OPERATING SYSTEMS

MULTIPROCESSOR, MULTICORE, REAL-TIME SCHEDULING

Classifications of Multiprocessor Systems

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

- There is **a master, general-purpose processor**;
- Specialized processors are controlled by the master processor and provide services to it

Tightly coupled multiprocessor

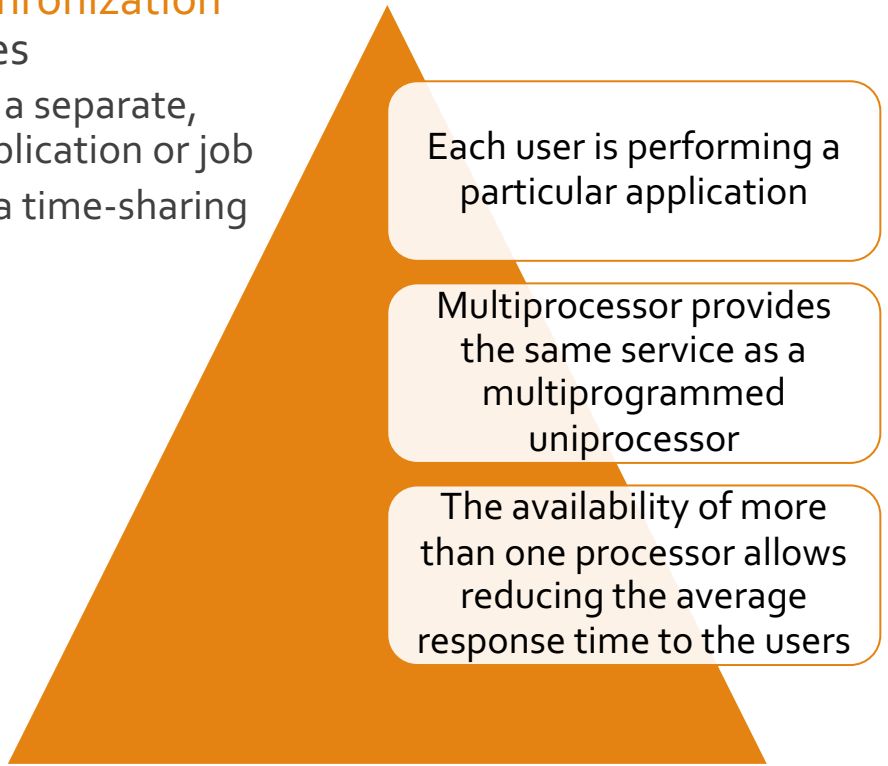
- Consists of **a set of processors that share a common main memory** and are under the integrated control of an operating system

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

Independent Parallelism

- No explicit synchronization among processes
  - Each represents a separate, independent application or job
  - Typical use is in a time-sharing system



Coarse and Very Coarse Grained Parallelism

- There is synchronization among processes, but at a very gross level
- Easily handled as a set of concurrent processes running on a multiprogrammed uniprocessor
- Can be supported on a multiprocessor with little or no change to user software

Medium-Grained Parallelism

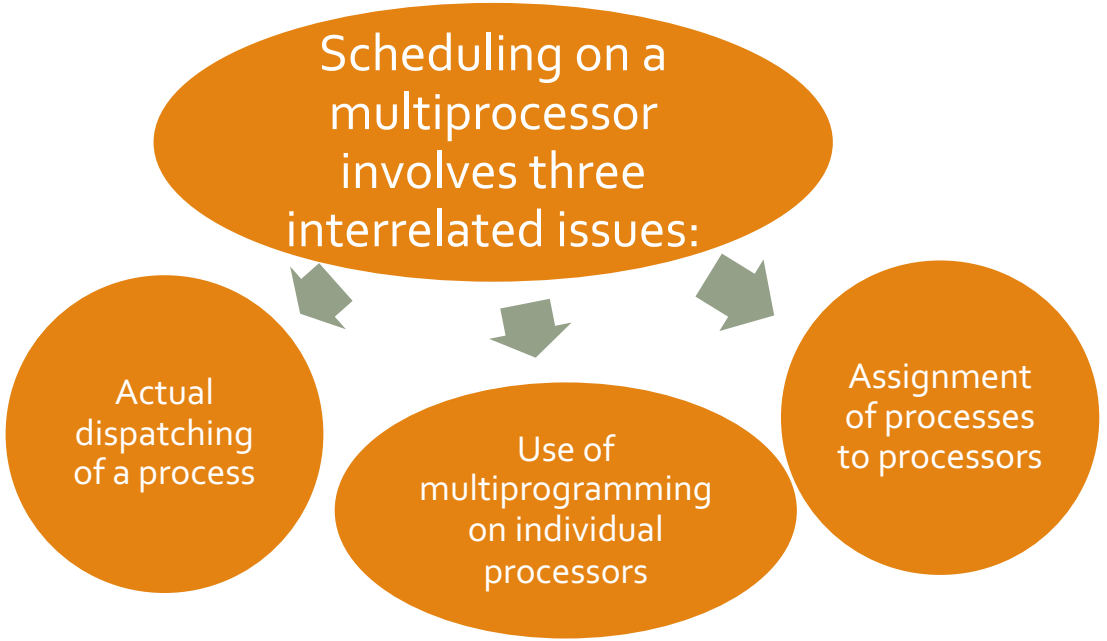
- Single application can be effectively implemented as a **collection of threads** within a single process
  - Programmer must explicitly specify the potential parallelism of an application
  - There needs to be a high degree of coordination and interaction among the threads of an application
- The various threads of an application interact frequently
  - scheduling decisions concerning one thread may affect the performance of the entire application

