THE UNIVERSITY OF NEWCASTLE AUSTRALIA

FACULTY OF ENGINEERING AND BUILT ENVIRONMENT

www.newcastle.edu.au

OPERATING SYSTEMS

Week 4

Much of the material on these slides comes from the recommended textbook by William Stallings

Detailed content

Weekly program

- ✓ Week 1 – Operating System Overview
- ✓ Week 2 – Processes and Threads
- ✓ Week 3 – Scheduling
- ❑ **Week 4 – Real-time System Scheduling and Multiprocessor Scheduling**
- ❑ Week 5 – Concurrency: Mutual Exclusion and Synchronization
- ❑ Week 6 – Concurrency: Deadlock and Starvation
- ❑ Week 7 – Memory Management
- ❑ Week 8 – Disk and I/O Scheduling
- ❑ Week 9 – File Management
- ❑ Week 10 – Real-world Operating Systems: Embedded and Security
- ❑ Week 11 – Real-world Operating Systems: Distributed Operating Systems
- ❑ Week 12 – Revision of the course
- ❑ Week 13 – Extra revision (if needed)

Key Concepts From Last Week

- Two main parameters: Selection function and Selection Mode
- A variety of algorithms have been developed:
  - FCFS
  - RR
  - SPN
  - SRT
  - HRRN
  - FB
- In Fair-Share scheduling the scheduling decisions are made on the basis of process sets rather individual processes.

Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s – e]	$\max\left(\frac{w+s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Through-Put	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

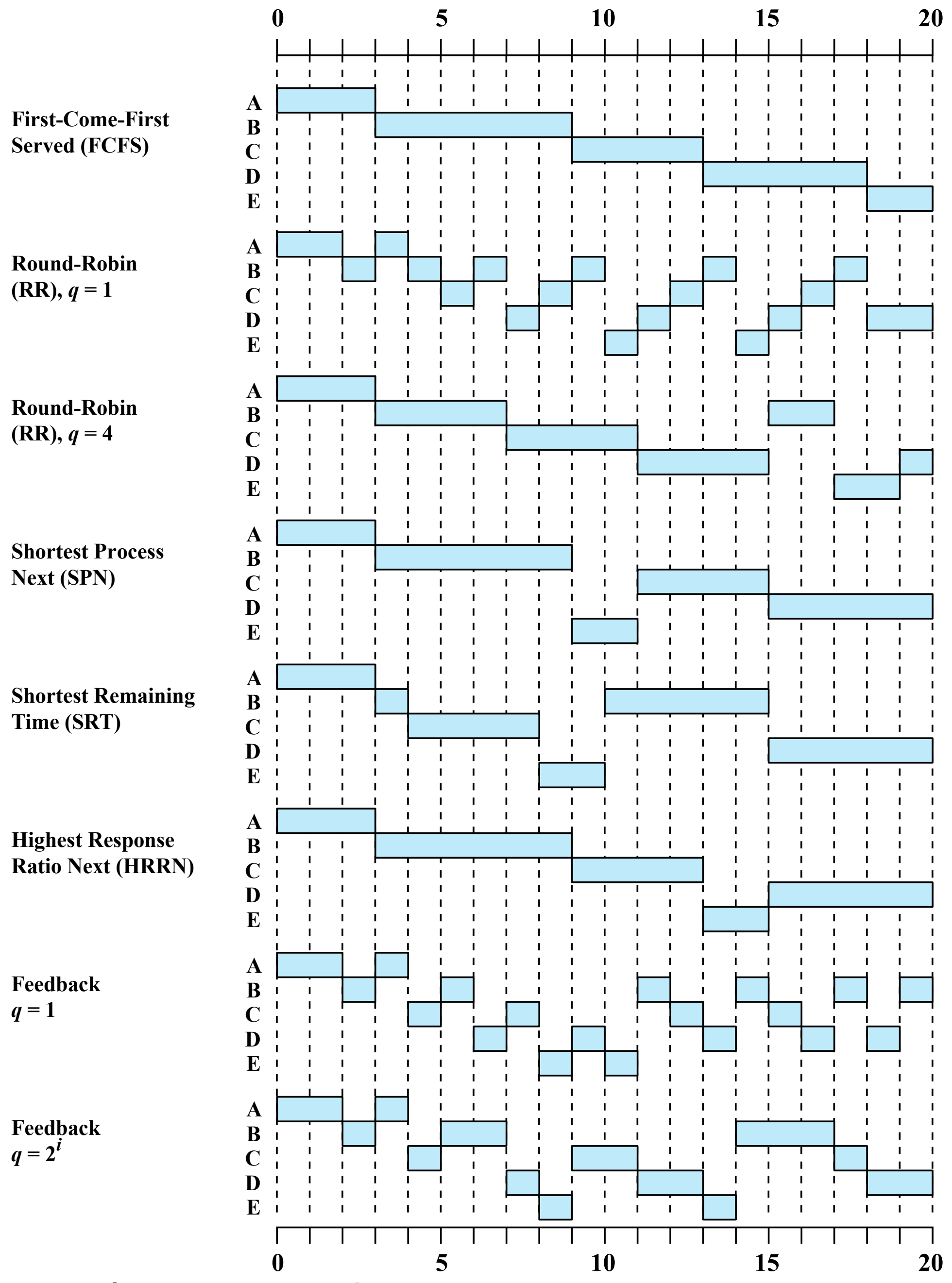


Figure 9.5 A Comparison of Scheduling Policies

A Comparison of Scheduling Policies

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time (T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$						
Finish Time	4	18	17	20	15	
Turnaround Time (T_r)	4	16	13	14	7	10.80
T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$						
Finish Time	3	17	11	20	19	
Turnaround Time (T_r)	3	15	7	14	11	10.00
T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN						
Finish Time	3	9	15	20	11	
Turnaround Time (T_r)	3	7	11	14	3	7.60
T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT						
Finish Time	3	15	8	20	10	
Turnaround Time (T_r)	3	13	4	14	2	7.20
T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN						
Finish Time	3	9	13	20	15	
Turnaround Time (T_r)	3	7	9	14	7	8.00
T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$						
Finish Time	4	20	16	19	11	
Turnaround Time (T_r)	4	18	12	13	3	10.00
T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$						
Finish Time	4	17	18	20	14	
Turnaround Time (T_r)	4	15	14	14	6	10.60
T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

Week 04 Lecture Outline

Real-time System Scheduling and Multiprocessor Scheduling

- ❑ Synchronization Granularity
- ❑ Design Issues in Multiprocessor Scheduling
- ❑ Process Scheduling and Thread Scheduling
- ❑ Approaches to Multiprocessor Scheduling
    - ❑ Load Sharing
    - ❑ Gang Scheduling
    - ❑ Dedicated Processor Assignment
    - ❑ Dynamic Scheduling
- ❑ Multicore thread scheduling
- ❑ Real Time Systems: Hard RT vs Soft RT
- ❑ Characteristics and features of RT systems
- ❑ Real time scheduling approaches
- ❑ Deadline Scheduling
- ❑ RT Scheduling algorithms:
    - ❑ EDFS
    - ❑ RMS

Videos to watch before lecture

Multiprocessor Scheduling

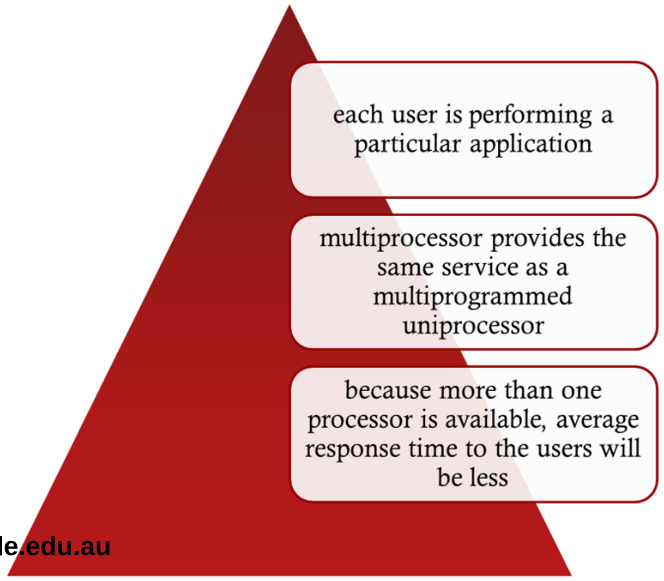
- There are many kinds of multiprocessors:
  - **Loosely coupled multiprocessor**: consists of a collection of relatively autonomous systems, each with its own main memory and I/O channels.
    - Example: clusters, network of workstations,.
  - **Functionally specialised processors**: these are slave processors which provide services for a general purpose processor.
    - Examples: I/O processor, PIPADS image processor linked to i486 master, array processor
  - **Tightly coupled multiprocessing**: a set of processors which share an operating system, and often share a large amount of their memory. Examples: Vax dual processor, Sun10/54, Intel Hypercube, Maspar, Connection machines.

Multiprocessor Scheduling

- We are concerned here with tightly coupled multiprocessing. There are many widely differing architectures and theoretical models for such machines. In general, they can offer:
  - *Reliability*: through redundancy, no total loss of service - just degraded performance
  - *Programming* convenience: through scheduling different threads on different processors.
  - *Performance*: either through parallel algorithms, or through multiprogramming support.

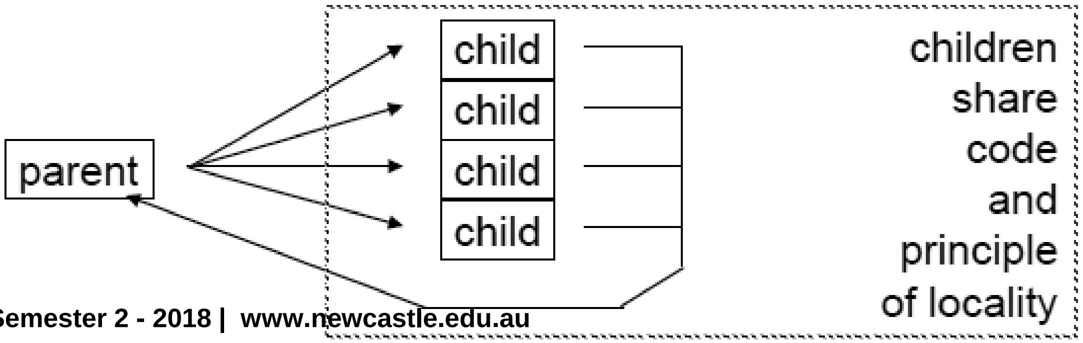
Synchronization Granularity

- Parallelism can be categorised on granularity (frequency of synchronisation):
  - **Independent parallelism**:
    - different jobs run independently on different processors.
    - no explicit synchronization among processes
    - Typical use: Time sharing system: more processors lowers average user response time
    - This is similar to running a network of workstations but more cost-effective


each user is performing a particular application
multiprocessor provides the same service as a multiprogrammed uniprocessor
because more than one processor is available, average response time to the users will be less

Synchronization Granularity

- **Coarse and very coarse grained parallelism**: processes synchronise at a very gross level
  - Can be 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
  - For instance, a process may spawn several children, that run on separate processors, and the results are accumulated in the parent process.
  - Synchronisation interval is 200 – 1,000,000 instructions.



