Topic 1: Preliminaries

Basic Notions
Scheduling Models

Basic Notions

The notion of ***task*** is used to express some well-defined activity or piece of work

Planning in practical applications requires some knowledge about the tasks

This knowledge does not regard their nature, but rather general properties such as

- **processing times,**
- **relations** between the tasks concerning the order in which the tasks can be processed**,**
- **release times** which inform about the earliest times the tasks can be started,
- **deadlines** that define the times by which the tasks must be completed**,**
- **due dates** by which the tasks should be completed together with cost functions that define penalties in case of due date violations**,**
- **additional resources** (for example, tools, storage space, data)

Based on these data one could try to develop a ***work plan*** or ***time schedule*** that specifies for each task when it should be processed, on which machine or processor, including preemption points, etc.

Depending on how much is known about the tasks to be processed, we distinguish between three main directions in scheduling theory:

☞ ***Deterministic*** or ***static*** or ***off-line scheduling*** assumes that **all information** required to develop a schedule **is known in advance**, before the actual processing takes place

Especially in production scheduling and in real-time applications the deterministic scheduling discipline plays an important role

☞ ***Non-deterministic scheduling*** is less restrictive: **only partial information is known**

for example, computer applications where tasks are pieces of software with unknown run-time

☞ ***On-line scheduling:*** In many situations detailed knowledge of the nature of the tasks is available, but the **time at which tasks occur is open**

> If the demand of executing a task arises a decision upon acceptance or rejection is required, and, in case of acceptance, the task start time has to be fixed

In this situation schedules cannot be determined off-line, and we then talk about *on-line* scheduling or *dynamic* scheduling

☞ ***Non-clairvoyant scheduling:*** consider problems of **scheduling** jobs with unspecied execution time requirements

☞ ***Stochastic scheduling:*** only probabilistic information about parameters is available

> In this situation probability analysis is typical means to receive information about the system behavior

- For each type of scheduling one can find justifying applications

Here, off-line scheduling (occasionally also on-line scheduling) is considered

Deterministic Scheduling Problems

- Between tasks there are relations describing the **relative order** in which the tasks are to be performed
  - order of task execution can be restricted by conditions like ***precedence constraints***
- *Preemption* of task execution can be allowed or forbidden
- **Timing conditions** such as task ***release times*, *deadlines* or *due dates*** may be given
  - In case of due dates *cost functions* may define *penalties* depending on the amount of lateness
- There may be conditions for *time lags* between pairs of tasks, such as setup delays
- In so-called *shop problems* sequences of tasks, each to be performed on some specified machine, are defined
  - An example is the well-known flow shop or assembly line processing

Scheduling problems are characterized not only by the tasks and their specific properties, but also by information about the **processing devices**

Processors or *machines* for processing the tasks can be ***identical***, can have different speeds (***uniform***), or their processing capabilities can be ***unrelated***

Deterministic Scheduling Problems

The problem is to determine an appropriate *schedule*, i.e. one that satisfies all conditions imposed on the tasks and processors

A schedule essentially defines the start times of the tasks on a specified processor

Generally there may exist several possible schedules

An important is to define an *optimization criterion*

Common criteria are:

- minimization of the *makespan* of the total task set,
- minimization of the *mean waiting* time of the tasks

The optimization criterion allows to choose an appropriate schedule

Such schedules are then used as a planning basis for carrying out the various activities

Unfortunately, finding optimal schedules is in general a very difficult process

Except for simplest cases, these problems turn out to be NP-hard, and hence the time required computing an exact solution is beyond all practical means

In this situation, algorithmic approaches for *sub-optimal* schedules seem to be the only possibility

The Scheduling Model

The Scheduling Model

- Deterministic Model
- Optimization Criteria
- Scheduling Problem and $\alpha \mid \beta \mid \gamma$ - Notation
- Scheduling Algorithms

Tasks, Processors, etc.

Set of tasks $\mathcal{T} = \{T_1,\ T_2, \dots, T_n\}$

Set of resource types $\mathcal{R} = \{R_1,\ R_2, \dots, R_s\}$

Set of processors $\mathcal{P} = \{P_1,\ P_2, \dots, P_m\}$

Examples of processors:

CPUs in e.g. a multiprocessor system

Computers in a distributed processing environment

Production machines in a production environment

Processors may be

- *parallel*: they are able to perform the same functions
- *dedicated*: they are specialized for the execution of certain tasks

Parallel processors have the same execution capabilities

Three types of **parallel processors** are distinguished

- *identical*: if all processors from set $\mathcal{P}$ have equal task processing speeds
- *uniform* : if the processors differ in their speeds, but the *speed* b_i of each processor is constant and does not depend on the tasks in $\mathcal{T}$
- *unrelated*: if the speeds of the processors depend on the particular task

  unrelated processors are more specialized: on certain tasks, a processor may be faster than on others

Characterization of a task T_j

– Vector of *processing times* $p_j = [p_{ij}, \dots, p_{mj}]$, where p_{ij} is the time needed by processor P_i to process T_j

Identical processors: $p_{1j} = \cdots = p_{mj} = p_j$

Uniform processors: $p_{ij} = {}^{p_j}/_{b_i}, i = 1, \dots, m$

p_j = *standard processing time* (usually measured on the slowest processor),

b_i is the *processing speed factor* of processor P_i

Processing times are usually not known a priori in computer systems

Instead of exact values of processing times one can take their estimate

However, in case of deadlines exact processing times or at least upper bounds are required

Arrival time (or *release* or *ready time*) r_j … is the time at which task T_j is ready for processing

if the arrival times are the same for all tasks from $\mathcal{T}$, then $r_j = 0$ is assumed for all tasks

- *Due date* d_j … specifies a time limit by which T_j **should be** completed

  problems where tasks have due dates are often called "soft" real-time problems. Usually, penalty functions are defined in accordance with due dates

- *Penalty functions* G_j define penalties in case of due date violations

- *Deadline* $\tilde{d}_j$ … "hard" real time limit, by which T_j **must be** completed

- *Weight* (*priority*) w_j ... expresses the relative urgency of T_j

The Scheduling Model. Deterministic Model

– *Preemption / non-preemption:*

A scheduling problem is called *preemptive* if each task may be preempted at any time and its processing is resumed later, perhaps on another processor

If preemption of tasks is not allowed the problem is called *non-preemptive*

– *Resource requests:*

besides processors, tasks may require certain *additional resources* during their execution

Resources are usually scarce, which means that they are available only in limited amounts

In computer systems, exclusively accessible devices or data may be considered as resources

We assume without loss of generality that all these parameters, $p_j, r_j, d_j, \tilde{d}_j, w_j$ and $R_l(T_j)$ are integers. This assumption is equivalent to permitting arbitrary rational values

Conditions among the set of tasks $\mathcal{T}$: *precedence constraints*

$T_i \prec T_j$ means that the processing of T_i must be completed before T_j can be started

We say that a *precedence relation* $\prec$ is defined on set $\mathcal{T}$

mathematically, a precedence relation is a ***partial order***

The tasks in $\mathcal{T}$ are called *dependent*

if the *relation* $\prec$ is non-empty

otherwise, the tasks are called *independent*

T_i is called a *predecessor* of T_j if there is a sequence of asks $T_{\alpha_1}, \ldots, T_{\alpha_l}$ $(l \geq 0)$ with $T_i \prec T_{\alpha_1} \prec \ldots \prec T_{\alpha_l} \prec T_j$. Likewise, T_j is called a *successor* of T_i.

If $T_i \prec T_j$. , but there is no task T_α with $T_i \prec T_\alpha \prec T_j$. then T_i is called an *immediate predecessor* of T_j, and T_j an *immediate predecessor* of T_i

A task that has no predecessor is called ***start*** task

A task without successor is referred to as ***final*** task

Special types of precedence graphs are

- *chain dependencies*: the partial order is the union of linearly ordered disjoint subsets of tasks
- *tree dependencies*: the precedence relation is tree-like;
  - ***out-tree***: if all task dependencies are oriented away from the root
  - ***in-tree***: if all dependencies are oriented towards the root

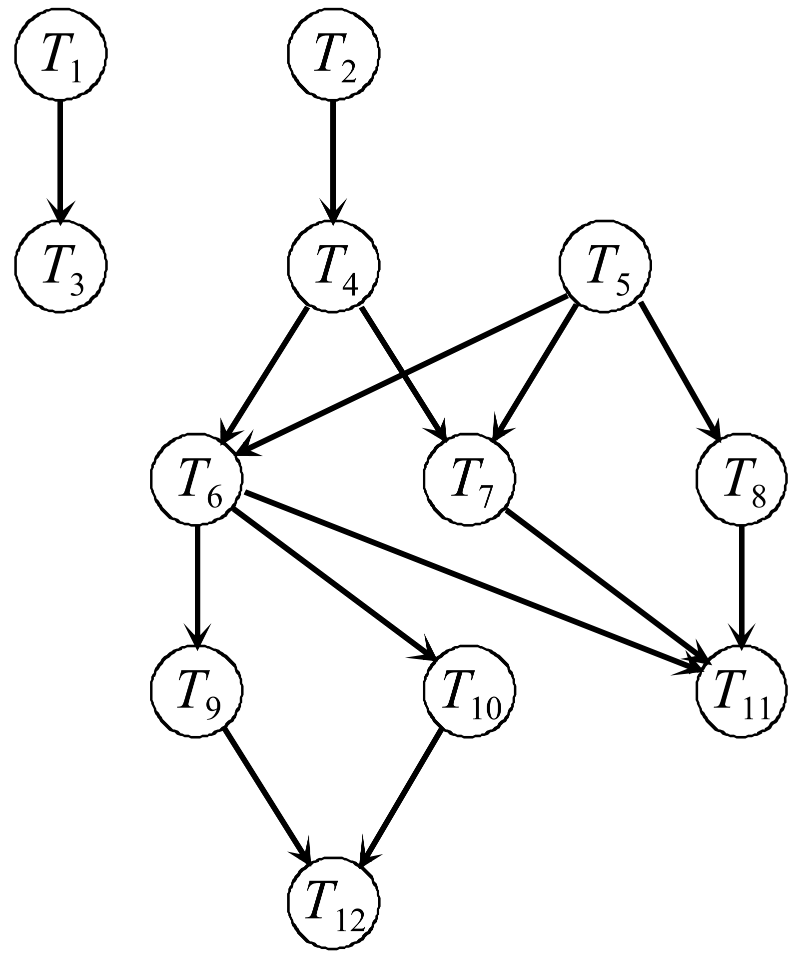
Representation of tasks with precedence constraints:

– *task-on-node graph (Hasse diagram)*

For each $T_i \prec T_j$, an edge is drawn between the corresponding nodes

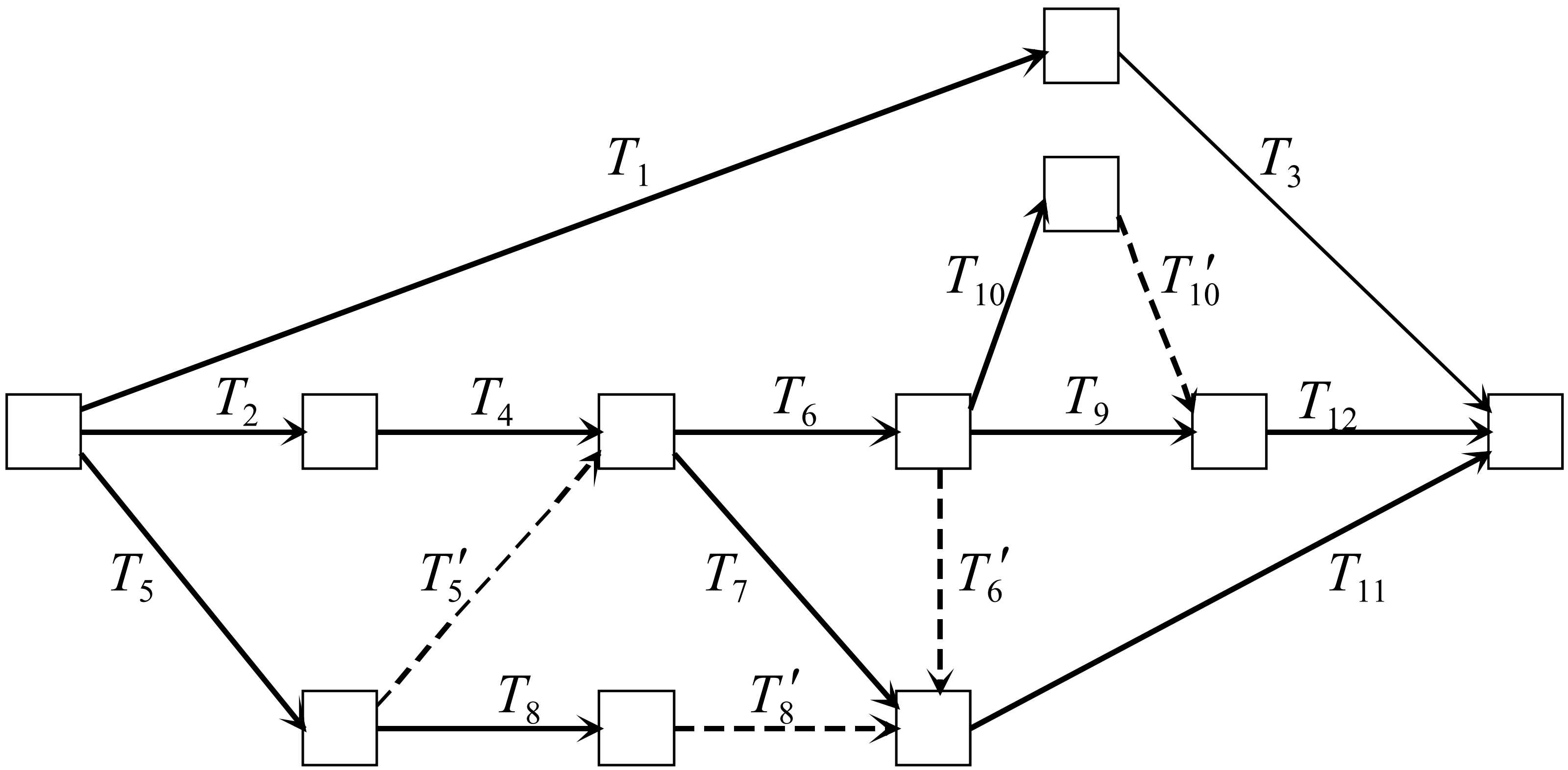
The situation $T_i \prec T_j$ and $T_j \prec T_k$ is called *transitive dependency* between T_i and T_k.

