3. Introduction to Real-Time Scheduling Policies

© Lothar Thiele

ETH Zurich, Switzerland

Contents of Lectures (Lothar Thiele)

Topics

Basic Terms

- ***Real-time systems***
  - ***Hard***: A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control. Examples are sensory data acquisition, detection of critical conditions, actuator servoing.

  - ***Soft***: A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior. Examples are command interpreter of the user interface, displaying messages on the screen.

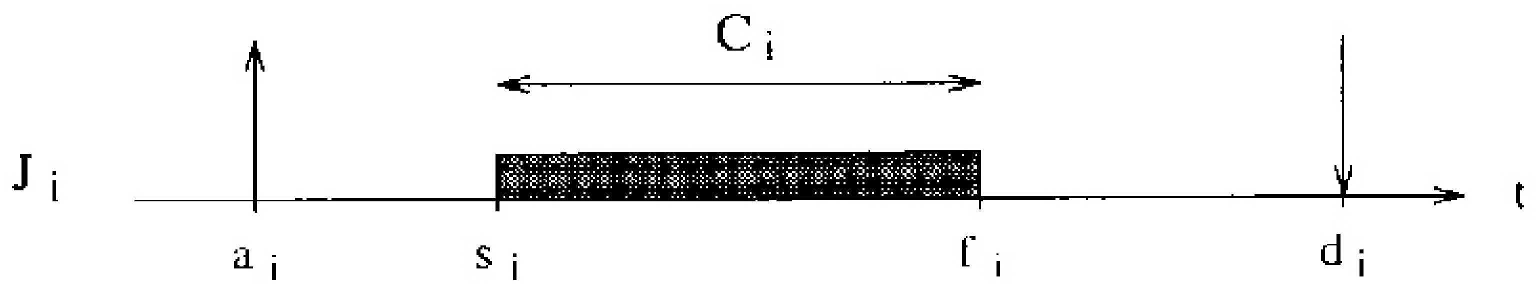
Schedule

- Given a set of ***tasks*** $J = \{J_1, J_2, ...\}$:
  - A ***schedule*** is an assignment of tasks to the processor, such that each task is executed until completion.
  - A ***schedule*** can be defined as an integer step function $\sigma : R \rightarrow N$ where $\sigma(t)$ denotes the task which is executed at time *t*. If $\sigma(t) = 0$ then the processor is called idle.
  - If $\sigma(t)$ changes its value at some time, then the processor performs a ***context switch***.
  - Each interval, in which $\sigma(t)$ is constant is called a ***time slice***.
  - 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.

Schedule and Timing

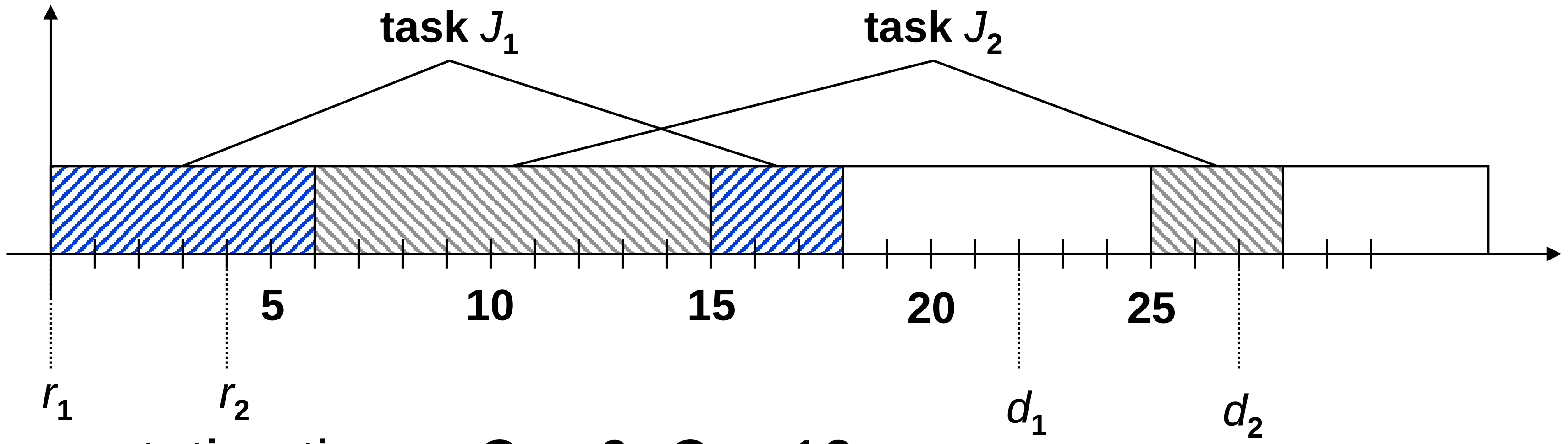
- A schedule is said to be ***feasible***, if all task can be completed according to a set of specified constraints.
- A set of tasks is said to be ***schedulable***, if there exists at least one algorithm that can produce a feasible schedule.
- ***Arrival time*** a_i or ***release time*** r_i is the time at which a task becomes ready for execution.
- ***Computation time*** C_i is the time necessary to the processor for executing the task without interruption.
- ***Deadline*** d_i is the time at which a task should be completed.
- ***Start time*** s_i is the time at which a task starts its execution.
- ***Finishing time*** f_i is the time at which a task finishes its execution.

Schedule and Timing



- Using the above definitions, we have $d_i \geq r_i + C_i$
- ***Lateness*** $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that if a task completes before the deadline, its lateness is negative.
- ***Tardiness* or *exceeding time*** $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- ***Laxity or slack time*** $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

Example



Computation times: $C_1 = 9$, $C_2 = 12$

Start times: $s_1 = 0$, $s_2 = 6$

Finishing times: $f_1 = 18$, $f_2 = 28$

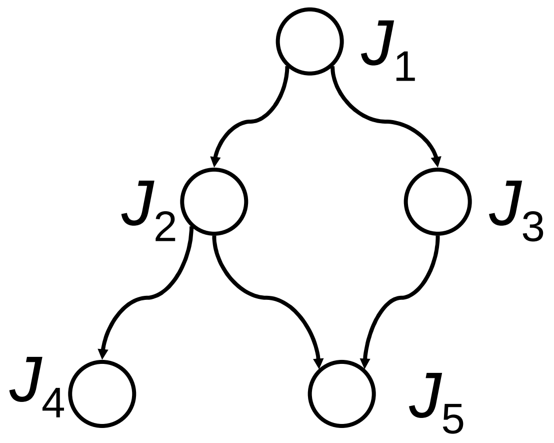
Lateness: $L_1 = -4$, $L_2 = 1$

Tardiness: $E_1 = 0$, $E_2 = 1$

Laxity: $X_1 = 13$, $X_2 = 11$

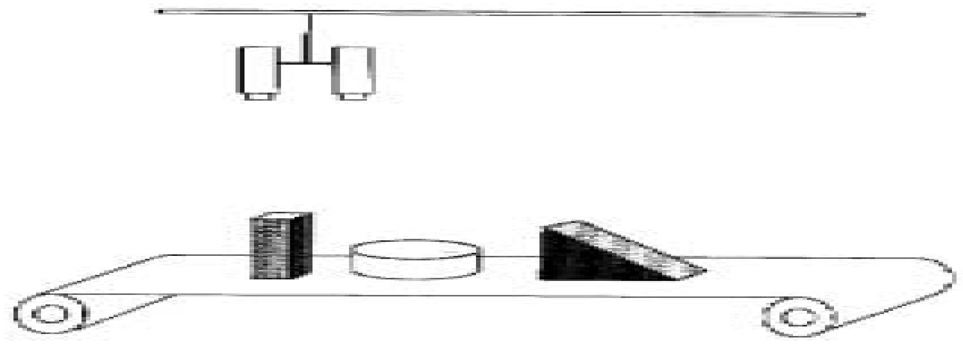
Precedence Constraints

- ***Precedence relations*** between graphs can be described through an acyclic directed graph *G* where tasks are represented by nodes and precedence relations by arrows. *G* induces a partial order on the task set.
- There are ***different interpretations possible***:
  - All successors of a task are activated (concurrent task execution).
  - One successor of a task is activated (non-deterministic choice).

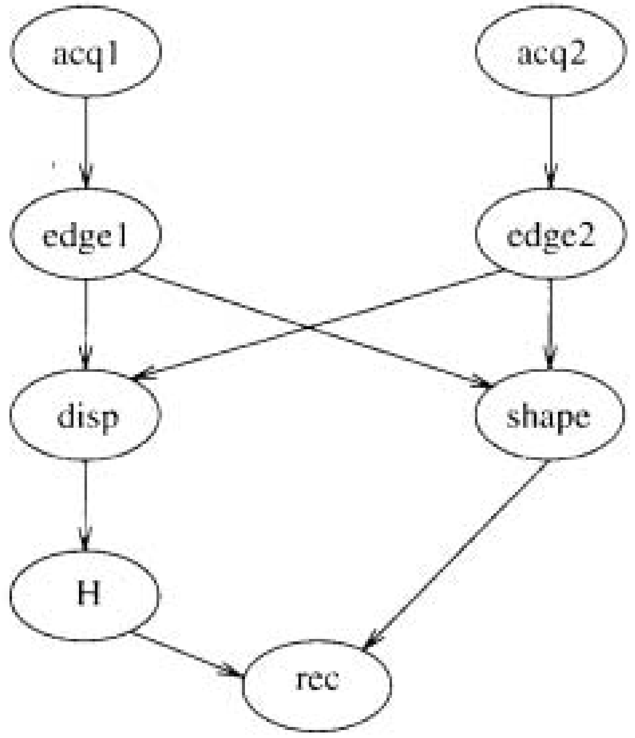


Precedence Constraints

- ***Example*** (concurrent activation):



  - Image acquisition $acq1\ \ acq2$
  - Low level image processing $edge1\ edge2$
  - Feature/contour extraction $shape$
  - Pixel disparities $disp$
  - Object size H
  - Object recognition rec

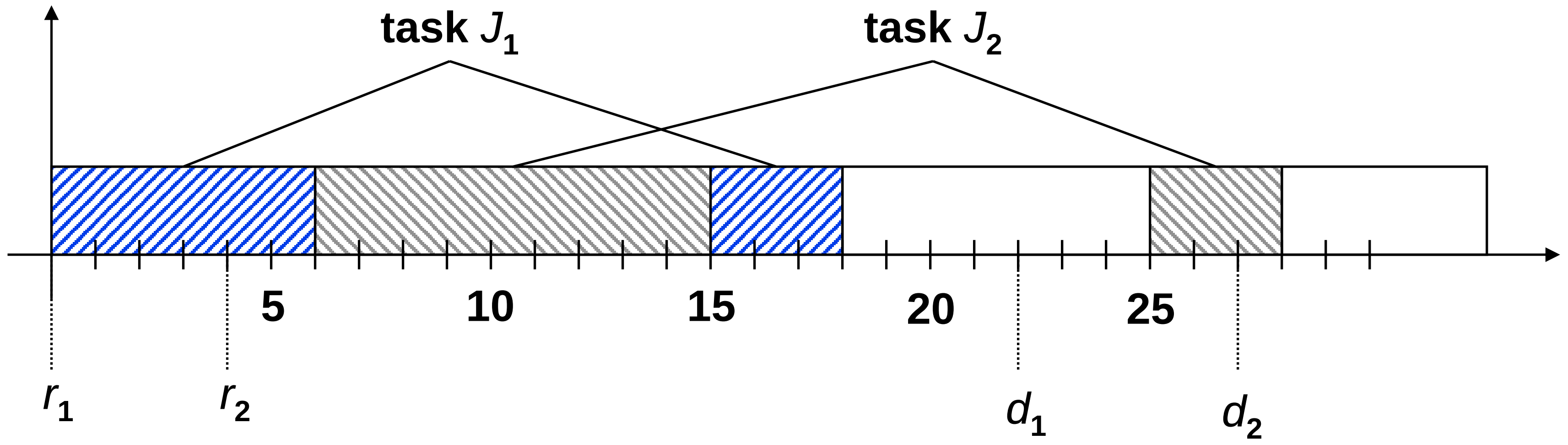


Metrics

- Average response time: $$\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - r_i)$$
- Total completion time: $$t_c = \max_i(f_i) - \min_i(r_i)$$
- Weighted sum of completion time: $$t_w = \frac{\sum_{i=1}^{n} w_i(f_i - r_i)}{\sum_{i=1}^{n} w_i}$$
- ***Maximum lateness***: $$L_{\max} = \max_i(f_i - d_i)$$
- ***Maximum number of late tasks***: $$N_{\text{late}} = \sum_{i=1}^{n} miss(f_i)$$

