Sharif University of Technology
Department of Computer Science and Engineering

Lec. 2:
Basic Concepts

Real-Time Computing

S. Safari
Fall 2023

Introduction

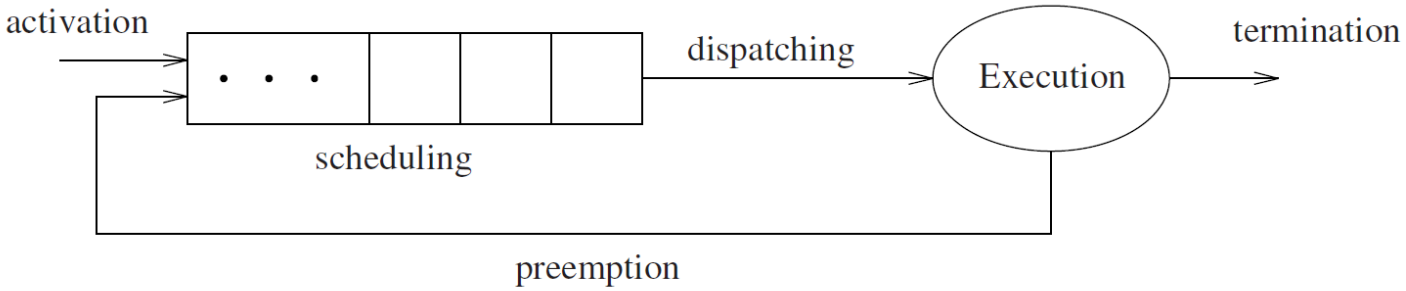
- A **process** is a computation that is executed by the CPU in a sequential fashion.
  - In this course, the term ***process*** is used as synonym of *task* and *thread*.
- Concurrent tasks means tasks that can overlap in time.
  - When a single processor has to execute a set of concurrent tasks the CPU has to be assigned to the various tasks according to a predefined criterion, called a ***scheduling policy***.
  - The set of rules that, at any time, determines the order in which tasks are executed is called a ***scheduling algorithm***.
  - The specific operation of allocating the CPU to a task selected by the scheduling algorithm is referred as ***dispatching***.

Introduction (2)

- A task that could potentially execute on the CPU can be:
  - In execution (if it has been selected by the scheduling algorithm).
  - Waiting for the CPU (if another task is executing).
- **Active task**: A task that can potentially execute on the processor, independently on its actual availability.
  - **Ready task**: A task waiting for the processor.
  - **Running task**: The task in execution.

Introduction (3)

- All ready tasks waiting for the processor are kept in a queue, called ready queue.
  - Operating systems that handle different types of tasks may have more than one ready queue.
  - The operation of suspending the running task and inserting it into the ready queue is called ***preemption***.

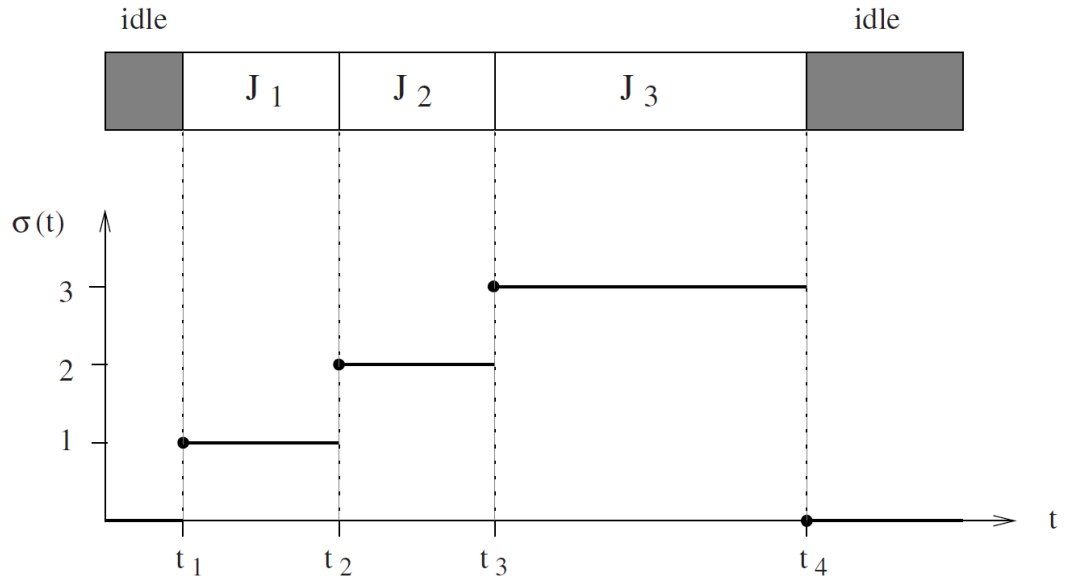


Preemption

- In dynamic real-time systems, preemption is important for three reasons [SZ92]:
  - Tasks performing exception handling may need to preempt existing tasks so that responses to exceptions may be issued in a timely fashion.
  - When tasks have different levels of criticality (expressing task importance), preemption permits executing the most critical tasks, as soon as they arrive.
  - Preemptive scheduling typically allows higher efficiency, in the sense that it allows executing real-time task sets with higher processor utilization.
- Preemption destroys program locality and introduces a runtime overhead that inflates the execution time of tasks.

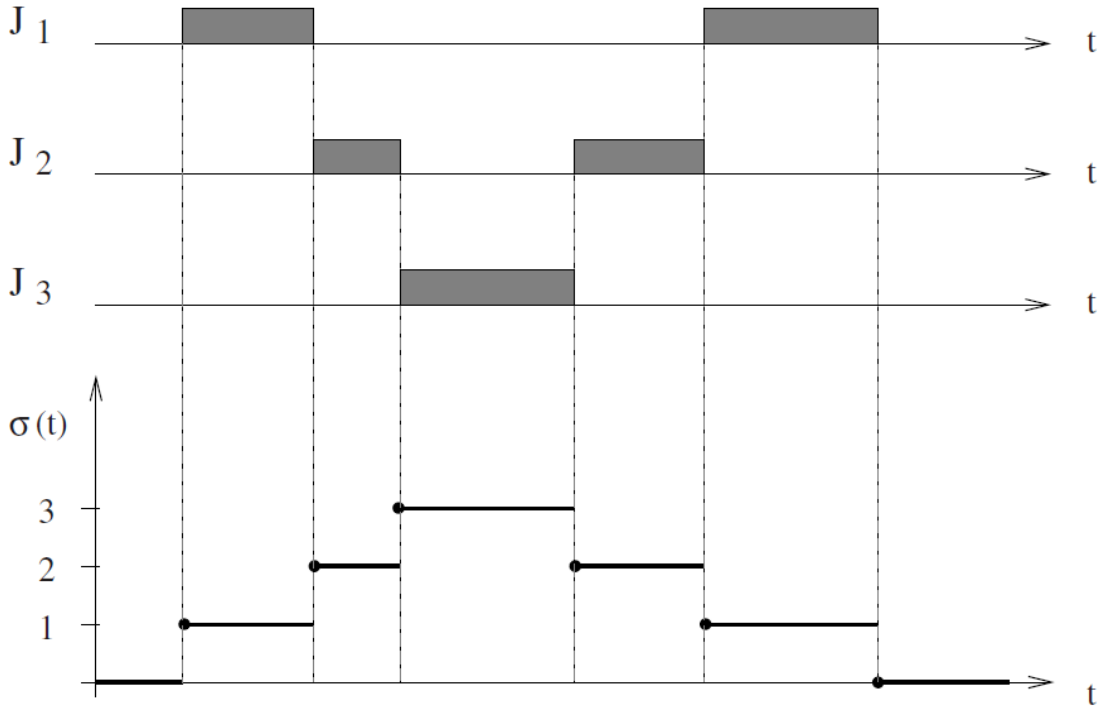
An Example of Schedule

- An example of schedule obtained by executing three tasks: J_1, J_2, J_3.
- At times t_1, t_2, t_3, and t_4, the processor performs a context switch.
- Each interval $[t_i, t_{i+1})$ in which σ(t) is constant is called ***time slice***. Interval $[x, y)$ identifies all values of t such that $x \leq t < y$.



An Example of A Preemptive Schedule

- A preemptive schedule is a schedule in which the running task can be arbitrarily suspended at any time, to assign the CPU to another task according to a predefined scheduling policy.



Feasible Schedule and Schedulable Tasks

- In preemptive schedules, tasks may be executed in disjointed interval of times.
- A schedule is said to be ***feasible*** if all tasks can be completed according to a set of specified constraints.
- A set of tasks is said to be ***schedulable*** if there exists at least one algorithm that can produce a feasible schedule.