Transitive dependencies are not explicitly represented



task-on-arc graph, *activity network.* Arcs represent tasks and nodes time events

Example 1: $\mathcal{T} = \{T_1, \ldots, T_{10}\}$ with precedences as shown by the above Hasse diagram. A corresponding activity network:



Task T_j is called *available* at time t if $r_j \leq t$ and all its predecessors (with respect to the precedence constraints) have been completed by time t

Schedules

Schedules or work plans generally …

inform about the times and on which processors the tasks are executed

To demonstrate the principles, the schedules are described for the special case of:

- parallel processors
- tasks have no deadlines
- tasks require no additional resources

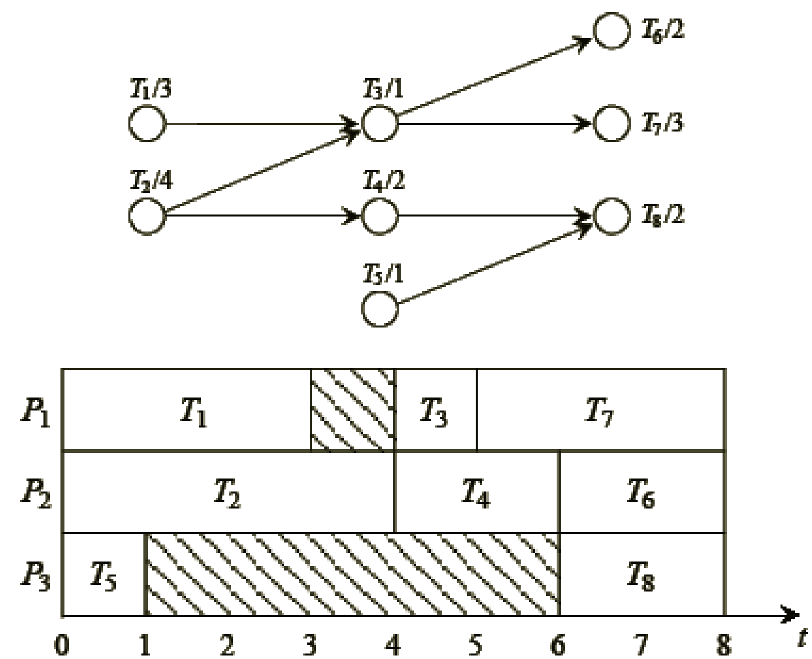
Release times and precedence constraints may occur

(3) Graphic representation: **Gantt chart** - this is a two-dimensional diagram

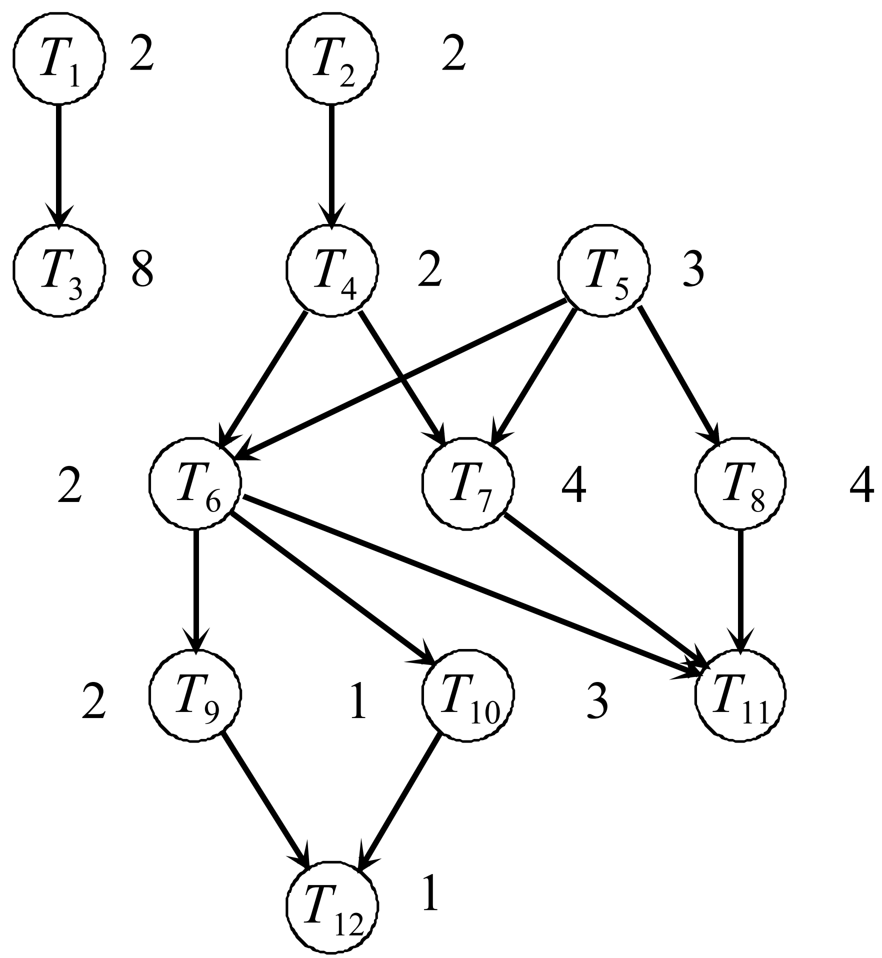
The abscissa represents the time axis that usually starts with time 0 at the origin

Each processor is represented by a line

For a task T_j to be processed by P_i a bar of length $p(T_j)$ and that begins at the time marked by $s(T_j)$, is entered in the line corresponding to P_i



Example 1: $\mathcal{T} = \{T_1, ..., T_{12}\}$ with precedences as shown by the Hasse diagram:

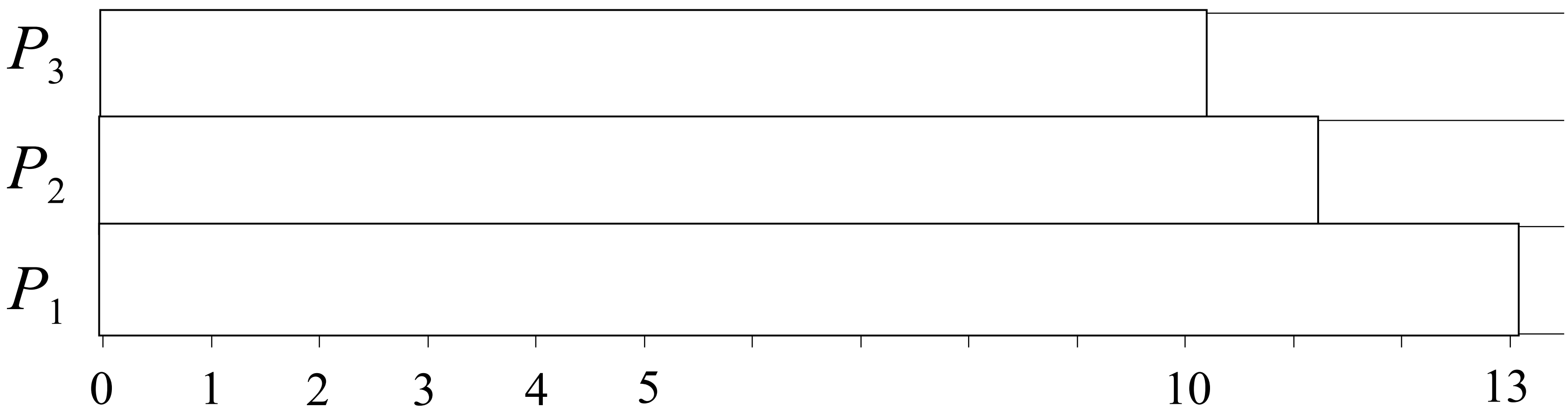


Example 2: non-preemptive schedule

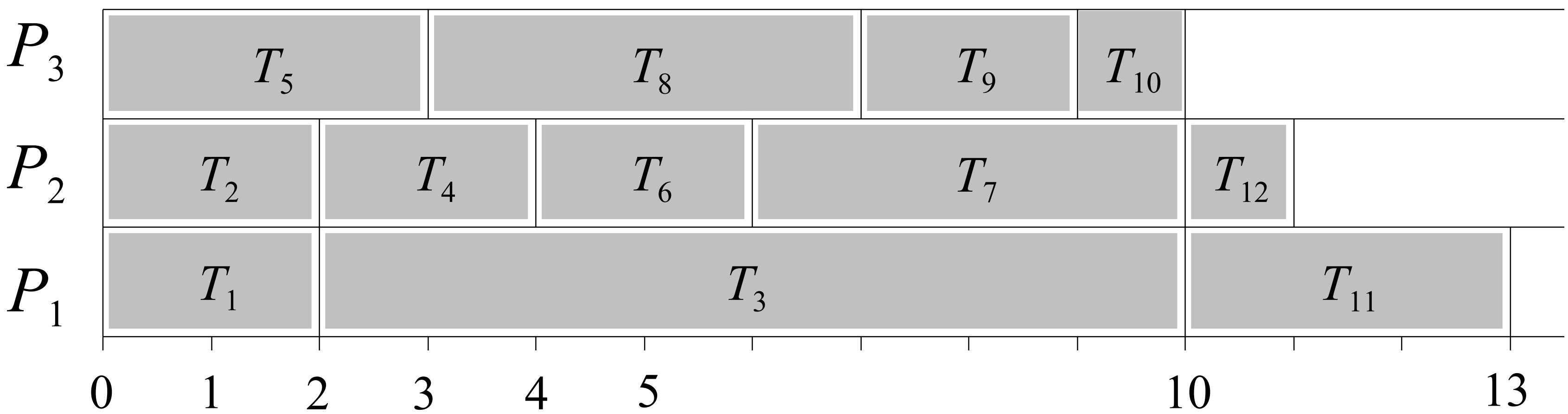
In the above example, let $(2, 2, 8, 2, 3, 2, 4, 4, 2, 1, 3, 1)$ be the vector of processing times, and assume all release times $= 0$

Assume furthermore that there are 3 identical processors ($\mathcal{P} = \{P_1, \ldots, P_3\}$) available for processing the tasks

Gantt chart of a non-preemptive schedule:



The Scheduling Model. *Schedule representation*



The Scheduling Model. Deterministic Model

Given a schedule ς, the following can be determined for each task T_j :

flow time, turnarround, response $F_j := c_j - r_j$

lateness $L_j = c_j - d_j$

tardiness $D_j = \max\{c_j - d_j, 0\}$

tardy task $U_j = \begin{cases} 0 & \text{if } D_j = 0 \\ 1 & \text{else} \end{cases}$

Evaluation of schedules

Maximum makespan	$C_{max} = max\{c_j \mid T_j \in \mathcal{T}\}$
Mean flow time	$\bar{F} := (1/n) \Sigma F_j$
Mean weighted flow time	$\overline{F_w} := (\Sigma w_j F_j) / (\Sigma w_j)$
Maximum lateness	$L_{max} = max\{L_j \mid T_j \in \mathcal{T}\}$
Mean tardiness	$\bar{D} := (1/n) \Sigma D_j$
Mean weighted tardiness	$\overline{D_w} := (\Sigma w_j D_j) / (\Sigma w_j)$
Mean sum of tardy tasks	$\bar{U} := (1/n) \Sigma U_j$
Mean weighted sum of tardy tasks	$\overline{U_w} := (\Sigma w_j U_j) / (\Sigma w_j)$

Given a set of tasks and a processor environment there are generally many possible schedules

Evaluating schedules: distinguish between *good* and *bad* schedules

This leads to different *optimization criteria*

Minimizing the maximum makespan C_{max}

C_{max} criterion: C_{max}-optimal schedules have minimum makespan

the total time to execute all tasks is minimal

Minimizing *schedule length* is important from the viewpoint of the owner of a set of processors (machines):
This leads to both, the maximization of the processor utilization factor (within schedule length C_{max}), and the minimization of the maximum in-process time of the scheduled set of tasks

Deadline related criteria

If deadlines are specified for (some of) the tasks we are interested in a schedule in which all tasks complete before their deadlines expire

Question: does there exist a schedule that fulfills all the given conditions?