Fine-Grained Parallelism

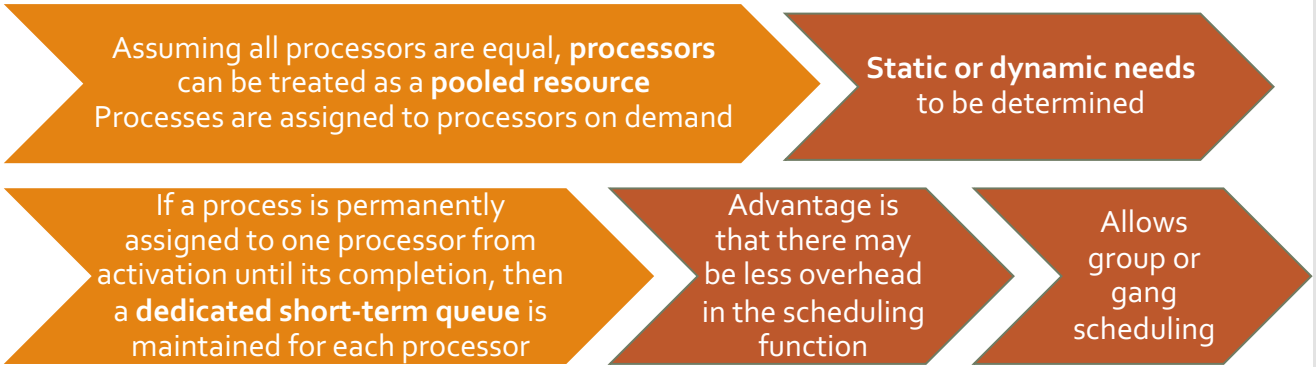
- Represents a much more complex use of parallelism than is found in the use of threads
- Is a specialized and fragmented area with many different approaches

Design Issues

- The approach taken will depend
  - on the degree of granularity of applications
  - and on the number of processors available



Assignment of Processes to Processors



- 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
  - Another option is dynamic load balancing

Assignment of Processes to Processors

- Both dynamic and static methods require some way of assigning a process to a processor
- Approaches
  - Master/Slave
  - Peer

Master/Slave Architecture

- Key kernel functions always run on a particular processor, the master
  - master is responsible for scheduling
  - slaves send service requests to the master
- Simple implementation: 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

Disadvantages:

- Failure of master brings down whole system
- Master can become a performance bottleneck

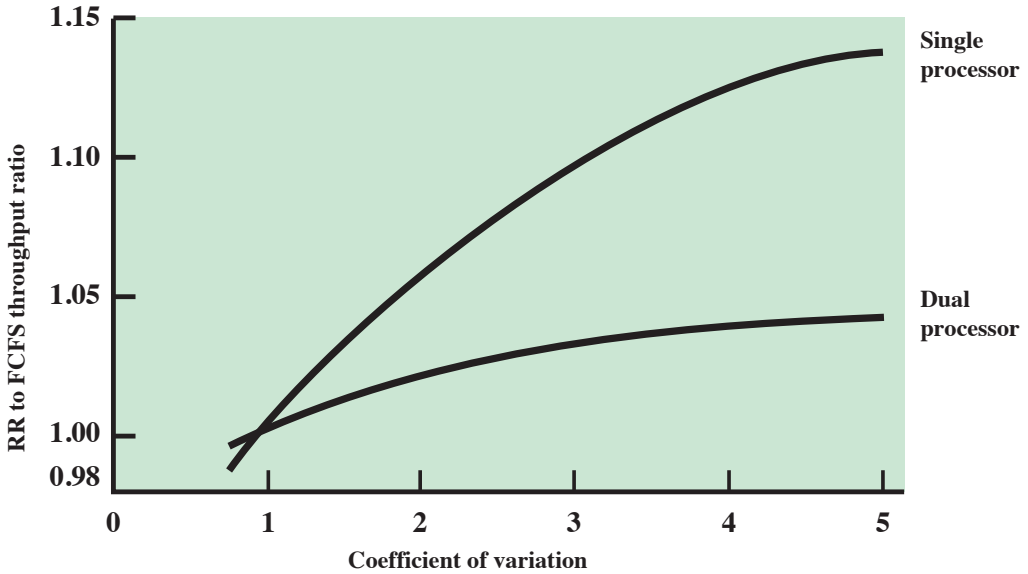
Peer Architecture

- Kernel can execute on any processor
- Each processor does self-scheduling from the pool of available processes

Complicates the operating system

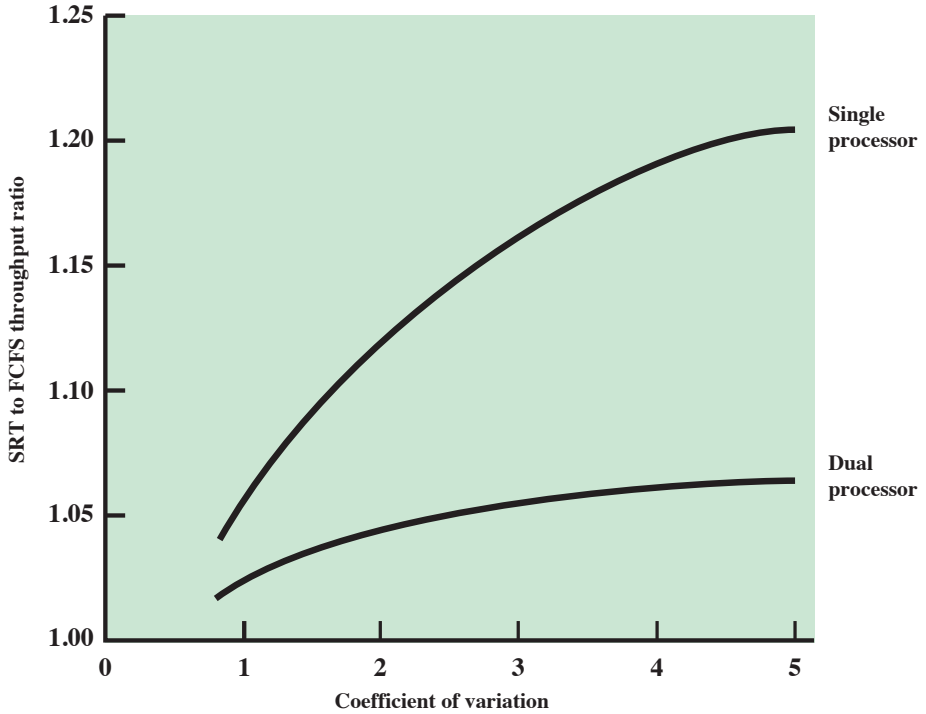
- Operating system must ensure that
  - two processors do not choose the same process
  - and that the processes are not somehow lost from the queue

