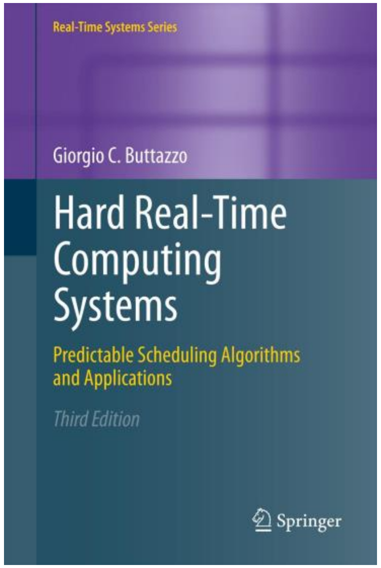
2IMN20 - Real-Time Systems

Scheduling 101

Geoffrey Nelissen

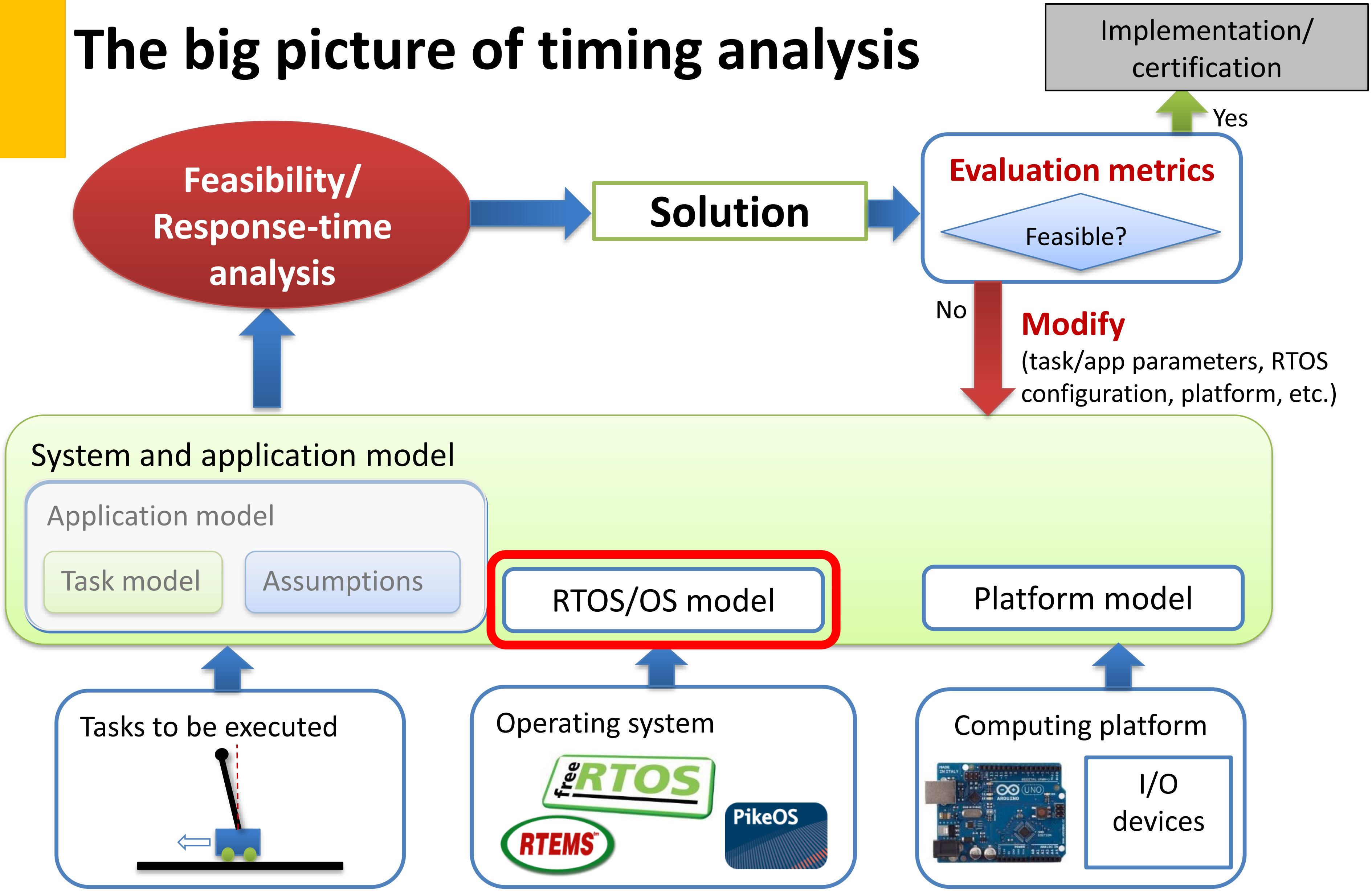
2023-2024

Buttazzo's book, chapters 2 and small part of 3



Disclaimer:
Most slides were provided by Dr. Mitra Nasri
Some slides have been taken from [Giorgio Buttazzo]

