CICESE Research Center
Ensenada, Baja California, México
UDELAR, Montevideo

Algoritmos y métodos de calendarización Optimization in Cluster, Grid y Cloud computing

2022

Topic 1:	**Preliminaries**
Topic 2:	**Scheduling on Parallel Processors**
Topic 3:	**Scheduling Multiprocessor Tasks**

Dr. Andrei Tchernykh
https://usuario.cicese.mx/~chernykh/

Outline

Outline

References

1. J. Blazewicz, K. Ecker, G. Schmidt, J. Weglarz, Scheduling in Computer and Manufacturing Systems, Springer, pp. 495, 2001 ISBN:3540419314

2. Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Edited by Joseph Y-T. Leung. Published by CRC Press, Boca Raton, FL, USA, 2004

3. J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, J. Weglarz, Handbook on Scheduling. From Theory to Applications, Springer, pp. 647, 2007 ISBN:978-3-540-28046-0

Topic 1: Preliminaries

1.1 Application Area:

- Scheduling in Processor and Operating Systems
- Production Scheduling
- Technical and Industrial Processes
- Control Systems

1.2 Basic Notions

1.3 Scheduling Models

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

Application Area: *Scheduling in Processor and Operating Systems*

In *operating systems* there are often hundreds of processes waiting to get access to the processor

Following some implemented strategy, the scheduler decides which process gets the next access

Depending on the particular implementation the strategy takes various parameters into account, such as

- priority of the process
- its parent priority
- already consumed CPU time
- assigned resources

The scheduler (process dispatcher) is designed to optimize some system performance:

- optimizing throughput: maximize the number of completed processes per time unit
- minimizing the makespan of specified processes
- maximizing profit for the owner of the machine

Application Area: *Scheduling in Processor and Operating Systems*

Questions regarding the scheduling of activities in computers occur at **different levels**:

- *Inside processors*: sequencing of micro-operations; pipelining
- scheduling strategies in *single processor operating systems*:
  - round robin
  - priority based dispatcher algorithms
  - multilevel strategies
- *multiprocessor systems*, consisting of a CPU, co-processors, and I/O processors:
  process handling,
  assignment of activities to the special purpose processors
- *parallel processing* on a large number of identical processors as in massive parallelism:
  Work distribution, while taking into account the network connectivity and communication delays

Application Area: *Scheduling in Processor and Operating Systems*

- *distributed processing* (several computers (workstations, PC's, etc.) are connected in a local area network (LAN), or Grids:

  Applications like computer integrated manufacturing:

  - accesses to scarce network resources
  - sequencing the activities

  need sophisticated scheduling strategies

- *real-time operating systems* in parallel or distributed systems need careful handling of activities with deadlines

Application Area: *Production Scheduling*

Another example of practical interest concerns *production systems*

Typical in this area is the demand for optimal working plans for assembly lines and for flexible manufacturing machines, e.g. in production cells

General requirements:

- production due dates
- resource balancing
- maximal production throughput
- minimum storage cost

Application Area: *Production Scheduling*

Examples:

- Control of **robot movement** has to deal with optical and other data, and concerns the real time coordination of moving the arm(s)
- **Assembly lines** are of pipeline structure; their optimal design leads to flow shop problems
- Organizing flexible **manufacturing machines** leads to problems of optimizing lot sizes under the requirement of optimal throughput while minimizing overhead due to tool change delays and other setup costs
- Optimal **routing** of automated guided vehicles (AGV's) leads to questions that again require careful planning and sequencing

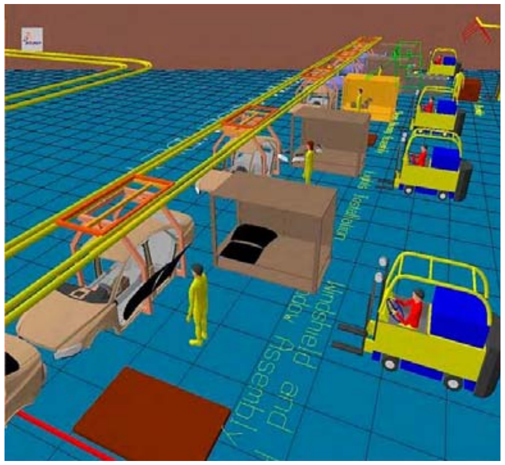
In a manufacturing environment deterministic scheduling is also known as ***predictive***

Its complement is ***reactive*** *scheduling*, which can also be regarded as deterministic scheduling with a shorter planning horizon

Application Area: *Technical and Industrial Processes*

Computer-integrated manufacturing (CIM) is a method of manufacturing in which the entire production process is controlled by computer.

Typically, it relies on closed-loop control processes, based on real-time input from sensors. It is also known as **flexible design and manufacturing**



Application Area: *Technical and Industrial Processes*

Activities from

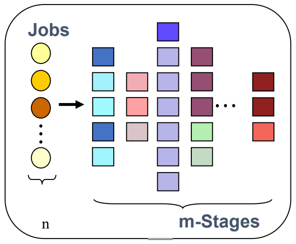
- production planning
- computer aided design
- work planning
- manufacturing
- quality control

have to be coordinated

The objectives are similar: better capacity planning, maximal throughput, minimum storage cost, etc.

Total number of possible solutions

$$n!\left(\prod_{i=1}^{m} m_i\right)^n$$

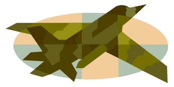


Application Area: *Control Systems*

In *real-time systems* the particular situation dictates conditions different from those before:

some processes must be activated *periodically* with a fixed rate, and others have to meet given *deadlines*

In such systems, meeting the deadlines can be a crucial condition for the correct operation of the environment



Examples of application areas are

- aircraft control,
- power plants, heat control, turbine speed control,
- frequency and voltage stabilization etc.,
- security systems in transportation systems such as air bags and ABS

Basic Notions

Basic Notions. Introduction

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.

Basic Notions. Introduction

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

Basic Notions. Introduction

☞ ***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

The deterministic scheduling or planning problems arising in different applications have often strong similarities

hence essentially the same basic model can be used

Common aspects in these applications:

processes consist of complex activities to be scheduled
they can be modeled by means of tasks or jobs

- Tasks usually need one of the **available *machines***, maybe even a special machine, and additional *resources* of limited availability
- 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

Deterministic Scheduling Problems

- **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 for a given set of tasks

An important condition describes the intended properties of a schedule, as defined by 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

Deterministic Scheduling Problems

Because of the complexity nature the theory deals with simplified models, and, when dealing with practical problems, rather improper simplifications are made in the corresponding models

as a consequence, there is a big gap between practice and theory

The question arises whether or not the theory of scheduling is of any use for the practice

Hence we are faced with principal questions like

– what can we gain from theory?

– what can theoretical solutions tell us for the application?

– is the still huge effort for solving theoretical problems justified?

The Scheduling Model

The Scheduling Model

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

The Scheduling Model. Deterministic Model

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

The Scheduling Model. Deterministic Model

Parallel processors have the same execution capabilities

Three types of **parallel processors** are distinguished

- o *identical*: if all processors from set $\mathcal{P}$ have equal task processing speeds
- o *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 $\mathcal{T}$
- o *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

The Scheduling Model. Deterministic Model

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

The Scheduling Model. Deterministic 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

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

The Scheduling Model. Deterministic Model

In manufacturing environments tools, material, transport facilities, etc. can be treated as additional resources

The resources considered here are assumed to be *discrete* and *renewable*

Assumption: ***s types*** of additional resources $R_1, R_2, \dots, R_s$ are available in respectively $m_1, m_2, \dots, m_s$ units

Each task T_j requires for its processing one processor and certain fixed amounts of these additional resources:

resource requirement vector $R(T_j) = [R_1(T_j), R_2(T_j), \dots, R_s(T_j)]$

$R_l(T_j)$ denotes the number of units of resource R_l required

for the processing T_j $(0 \le R_l(T_j) \le m_l,\ l = 1,2,\dots,s)$

Obviously the situation may occur that, due to resource limitations, subsets of tasks cannot be processed at the same time. All required resources are granted to a task before its processing begins or resumes (in the case of preemptive scheduling), and they are returned by the task after its completion or in the case of its preemption

The Scheduling Model. Deterministic Model

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*

The Scheduling Model. Deterministic Model

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

- o chain dependencies: the partial order is the union of linearly ordered disjoint subsets of tasks
- o 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

The Scheduling Model. Deterministic Model

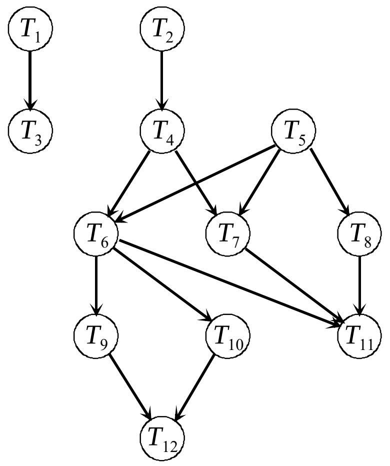
Representation of tasks with precedence constraints:

– *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.

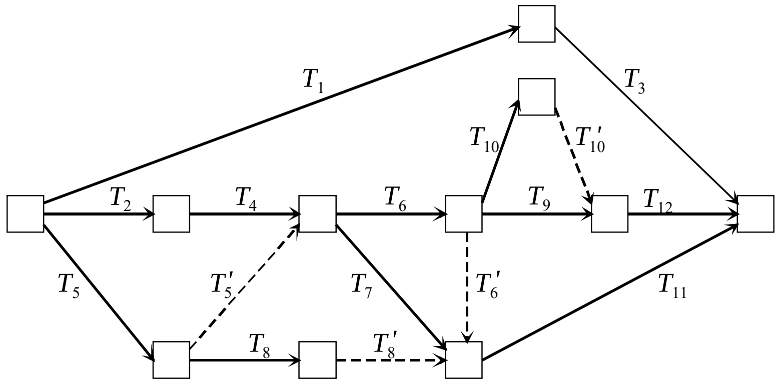
Transitive dependencies are not explicitly represented

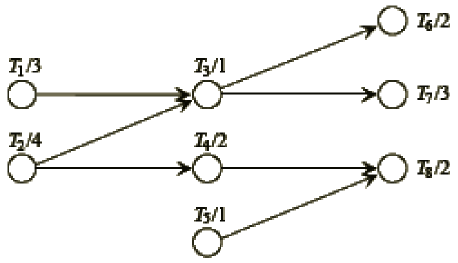


The Scheduling Model. Deterministic Model

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

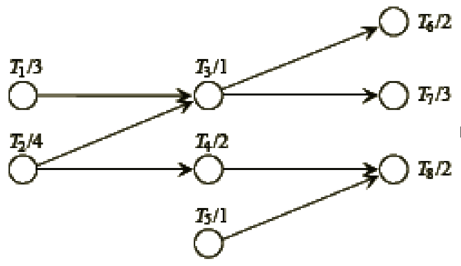
Example 1: $\mathcal{T} = \{T_1, ..., T_{10}\}$ with precedences as shown by the above Hasse diagram. A corresponding activity network:



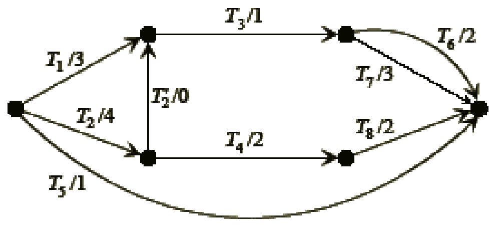


Task-on-node

task-on-arc graph ?????



