Definition of Scheduling Problem (Section 2.3)

- A scheduling problem is defined using 3 sets:
  - a set of n tasks Γ = {τ1, τ2, . . . , τn},
  - a set of m processors P = {P1, P2, . . . , Pm} and
  - a set of s types of resources R = {R1,R2, . . . , Rs}.
  - In addition, precedence relations among tasks can be specified through a directed acyclic graph, and timing constraints can be associated with each task.
- In this context, 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 and as a result computationally intractable

Complexity of Scheduling Algorithms

- In dynamic real time systems, scheduling decisions must be taken on line 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 polynomial way is called an exponential time algorithm

Example of Polynomial and Exponential Time Algorithms

- Let us consider two algorithms with complexity functions n and 5^n, respectively, and let us assume that an elementary step for these algorithms lasts 1 μs.
- If the input length of the instance is $n = 30$:
  - Polynomial algorithm solved the problem in 30 μs
  - Exponential algorithm needs about $3 \cdot 10^5$ centuries to solve the problem

Real-Time Scheduling Objective

- One of the research objectives in real-time scheduling is to identify simpler, but still practical, problems that can be solved in polynomial time.
- Some possible steps to reduce the complexity of constructing a feasible schedule are:
  - simplify the computer architecture (for example, by restricting to the case of uniprocessor system),
  - adopt a preemptive model,
  - use fixed priorities,
  - remove precedence and/or resource constraints,
  - assume simultaneous task activation,
  - Assume homogeneous task sets (solely periodic or solely aperiodic activities),
  - etc.

Classification of Scheduling Algorithms

Main classes 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

Classification of Scheduling Algorithms

Main classes of scheduling algorithms:

- 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

Main classes of scheduling algorithms:

- 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.

Classification of Scheduling Algorithms

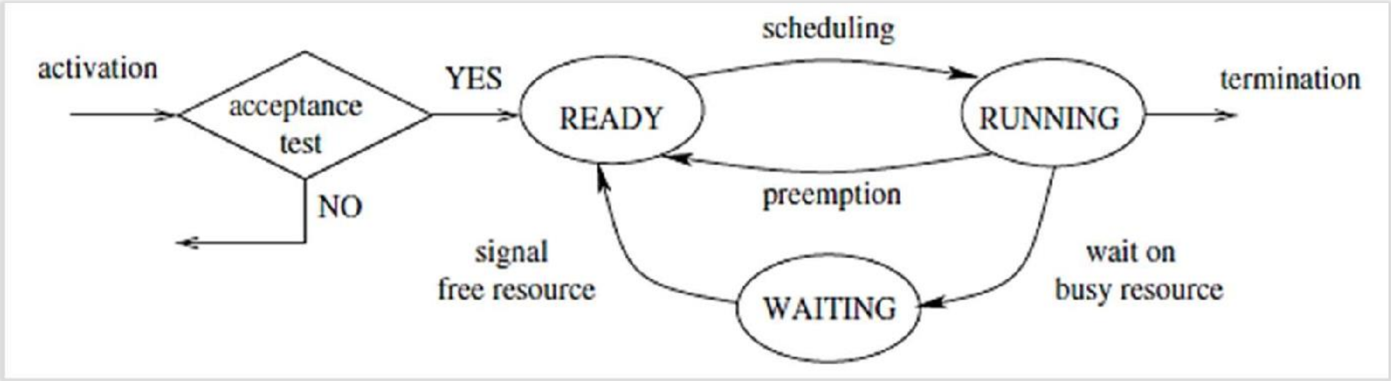
Main classes of scheduling algorithms:

- Optimal vs. Heuristic.
- An algorithm is said to be 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 said to be optimal if it is able to find a feasible schedule, if one exists.
- An algorithm is said to be 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; that is, before task execution
- The system must plan its actions by looking ahead in the future and by 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 off line, and the entire schedule can be stored in a table that contains all guaranteed tasks arranged in the proper order
- In **dynamic real-time systems** (typically consisting of firm tasks), tasks can be created at runtime; hence the guarantee must be done online every time a new task is created

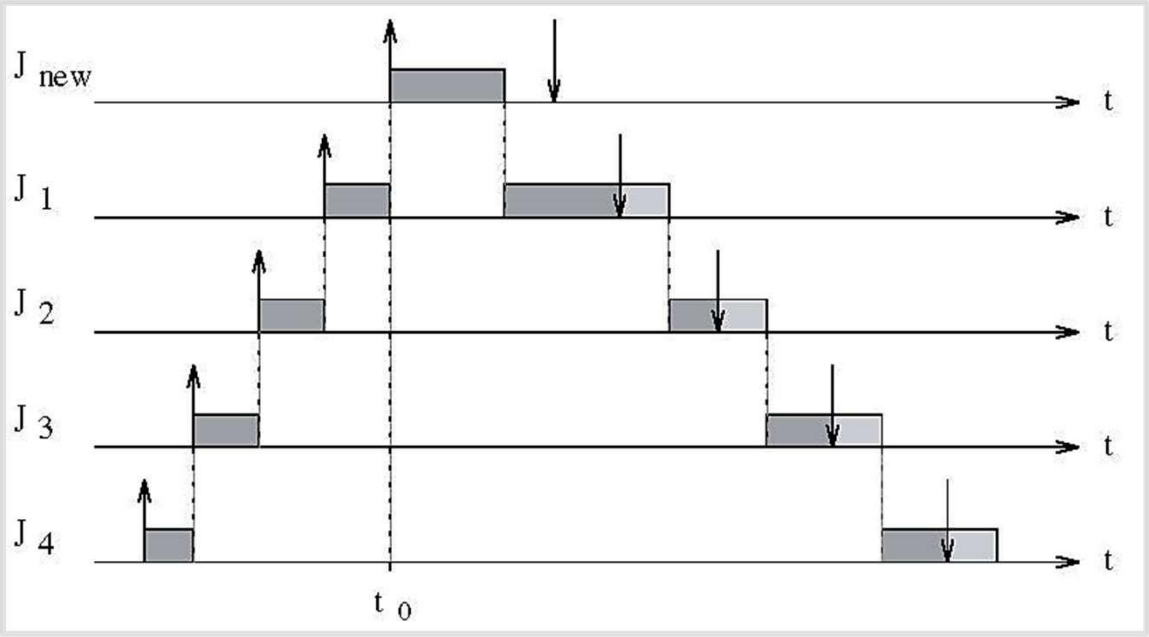
Scheme of Guarantee Mechanism for Dynamic Systems, Online Guarantee Test



- If Γ is the current task set that has been previously guaranteed, a newly arrived task τ_{new} is accepted into the system if and only if the task set Γ' = Γ∪ {τ_{new}} is found schedulable.
- If Γ' is not schedulable, then task τ_{new} is rejected to preserve the feasibility of the current task set
- Implementing a guaranteed mechanism will deter potential system overload situations, such as transient overload, and avoid negative effects

Transient Overload

- One problematic scenario caused by transient overload is called domino effect:
  - The arrival of a new task causes all previously guaranteed tasks to miss their deadlines.



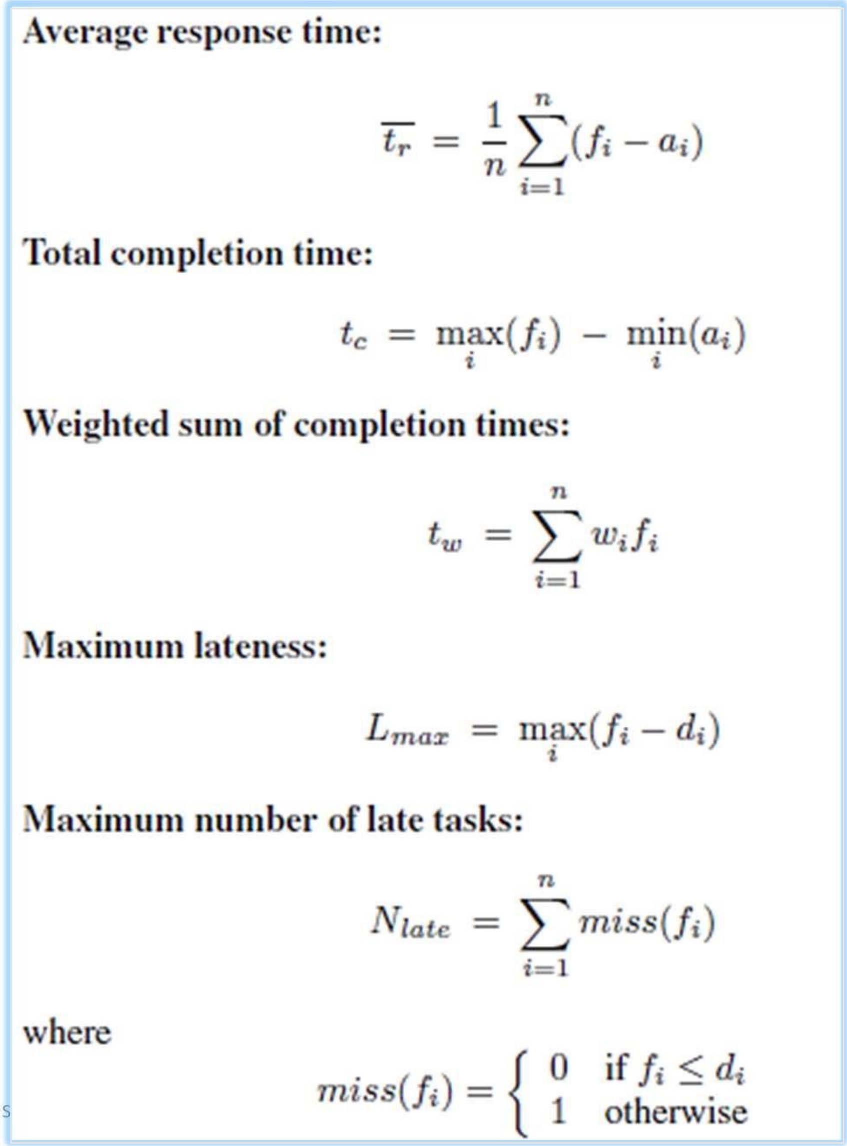
A guaranteed mechanism will allow us to detect these potential overload situations

Best-Effort Algorithms

- To efficiently support soft real-time applications that do not have hard timing requirements, a best-effort approach may be adopted for scheduling
- A best-effort scheduling algorithm tries to "do its best" to meet deadlines, but there is no guarantee of finding a feasible schedule
- In best-effort algorithms a task is aborted **only under real overload conditions**; as a result these algorithms could perform better than guarantee-based schemes

Metrics for Performance Evaluation

- Evaluating the performance of a scheduling algorithms – typically done through a cost function defined over the task set
  - Classical scheduling algorithms try to minimize functions such as: →
- The metric used in the scheduling algorithm has implications on the performance of the real-time system



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

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

Lateness Criteria

- Minimizing the **maximum lateness** can be useful
- But, in general, minimizing the maximum lateness does not minimize the number of tasks that miss their deadlines and does not necessarily prevent one or more tasks from missing their deadline
- See example