The big picture of timing analysis

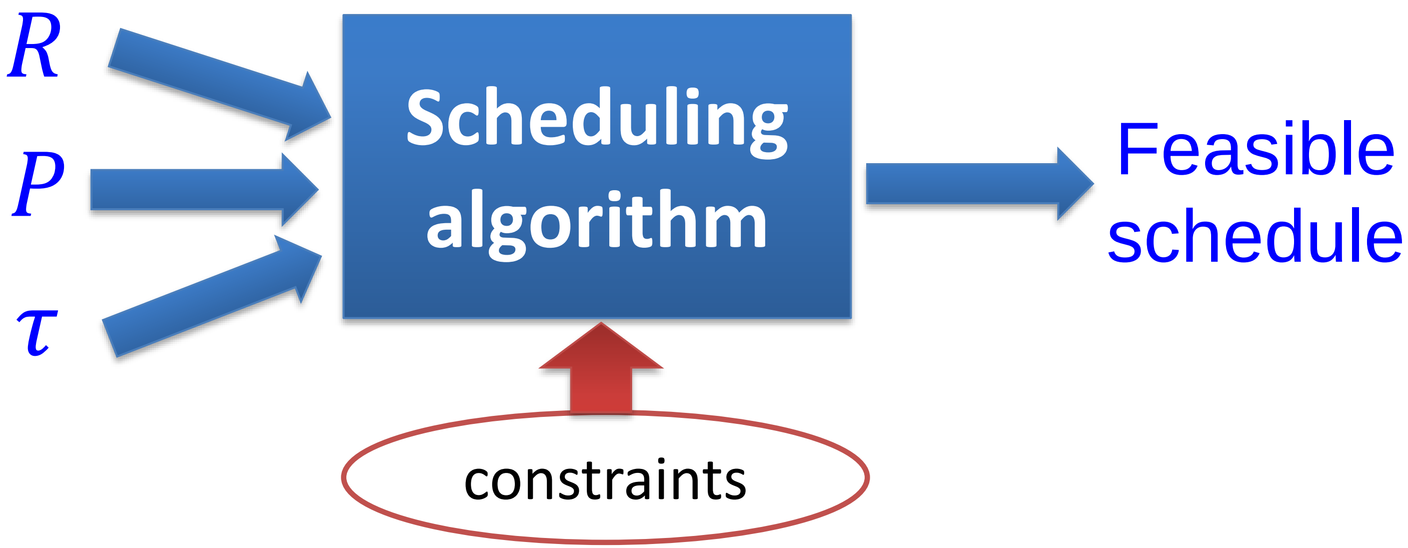


Agenda

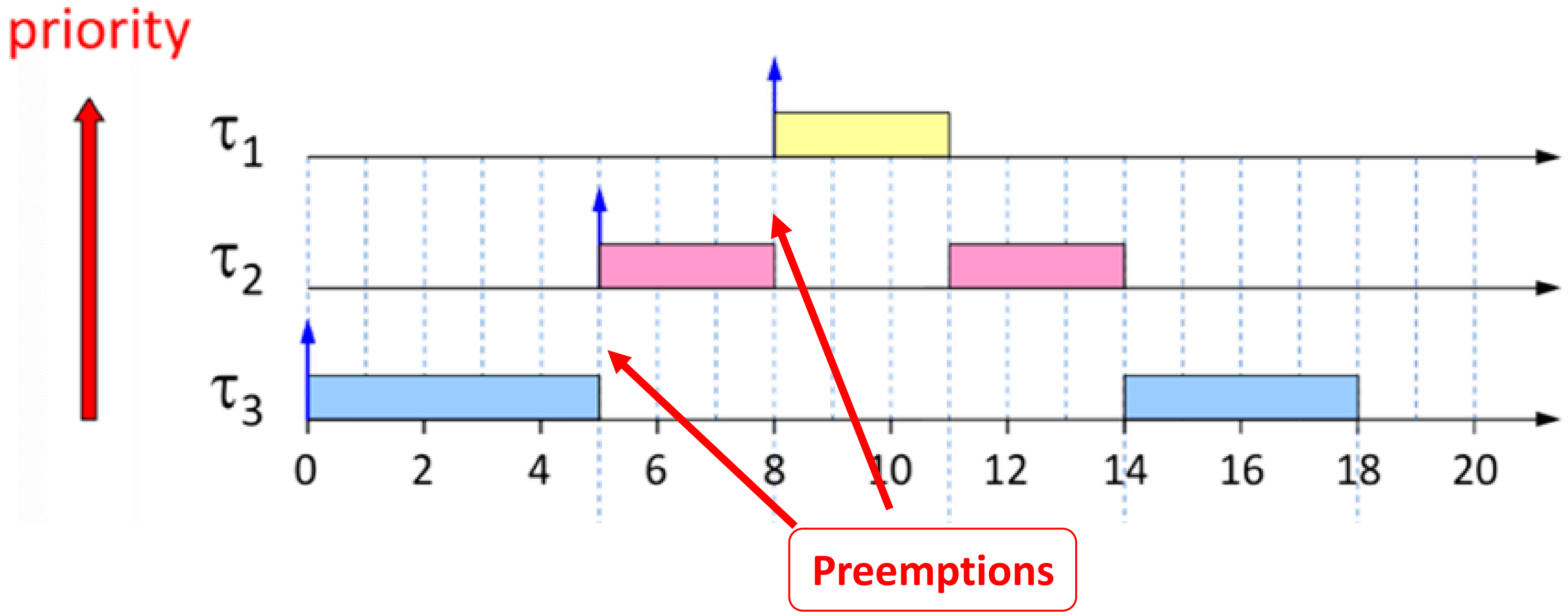
- Scheduling
- Table-driven scheduling
- Online scheduling policies

General scheduling problem

Given a set τ of n tasks, a set $\boldsymbol{P}$ of m processors, and a set $\boldsymbol{R}$ of r resources,
find an assignment of $\boldsymbol{P}$ and $\boldsymbol{R}$ to τ for any time instant t that produces a feasible schedule under a set of constraints.



Example schedule: FP Preemptive scheduling



Scheduling policy/algorithm (for tasks on processors)

A **scheduling policy** is an algorithm that determines what task executes when on what processor, i.e., produces a schedule.