Task-on-node



Task-on-arc

Error????

The Scheduling Model. Deterministic Model

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

The Scheduling Model. Deterministic Model

A *schedule* $\mathcal{S}$ is an assignment of processors to the tasks from $\mathcal{T}$ (or an assignment of the tasks to the processors) such that:

– task T_j is processed in the time interval $[r_j, \infty)$ for p_j time units,

– all tasks are completed,

– at each instant of time, each processor works on at most one task,

– at each instant of time, each task is processed by at most one processor,

– if tasks T_i, T_j are in relation $T_i \prec T_j$ then the processing of T_j is not started before T_i has been completed,

– if T_j is non-preemptive then processing of T_j is not interrupted;
if T_j is preemptive then T_j may be interrupted only a *finite* number of times

If all tasks are non-preemptive then the schedule is called *non-preemptive*
If all tasks are preemptive, then the schedule is called *preemptive*

The Scheduling Model. *Schedule representation 1*

(1) One possibility to describe schedules is by means of a
function $\varsigma^{\mathbb{R}}: \mathbb{R}^{\geq 0} \rightarrow (\mathrm{T} \cup \{\Lambda\})^m$

the non negative real number values of $\mathbb{R}^{\geq 0}$ are interpreted as *time*

Λ denotes an *idle task*, which describes the possibility that one or more processors are not active

Function $\varsigma^{\mathbb{R}}$ specifies for each point of time a vector of tasks of length m

The ith component of this vector specifies the task processor P_i is currently working on

This way $\varsigma^{\mathbb{R}}$ defines for each point of time the activities of each processor

The Scheduling Model. *Schedule representation 1*

For practical reasons we assume that the image set of $\varsigma^{\mathbb{R}}$ is of finite cardinality
In other words, we allow only finitely many changes of activity patterns for the processors

If tasks are processed preemptively this assumption implies only *finitely* many preemptions for each task

This allows a more practical description of $\varsigma^{\mathbb{R}}$ where tuples of $\varsigma^{\mathbb{R}}(t)$ are specified only for those points of time at which the value of $\varsigma^{\mathbb{R}}$ changes
Between succeeding points of time the task assignment is then considered to be constant

In this connection it makes sense to speak about *intervals* of task assignments during which the task assignment is constant

The Scheduling Model. *Schedule representation 1*

Let $s(T_j)$ be the **start time** of T_j , i.e. the earliest point of time at which T_j occurs in one of the tuples $S^{\mathbb{R}}(t)$

Let $c(T_j)$be the **completion time** of T_j , i.e. the end point of the last interval that contains T_j

Then $\varsigma^{\mathbb{R}}$ must fulfill the following conditions:

– the sum of lengths of intervals in which T_j is processed is p_j ($j = 1, ..., n$),

– $s(T_j) \geq r_j$,

– the task in each tuple are pairwise different or Λ,

– if $T_i \prec T_j$ then $c(T_i) \leq s(T_j)$

The Scheduling Model. *Schedule representation 2*

(2) An alternative definition specifies only the start times of the tasks
This is, however, **improper** for preemptive schedules:

A *non-preemptive* schedule can be defined as a mapping $\varsigma^{T}: \mathcal{T} \rightarrow \mathbb{R}^{\geq 0} \times \mathcal{P}$;
$\varsigma^{T}(T_j) = (t, P_i)$ means that T_j is started at time t on processor P_i

Let $s(T_j)$ be the start time of T_j, and $c(T_j)$ ($= s(T_j) + p_j$) be its completion time

Then the above conditions translate into:

– $s(T_j) \geq r_j$,

– ς^{T} is total (i.e. ς^{T} specifies one tuple for each task)

– if $\varsigma^{T}(T_j) = (t, P_i)$ then no other task T' can have an image (t', P_i) with $t' \in [t, t + p_j)$,

– if $T_i \prec T_j$ then $c(T_i) \leq s(T_j)$

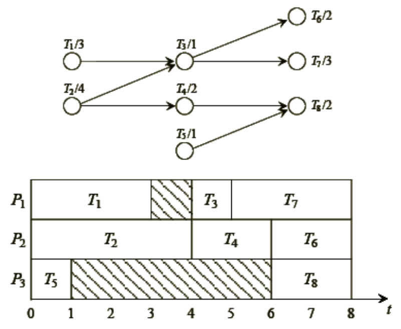
The Scheduling Model. *Schedule representation 3*

(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

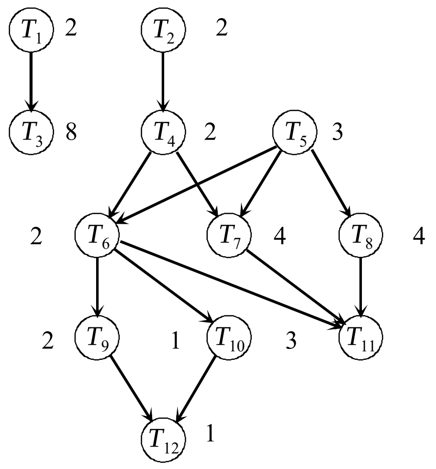
Each processor is represented by a line

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



The Scheduling Model. *Schedule representation*

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



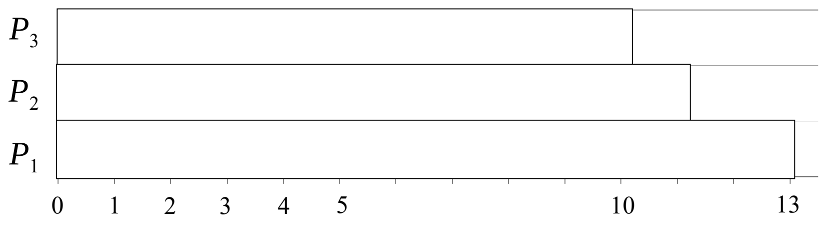
The Scheduling Model. *Schedule representation*

Example 2: non-preemptive schedule

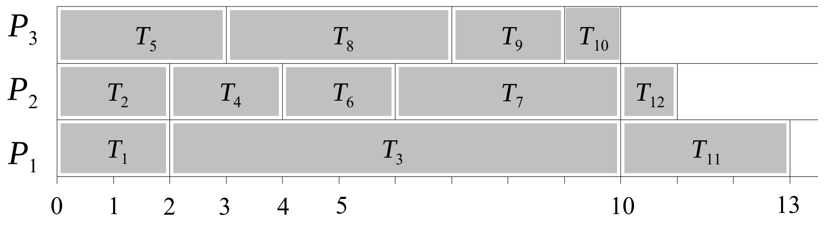
In the above example, let $(2, 2, 8, 2, 3, 2, 4, 4, 2, 1, 3, 1)$ be the vector of processing times, and assume all release times $= 0$

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

Gantt chart of a non-preemptive schedule:



The Scheduling Model. *Schedule representation*



The Scheduling Model. *Schedule representation*

The corresponding formal description by a function $\varsigma^{\mathbb{R}}: \mathbb{R}^{\geq 0} \rightarrow (\mathrm{T} \cup \{\Lambda\})^m$ is:

$\varsigma^{\mathbb{R}}(0) = \dots..$

The corresponding formal description by a function $\varsigma^{T}: \mathcal{T} \rightarrow \mathbb{R}^{\geq 0} \times \mathcal{P}$ is:

$\varsigma^{T}(T_1) = \dots..$

The corresponding formal description by a function $\varsigma^{\mathbb{R}}: \mathbb{R}^{\geq 0} \rightarrow (\mathrm{T} \cup \{\Lambda\})^m$ is:

$\varsigma^{\mathbb{R}}(0) = (T_1, T_2, T_5)$, $\varsigma^{\mathbb{R}}(2) = (T_3, T_4, T_5)$, $\varsigma^{\mathbb{R}}(3) = (T_3, T_3, T_8)$, etc.

The corresponding formal description by a function $\varsigma^{T}: \mathcal{T} \rightarrow \mathbb{R}^{\geq 0} \times \mathcal{P}$ is:

$\varsigma^{T}(T_1) = (0, P_1)$, $\varsigma^{T}(T_2) = (0, P_2)$, $\varsigma^{T}(T_3) = (2, P_1)$, etc.

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

The Scheduling Model. Deterministic Model. *Optimization Criteria*

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	$\overline{U_w} := (\Sigma\ w_j U_j) / (\Sigma\ w_j)$

The Scheduling Model. Deterministic Model. *Optimization Criteria*

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

The Scheduling Model. Deterministic Model. *Optimization Criteria*

Minimizing the mean weighted flow time $\overline{F_w}$

A schedule is $\overline{F_w}$-optimal if the mean flow time of tasks is minimized:

the average duration of residence of the tasks is as short as possible

Different *weights* for the tasks allow to express the *urgency* of tasks

The *mean flow time* criterion is important from the user's viewpoint since it yields a minimal mean response time and the mean in-process time of the scheduled task set

The Scheduling Model. Deterministic Model. *Optimization Criteria*

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

The Scheduling Model. Deterministic Model. *Optimization 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 $\overline{U_w}$

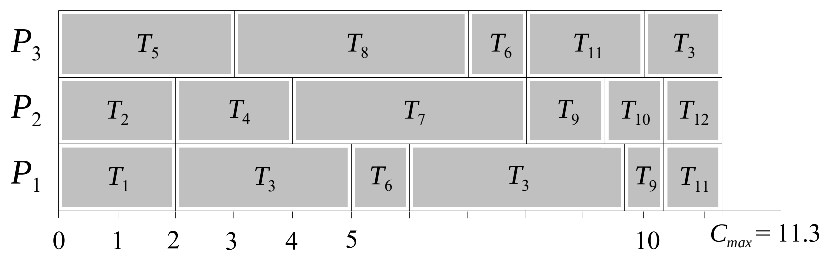
This criterion considers only the number of tardy tasks

Individual weights for the tasks are again possible

The Scheduling Model. Deterministic Model. *Optimization Criteria*

Example 3

Gantt chart of a preemptive schedule:



The Scheduling Model. Deterministic Model. *Optimization Criteria*

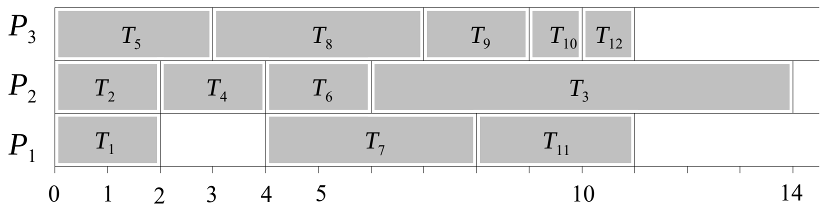
(1) In the schedule of example the flow time of tasks

$F(T_1)=2$,$F(T_2)$ = 2,F(T_3)=???, etc.