Performance of scheduling algorithms on multiprocessor architectures



(a) Comparison of RR and FCFS

Performance of scheduling algorithms on multiprocessor architectures



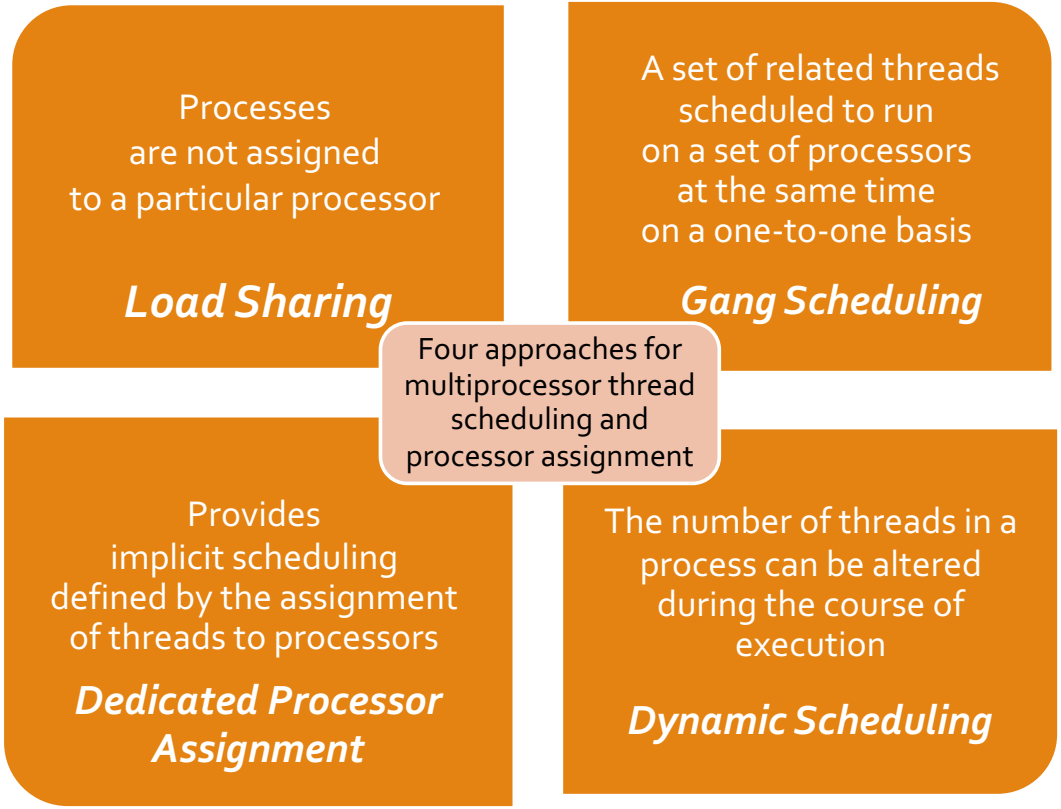
(b) Comparison of SRT and FCFS

Thread Scheduling

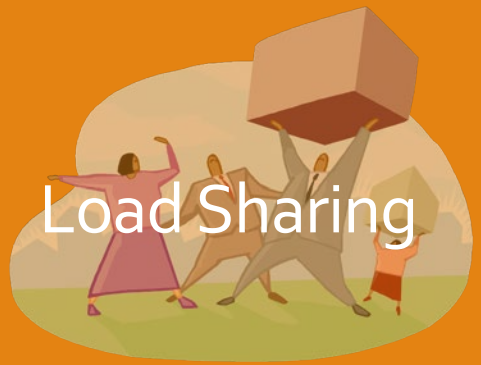
Thread Scheduling

- Thread execution is separated from the rest of the definition of a process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
- On a uniprocessor threads can be used
  - as a program structuring aid
  - to overlap I/O with processing
- In a multiprocessor system threads can be used to exploit true parallelism in an application
  - Dramatic gains in performance are possible
  - Small differences in thread management and scheduling can have an impact on applications that require significant interaction among threads

Approaches to Thread Scheduling



Load Sharing

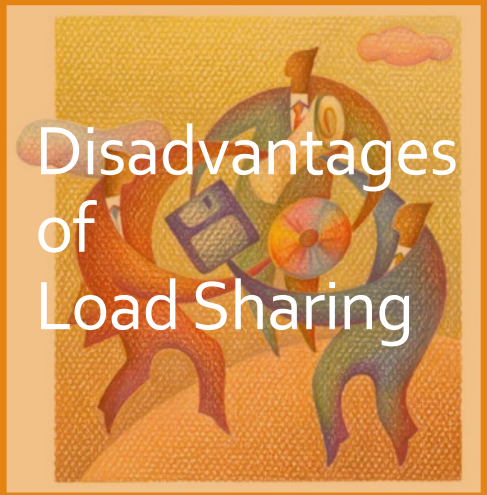


- Simplest approach, directly derived from techniques used in uniprocessor environments
- Versions of load sharing
  - First-come-first-served (FCFS)
  - Smallest number of threads first
  - Pre-emptive smallest number of threads first

ADVANTAGES:

- Load is distributed evenly across the processors, assuring that no processor is idle while work is available to do
- No centralized scheduler required
- The global queue can be organized and accessed using any of the schemes used in uniprocessor systems