Types of Task Constraints

- Typical constraints that can be specified on real-time tasks are of three classes:
  - Timing constraints
  - Precedence relations
  - Mutual exclusion constraints on shared resources

Timing Constraints

- A typical timing constraint on a task is the ***deadline***.
  - Represents the time before which a process should complete its execution without causing any damage to the system.
  - ***Relative deadline***: If a deadline is specified with respect to the task arrival time.
  - ***Absolute deadline***: if it is specified with respect to time zero.

Categories of Real-Time Tasks

- Depending on the consequences of a missed deadline, real-time tasks are usually distinguished in three categories:
  - **Hard**: A real-time task is said to be hard if missing its deadline may cause catastrophic consequences on the system under control.
  - **Firm**: A real-time task is said to be firm if missing its deadline does not cause any damage to the system, but the output has no value.
  - **Soft**: A real-time task is said to be soft if missing its deadline has still some utility for the system, although causing a performance degradation.

Characterizations of A Real-Time Task

- **Arrival time** a_i: The time at which a task becomes ready for execution; it is also referred as *request time* or *release time* and indicated by r_i.
- **Computation time** C_i: The time necessary to the processor for executing the task without interruption.
- **Absolute Deadline** d_i: The time before which a task should be completed to avoid damage to the system.
- **Relative Deadline** D_i: The difference between the absolute deadline and the request time: $D_i = d_i - r_i$.

Characterizations of A Real-Time Task (2)

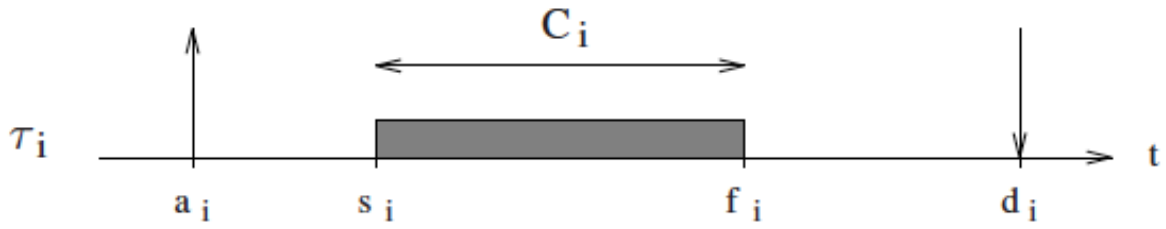
- **Start time s_i**: The time at which a task starts its execution.
- **Finishing time f_i**: The time at which a task finishes its execution.
- **Response time R_i**: The difference between the finishing time and the request time: $R_i = f_i - r_i$.
- **Criticality** is a parameter related to the consequences of missing the deadline (typically, it can be hard, firm, or soft).
- **Value v_i** represents the relative importance of the task with respect to the other tasks in the system.

Characterizations of A Real-Time Task (3)

- **Lateness L_i:** $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline.
  - If a task completes before the deadline, its lateness is negative.
- **Tardiness** or *Exceeding time* E_i: $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- **Laxity** or *Slack time* X_i: $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

Characterizations of A Real-Time Task (4)

- Some of the parameters defined in the previous slides are illustrated in the following figure.

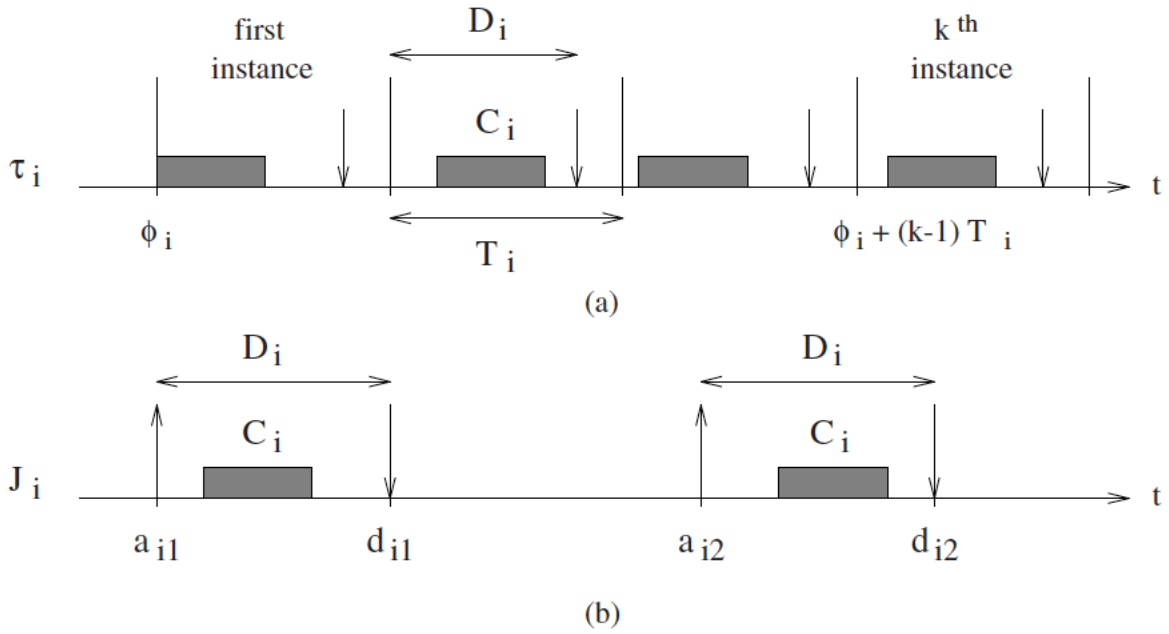


The Regularity of Activation

- Tasks can be defined as *periodic* or *aperiodic*.
- **Periodic tasks** consist of an infinite sequence of identical activities, called *instances* or *jobs*, that are regularly activated at a constant rate.
- **Aperiodic tasks** also consist of an infinite sequence of identical jobs (or instances);
  - Their activations are not regularly interleaved.
- An aperiodic task where consecutive jobs are separated by a minimum inter-arrival time is called a *sporadic task*.

The Regularity of Activation (2)

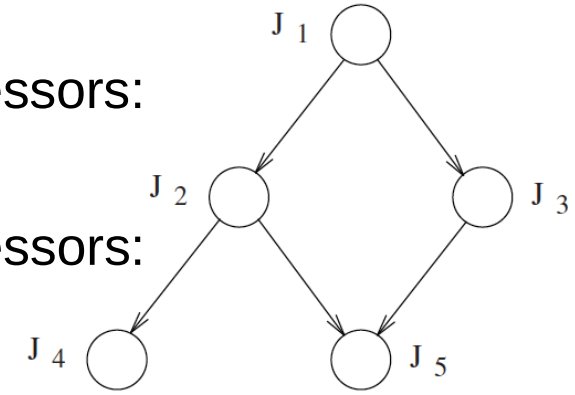
- A periodic task τ_i; an aperiodic job J_i; the generic k^{th} job of a periodic task $\tau_{i,k}$.
- The activation time of the first periodic instance ($\tau_{i,1}$) is called *phase*.
  - The activation time of the k^{th} instance is given by $\varphi_i+(k-1)T_i$.
  - Note: T_i is the activation period of the task. A periodic process can be completely characterized by its phase φ_i, its computation time C_i, its period T_i, and its relative deadline D_i.



Precedence Constraints

- Precedence relations are usually described through a directed acyclic graph G.
  - Tasks are represented by nodes and precedence relations by arrows
- The notation $J_a \prec J_b$ specifies that task J_a is a *predecessor* of task J_b, meaning that G contains a directed path from J_a to J_b.
- The notation $J_a \rightarrow J_b$ specifies that task J_a is an *immediate predecessor* of J_b, meaning that G contains an arc directed from node J_a to node J_b.