Example 4: non-preemptive schedule with due dates

For the task set as specified before, let in addition due dates be given by the vector (8, 2, 16, 4, 4, 8, 8, 8, 10, 8, 10, 11).

In the schedule below, task T_{10} with due date 8 violates its due date by two time units.



(2) In the schedule of example task T_{10} has lateness ???; for all other tasks L_j is less equal ???.

The tardiness of T_{10} = ????, and it is ??? for all other tasks;

hence $U_{10} = ???$, and $U_j = ???$ for all other tasks

(3) In the same schedule task T_1 has earliness ??, T_2has earliness ??, etc.

Examples

(1) In the schedule of example 3 the flow time of tasks T_1 and T_2 is 2, that of T_3 is 11.3, etc.

(2) In the schedule of example 4 task T_{10} has lateness 2; for all other tasks L_j is less than or equal 0. The tardiness of T_{10} , and it is 0 for all other tasks; hence $U_{10} = 1$, and $U_j = 0$ for all other tasks

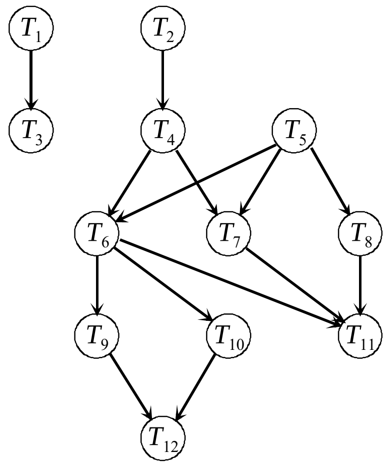
(3) In the same schedule task T_1 has earliness 6, T_2has earliness 0, etc.

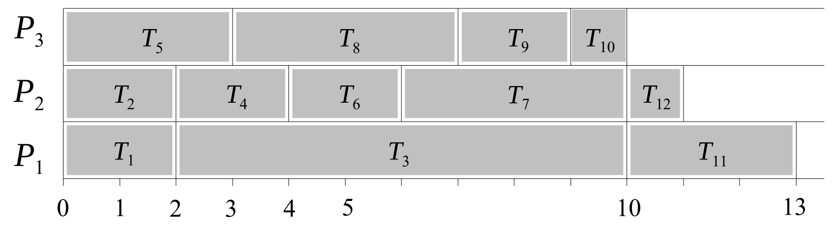
Example

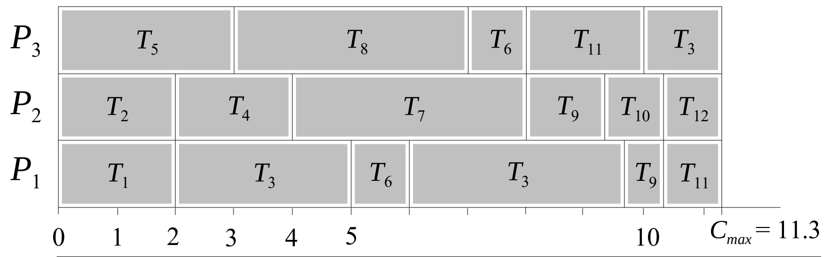
Consider the task set as in Example 1, with processing times and due dates as specified in the respective Examples 2 and 3.

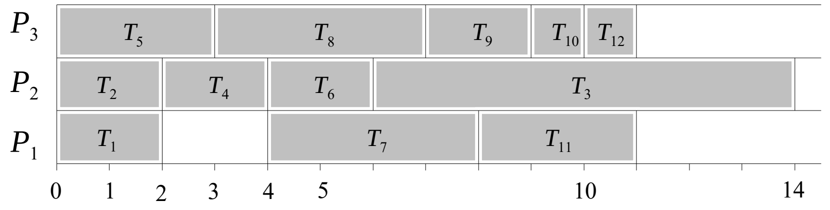
processing times (2, 2, 8, 2, 3, 2, 4, 4, 2, 1, 3, 1),

due dates (8, 2, 16, 4, 4, 8, 8, 8, 10, 8, 10, 11).









Compute the following values

Criterion	Example 2	Example 3	Example 4
C_{max}		🟨	
$\bar{F}$	🟧		🟧
L_{max}	🟩		🟩
$\bar{D}$	🟦		🟦
$\bar{E}$		🟪	
$\bar{U}$			🟥

we compute the following values r_j=0(the smallest values are shaded):

Criterion	Example 2	Example 3	Example 4
C_{max}	13.000	11.333	14.000
$\bar{F}$	7.250	7.392	7.250
L_{max}	3.000	2.333	2.000
$\bar{D}$	0.580	0.360	0.167
$\bar{E}$	1.417	1.058	1.083
$\bar{U}$	0.250	0.333	0.167

we compute the following values r_j>=0(the smallest values are shaded):

Criterion	Example 2	Example 3	Example 4
$\bar{F}$	3.33	3.27	3.5

α | β | γ - *Notation*

The Scheduling Model. *Scheduling Problems and α | β | γ - 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

The Scheduling Model. *Scheduling Problems and α | β | γ - Notation*

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		α_2	
$\varnothing$	single processor	$\varnothing$	the number of processors is assumed to be variable
P	identical processors	k	the number of processors is equal to k (k is a positive integer)
Q	uniform processors	∞	the number of processors is unlimited
R	unrelated processors		

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 $\beta_2 \in \{\varnothing, res\ \lambda\delta\rho\}$ characterizes ***additional resources***

β_2		
$\varnothing$	there are specified resource constraints	
$res\ \lambda\delta\rho$	$\lambda, \delta, \rho \in \{\cdot, k\}$ denote respectively the number of resource types, resource limits and resource requirements	
	$\lambda, \delta, \rho = \cdot$	the respective numbers of resource types, resource limits and resource requirements are arbitrary
	$\lambda, \delta, \rho = k$	respectively, each resource is available in the system in the amount of k units and the resource requirement of each task is at most equal to k units

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

uniconnected activity network (*uan*), which is defined as a graph in which any two nodes are connected by a directed path in one direction only. Thus, all nodes are uniquely ordered.

$\beta_3 = \varnothing, prec, tree, chains$: 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***

β_4	
$\varnothing$	all ready times are zero
r_j	ready times differ per task

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

β_5	
$\varnothing$	tasks have arbitrary processing times
$p_j = p$	all tasks have processing times equal to p units
$\underline{p} \le p_j \le \bar{p}$	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

The Scheduling Model. *Scheduling Problems and α | β | γ - Notation*

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^*

The Scheduling Model. *Scheduling Algorithms*

A *scheduling algorithm* for a scheduling problem $\alpha \mid \beta \mid \gamma$

constructs a schedule for each *instance* of $\alpha \mid \beta \mid \gamma$

In general, we are interested in algorithms that find *optimal* schedules with respect to γ

the above objective criteria are minimization criteria

Final remark about the presented model:

Though the model considers already quite a number of parameters, it is still very restricted

Modeling practical situations, however, mostly require the inclusion of many more parameters and conditions, in particular for the tasks

Examples are communication times, periodic tasks, coupled tasks, setup times for tasks and resources, renewable resources, multiprocessor tasks, and many more

The Scheduling Model. *Summary*

The purpose of this chapter was to introduce the basic notions in scheduling theory:

- deterministic scheduling
- scheduling model
- schedule representation and evaluation
- three-field notation

Topic 2
Scheduling on Parallel Processors

Independent tasks

Identical Processors *P* | | Cmax

The first problem considered is *P* | | Cmax where

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

Identical Processors *P* | | Cmax

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 $\rho^{seq} \leq 2 - \frac{1}{m}$.

Identical Processors. *LPT Algorithm for P | | C*max

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 | | C*max

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

□

an example showing that this bound can be achieved.

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

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

Identical Processors. *LPT Algorithm for P | | C*max

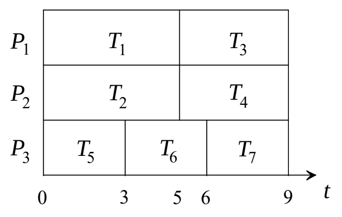
Example: m = 3 identical processors; $n = 2m + 1$,

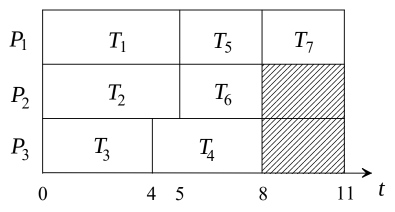
$\boldsymbol{p} = [2m - 1, 2m - 1, 2m - 2, 2m - 2, \ldots, m + 1, m + 1, m, m, m]$.

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

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

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





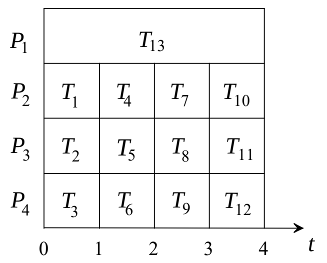
(a) *an optimal schedule,* **(b)** *LPT schedule.*

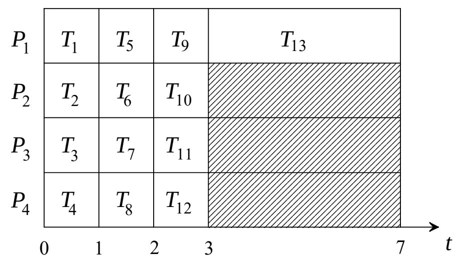
Identical Processors. *LPT Algorithm for P | | C*max

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

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

***Problem P* | *pmtn* | $C_{\max}$**

- relax some constraints imposed on problem *P* | | $C_{\max}$ and allow preemptions of tasks.
- It appears that problem *P* | *pmtn* | $C_{\max}$ 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 | pmtn | C_max
begin
$C^*_{max} := max\{\Sigma_{j=1}^{n} p_j/m, max\{p_j | j = 1,...,n\}\}$; -- 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* | *pmtn* | $C_{\max}$

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

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 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