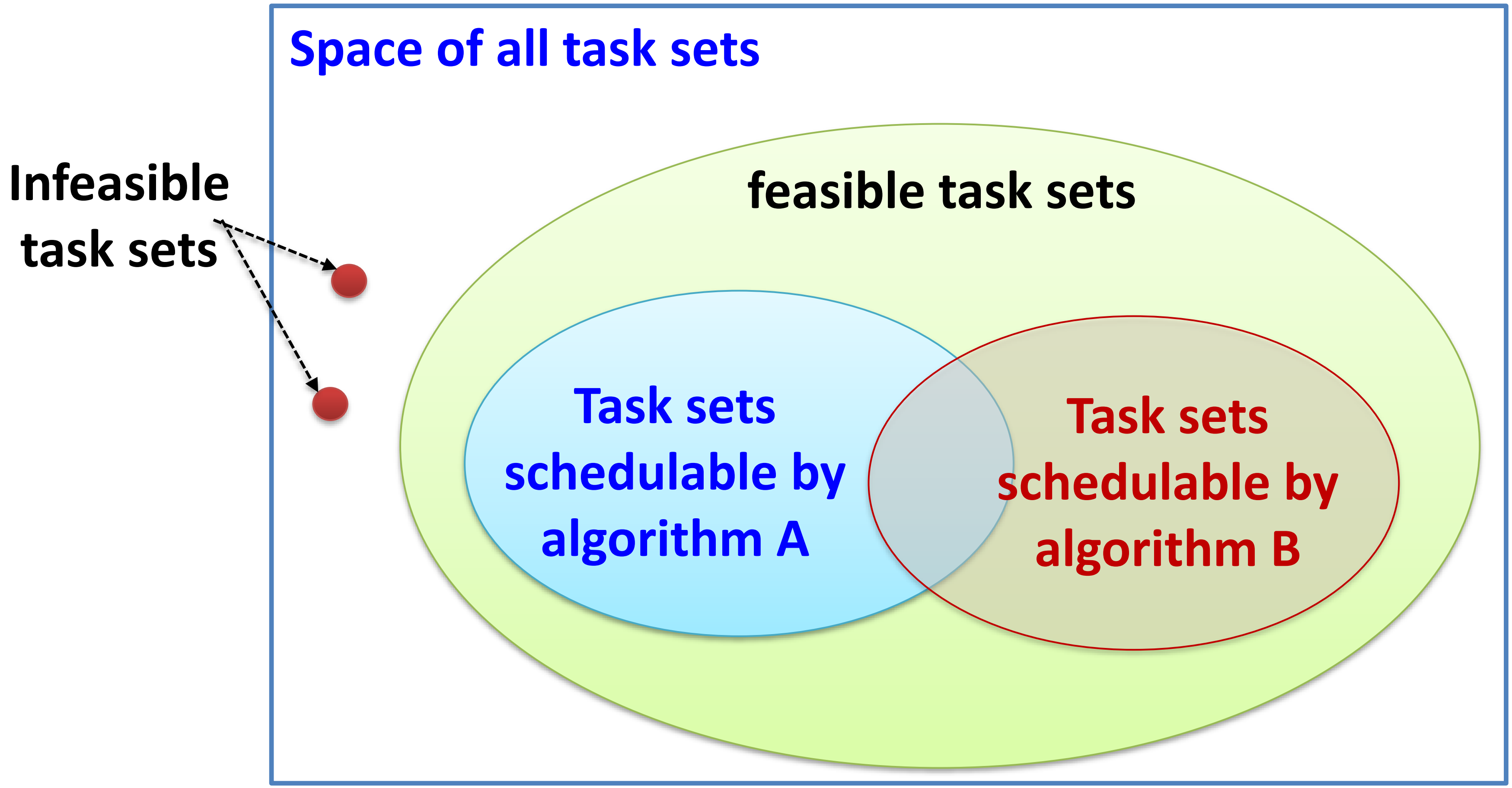
Definitions

A schedule S is said to be **feasible** if it satisfies all given requirements, for example, “no deadline is missed in S”.

A task set τ is said to be **feasible**, if *there always exists at least one feasible schedule for it*

A task set τ is said to be **schedulable** with a scheduling algorithm A, **if A always generates a feasible schedule** for τ.

Feasibility vs. schedulability



Properties of scheduling algorithms

- **Preemptive** vs. **non-preemptive**
- **Work-conserving** vs. **non-work-conserving**
- **Offline** vs. **online**
- **Optimal** vs. **non-optimal**

Properties of scheduling algorithms

- **Preemptive** vs. **non-preemptive**
- **Work-conserving** vs. **non-work-conserving**
- **Offline** vs. **online**
- **Optimal** vs. **non-optimal**

Work-conserving vs. non-work-conserving

- **Work-conserving**

  Such algorithm **does not** **leave the processor** **idle** as long as there is a ready task in the system (a task is in the ready queue).