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

Metrics Example



Average response time: $\overline{t_r} = \frac{1}{2}(18+24) = 21$

Total completion time: $t_c = 28 - 0 = 28$

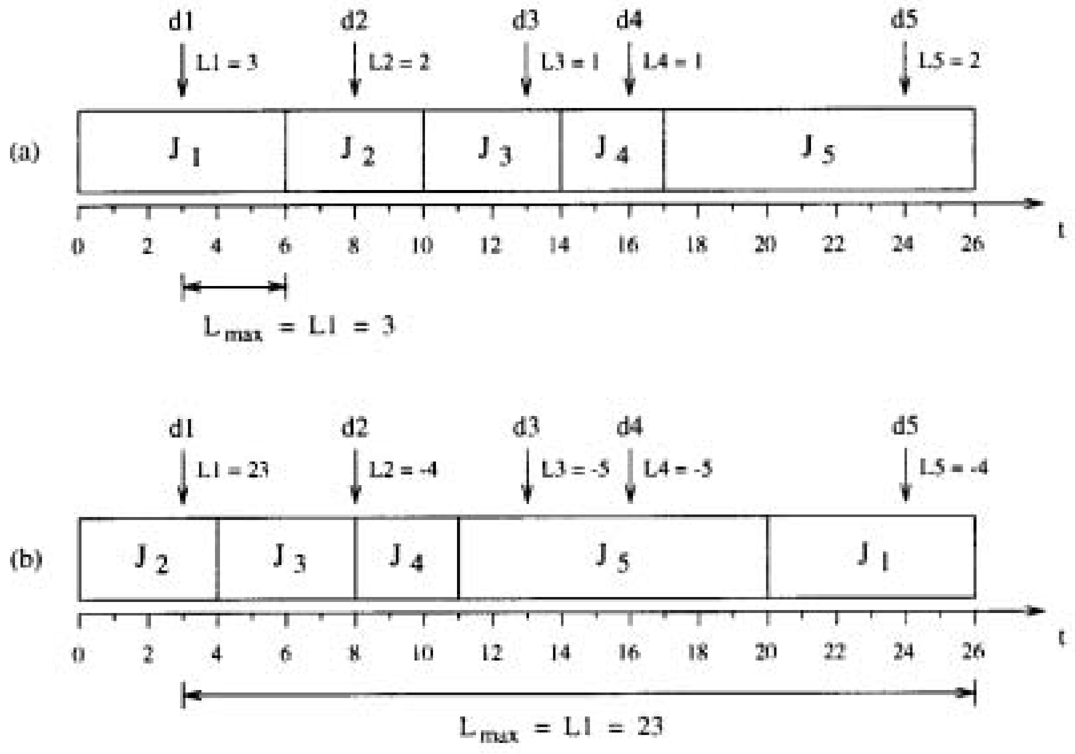
Weighted sum of compl. time: $w_1 = 2, w_2 = 1: \; t_w = \frac{2 \cdot 18 + 24}{3} = 20$

Maximum number of late tasks: $N_{\text{late}} = 1$

Maximum lateness: $L_{\max} = 1$

Scheduling Example

- In (a), the maximum lateness is minimized, but all tasks miss their deadlines.
- In (b), the maximal lateness is larger, but only one task misses its deadline.



Topics

- Basic Models and Terms
- ***Aperiodic Task Sets***
- Periodic Task Sets
- Mixed Aperiodic and Periodic Task Sets
- Shared Resources

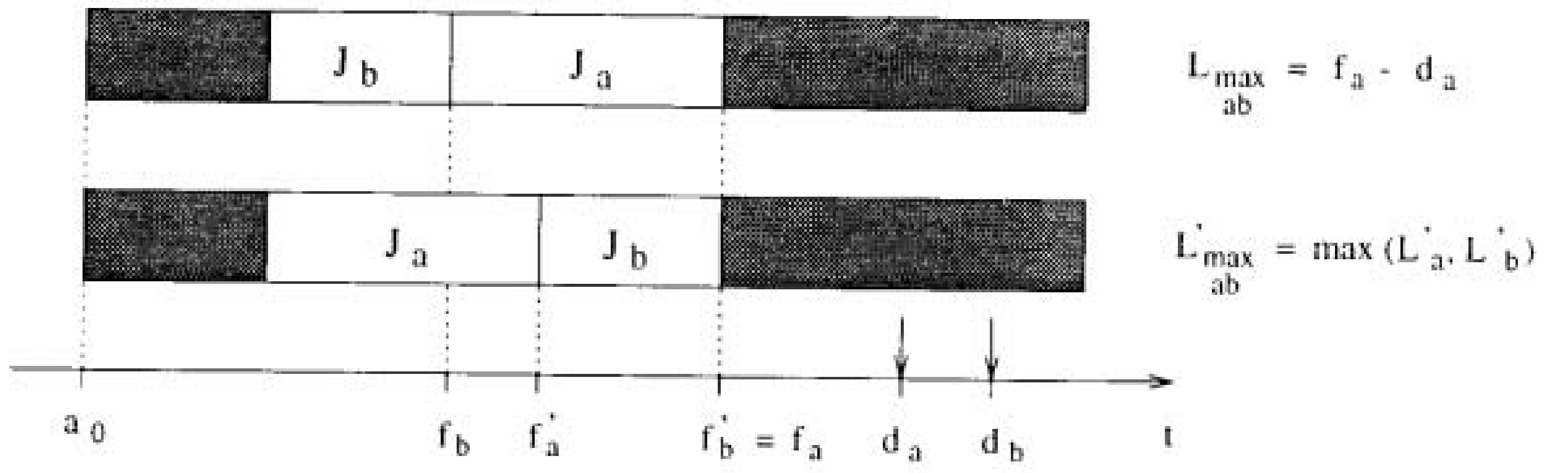
Overview

- Scheduling of ***aperiodic tasks*** with real-time constraints:
  - Table with some known algorithms:

	Equal arrival times non preemptive	Arbitrary arrival times preemptive
Independent tasks	EDD (Jackson)	EDF (Horn)
Dependent tasks	LDF (Lawler)	EDF* (Chetto)

Earliest Deadline Due (EDD)

- ***Jackson's rule***: Given a set of ***n*** tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.
- ***Proof concept***:



if $(L'_a \geq L'_b)$ then $L'_{max_{ab}} = f'_a - d_a < f_a - d_a$

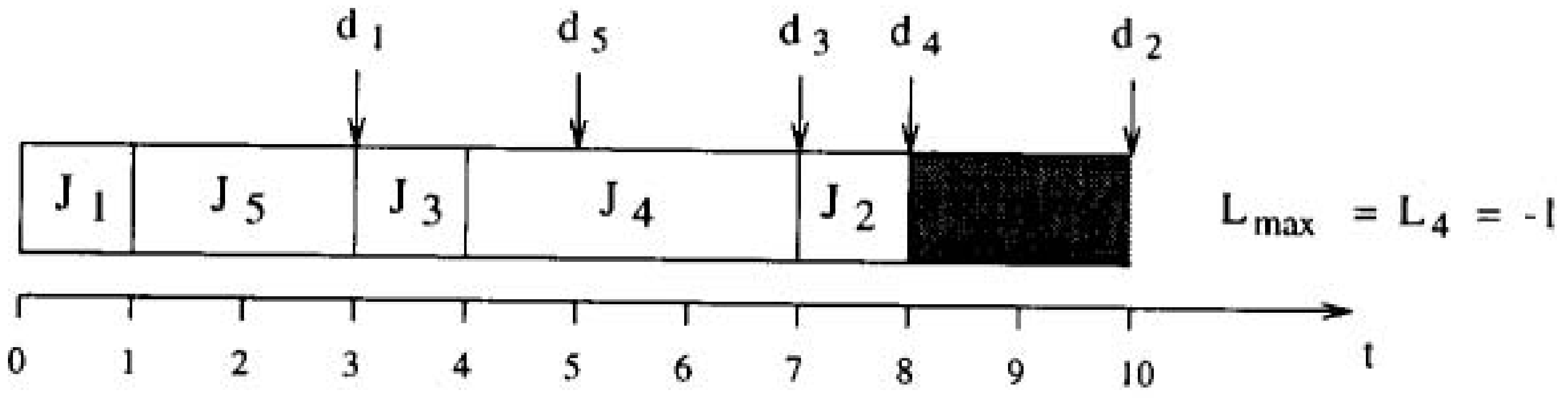
if $(L'_a \leq L'_b)$ then $L'_{max_{ab}} = f'_b - d_b < f_a - d_a$

in both cases: $L'_{max_{ab}} < L_{max_{ab}}$

Earliest Deadline Due (EDD)

- ***Example 1***:

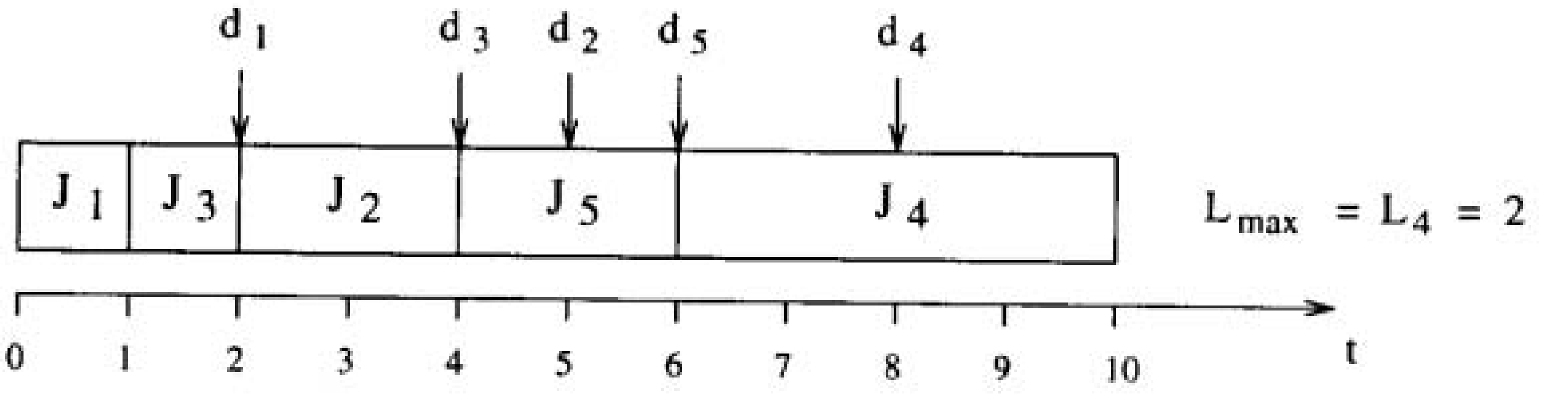
	J_1	J_2	J_3	J_4	J_5
C_i	1	1	1	3	2
d_i	3	10	7	8	5



Earliest Deadline Due (EDD)

- *Example 2*:

	J_1	J_2	J_3	J_4	J_5
C_i	1	2	1	4	2
d_i	2	5	4	8	6



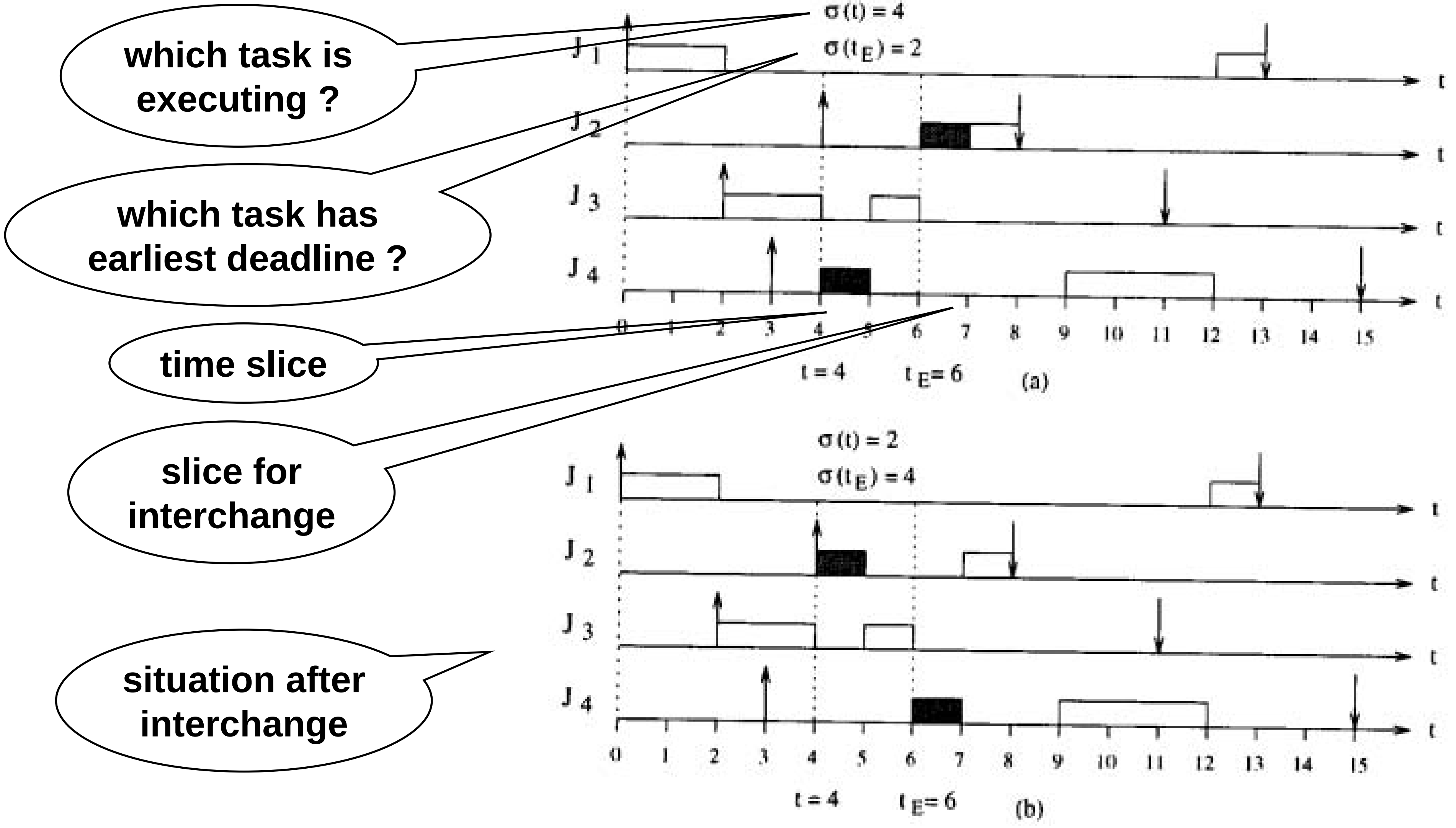
Earliest Deadline First (EDF)

- ***Horn's rule***: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.
- ***Concept of proof***: For each time interval $[t, t+1)$ it is verified, whether the actual running task is the one with the earliest absolute deadline. If this is not the case, the task with the earliest absolute deadline is executed in this interval instead. This operation cannot increase the maximum lateness.

Earliest Deadline First (EDF)

- Used quantities and terms:
  - $\sigma(t)$ identifies the task executing in the slice $[t, t+1)$
  - $E(t)$ identifies the ready task that, at time t, has the earliest deadline
  - $t_E(t)$ is the time $(\geq t)$ at which the next slice of task $E(t)$ begins its execution in the current schedule

Earliest Deadline First (EDF)



Earliest Deadline First (EDF)

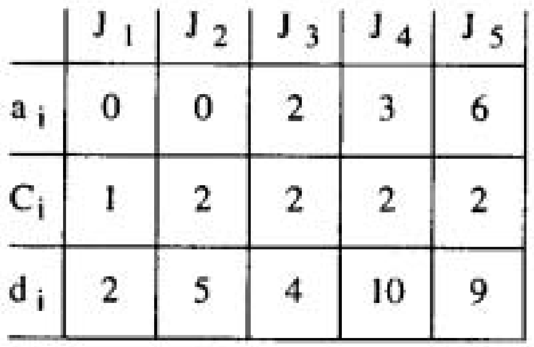
remaining worst-case execution time of task k; tasks ordered by deadline

- ***Guarantee***:
  - worst case finishing time of task i: $f_i = \sum_{k=1}^{i} c_k(t)$
  - EDF guarantee condition: $\forall i = 1,\dots,n \quad \sum_{k=1}^{i} c_k(t) \le d_i$
  - algorithm:

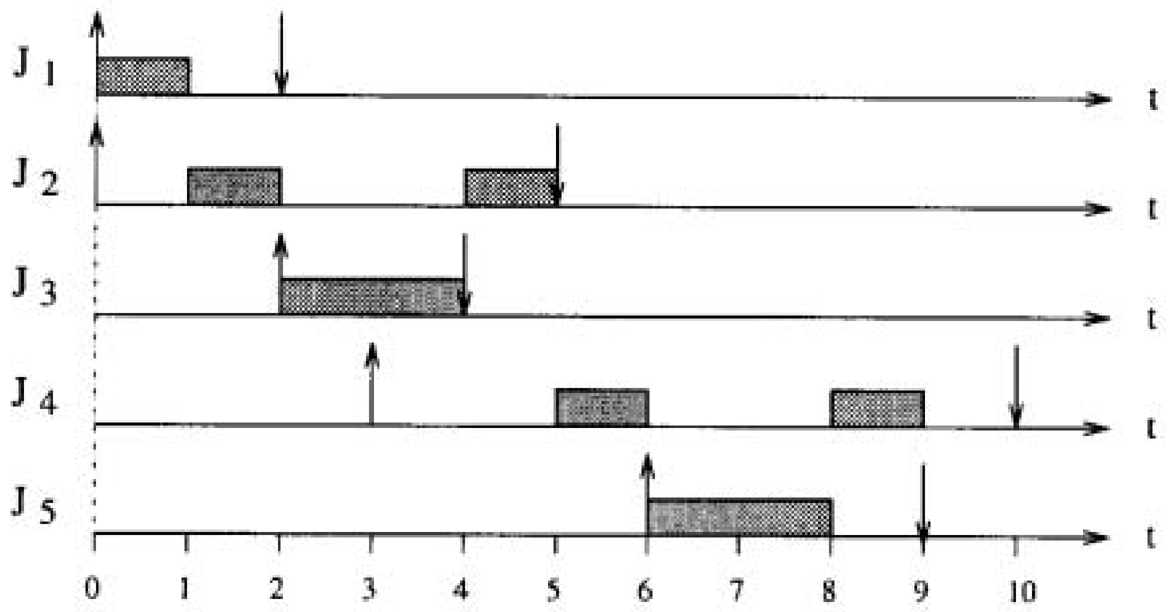
```
Algorithm: EDF_guarantee (J, $J_{new}$)
{       J'=J∪{$J_{new}$};  /* ordered by deadline */
        t = current_time();
        $f_0$ = 0;
        for (each $J_i$∈J') {
                $f_i$ = $f_{i-1}$ + $c_i$(t);
                if ($f_i$ > $d_i$) return(INFEASIBLE);
        }
        return(FEASIBLE);
}
```

Earliest Deadline First (EDF)

- ***Example***:



	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9



Topics

- Basic Models and Terms
- Aperiodic Task Sets
- ***Periodic Task Sets***
- Mixed Aperiodic and Periodic Task Sets
- Shared Resources

Overview

- Table of some known ***preemptive scheduling*** algorithms for ***periodic tasks***:

	Deadline equals period	Deadline smaller than period
static priority	RM (rate-monotonic)	DM (deadline-monotonic)
dynamic priority	EDF	EDF*

Model of Periodic Tasks

- ***Examples***: sensory data acquisition, low-level servoing, control loops, action planning and system monitoring. When a control application consists of several concurrent periodic tasks with individual timing constraints, the OS has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline.
- ***Definitions***:

  Γ : denotes a set of periodic tasks
  τ_i : denotes a generic periodic task
  $\tau_{i,j}$: denotes the *j*th instance of task *i*
  $r_{i,j}, s_{i,j}, f_{i,j}, d_{i,j}$:
  denotes the release time, start time, finishing time, absolute deadline of the *j*th instance of task *i*
  Φ_i : phase of task *i* (release time of its first instance)
  D_i : relative deadline of task *i*

Model of Periodic Tasks

- The following ***hypotheses*** are assumed on the tasks:
  - The instances of a periodic task are ***regularly activated*** at a constant rate. The interval T_i between two consecutive activations is called period. The release times satisfy

$$r_{i,j} = \Phi_i + (j-1)T_i$$

  - All instances have the ***same worst case execution time*** C_i
  - All instances of a periodic task have the ***same relative deadline*** D_i. Therefore, the absolute deadlines satisfy

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

Model of Periodic Tasks

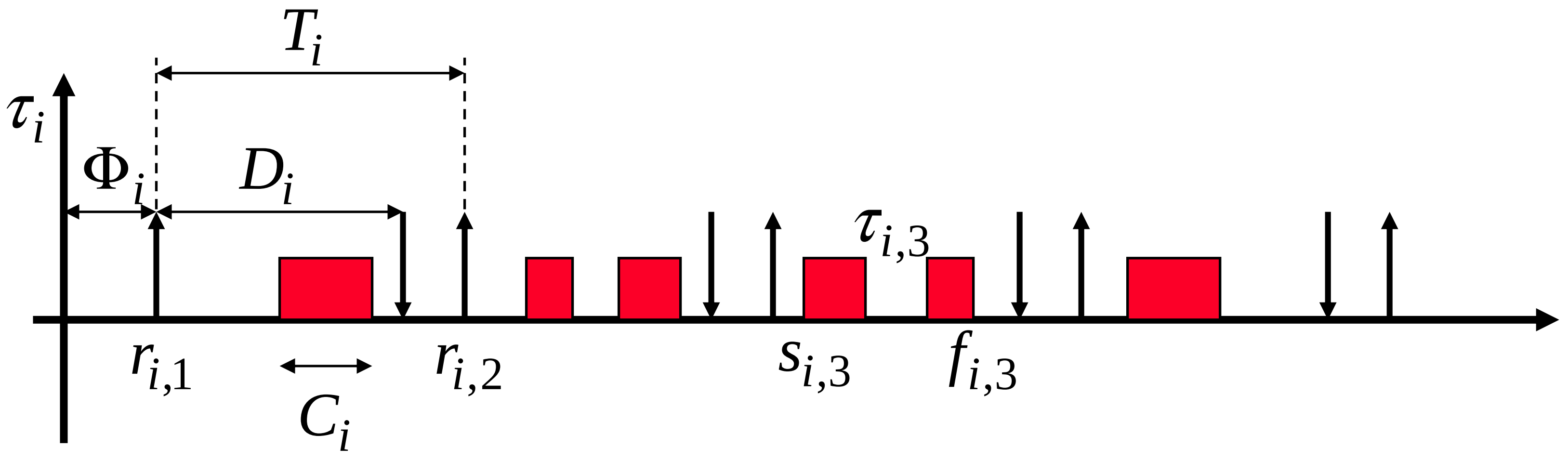
- The following ***hypotheses*** are assumed on the tasks cont':
  - Often, the relative deadline equals the period $D_i = T_i$ and therefore

    $$d_{i,j} = \Phi_i + jT_i$$

  - All periodic tasks are ***independent***; that is, there are no precedence relations and no resource constraints.
  - No task can suspend itself, for example on I/O operations.
  - All tasks are released as soon as they arrive.
  - All overheads in the OS kernel are assumed to be zero.

Model of Periodic Tasks

- ***Example***:



Rate Monotonic Scheduling (RM)

- ***Assumptions***:
  - Task priorities are assigned to tasks before execution and do not change over time (***static priority assignment***).
  - RM is intrinsically ***preemptive***: the currently executing task is preempted by a task with higher priority.
  - ***Deadlines*** equal the periods $D_i = T_i$.

- ***Algorithm***: Each task is assigned a priority. Tasks with higher request rates (that is with shorter periods) will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.

Rate Monotonic Scheduling (RM)

- ***Optimality***: RM is optimal among all fixed-priority assignments in the sense that not other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM.
- The ***proof*** is done by considering several cases that may occur, but the main ideas are as follows:
  - ***A critical instant for any task occurs whenever the task is released simultaneously with all higher priority tasks***. The tasks schedulability can easily be checked at their critical instances. If all tasks are feasible at their critical instants, then the task set is schedulable in any other condition.
  - Show that, given two periodic tasks, if the schedule is feasible by an arbitrary priority assignment, then it is also feasible by RM.
  - Extend the result to a set of *n* periodic tasks.

Rate Monotonic Scheduling (RM)

- ***Schedulability analysis***: A set of periodic tasks is schedulable with RM if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

  This condition is sufficient but not necessary (in general). The proof of this condition is rather involved.

- The term

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

  denotes the ***processor utilization factor*** *U* which is the fraction of processor time spent in the execution of the task set.

Deadline Monotonic Scheduling (DM)

- ***Assumptions*** are as in rate monotonic scheduling, but
  - deadlines may be smaller than the periodic, i.e.

$$C_i \leq D_i \leq T_i$$

- ***Algorithm***: Each task is assigned a priority. Tasks with smaller deadlines will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.

- ***Schedulability analysis***: A set of periodic tasks is schedulable with DM if

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{1/n} - 1\right)$$

This condition is sufficient but not necessary (in general).