Disadvantages of Load Sharing



- Central queue occupies a region of memory that must be accessed in a manner that enforces mutual exclusion
  - can lead to bottlenecks
- Pre-emptive threads are unlikely to resume execution on the same processor
  - caching can become less efficient
- If all threads are treated as a common pool of threads
  - it is unlikely that all of the threads of a program will gain access to processors at the same time
    - the process switches involved may seriously compromise performance

Gang Scheduling

- Simultaneous scheduling of the threads that make up a single process

BENEFITS

- Synchronization blocking may be reduced, less process switching may be necessary, and performance will increase
- Scheduling overhead may be reduced

- Useful for medium-grained to fine-grained parallel applications
  - Performance severely degrades when any part of the application is not running while other parts are ready to run
- Also beneficial for any parallel application

Dedicated Processor Assignment

- When an application is scheduled, each of its threads is assigned to a processor that remains dedicated to that thread until the application runs to completion
- If a thread of an application is blocked waiting for I/O or for synchronization with another thread, then that thread's processor remains idle
  - There is no multiprogramming of processors
- Defense of this strategy
  - In a highly parallel system, with tens or hundreds of processors, processor utilization is no longer so important as a metric for effectiveness or performance
  - The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program

Dynamic Scheduling

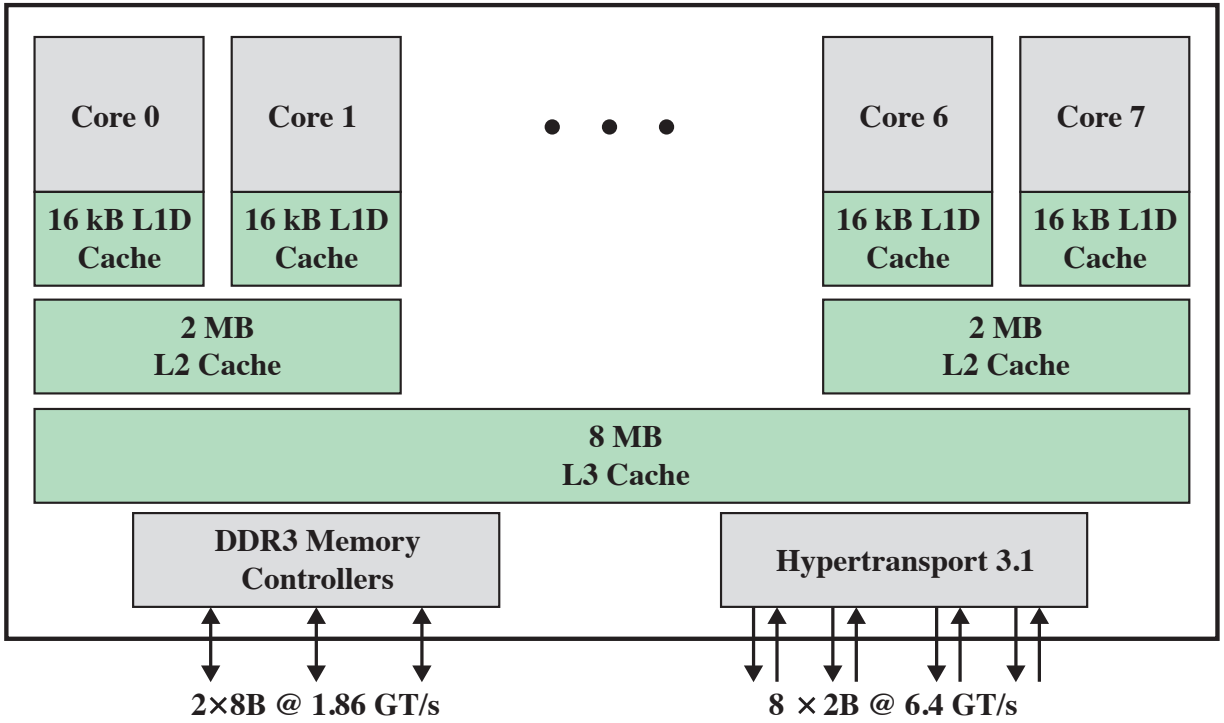
- For some applications it is possible to provide language and system tools that permit the number of threads in the process to be altered dynamically
  - This would allow the operating system to adjust the load to improve utilization
- Both the operating system and the application are involved in making scheduling decisions
  - The scheduling responsibility of the operating system is primarily limited to processor allocation
  - This approach is superior to gang scheduling or dedicated processor assignment for applications that can take advantage of it