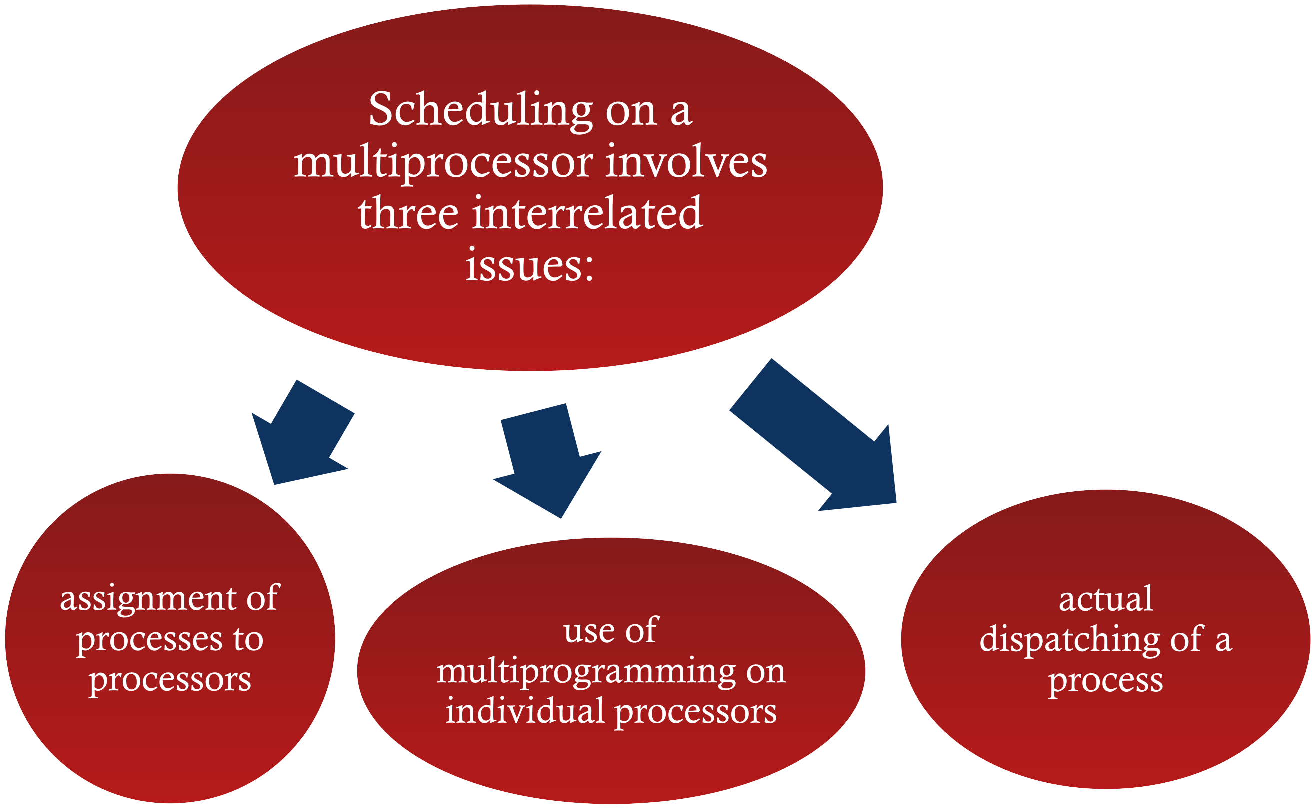
Synchronization Granularity

- **Medium grained parallelism**: a single application can be implemented as a number of threads.
  - This parallelism may be specified by the programmer.
  - There needs to be a high degree of coordination and interaction among the threads of an application, leading to a medium-grain level of synchronization
  - Threads interact very frequently *and scheduling of one thread affects performance of whole application.*
  - Synchronisation interval is 20 - 200 instructions.

Synchronization Granularity

- **Fine grained parallelism**:
  - Represents a much more complex use of parallelism than is found in the use of threads
  - The programmer must use special instructions and write **parallel programs**.
  - Tends to be very specialised and fragmented with many different approaches
  - Synchronisation interval is less than 20 instructions, possibly at the single instruction level.

Design Issues


Scheduling on a multiprocessor involves three interrelated issues:
assignment of processes to processors
use of multiprogramming on individual processors
actual dispatching of a process

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

Design Issues

- Assignment of processes to CPUs for uniform multiprocessor
  - *Static*: a process is assigned to a processor for the total life of the process. A short-term queue is maintained for each processor
    - a simple approach, with very little scheduling overhead.
    - the main disadvantage is that one processor may be idle while another has a long queue.
  - *Dynamic*: a process may change processors during its lifetime. A single global queue is maintained - jobs scheduled to any available processor.
    - more efficient than static assignment
    - if shared-memory, context info available to all processors, cost of scheduling is independent of processor identity

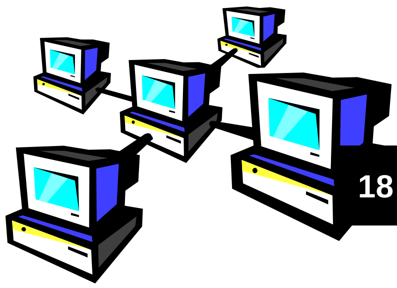
Design Issues

- Means of assigning processes to processors
  - Master-slave architecture: the OS runs on a particular processor and schedules user tasks on other processors. Resource control is simple.
    - very simple and requires little enhancement to a uniprocessor multiprogramming operating system.
    - the master can be performance bottleneck and failure of the master is disastrous.
  - Peer architecture: the OS can run on any processor, and effectively each processor schedules itself.
    - complicates the OS
    - conflict resolution (two processors want the same job or resource) becomes complex.

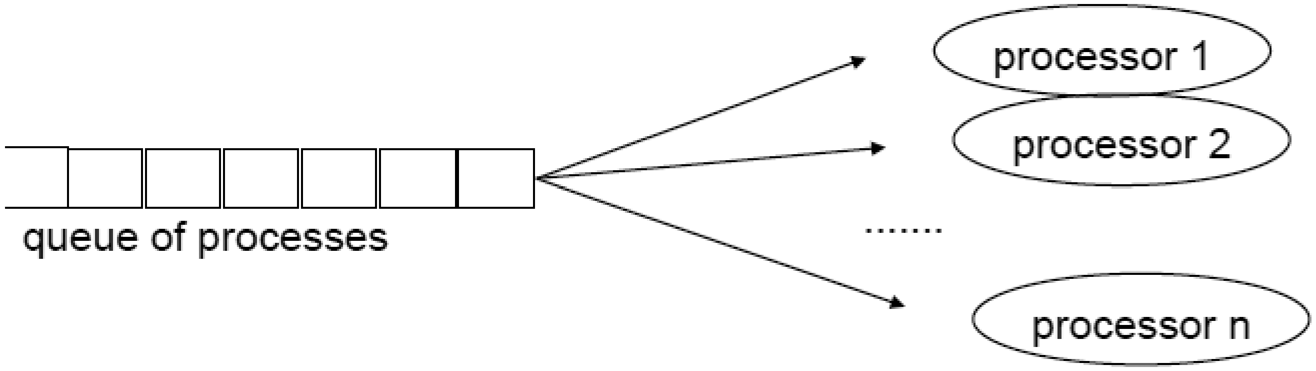
Design Issues

- **Multiprogramming on individual processors**
  - for **coarse grained parallelism**, multiprogramming is necessary for each processor, in the same way as for a uniprocessor - to get higher usage and better performance.
  - however if there are many processors (medium- or fine-grained parallelism) then it may not be necessary to keep all processors busy all the time.
  - Better throughput may be achieved if some processors are sometimes idle. WHY?
  - Thus multiprogramming on individual processors may not be necessary. We're looking for best <u>average</u> performance of applications.
- **Process Dispatching**
  - the complex selection procedure involved in uniprocessors may not be necessary and could even be detrimental; a simple FIFO strategy may be best - less overhead.
  - the issues can be quite different for scheduling **threads**.

Process Scheduling

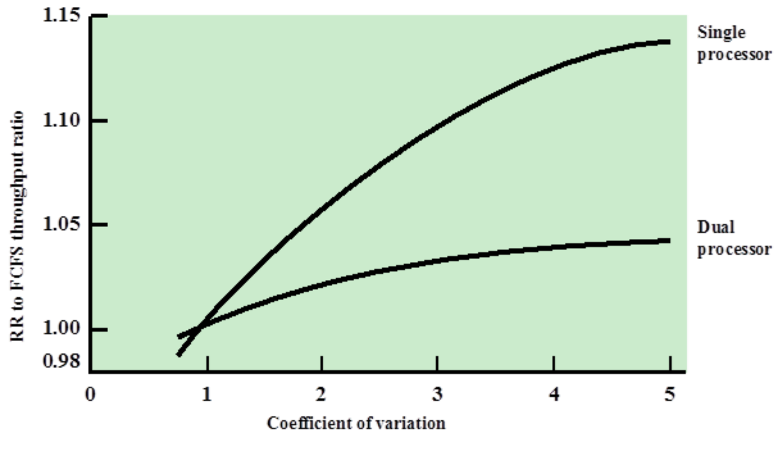


- Generally processes are *not* dedicated to processors.
- Processes can be scheduled from a single queue to multiple processors. Each processor chooses its next process from the queue.
- Or there is a multiple queue priority scheme.