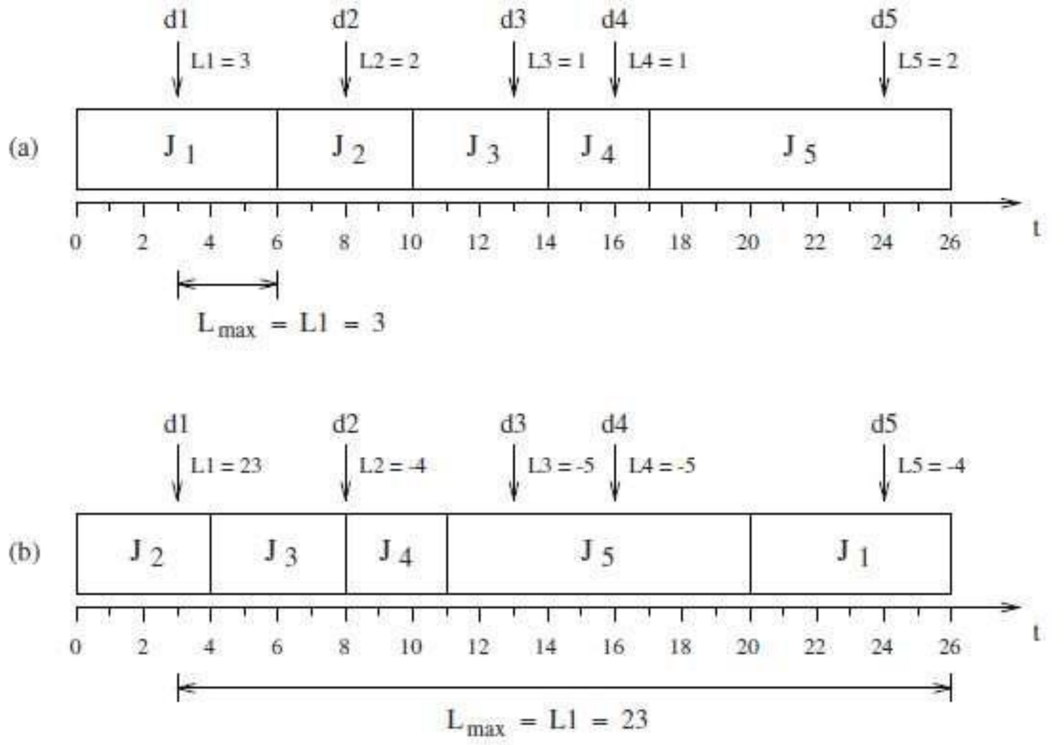
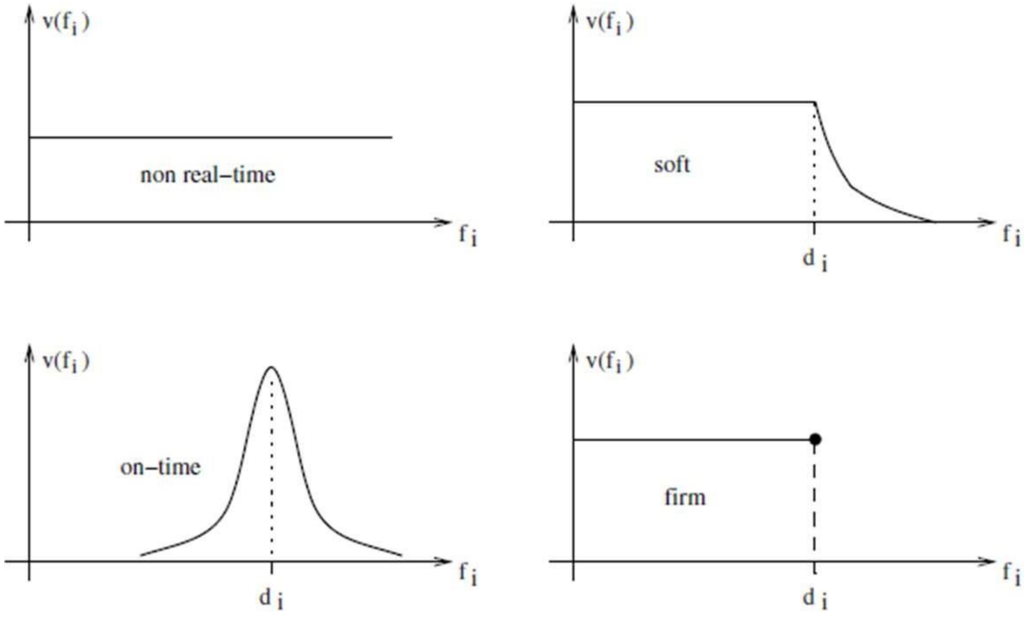


Figure 2.16 The schedule in (a) minimizes the maximum lateness, but all tasks miss their deadline. The schedule in (b) has a greater maximum lateness, but four tasks out of five complete before their deadline.

Typical Utility Functions

- Utility function – will describe the benefit of **executing a task related to its completion time**
- When utility functions are defined on tasks, the performance of the scheduling algorithm can be **measured by the cumulative value**



$$Cumulative\_value = \sum_{i=1}^{n} v(f_i).$$

Scheduling Anomalies

- We will just enumerate some scheduling anomalies that you should be aware about:
    1. If tasks have deadlines, **then adding resources** (for example, an extra processor) **or relaxing constraints** (less precedence among tasks or fewer execution times requirements) can actually make things worse
    2. **Reducing the computation time of each task** will not necessarily improve the global completion time
    3. **Weakening precedence constraints** will not necessarily improve the global completion time
    4. In the presence of shared resources**, the schedule length of a task set can increase** when reducing tasks' computation times
    5. **Increasing the computation speed of tasks in a non-preemptive kernel** will not necessarily optimize the global completion time.

Homework

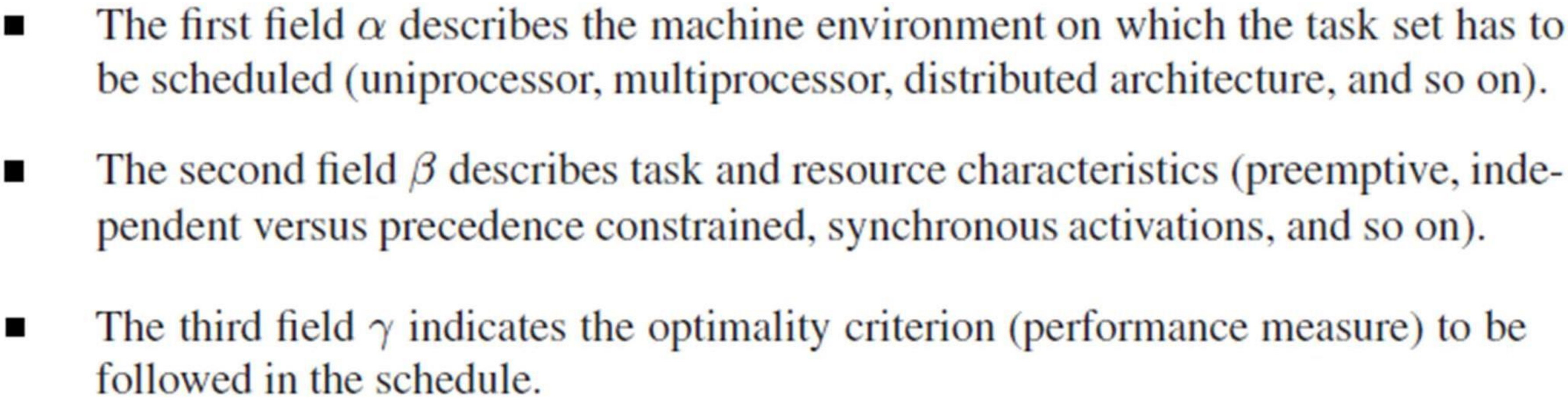
- Exercises 2.1 to 2.5 on page 51 in the textbook
- Solutions are presented in Chapter 13 of the book

Aperiodic Task Scheduling (Chapter 3)

- We present various algorithms for scheduling real-time aperiodic tasks on a single machine environment
- Each algorithm represents a solution for a particular scheduling problem, which is expressed through a set of assumptions on the task set and by an optimality criterion to be used on the schedule
- Many of these algorithms can be extended to work on multi-processor or distributed architectures
- Algorithms presented in this section are:
  - Jackson's algorithm
  - Horn's algorithm
  - Non-preemptive scheduling algorithms
  - Scheduling algorithms with precedence constraints

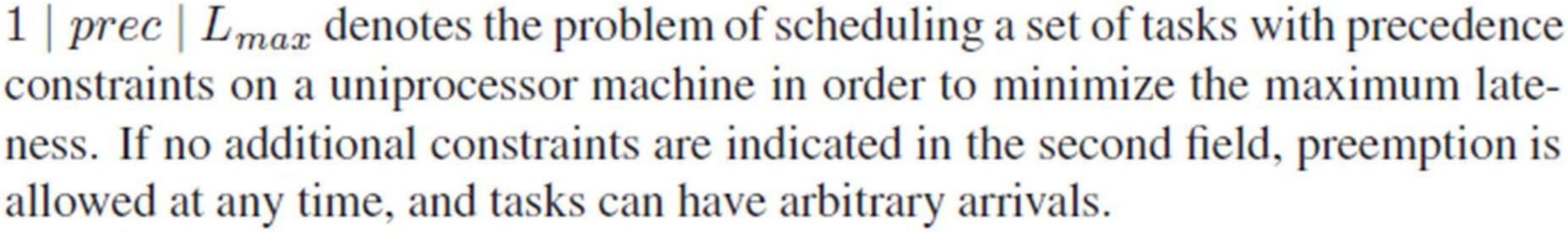
Systematic Notation

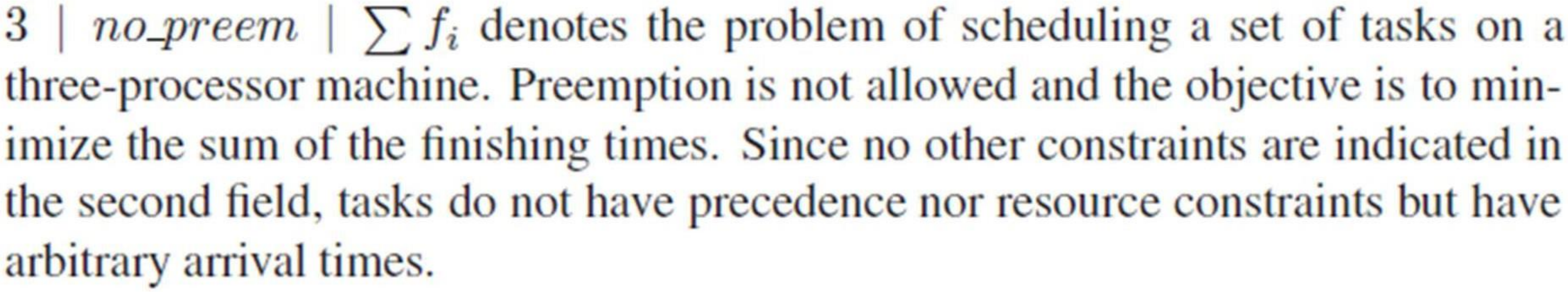
To facilitate the description of the scheduling problems presented we introduce a systematic notation which classifies all algorithms using three fields α | β | γ, having the following meaning:

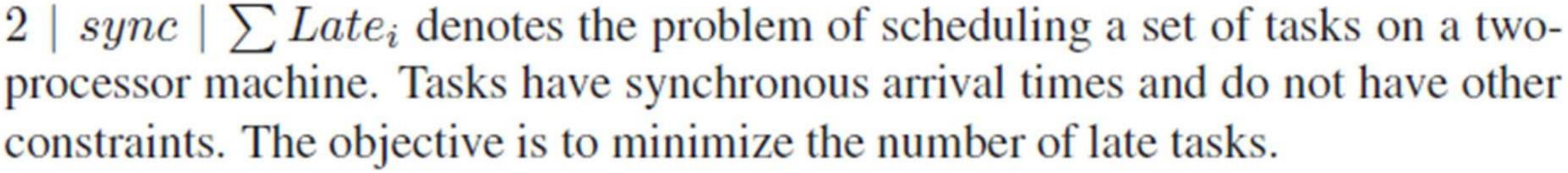


- The first field α describes the machine environment on which the task set has to be scheduled (uniprocessor, multiprocessor, distributed architecture, and so on).
- The second field β describes task and resource characteristics (preemptive, independent versus precedence constrained, synchronous activations, and so on).
- The third field γ indicates the optimality criterion (performance measure) to be followed in the schedule.