Multicore Thread Scheduling

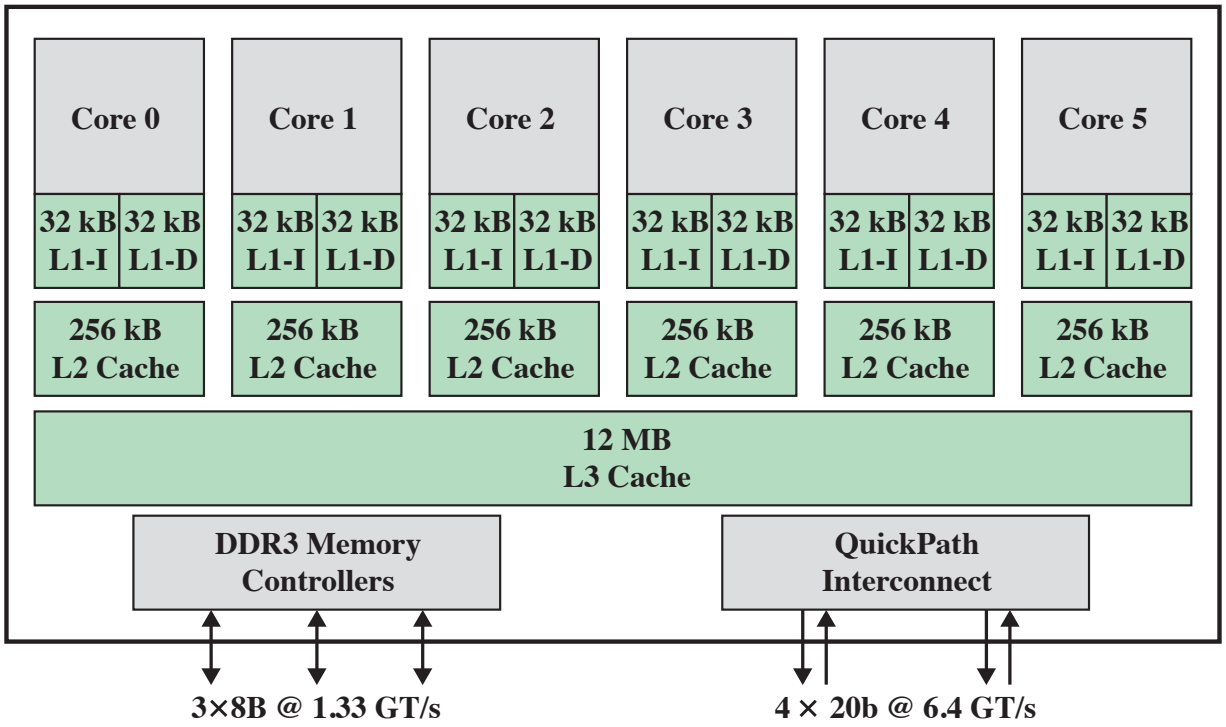
Thread Scheduling in multicore systems

- Scheduling techniques for multicore systems are similar to scheduling techniques used in multiprocessor systems
  - e.g., Windows e Linux
- However, while the goal of **multiprocessor scheduling** is to keep all processors busy, the goal of **scheduling** for **multicore** systems is to minimize the off-chip memory access
  - Load sharing approaches do not provide any performance improvements in multicore systems

AMD Bulldozer Architecture



Intel Core i7-990X Architecture



Cache Sharing

Cooperative resource sharing

- Multiple threads access the same set of main memory locations
- Examples
  - Applications that are multithreaded
  - Producer-consumer thread interaction

Resource contention

- Threads, if operating on adjacent cores, compete for cache memory locations
  - If much of the cache is dynamically allocated to one thread, the competing thread necessarily has less cache space available and thus suffers performance degradation
- Objective of contention-aware scheduling is to allocate threads to cores to maximize the effectiveness of the shared cache memory and minimize the need for off-chip memory accesses

Real-time systems

Real-Time Systems

- The operating system, and in particular the scheduler, is perhaps the most important component

Examples

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

- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced
  - 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

Hard and Soft Real-Time Tasks

Hard real-time task

- One that must meet its deadline
  - Otherwise it will cause unacceptable damage or a fatal error to the system

Soft real-time task

- Has an associated deadline that is desirable but not mandatory
  - It still makes sense to schedule and complete the task even if it has passed its deadline

Periodic and Aperiodic Tasks

- **Periodic tasks**
  - Requirement may be stated as
    - Once per period T
    - Exactly T units apart
- **Aperiodic tasks**
  - Have a deadline by which they must finish or start
  - May have a constraint on both start and finish time

Characteristics of Real Time Systems

Real-time operating systems have requirements in five general areas:

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

Determinism

- Concerned with how long an operating system delays before acknowledging an interrupt
- Operations are performed 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

The extent to which an operating system can deterministically satisfy requests depends on:

The speed with which it can respond to interrupts

Whether the system has sufficient capacity to handle all requests within the required time

Responsiveness

- How long it takes an operating system to service the interrupt after acknowledgment
- Together with determinism make up the response time to external events
- Critical for real-time systems that must meet timing requirements imposed by individuals, devices, and data flows external to the system

Responsiveness includes

- Amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR)
- Amount of time required to perform the ISR
- Effect of interrupt nesting

User Control

- It is essential to allow the user fine-grained control over task priority
  - generally much broader in a real-time operating system than in ordinary operating systems
- User should be able to distinguish between hard and soft tasks and to specify relative priorities within each class
- May allow user to specify such characteristics as

Paging or process swapping

What processes must always be resident in main memory

What disk transfer algorithms are to be used

What rights the processes in various priority bands have

Reliability

- More important for real-time systems than non-real time systems
- RTOS respond to and control events in real time so loss or degradation of performance may have catastrophic consequences such as:
  - **Financial loss**
  - **Major equipment damage**
  - **Loss of life**

Fail-Soft Operation

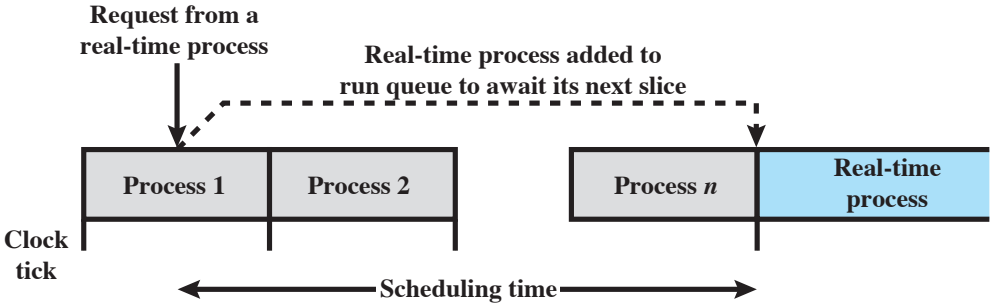
- Refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible
- Important aspect is **stability**
  - A real-time system is stable
    - if the system will meet the deadlines of its most critical, highest-priority tasks
    - even if some less critical task deadlines are not always met
- The following features are common to most RTOS
  - A stricter use of priorities than in an ordinary OS. Preemptive scheduling is designed to meet real-time requirements
  - Interrupt latency is bounded and relatively short
  - More precise and predictable timing characteristics than general purpose OSs