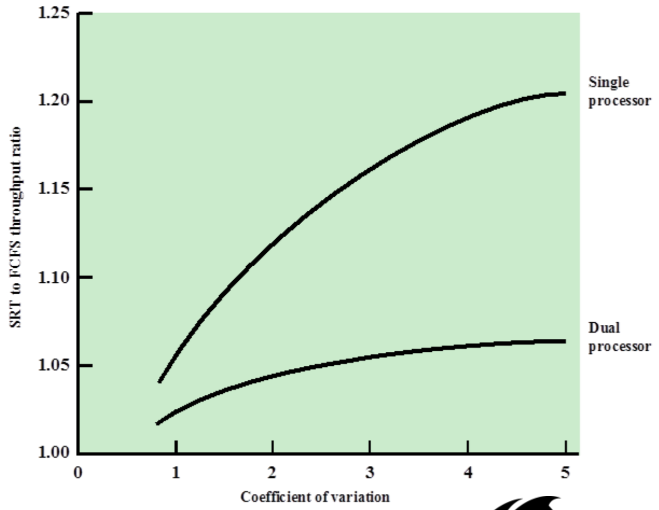


Process Scheduling

- Experience has shown that for **processes** (tasks are relatively independent), a simple FCFS policy for scheduling is best.
  - If the job mix is highly variable, then Round Robin is better than FCFS, both on a single processor and a dual processor.
  - However, the improvement of RR over FCFS is very small for the dual processor, and becomes even smaller for more than two processors.



(a) Comparison of RR and FCFS



(b) Comparison of SRT and FCFS

Thread Scheduling

- Threads are different from processes in ways that are important for scheduling:
  - the overhead of thread switching is smaller than for process switching
  - threads share resources (including memory) and there is some principle of locality.
- Threads are important for exploiting medium grained parallelism. Thread scheduling is important for exploiting true parallelism as much as possible.
  - Threads simultaneously running on separate processors give big performance gains.
  - Thread scheduling models significantly impact performance.
- Experience with thread scheduling is increasing; research is still active and ideas are broad.

Approaches to Thread Scheduling

Four approaches for multiprocessor thread scheduling and processor assignment are:

Load Sharing/ Self Scheduling

processes are not assigned to a particular processor

Gang Scheduling

a set of related thread scheduled to run on a set of processors at the same time, on a one-to-one basis

Dedicated Processor Assignment

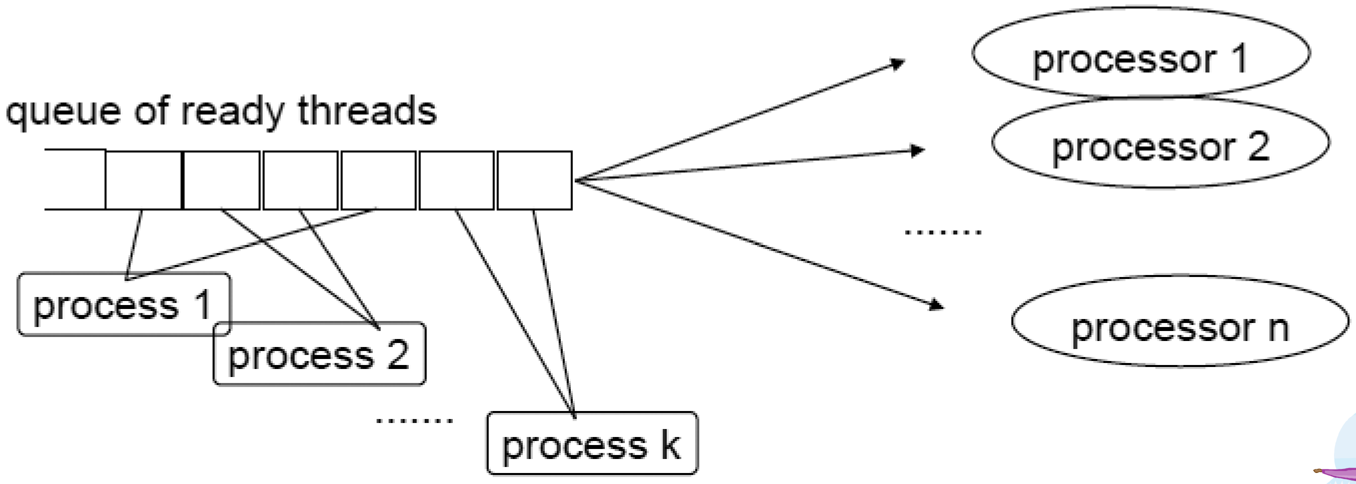
provides implicit scheduling defined by the assignment of threads to processors

Dynamic Scheduling

the number of threads in a process can be altered during the course of execution

Load Sharing/Self Scheduling

- For **Self Scheduling,** processes are not dedicated to any particular processor. A global queue of ready threads is maintained, and an idle processor selects a thread from the queue.



Load Sharing/Self Scheduling

- The advantages of self scheduling are:
  - even load distribution, no processor is idle while there is work to do.
  - no centralised scheduler is required, use this processor to determine next thread to run.
  - the global queue can be organised in an appropriate way for instance, on priority, or execution history, or estimated need .
- Self scheduling is **quite common**.

Load Sharing/Self Scheduling

- There are three types of self scheduling:
  - **FCFS**: the ready queue of threads is maintained as a FIFO queue; an idle processor always chooses the process at the head of the queue. Each thread is run non-preemptively (that is, it is run until it either completes or blocks itself by an I/O request)
  - **Smallest number of threads first**: highest priority is given to processes with the smallest number of unscheduled threads.
  - **Pre-emptive smallest number of threads first**: highest priority is given to processes with the smallest number of unscheduled threads. If a job arrives with a smaller number of threads than one which is running, then the running job will be pre-empted.
- Some research suggests that FCFS is better than the other two (more complicated) policies.

Load Sharing/Self Scheduling

- **Disadvantages** of self scheduling are:
  - Mutual exclusion on the central queue must be enforced. This can be a bottleneck, especially with a large number of processors (10's -> 100's).
  - Since pre-empted threads are unlikely to resume execution on the same processor, local caching becomes less effective.
  - It is unlikely that all threads of a program will gain access to processors at the same time. This limits thread communication and is serious if high coordination is required.

Gang Scheduling

- In **Gang Scheduling** (*group scheduling*, *co-scheduling*), a set of related threads are scheduled on a set of processors at the same time. Some rationale behind gang scheduling is:
  - if closely related threads execute in parallel, then synchronisation blocking will be reduced, and less process switching will occur.
  - scheduling overhead is reduced by making a single decision for a group of threads.
  - Useful for medium-grain or fine-grained parallel applications
  - Process switching is minimized

- The set or **gang** of threads can be defined by the programmer, or it can be automatically defined (for example, all threads of a particular process can be a gang).

Processor Allocation in Gang Scheduling

- **Uniform Scheduling**: Each application gets 1/M of the available time in N processor using time slicing.
- **Weighted**: The amount of N processor time an application gets is weighted by the number of threads in that application

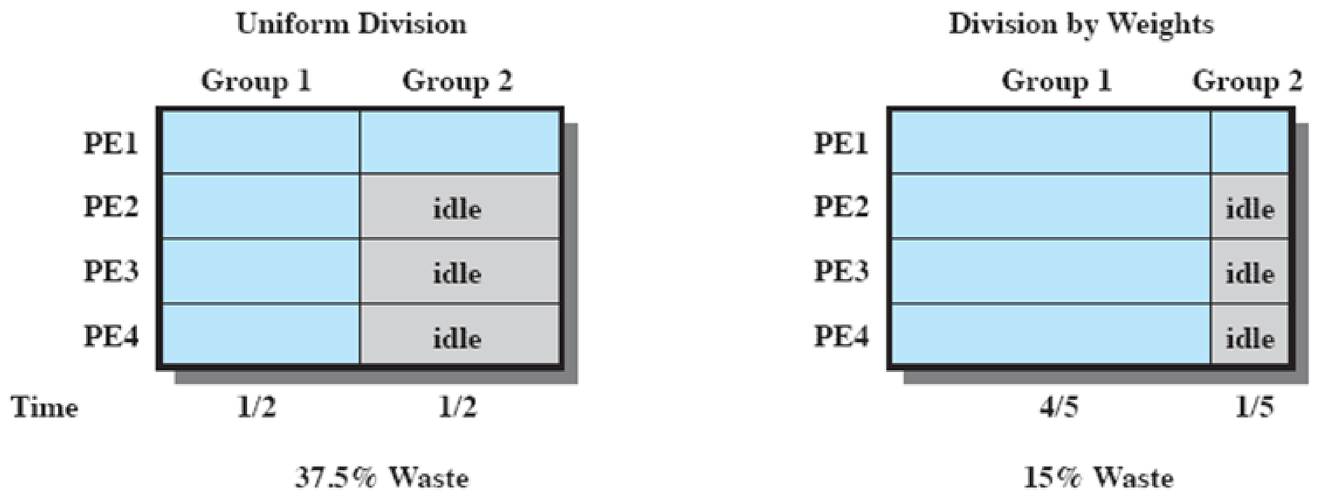
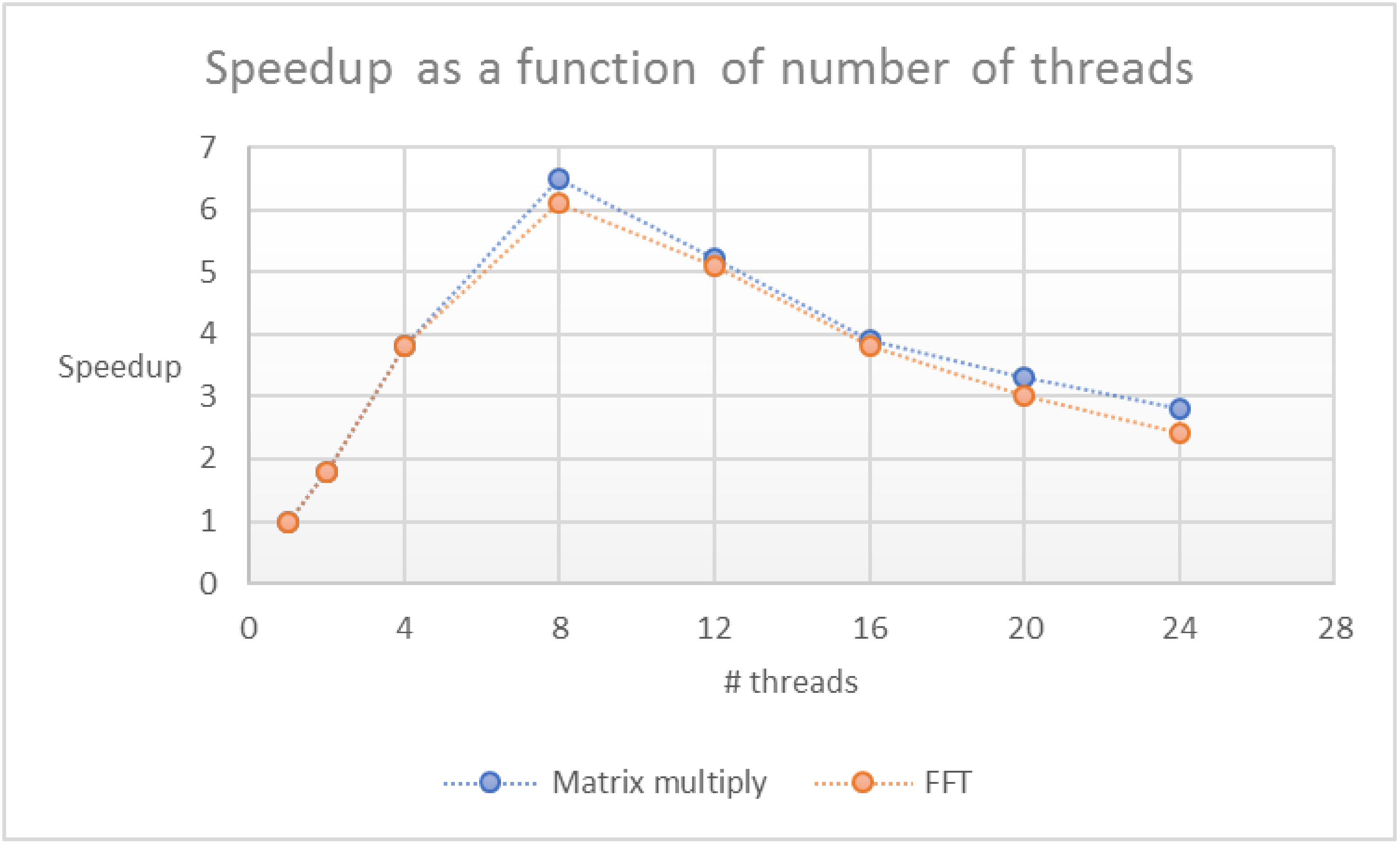