Deadline Monotonic Scheduling (DM)

- There is also a ***necessary and sufficient schedulability test*** which is computationally more involved. It is based on the following observations:
  - The worst-case processor demand occurs when all tasks are released simultaneously; that is, at their ***critical instances***.
  - For each task ***i***, the sum of its processing time and the interference (preemption) imposed by higher priority tasks must be less than or equal to D_i .
  - A measure of the ***worst case interference*** for task ***i*** can be computed as the sum of the processing times of all higher priority tasks released before some time t where tasks are ordered according to $i < j \Leftrightarrow D_i < D_j$:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

Deadline Monotonic Scheduling (DM)

- The ***longest response time*** R_i of a periodic task *i* is computed, at the critical instant, as the sum of its computation time and the interference due to preemption by higher priority tasks

$$R_i = C_i + I_i$$

- Hence, the ***schedulability test*** needs to compute the smallest R_i that satisfies

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

  for all tasks *i*. Then, $R_i \leq D_i$ must hold for all tasks *i*.
- It can be shown that this condition is necessary and sufficient.

Deadline Monotonic Scheduling (DM)

- The ***longest response times*** R_i of the periodic tasks *i* can be computed iteratively by the following algorithm:

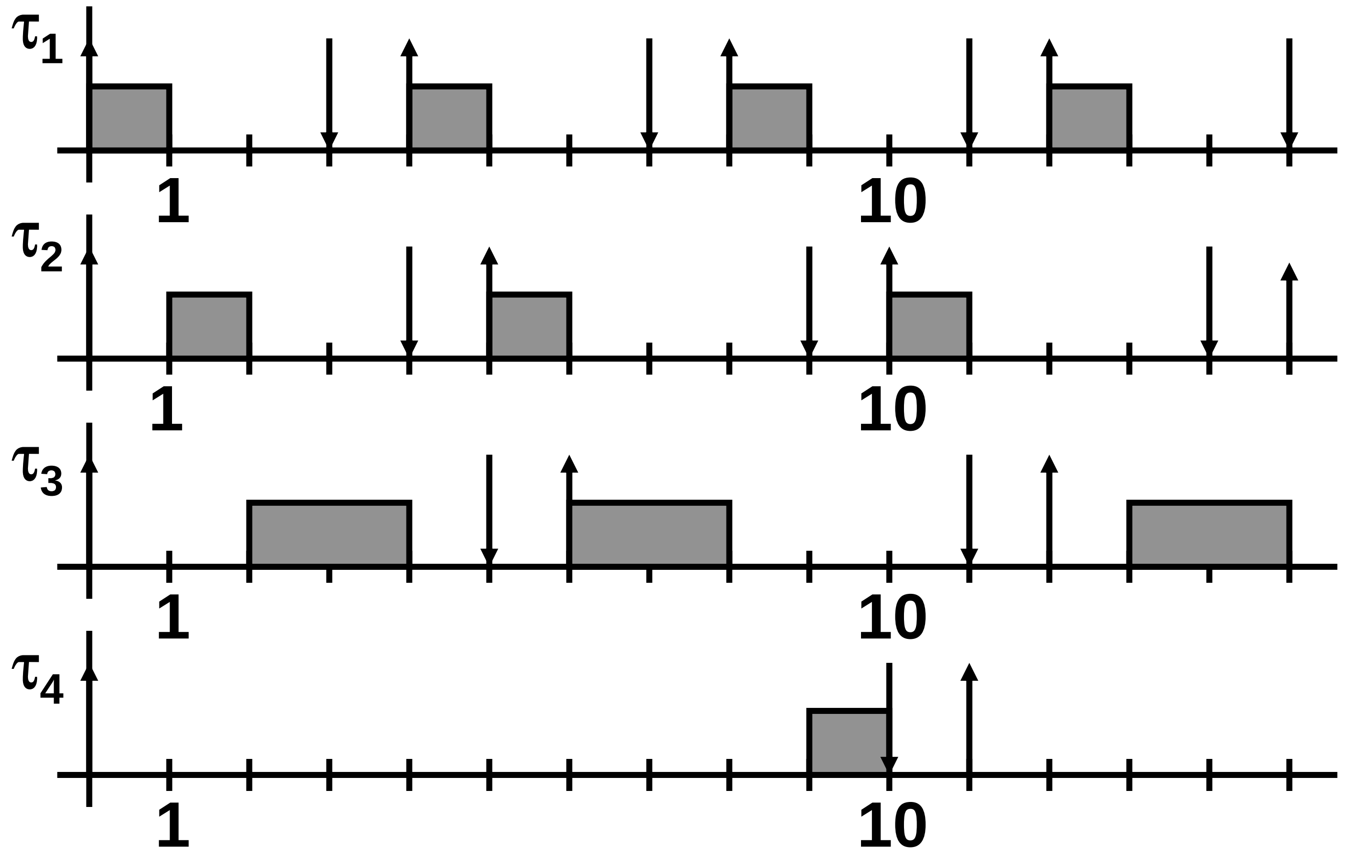
```
Algorithm: DM_guarantee ($\Gamma$)
{       for (each $\tau_i \in \Gamma$){
                I = 0;
                do {
                        R = I + $C_i$;
                        if (R > $D_i$) return(UNSCHEDULABLE);
                        I = $\sum^{j=1,\ldots,(i-1)} \lceil R/T_j \rceil$ $C_j$;
                } while (I + $C_i$ > R);
        }
        return(SCHEDULABLE);
}
```

DM Example

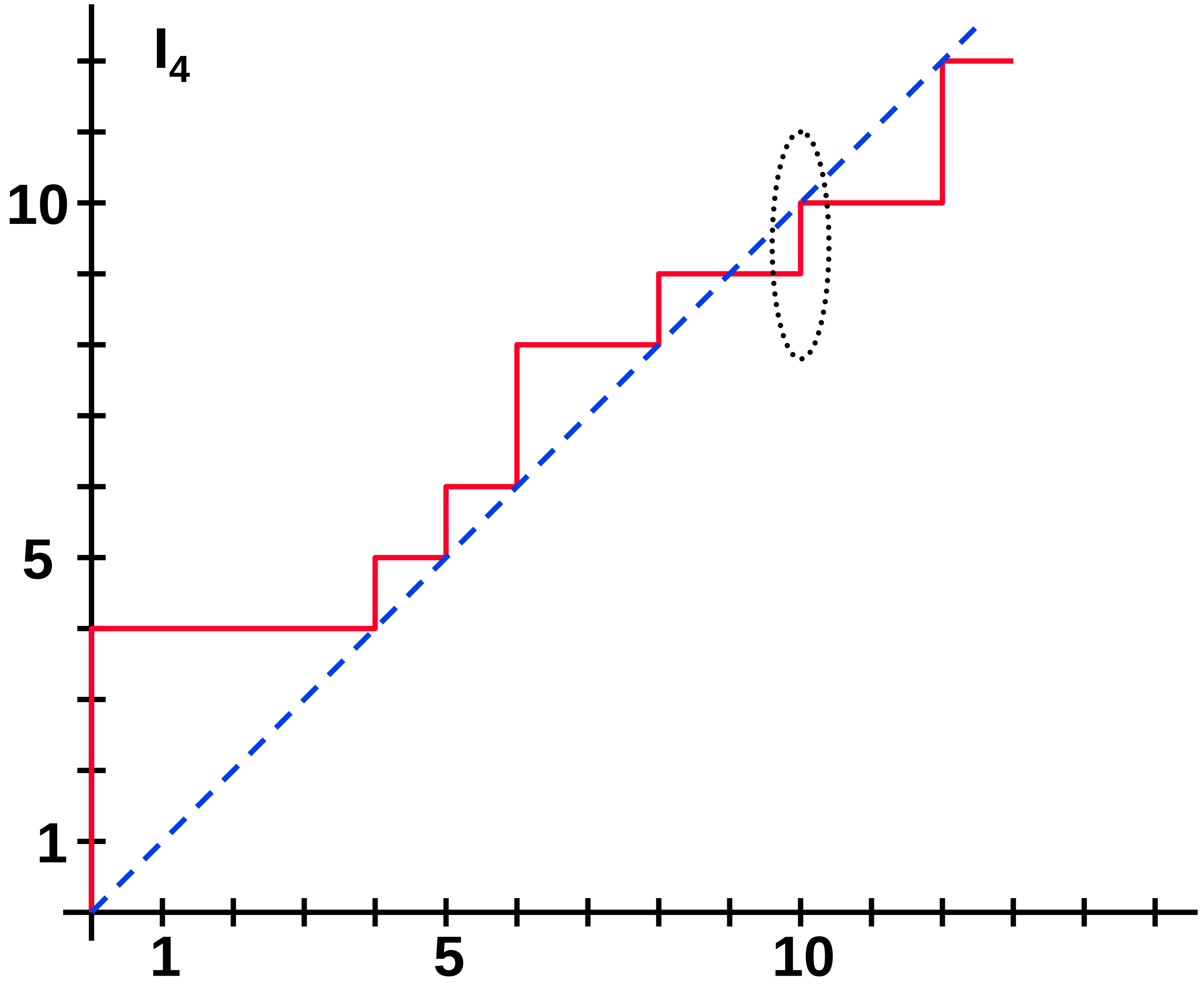
- ***Example***:
  - Task 1: $C_1 = 1; T_1 = 4; D_1 = 3$
  - Task 2: $C_2 = 1; T_2 = 5; D_2 = 4$
  - Task 3: $C_3 = 2; T_3 = 6; D_3 = 5$
  - Task 4: $C_4 = 1; T_4 = 11; D_4 = 10$
- ***Algorithm*** for task 4:
  - Step 0: $R_4 = 1$
  - Step 1: $R_4 = 5$
  - Step 2: $R_4 = 6$
  - Step 3: $R_4 = 7$
  - Step 4: $R_4 = 9$
  - Step 5: $R_4 = 10$

DM Example

$U = 0.874 \qquad \sum_{i=1}^{n} \frac{C_i}{D_i} = 1.08 > n\left(2^{1/n} - 1\right) = 0.757$



DM Example



EDF Scheduling (earliest deadline first)

- ***Assumptions***:
  - dynamic priority assignment
  - intrinsically preemptive
  - $D_i \leq T_i$
- ***Algorithm***: The currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

- ***Optimality***: No other algorithm can schedule a set of periodic tasks if the set that can not be scheduled by EDF.
- The ***proof*** is simple and follows that of the aperiodic case.

EDF Scheduling

- A necessary and sufficient ***schedulability test*** if $D_i = T_i$:
  - A set of periodic tasks is schedulable with EDF if and only if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} = U \leq 1$$

- The term

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

denotes the ***average processor utilization***.

EDF Scheduling

- If the utilization satisfies $U > 1$, then there is no valid schedule: The total demand of computation time in interval $T = T_1 \cdot T_2 \cdot \ldots \cdot T_n$ is
$$\sum_{i=1}^{n} \frac{C_i}{T_i} T = UT > T$$
and therefore, it exceeds the available processor time.

- If the utilization satisfies $U \leq 1$, then there is a valid schedule (proof by contradiction): Assume that deadline is missed at some time t_2 with $U \leq 1$.