- Tasks with no predecessors: *beginning tasks*;
- Tasks with no successors: *ending tasks*



$J_1 \prec J_2$

$J_1 \rightarrow J_2$

$J_1 \prec J_4$

$J_1 \not\rightarrow J_4$

Resource Constraints

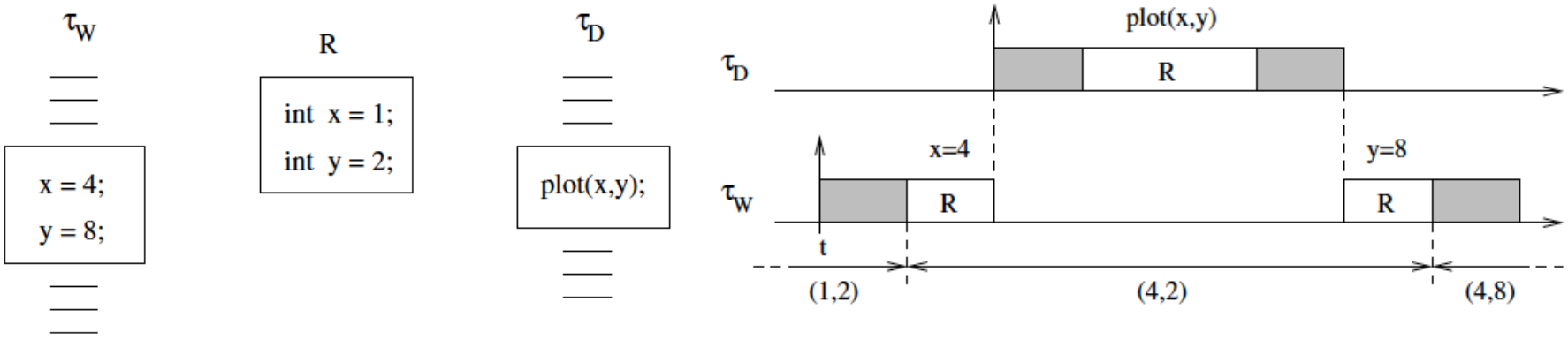
- From a process point of view, a *resource* is any software structure that can be used by the process to advance its execution.
- A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource*.
- Some resources:
  - A data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device.

A Mutually Exclusive Resource

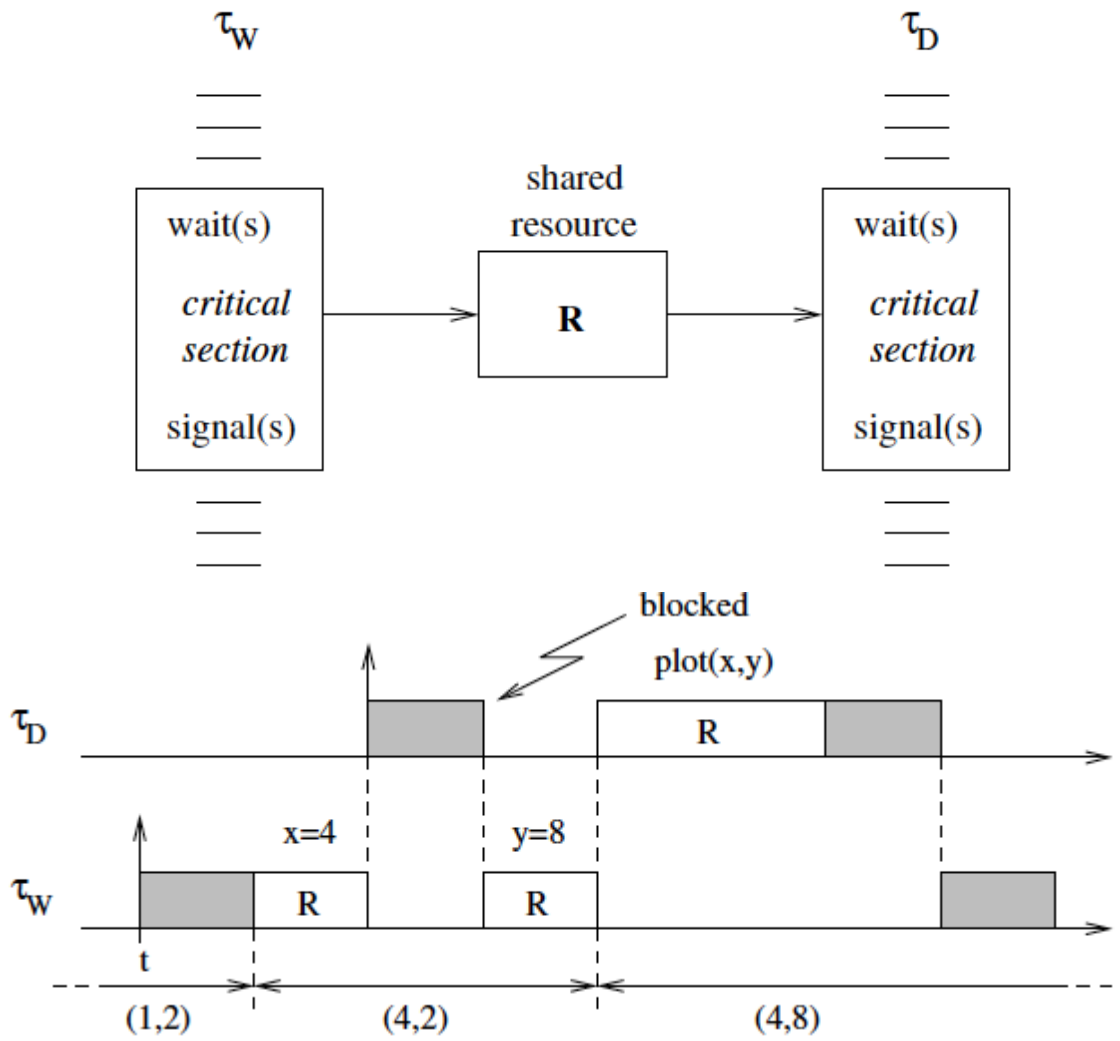
- Many shared resources do not allow simultaneous accesses by competing tasks, but require their mutual exclusion.
  - This means that a task cannot access a resource *R* if another task is inside *R* manipulating its data structures.
  - In this case, *R* is called a *mutually exclusive resource*.
  - A piece of code executed under mutual exclusion constraints is called a *critical section*.

An Example of Mutual Exclusion

- Two tasks cooperate to track a moving object: task τ_W gets the object coordinates from a sensor and writes them into a shared buffer R, containing two variables (x, y); task τ_D reads the variables from the buffer and plots a point on the screen to display the object trajectory.

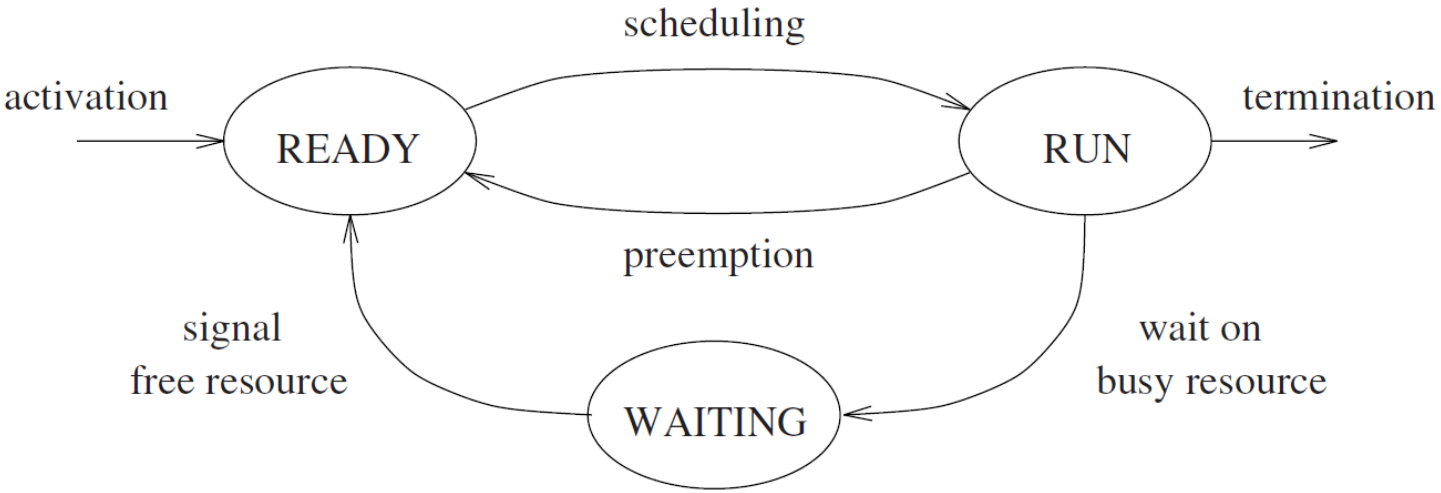


Solution: Semaphores



Waiting State

- When a running task executes a wait primitive on a locked semaphore, it enters a waiting state, until another task executes a signal primitive that unlocks the semaphore.
  - Note that when a task leaves the waiting state, it does not go in the running state, but in the ready state, so that the CPU can be assigned to the highest-priority task by the scheduling algorithm.



Definition of Scheduling Problems

- A set of n tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$,
- A set of m processors $P = \{P_1, P_2, \ldots, P_m\}$
- A set of s types of resources $R = \{R_1, R_2, \ldots, R_s\}$
- Precedence relations among tasks can be specified through a directed acyclic graph, and timing constraints can be associated with each task.
- **Scheduling** means assigning processors from P and resources from R to tasks from Γ in order to complete all tasks under the specified constraints.
- This problem, in its general form, has been shown to be NP-complete [GJ79] and hence computationally intractable.