Notation Examples

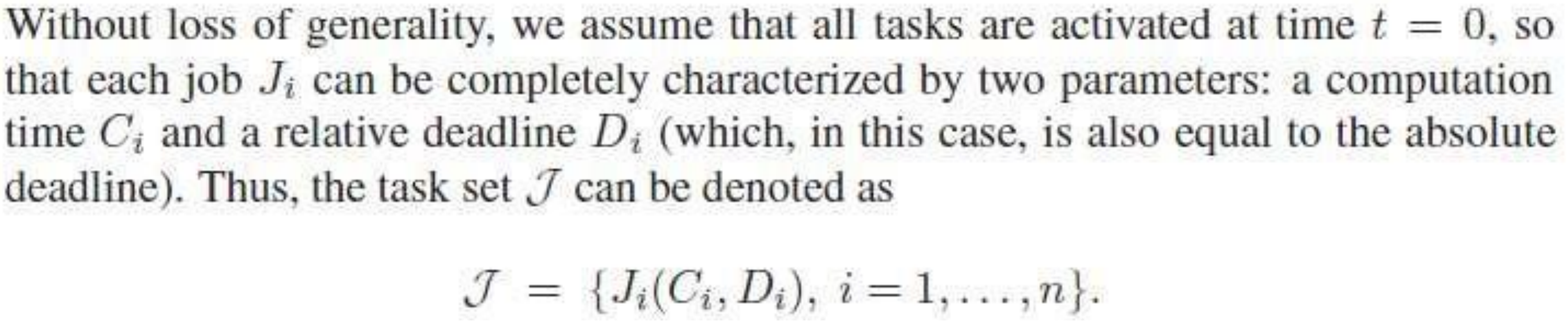
- 
  $1 \mid prec \mid L_{max}$ denotes the problem of scheduling a set of tasks with precedence constraints on a uniprocessor machine in order to minimize the maximum lateness. If no additional constraints are indicated in the second field, preemption is allowed at any time, and tasks can have arbitrary arrivals.

- 
  $3 \mid no\_preem \mid \sum f_i$ denotes the problem of scheduling a set of tasks on a three-processor machine. Preemption is not allowed and the objective is to minimize the sum of the finishing times. Since no other constraints are indicated in the second field, tasks do not have precedence nor resource constraints but have arbitrary arrival times.

- 
  $2 \mid sync \mid \sum Late_i$ denotes the problem of scheduling a set of tasks on a two-processor machine. Tasks have synchronous arrival times and do not have other constraints. The objective is to minimize the number of late tasks.

Scheduling Problem

- The problem considered by this algorithm is 1 | sync | L_{max}.
  - All tasks consist of a single job, have synchronous arrival times, but can have different computation times and deadlines.
  - Also, task must be independent.



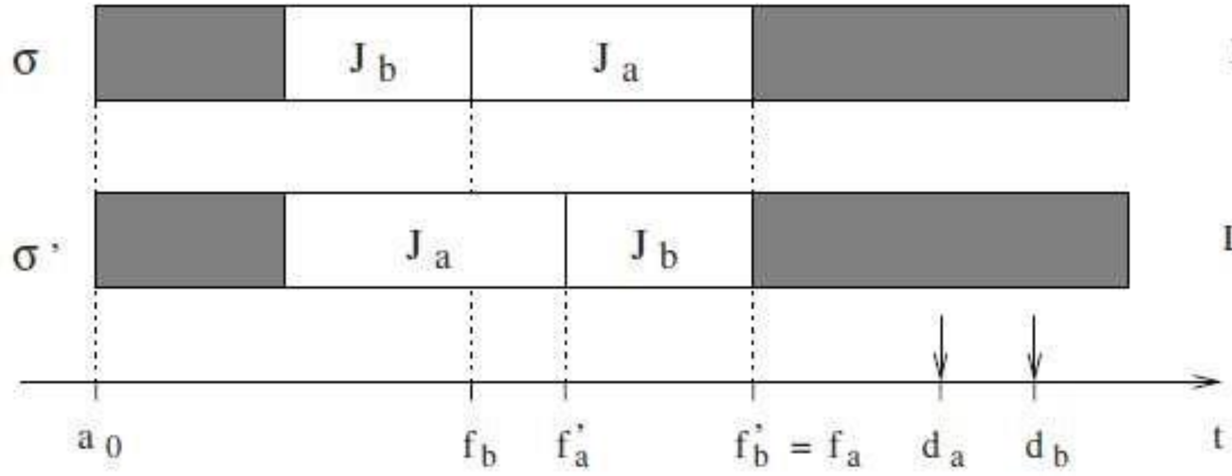
Without loss of generality, we assume that all tasks are activated at time $t = 0$, so that each job J_i can be completely characterized by two parameters: a computation time C_i and a relative deadline D_i (which, in this case, is also equal to the absolute deadline). Thus, the task set $\mathcal{J}$ can be denoted as

$$\mathcal{J} = \{J_i(C_i, D_i),\ i = 1, \ldots, n\}.$$

Jackson’s Algorithm

- A simple algorithm that solves this problem was found by Jackson in 1955. It is called Earliest Due Date (EDD) and can be expressed by the following rule:
- Theorem (Jackson’s rule): Given a set of n independent tasks, any algorithm that executes the tasks in order of nondecreasing deadlines is optimal with respect to minimizing the maximum lateness.
- The complexity required by Jackson’s algorithm to build the optimal schedule is due to the procedure that sorts the tasks by increasing deadlines.
    - As a result, if the task set consists of n tasks, the complexity of the EDD algorithm is O(n log n).

Proof of theorem: … in class work …



$L_{\underset{ab}{\max}} = f_a - d_a$

$L'_{\underset{ab}{\max}} = \max(L'_a, L'_b)$

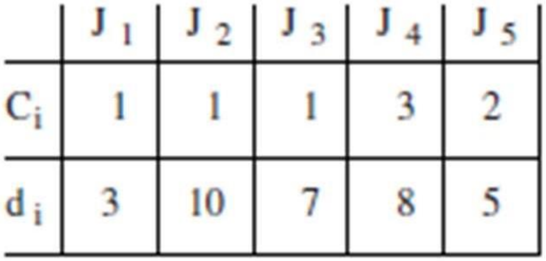
if $(L'_a \geq L'_b)$ then $L'_{\underset{ab}{\max}} = f'_a - d_a < f_a - d_a$

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

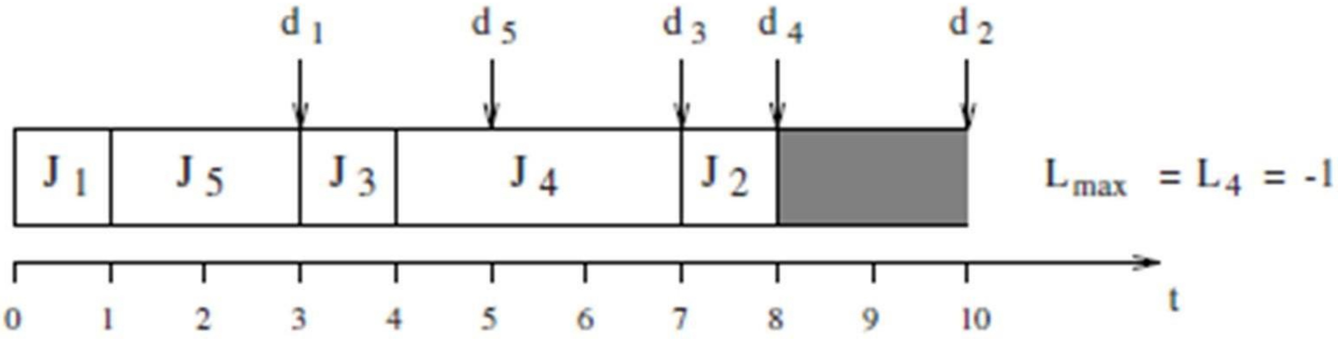
in both cases: $L'_{\underset{ab}{\max}} < L_{\underset{ab}{\max}}$

Example 1

Consider a set of 5 tasks, simultaneously activated at time t = 0; …



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

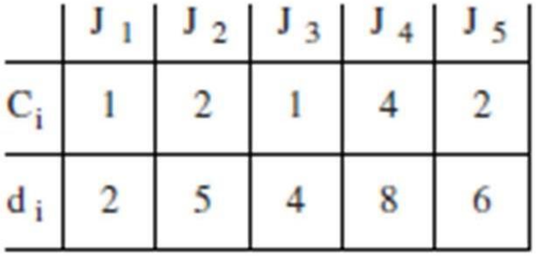


EDD Optimality and Task Schedule Feasibility

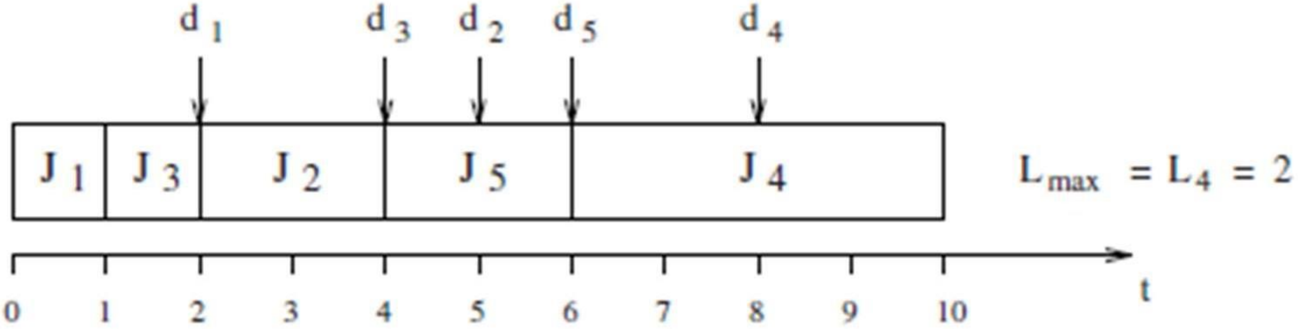
- Note that the optimality of the EDD algorithm cannot guarantee the feasibility of the schedule for any task set.
- EDD only guarantees that if a feasible schedule exists for a task set, then it will find it.

Example 2

This example task set cannot be feasibly scheduled …



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



EDD Schedulability Guarantee (See pp. 58)

- To guarantee that a set of task can be feasibly scheduled by EDD algorithm we need to show that in the worst-case scenario all tasks can complete before their deadlines:

  $\forall i = 1, . . . , n \quad f_i \leq d_i$

- If fact, for a random set of n tasks we can assume that the tasks are denoted by their increasing deadlines starting with

  $J_1, J_2, \ldots, J_n$; such that: $d_1 < d_2 < \ldots < d_n$

- Then, the guarantee test can be performed by verifying the n conditions for the worst-case finishing times of all tasks:

  $\forall \mathrm{i} = 1, . . . , \mathrm{n}; \quad \sum_{\mathrm{k}=1}^{\mathrm{i}} C_{\mathrm{k}} \leq d_{\mathrm{i}}$

Early Deadline First (EDF)/Horn's Algorithm

- Horn algorithm: solves the problem of scheduling a set of n independent tasks on a uniprocessor system, when tasks may have dynamic arrivals and preemption is allowed (1 | preem | L_{max}) – the algorithm is also called Earliest Deadline First (EDF).
- Theorem (Horn): 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 all the ready tasks is optimal with respect to minimizing the maximum lateness L_{max}.

EDF Optimality

EDF is optimal in the sense of feasibility -- if there exists a feasible schedule for a task set J, then EDF can find it.

- The proof can easily be extended to show that EDF also minimizes the maximum lateness. Proof of optimality of the EDF algorithm, see pp. 59-61.
- For the proof we divide the schedule into time slices of one unit of time each

Notations:

σ – a schedule produced by a generic algorithm A

σ_{EDF} – the schedule obtained by the EDF algorithm

$\sigma(t)$ identifies the task executing in the time 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.