EDF Scheduling

- Within an interval $[t_1, t_2]$ the total computation time demanded by periodic tasks is bounded by

$$C_p(t_1,t_2) = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \le \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)U$$

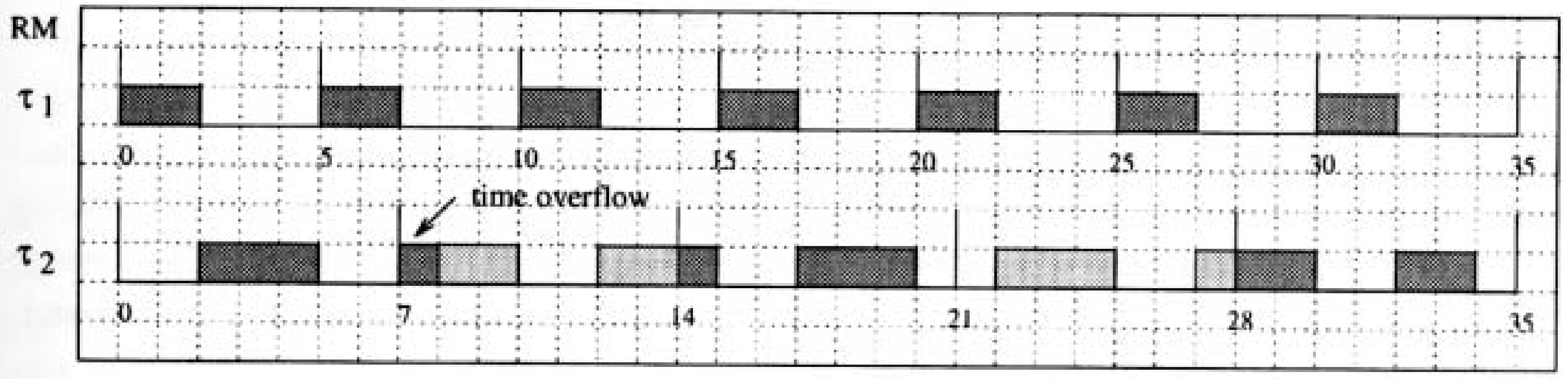
number of complete periods of task i in the interval [t_1, t_2]

- Since the deadline at time t_2 is missed, we must have:

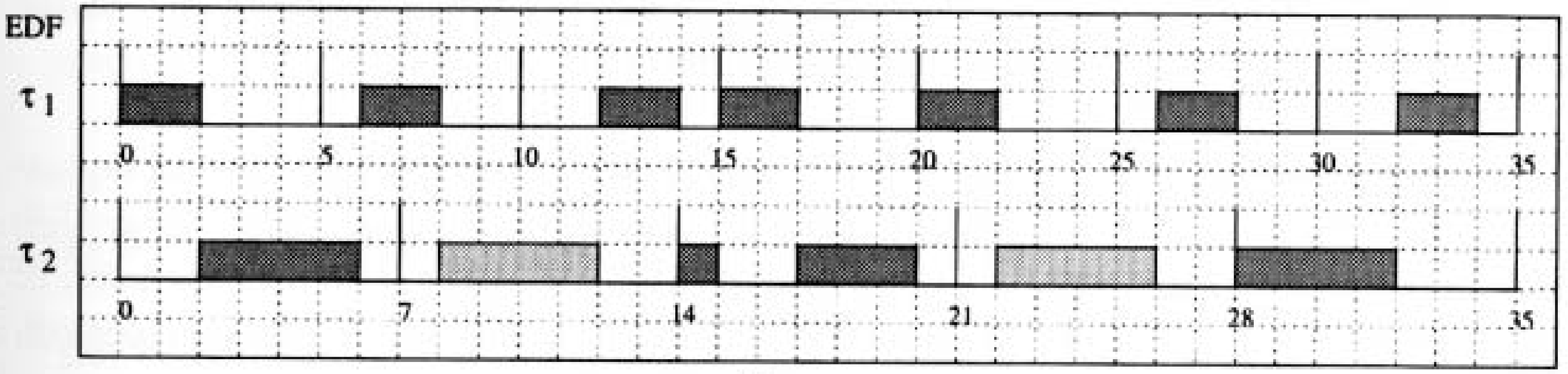
$$t_2 - t_1 < C_p(t_1,t_2) \le (t_2 - t_1)U \Rightarrow U > 1$$

Periodic Tasks

- ***Example***: 2 tasks, deadline = periods, ***U* = 97%**



(a)



(b)

Topics

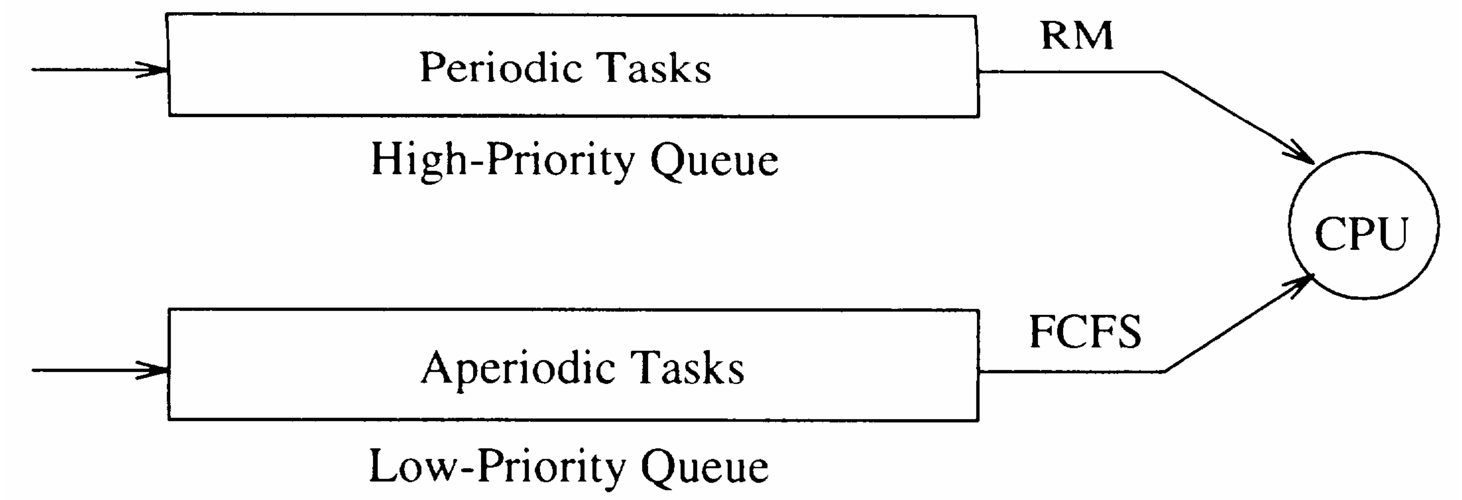
- Basic Models and Terms
- Aperiodic Task Sets
- Periodic Task Sets
- ***Mixed Aperiodic and Periodic Task Sets***
- Shared Resources

Problem of Mixed Task Sets

- In many applications, there are as well aperiodic as periodic tasks.
- ***Periodic tasks***: time-driven, execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates.
- ***Aperiodic tasks***: event-driven, may have hard, soft, non-real-time requirements depending on the specific application.
- ***Sporadic tasks***: Offline guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is by assuming a maximum arrival rate for each critical event. Aperiodic tasks characterized by a minimum interarrival time are called ***sporadic***.

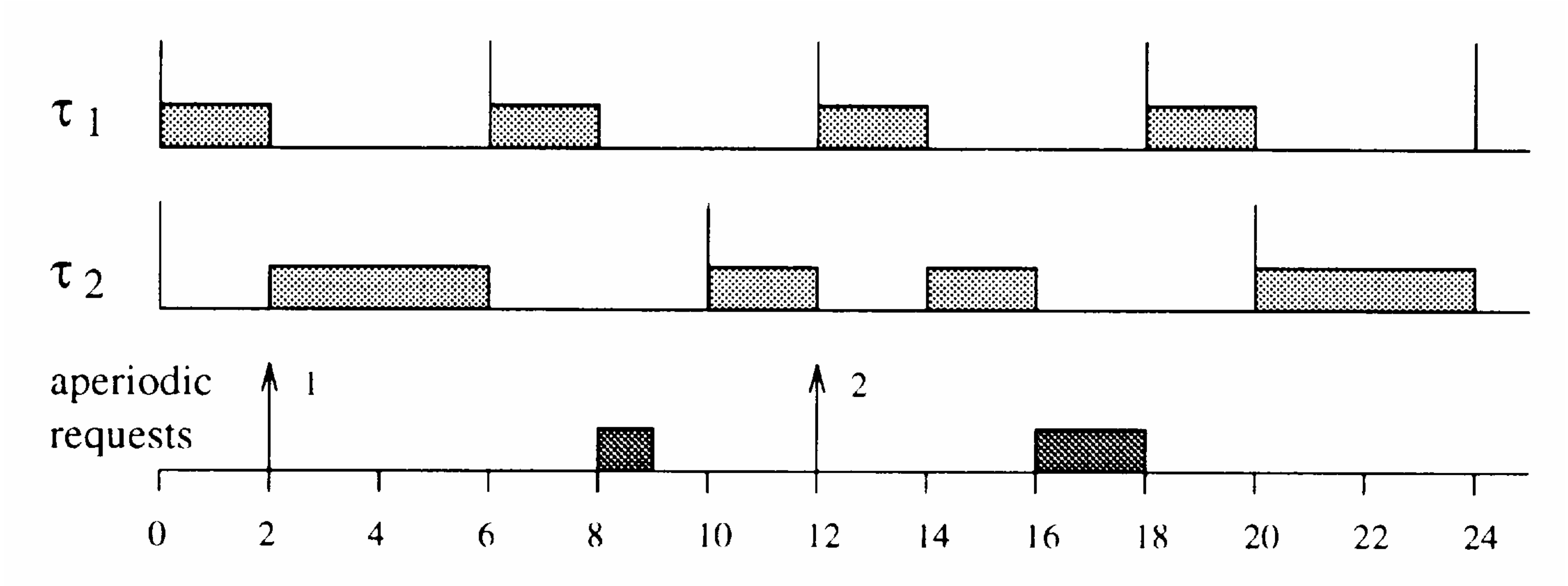
Background Scheduling

- ***Simple solution*** for RM and EDF scheduling of periodic tasks:
  - Processing of aperiodic tasks in the background, i.e. if there are no periodic request.
  - Periodic tasks are not affected.
  - Response of aperiodic tasks may be prohibitively long and there is no possibility to assign a higher priority to them.



Background Scheduling

- ***Example*** (rate monotonic periodic schedule):



RM - Polling Server

- ***Idea***: Introduce an ***artificial periodic task*** whose purpose is to service aperiodic requests as soon as possible (therefore, “server”).
  - Like any periodic task, a server is characterized by a period T_s and a computation time C_s.
  - The server is scheduled with the same algorithm used for the periodic tasks and, once active, it serves the aperiodic requests within the limit of its server capacity.
  - Its priority (period!) can be chosen to match the response time requirement for the aperiodic tasks.

RM - Polling Server

- ***Function of polling server (PS)***
  - At regular intervals equal to T_s , PS becomes active and serves any pending aperiodic requests within the limit of its capacity C_s .
  - If no aperiodic requests are pending, PS suspends itself until the beginning of the next period and the time originally allocated for aperiodic service is ***not preserved for aperiodic execution***.
- ***Disadvantage***: If an aperiodic requests arrives just after the server has suspended, it must wait until the beginning of the next polling period.

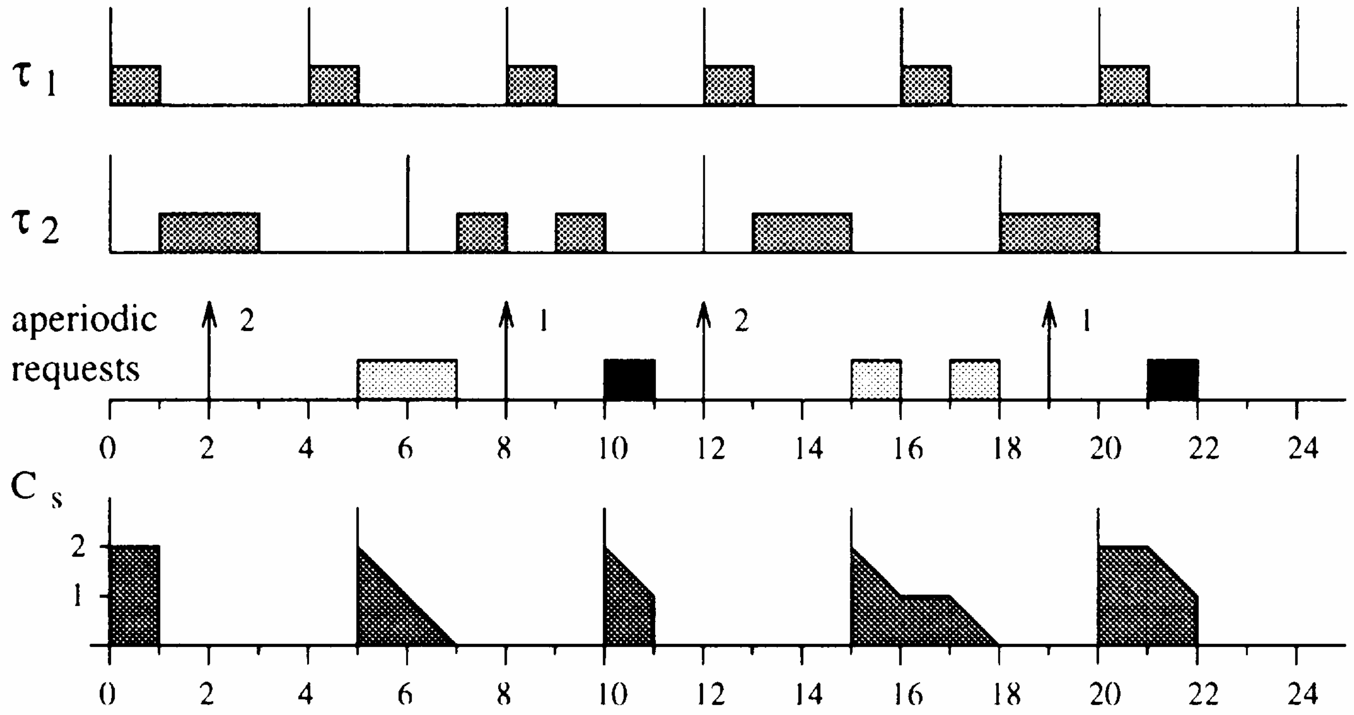
RM - Polling Server

- ***Example***

	C_i	T_i
τ_1	1	4
τ_2	2	6

Server

$C_s = 2$
$T_s = 5$



RM - Polling Server

- ***Schedulability analysis*** of periodic tasks
  - As in the case of RM as the interference by a server task is the same as the one introduced by an equivalent periodic task.
  - A set of periodic tasks and a server task can be executed within their deadlines if

$$\frac{C_s}{T_s} + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq (n+1)\left(2^{1/(n+1)} - 1\right)$$

  - Again, this test is sufficient but not necessary.