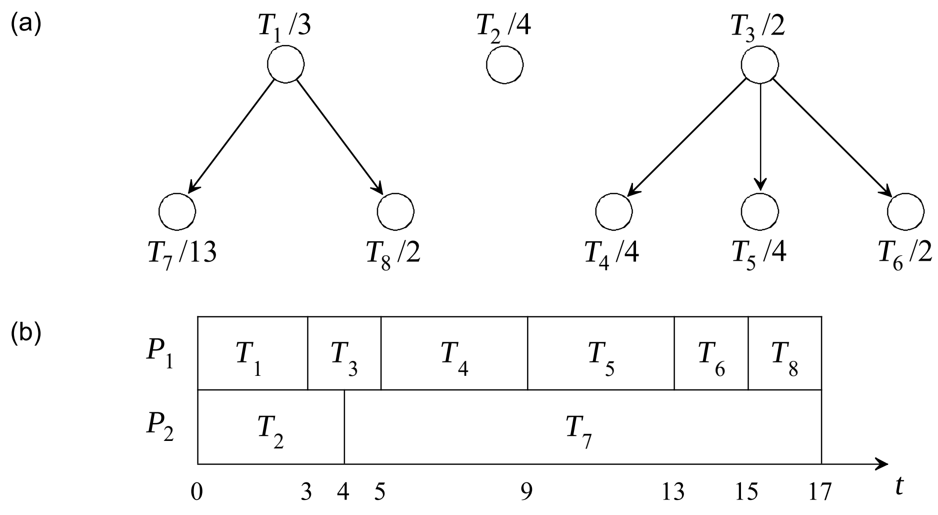
- the schedule length for problem $P \mid prec \mid C_{max}$
-

may increase

if:

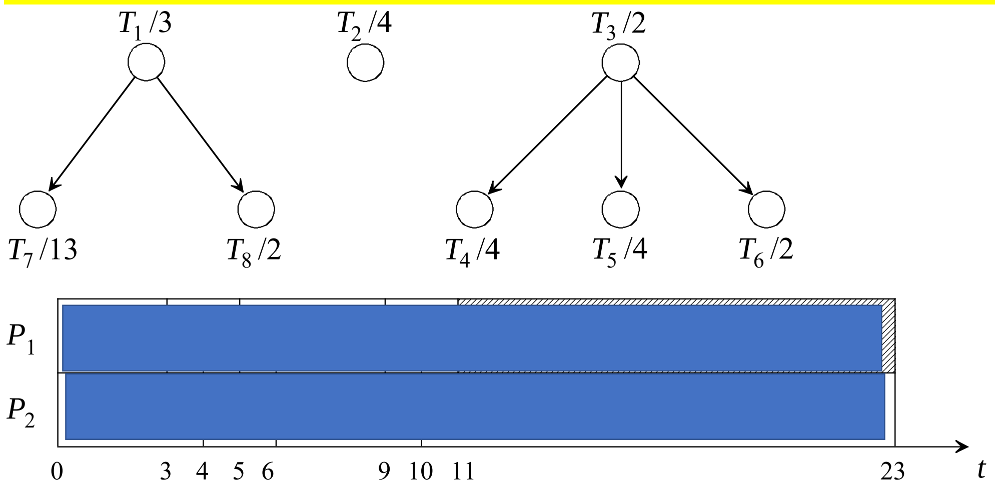
- the number of processors *increases*,
- task processing times *decrease*,
- precedence constraints are *weakened*, or
- the priority list changes

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



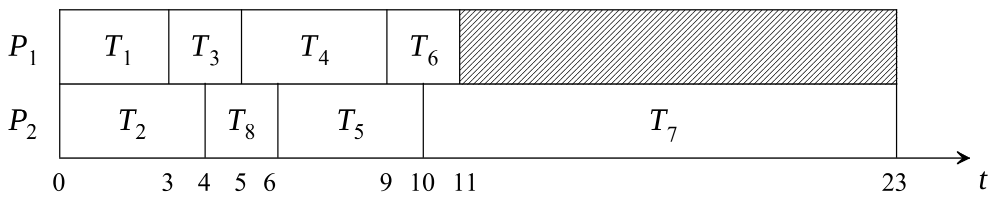
(a) *A task set, m = 2, L =* (T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8),
(b) *an optimal schedule*

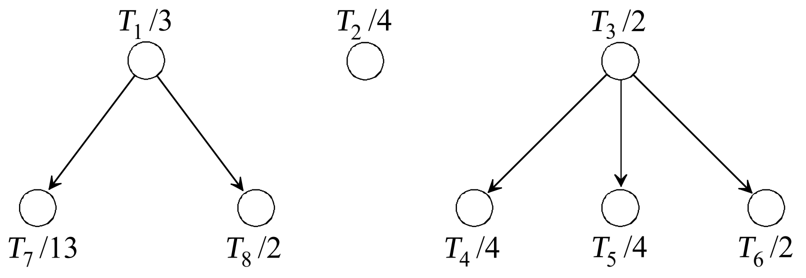
Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies



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

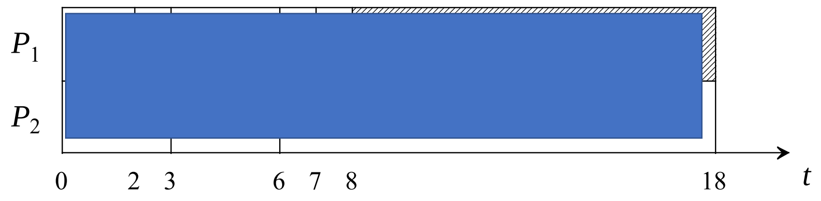
Identical Processors, $P \mid prec \mid C_{max}$, Graham anomalies

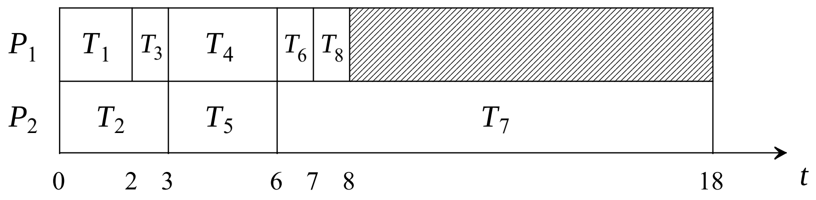




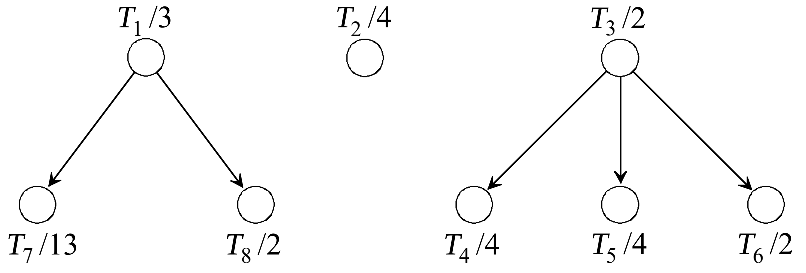
(T_1, T_2, T_3, T_4, T_5, T_6, T_7,T_8).

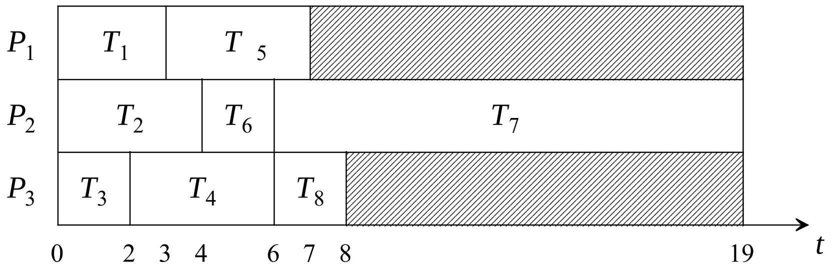
Processing times decreased; $p'_j = p_j - 1$, $j = 1, 2, ..., n$.





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

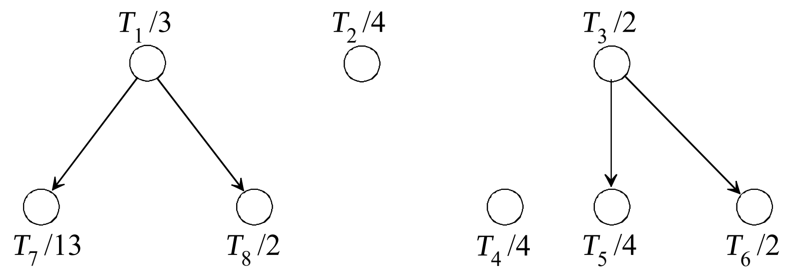




Number of processors increased, $m = 3$

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

(a)



(b)

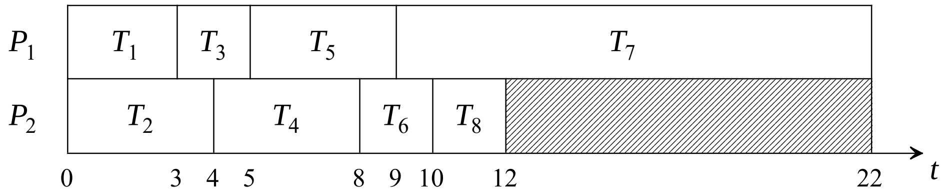


Figure 4-6 (a) *Precedence constraints weakened*, (b) *resulting list schedule.*

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.

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

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

Then the following theorem is valid.

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

4.1.3.1 Theorem . *Under the above assumptions*,

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

Proof. Let us consider schedule S' obtained by processing task set 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

Let T_{j1} denote a task completed in S' at time $C'_{\max}$, i.e. $C_{j1} = C'_{\max}$.
Two cases may occur:

1. The starting time s_{j1} of T_{j1} is an interior point of B . Then by the definition of B there is some processor P_i which for some $\varepsilon > 0$ is idle during interval $[s_{j1} - \varepsilon, s_{j1})$. Such a situation may only occur if we have $T_{j2} \prec' T_{j1}$ and $C_{j2} = s_{j1}$ for some task T_{j2} .

2. The starting time of T_{j1} is not an interior point of B. Let us also suppose that s_{j1} ¹ $\neq 0$. Define $x_1 = \sup\{x \mid x < s_{j1}$, and $x \in$ B $\}$ or $x_1 = 0$ if set B is empty.
By the construction of A and B , we see that $x_1 \in$ A , and processor P_i is idle in time interval $[x_1 - \varepsilon, x_1)$ for some $\varepsilon > 0$. But again, such a situation may only occur if some task $T_{j2} \prec' T_{j1}$ is processed during this time interval.

It follows that either there exists a task $T_{j2} \prec' T_{j1}$ such that $y \in [C_{j2} , s_{j1})$ implies $y \in$ A or we have: $x < s_{j1}$ implies either $x \in$ A or $x < 0$.