EDF Optimality Proof

- If σ ≠ σ_{EDF} , then in σ there exists a time t such that σ(t) ≠ E(t).
- The basic idea used in the proof is that interchanging the position of σ(t) and E(t), in the above situation, cannot increase the maximum lateness.
- If the schedule σ starts at time t = 0 and D is the latest deadline of the task set: $D = \max_{i} \{d_i\}$

=> σ_{EDF} can be obtained from σ by at most D transpositions

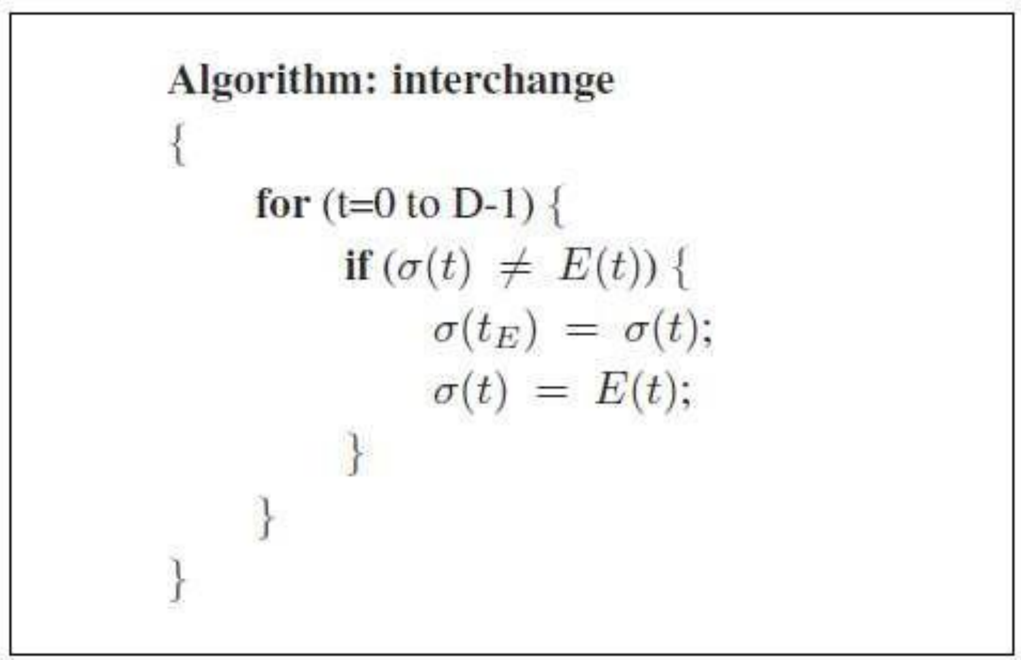
EDF Optimality Proof

For each time slice t, do:

- Is the task σ(t) scheduled in the slice t the one with the earliest deadline, E(t).
  - YES: do nothing
  - NO: exchange the slice of task E(t) as anticipated at time t and postpone the slice of task σ(t) to time t_E.

Similarly with Jackson's theorem proof, it can be shown that after each transposition the maximum lateness cannot increase;

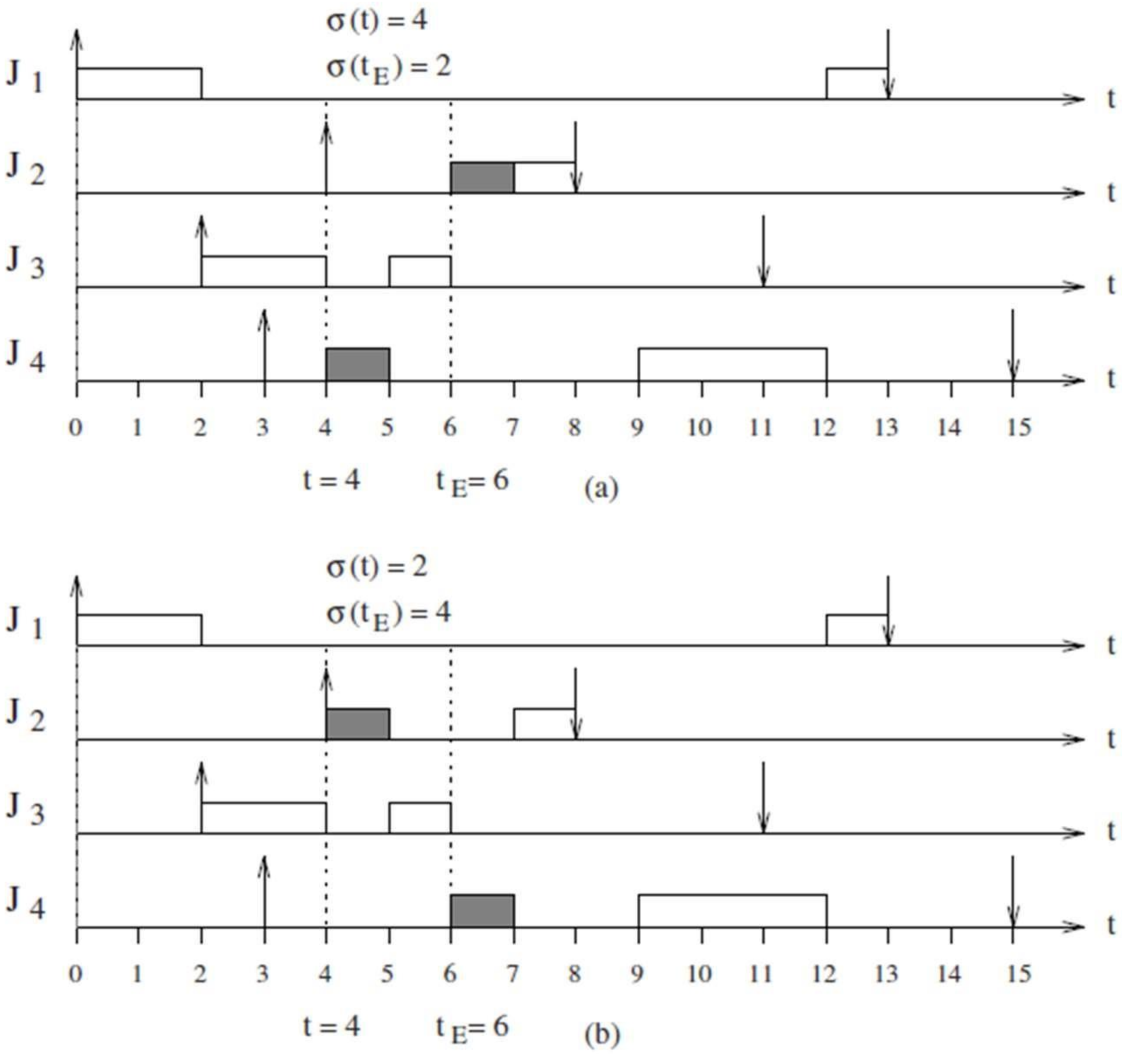
=> therefore, EDF is optimal.



```
Algorithm: interchange
{
    for (t=0 to D-1) {
        if ($\sigma(t) \neq E(t)$) {
            $\sigma(t_E) = \sigma(t);$
            $\sigma(t) = E(t);$
        }
    }
}
```

EDF Optimality Proof: Example of Applying Interchange Algorithm to a Schedule

In class work …

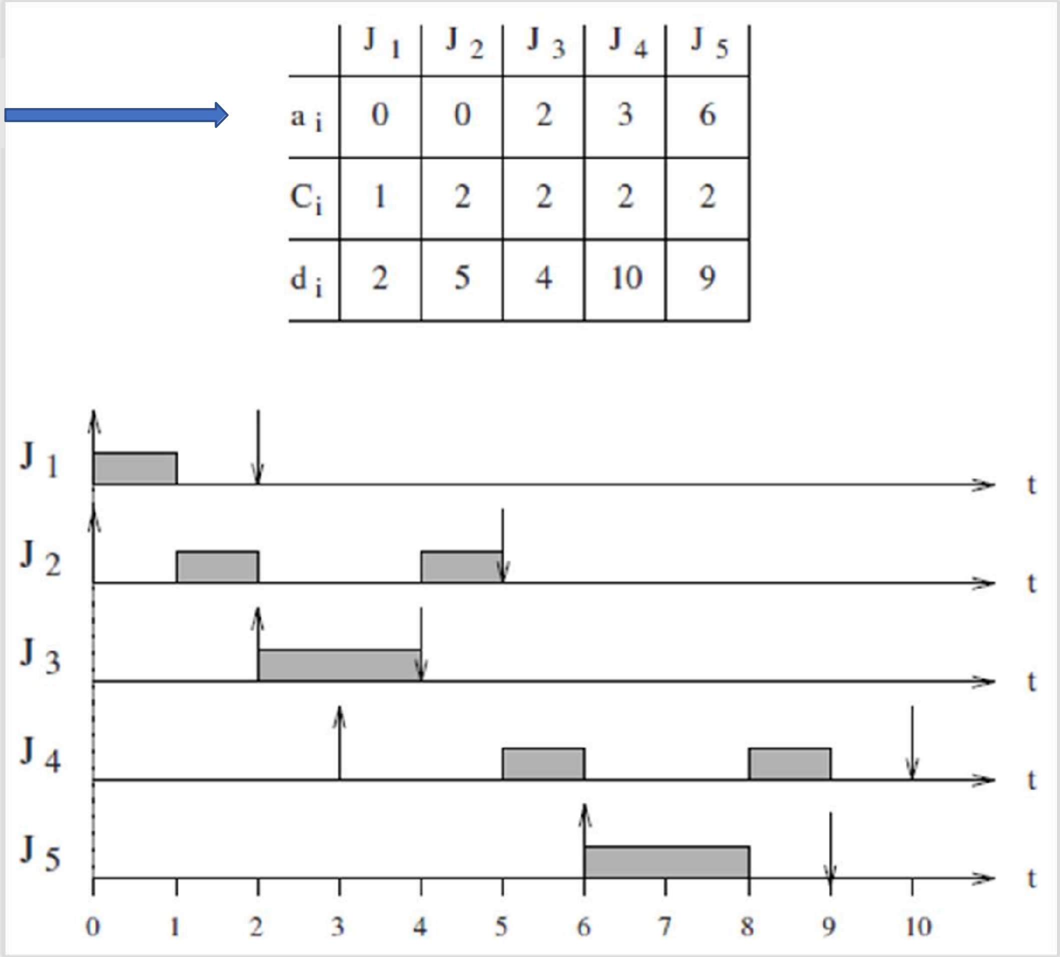


EDF Example

Explain …

5 tasks set

EDF schedule



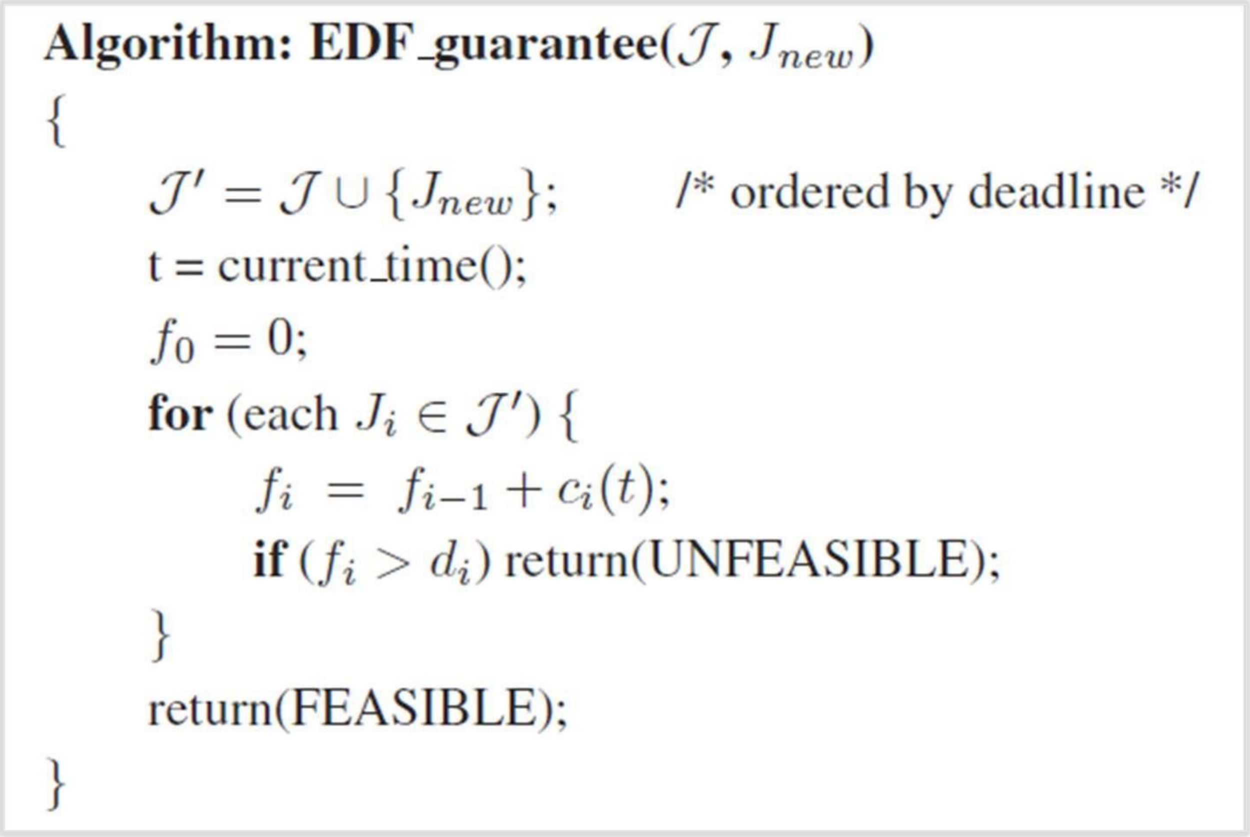
	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9

EDF Feasibility Guarantee Criteria (See pp. 63)

- When tasks have dynamic activations and the arrival times are not known a priori, the guaranteed test must be done dynamically, whenever a new task enters the system.
- Let J be the current set of active tasks, which have been previously guaranteed, and let J_{new} be a newly arrived task.
  - In order to accept J_{new} in the system we must guarantee that the new task set $J' = J \cup \{J_{new}\}$ is also schedulable.
- Without loss of generality, we can assume that all tasks in J' (including J_{new}) are ordered by increasing deadlines, so that J_1 is the task with the earliest deadline.