RM - Polling Server

- ***Aperiodic guarantee*** of aperiodic activities.
- ***Assumption***: An aperiodic task is finished before a new aperiodic request arrives.
  - Computation time C_a, deadline D_a .
  - ***Sufficient schedulability test***:

$$(1+\left\lceil \frac{C_a}{C_s} \right\rceil)T_s \le D_a$$

If the server task has the highest priority there is a necessary test also.

The aperiodic task arrives shortly after the activation of the server task.

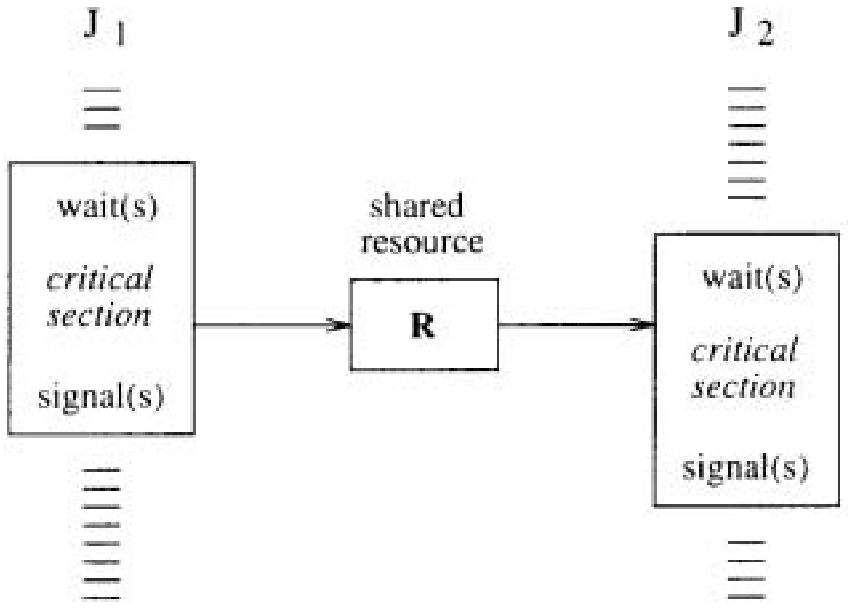
Maximal number of necessary server periods.

Topics

- Basic Models and Terms
- Aperiodic Task Sets
- Periodic Task Sets
- Mixed Aperiodic and Periodic Task Sets
- ***Shared Resources***

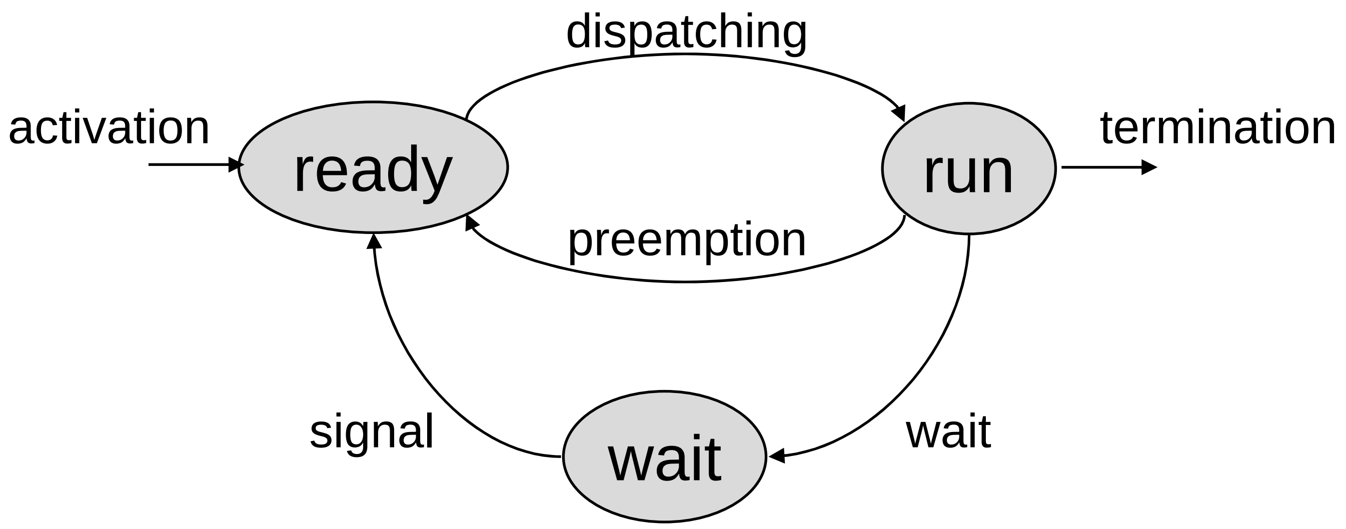
Resource Sharing

- ***Examples*** of common resources: data structures, variables, main memory area, file, set of registers, I/O unit, … .
- Many shared resources do not allow simultaneous accesses but require ***mutual exclusion*** (***exclusive resources***). A piece of code executed under mutual exclusion constraints is called a ***critical section***.



Terms

- A task waiting for an exclusive resource is said to be ***blocked*** on that resource. Otherwise, it proceeds by entering the ***critical section*** and ***holds*** the resource. When a task leaves a critical section, the associated resource becomes ***free***.
- Waiting state caused by resource constraints:



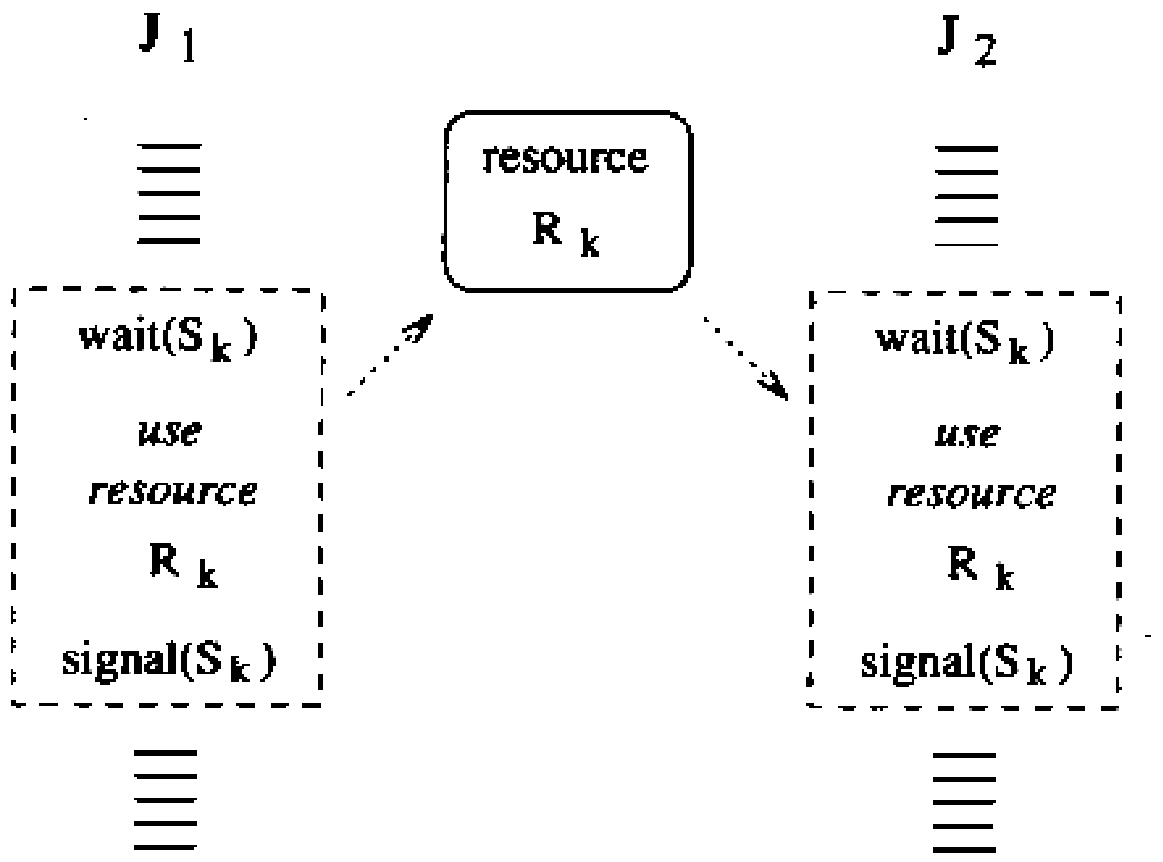
Terms

- Each ***exclusive resource*** *R_i* must be protected by a different ***semaphore*** *S_i* and each critical section operating on a resource must begin with a *wait(S_i)* primitive and end with a *signal(S_i)* primitive.

- All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a ***wait*** on a ***locked semaphore***, it enters a ***waiting state***, until another tasks executes a ***signal*** primitive that ***unlocks the semaphore***.