Figure 10.3 Example of Scheduling Groups with Four and One Threads [FEIT90b]

Dedicated Processor Assignment

- In **Dedicated Processor Assignment**, a group of processors is assigned to a job for the complete duration of the job.
- An extreme form of gang scheduling
- Each thread is assigned to a processor and this processor remains dedicated to that thread until the job completes.
- This approach results in idle processors (there is no multiprogramming).
- 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

Dedicated Processor Assignment



# of processors in the system is 16

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
  - For example, the OS can partition the processors among the jobs, and each job partitions its threads among its processors.

- Not suitable for all applications.

Dynamic Scheduling

- The only responsibility of the OS is to allocate processors to jobs.
- When a job requests processors (either when the job arrives or when it creates new threads), the OS acts as follows:
  - if there are idle processors, then the request is satisfied.
  - otherwise, if the job is new then it is allocated a single processor by pre-empting a job which has more than one processor.
  - if the request cannot be satisfied, the request waits in a queue for a processor to become available, or until the job rescinds the request (because it's not needed any more).
  - when processors are released, scan queue of unsatisfied requests. Assign one processor per new process, then allocate to existing requesting processes on a FCFS basis.
- Can be superior to gang and dedicated processor scheduling, but overheads can negate this.

Memory Management VS Processor Management

- Processor scheduling on a multiprocessor is rather like memory management on a uniprocessor. Similar analyses can apply.
- For instance
  - the question of **how many processors to allocate to a job** is analogous to the question of **how many pages to allocate to a job**.
  - **processor thrashing** occurs when the scheduling of threads whose services are required induces the de-scheduling of threads whose services will soon be needed.
  - **processor fragmentation** occurs when a number of processors are left over when others are allocated, and the leftover processors are insufficient to satisfy requests from waiting jobs.
- The connection between processor and memory availability is deep and has a theoretical basis.

Multicore Thread Scheduling

- Contemporary OSs, such as Windows and Linux, essentially treat scheduling in multicore systems in the same fashion as a multiprocessor system.
  - Focus on keeping processors busy by load balancing
  - Unlikely to produce the desired performance benefits of the multicore architecture

- As the number of cores per chip increases,
  - a need to minimize access to off chip memory takes precedence over a desire to maximize processor utilization.
  - Means: use of caches to take advantage of locality.
  - Complicated by some of the cache architectures used on multicore chips
    - Specifically when a cache is shared by some but not all of the cores.

Cache Sharing

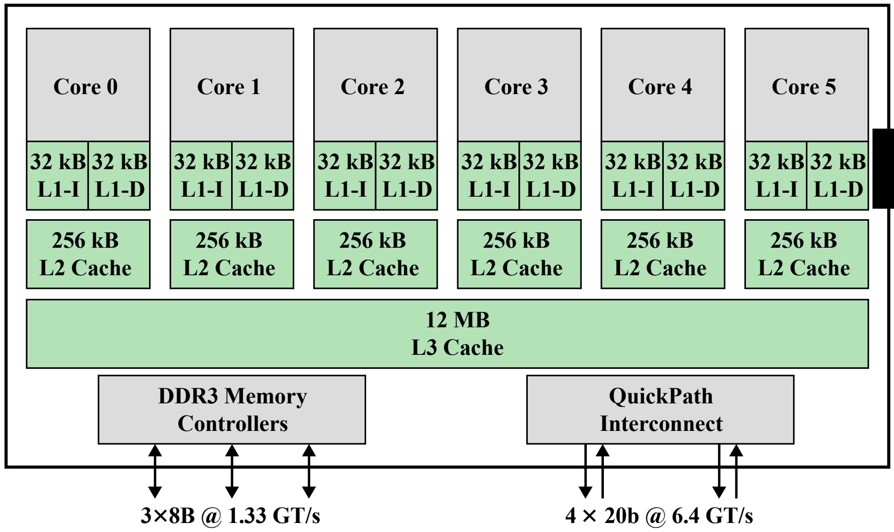


Figure 1.20 Intel Core i7-990X Block Diagram

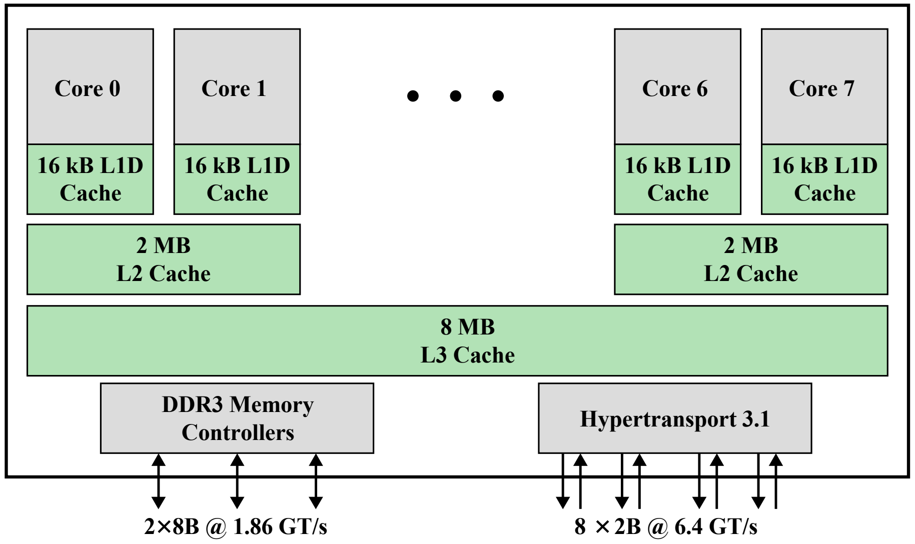


Figure 10.3 AMD Bulldozer Architecture

Cache Sharing

Two aspects of 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 more 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

- 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

Real Time System

- Correctness of the system may depend not only on the logical result of the computation but also ***on the time*** when these results are produced

- e.g.
  - Tasks attempt to control events or to react to events that take place in the outside world
  - These external events occur in *real time* and processing must be able to keep up
  - Processing must happen in a timely fashion
    - Neither too late, nor too early

Hard-Real time system

- Requirements:
  - **Must *always* meet all deadlines** (time guarantees)
  - You have to guarantee that in any situation these applications are done in time,
  - otherwise it will cause unacceptable damage or a fatal error to the system

- Examples:
  1. If the landing of a fly-by-wire jet cannot react to sudden side-winds within some milliseconds, an accident might occur.
  2. An airbag system or the ABS has to react within milliseconds

Soft-Real Time Systems

- Requirements:
  - **Must mostly meet all deadlines**
  - 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

- Examples:
  1. Multimedia: 100 frames per day might be dropped (late)
  2. Car navigation: 5 late announcements per week are acceptable
  3. Washing machine: washing 10 sec over time might occur once in 10 runs, 50 sec once in 100 runs.