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

The above procedure can be inductively repeated, forming a chain T_{j3} , T_{j4} ,…, until we reach task T_{jr} for which $x < s_{jr}$ implies either $x \in$ A or $x < 0$. Hence there must exist a chain of tasks

$$T_{jr} \prec' T_{jr-1} \prec' \cdots \prec' T_{j2} \prec' T_{j1}$$

such that at each moment $t \in$ B , some task T_{jk} is being processed in S'. This implies that

$$\sum_{\phi' \in S'} p'_{\phi'} \leq (m'-1) \sum_{k=1}^{r} p'_{jk}$$

where the sum on the left-hand side is made over all idle-time tasks ϕ' in S'. But by (5.1.8) and the hypothesis $\prec' \subseteq \prec$ we have

$$T_{jr} \prec T_{jr-1} \prec \cdots \prec T_{j2} \prec T_{j1} .$$

Hence,

$$C_{\max} \geq \sum_{k=1}^{r} p_{jk} \geq \sum_{k=1}^{r} p'_{jk} .$$

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

we have

$$C'_{max} = \frac{1}{m'}\left(\sum_{k=1}^{n} p_k + \sum_{\phi' \in S'} p'_{\phi'}\right) \le \frac{1}{m'}\left(m\, C_{max} + (m'-1)\, C_{max}\right).$$

It follows that

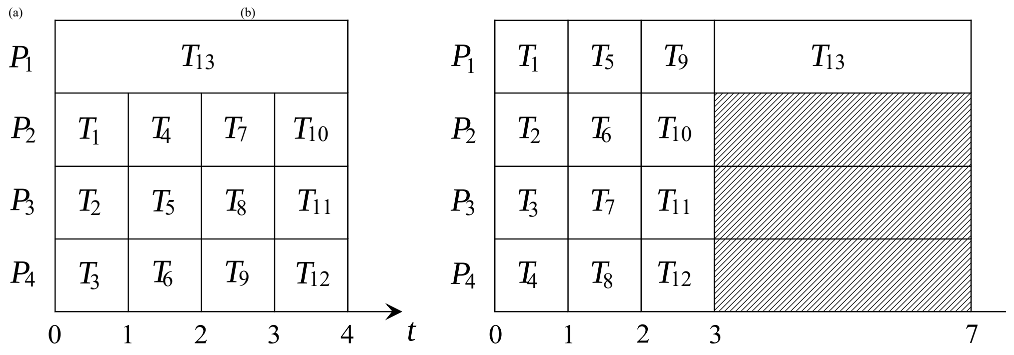
$$\frac{C'_{max}}{C_{max}} \le 1 + \frac{m-1}{m'}$$

and the theorem is proved. □

From the above theorem, the *absolute performance ratio* for an arbitrary list scheduling algorithm solving problem $P \mid\mid C_{max}$ can be derived.

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* $R_{LS} \le 2 - \frac{1}{m}$ if $m' = m$.



Schedules for Corollary

(a) *an optimal schedule,*

(b) *an approximate schedule.*

Unit Execution Time Tasks

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$

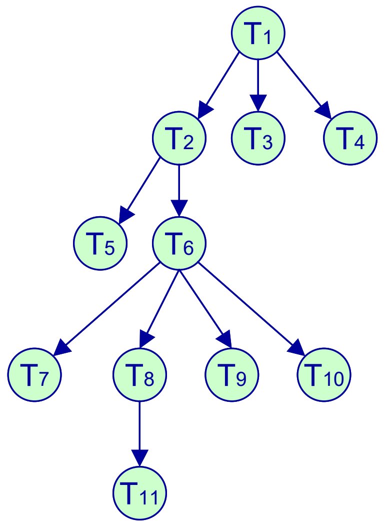
Problem $P \mid prec, p_j = 1 \mid C_{max}$

This problem is known to be NP-hard

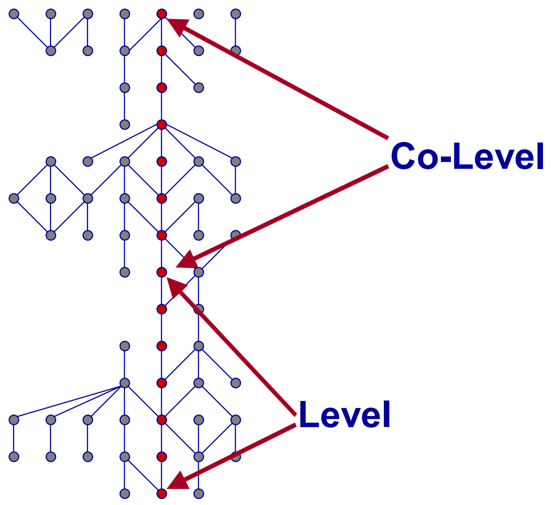
Arbitrary list scheduling algorithms: $R_{LS} \le 2 - \frac{1}{m}$ still holds in this case

However, under special assumptions polynomial time algorithms exist

Out-tree



Level



Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$

Hu's algorithm (Hu (1969) for the problem P **| *in-tree***, $p_j = 1 \mid C_{max}$

o level algorithm" or "critical path algorithm"

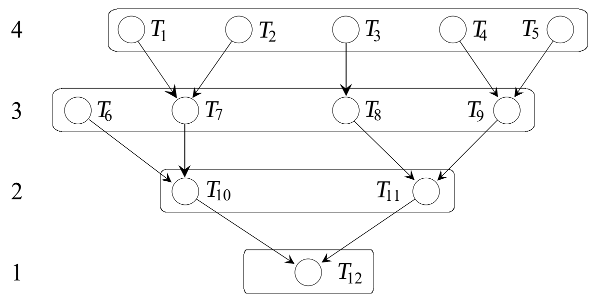
Task level in an in-tree: is defined as the number of tasks in the path to the root of the graph

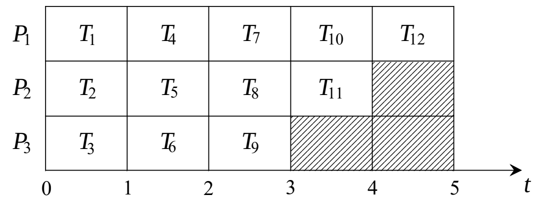
Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$

```
Algorithm Hu for P | in-tree, p_j = 1 | C_max .
begin
Calculate levels of the tasks;
t := 0;
repeat
  Construct list L_t of all the tasks without predecessors;
      -- all these tasks either have no predecessors
      -- or their predecessors have been assigned in interval [0,t-1]
    Order L_t in non-increasing order of task levels;
    Assign the first m tasks (if any) of L_t to processors;
    Remove the assigned tasks from the graph and from the list;
    t := t + 1;
until all tasks have been scheduled;
end;
```

The algorithm can be implemented to run in $O(n)$ time

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$





An example of the application of Algorithm for three processors.

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$

Scheduling forests: A forest consisting of in-trees can be scheduled by adding a dummy task that is an immediate successor of only the roots of in-trees, and then by applying Algorithm.

Scheduling out-forests: A schedule for an out-tree can be constructed by changing the orientation of arcs, applying Algorithm to the obtained in-tree and then reading the schedule backwards, i.e. from right to left

Remark: The problem of scheduling *opposing forests* (that is, combinations of in-trees and out-trees) on an arbitrary number of processors is NP-hard (Garey, et al 1983)

Another restriction is to limit the number of processors to 2: this problem is easily solvable even for arbitrary precedence graphs (Coffman and Graham 1972, and others):

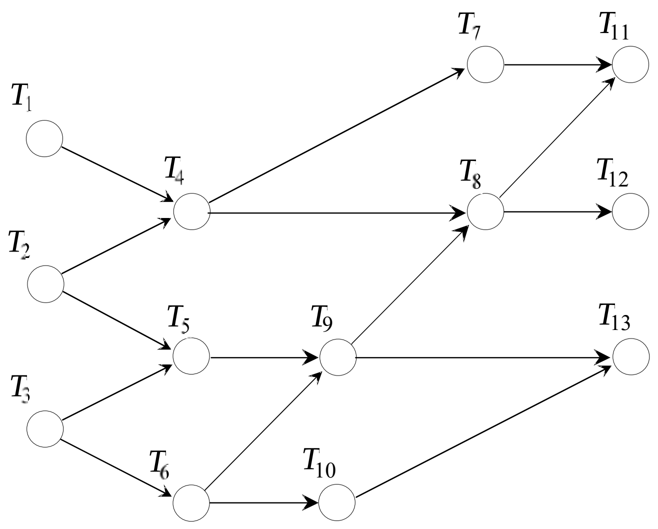
Problem $P2 \mid prec, p_j = 1 \mid C_{max}$ can be solved in polynomial time (quadratic in the number of tasks) [Coffman and Graham 1972]

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$

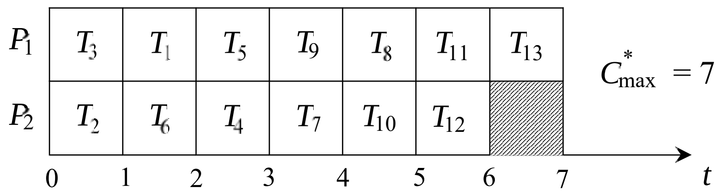
Agorithm given by Coffman and Graham

- to find the shortest schedule for problem $P2 \mid prec, p_j = 1 \mid C_{max}$.
- The algorithm uses *labels* assigned to tasks, which take into account the levels of the tasks and the numbers of their immediate successors.
- can be implemented to run in time which is almost linear in n and in the number of arcs in the precedence graph; thus its time complexity is practically $O(n^2)$.

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$



Identical Processors, P | prec, $p_j = 1$ | C_{max}



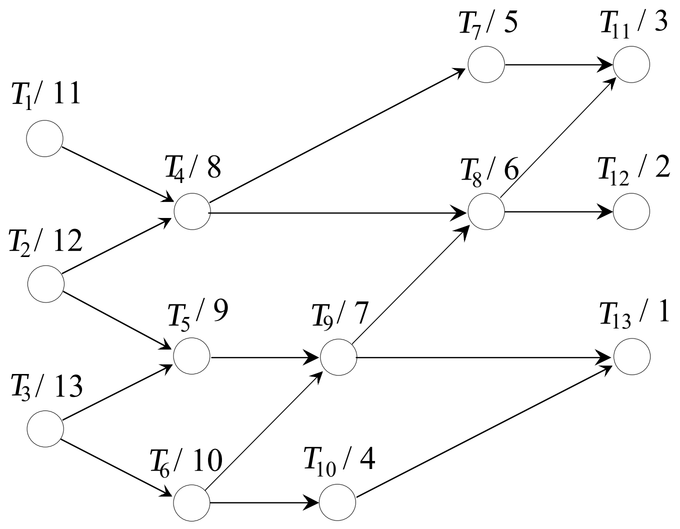
An example of the application of Algorithm (tasks are denoted by T_j /label).

Identical Processors, P | prec, $p_j = 1$ | C_{max}

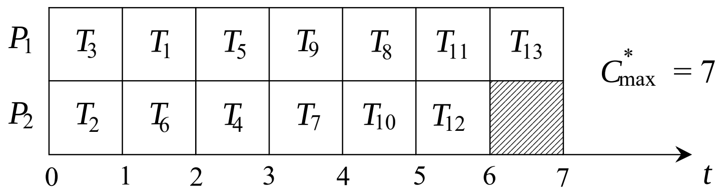
Algorithm of *Coffman and for P2 | prec, p_j = 1 |* C_{max}.

```
begin
Assign label 1 to any task $T_0$ for which isucc($T_0$) = ∅;
      -- recall that isucc($T$) denotes the set of all immediate successors of $T$
$j$ := 1;
repeat
   Construct set S consisting of all unlabeled tasks whose successors are labeled;
   for all $T \in$ S do
      begin
      Construct list $L(T)$ consisting of labels of tasks belonging to isucc($T$);
      Order $L(T)$ in decreasing order of the labels;
      end;
   Order these lists in increasing lexicographic order $L(T_{[1]}) <\cdot ... <\cdot L(T_{[|S|]})$;
                           -- see Section 2.1 for definition of $<\cdot$
   Assign label $j$ + 1 to task $T_{[1]}$;
   $j := j + 1$;
until $j = n$;   -- all tasks have been assigned labels
```

Identical Processors, P | prec, $p_j = 1$ | C_{max}



Identical Processors, P | prec, $p_j = 1$ | C_{max}



An example of the application of Algorithm (tasks are denoted by T_j /label).