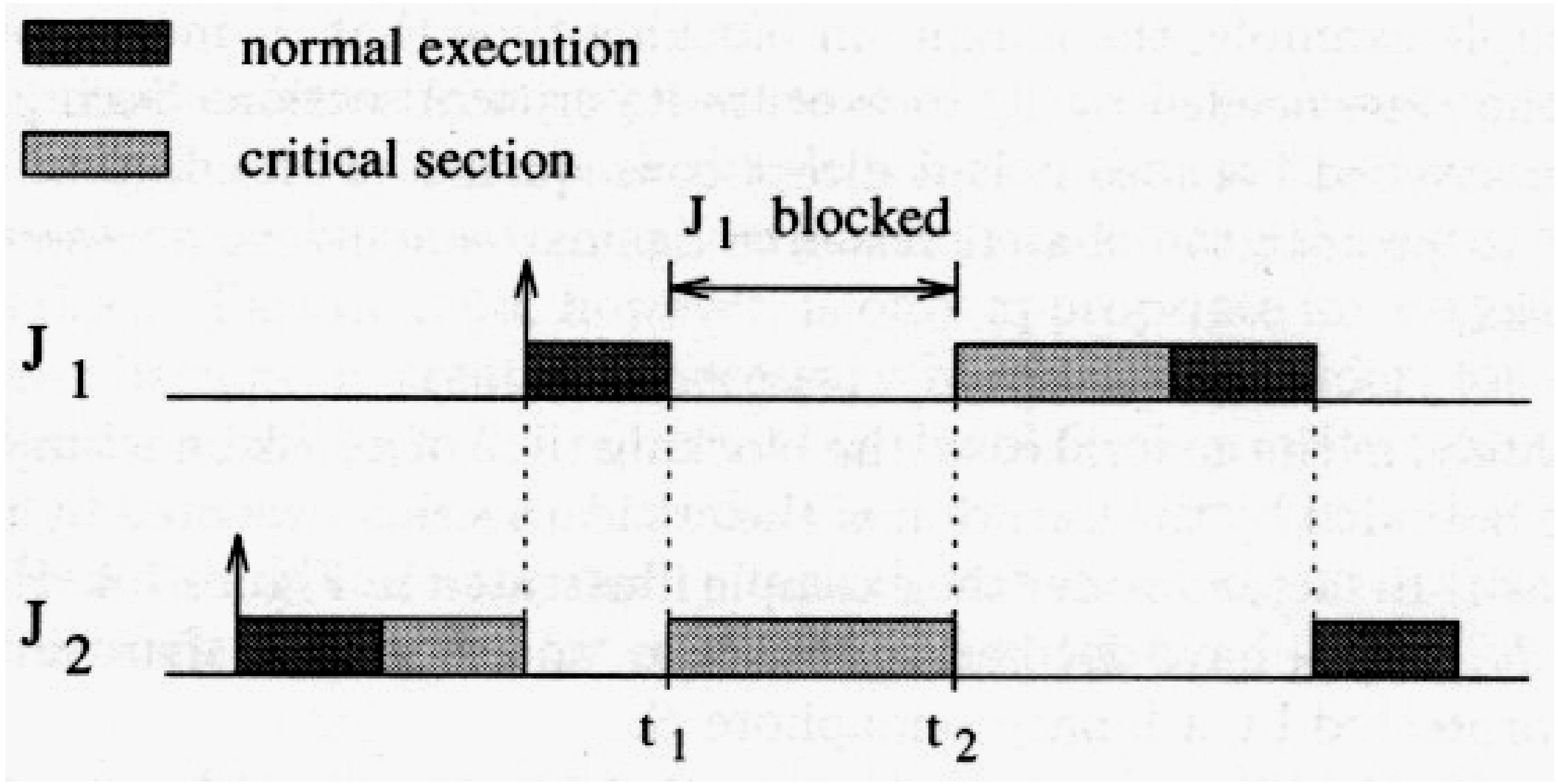
Blocking on an exclusive resource

- Software structure

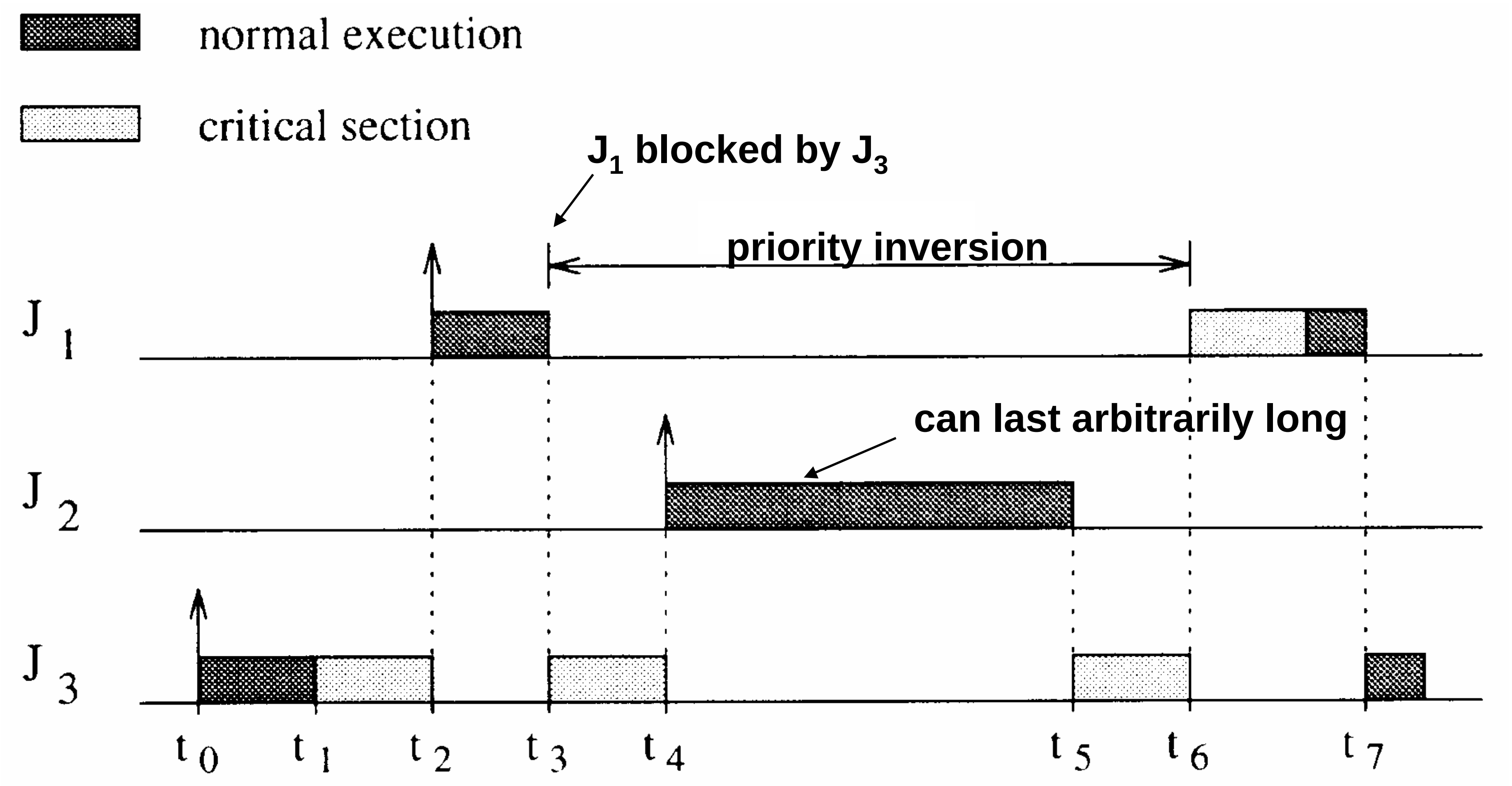


Priority Inversion (1)

- Unavoidable blocking



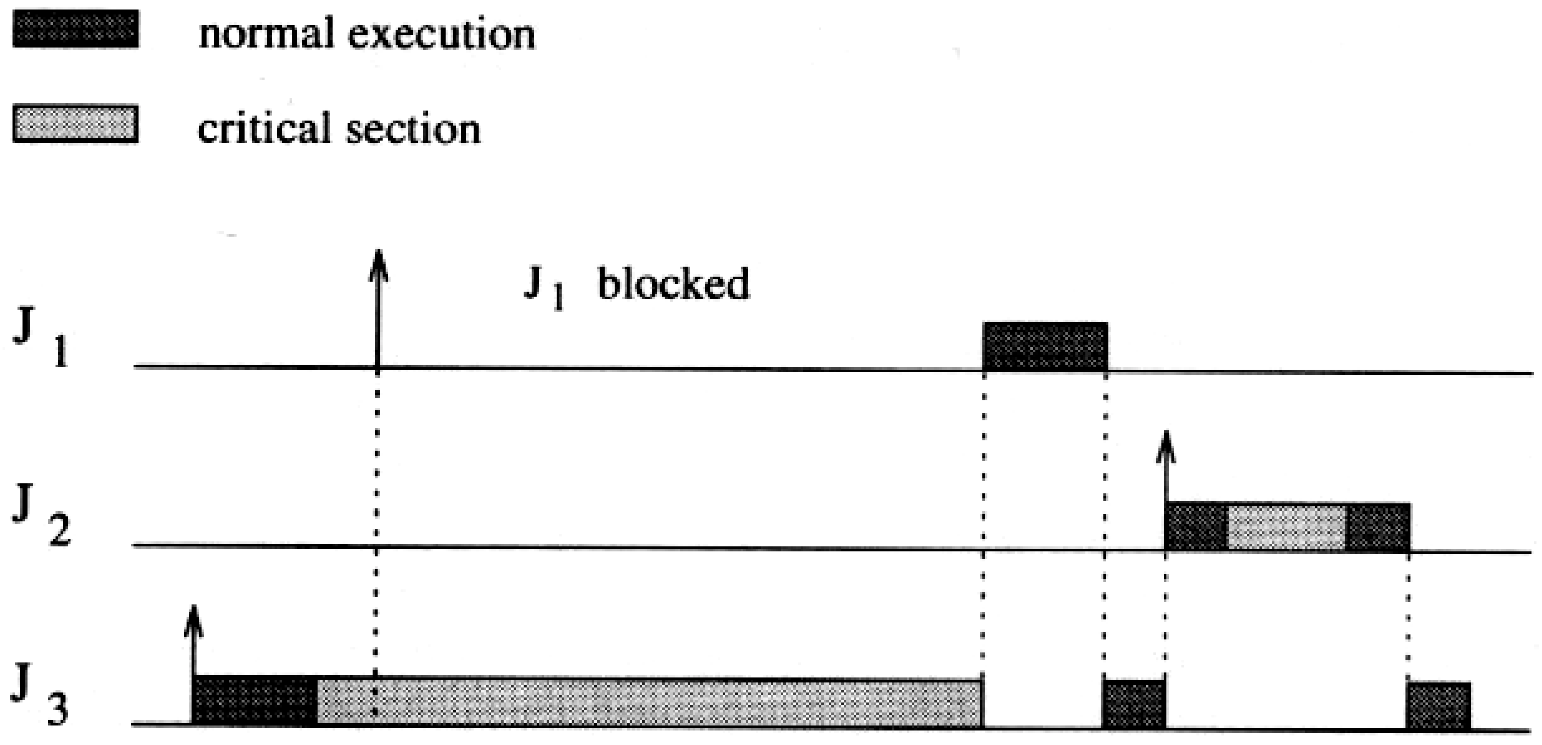
Priority Inversion (2)



[But97, S.184]

Solutions

- ***Disallow preemption*** during the execution of all critical sections. Simple, but creates unnecessary blocking as unrelated tasks may be blocked.



Resource Access Protocols

- ***Basic idea***: Modify the priority of those tasks that cause blocking. When a task J_i blocks one or more higher priority tasks, it temporarily assumes a higher priority.

- ***Methods***:
  - ***Priority Inheritance Protocol*** (PIP), for static priorities
  - Priority Ceiling Protocol (PCP), for static priorities
  - Stack Resource Policy (SRP),
    for static and dynamic priorities
  - others …

Priority Inheritance Protocol (PIP)

- ***Assumptions***:

  n periodic tasks which cooperate through m shared resources; fixed priorities, deadlines equal periods, all critical sections on a resource begin with a *wait*(S_i) and end with a *signal*(S_i) operation.

- ***Basic idea***:

  When a task $\boldsymbol{J_i}$ blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

- ***Terms***:

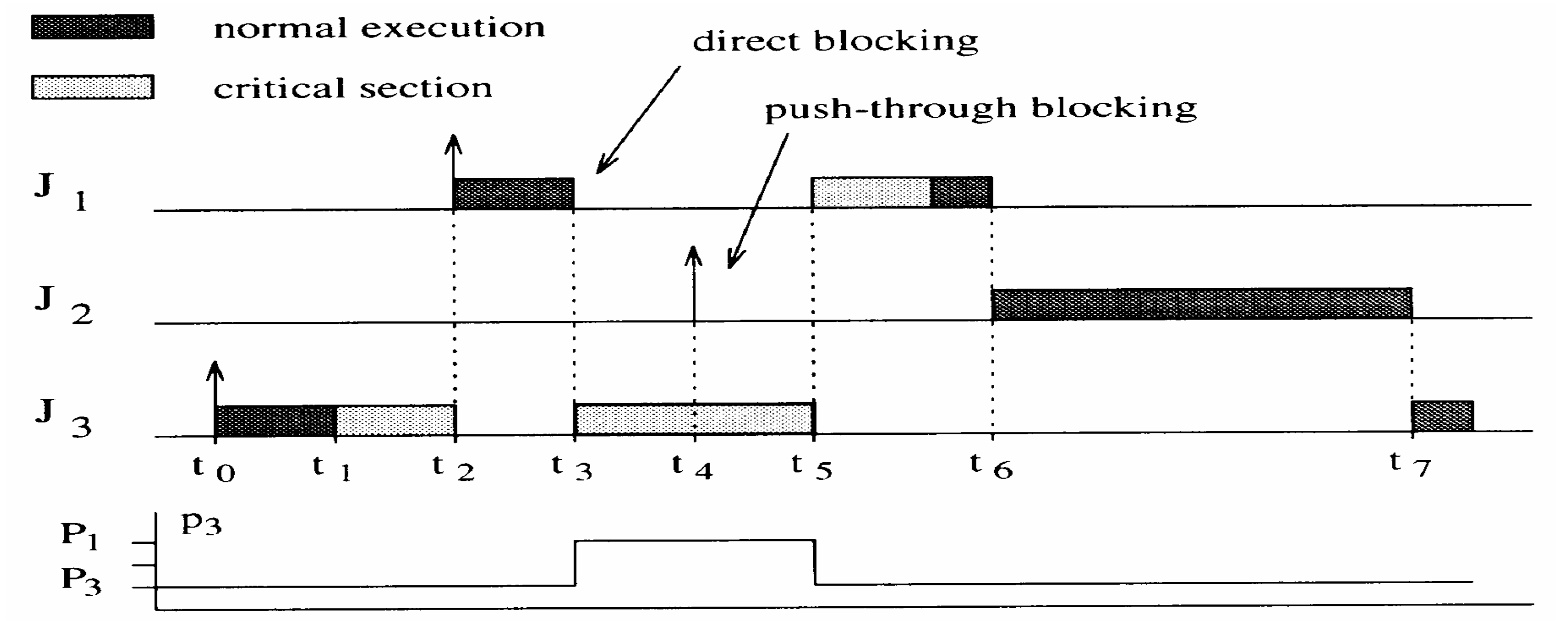
  We distinguish a fixed ***nominal priority*** P_i and an ***active priority*** p_i larger or equal to P_i. Jobs $J_1, \ldots J_n$ are ordered with respect to nominal priority where J_1 has ***highest priority***. Jobs do not suspend themselves.