Properties of Real-Time Tasks

- To schedule a real time task, its properties must be known

- The most relevant properties are
  - Arrival time (or release time)
  - Maximum execution time (service time)
  - Deadline
    - Starting deadline
    - Completion deadline

Categories of Real Time Tasks

- **Periodic**
  - Each task is repeated at a regular interval
  - Max execution time is the same each period
  - Arrival time is usually the start of the period
  - Deadline is usually the end

- **Aperiodic (sporadic)**
  - Each task can arrive at any time

Characteristics of Real-Time System

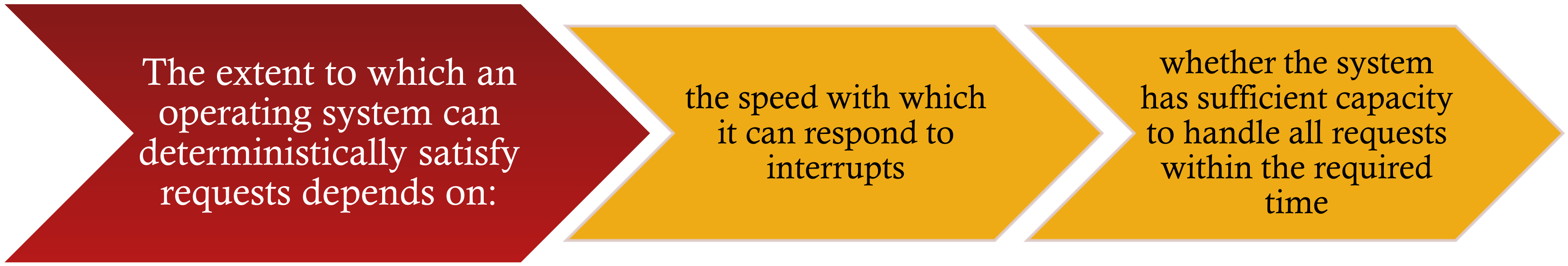
Real-time operating systems have requirements in five general areas:

- Determinism
- Responsiveness
- User Control
- Reliability
- Fail-safe 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

- Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt

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

- OS Response Time = Determinism + Responsiveness
- 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

User Control

- Generally much broader in a real-time operating system than in ordinary operating systems
- It is essential to allow the user fine-grained control over task priority
- 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
- Real-time systems 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

- 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

- 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

Characteristics of Real-Time System

- Determinism
  - how long an OS delay before acknowledging an interrupt
- Responsiveness
  - After acknowledgment, how long does it take to service the interrupt.
- User Control
  - Fine-grained control over task priority
- Reliability
- Fail-safe operation
  - Preserve as much capability and data as possible when system fail
  - Attempt to correct the problem or minimize its effects while continuing to run
- Stability
  - at least meet all task deadlines of most critical, high-priority tasks even if some less critical task deadlines are not always met.

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

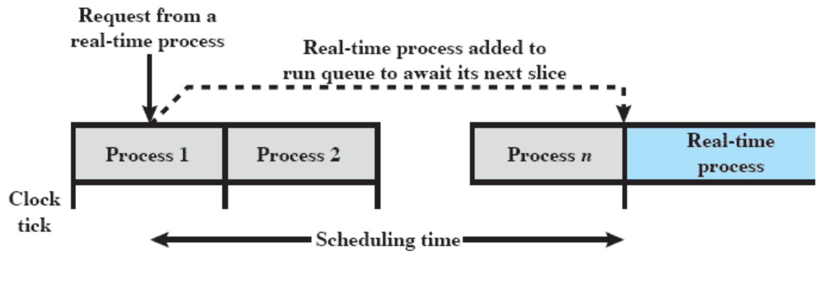
Common Features of Real-Time Systems

Compared to general purpose OS, in RT OS

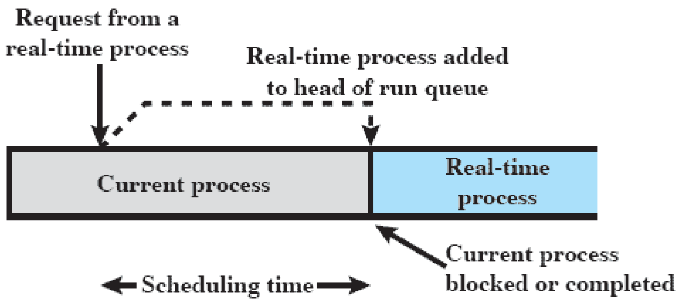
- Priorities are used more strictly
    - Preemptive scheduling designed to meat RT requirements
- Interrupt latency is bounded and relatively short
- More precise and predictable timing characteristics

- Heart of RT system is short-term task scheduler
    - Fairness, minimizing average response time are not paramount
    - Deadline is most important!!
- Most RT OS are unable to deal deadlines directly
    - Designed to be as responsive as possible to RT tasks scheduling
    - Require deterministic response time in milli/micro second ranges

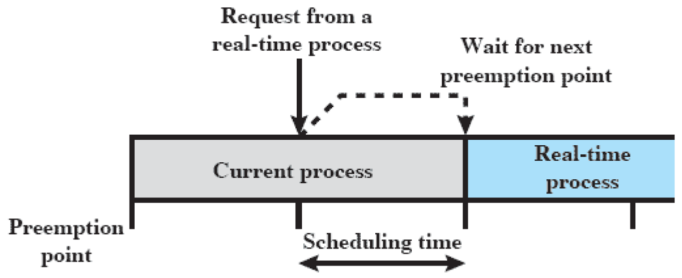
Real-time scheduling of process


Request from a real-time process
Real-time process added to run queue to await its next slice
Process 1
Process 2
Process n
Real-time process
Clock tick
Scheduling time

(a) Round-robin Preemptive Scheduler

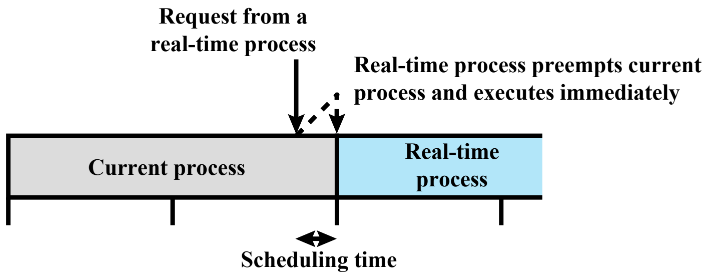

Request from a real-time process
Real-time process added to head of run queue
Current process
Real-time process
Current process blocked or completed
Scheduling time

(b) Priority-Driven Nonpreemptive Scheduler



(c) Priority-Driven Preemptive Scheduler on Preemption Points

Scheduling delay: order of milliseconds


Request from a real-time process
Real-time process preempts current process and executes immediately
Current process
Real-time process
Scheduling time

(d) Immediate Preemptive Scheduler

Scheduling delay: order of 100 µS or less

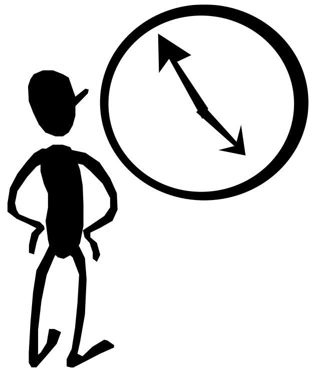
Real Time Scheduling

- RTS accepts an activity *A* and guarantees its requested (timely) behaviour *B* if and only if
  - RTS finds a *schedule*
    - that includes all already accepted activities A_i and the new activity *A*,
    - that guarantees all requested timely behaviour B_i and *B*, and
    - that can be enforced by the RTS.

- Otherwise, RT system rejects the new activity *A*.

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



Real-time Scheduling Approaches

- **Static table-driven scheduling**
  - Given a set of tasks and their properties, a schedule (table) is precomputed offline.
    - Used for periodic task set
    - Requires entire schedule to be recomputed if we need to change the task set

- **Static priority-driven scheduling**
  - Given a set of tasks and their properties, each task is assigned a fixed priority based on some static analysis
  - A preemptive priority-driven scheduler used in conjunction with the assigned priorities
    - Used for periodic task sets

Real-time Scheduling Approaches

- **Dynamic planning-based scheduling**
  - Task arrives prior to execution
  - The scheduler determines whether the new task can be admitted
    - Can all other admitted tasks and the new task meet their deadlines?
      - If no, reject the new task
  - Can handle both periodic and aperiodic tasks

Real-time Scheduling Approaches

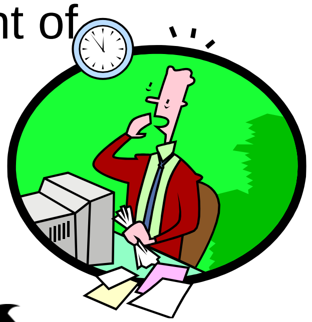
- **Dynamic best effort Scheduling**
  - No feasibility analysis is performed
  - When tasks arrives system assigns a priority to the task based on its characteristics
  - System tries to meet all deadlines, abort the started process which has missed the deadline
  - Can’t guarantee the timing constraint of a task will be met until it is completed
  - Usually tasks are aperiodic
  - Used by many current commercial RT systems

Deadline Scheduling

- Real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible
  - emphasize rapid interrupt handling
  - task dispatching

- 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

- 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
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

Usage of Deadlines in RT Scheduling Function

- **Which Task to Schedule?**
  - For a given preemption strategy and using either starting or completion deadlines, a policy that schedules that task with the earliest deadlines, minimizes the fraction of tasks that miss their deadlines
- **What sort of Preemption is allowed?**
  - When starting deadlines are specified nonpreemptive scheduler makes sense
    - RT task should block itself after completing the mandatory or critical portion of its execution
  - When ending deadlines are specified preemptive scheduler is most appropriate
    - Task X is running and 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.

Scheduling in Real-Time Systems

- We will consider periodic systems

- Schedulable real-time system
  - Given
    - n periodic events
    - event i occurs within period T_i and requires C_i seconds
    - $U_i = \frac{C_i}{T_i}$ is called processor utilization