Such a schedule is called *valid* (*feasible*)

Here we are faced in principle with a *decision problem*

If, however, a valid schedule exits, we would of course like to get it explicitly

If a valid schedule exists we may wish to find a schedule that has certain additional properties, such as minimum makespan or minimum mean flow

Hence in deadline related problems we often additionally impose one of the other criteria

Minimizing the maximum lateness L_{max}

This concerns tasks with due dates

Minimizing L_{max} expresses the attempt to keep the maximum lateness small, no matter how many tasks are late

Due date involving criteria are of great importance in manufacturing systems, especially for specific customer orders

Minimizing the mean weighted tardiness $\overline{D_w}$

This criterion considers a weighted sum of tardinesses

Minimizing mean weighted tardiness means that a task with large weight should have a small tardiness

Minimizing the weighted sum of tardy tasks $\overline{U_w}$

This criterion considers only the number of tardy tasks

Individual weights for the tasks are again possible

$\alpha \mid \beta \mid \gamma$ - *Notation*

Scheduling problem Π is defined by a set of parameters for processors, tasks, and an optimality criterion

An *instance* I of problem Π is specified by particular values for the problem parameters

The parameters are grouped in *three fields* $\alpha \mid \beta \mid \gamma$:

- α specifies the processor environment,
- β describes properties of the tasks, and
- γ the definition of an optimization criterion

The *terminology* introduced below aims to classify scheduling problems

Component α specifies the processors

$\alpha = \alpha_1, \alpha_2$ describes the processor environment

Parameter $\alpha_1 \in \{\varnothing, P, Q, R\}$ characterizes the ***type of processor***

parameter $\alpha_2 \in \{\varnothing, k\}$ denotes the ***number of available processors***:

α_1	
$\varnothing$	single processor
P	identical processors
Q	uniform processors
R	unrelated processors

α_2	
$\varnothing$	the number of processors is assumed to be variable
k	the number of processors is equal to k (k is a positive integer)
∞	the number of processors is unlimited

Component β specifies the tasks

$\beta = \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6$ describes task and resource characteristics

Parameter $\beta_2 \in \{\varnothing, pmtn\}$ indicates the possibility of ***task preemption***

β_1	
$\varnothing$	no preemption is allowed
$pmtn$	preemptions are allowed

Parameter $\beta_3 \in \{\varnothing, prec, tree, chains\}$ reflects the ***precedence constraints***

$\beta_3 = \varnothing, prec, tree, chains$: denotes respectively independent tasks, general precedence constraints, tree or a set of chains precedence constraints

Parameter $\beta_4 \in \{\varnothing, r_j\}$ describes ***ready times***

Parameter $\beta_5 \in \{\varnothing, p_j = p, \underline{p} \leq p_j \leq \bar{p}\}$ describes ***task processing times***

β_5	
$\varnothing$	tasks have arbitrary processing times
$p_j = p$	all tasks have processing times equal to p units
$\underline{p} \leq p_j \leq \bar{p}$	no p_j is less than $\underline{p}$ or greater than $\bar{p}$

Parameter $\beta_6 \in \{\varnothing, \tilde{d}_j\}$ describes ***deadlines***

β_6	
$\varnothing$	no deadlines or due dates are assumed in the system
$\tilde{d}_j$	deadlines are imposed on the performance of a task set

Component γ : Specifying the objective criterion

γ	description
C_{max}	***schedule length or makespan***
ΣC_j	***mean flow time***
$\Sigma w_j C_j$	***mean weighted flow time***
L_{max}	***maximum lateness***
ΣD_j	***mean tardiness***
$\Sigma w_j D_j$	***mean weighted tardiness***
ΣU_j	***number of tardy tasks***
$\Sigma w_j U_j$	***weighted number of tardy tasks***
$-$	***means testing for feasibility***

A schedule for which the value of a particular performance measure γ is at its minimum will be called *optimal :* The corresponding value of γ is denoted by γ^*

Topic 2

Scheduling on Parallel Processors

2.1 Minimizing Schedule Length

- **Identical Processors**
- Uniform Processors

2.2 Minimizing Mean Flow Time

- Identical Processors
- Uniform Processors

2.3 Minimizing Due Date Involving Criteria

- Identical Processors
- Uniform Processors

Independent tasks

Identical Processors $P \mid\mid C_{max}$

The first problem considered is $P \mid\mid C_{max}$ where

- a set of n independent tasks p_i
- on m identical processors
- minimize schedule length.

Identical Processors $P \mid \mid C_{max}$

Identical Processors. List Scheduling

$W_{seq} = \sum_{i=1}^{n} p_i$ be the total work of all jobs
p_{max} is the maximum processing time of a job.
W_{idle} be the total idle intervals, $W_{idle} \leq p_{max}(m-1)$
$C_{max} \leq \frac{W_{seq}+W_{idle}}{m}$ is the completion time of the set of tasks.

$C_{max} \leq \frac{W_{seq}+p_{max}(m-1)}{m}$, $C_{max} \leq \frac{W_{seq}}{m} + \frac{(m-1)}{m} p_{max}$
$\frac{W_{seq}}{m}$ and p_{max} are lower bounds of C_{opt}^{seq}, it follows that the worst-case performance bound is $\rho^{seq} \leq 2 - \frac{1}{m}$.

Approximation algorithm for $P \mid\mid C_{max}$:

One of the simplest algorithms is the *LPT algorithm* in which the tasks are arranged in order of non-increasing p_j .

Algorithm LPT *for* $P \mid\mid C_{max}$.
begin
Order tasks such that $p_1 \geq \cdots \geq p_n$;
for i = 1 to m do $s_i := 0$;
  -- processors P_i are assumed to be idle from time $s_i = 0$ on
$j := 1$;
repeat
  $s_k := \min\{ s_i \}$;
  Assign task T_j to processor P_k at time s_k;
    -- the first non-assigned task from the list is scheduled on the first processor that becomes free
  $s_k := s_k + p_j$; $j := j + 1$;
until $j = n$;   -- all tasks have been scheduled
end;

Identical Processors. *LPT Algorithm for P | | Cmax*

Theorem *If the LPT algorithm is used to solve problem* $P\,||\,C_{\max}$, *then* $R_{LPT} = \dfrac{4}{3} - \dfrac{1}{3m}$.

□

an example showing that this bound can be achieved.

Let $n = 2m + 1$, $\boldsymbol{p} = [2m - 1, 2m - 1, 2m - 2, 2m - 2, \ldots, m + 1, m + 1, m, m, m]$.

For $m = 3$, Next figure shows two schedules, an optimal one and an *LPT* schedule.

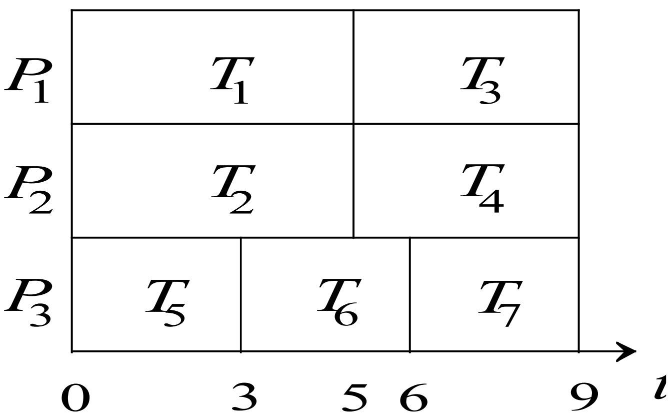
Example: $m = 3$ identical processors; $n = 2m + 1$,

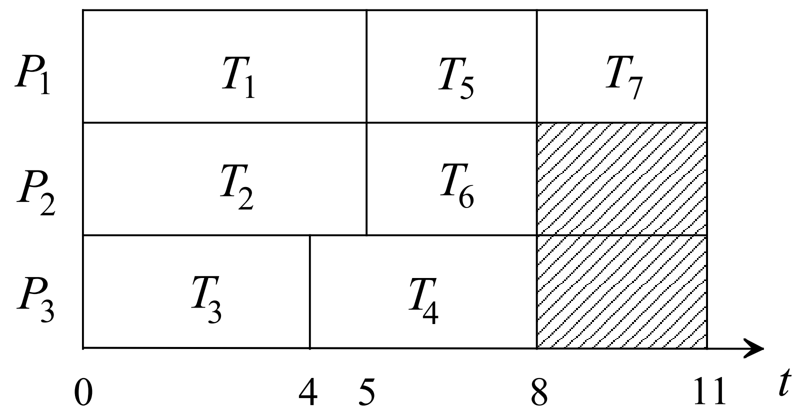
$\mathbf{p} = [2m - 1, 2m - 1, 2m - 2, 2m - 2, ..., m + 1, m + 1, m, m, m]$.

Time complexity of this algorithm is $O(n\log n)$

- the most complex activity is to sort the set of tasks.

For $m = 3$, $\mathbf{p} = [5, 5, 4, 4, 3, 3, 3]$.





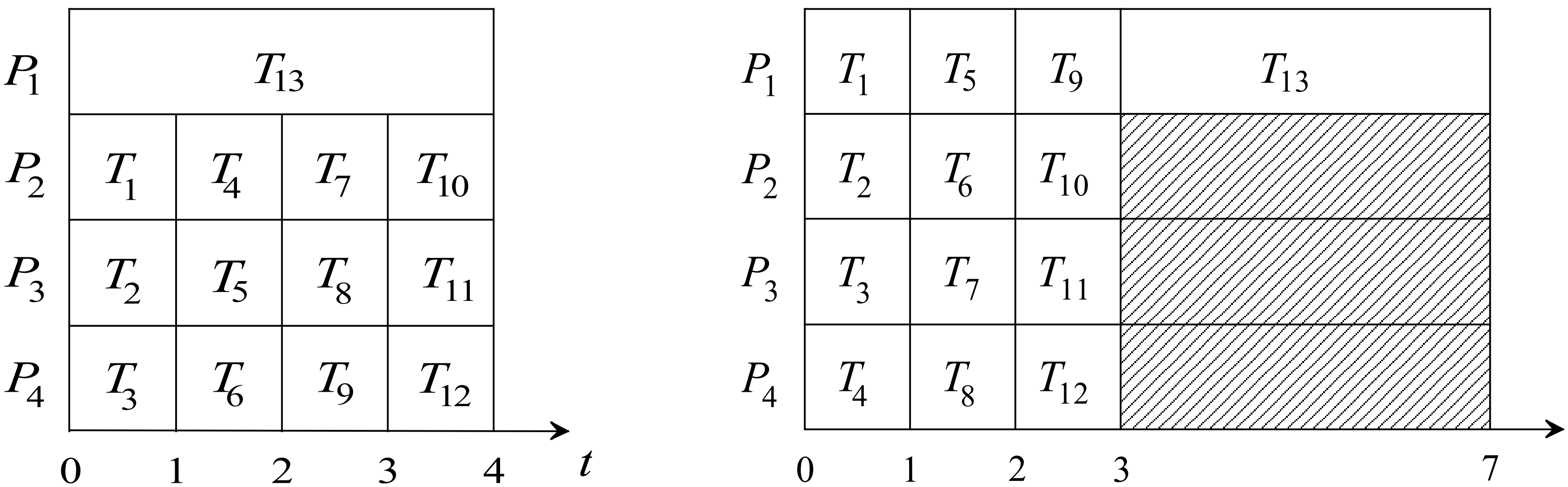
(a) *an optimal schedule,* **(b)** *LPT schedule.*

Identical Processors. *LPT Algorithm for P | | Cmax*

Example: $n = (m - 1)m + 1$, $\boldsymbol{p} = [1, 1,\ldots,1, 1, m]$, $\prec$ is empty,

$L = (T_n\ , T_1\ , T_2\ ,\ldots,T_{n-1})$, $L' = (T_1\ , T_1\ ,\ldots,T_n)$.

The corresponding schedules for $m = 4$



(a) an optimal schedule, (b) an approximate schedule

Preemptions

Problem $P \mid pmtn \mid C_{max}$

- relax some constraints imposed on problem $P \mid\mid C_{max}$ and allow preemptions of tasks.
- It appears that problem $P \mid pmtn \mid C_{max}$ can be solved very efficiently.

It is easy to see that the length of a preemptive schedule cannot be smaller than the maximum of two values:

- the maximum processing time of a task and
- the mean processing requirement on a processor:

The following algorithm given by McNaughton (1959) constructs a schedule whose length is equal to C^*_{max} .

$$C^*_{\max} = \max\{\max_j \{p_j\}, \frac{1}{m}\sum_{j=1}^{n} p_j\} .$$

Identical Processors, *P* | *pmtn* | C_{max}ю *McNaughton's rule*

Algorithm *McNaughton's rule for* $P \mid pmtn \mid C_{max}$

begin

$C^*_{max} := max\{\sum_{j=1}^{n} p_j/m, max\{p_j | j = 1,\dots,n\}\}$; -- min schedule length

$t := 0; i := 1; j := 1;$

repeat

  if $t + p_j \le C^*_{max}$

  then begin

    Assign task T_j to processor P_i , starting at time t;

    $t := t + p_j; j := j + 1$**;**

        -- assignment of the next task continues at time $t + p_j$

    end

  else begin

    Starting at time t, assign task T_j for $C^*_{max} - t$ units to P_i **;**

      -- task T_j is preempted at time C^*_{max},

      -- assignment of T_j continues on the next processor at time 0

    $p_j := p_j - (C^*_{max} - t); t := 0; i := i + 1;$

    end;

until $j = n$;   -- all tasks have been scheduled

end;

Identical Processors, $P \mid pmtn \mid C_{max}$

Remarks: The algorithm is optimal. Its time complexity is $O(n)$

Question of practical applicability:

Generally preemptions are not free of cost (delays)

Generally, two kinds of preemption costs have to be considered: time and finance.