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$. Known Results

- 1961: *P*| *in-tree, out-tree, pj*=1 |*Cmax* - Hu's Level algorithm is optimal and of linear time complexity
- 1966: *P*| *prec* |*Cmax* - Graham showed that for List Scheduling algorithms the performance bound *r*= 2-1/*m*
- 1969: *P*| |*Cmax* Graham used the LPT algorithm with *r*= 4/3 – 1/3*m*
- 1972: *P*2| *prec*, *pj*=1 |*Cmax* Coffman proved that problem can be solved in quadratic time

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$. Known Results

- 1975: *P*2| *prec*, *pj*=1 |*Cmax* Chen & Liu found *R_level*=4/3
- 1976: *P*| *prec*, *pj*=1 |*Cmax* R_{Level} $2 - \frac{1}{m-1}$ *for* $m \geq 3$
- 1976: *P*| *prec*, *pj*=1 |*Cmax* Coffman & Sethi proved the bound 1+ 1/*n* - 1/*nm* for *P*| |*Cmax* taking into account the number of tasks
- 1977: *P*2| *prec*, *pj*=1 |*Cmax* Garey&Johnson - *Latest Possible Start Time algorithm*, optimal
- 1981: *P*| *opposing forest*, *pj*=1 |*Cmax* Kunde, *Critical Path algorithm* *r*=2-2/(*m*+1)
- 1994: *P*| *prec*, *pj*=1|*Cmax* Braschi and Trystram found *r* = 2 - 2/*m* - (*m*-3) /(*mCmax*) for *m*>=31996: *Pm*| *prec*, *pj*=1|*Cmax* is with unknown time complexity

Identical Processors, $P \mid prec, p_j = 1 \mid C_{max}$. Known Results

l_c is the length of a longest chain of tasks

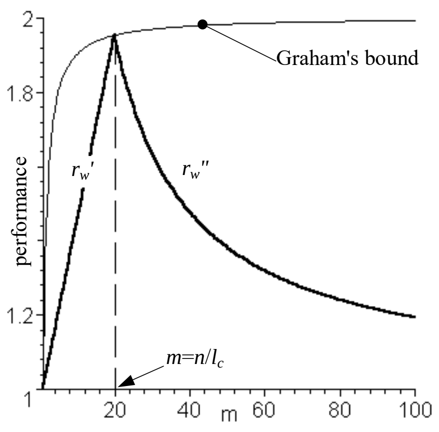
Theorem [Tchernykh et al 2000]. *Given a set* T *of* $\boldsymbol{n}$ *unit execution time tasks, the performance of the general list strategy can be estimated by*

$$R = \min\{R', R''\},$$

with $R' \leq 1 + \frac{l_c}{n}(m-1)$

and $R'' \leq 1 + \frac{1}{m}\left(\frac{n}{l_c} - 1\right).$ (1)

Furthermore,
R' *is tight in the case of* $l_c \leq n/m$,
and R'' *is tight in the case of* $l_c > n/m$



Preemptions

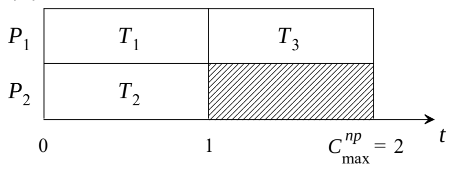
Identical Processors. $P \mid pmtn, prec \mid C_{max}$

What can be gained by allowing preemptions?

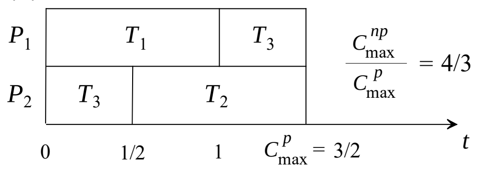
Coffman and Garey (1991) compared problems $P2 \mid prec \mid C_{max}$ and $P2 \mid pmtn, prec \mid C_{max}$: $(3/4)C_{max}^{non\text{-}preemptive} \le C_{max}^{preemptive} \le C_{max}^{non\text{-}preemptive}$

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

(a) non-preemptive schedule:



(b) preemptive schedule:



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

In the general case of dependent tasks of arbitrary length, one can construct optimal preemptive schedules.

- the *level* of task T_j in a precedence graph is now the sum of processing times (including p_j) of tasks along the longest path between T_j and a terminal task (a task with no successors).

The algorithm uses a notion of a *processor shared schedule*, in which a task receives some fraction β (≤ 1) of the processing capacity of a processor.

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

Algorithm *by Muntz and Coffman for P2 | pmtn, prec |* C_{max} *and P | pmtn, forest |* C_{max} .

```
begin
for all $T \in \mathcal{T}$ do Compute the level of task $T$;
$t$ := 0; $h$ := $m$;
repeat
  Construct set Z  of tasks without predecessors at time $t$;
  while $h$ > 0 and |Z | > 0 do
    begin
    Construct subset S  of Z  consisting of tasks at the highest level;
    if | S | > $h$
    then
      begin
      Assign $\beta$ := $h$/|S | of a processing capacity to each of the tasks from S ;
      $h$ := 0;   -- a processor shared partial schedule is constructed
      end
    else
      begin
      Assign one processor to each of the tasks from S ;
      $h$ := $h$-|S |;  -- a "normal" partial schedule is constructed
```

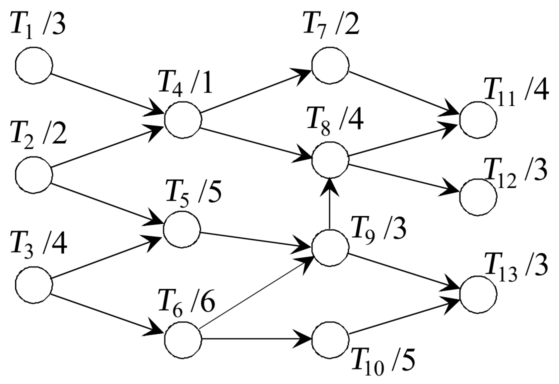
```
      end;
    Z  := Z - S ;
    end; -- the most "urgent" tasks have been assigned at time $t$
  Calculate time $\tau$ at which either one of the assigned tasks is finished or a point is
    reached at which continuing with the present partial assignment means that a
    task at a lower level will be executed at a faster rate $\beta$ than a task at a higher
    level;
  Decrease levels of the assigned tasks by $(\tau - t)\beta$;
  $t$ := $\tau$; $h$ := $m$;
      -- a portion of each assigned task equal to $(\tau - t)\beta$ has been processed
until all tasks are finished;

call Algorithm (McNaughton's rule) to re-schedule portions of the processor shared
  schedule to get a normal one;
end;
```

The above algorithm can be implemented to run in $O(n^2)$ time.

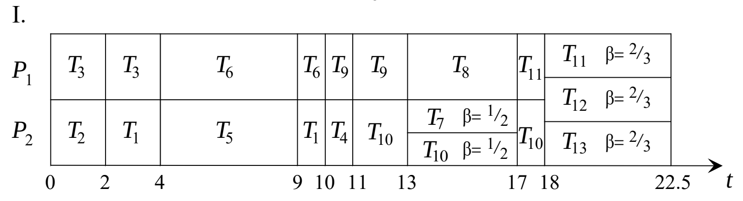
Identical Processors. *P | pmtn, prec | C*max

(a) *a task set (nodes are denoted by T_j /p_j),*

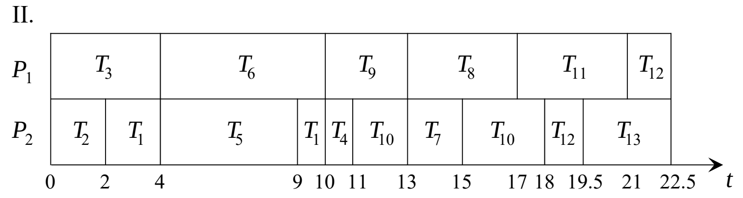


Identical Processors. *P | pmtn, prec | C*max

(b) *I: a processor-shared schedule, II: an optimal schedule*



I.
P_1: T_3 T_3 T_6 T_6 T_9 T_9 T_8 T_{11} T_{11} β= 2/3 T_{12} β= 2/3
P_2: T_2 T_1 T_5 T_1 T_4 T_{10} T_7 β= 1/2 T_{10} β= 1/2 T_{10} T_{13} β= 2/3
0 2 4 9 10 11 13 17 18 22.5 t



II.
P_1: T_3 T_6 T_9 T_8 T_{11} T_{12}
P_2: T_2 T_1 T_5 T_1 T_4 T_{10} T_7 T_{10} T_{12} T_{13}
0 2 4 9 10 11 13 15 17 18 19.5 21 22.5 t

Topic 2
Scheduling on Parallel Processors

Uniform Processors *Problem Q | p_j = 1 | C*max

Processors or *machines* for processing the tasks can be

- *parallel* - performing the same functions
- *dedicated* - specialized for the execution of certain tasks.

Parallel*:* Three types of parallel processors are distinguished depending on their speeds

- *identical* processors have equal task processing speeds
- *uniform* processors differ in their speeds, but the *speed* b_i of each processor is constant and does not depend on the task
- *unrelated* speeds of the processors depend on the particular task processed

Uniform Processors. *Problem $Q \mid p_j = 1 \mid C_{max}$*

Problem $Q \mid p_j = 1 \mid C_{max}$

- independent tasks
- non-preemptive scheduling
- UET

- problem with arbitrary processing times is already NP-hard for identical processors
- all we can hope to find is a polynomial time optimization algorithm for tasks with unit standard processing times only.

- a transportation network formulation has been presented by Graham et al. for problem $Q \mid p_j = 1 \mid C_{max}$.

Uniform Processors. *Problem $Q \mid p_j = 1 \mid C_{max}$*

Let there be n sources j, $j = 1, 2,\dots,n$,

and mn sinks (i, k), $i = 1, 2,\dots,m$ and $k = 1, 2,\dots,n$.

Sources correspond to tasks and sinks to processors and positions of tasks on them (number of tasks processed on the processor) .