- Then the load can only be handled if

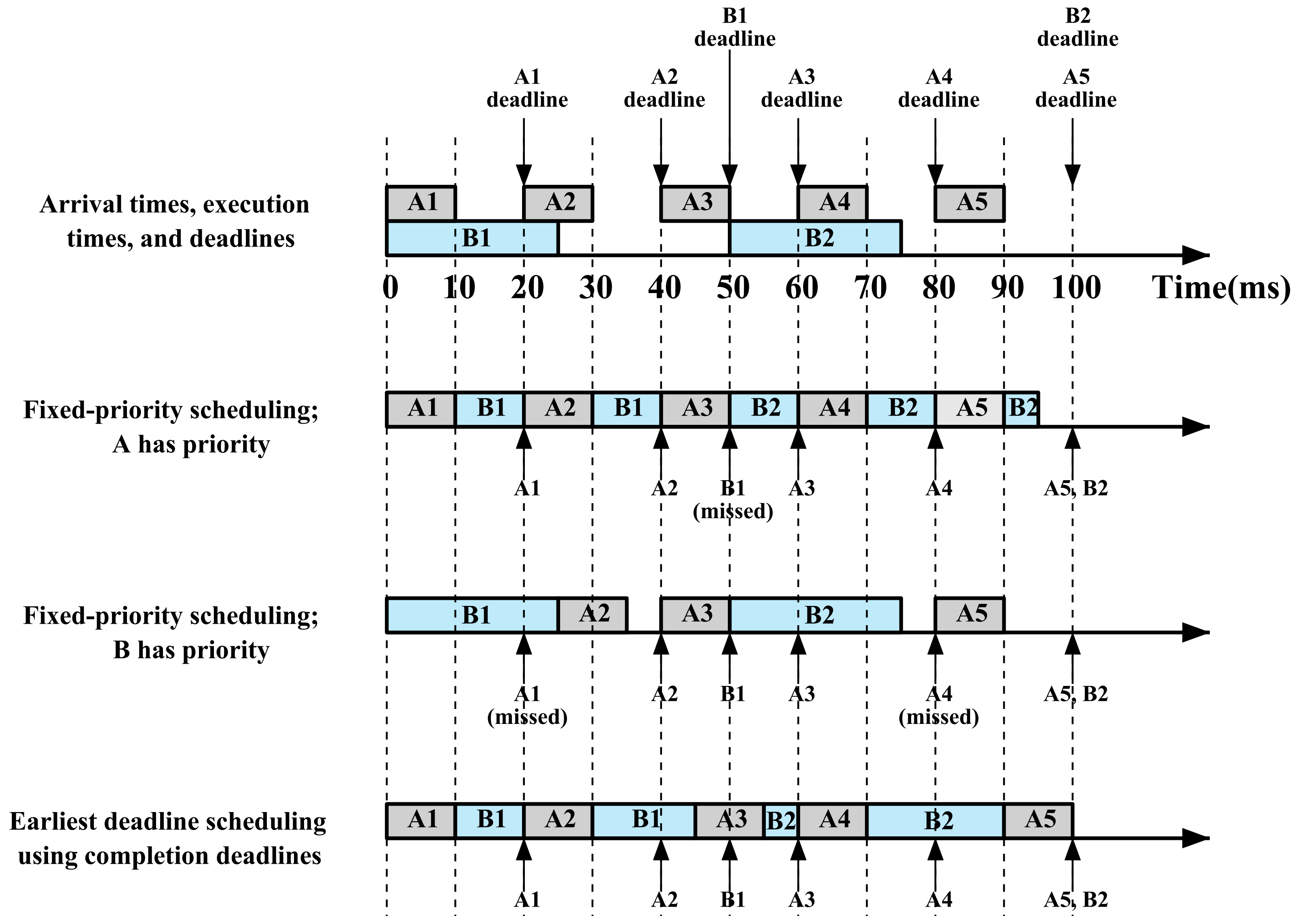
$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 1$$

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
• • •	• • •	• • •	• • •

Real-time Scheduling Algorithms

- **Earliest Deadline First Scheduling**
  - The task with the earliest deadline is chosen next
  - The deadline could be
    - Completion deadline
    - Starting deadline



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

Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

Table 10.4
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

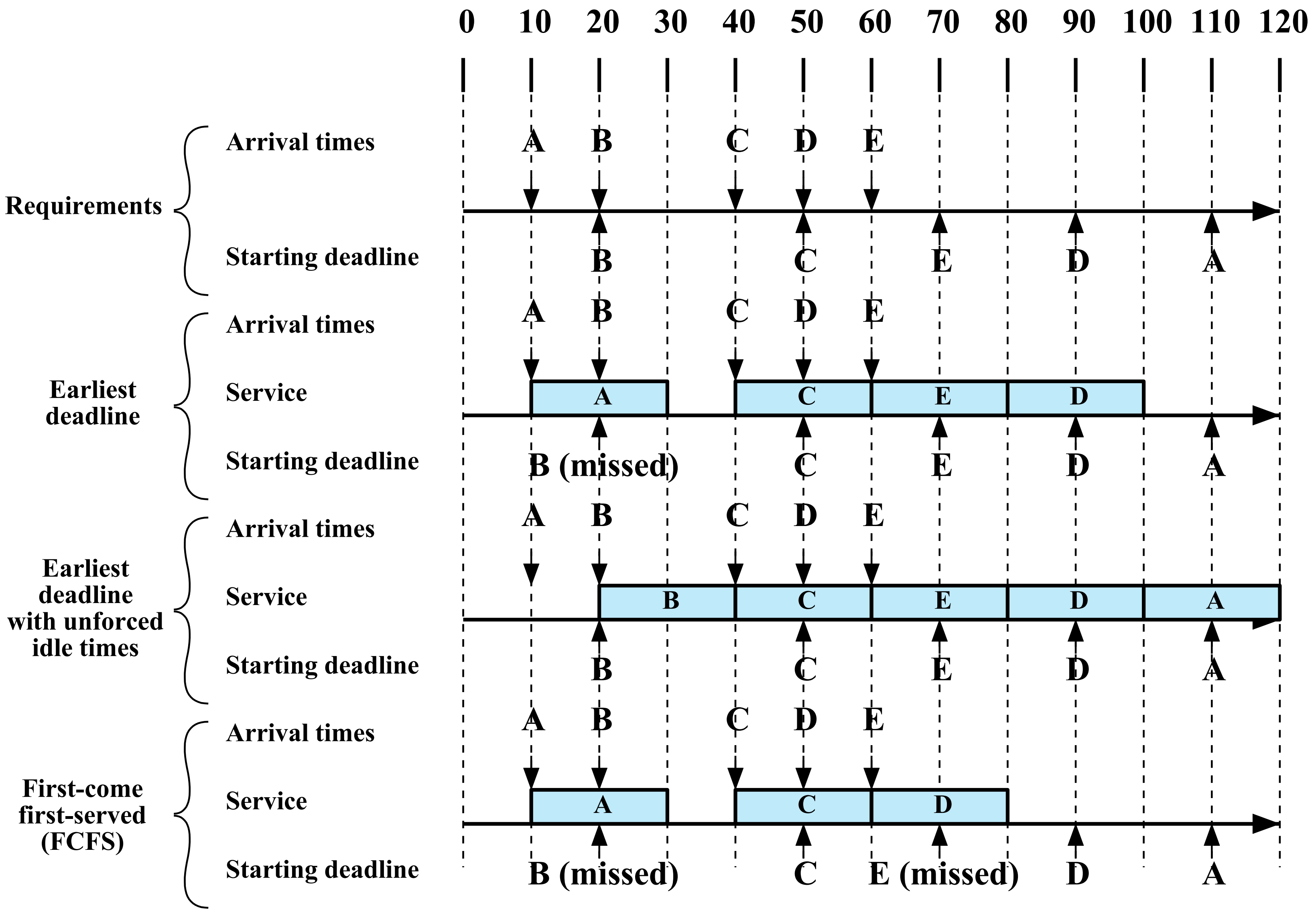


Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

Real-time Scheduling Algorithms

- **Rate Monotonic Scheduling**
  - Static Priority-driven scheduling
  - Priorities are assigned based on the period of each task
    - The shorter the period, the higher the priority
  - Task P has a period of T then rate of the task P is 1/T
  - If C is the execution time for task P then
  - $U = \frac{C}{T}$ is called processor utilization for task P
  - For RMS the following inequality holds