Time delays are not crucial if the delay caused by a single preemption is small compared to the time the task continuously spends on the processor

Financial costs connected with preemptions, on the other hand, reduce the total benefit gained by preemptive task execution; but again, if the profit gained is large compared to the losses caused by the preemptions the schedule will be more useful and acceptable.

Identical Processors, *P* | *pmtn* | C_{max}

k-preemptions: Given $k \in I\!N$; (The value for *k* (*preemption granularity*) should be chosen large enough so that the time delay and cost overheads connected with preemptions are negligible).

- Tasks with processing times less than or equal to *k* are not preempted
- Task preemptions are only allowed after the tasks have been processed continuously for *k* time units

For the remaining part of a preempted task the same condition is applied

If $k = 0$: the problem reduces to the "classical" preemptive scheduling problem.

If for a given instance *k* is larger than the longest processing time among the given tasks: no preemption is allowed and we end up with non-preemptive scheduling

Another variant is the *exact-k-preemptive* scheduling problem where task preemptions are only allowed at those moments when the task has been processed exactly an integer multiple of *k* time units

Precedence constraints

Identical Processors, $P \mid prec \mid C_{max}$

Given: task set $\mathcal{T}$ with

- vector of processing times $\boldsymbol{p}$
- precedence constraints $\prec$
- priority list L
- m *identical processors*

Let C_{max} be the length of the list schedule

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

The above parameters can be changed:

– vector of processing times $\boldsymbol{p'} \leq \boldsymbol{p}$ (component-wise),

– relaxed precedence constraints $\prec' \subseteq \prec$,

– priority list L'

– and another number of processors m'

Let the new value of schedule length be C'_{max} .

List scheduling algorithms have unexpected behavior:

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

- the schedule length for problem $P \mid prec \mid C_{max}$
-

may increase

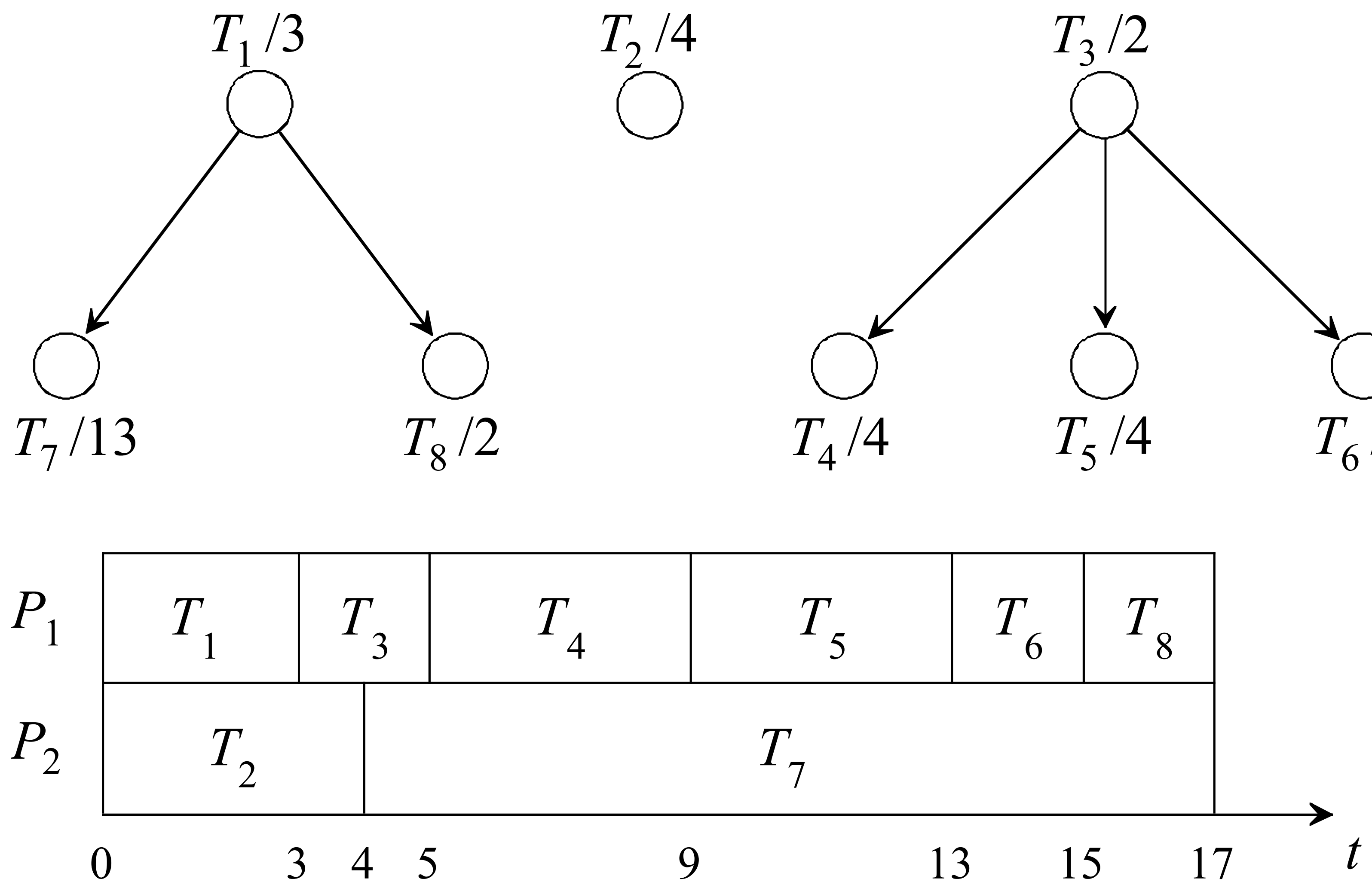
if:

- the number of processors *increases*,
- task processing times *decrease*,
- precedence constraints are *weakened*, or
- the priority list changes

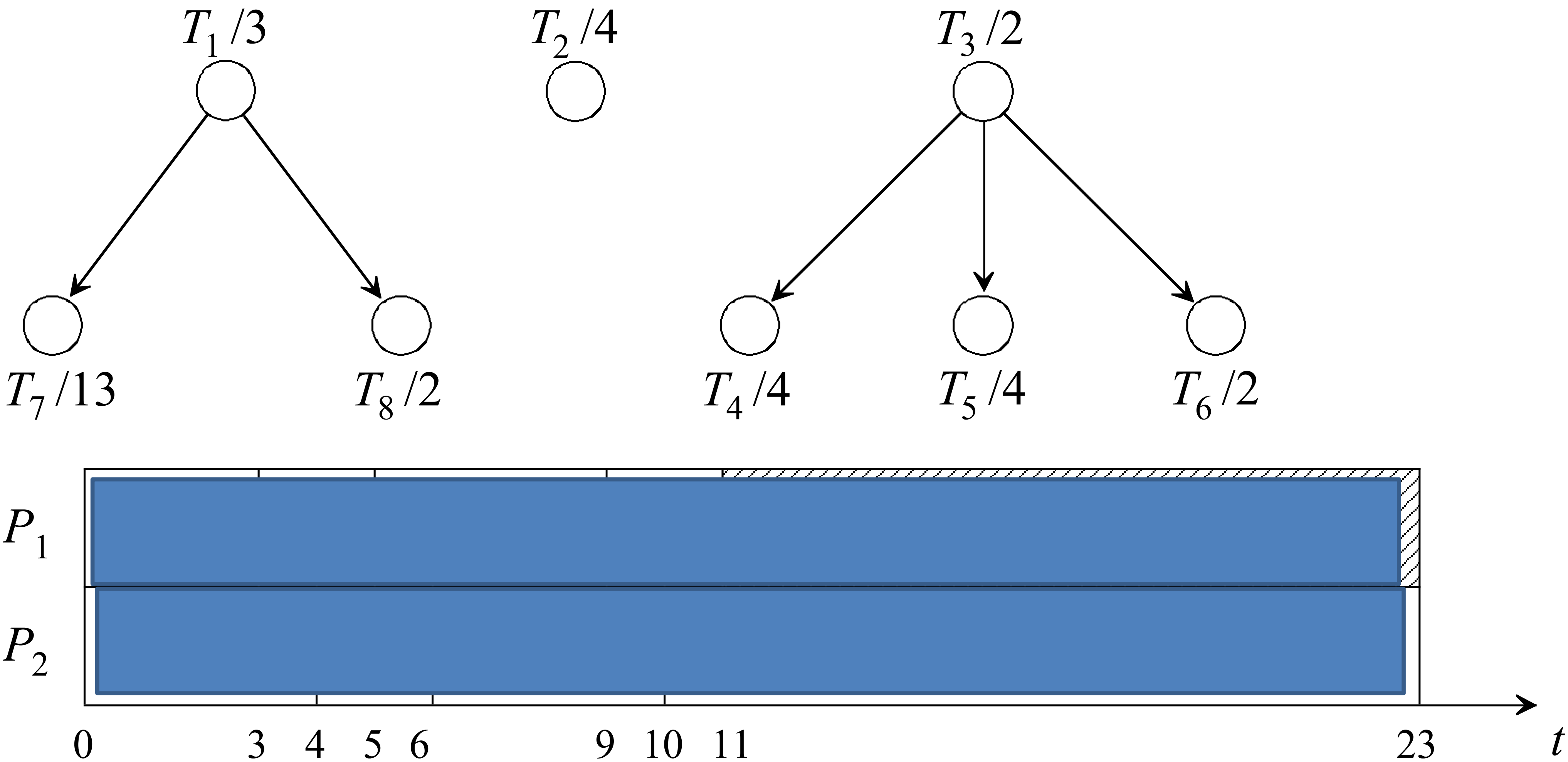
Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

(a)

(b)

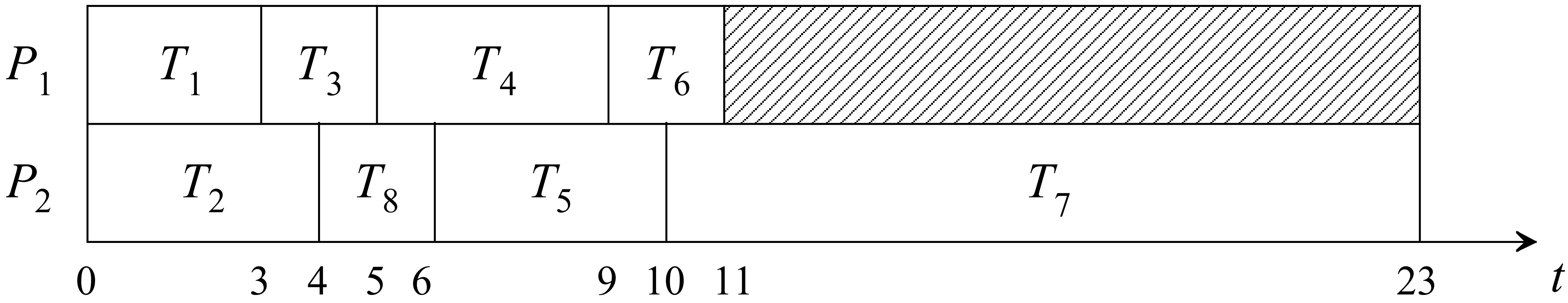


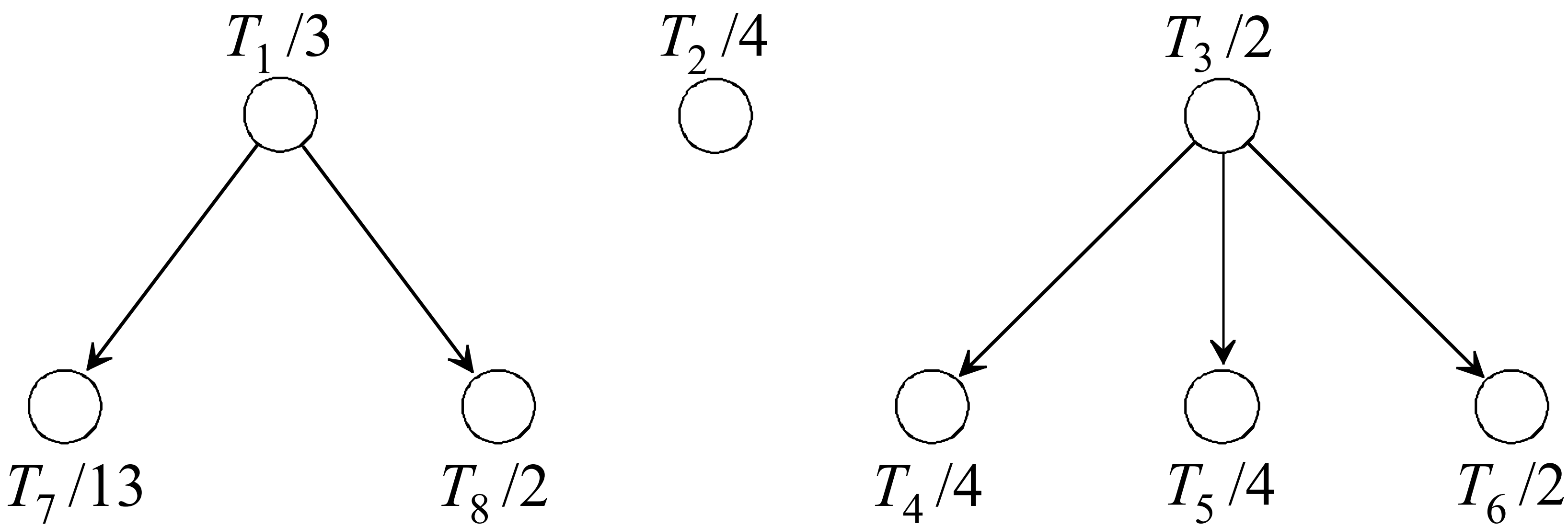
(a) *A task set*, $m = 2$, $L = (T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8)$,
(b) *an optimal schedule*



A new list $L' = (T_1, T_2, T_3, T_4, T_5, T_6, T_8, T_7)$.