Let $c_{ijk} = k/b_i$ be the cost of arc $(j, (i, k))$; this value corresponds to the completion time of task T_j processed on P_i in the k^{th} position. The arc flow x_{ijk} has the following interpretation:

$$x_{ijk} = \begin{cases} 1 & \text{if } T_j \text{ is processed in the } k^{th} \text{ position on } P_i \\ 0 & \text{otherwise.} \end{cases}$$

Uniform Processors. *Problem $Q \mid p_j = 1 \mid C_{max}$*

The min-max transportation problem can be now formulated as follows:

$$\textit{Minimize} \qquad \max_{i\,j\,k} \{c_{ijk}\, x_{ijk}\}$$

$$\textit{subject to} \quad \sum_{i=1}^{m} \sum_{k=1}^{n} x_{ijk} = 1 \qquad \text{for all } j$$

$$\sum_{j=1}^{n} x_{ijk} \le 1 \qquad \text{for all } i, k \,,$$

$$x_{ijk} \ge 0 \quad \text{for all } i, j, k \,.$$

Uniform Processors. *Problem $Q \mid p_j = 1 \mid C_{max}$*

minimum schedule length is given as $C^*_{max} = \sup\, \{t \mid \sum_{i=1}^{m} \lfloor t b_i \rfloor < n\}$.

lower bound on the schedule length for the above problem is

$$C' = n \,/ \sum_{i=1}^{m} b_i \le C^*_{max}$$

Bound C' can be achieved e.g. by a preemptive schedule.

If we assign $k_i = \lfloor C' b_i \rfloor$ tasks to processor P_i ,

- these tasks may be processed in time interval [0, C'].
- However, $l = n - \sum_{i=1}^{m} k_i$ tasks remain unassigned.
- $l \le m - 1$, since $C' b_i - \lfloor C' b_i \rfloor < 1$ for each i.
- The remaining l tasks are assigned to those P_i for which $\min_i \{(k_i + 1) / b_i\}$ is reached
- k_i is increased by one after the assignment of a task to a particular processor P_i .

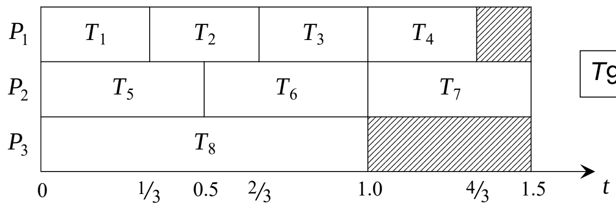
This procedure is repeated until all tasks are assigned. We see that this approach results in an $O(m^2)$-algorithm for solving problem $Q \mid p_j = 1 \mid C_{max}$.

Uniform Processors. *Problem Q | pj = 1 | Cmax*

Example $n = 9$ tasks, $m = 3$ uniform processors, processing speeds $\boldsymbol{b} = [3, 2, 1]$.

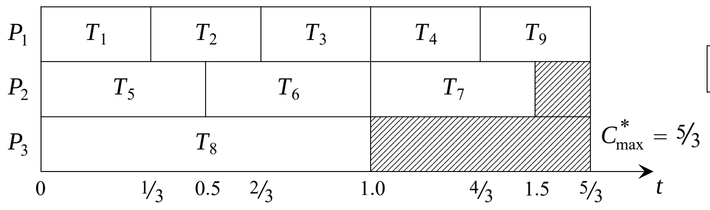
$C' = 9/6 = 1.5$.

The numbers of tasks assigned to processors at the first stage are, respectively, 4, 3, and 1.



T_9 has not yet been assigned.

Uniform Processors. *Problem Q | pj = 1 | Cmax*



optimal schedule

Uniform Processors. *Problem Q | pj = 1 | Cmax*

One heuristic algorithm is a list scheduling algorithm.

Tasks are ordered on the list in non-increasing order of their processing times and processors are ordered in non-increasing order of their processing speeds.

Now, whenever a machine becomes free it gets the first non-assigned task of the list; if there are two or more free processors, the fastest is chosen.

The worst-case behavior of the algorithm has been evaluated for the case of an $m + 1$ processor system, m of which have processing speed factor equal to 1 and the remaining processor has processing speed factor b. The bound is as follows.

$$R = \begin{cases} \dfrac{2(m+b)}{b+2} & \text{for } b \le 2 \\ \dfrac{m+b}{2} & \text{for } b > 2\,. \end{cases}$$

It is clear that the algorithm does better if, in the first case ($b \le 2$), m decreases faster than b, and if b and m decrease in case of $b > 2$.

Topic 3
Scheduling on Parallel 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 $P \mid r_j, \tilde{d}_j \mid -$

If **deadlines** are given:

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

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

EDF algorithm is optimal

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

The problems

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

are NP-complete

Algorithmic approaches:

– exhaustive search

– heuristic algorithms

– approximation algorithms

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

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
MUF	Maximum Urgency First scheduling.

-

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

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.

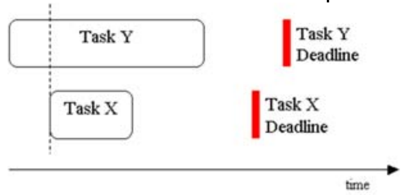
– EDF scheduler views task deadlines as more important than task priorities.

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

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

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 $P \mid r_j, \widetilde{d_j} \mid -$

LL scheduling, while excellent for highly time-critical tasks, might be overkilled for less time-sensitive tasks.

And so there is a third interesting variant of deadline scheduling, called "**Maximum Urgency First**" (or "**MUF**") scheduling.

It is really a mixture of some "LL" deadline scheduling, with some traditional priority-based preemptive scheduling.

In "**MUF**" scheduling, high-priority time-critical tasks are scheduled with "LL" deadline scheduling, while within the same scheduler other (lower-priority) tasks are scheduled by good old-fashioned priority-based preemption.

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

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
least laxity 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* | *pmtn*, $r_j, \tilde{d}_j$| –**

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

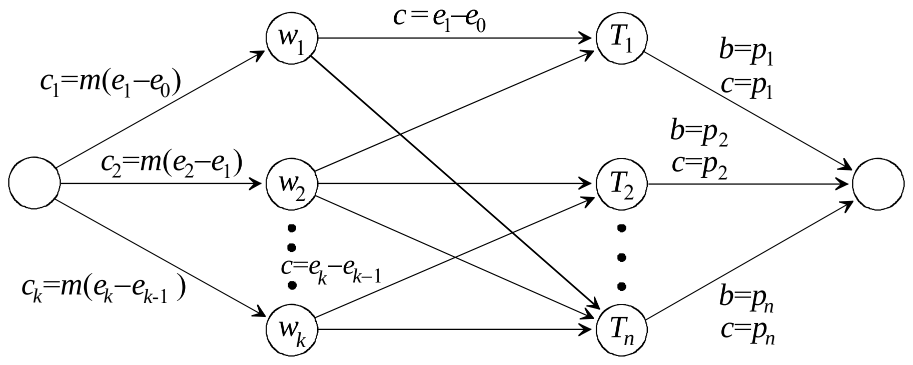
Given an instance of $P \mid pmtn, r_j, \tilde{d}_j \mid$,

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* | *pmtn*, $r_j, \tilde{d}_j$| –**



Identical Processors. Deadline Criteria ***P* | *pmtn*, $r_j, \tilde{d}_j$| –**

Flow conditions:

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

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

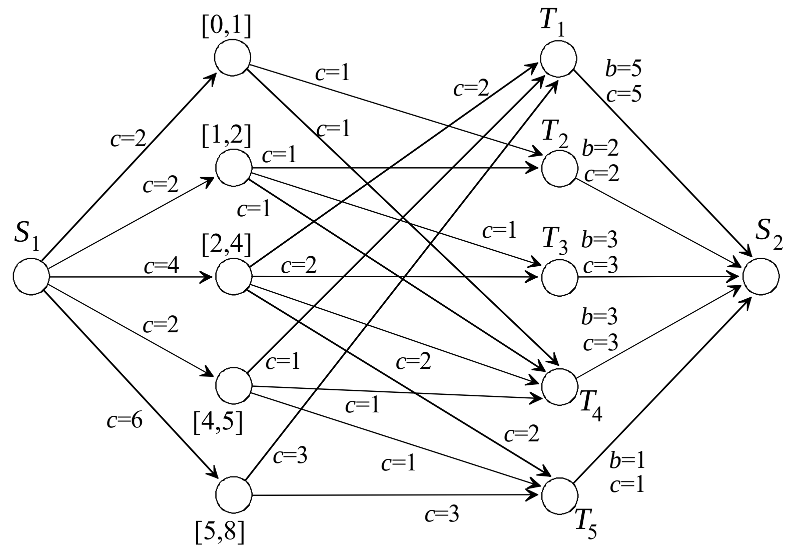
– If task T_j is allowed to be processed in interval $[e_{i-1}, e_j]$ then w_i is joined to T_j by an arc of capacity $e_i - e_{i-1}$

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

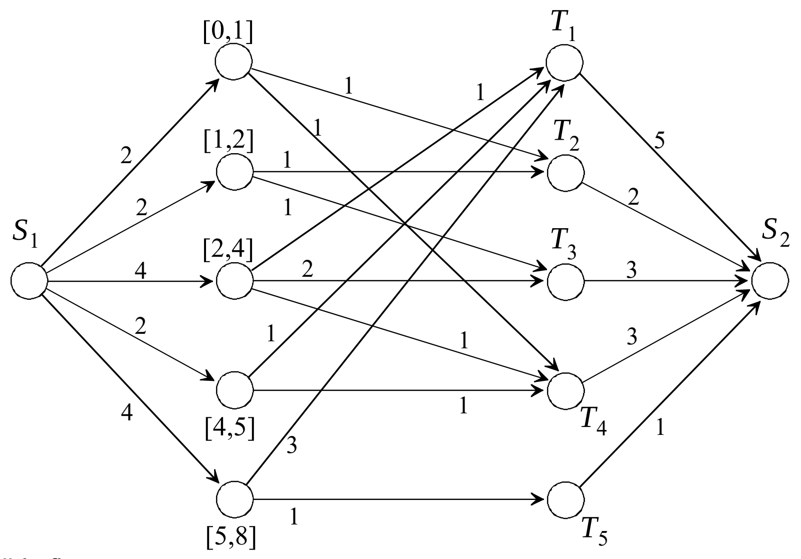
Finding a feasible flow pattern corresponds to constructing a feasible schedule; this test can be made in $O(n^3)$ time

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 $\boldsymbol{d} = [8, 2, 4, 5, 8]$.