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

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

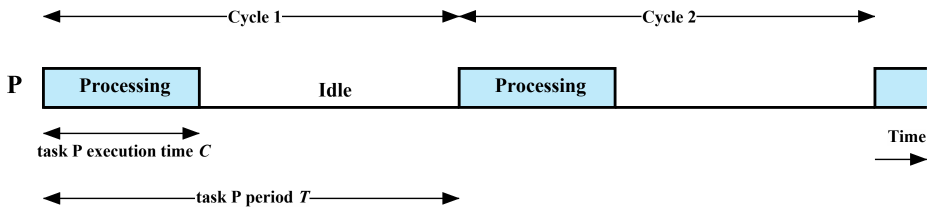


Figure 10.8 Periodic Task Timing Diagram

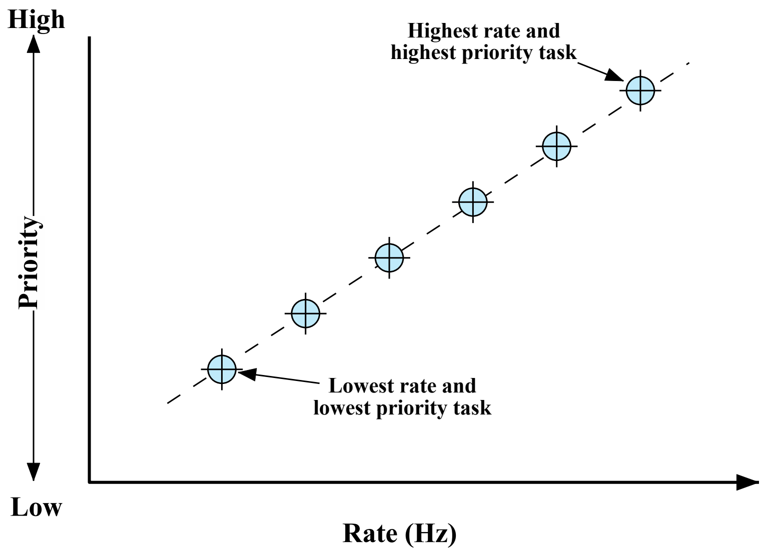


Figure 10.7 A Task Set with RMS

Real-time Scheduling Algorithms

- For RMS the following inequality holds

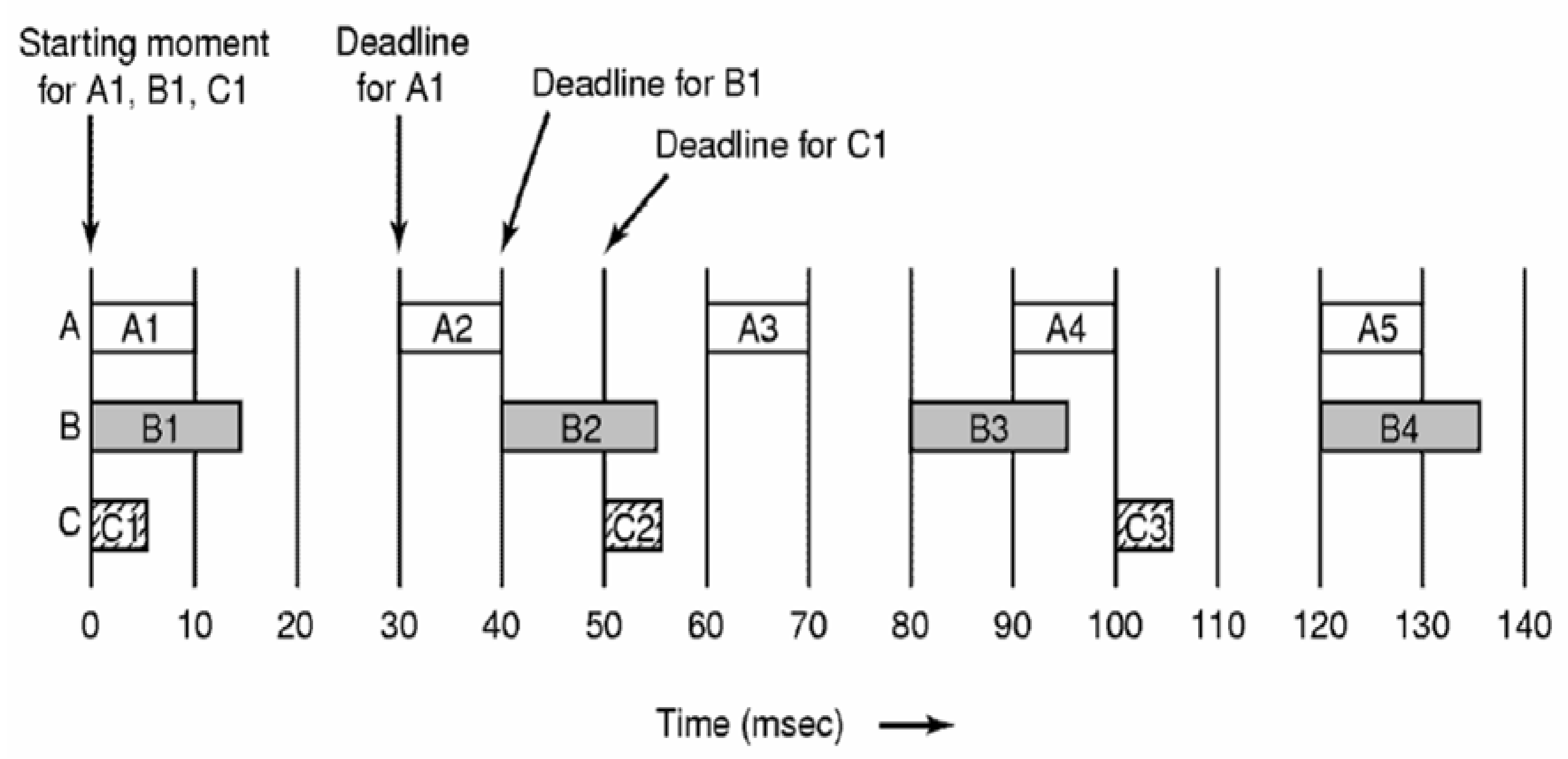
$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

- Example:
  - Task P1 : C1 =20, T1=100; U1 = 0.2
  - Task P2 : C2 =40, T2=150; U2 = 0.267
  - Task P3 : C3 =100, T3=350; U3 = 0.286
  - $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} = 0.753 < 0.779$
  - If RMS is used these tasks will be successfully scheduled.

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

Example

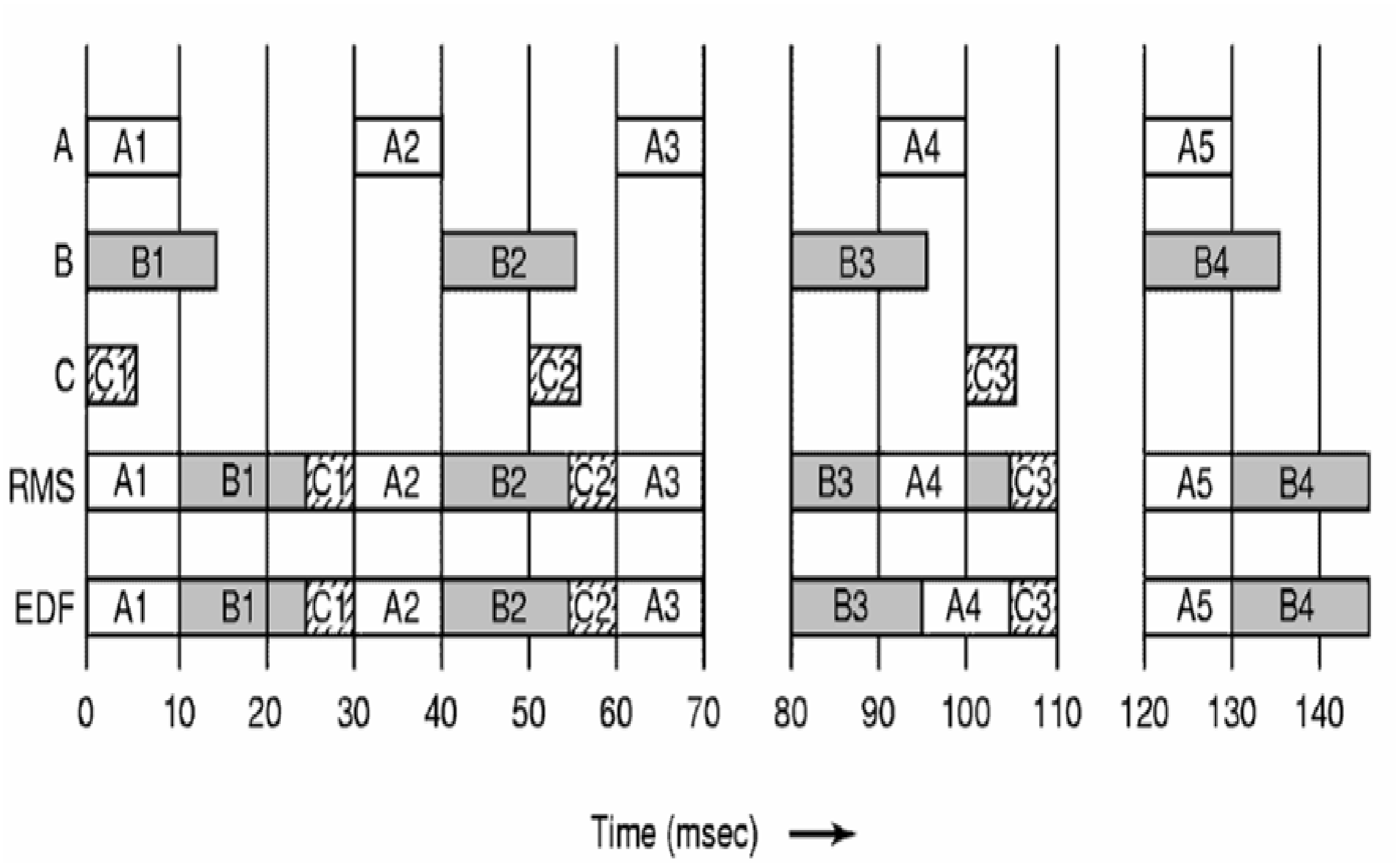
Task A : C1 = 10,   T1= 30;
Task B : C2 = 15,   T2= 40;
Task C : C3 = 05,   T3= 50;



Task A : C1 = 10, T1= 30;
Task B : C2 = 15, T2= 40;
Task C : C3 = 05, T3= 50;

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

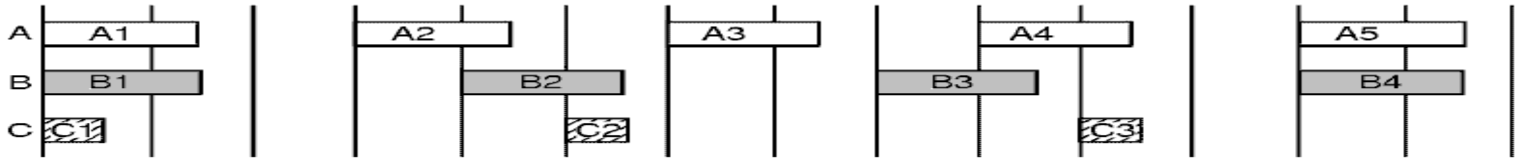
$$\frac{10}{30} + \frac{15}{40} + \frac{5}{50} = 0.808$$