Guarantee Test Conditions … in class work …

EDF Guarantee Algorithm



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

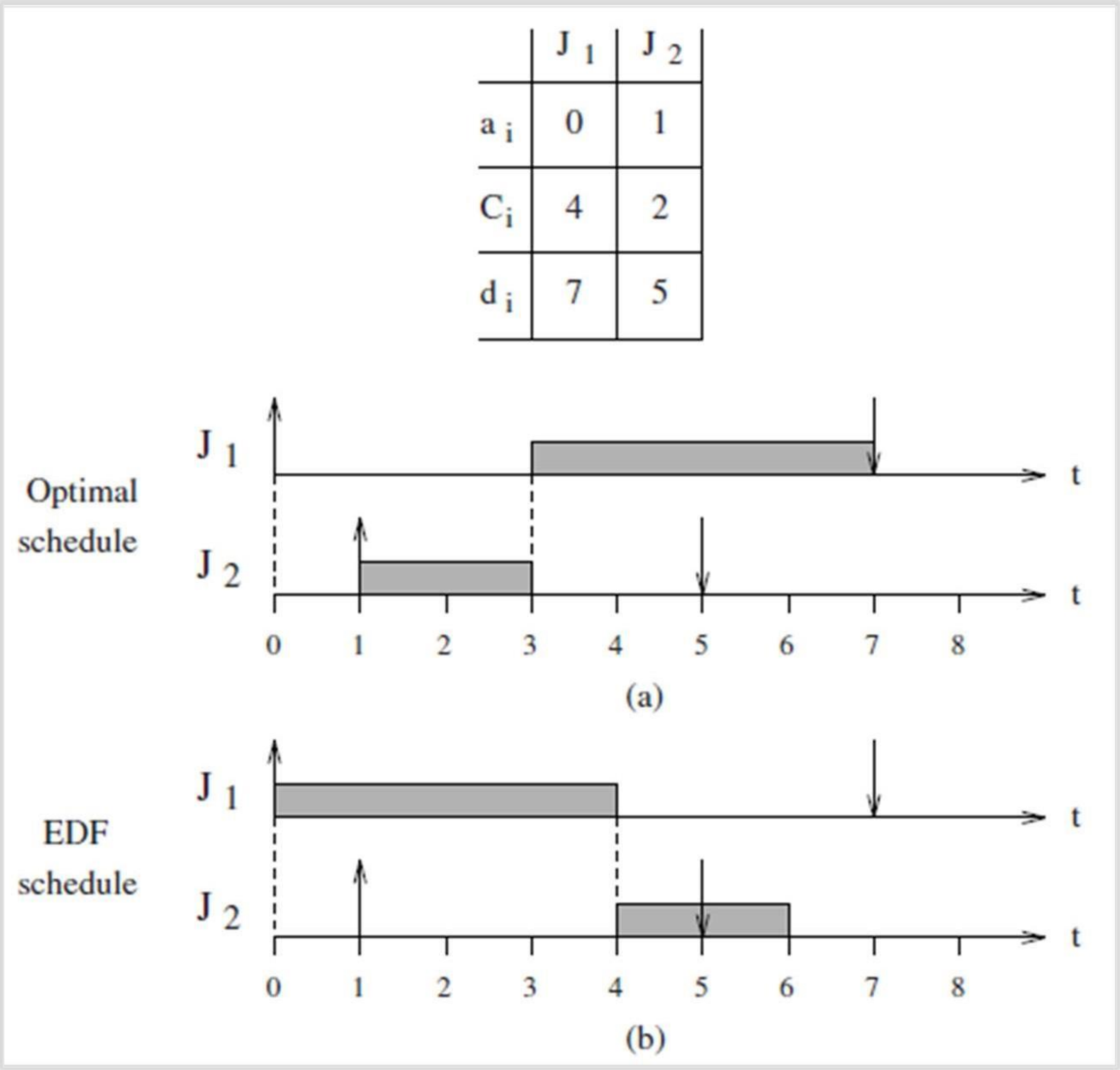
Non-Preemptive Scheduling

- When preemption is not allowed and tasks can have arbitrary arrivals, the problem of minimizing the maximum lateness and the problem of finding a feasible schedule become NP-hard
- It can be shown that EDF is no longer optimal if tasks cannot be preempted during their execution

EDF Example for Non-Preemptive Model

This example shows that EDF algorithm does not find the optimal schedule when preemption is not allowed even though a feasible schedule exists for that task set (see Figure a).

Why? …



	J_1	J_2
a_i	0	1
C_i	4	2
d_i	7	5

(a)

(b)

Non-Idle Algorithm

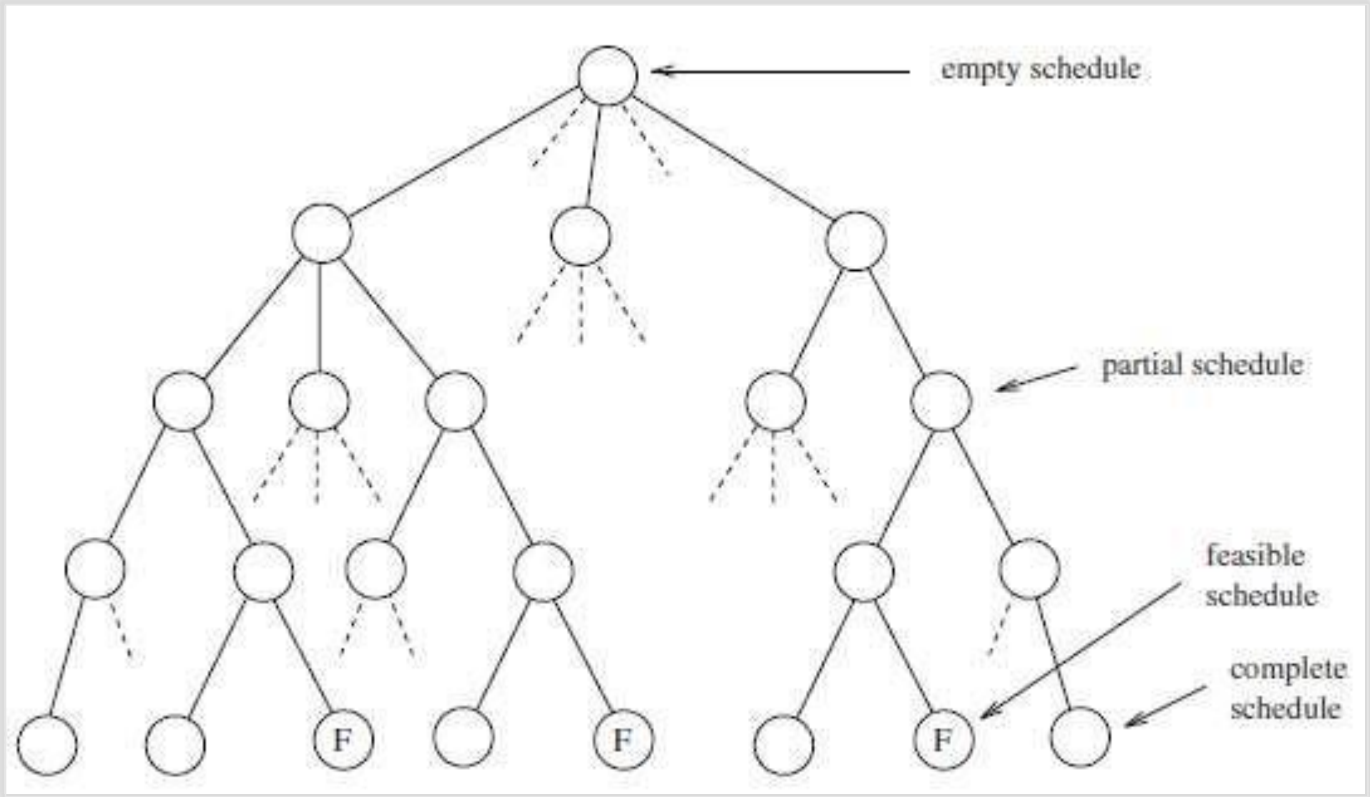
- In the optimal schedule shown in previous slide the processor remains idle in the interval [0, 1) although J1 is ready to execute.
  - If arrival times are not known a priori, then no online algorithm can decide whether to stay idle at time 0 or execute task J1.
- A scheduling algorithm that does not permit the processor to be idle when there are active jobs is called a *non-idle algorithm*.
- When arrival times are known a priori, non-preemptive scheduling problems are usually treated by branch-and-bound algorithm that performs well in the average case

Non-Preemptive Schedule Search Tree

At each step of the search, the partial schedule associated with a vertex is extended by inserting a new task. For n number of tasks in the set:

- Path length = n
- Nr. of leaves = n!

Tree search complexity ??



Note: arrival times are known a priori – search tree space.
Algorithm goal: search for a leaf corresponding to a feasible schedule.
-> the root is an empty schedule,
-> an intermediate vertex is a partial schedule,
-> a terminal vertex (leaf) is a complete schedule, feasible or non-feasible

Non-Preemptive Scheduling Algorithms

- We see that an exhaustive search through the search tree space in the previous slide has a complexity of O(n·n!) which is computationally intractable
- The objective of the algorithms presented is to limit the search space and reduce the computational complexity of the algorithm
  - BRATLEY'S ALGORITHM
    - uses additional information to prune the tree and reduce the complexity in the average case.
  - THE SPRING ALGORITHM
    - adopts suitable heuristics to follow promising paths on the tree and build a complete schedule in polynomial time

BRATLEY'S ALGORITHM (1 | NO PREEM | FEASIBLE)

- The algorithm solves the problem of finding a feasible schedule of a set of non-preemptive tasks with arbitrary arrival times, utilizing the search tree space.
  - Start with an empty schedule and,
  - At each step of the search, visits a new vertex in the search tree space and add a task in the partial schedule.
  - Pruning technique applied -- a branch is abandoned when:
    - the addition of any node to the current path causes a missed deadline;
    - a feasible schedule is found at the current path.