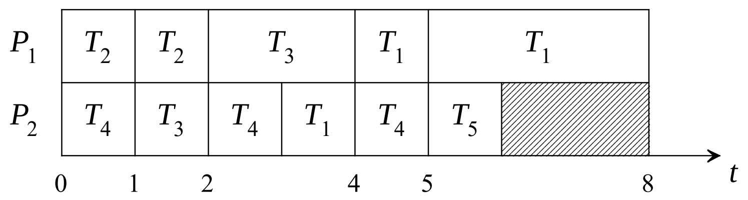


(a) corresponding network



(b) feasible flow pattern

(c) optimal schedule



P_1
P_2
T_2 T_2 T_3 T_1 T_1
T_4 T_3 T_4 T_1 T_4 T_5
0 1 2 4 5 8
t

Minimizing Maximum Lateness

$$L_{max} = max\{L_j \mid T_j \in \mathcal{T}\}$$

lateness $L_j = c_j - d_j$

Identical Processors. $P \mid\mid L_{max}$

- ***m* = 1 processor:** *earliest due date algorithm* (*EDD* rule) of Jackson [Jac55] : *tasks are scheduled in order of non-decreasing due dates*

The *EDD* rule also minimizes maximum lateness and maximum tardiness

- **$m \geq 1$ identical processors:** NP-hard C_{max}–problems are also NP-hard under the L_{max} criterion

  for example: $\boldsymbol{P} \mid\mid L_{max}$ is NP-hard

- unit processing times of tasks make the problem easy, and $P \mid p_j = 1, r_j \mid L_{max}$ can be solved by an obvious application of the *EDD* rule.

- Moreover, problem $P \mid p_j = p, r_j \mid L_{max}$ can be solved in polynomial time by an extension of the single processor algorithm.

Identical Processors. $P \mid r_j \mid L_{max}$

Problem **1 |** $r_j \mid L_{max}$ is strongly NP-hard (Lenstra et al., 1977)

solution methods based on branch and bound are known

Assumption of unit execution times

(1 | $r_j,\ p_j$ = 1 | L_{max} , r_j an integer): a modification of Jackson's *EDD* rule is optimal

Identical Processors. P | pmtn | L_{max}

The preemptive mode of processing makes the problem much easier.

Single processor problem 1 | *pmtn*, $r_j \mid L_{max}$:

A modification of Jackson's rule due to **Horn** (1974) solves the problem optimally in polynomial time

```
Algorithm for problem 1 | pmtn, r_j | L_max  (Horn, 1974)
begin
repeat
  $\rho_1 := min\{r_j | r_j \in T\}$;
  if all tasks are available at time $\rho_1$
  then $\rho_2 := \infty$
  else $\rho_2 := min\{r_j | r_j \neq \rho_1\}$;
  $\mathcal{E} := \{T_j \mid r_j = \rho_1\}$;
  Choose $T_k \in \mathcal{E}$ such that $d_k = min\{d_j | T_j \in \mathcal{E}\}$
  $l := min\{p_k, \rho_2 - \rho_1\}$;
  Assign $T_k$ to the interval $[\rho_1, \rho_1 + l)$;
  if $p_k \le l$
  then $\mathcal{T} := \mathcal{T} - \{T_k\}$
  else $p_k := p_k - l$;
  for all $T_j \in \mathcal{E}$ do $r_j := \rho_1 + l$;
  until $\mathcal{T} = \emptyset$;
end;
```

Identical Processors. P | *pmtn*, $r_j \mid L_{max}$

polynomial time algorithm by Labetoulle et al, 1984

The idea is to determine the smallest possible value of L_{max} such that there exists a feasible solution for the deadline problem $P \mid pmtn, r_j, \tilde{d}_j \mid -$ where deadlines are defined by $\tilde{d}_j := d_j + L_{max}$

Feasibility testing of problem $P \mid pmtn, r_j,\ \tilde{d}_j \mid -$ is done by applying the network flow approach

i.e. for deciding whether or not for a given set of ready times and deadlines a schedule with no late task exists

If there is no feasible flow pattern: a corresponding schedule can still be constructed, but L_{max} will turn out to be > 0

In other words, if the instance is changed such that all the deadlines are increased by L_{max} , a feasible network flow would exist

Identical Processors. *P* | *pmtn*, r_j| L_{max}

To find a schedule with minimum L_{max} , a binary search can be performed:

the deadlines are increased by $L_{max}/2$ (instead of L_{max}) and this new instance is checked for feasibility by means of the network flow computation.

This procedure can be implemented to solve problem P | $pmtn$, r_j| L_{max} in time $O(n^3 min\{n^2, logn + \log\ max\{p_j\}\}$

The fundamental approach in that area is testing feasibility of problem P | pmtn, r_j, $\widetilde{d_j}$| – via the network flow approach [Hor74].

Using this approach repetitively, one can then solve the original problem P | pmtn | L_{max} by changing due dates (deadlines) according to a binary search procedure.

Identical Processors. L_{max}- problems with precedences

We just mention some results

- Problem *P* | *prec* | L_{max} : A general approach is to modify the due dates, depending on the number and due dates of their successors.
- Scheduling unit processing time tasks can result in polynomial time scheduling algorithms:

  Problem *P* | *in-tree*, p_j = 1 | L_{max} can be solved in *O*($nlogn$) time (Brucker 1976),

  but surprisingly *P* | *out-tree*, p_j = 1 | L_{max} is NP-hard (Brucker et al., 1977).

- Problem *P*2 | *prec*, p_j = 1 | L_{max} with arbitrary precedences: using a different way of computing modified due dates allows to solve the problem in $O(n^2)$ time (Garey et al, 1976).

Identical Processors. L_{max}- problems with precedence

- Problem *P* | *prec*, r_j | L_{max}
- with *m* = 1 processor:

  Example: Consider five tasks with release times ***r*** = [0, 2, 3, 0, 7], processing times ***p*** = [2, 1, 2, 2, 2], and tails ***d*** = [7, 10, 6, 9, 10],

  a) the precedence constraint $T_4 \prec T_2$;

  b) No precedence constraint

Identical Processors. L_{max}- problems with precedence

- Problem *P* | *prec*, r_j | L_{max} with *m* = 1 processor:

  Example: Consider five tasks with release times ***r*** = [0, 2, 3, 0, 7], processing times ***p*** = [2, 1, 2, 2, 2], and tails ***d*** = [7, 10, 6, 9, 10], and the precedence constraint $T_4 \prec T_2$; note that $r_4 + p_4 \leq r_2$ and $d_4 \geq\ d_2 - p_2$.

  If the constraint $T_4 \prec T_2$ is ignored, the unique optimal schedule is given by $(T_1,\ T_2,\ T_3,\ T_4,\ T_5)$ with value L^*_{max} -1. Explicit inclusion of this constraint leads to $L^*_{max} = 0$.

Identical Processors. L_{max}- problems with precedences

- Allowing preemptions:

  The following problems are solvable **in polynomial time**:

  $P \mid pmtn, in\text{-}tree \mid L_{max}$,

  $P2 \mid pmtn, prec \mid L_{max}$,

  $P2 \mid pmtn, prec, r_j \mid L_{max}$

Algorithms for these problems employ essentially the same techniques for dealing with precedence constraints as the corresponding algorithms for tasks with unit execution time

Summary

Four different types of problems are considered:

- a deadline problem
- three due date problems
  - minimizing maximum lateness,
  - weighted number of tardy tasks,
  - and maximum weighted tardiness

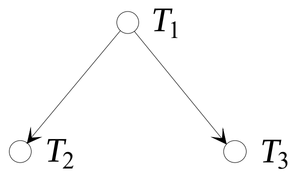
All these problems could be solved in polynomial time only under very special restrictions

Scheduling on Parallel Processors

Communication Delays and Multiprocessor Tasks

Scheduling Uniprocessor Tasks with Communication Delays

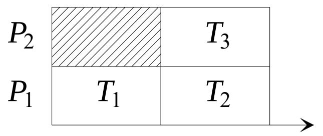
The following simple example serves as an introduction to the problems.
Let there be given three tasks with precedences as shown in Figure (a).



(a) *Precedence graph*

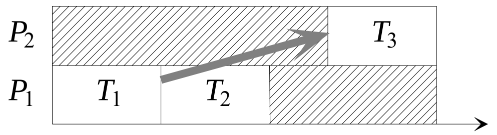
The computational results of task T_1 are needed by both successor tasks, T_2 and T_3
We assume unit processing times.
For task execution there are two identical processors, connected by a communication link.
To transmit the results of computation T_1 along the link takes 1.5 units of time.

Scheduling Uniprocessor Tasks with Communication Delays



(b) *Schedule without consideration of communication delays*

The schedule in Figure (b) shows a schedule where communication delays are not considered.

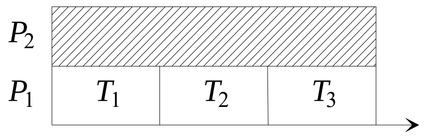


(c) *Schedule considering communication from T_1 to T_3*

The schedule (c) is obtained from (b) by introducing a communication delay between T_1 and T_3

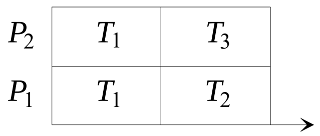
Scheduling Uniprocessor Tasks with Communication Delays

Schedule (d) demonstrates that there are situations where a second processor does not help to gain a shorter schedule.



(d) *Optimal schedule without task duplication*

The fourth schedule, (e), demonstrates another possibility: if task T_1 is processed on both processors, an even shorter schedule is obtained. The latter case is usually referred to as *task duplication*.



(e) *Optimal schedule with task duplication*

Scheduling Uniprocessor Tasks with Communication Delays

Communication delays are the same for all tasks

- So-called *uniform delay scheduling*.

Other approaches distinguish between *coarse grain* and *fine grain* parallelism:

- high computation-communication ratio can be expected in coarse grain parallelism.

As pointed out before, *task duplication* often leads to shorter schedules; this is in particular the case if the communication times are large compared to the processing times.

Bin Packing Problem

Outline

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.

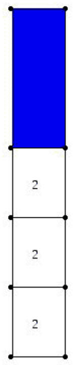
However, in the problem being considered here we are not allowed to have part of a weight in one container and part in another.

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.

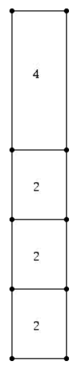
Outline

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).

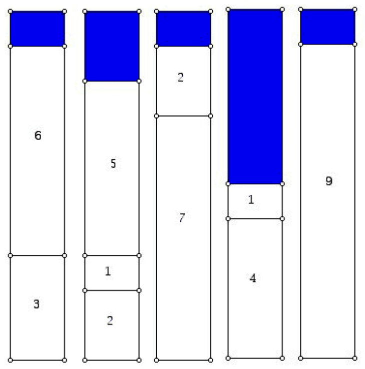
The idea behind this procedure is to open a bin and place the items into it in the order they appear in the list.

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)

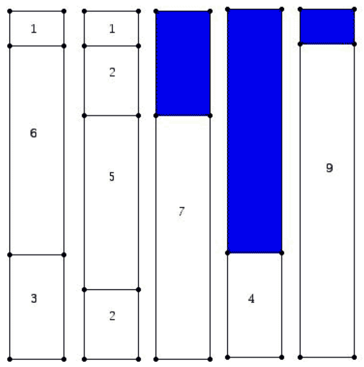
The simplest way to carry out this idea is known as *First Fit*.

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.

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

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:

- o suppose that we currently have only 3 bins open with capacity 10
- o *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?