Time Complexity of Online Scheduling Problems

- The complexity of scheduling algorithms is of high relevance in dynamic real-time systems.
  - Scheduling decisions must be taken online during task execution.
- A *polynomial algorithm* is one whose time complexity grows as a polynomial function *p* of the input length *n* of an instance.
  - The complexity of such algorithms is denoted by $O(p(n))$.
  - Each algorithm whose complexity function cannot be bounded in that way is called an *exponential time algorithm*.
  - The online scheduling problems must be solved in polynomial time.

Classification of Scheduling Algorithms

- **Preemptive vs. Non-preemptive**
  - In preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
  - In non-preemptive algorithms, a task, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as the task terminates its execution.
- **Static vs. Dynamic**
  - Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
  - Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.

Classification of Scheduling Algorithms (2)

- **Off-line vs. Online**
  - A scheduling algorithm is used off line if it is executed on the entire task set before tasks activation. The schedule generated in this way is stored in a table and later executed by a dispatcher.
  - A scheduling algorithm is used online if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.
- **Optimal vs. Heuristic**
  - An algorithm is optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then an algorithm is optimal if it is able to find a feasible schedule, if one exists.
  - An algorithm is heuristic if it is guided by a heuristic function in taking its scheduling decisions. A heuristic algorithm tends toward the optimal schedule, but does not guarantee finding it.

Guarantee-Based Algorithms

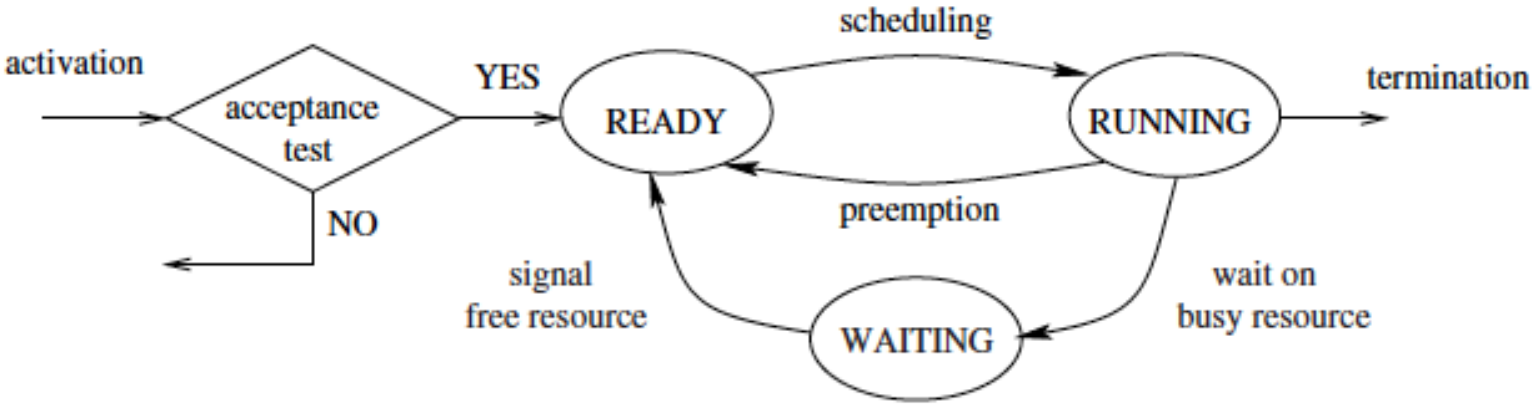
- In hard real-time applications that require highly predictable behavior, the feasibility of the schedule should be guaranteed in advance.
  - Before task execution
- Checking the feasibility of the schedule before tasks' execution
- Assuming a worst-case scenario
- In static real-time systems, where the task set is fixed and known a priori, all task activations can be precalculated offline, and the entire schedule can be stored in a table that contains all guaranteed tasks arranged in the proper order.

Guarantee-Based Algorithms (2)

- At runtime, a dispatcher simply removes the next task from the table and puts it in the running state.
- The main advantage of the static approach is that the runtime overhead does not depend on the complexity of the scheduling algorithm.
- This allows very sophisticated algorithms to be used to solve complex problems or find optimal scheduling sequences.
- On the other hand, however, the resulting system is quite inflexible to environmental changes; thus, predictability strongly relies on the observance of the hypotheses made on the environment.

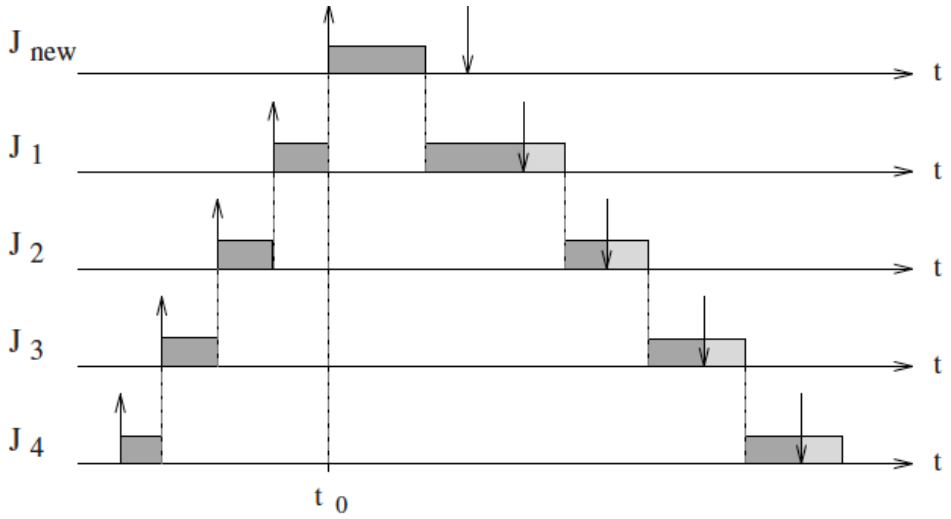
Guarantee-Based Algorithms (3)

- Dynamic real-time systems (typically consisting of firm tasks)
  - Tasks can be created at runtime
  - The guarantee must be done *online* every time a new task is created.



Domino Effect

- The benefit of having a guarantee mechanism is that potential overload situations can be detected in advance to avoid negative effects on the system.
- One of the most dangerous phenomena caused by a transient overload is called *domino effect*.
  - It refers to the situation in which the arrival of a new task causes all previously guaranteed tasks to miss their deadlines.



Best-Effort Algorithms

- In certain real-time applications, computational activities have soft timing constraints.
  - Missing soft deadlines do not cause catastrophic consequences, but only a performance degradation.
- For example: Typical multimedia applications
  - The objective: Handling different types of information (such as text, graphics, images, and sound) in order to achieve a certain quality of service for the users
  - Missing a deadline may only affect the performance of the system.
- To efficiently support soft real-time applications, a *best-effort* approach may be adopted for scheduling.
  - There is no guarantee of finding a feasible schedule.
  - Best-effort algorithms perform much better than guarantee-based schemes in the average case.

Metrics for Performance Evaluation

- The performance of scheduling algorithms is typically evaluated through a cost function defined over the task set.
  - For example, classical scheduling algorithms try to:
    - Minimize the average response time, the total completion time, the weighted sum of completion times, or the maximum lateness.

Average response time:

$$\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$$

Total completion time:

$$t_c = \max_i(f_i) - \min_i(a_i)$$

Weighted sum of completion times:

$$t_w = \sum_{i=1}^{n} w_i f_i$$

Maximum lateness:

$$L_{max} = \max_i(f_i - d_i)$$

Maximum number of late tasks:

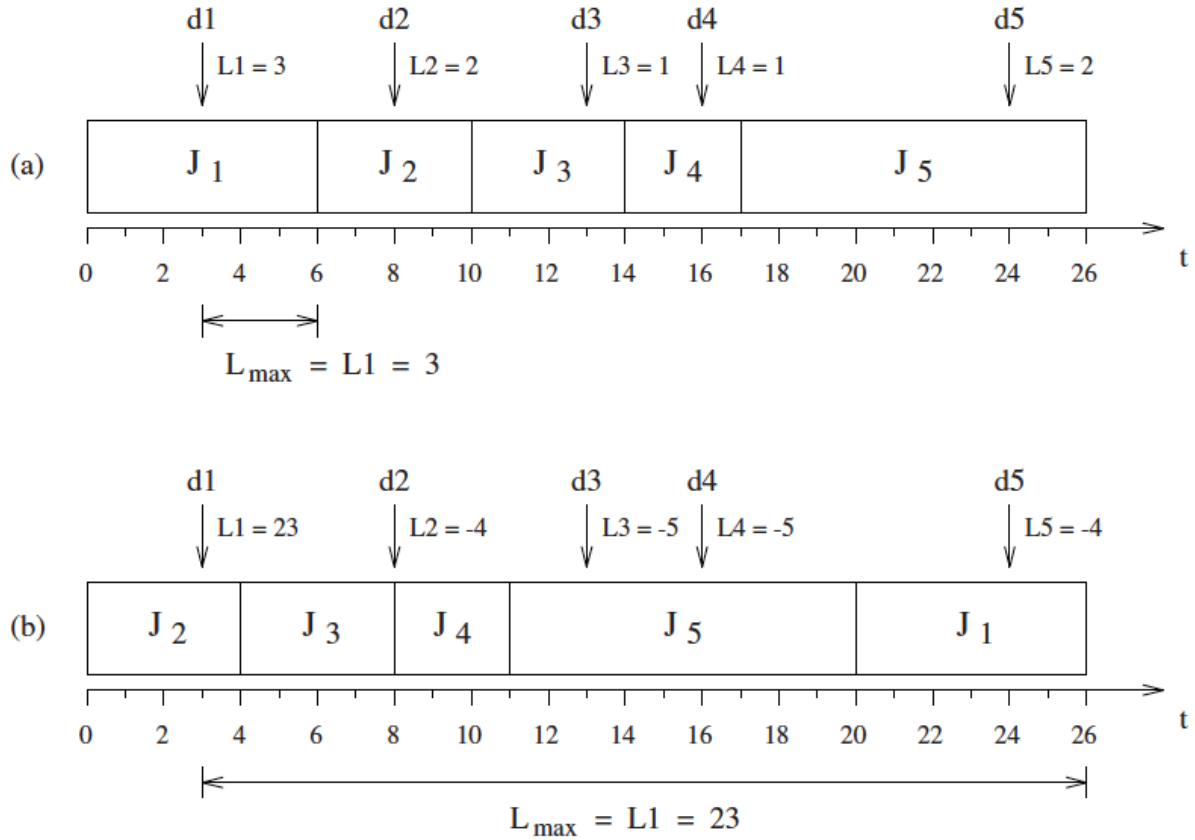
$$N_{late} = \sum_{i=1}^{n} miss(f_i)$$

where

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

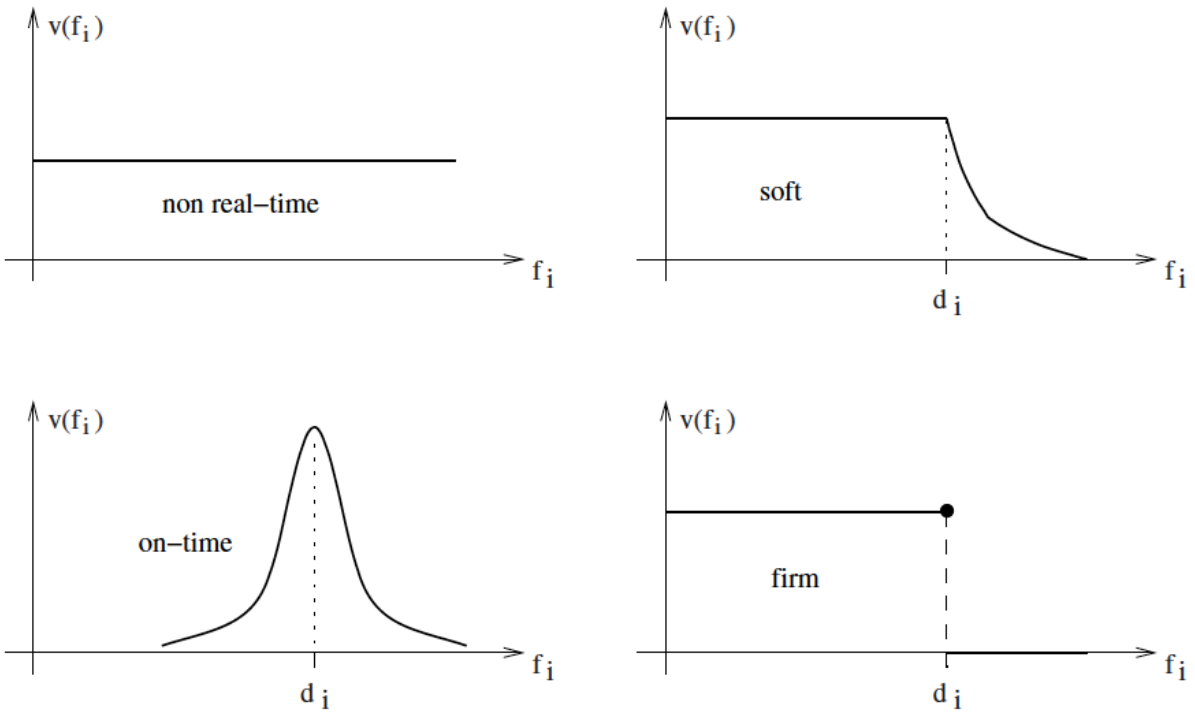
Example

- The schedule minimizes the maximum lateness, but all tasks miss their deadline. The schedule shown in Figure b has a greater maximum lateness, but four tasks out of five complete before their deadline.



Utility Functions

- Utility function: The value associated with the task as a function of its completion time.
  - The benefit of executing a task may depend not only on the task importance but also on the time at which it is completed



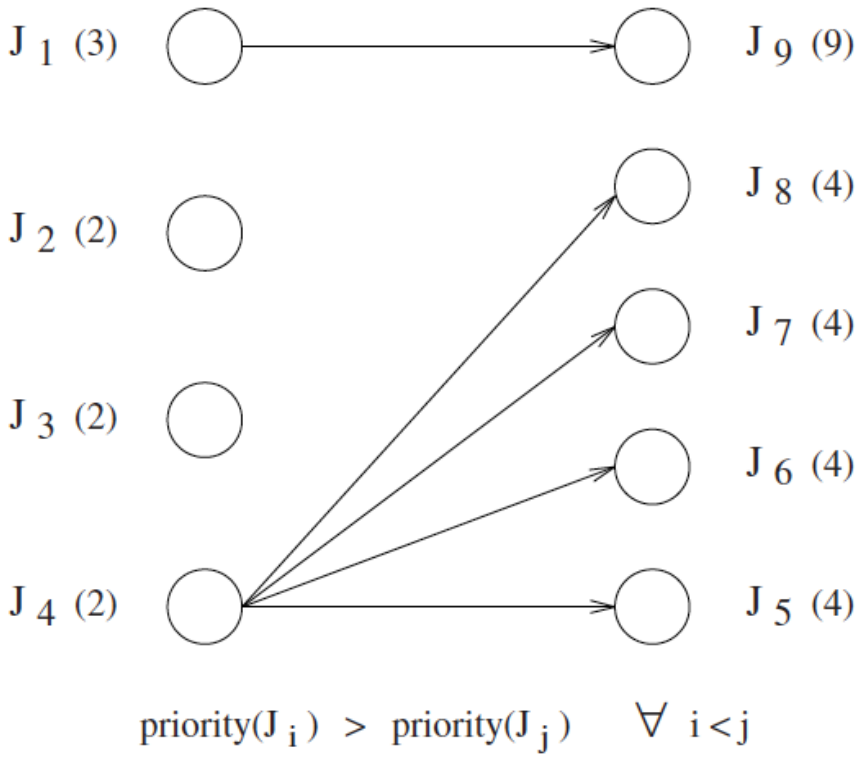
Scheduling Anomalies

- **Real-time computing** is not equivalent to **fast computing**.
- Richard's anomalies were described by Graham in 1976 and refer to task sets with precedence relations executed in a multiprocessor environment.