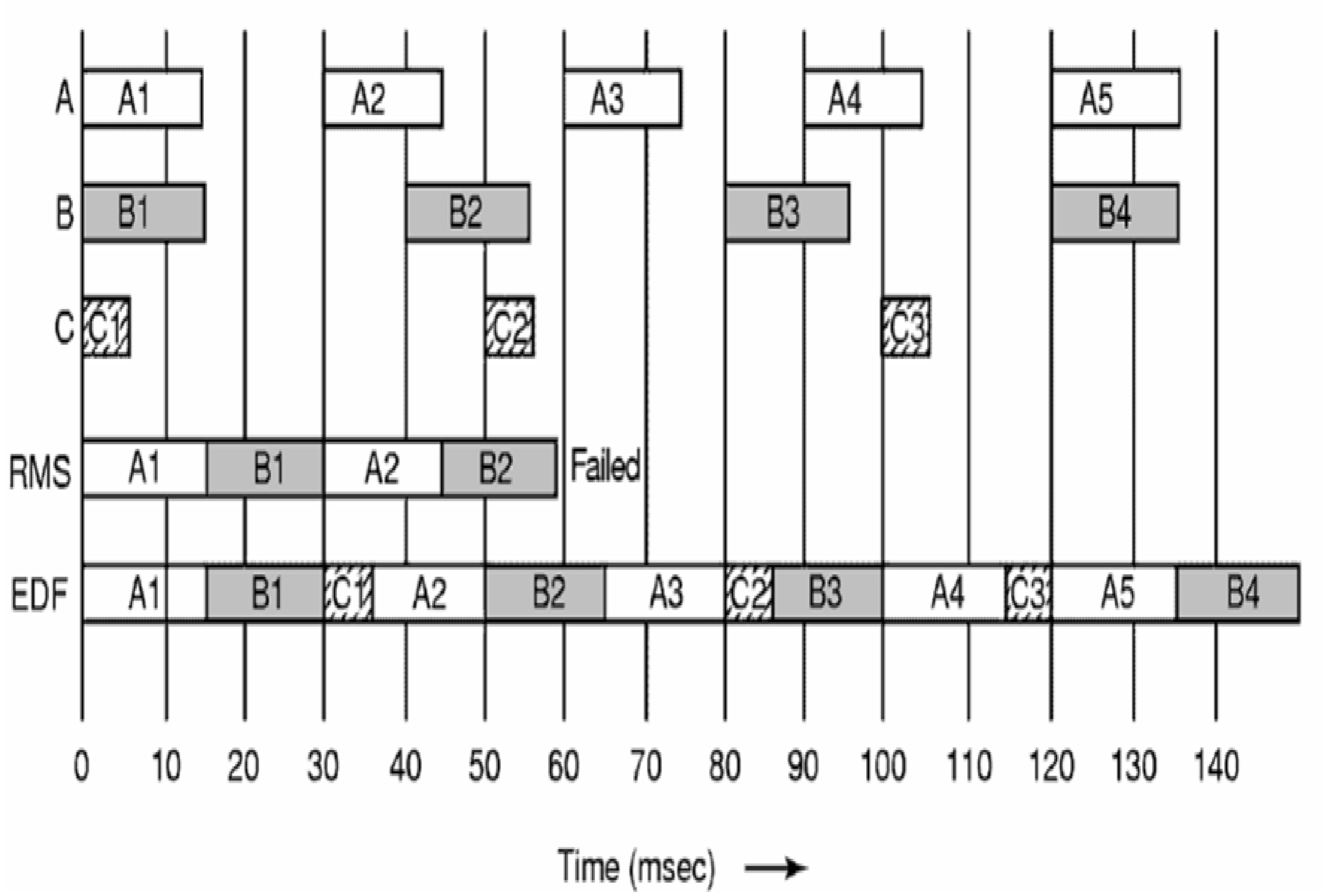


Let’s Modify the Example Slightly

- Increase A’s CPU requirement to 15 msec
- The system is still schedulable

$$\frac{15}{30} + \frac{15}{40} + \frac{5}{50} = 0.975$$




A
A1
A2
A3
A4
A5
B
B1
B2
B3
B4
C
C1
C2
C3
RMS
A1
B1
A2
B2
Failed
EDF
A1
B1
C1
A2
B2
A3
C2
B3
A4
C3
A5
B4
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
Time (msec)

RMS failed, why?

- It has been proven that RMS is only guaranteed to work if the CPU utilisation is not too high
  - For three tasks, CPU utilisation must be less than 0.780
- We were lucky with our original example

RMS vs EDFS

- For RMS: $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n\left(2^{\frac{1}{n}} - 1\right)$ For EDFS: $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 1$
  - EDFS can achieve greater overall processor utilization
- Still RMS is widely adopted for use in industrial applications:
  - The performance difference is small in practice. The upper bound for RMS is a conservative one and in practice upto 90% utilization is often achieved.
  - Most hard-RT system also have soft-RT components which are not used with RMS scheduling of hard-RT tasks
  - Stability is easier to achieve with RMS.

Priority Inversion

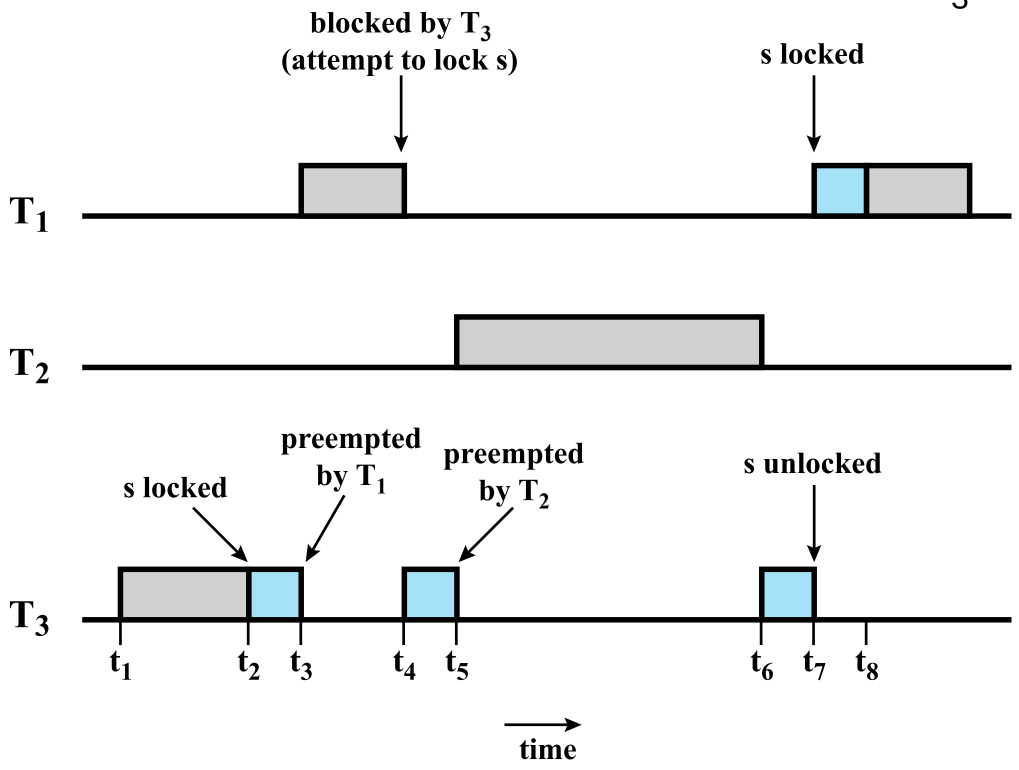
- Can occur in any priority-based preemptive scheduling scheme
- Particularly relevant in the context of real-time scheduling
- Best-known instance involved the Mars Pathfinder mission
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task

Unbounded Priority Inversion

- the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks

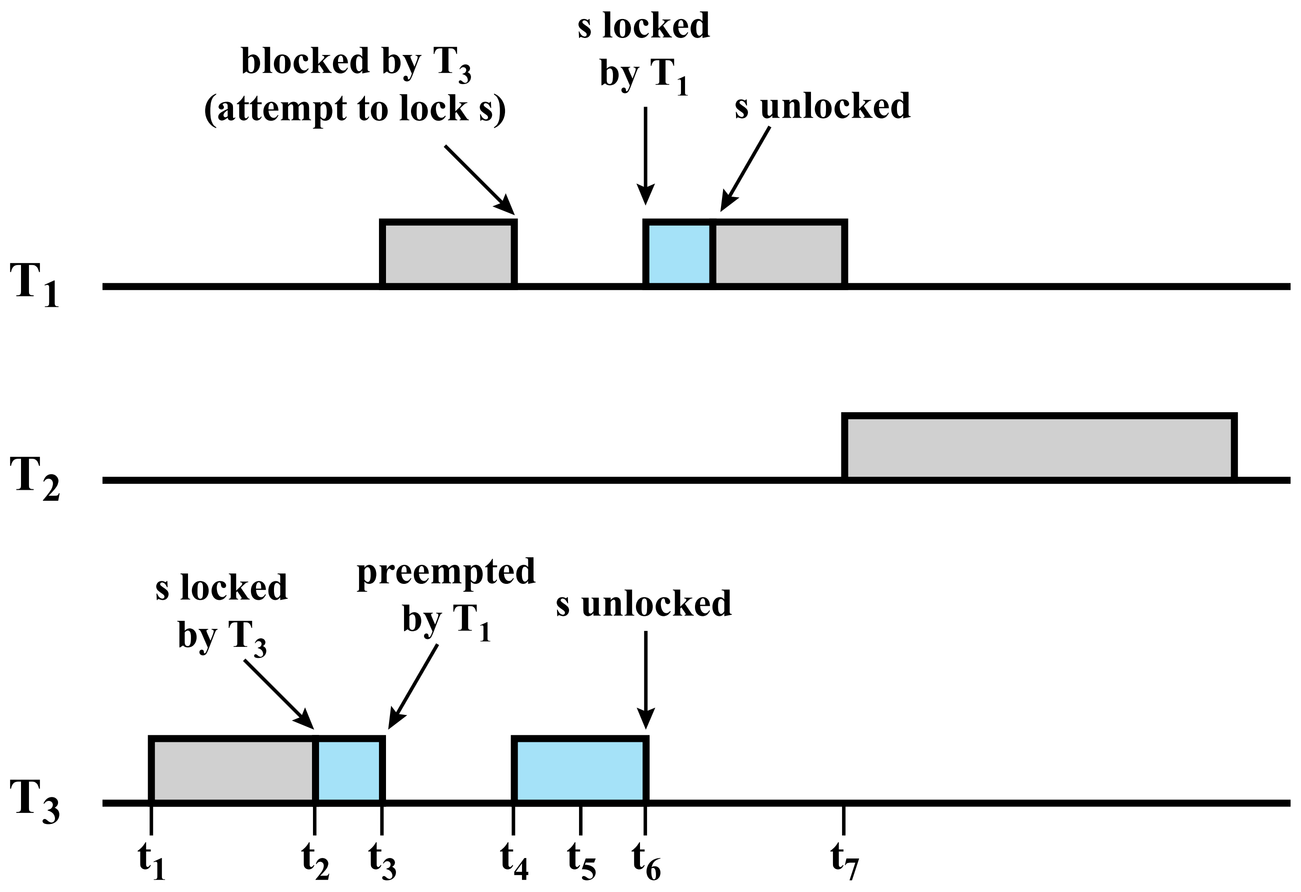
Unbounded Priority Inversion

T_1: periodic system health check
T_2: Process image data
T_3: Occasional test on status


blocked by T_3 (attempt to lock s)
s locked
T_1
T_2
s locked
preempted by T_1
preempted by T_2
s unlocked
T_3
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
time

(a) Unbounded priority inversion

Priority Inheritance



(b) Use of priority inheritance

Priority Ceiling

- A priority is associated with each resource.
  - The assigned priority is one level higher than the priority of its highest-priority user.
- Scheduler dynamically assigns the resource’s priority to any process that access the resource
  - After the process is done with the resource the process priority returns to normal

Summary

- RT processes are executed in connection with some process or event external to the computer system
- Therefore, RT scheduling must meet one or more deadlines
- Traditional criteria for scheduling does not apply
- RT systems are characterized by
  - Determinism, Responsiveness, User Control, Reliability, Fail-soft operation
- Both static and dynamic scheduling is possible in RT system
  - Static
    - Table drive approach VS Priority driven preemptive approach
  - Dynamic
    - Planning base approach VS Best effort approach

Summary

- Key factor is meeting the deadlines not processor utilization or etc.
- Algorithms that rely heavily on preemption and on reacting to relative deadlines are appropriate for this purpose
- Two prominent scheduling algorithms are
  - Earliest Deadline First (EDF)
  - Rate Monitoring Scheduling (RMS)

References

- **Operating Systems – Internal and Design Principles**
  - By William Stallings
- Chapter 10