- **Non-work conserving**

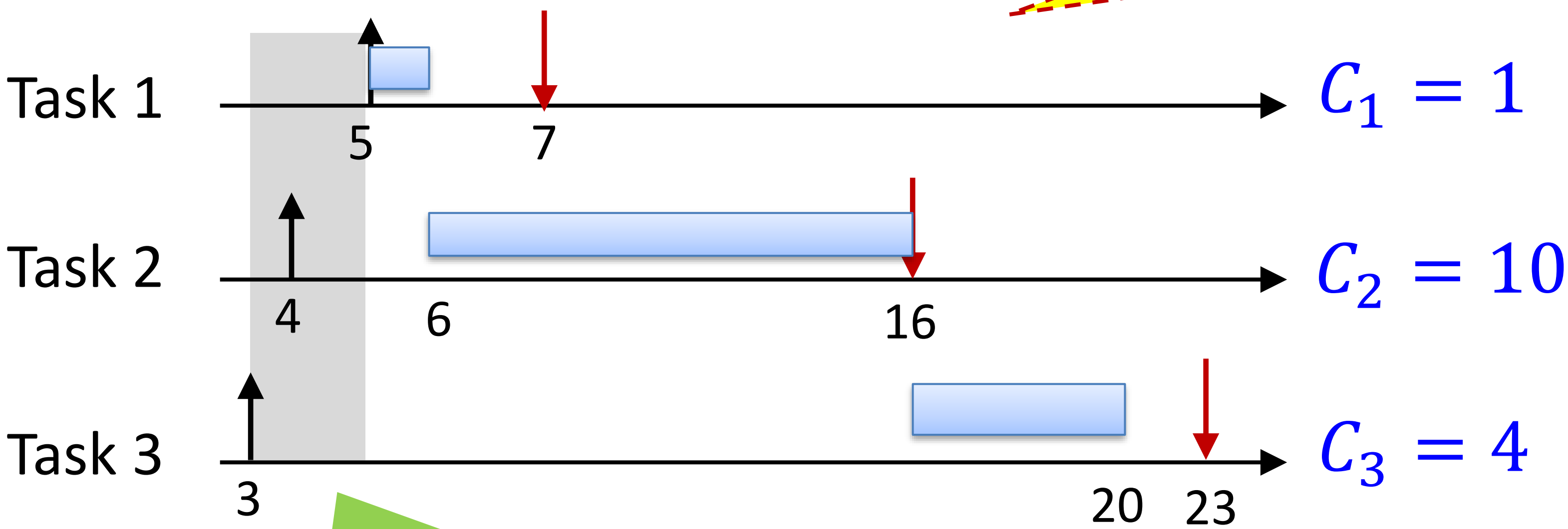
  Such algorithm **may** **leave the processor** **idle** even if there is a ready task in the ready queue.

Q: Why would we want to leave the processor idle when there is something to execute?

See next slide

Work-conserving vs. non-work-conserving

Q: Is this job set feasible under non-preemptive scheduling?



The processor must remain idle in $[3, 5)$ even though Tasks 2 and 3 are in the ready queue

→ We must use a non-work-conserving algorithm

Optimal v.s. non-optimal

- **Optimal**

  They generate a schedule that minimizes a cost function, defined based on an optimality criterion.

- **Non-optimal (heuristic)**

  They generate a schedule according to a heuristic function that tries to satisfy an optimality criterion, but there is no guarantee of success.

Example for some optimization goals

- **Feasibility**: Find a feasible schedule if there exists one.
- Minimize the **maximum lateness (i.e., tardiness)**
- Minimize the **number of deadline misses**
- Minimize the **average waiting time**
- Assign a value to each task, then maximize the **cumulative value** of the feasible tasks

Offline vs. online

- **Offline**

  all scheduling decisions are taken before tasks activations: the schedule is stored in a table (**table-driven scheduling**).

- **Online**

  scheduling decisions are taken at run time based on the set of active tasks and the system state.

A closer look at offline scheduling policies

Table-driven scheduling

- Store the schedule in a table that is prepared offline
- Dispatch jobs according to the table

Task	Start time
1	0
2	10
3	12
1	20
4	28

Cycle = 40

Table-driven scheduling

- Store the schedule in a table that is prepared offline
- Dispatch jobs according to the table

Task	Start time
1	0
2	10
3	12
1	20
4	28

Cycle = 40

Timer interrupt handler:

1. Dispatcher reads the table entry at the current index i and dispatches the task
2. Increments i
3. Sets the timer interrupt to the "start time" of the current table entry at index i
4. If the last entry is reached. Reset the time at the end of the cycle time, and set index i to 0

Resulting schedule

Advantages of table-driven scheduling

Any guess?

- **Extremely predictable**
  - Provides full knowledge about when the tasks will be executing
  - Hence, you can easily analyze system performances
- **Easy to certify**
  - Avionics industry uses partly table-driven scheduling
- **Extremely flexible for schedule optimizations**
  - It is easy to include optimization criteria while building the schedule
  - Can handles various system constraints such as precedence constraints, etc.
- **Low runtime overhead**
  - A true O(1) algorithm for scheduling on most hardware platforms
- **Very small "code" footprint**
  - Only requires a few instructions to implement

Disadvantages of table-driven scheduling

Any guess?

- **Requires concrete knowledge of** task release times
  - Cannot be applied on event-based systems or dynamic workloads
- **Building an optimal schedule might be computationally expensive**
  - recall: the general scheduling problem is NP-Hard

What else?