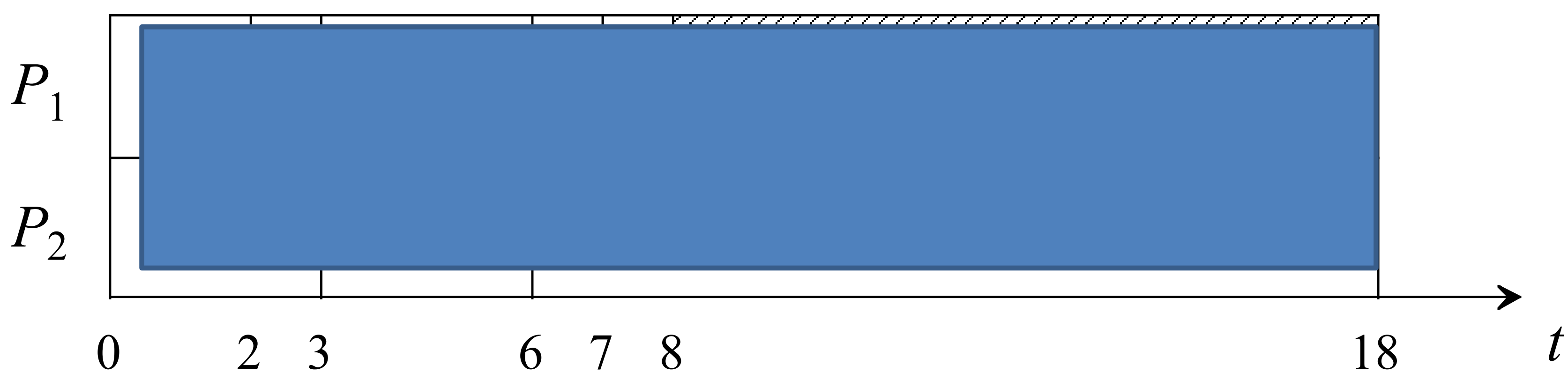
Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

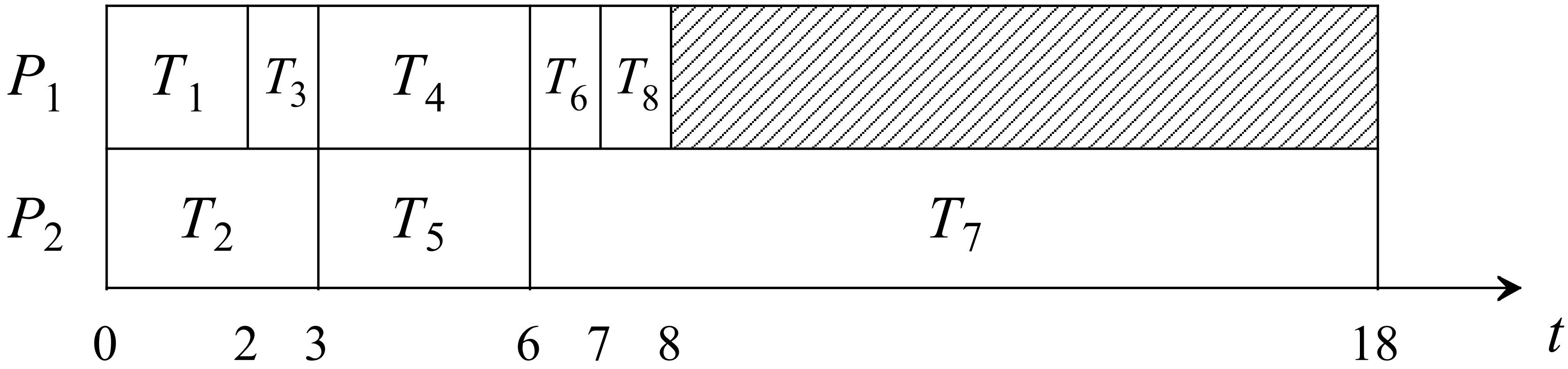




$(T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8)$.

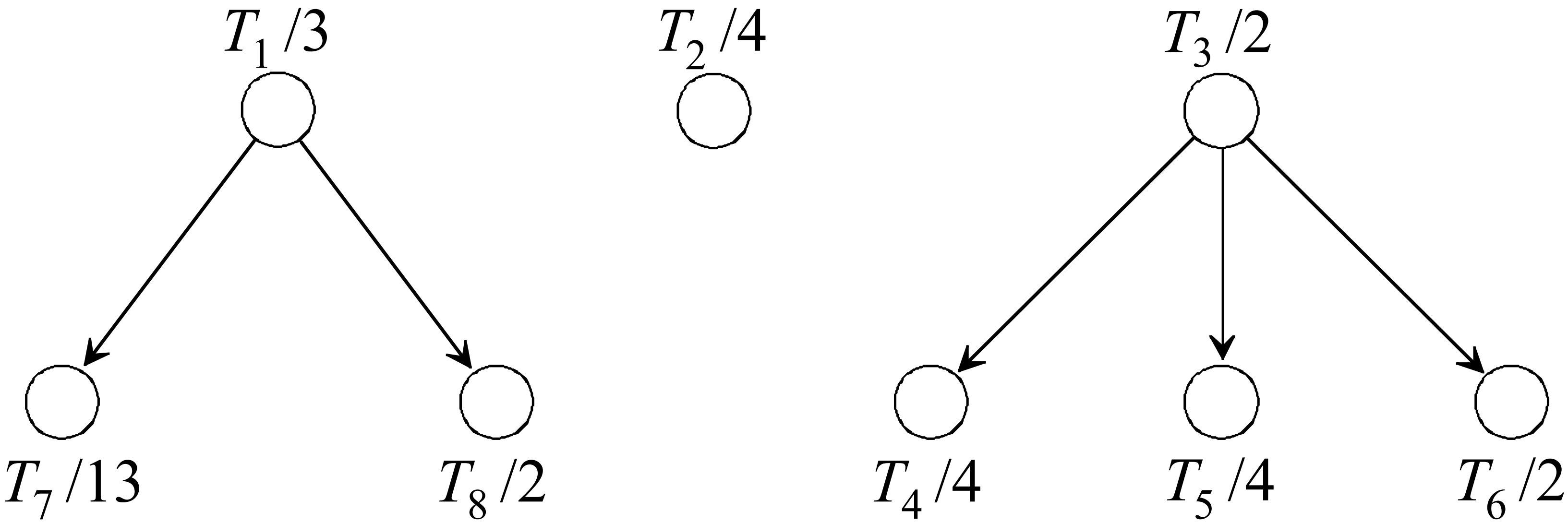
Processing times decreased; $p'_j = p_j - 1,\ j = 1, 2, ..., n.$

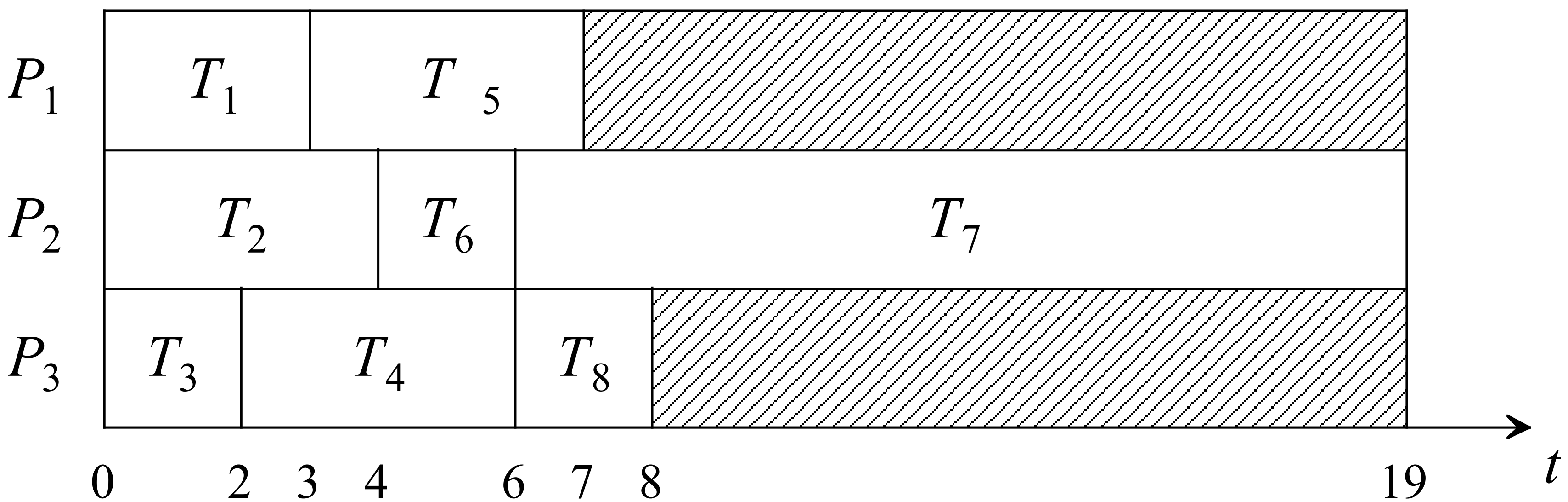




P_1
P_2
T_1
T_3
T_4
T_6
T_8
T_2
T_5
T_7
0
2
3
6
7
8
18
t

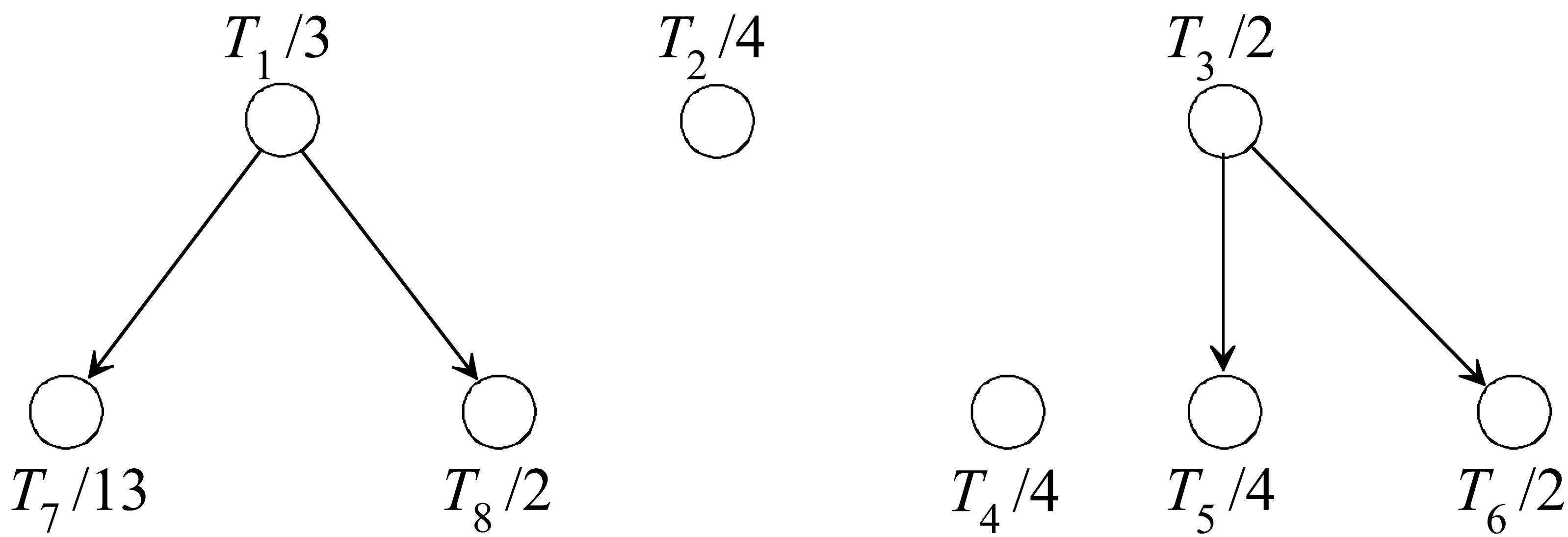
Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies





