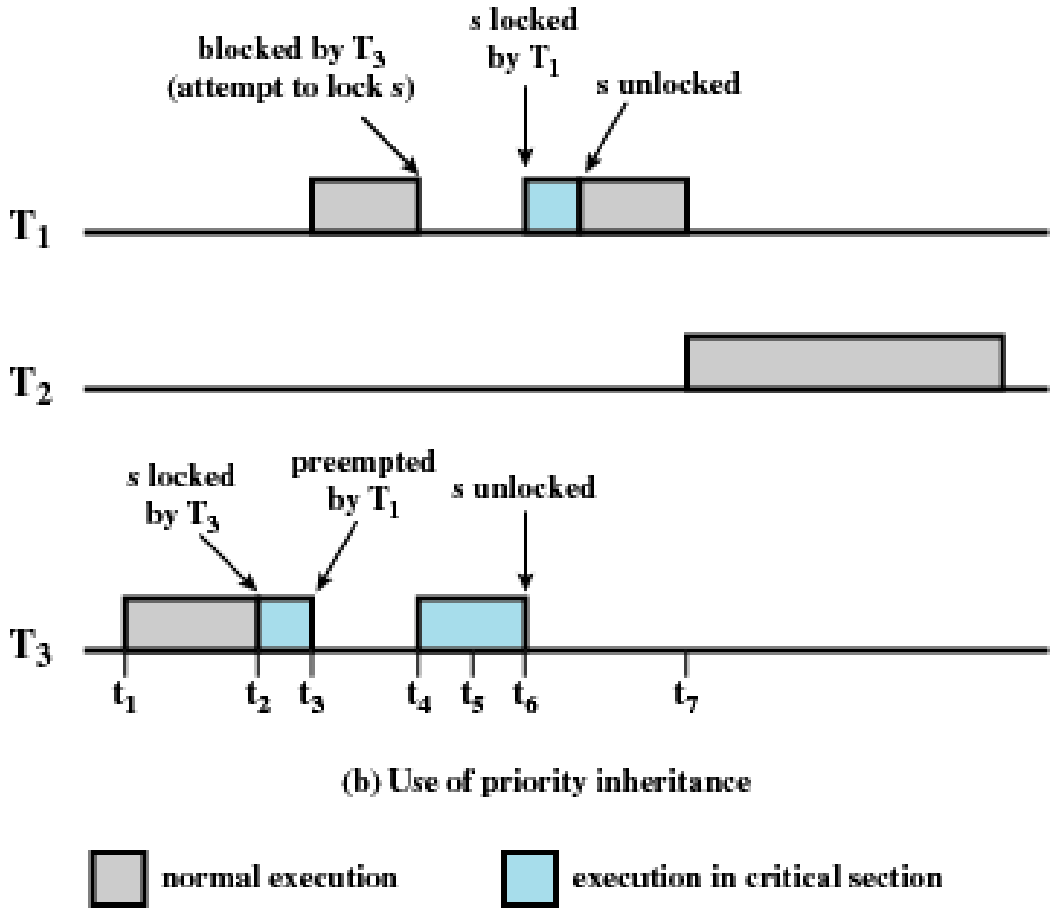
- In practical systems, two alternative approaches are used to avoid unbounded priority inversion: priority inheritance protocol and priority ceiling protocol.
- The basic idea of priority inheritance is that a lower-priority task inherits the priority of any higher-priority tasks pending on a resource 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.
- Using the Pathfinder example again, the following scenario illustrates how priority inheritance resolves the unbounded priority inversion that occurred.

Priority Inheritance Resolving Pathfinder Unbounded Priority Inversion

- The relevant set of events is:
  - t_1: T_3 begins executing
  - t_2: T_3 locks semaphore *s* and enters its critical section.
  - t_3: T_1 which has 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 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.
- The illustration on the next page shows this sequence of events.

Priority Inheritance

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



Priority Inversion

Priority Ceiling

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