Theorem (Graham, 1976). *If a task set is optimally scheduled on a multiprocessor with some priority assignment, a fixed number of processors, fixed execution times, and precedence constraints, then increasing the number of processors, reducing execution times, or weakening the precedence constraints can increase the schedule length.*

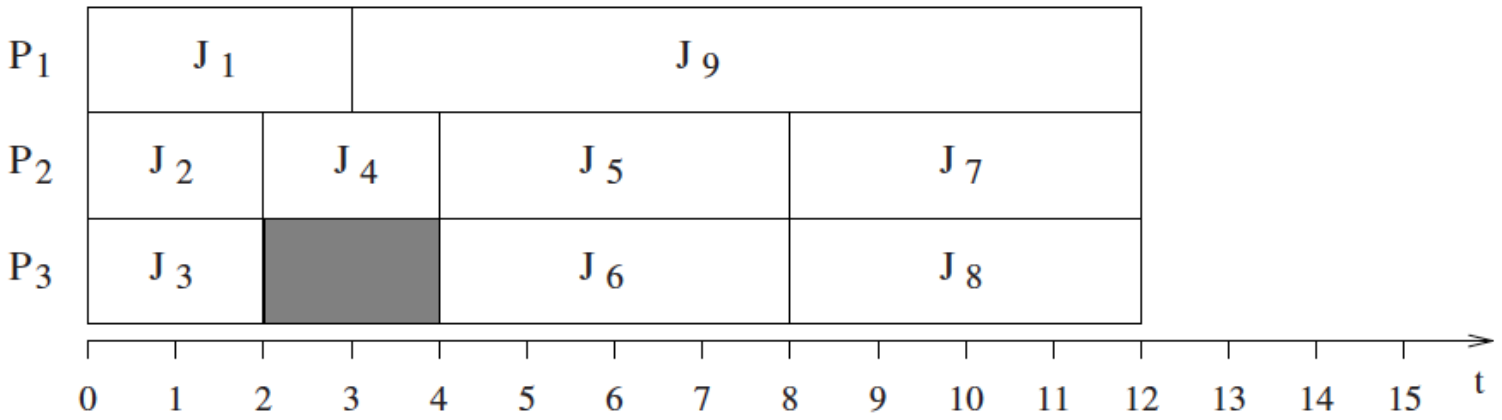
An Example of Graham's Theorem

- Let us consider a task set consisting of nine jobs $J = \{J_1, J_2, \ldots, J_9\}$, sorted by decreasing priorities, so that J_i priority is greater than J_j priority if and only if $i < j$.



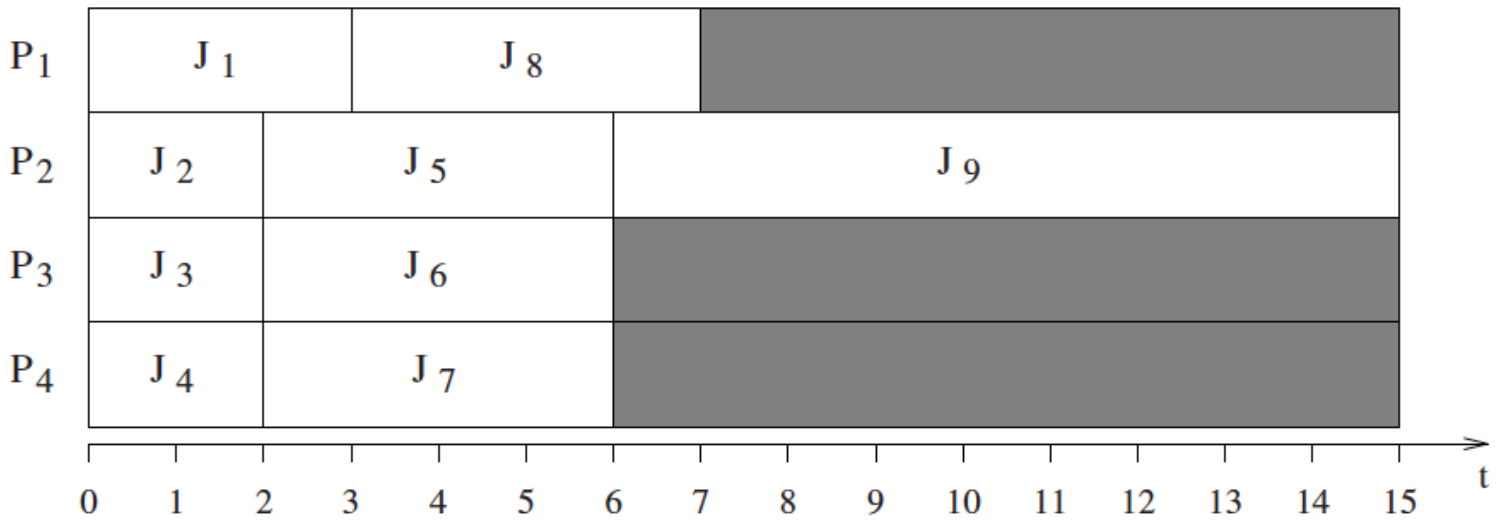
An Example of Graham's Theorem (2)

- If this task set is executed on a platform with three processing cores, where the highest priority task is assigned to the first available core, the resulting schedule σ^* is illustrated in following figure, where the global completion time is t_c = 12 units of time.



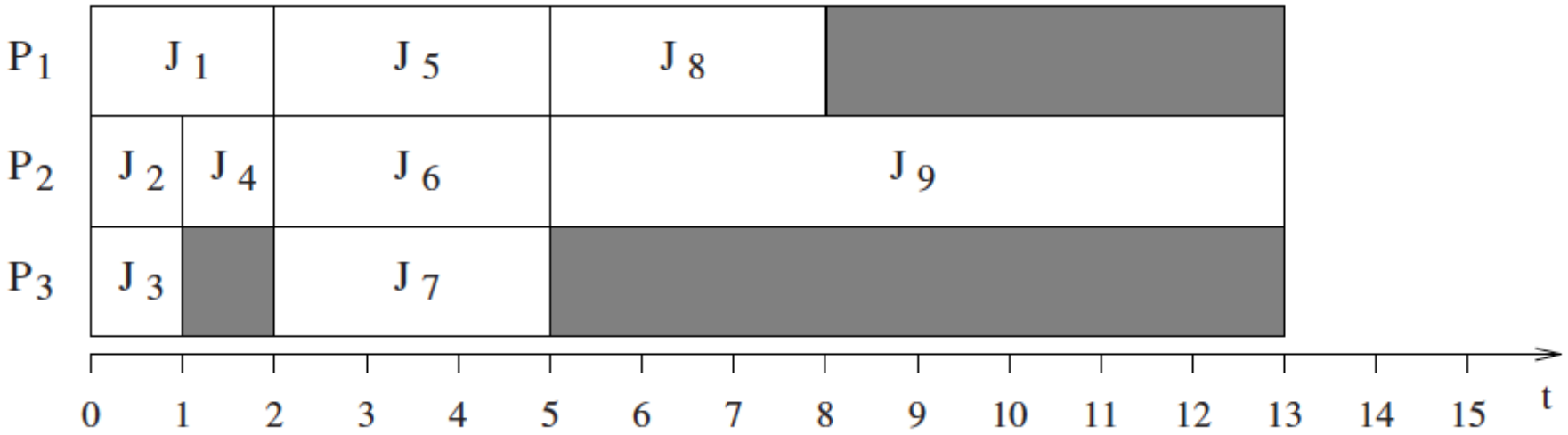
An Example of Graham's Theorem: Number of Processing cores increased

- Considering four processing cores
- Obtaining the schedule which is characterized by a global completion time of t_c = 15 units of time.



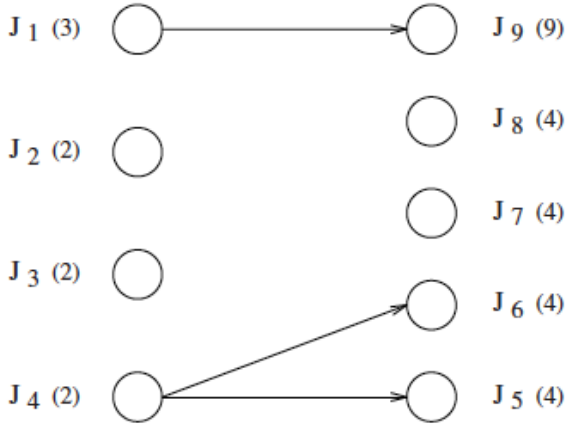
An Example of Graham's Theorem: Computation Times Reduced

- One could think that the global completion time of the task set J could be improved by reducing tasks' computation times of each task. However, reducing the computation time of each task by one unit of time, the schedule length will increase with respect to the optimal schedule σ^*, and the global completion time will be $t_c = 13$.

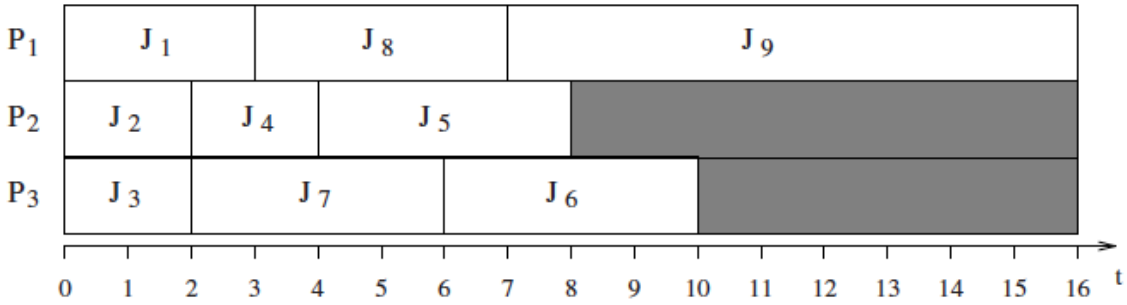


An Example of Graham's Theorem: Precedence Constraints Weakened

- Removing the precedence relations between job J_4 and jobs J_7 and J_8
  - Obtaining the schedule which is characterized by a global completion time of t_c = 16 units of time.