Number of processors increased, $m = 3$

(a)



(b)

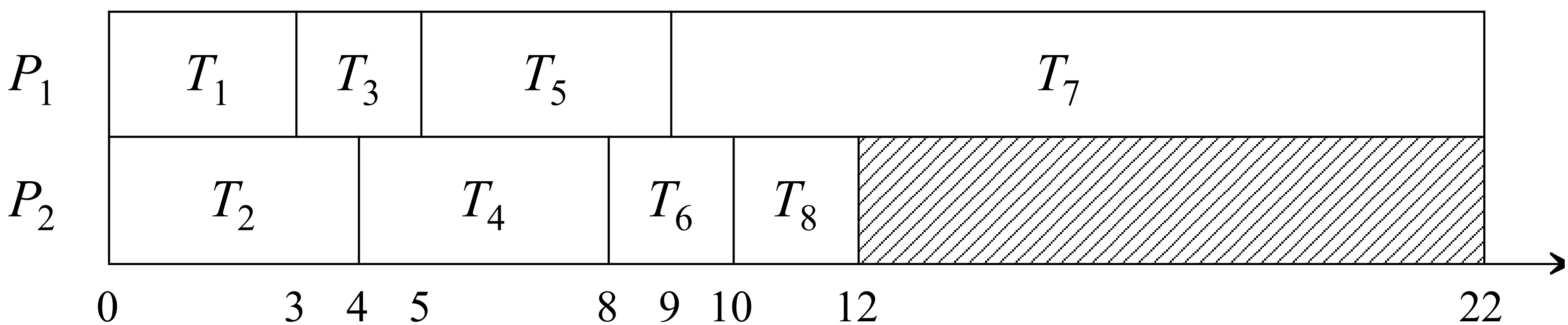


Figure 4-6 (a) *Precedence constraints weakened*, (b) *resulting list schedule.*

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

These list scheduling anomalies have been discovered by Graham [Gra66], who has also evaluated the maximum change in schedule length that may be induced by varying one or more problem parameters.

- Let the processing times of the tasks be given by vector $\boldsymbol{p}$,
- let T be scheduled on m processors using list L, and
- let the obtained value of schedule length be equal to $C_{\max}$.

On the other hand, let the above parameters be changed:

- a vector of processing times $\boldsymbol{p}' \leq \boldsymbol{p}$ (for all the components),
- relaxed precedence constraints $\prec' \subseteq \prec$,
- priority list L' and the number of processors m'.
- Let the new value of schedule length be $C'_{\max}$.

Then the following theorem is valid.

4.1.3.1 Theorem . *Under the above assumptions*,

$$\frac{C'_{max}}{C_{max}} \le 1 + \frac{m-1}{m'}$$

Proof. Let us consider schedule S' obtained by processing task set $\mathcal{T}$ with primed parameters.

Let the interval $[0, C'_{max})$ be divided into two subsets, A and B , defined in the following way:

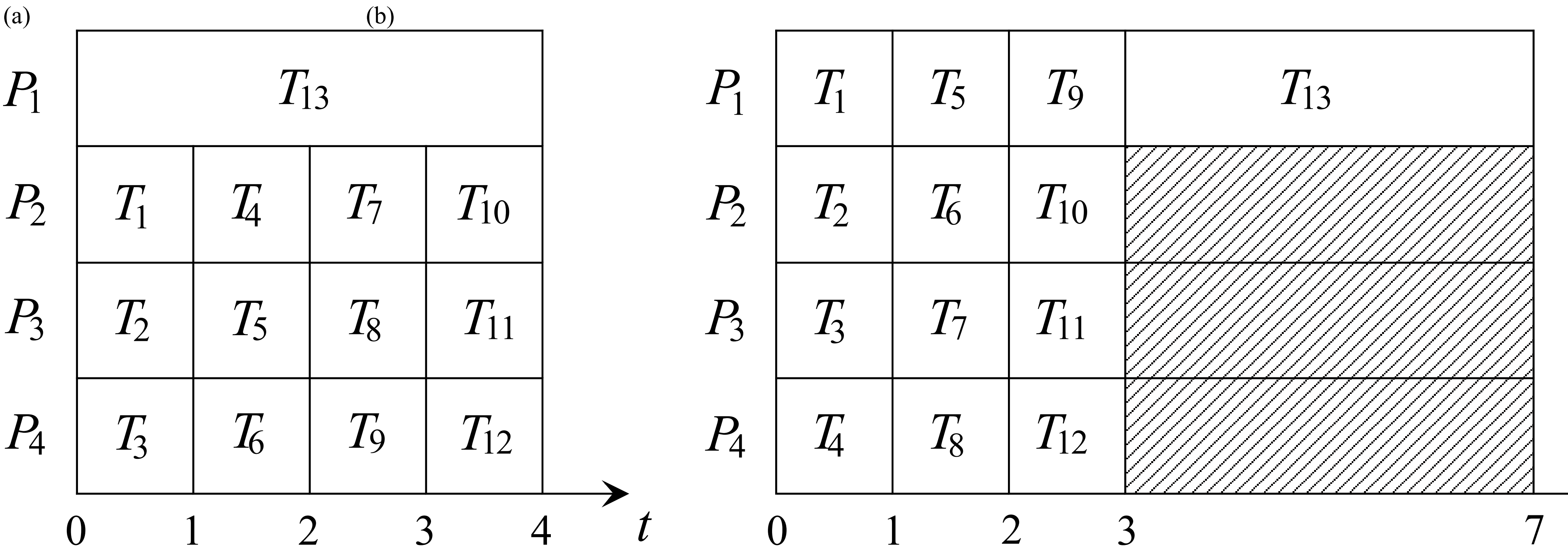
A $= \{t \in [0, C'_{max}) \mid$ all processors are busy at time $t\}$,

B $= [0, C'_{max})$ - A .

Notice that both A and B are unions of disjoint half-open intervals.

Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

Corollary (Graham 1966) *For an arbitrary list scheduling algorithm LS for* $P \mid\mid C_{max}$ *we have* $\boldsymbol{R_{LS} \le 2 - \frac{1}{m}}$ **if** $m' = m$.



Schedules for Corollary

(a) *an optimal schedule,*

(b) *an approximate schedule.*

Preemptions

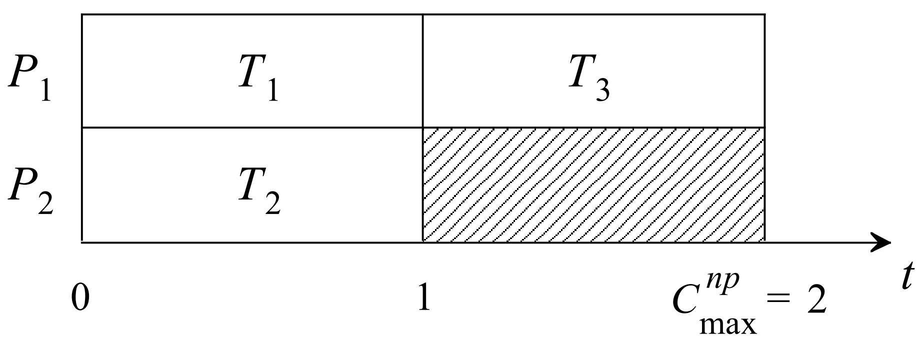
Identical Processors. $P \mid pmtn, prec \mid C_{max}$

What can be gained by allowing preemptions?

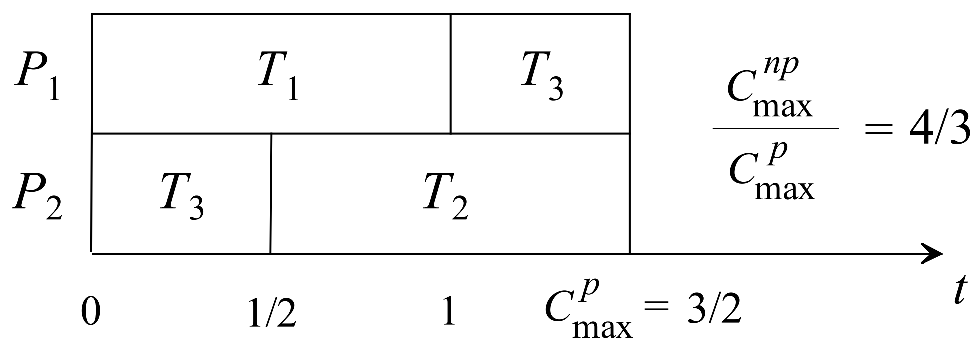
Coffman and Garey (1991) compared problems $P2 \mid prec \mid C_{max}$ and $P2 \mid pmtn, prec \mid C_{max}$: $(3/4)C_{max}^{non\text{-}preemptive} \leq C_{max}^{preemptive} \leq C_{max}^{non\text{-}preemptive}$

Example showing the (3/4)-bound (with three even independent tasks):

(a) non-preemptive schedule:



(b) preemptive schedule:



Topic 3

Scheduling on Parallel Processors

3.1 Minimizing Schedule Length
Identical Processors
Uniform and Unrelated Processors
3.2 Minimizing Mean Flow Time
Identical Processors
Uniform and Unrelated Processors
3.3 Minimizing Due Date Involving Criteria
Identical Processors
Uniform and Unrelated Processors

Model

Arrival time (or *release* or *ready time*) r_j … is the time at which task T_j is ready for processing

if the arrival times are the same for all tasks from $\mathcal{T}$, then $r_j = 0$ is assumed for all tasks

– *Due date* d_j … specifies a time limit by which T_j **should be** completed

problems where tasks have due dates are often called "soft" real-time problems. Usually, penalty functions are defined in accordance with due dates

– *Penalty functions* G_j define penalties in case of due date violations

– *Deadline* $\tilde{d}_j$ … "hard" real time limit, by which T_j **must be** completed

– *Weight* (*priority*) w_j ... expresses the relative urgency of T_j

Identical Processors. Deadline Criteria $P \mid r_j, \widetilde{d}_j \mid -$

If **deadlines** are given:

- check if a feasible schedule exists (*decision problem*)

Single processor problem $P1 \mid \boldsymbol{p_j} = 1, \boldsymbol{d_j} \mid -$ *can be solved in polynomial time*

EDF algorithm is optimal

More than one processor: most problems are known to be NP-complete

The problems

$$P \mid p_j = 1, d_j \mid - \quad \text{and} \quad P \mid prec, p_j \in \{1, 2\}, d_j \mid -$$

are NP-complete

Algorithmic approaches:

- exhaustive search
- heuristic algorithms
- approximation algorithms

Identical Processors. Deadline Criteria $P \mid r_j, \widetilde{d_j} \mid -$

Scheduling strategies:

A strategy is called "feasible", if the algorithm generates schedules where all tasks observe their deadlines (assuming this is actually possible)

three interesting deadline scheduling strategies:

EDF	Earliest Deadline First scheduling
LL	Least Laxity scheduling

-

Identical Processors. Deadline Criteria $P \mid r_j, \widetilde{d_j} \mid -$

Earliest Deadline First Scheduling Policy

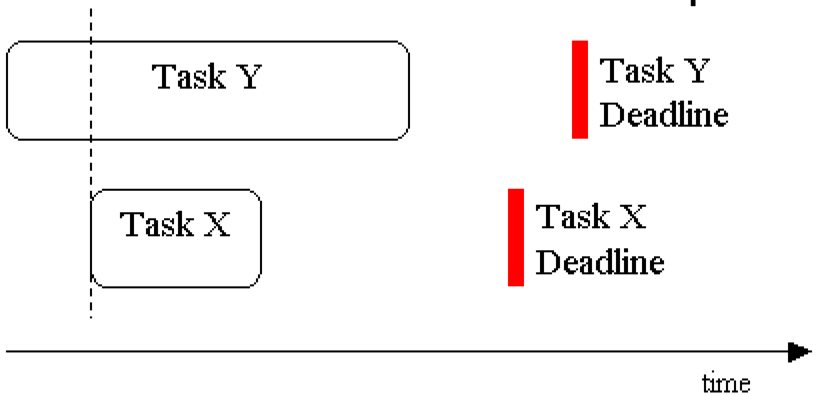
– means that the task that has the earliest deadline (task that has to be processed first) is to be scheduled next.

– EDF scheduler views task deadlines as more important than task priorities.

– Experiments have shown that the earliest deadline first policy is the most fair scheduling **algorithm**.

More complex deadline scheduler is the "**Least Laxity**" (or "**LL**") scheduler.

- takes into account both a task's deadline and its processing load,



EDF deadline scheduler would allow **Task X** to run before **Task Y**, even if **Task Y** normally has higher priority.

- However, it could cause Task Y to miss its deadline.
- So perhaps an "**LL**" scheduler would be better

Identical Processors. Deadline Criteria $P \mid r_j, \widetilde{d}_j \mid -$

Laxity is the value that describes how much computation there is still left before the deadline of the task if it ran to completion immediately. **Laxity** of a task is a measure for it's urgency.

Laxity = (Task Deadline – (Current schedule time + Rest of Task Exec. Time).

LL=D-t-Prest

It is the amount of time that the scheduler can “play with” before causing the task to fail to meet its deadline.

Least Laxity Scheduling Policy: the task that has the smallest laxity (meaning the least computation left before it's deadline) is scheduled next.