Example of search performed by Bratley's algorithm

… Tasks set …

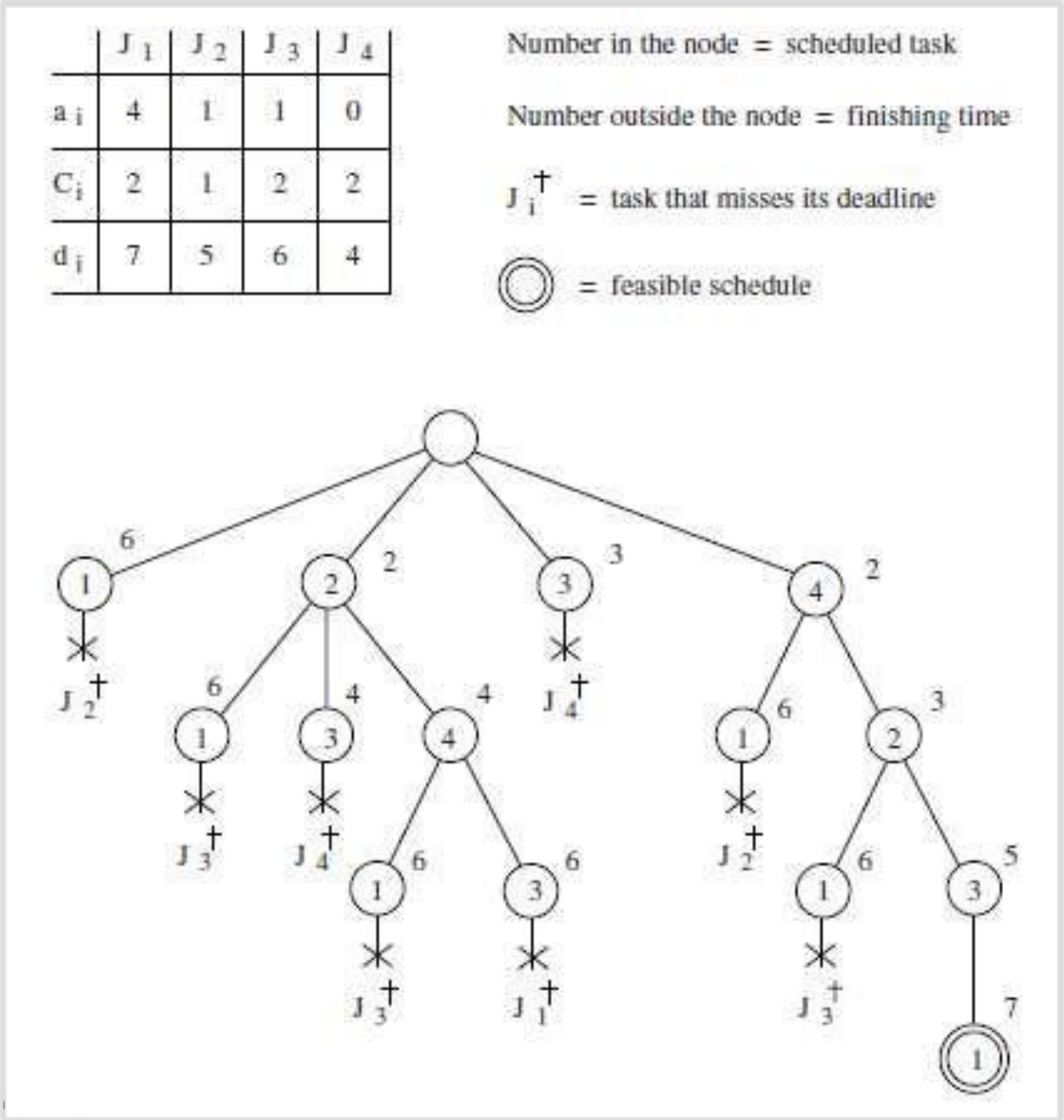
… Notations …

… Start with an empty schedule

… Extend the empty schedule …

… Prune if conditions met…

… Continue by extending and pruning until a feasible schedule is found or the search exhausted the entire tree space.



	J_1	J_2	J_3	J_4
a_i	4	1	1	0
C_i	2	1	2	2
d_i	7	5	6	4

Number in the node = scheduled task

Number outside the node = finishing time

$J_i^\dagger$ = task that misses its deadline

◎ = feasible schedule

Bratley Algorithm Usage

- Pruning techniques are effective for reducing the search space. Nevertheless, the worst-case complexity of the algorithm is still $O(n \cdot n!)$.
  - For this reason, Bratley's algorithm can only be used in off-line mode, when all task parameters (including the arrival times) are known in advance.
- Example of usage -- in a time-triggered system
  - Tasks are activated at predefined instants by a timer process.
  - The resulting schedule produced by Bratley's algorithm can be stored in a data structure, called task activation list.
  - Then, at run time, a dispatcher simply extracts the next task from the activation list and puts it in execution.

THE SPRING ALGORITHM

- Objective -- find a feasible schedule when tasks have different types of constraints, such as precedence relations, resource constraints, arbitrary arrivals, non-preemptive properties, and importance levels.
- The Spring algorithm is used in a distributed computer architecture and can also be extended to include fault-tolerance requirements.
- How does the algorithm reduce the tree search space complexity?
  - The search is driven by a heuristic function H, which actively directs the scheduling to a plausible path
  - On each level of the search, function H is applied to each of the tasks that remain to be scheduled.
  - The task with the smallest value determined by the heuristic function H is selected to extend the current schedule.

The Heuristic Function

- The heuristic function H, is a flexible mechanism used to define and modify the scheduling policy of the kernel.
    - Remember-- the task with the smallest value determined by the heuristic function H is selected to extend the current schedule
- Examples of H functions
    - if H = ai (arrival time) the algorithm behaves as First Come First Served;
    - if H = Ci (computation time) it works as Shortest Job First,
    - If H = di (deadline) the algorithm is equivalent to Earliest Deadline First.

Considering Resource Constraints in the Scheduling Algorithm

How can we include resource constraints in the scheduling algorithm?

Solution: each task Ji will include a binary array of resources

Ri = [R1(i), . . . , Rr(i)],

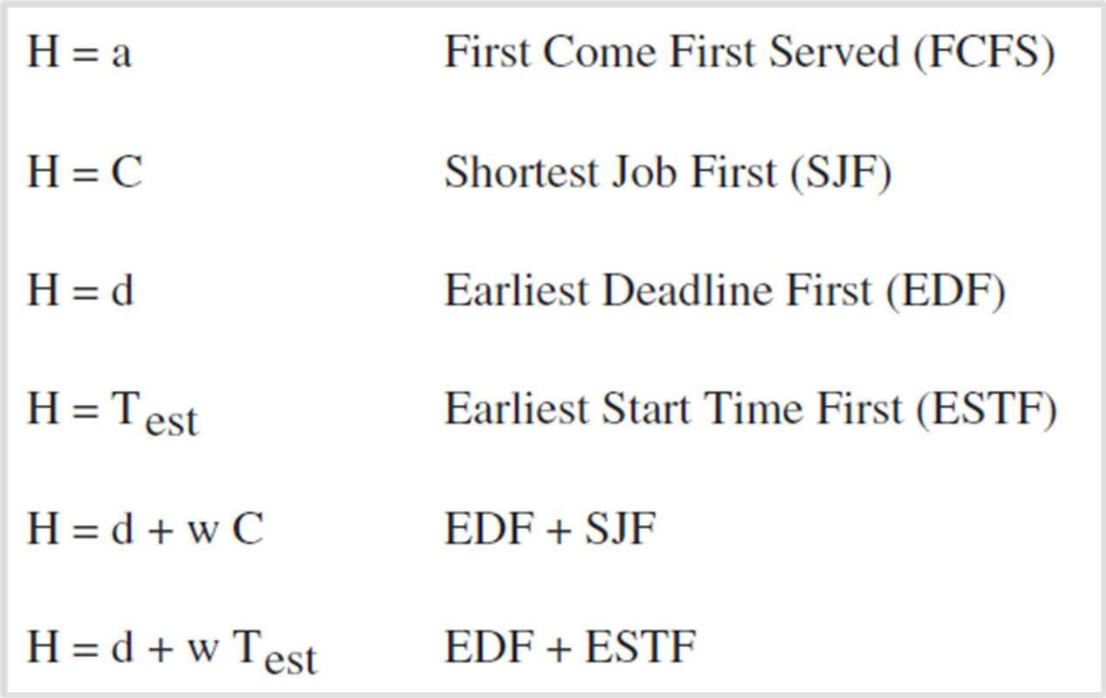
- Where Rk(i) = 0 if Ji does not use resource Rk, Rk(i) = 1 if Ji uses Rk in exclusive mode.

- Given a partial schedule, the algorithm determines, for each resource Rk, the earliest time the resource is available. This time is denoted as EATk (Earliest Available Time).
- Thus, the earliest start time that a task Ji can begin the execution without blocking on shared resources is:

  $T_{\mathrm{est}}(i) = \max\left[a_{\mathrm{i}}, \max_{\mathrm{k}}(EAT_{\mathrm{k}})\right]$; where, ai is the arrival time of Ji.
- Once T_{est} is calculated for all the tasks, a possible search strategy is to select the task with the smallest value of T_{est}.

Simple and Composed Heuristic Functions in Spring Algorithm

- Composed heuristic functions can also be used to integrate relevant information on the tasks, such as
  - H = d +W · C
  - H = d +W · T_{est},
- Where, W is a weight that can be adjusted for different application environments



H = a	First Come First Served (FCFS)
H = C	Shortest Job First (SJF)
H = d	Earliest Deadline First (EDF)
$H = T_{est}$	Earliest Start Time First (ESTF)
H = d + w C	EDF + SJF
$H = d + w\,T_{est}$	EDF + ESTF

Handling Precedence Constraints in the Scheduling Algorithm

How do we include precedence constraints?

Solution: a factor E, called eligibility, is added to the heuristic function.

- A task becomes eligible to execute ($E_i = 1$) only when all its ancestors in the precedence graph are completed.
- If a task is not eligible, then $E_i = \infty$; hence, it cannot be selected for extending a partial schedule.

Spring Algorithm Mechanism

Starting with a partial schedule, the algorithm extends the partial schedule and determines whether the current schedule is strongly feasible (?) … remains feasible by extending with the remaining tasks …

  - If this partial schedule is not to be strongly feasible, the algorithm stops the search process,
  - Otherwise the search continues until a complete feasible schedule is met.

- Since a feasible schedule is reached through n nodes and each partial schedule requires the evaluation of most n heuristic functions, the complexity of the Spring algorithm is $O(n^2)$.
- Backtracking can be used to continue the search after a failure …
- Disadvantage: the Spring algorithm is not optimal

Scheduling with Precedence Constraints

- The problem of finding an optimal schedule for a set of tasks with precedence relations is in general NP-hard.
- Optimal algorithms that solve the problem in polynomial time can be found under particular assumptions on the tasks. We present two algorithms:
  - LATEST DEADLINE FIRST (1 | PREC,SYNC | L_{MAX})
  - EDF with precedence constraints (1 | PREC,PREEM | L_{MAX})

LATEST DEADLINE FIRST (1 | PREC,SYNC | L_{MAX})