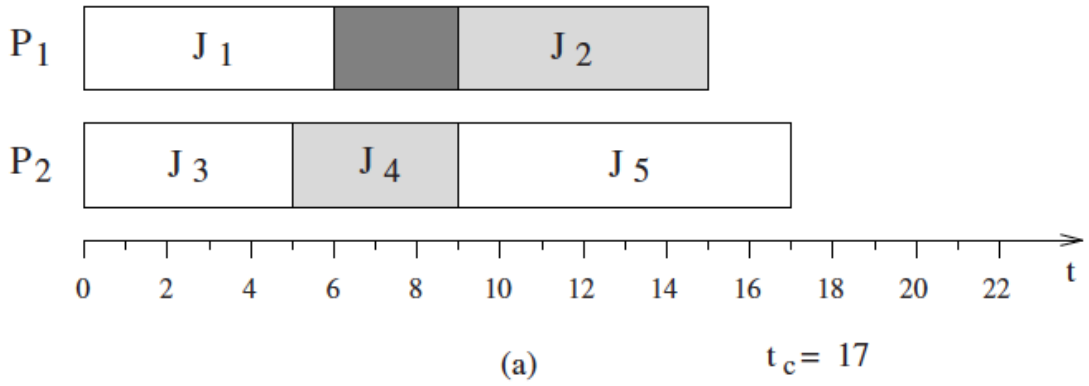
(a)



(b)

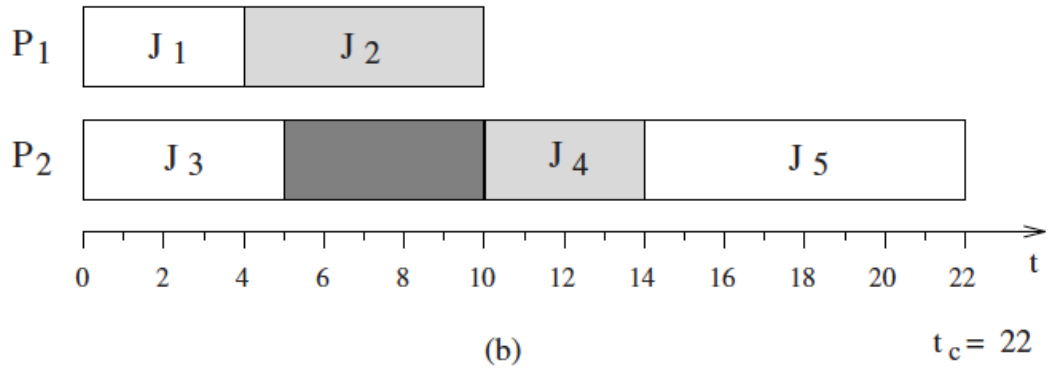
An Example of Graham's Theorem: Anomalies under Resource Constraints

- Five jobs are statically allocated on two processors:
  - Jobs J_1 and J_2 on processor P_1, and jobs J_3, J_4 and J_5 on processor P_2 (jobs are indexed by decreasing priority).
  - Jobs J_2 and J_4 share the same resource in exclusive mode; hence their execution cannot overlap in time.
- The total completion time is $t_c = 17$.



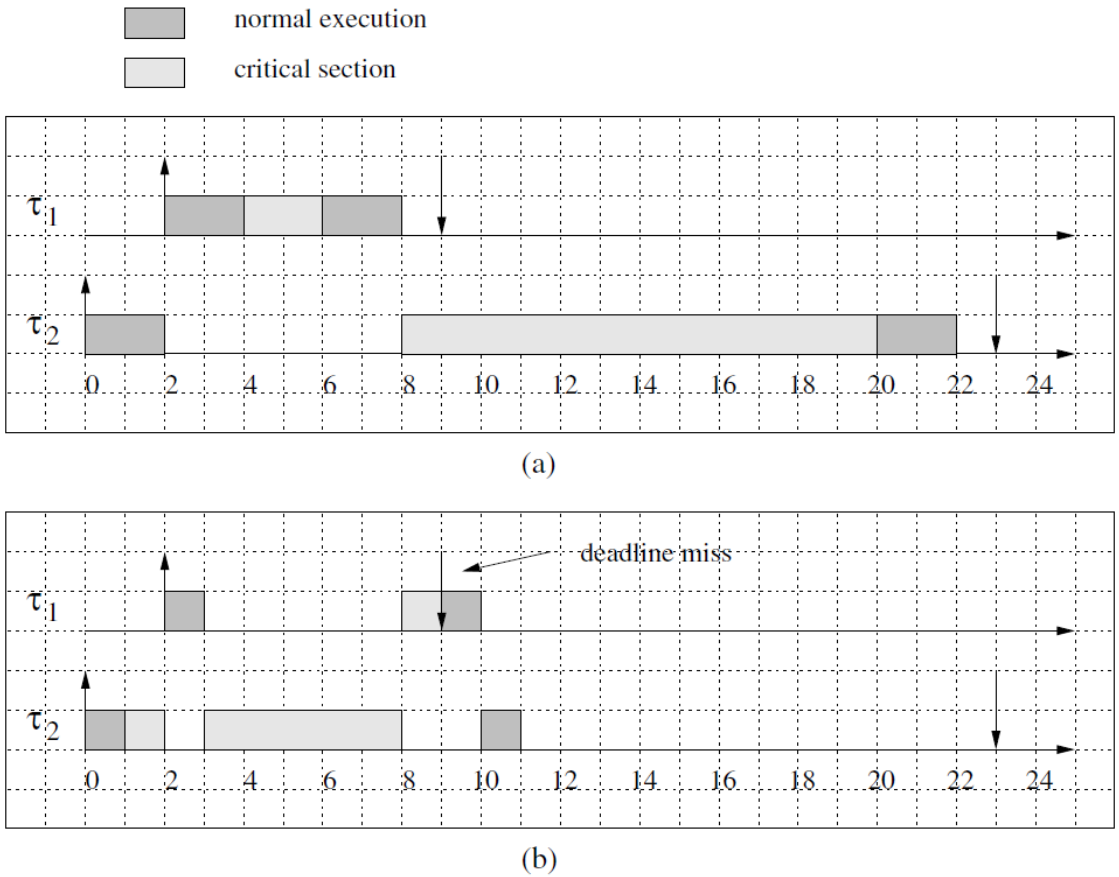
An Example of Graham's Theorem: Anomalies under Resource Constraints (2)

- Reducing the computation time of job J_1 on the first core, then J_2 can begin earlier and take the resource before J_4. As a consequence, job J_4 must now block over the shared resource and possibly miss its deadline.
- The blocking time experienced by J_4 causes a delay in the execution of J_5 (which may also miss its deadline), increasing the total completion time of the task set from 17 to 22.



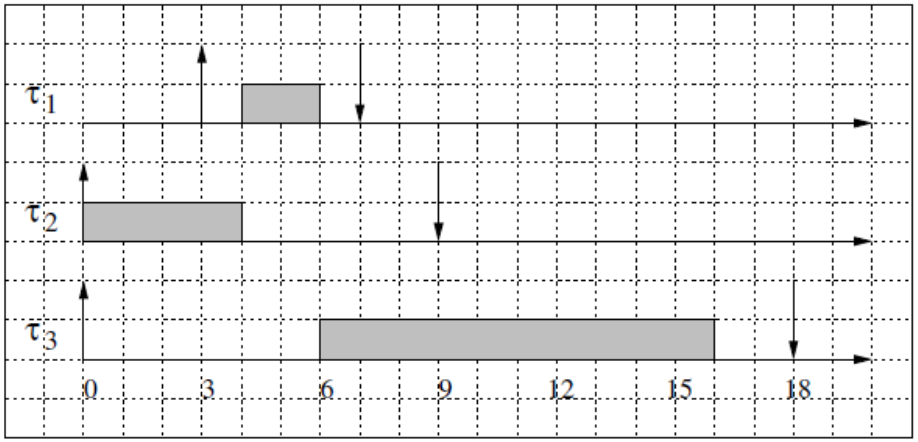
An Example of Graham's Theorem: Anomalies under Resource Constraints (Increasing Speed)

- A real-time application that is feasible on a given processor can become infeasible when running on a faster processor.