It eats up a large amount of memory to store the table!

Example:

For a system with 1000 jobs per hyperperiod,
and 32 bits to store a table entry, the table becomes as big as **4kB**

An **Arduino Mega** has only **8kB of RAM**

Animation: Spirited away (Hayao Miyazaki)

Memory is money!

- Many embedded systems have a limited processing power and memory because
  - memory is expensive
  - consumes energy

Arm Cortex MCU family

STM32 32-bit ARM Cortex MCUs | STM32F2 Series | STM32F3 Series | STM32F4 Series | STM32F7 Series

Total Parts: (752) for STM32 32-bit ARM Cortex MCUs | Matching Parts : (90)

Part Number	Package	Core	Operating Frequency (MHz) (Processor speed)	FLASH Size (kB) (Prog)	Internal RAM Size (kB)	I/Os (High Current)
STM32L011G4	UFQFPN 28 4x4 x0.55	ARM Cortex-M...	32	16	2	24
STM32L011K4	LQFP 32 7x7x1.4, ...	ARM Cortex-M...	32	16	2	28
STM32L021D4	TSSOP 14	ARM Cortex-M...	32	16	2	11
STM32L021F4	UFQFPN 20 3x3 x0.6	ARM Cortex-M...	32	16	2	16
STM32L021G4	UFQFPN 28 4x4 x0.55	ARM Cortex-M...	32	16	2	24
STM32L021K4	LQFP 32 7x7x1.4	ARM Cortex-M...	32	16	2	28
STM32L031F4	TSSOP 20	ARM Cortex-M...	32	16	8	15
STM32L071C8	LQFP 48 7x7x1.4	ARM Cortex-M...	32	64	20	37
STM32L071RZ	LQFP 64 10x10 x1.4, ...	ARM Cortex-M...	32	192	20	51
STM32L071VB	LQFP 100 14x14x1.4	ARM Cortex-M...	32	128	20	84

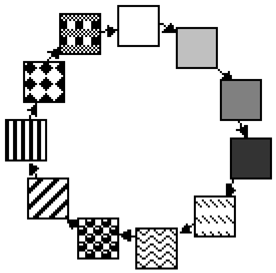
Online scheduling policies
Cyclic executive

See also Buttazzo Section 4.2.

Single-rate AFAP (As Fast As Possible)

```
while(1){
  Task_1();
  Task_2();
  ...
  Task_n();
}
```

A single (infinite) loop, calling each task one after another



Advantages:

- **Simple**
- **no preemption**
- **Fast to implement/deploy**

Disadvantages:

- **Timings** (start and finish time of each job) **depends on computation times of *all* tasks**;
  - Potentially *unbounded jitter*, i.e. *drift*
- **Energy inefficient** (always execute even when not needed)
- **No** option for **"background" tasks**

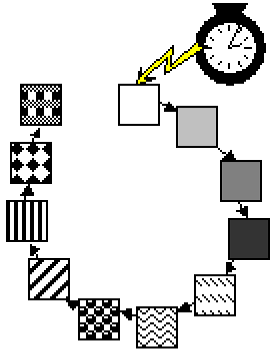
IRiS

Single-rate Time-driven AFAP

```
int k; /* activation counter */

k = -1;
while(1){
  k = k + 1;
  /* wait till absolute time phi + k * T */
  sleepUntil(phi+k*T)
  Task_1();
  Task_2();
  /* ... */
  Task_n();
}
```

Adds a timer to trigger the loop execution periodically



Advantages:

- Same as Single rate – AFAP
- **Resolves** problems with *drift* and *energy inefficiency* and *background tasks*

Disadvantage:

- All tasks are all executed at the ***same rate***
- We must have $\sum_{\tau_i \in \tau} C_i \leq T$