RTOS Scheduling

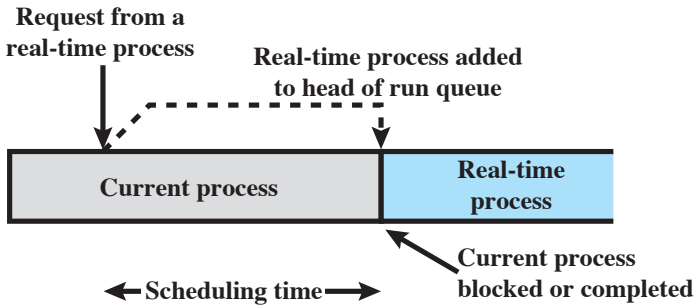
Real-Time Task Scheduling

- A large number of generic OS (Windows, Linux, macOS, etc.) provides support to manage real-time tasks in terms of maximising the responsiveness
  - without requiring the reduction of the average response time
  - without requiring fairness against all processes
- Generic OS do not provide any support to meet task deadlines

Real-Time Process Scheduling

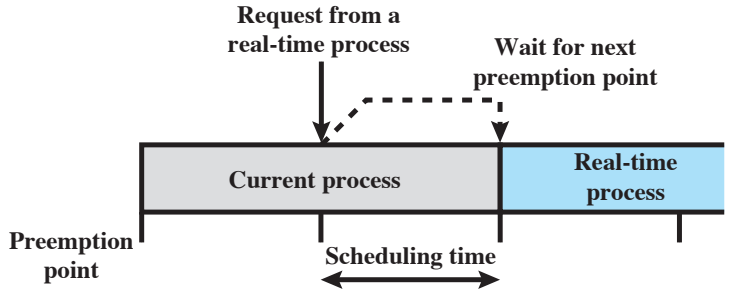


(a) Round-robin Preemptive Scheduler

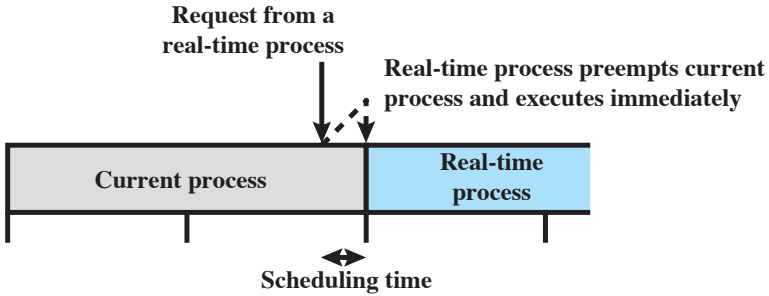


(b) Priority-Driven Nonpreemptive Scheduler

Real-Time Process Scheduling

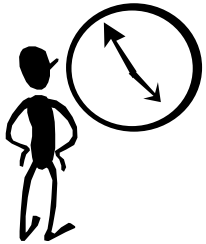


(c) Priority-Driven Preemptive Scheduler on Preemption Points



(d) Immediate Preemptive Scheduler

Real-Time Scheduling



Scheduling approaches depend on

- Whether a system performs schedulability analysis
  - If it does, whether it is done statically or dynamically
- Whether the result of the analysis itself produces a scheduler plan according to which tasks are dispatched at run time

Classes of Real-Time Scheduling Algorithms

Static table-driven approaches

- Static analysis of feasible schedules of dispatching
- Result is a schedule that determines, at run time, when a task must begin execution

Static priority-driven preemptive approaches

- A static analysis is performed but no schedule is drawn up
- Priorities are assigned to tasks and a traditional priority-driven preemptive scheduler can be used

Dynamic planning-based approaches

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

Dynamic best effort approaches

- No feasibility analysis is performed
- System tries to meet all deadlines and aborts any started process whose deadline is missed

This is the most frequent approach in commercial RTOS

Deadline Scheduling

- Objectives of a RTOS:
  - starting real-time tasks as rapidly as possible
  - and emphasize rapid interrupt handling and task dispatching
- Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times
- Priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time

Information Used for Deadline Scheduling

Ready time	• Time 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

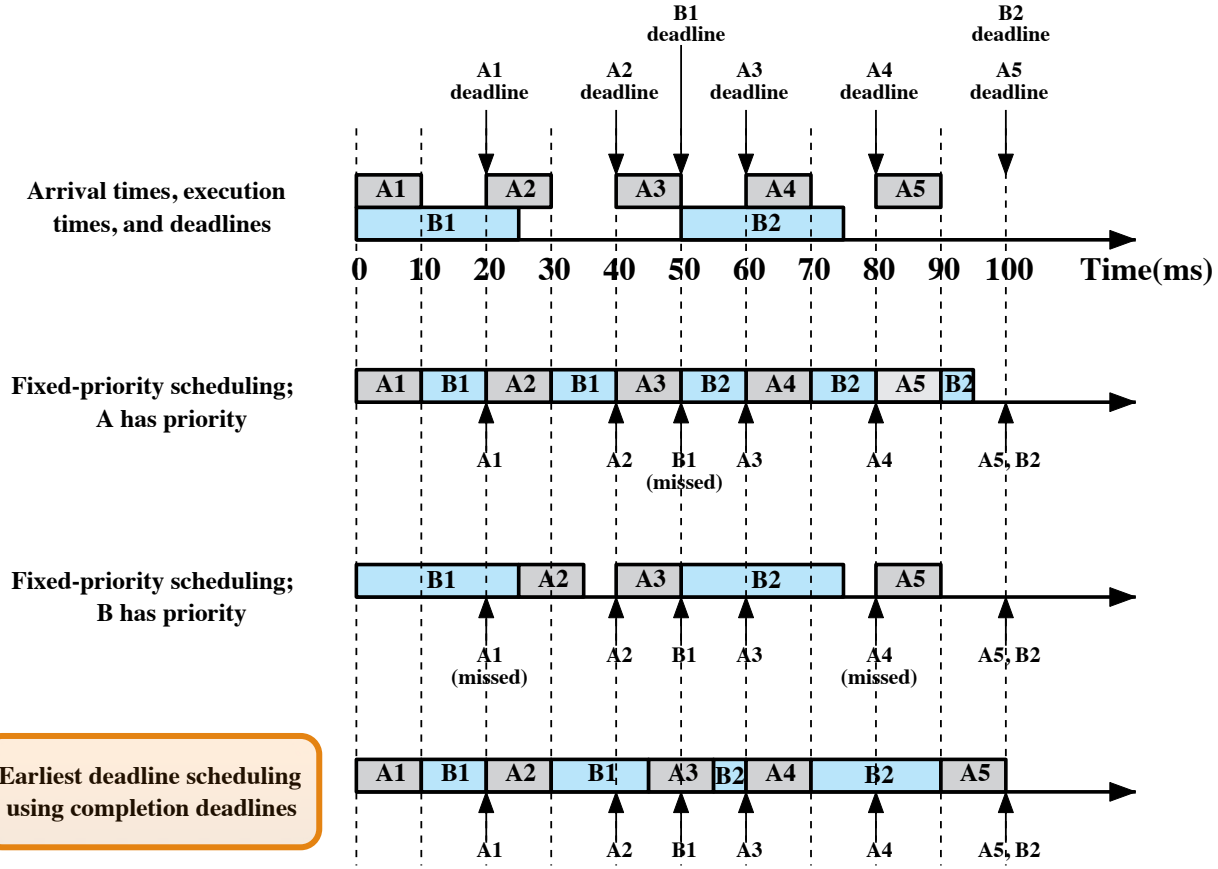
Information Used for Deadline Scheduling