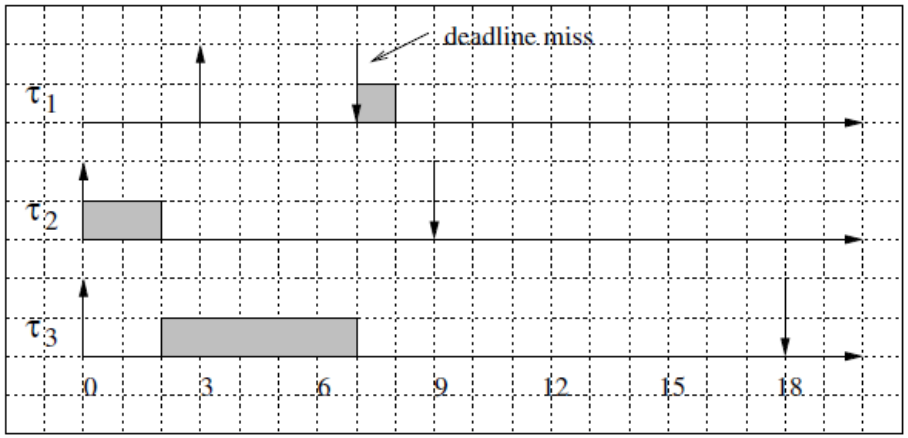


An Example of Graham's Theorem: Anomalies under Non-Preemptive Scheduling

- Tasks are assigned a fixed priority proportional to their relative deadline, thus τ_1 is the task with the highest priority and τ_3 is the task with the lowest priority.
- If task set is executed with double speed $S_2=2S_1$, τ_1 misses its deadline.
  - This happens because, when τ_1 arrives, τ_3 already started its execution and cannot be preempted (due to the non-preemptive mode).



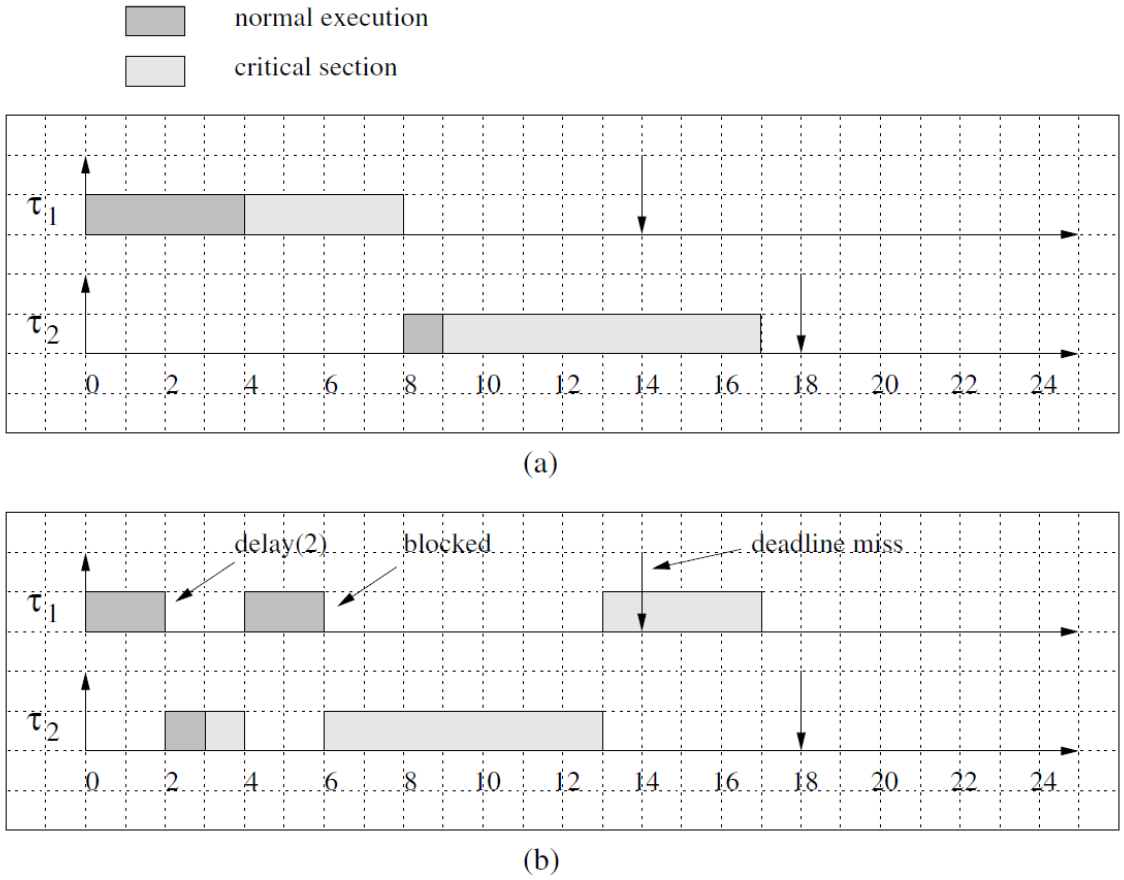
(a)



(b)

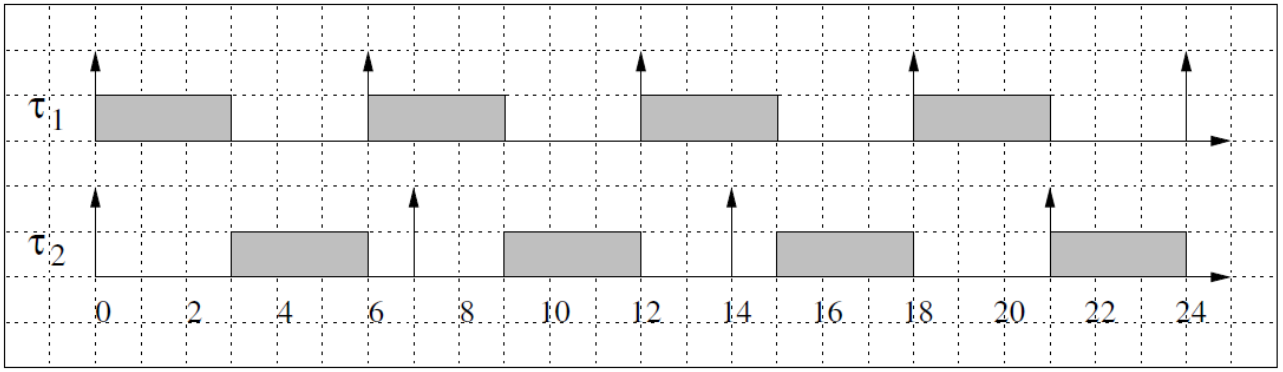
An Example of Graham's Theorem: Anomalies Using A Delay Primitive

- Another timing anomaly can occur when tasks using shared resources explicitly suspend themselves through a $delay(T)$ **system call**, which suspends the execution of the calling task for T units of time.

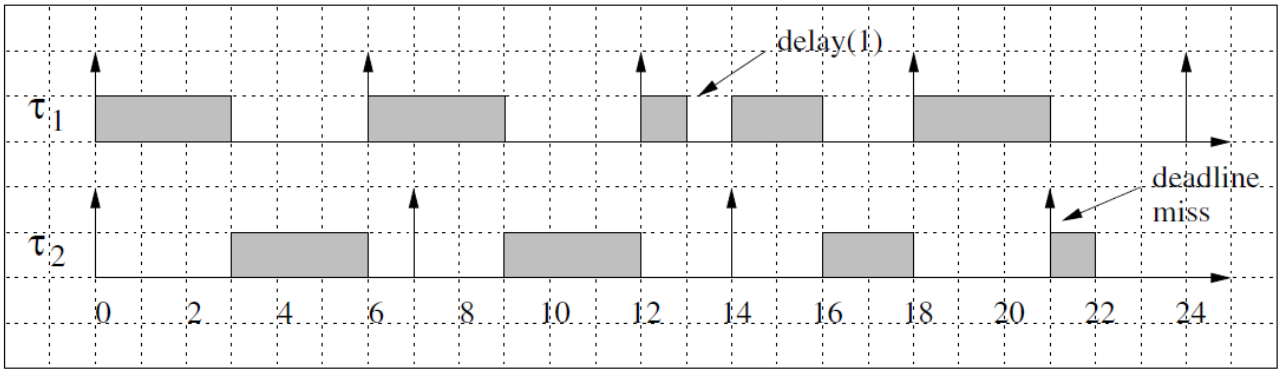


An Example of Graham's Theorem: Anomalies Using A Delay Primitive (2)

- Another case in which the suspension of a task can also cause a longer delay in a different task, even without sharing any resource.



(a)



(b)

Summary

- Definition of Process, Scheduling, Ready Queue, and etc.
- Preemption
- Types of Tasks Constraints
- Characterizations of Real-Time Tasks
- Precedence and Resource Constraints
- References