Thus, a **Least Laxity** deadline scheduler takes into account both deadline and processing load.

Identical Processors. Deadline Criteria $\boldsymbol{P} \mid \boldsymbol{r_j}, \widetilde{d_j} \mid -$

Example: *Comparison of strategies*

Set of independent tasks: T $= \{T_1, T_2, \ldots, T_6\}$
Tasks: (*deadline*, *total execution time*, *arrival time*):

T_1= (5, 4, 0), T_2= (6, 3, 0), T_3= (7, 4, 0),
T_4 = (12, 9, 2), T_5= (13, 8, 4), T_6= (15, 12, 2)

Execution on *three identical processors:*

EDF*-schedule* (no preemptions): total execution time is 16

LL-*schedule* (with preemptions): $\leq$ 8 preemptions,
total execution time is 15

optimal schedule with 3 preemptions, total execution time = 15

Execution on a *single*, *three times faster processor*:
possible with no preemptions; total execution time is 40/3

Hence: a larger number of processors is not necessarily advantageous

Identical Processors. Deadline Criteria $P \mid pmtn, r_j, \tilde{d}_j \mid -$

Feasibility testing of problem $P \mid pmtn, r_j, \tilde{d}_j \mid -$ is done by applying a network flow approach (**Horn** 1974)

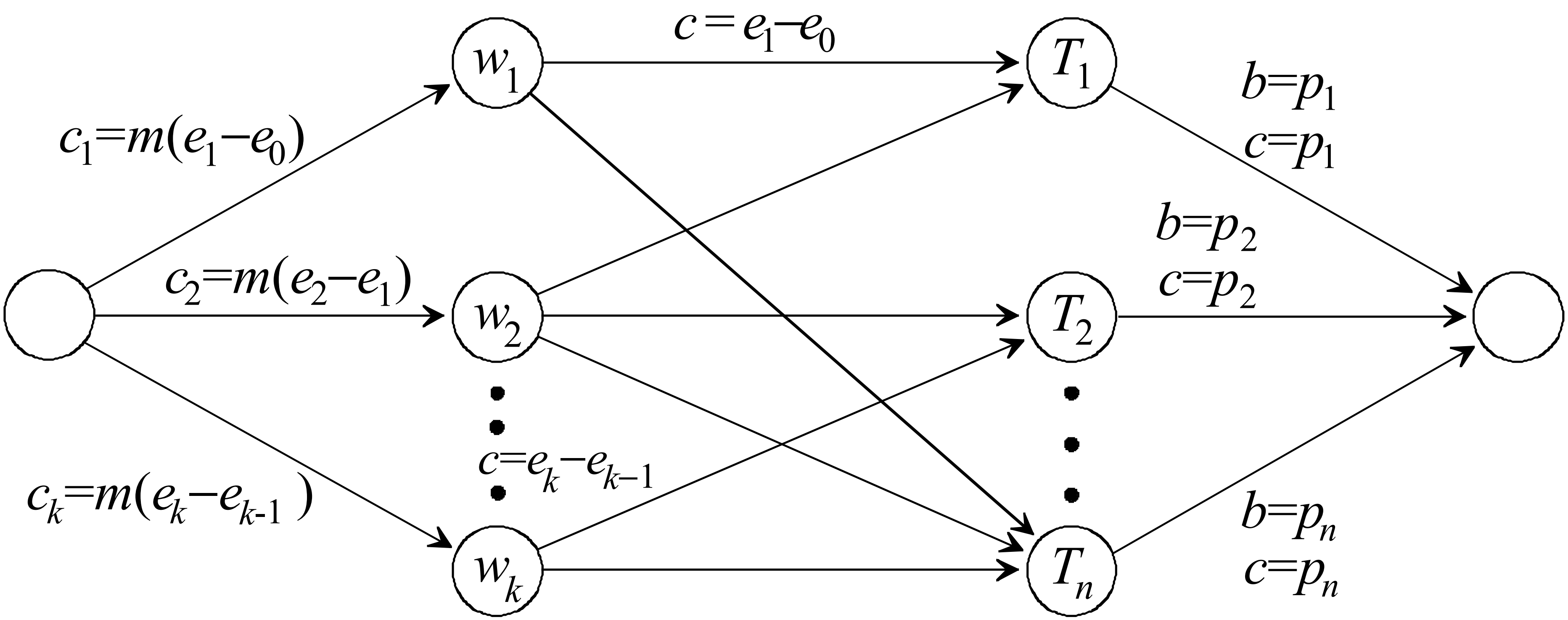
Given an instance of $P \mid pmtn, r_j, \tilde{d}_j \mid$,

let $e_0 < e_1 < \ldots < e_k$, $k \le 2n-1$ be the ordered sequence of release times and deadlines together (e_i stands for r_j or $\tilde{d}_j$) (time intervals)

Construct a network with source, sink and two sets of nodes (Figure):

- the first set (nodes w_i) corresponds to time intervals in a schedule;
  - node w_i corresponds to interval $[e_{i-1}, e_i], i = 1, 2, \ldots, k$
- the second set corresponds to the tasks

Identical Processors. Deadline Criteria $P \mid pmtn, r_j, \tilde{d}_j \mid -$



Identical Processors. Deadline Criteria ***P* | *pmtn*, $r_j, \widetilde{d_j}$| –**

Flow conditions:

– The capacity of an arc joining the source to node w_i is $m(e_i - e_{i-1})$

  ○ this corresponds to the total processing capacity of m processors in this interval

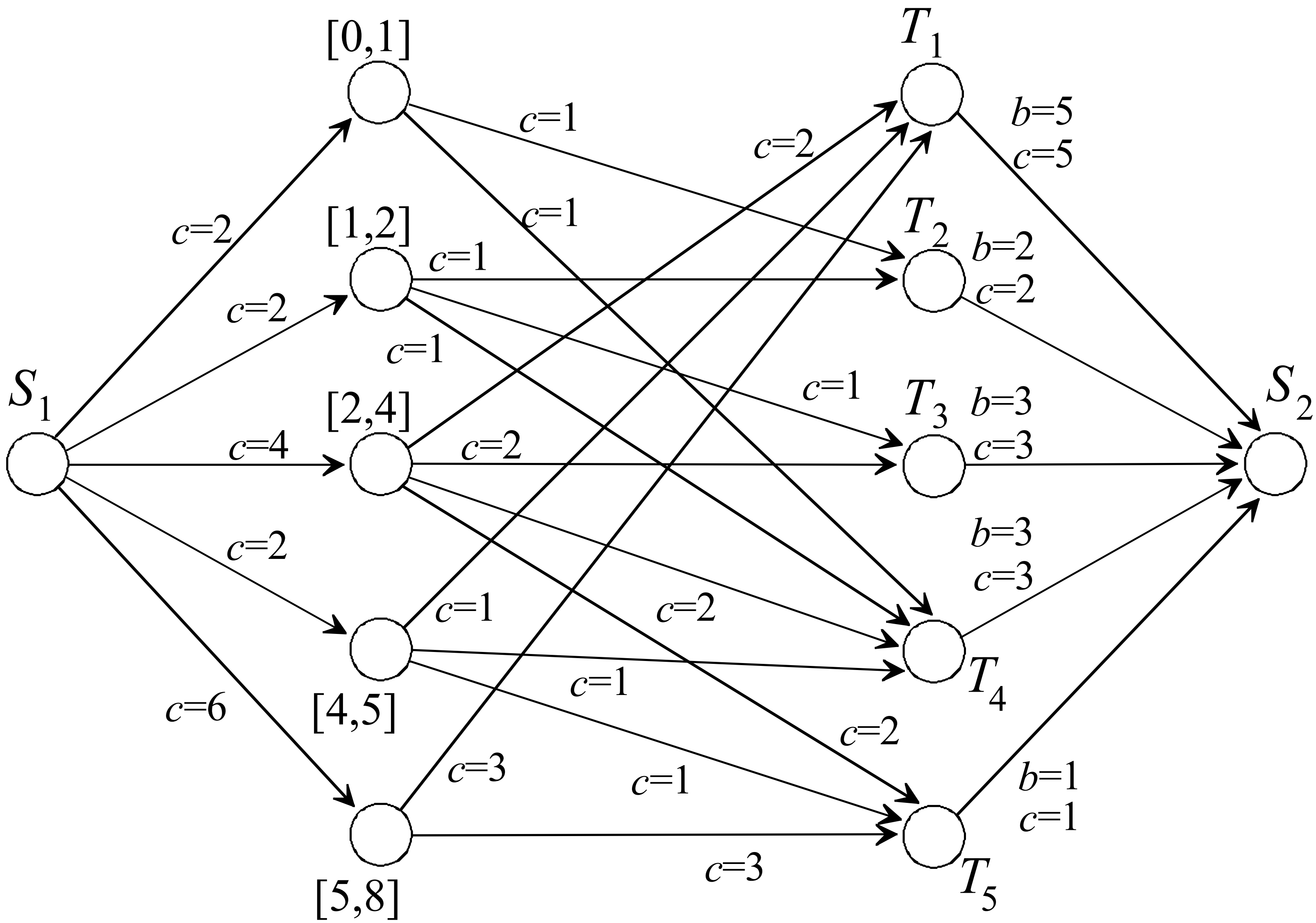
– If task T_j is allowed to be processed in interval $[e_{i-1}, e_j]$
  then w_i is joined to T_j by an arc of capacity $e_i - e_{i-1}$

– Node T_j is joined to the sink of the network by an arc with lower and upper capacity equal to p_j

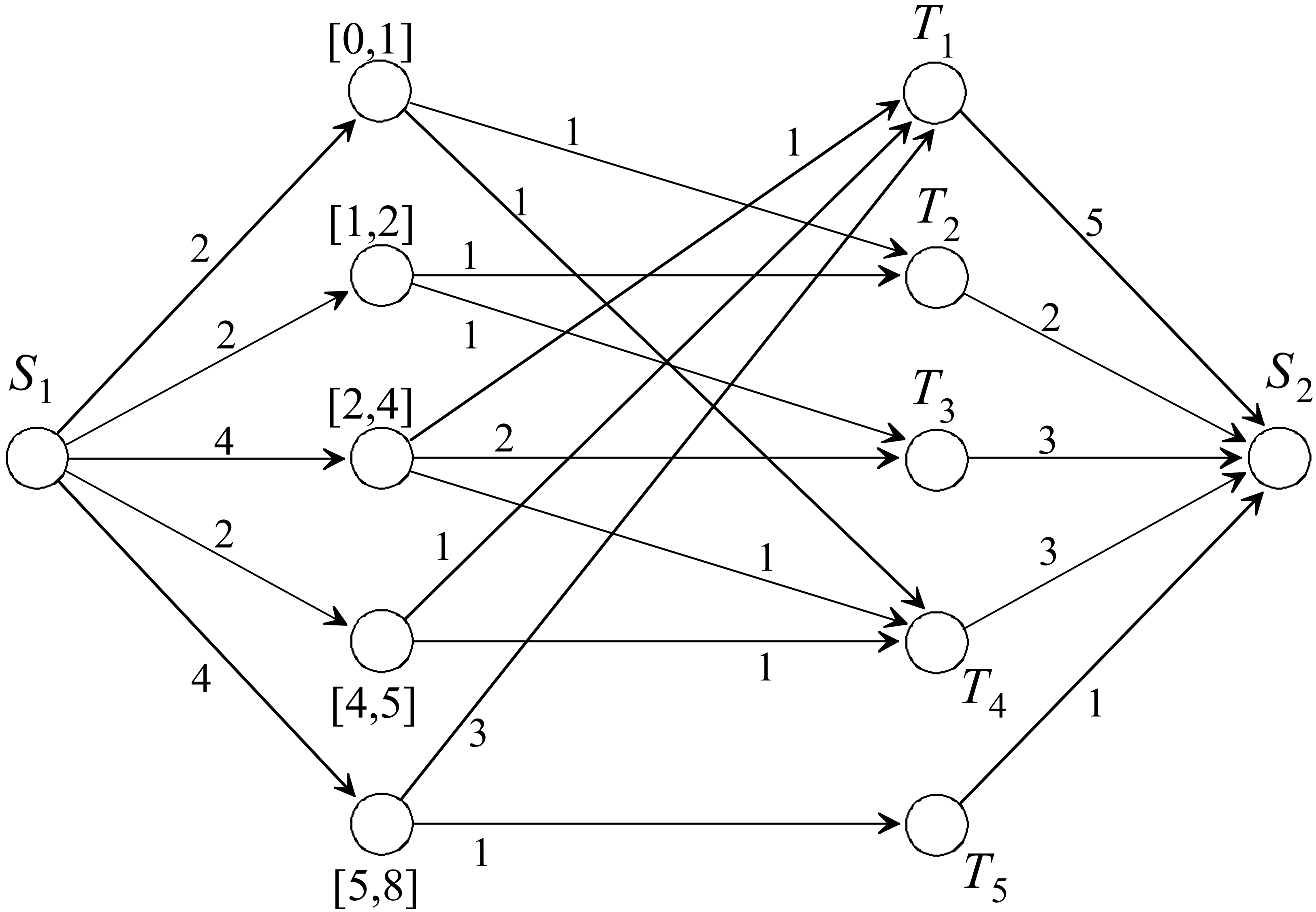
Finding a feasible flow pattern corresponds to constructing a feasible schedule; this test can be made in $O(n^3)$ time

the schedule is constructed on the basis of the flow values on arcs between interval and task nodes.