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

Example: 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
•	•	•	•
•	•	•	•
•	•	•	•

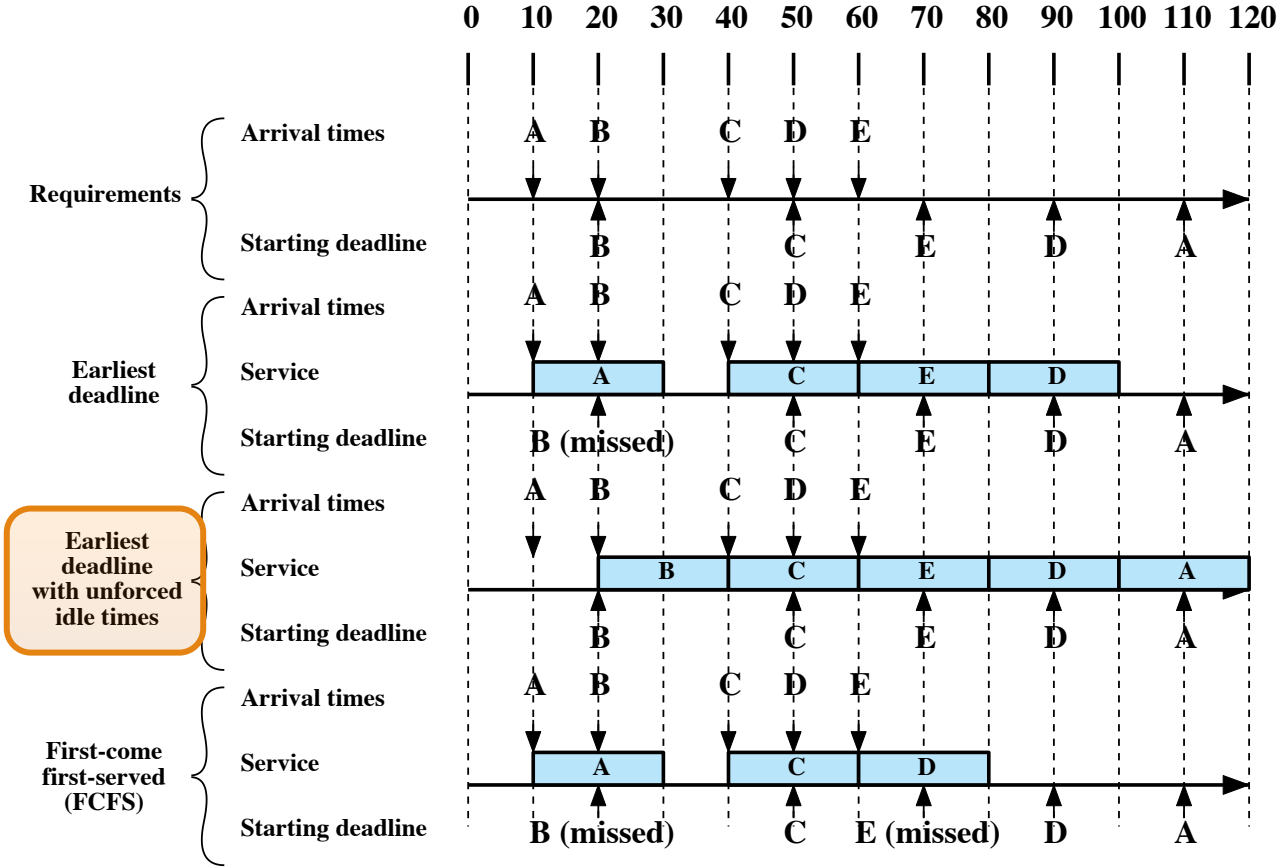
Scheduling of Periodic Real-Time Tasks



Example: Execution Profile of Five Aperiodic Tasks

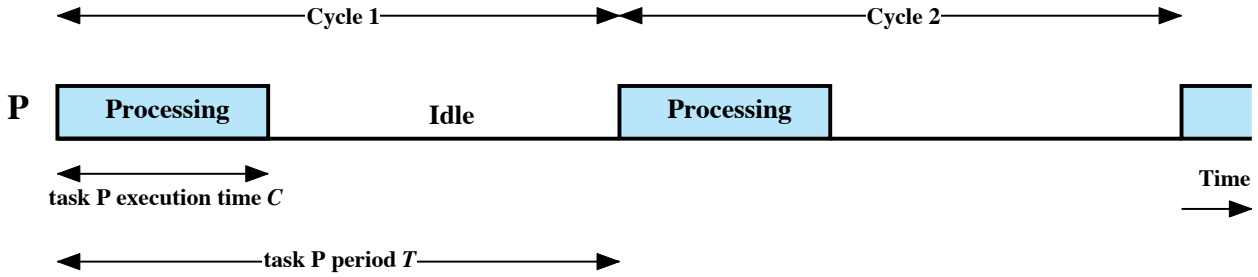
Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