Priority Inheritance Protocol (PIP)

- ***Algorithm***:
  - Jobs are scheduled based on their ***active priorities***. Jobs with the same priority are executed in a FCFS discipline.
  - When a job J_i tries to ***enter a critical section*** and the resource is blocked by a lower priority job, the job J_i is blocked. Otherwise it enters the critical section.
  - When a job J_i is ***blocked***, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k=p_i$ (it ***inherits*** the priority of the highest priority of the jobs blocked by it).
  - When J_k exits a critical section, it ***unlocks*** the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k, then p_k is set to P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.
  - Priority inheritance is ***transitive***, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

Priority Inheritance Protocol (PIP)

- Example:



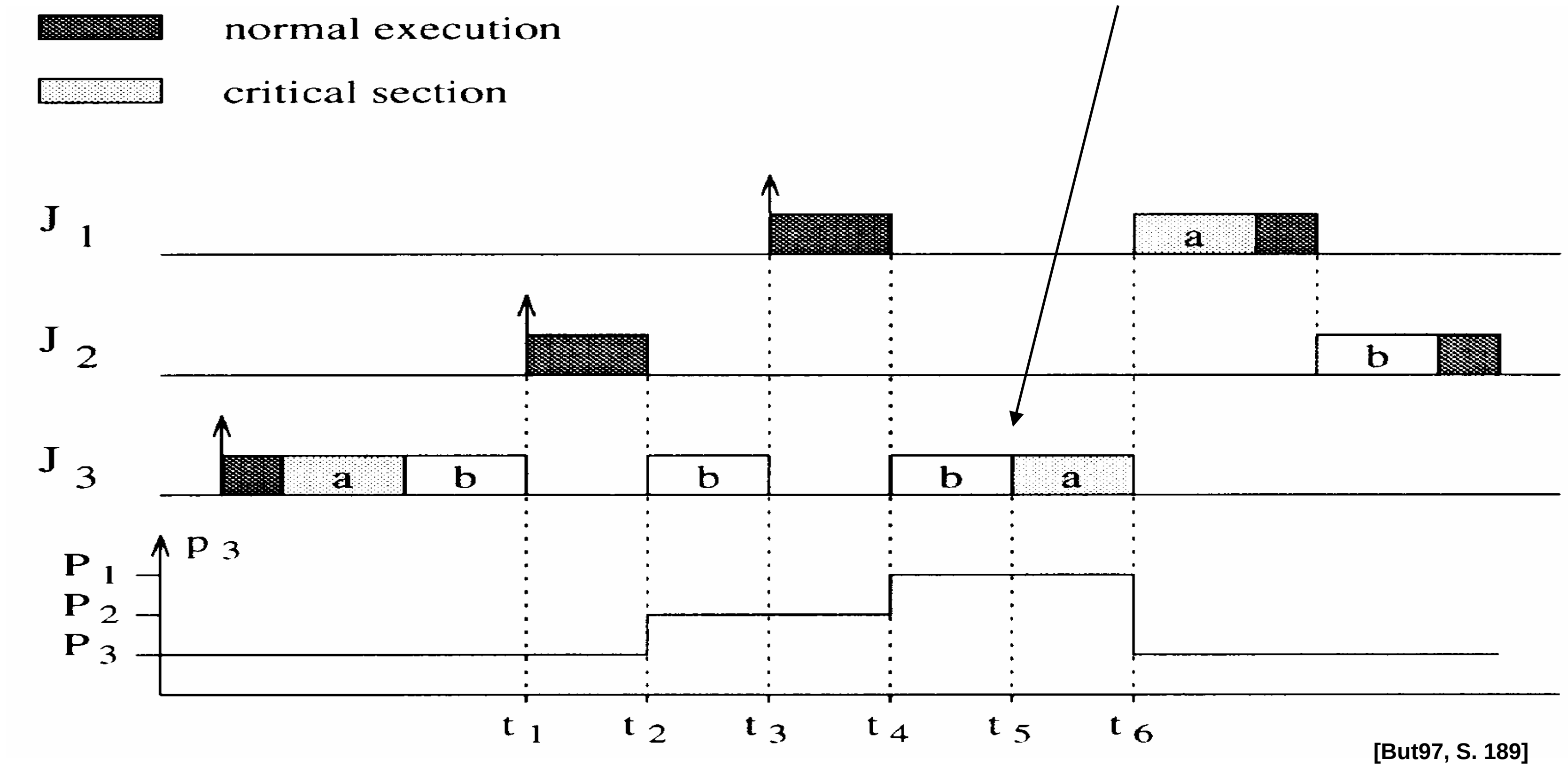
***Direct Blocking*: higher-priority job tries to acquire a resource held by a lower-priority job**

***Push-through Blocking*: medium-priority job is blocked by a lower-priority job that has inherited a higher priority form a job it directly blocks**

Priority Inheritance Protocol (PIP)

- Example with nested critical sections:

priority does not change

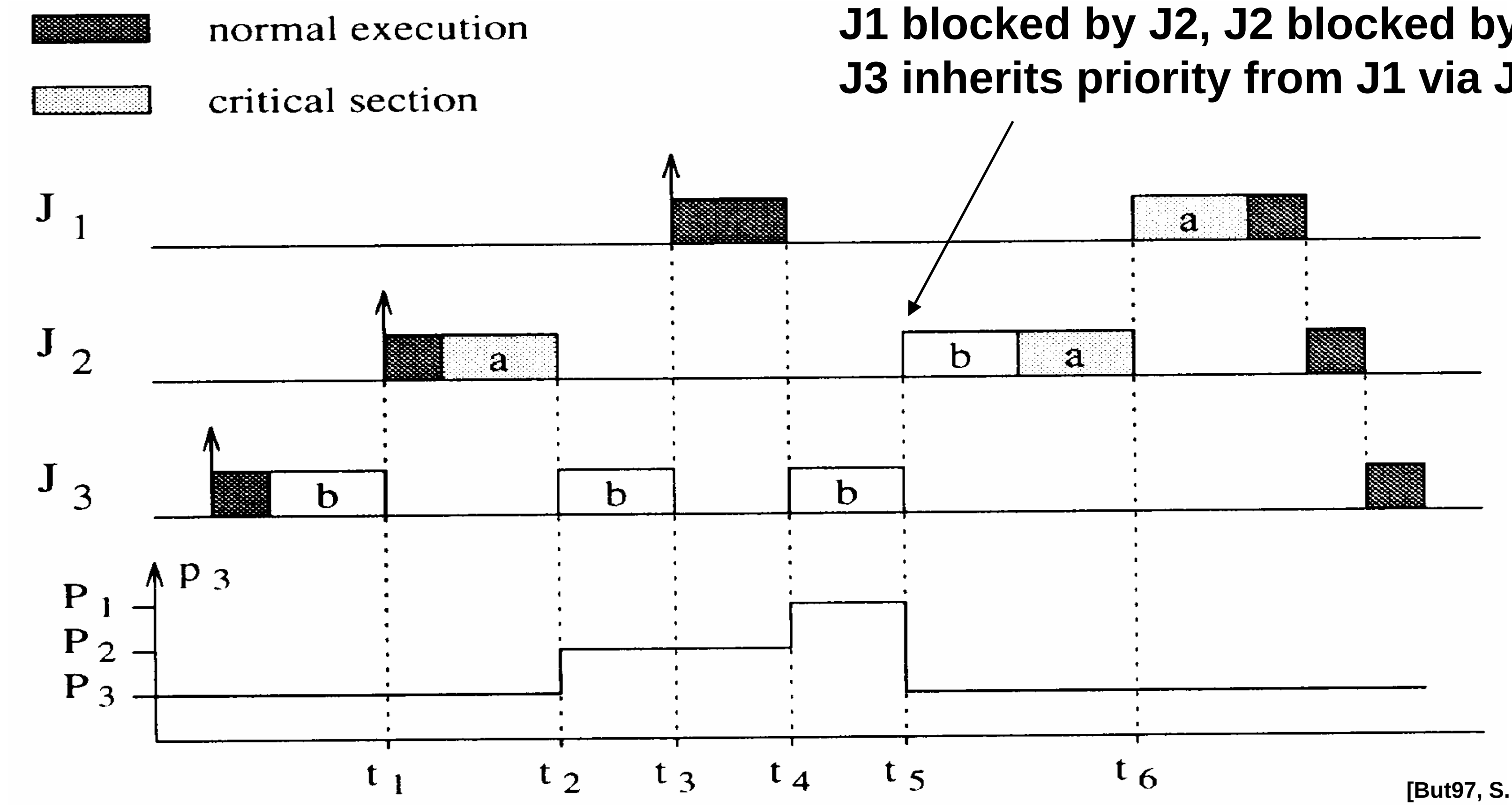


[But97, S. 189]

Priority Inheritance Protocol (PIP)

- Example of transitive priority inheritance:

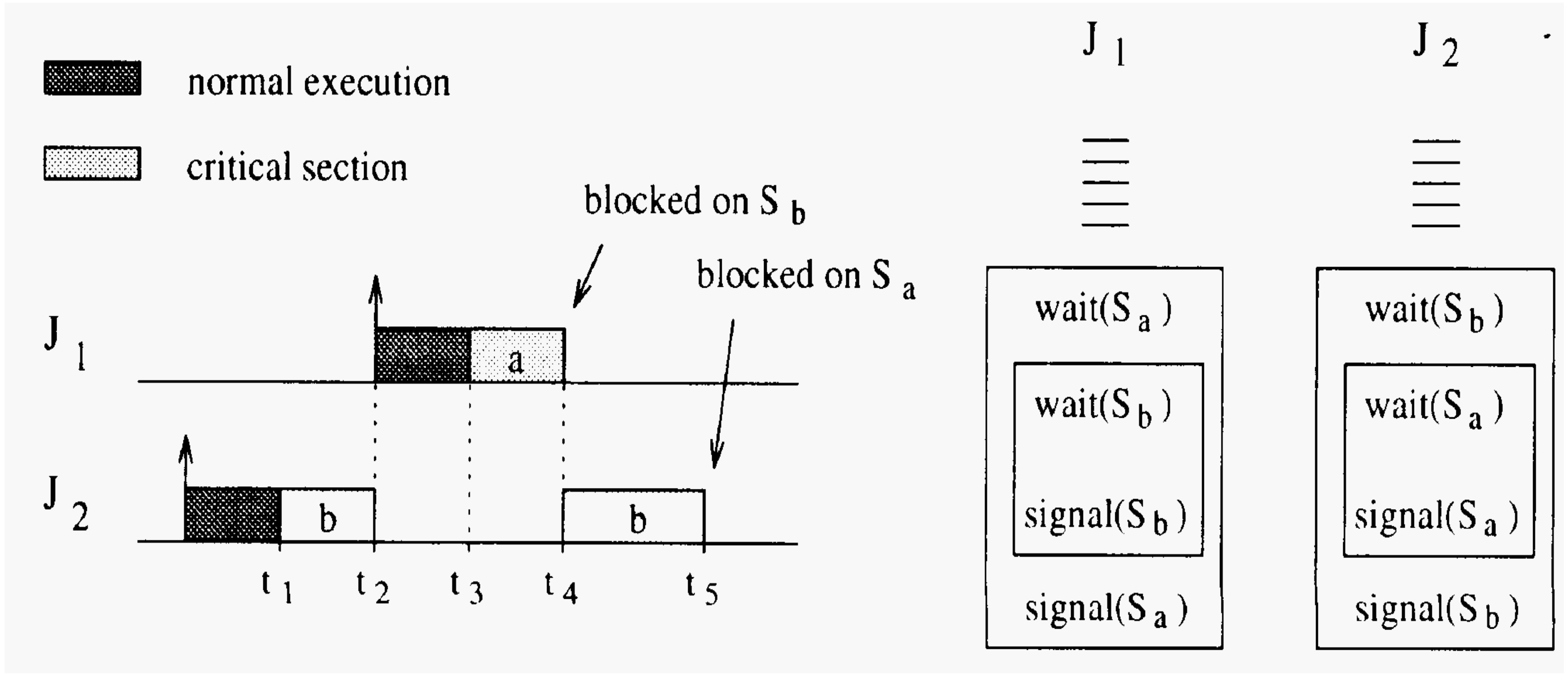
J1 blocked by J2, J2 blocked by J3.
J3 inherits priority from J1 via J2.



[But97, S. 190]

Priority Inheritance Protocol (PIP)

- Problem: ***Deadlock***



```
J1              J2
wait(Sa)        wait(Sb)
  wait(Sb)        wait(Sa)

  signal(Sb)      signal(Sa)
signal(Sa)      signal(Sb)
```

[But97, S. 200]