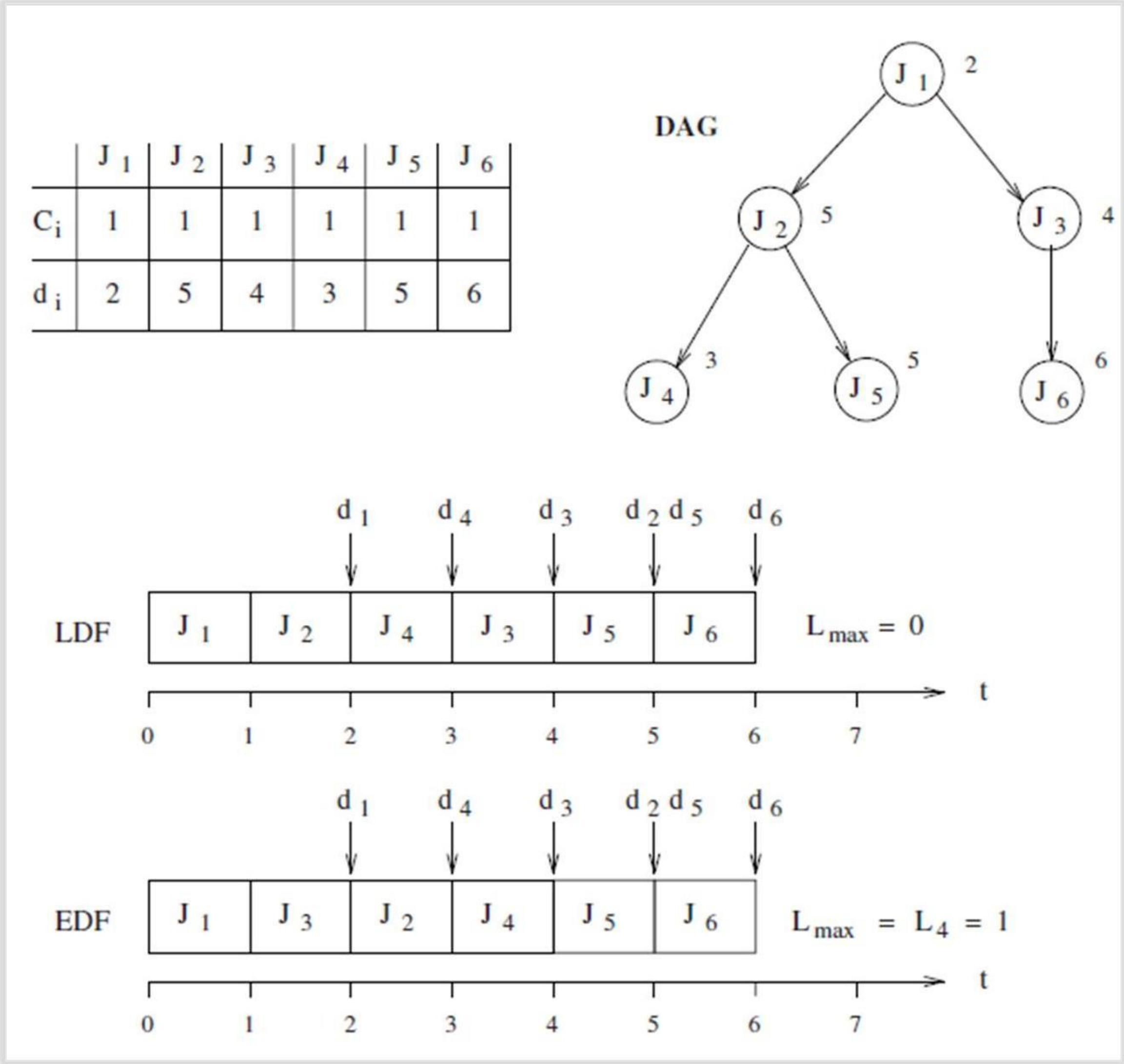
- Or, LDF algorithm, can be executed in polynomial time
- Given a set J of n tasks and a directed acyclic graph (DAG) describing their precedence relations, LDF builds the scheduling queue from tail to head:

  PROCEDURE: among the tasks without successors or whose successors have been all selected, LDF selects the task with the latest deadline to be scheduled last. This procedure is repeated until all tasks in the set are selected.

- At run time, tasks are executed in order from the queue
- See proof of LDF optimality in the reference book (pp 71-72)

LDF Example

- … task set …
- … precedence graph …
- LDF schedule → optimal under the precedence constraints
- EDF schedule → not optimal under precedence constraints,
    - L_{max}(EDF) > L_{max}(LDF)



	J_1	J_2	J_3	J_4	J_5	J_6
C_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6

EDF WITH PRECEDENCE CONSTRAINTS (1 | PREC, PREEM | L$_{MAX}$)

- The problem of scheduling a set of n tasks with precedence constraints and dynamic activations can be solved in polynomial time complexity only if tasks are preemptable
- Algorithm: the basic idea is to transform a set J of dependent tasks into a set J* of independent tasks by an adequate modification of timing parameters. Then, tasks are scheduled by the Earliest Deadline First (EDF) algorithm.
  - Transformation algorithm …
  - EDF …

Modification Rule for the Release Times

$s_b \geq r_b$ (that is, J_b must start the execution not earlier than its release time);

$s_b \geq r_a + C_a$ (that is, J_b must start the execution not earlier than the minimum finishing time of J_a).

Precedence constraint:
Ja → Jb

… start times …
… replace the release time r_b by: max(r_b, r_a + C_a) …

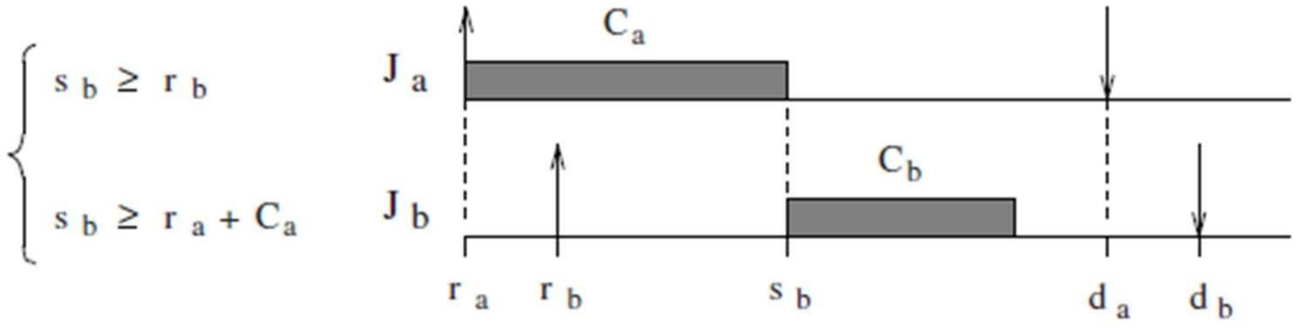
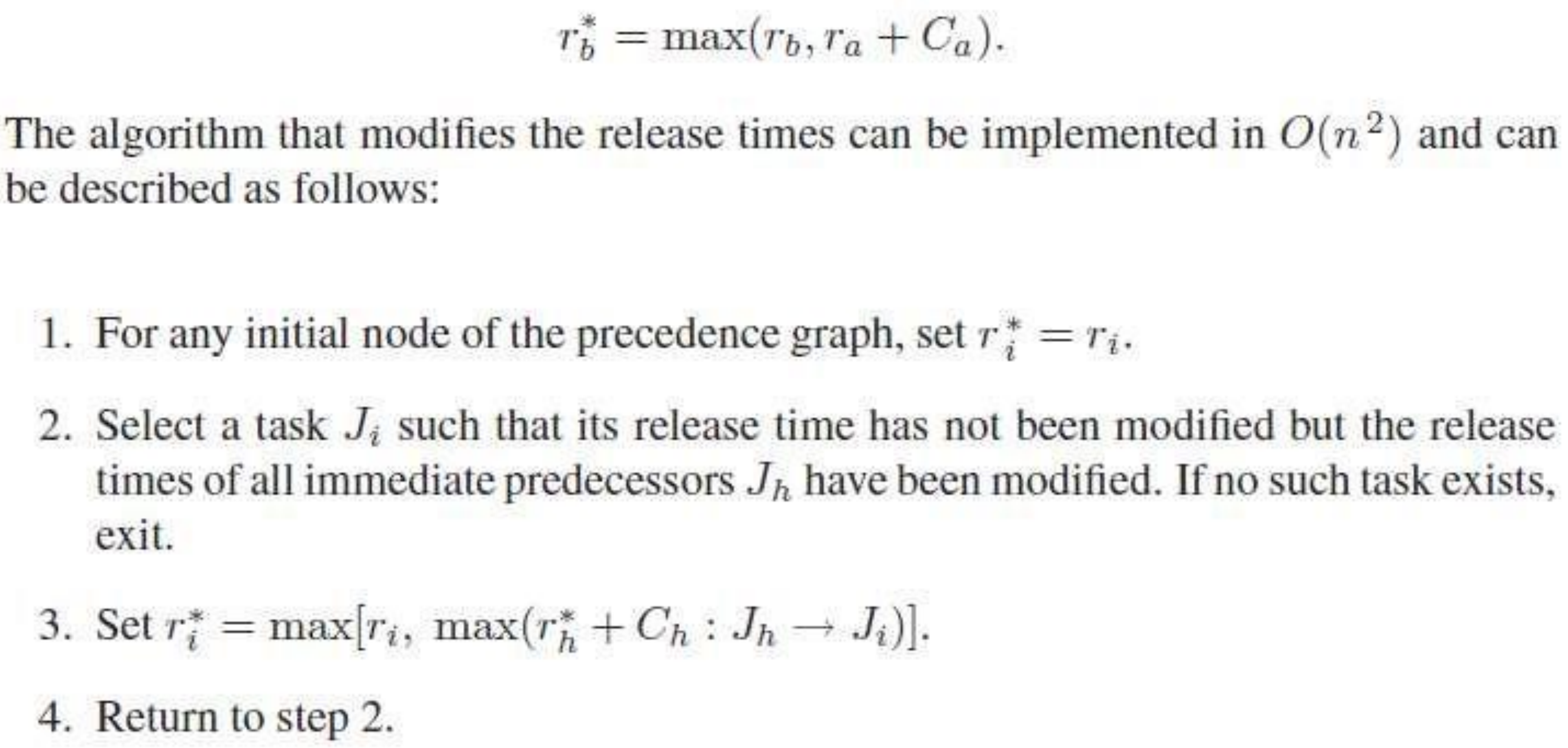


Figure 3.14 If $J_a \rightarrow J_b$, then the release time of J_b can be replaced by $\max(r_b, r_a + C_a)$.

Algorithm To Modify the Release Times

Let r_{b}^* be the new release time of J_b



$$r_b^* = \max(r_b, r_a + C_a).$$

The algorithm that modifies the release times can be implemented in $O(n^2)$ and can be described as follows:

1. For any initial node of the precedence graph, set $r_i^* = r_i$.
2. Select a task J_i such that its release time has not been modified but the release times of all immediate predecessors J_h have been modified. If no such task exists, exit.
3. Set $r_i^* = \max[r_i,\ \max(r_h^* + C_h : J_h \rightarrow J_i)]$.
4. Return to step 2.

Modification Rule for the Deadlines

Precedence constraint:

Ja → Jb

… finish times…

… replace deadline d_a by: $\min(d_a, d_b - C_b)$ …

$f_a \leq d_a$ (that is, J_a must finish the execution within its deadline);

$f_a \leq d_b - C_b$ (that is, J_a must finish the execution not later than the maximum start time of J_b).

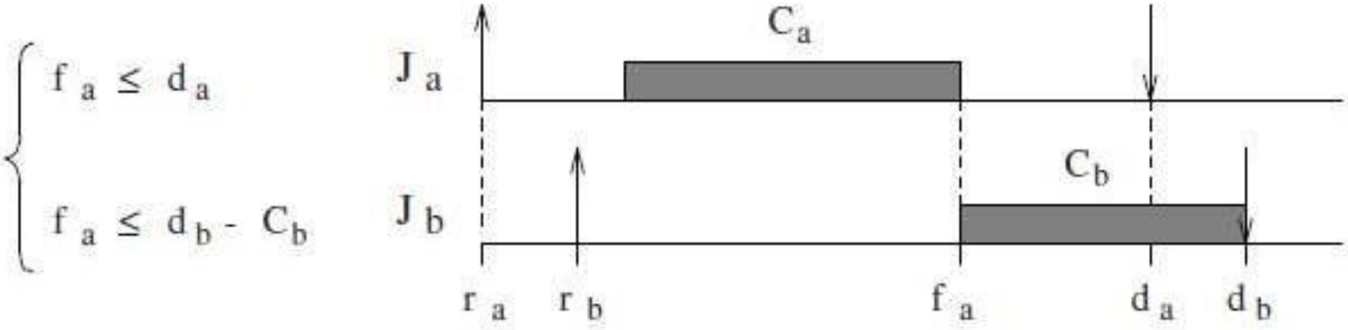
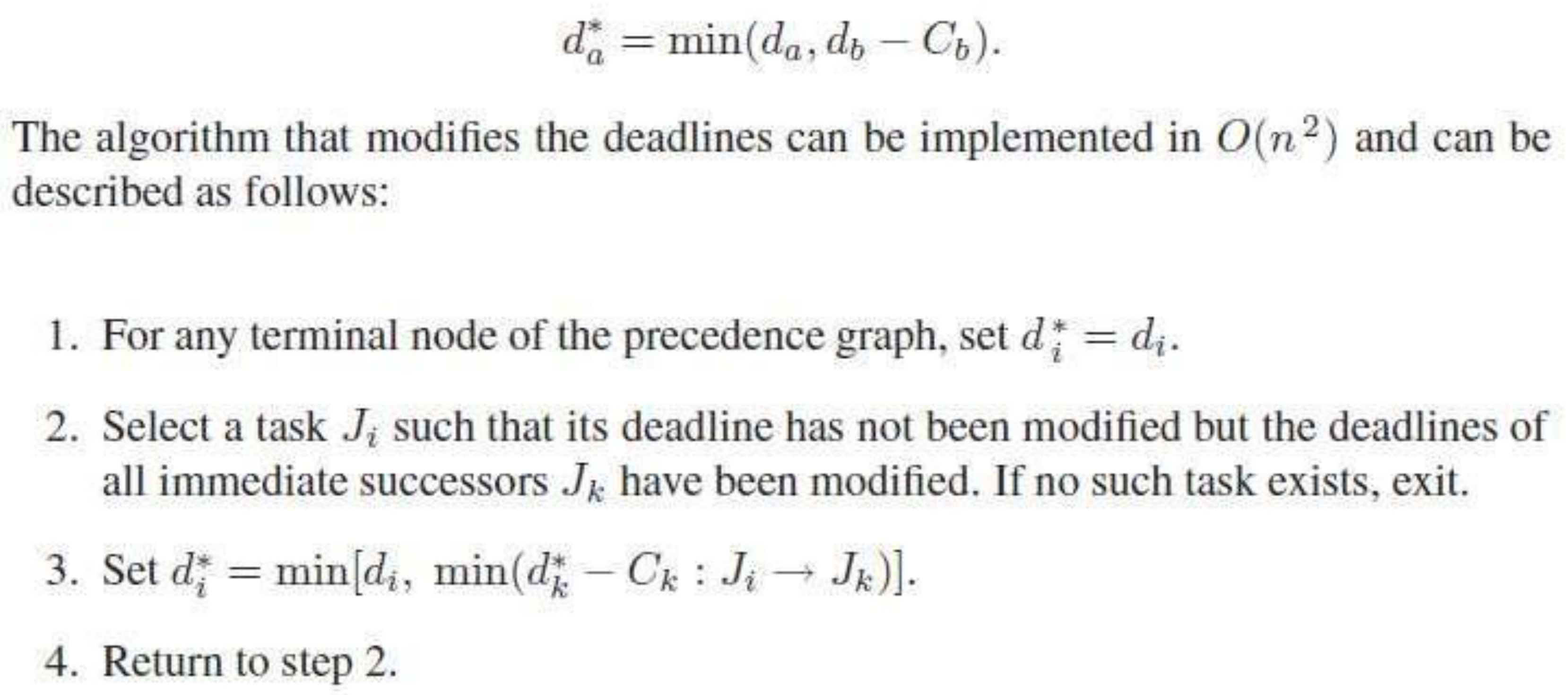