Scheduling of Aperiodic Real-Time Tasks



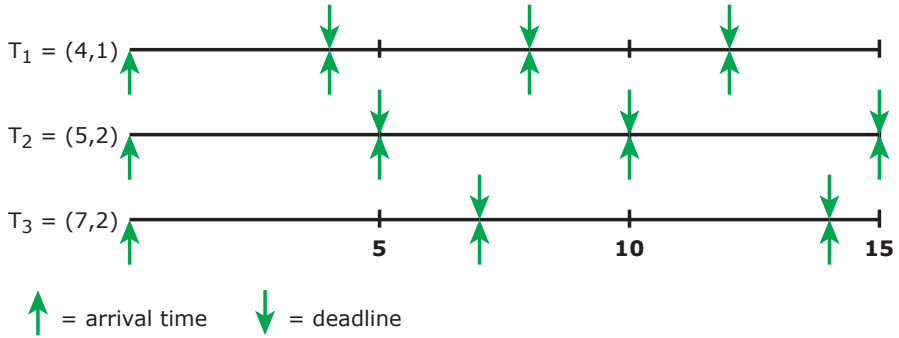
Rate Monotonic Scheduling

- Scheduling based on priorities assigned to tasks on the basis of their periods

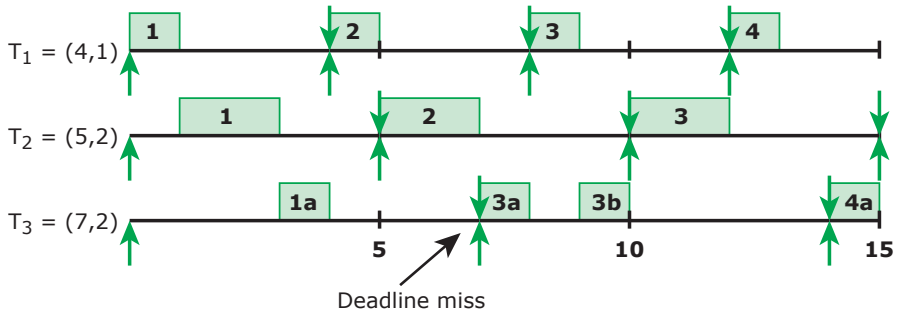


- The highest-priority task is the one with the shortest period.
  - The second highest-priority task is the one with the second shortest period, and so on.
- The plot of priorities of tasks as a function of their rate results in a monotonic function.

Rate Monotonic Scheduling Example



(a) Arrival times and deadines for task $T_i = (P_i, C_i)$; P_i = period, C_i = processing time



(b) Scheduling results

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
- Particularly relevant in the context of real-time scheduling
- Best-known instance involved the Mars Pathfinder mission

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

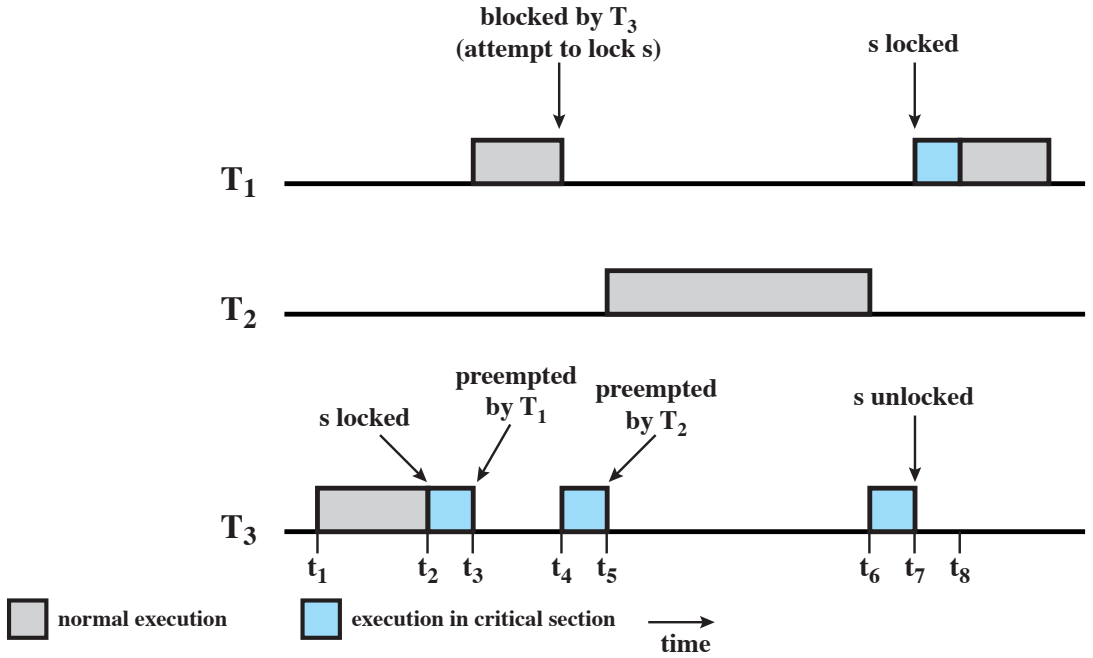
Priority Inversion Example

Some Pathfinder tasks in decreasing order of priority

T1 -> Periodically checks the health of the spacecraft systems and software

T2 -> Processes image data

T3 -> Performs an occasional test on equipment status



Priority Inheritance