Example. $n = 5$, $m = 2$, $\boldsymbol{p} = [5, 2, 3, 3, 1]$, $\boldsymbol{r} = [2, 0, 1, 0, 2]$, and $\boldsymbol{d} = [8, 2, 4, 5, 8]$.

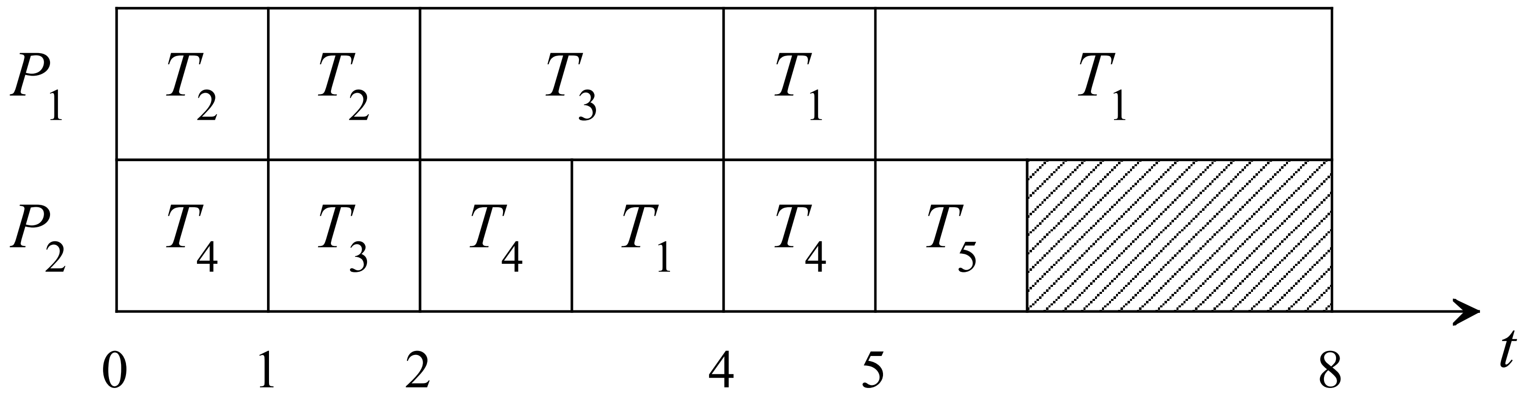


(a) corresponding network



(b) feasible flow pattern

(c) optimal schedule



Bin Packing Problem

1. Introduction

Metaphorically, there never seem to be enough bins for all one needs to store. Mathematics comes to the rescue with the *bin packing problem* and its relatives.

The bin packing problem raises the following question:

- given a finite collection of *n* weights $w_1, w_2, w_3, \ldots, w_n,$ and
- a collection of identical bins with capacity C (which exceeds the largest of the weights),
- what is the minimum number *k* of bins into which the weights can be placed without exceeding the bin capacity C?

Outline

We want to know how few bins are needed to store a collection of items.

This problem, known as the 1-dimensional bin packing problem, is one of many mathematical packing problems which are of both theoretical and applied interest.

It is important to keep in mind that "weights" are to be thought of as indivisible objects rather than something like oil or water.

For oil one can imagine part of a weight being put into one container and any left over being put into another container.

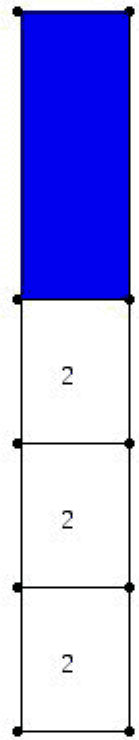
However, in the problem being considered here we are not allowed to have part of a weight in one container and part in another.

One way to visualize the situation is as a collection of rectangles which have height equal to the capacity C and a fixed width, whose exact size does not matter.

When an item is put into the bin it either falls to the bottom or is stopped at a height determined by the weights that are already in the bins.

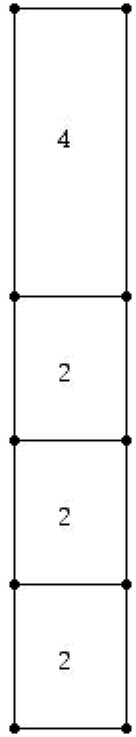
Outline

The diagram below shows a bin of capacity 10 where three identical weights of size 2 have been placed in the bin, leaving 4 units of empty space, which are shown in blue.



Outline

By contrast with the situation above, the bin below has been packed with weights of size 2, 2, 2 and 4 in a way that no room is left over.



Basic ideas

The bin packing problem asks for the minimum number *k* of identical bins of capacity C needed to store a finite collection of weights $w_1, w_2, w_3, \ldots, w_n$ so that no bin has weights stored in it whose sum exceeds the bin's capacity.

Traditionally

- capacity C is chosen to be 1 and
- weights are real numbers which lie between 0 and 1,
- for convenience of exposition, C is a positive integer and the weights are positive integers which are less than the capacity.

Example 1:

- Suppose we have bins of size 10. How few of them are required to store weights of size 3, 6, 2, 1, 5, 7, 2, 4, 1, 9?

Basic ideas

The weights to be packed above have been presented in the form of a *list* L ordered from left to right.

For the moment we will seek procedures (algorithms) for packing the bins that are "driven" by a given ***list* L** and a **capacity size C** for the bins.

The goal of the procedures is to **minimize the number of bins** needed to store the weights.

A variety of simple ideas as to how to pack the bins suggest themselves.

One of the simplest approaches is called *Next Fit* (NF).

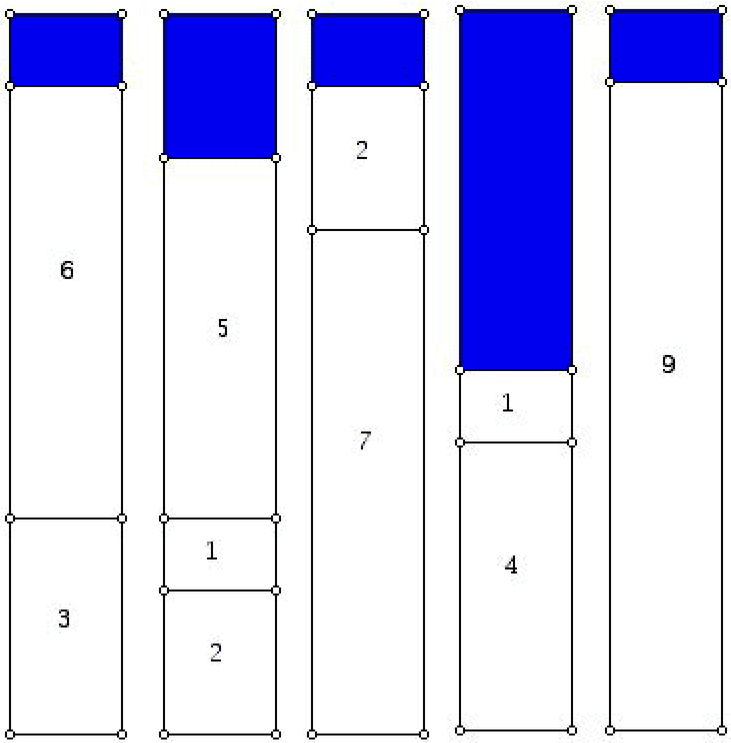
The idea behind this procedure is to open a bin and place the items into it in the order they appear in the list.

If an item on the list will not fit into the open bin, we close this bin permanently and open a new one and continue packing the remaining items in the list.

Basic ideas *Next Fit* (NF)

If some of the consecutive weights on the list exactly fill a bin, the bin is then closed and a new bin opened.

When this procedure is applied to the list above we get the packing shown below.



Basic ideas *Next Fit* (NF)

Next Fit is

- very simple,
- allows for bins to be shipped off quickly, because even if there is some extra room in a bin, we do not wait around in the hope that an item will come along later in the list which will fill this empty space.

One can imagine having a fleet of trucks with a weight restriction (the capacity C) and one packs weights into the trucks.

If the next weight cannot be packed into the truck at the loading dock, this truck leaves and a new truck pulls into the dock.

We keep track of how much room remains in the bin open at that moment.

In terms of how much time is required to find the number of bins for *n* weights, one can answer the question using a procedure that takes a linear amount of time in the number of weights (*n*).

Clearly, NF does not always produce an optimal packing for a given set of weights. You can verify this by finding a way to pack the weights in Example 1 into 4 bins.

Basic ideas *Next Fit* (NF)

Procedures such as NF are sometimes referred to as *heuristics* or *heuristic algorithms* because although they were conceived as ways to solve a problem optimally, they do not always deliver an optimal solution.

Can we find a way to improve on NF so as to design an algorithm which will always produce an optimal packing?

A natural thought would be that if we are willing to keep bins open in the hope that we will be able to fill empty space with items later in list L, we will typically use fewer bins.

Basic ideas *First Fit* (FF)

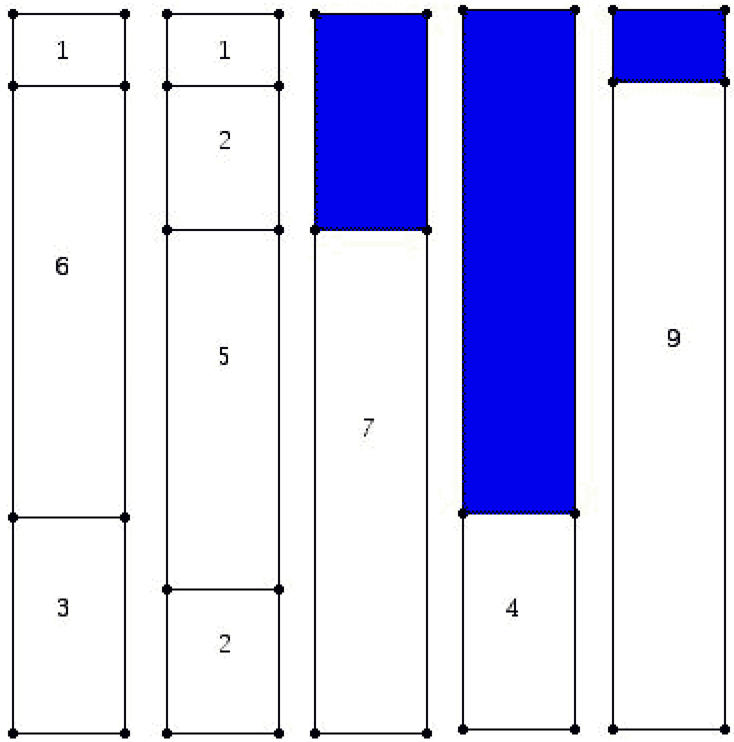
The simplest way to carry out this idea is known as *First Fit*.

We place the next item in the list into the first bin which has not been completely filled (thought of as numbered from left to right) into which it will fit.

- When bins are filled completely they are closed,
- If an item will not fit into any currently open bin, a new bin is opened.

Basic ideas *First Fit* (FF)

The result of carrying out First Fit for the list in Example 1 and with bins of capacity 10 is shown below:



Basic ideas *First Fit* (FF)

Both methods we have tried have yielded 5 bins.

We know that this is not the best we can hope for.

One simple insight is obtained by computing the total sum of the weights and dividing this number by the capacity of the bins.

Since we are dealing with integers, the number of bins we need must be at least $\lceil \Omega / C \rceil$ where $\Omega = \sum_{i=1}^{n} w_i$.

(Note that $\lceil x \rceil$ denotes the smallest integer that is greater than or equal to *x*).

Clearly, the number of bins must always be an integer. In Example 1, since Ω is 40 and C is 10, we can conclude that there is hope of using only 4 bins.

However, neither Next Fit nor First Fit achieves this value with the list given in Example 1. Perhaps we need a better procedure.

Basic ideas *Best Fit* (BF) and *Worst Fit* (WF)

Two other simple methods in the spirit of Next Fit and First Fit have also been looked at.

These are known as *Best Fit* (BF) and *Worst Fit* (WF).

For **Best Fit**, one again keeps bins open even when the next item in the list will not fit in previously opened bins, in the hope that a later smaller item will fit.

The criterion for placement is that we put the next item into the currently open bin (e.g. not yet full) which leaves the least room left over. (In the case of a tie we put the item in the lowest numbered bin as labeled from left to right.)

For **Worst Fit**, one places the item into that currently open bin into which it will fit with the most room left over.

The amount of time necessary to find the minimum number of bins using either FF, WF or BF is higher than for NF. What is involved here is *n* log *n* implementation time in terms of the number *n* of weights.

The distinction between First Fit, Best Fit and Worst Fit:

- suppose that we currently have only 3 bins open with capacity 10
- *remaining space* as follows:
  - Bin 4, 4 units,
  - Bin 6, 7 units, and
  - Bin 9 with 3 units.

Suppose the next item in the list has size 2.

First Fit puts this item in Bin 4, Best Fit puts it in Bin 9, and Worst Fit puts it in Bin 6!

One difficulty is that we are applying "good procedures" but on a "lousy" list. If we know all the weights to be packed in advance, is there a way of constructing a good list?