Figure 3.15 If $J_a \rightarrow J_b$, then the deadline of J_a can be replaced by $\min(d_a, d_b - C_b)$.

Algorithm to Modify the Deadlines

Let d_b^* be the new deadline time of J_b



$$d_a^* = \min(d_a, d_b - C_b).$$

The algorithm that modifies the deadlines can be implemented in $O(n^2)$ and can be described as follows:

1. For any terminal node of the precedence graph, set $d_i^* = d_i$.
2. Select a task J_i such that its deadline has not been modified but the deadlines of all immediate successors J_k have been modified. If no such task exists, exit.
3. Set $d_i^* = \min[d_i,\ \min(d_k^* - C_k : J_i \rightarrow J_k)]$.
4. Return to step 2.

Proof of Optimality

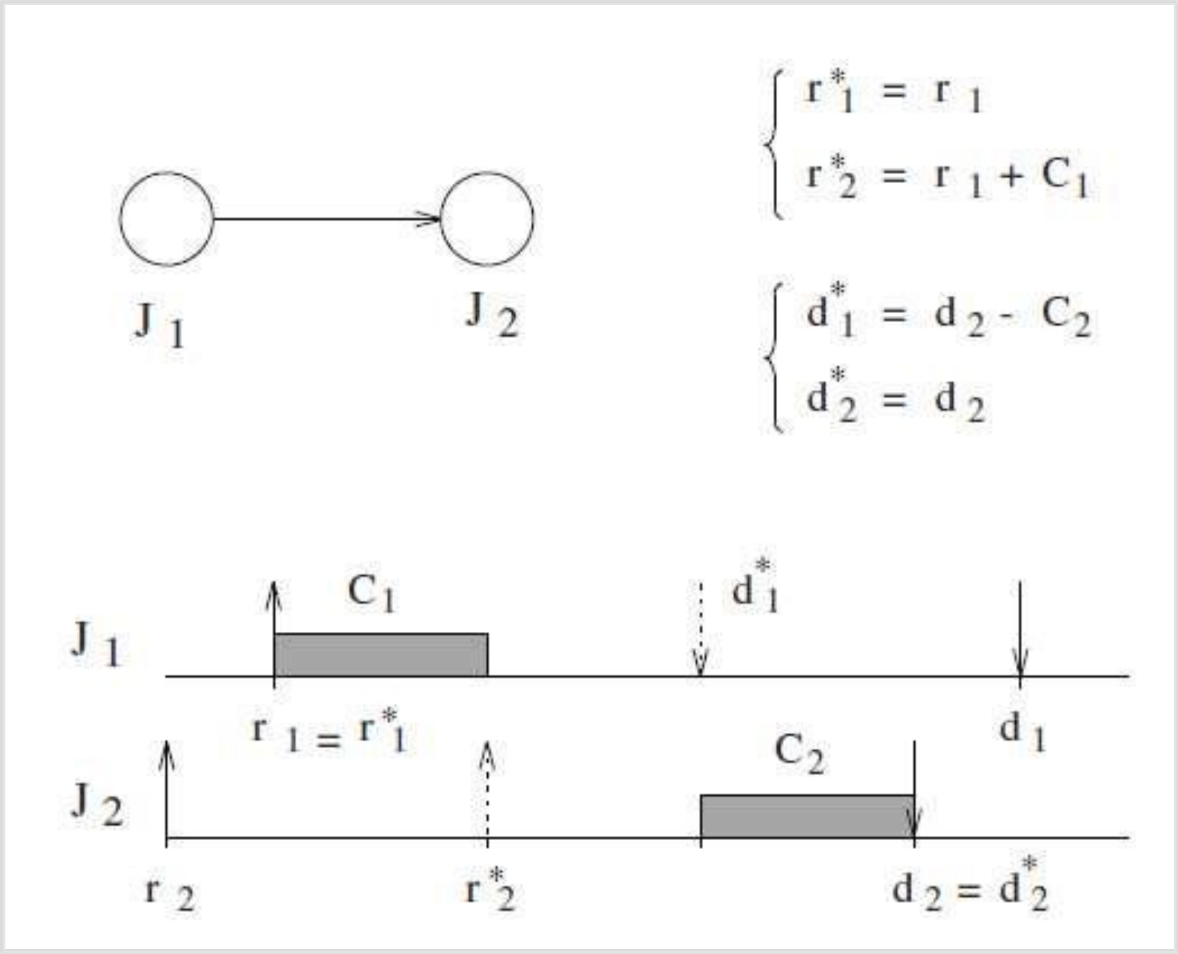
- The transformation algorithm ensures that if a feasible schedule exists for the modified task set J^* under EDF, then the original task set J is also schedulable; that is, all tasks in J meet both precedence and timing constraints
- In fact, if J^* is schedulable, all modified tasks start at or after time r^*_i and are completed at or before time d^*_i .
  - Since $r^*_i \geq$ ri and $d^*_i \leq$ di, the schedulability of J^* implies that J is also schedulable.
- However, we need to show that the precedence constraints in J are not violated

Proof of Optimality

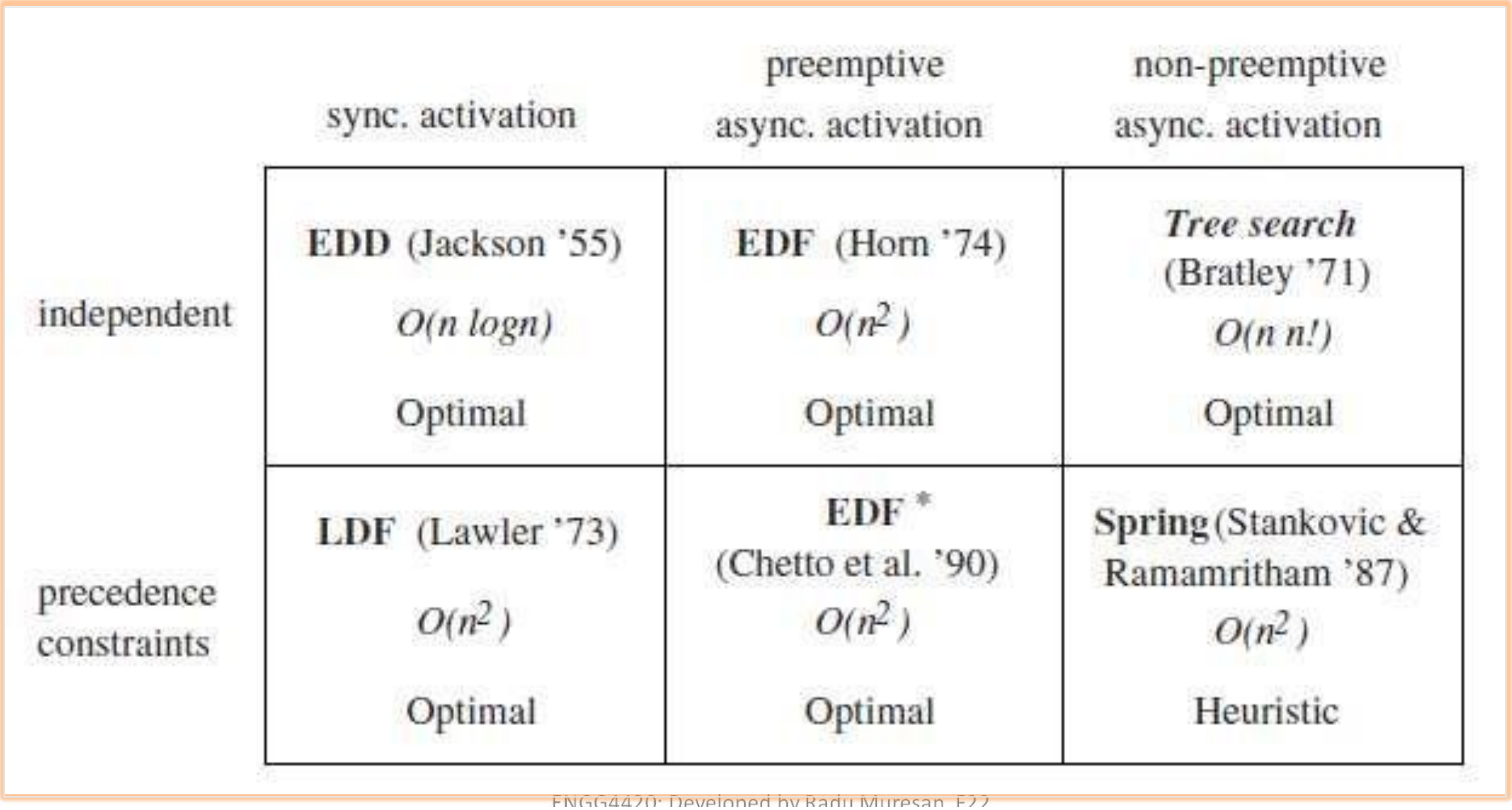
Case analysis: J1 must precede J2 (i.e., J1 → J2), but J2 arrives before J1 and has an earlier deadline.

Clearly, if the two tasks are executed under EDF, their precedence relation cannot be met.

However, if we apply the transformation algorithm, the time constraints are modified as shown … discuss …



Summary of Algorithms for Aperiodic Tasks



	sync. activation	preemptive async. activation	non-preemptive async. activation
independent	**EDD** (Jackson '55) $O(n\ logn)$ Optimal	**EDF** (Horn '74) $O(n^2)$ Optimal	***Tree search*** (Bratley '71) $O(n\ n!)$ Optimal
precedence constraints	**LDF** (Lawler '73) $O(n^2)$ Optimal	**EDF** * (Chetto et al. '90) $O(n^2)$ Optimal	**Spring** (Stankovic & Ramamritham '87) $O(n^2)$ Heuristic

Homework

- Exercises 3.1 to 3.5 pages 77-78 in the textbook (Solutions on pp 460-461)