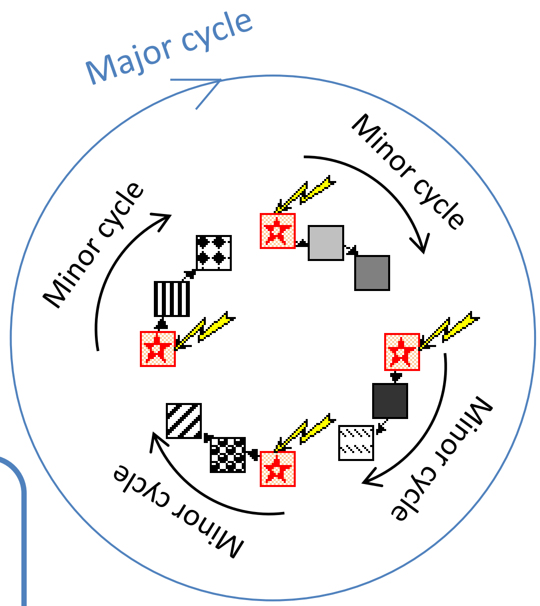
Multi rate Periodic

Division of time in a major and minor cycles

```
int k; /* cycle counter */

k = -1;
while(1){
  k = k + 1;
  /* wait till absolute time phi + k * T_1 */
  sleep(phi + k*T_1);
  Task_1();
  if( k % 2 == 0)
  { Task_2();
  } else {
    Task_3();
  }
}
```

Executes different set of tasks depending on the minor cycle number



Advantages:

- Same as single-rate time-driven AFAP
- **Reduced** start/finish-**jitter** of tasks
- **Not** all tasks execute at the **same frequency**
- No need for $\sum_{\tau_i \in \tau} C_i \leq T$

Summary

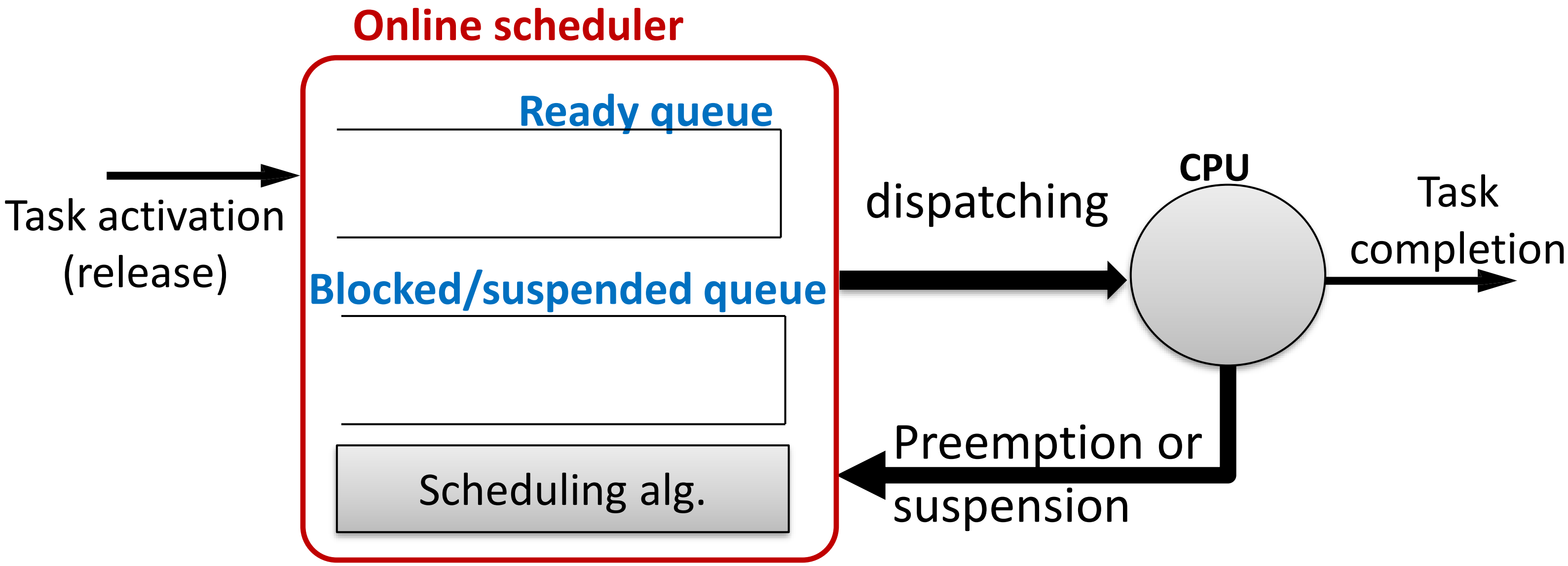
- Cyclic executives
  - Advantages
    - Fast to implement/deploy, no ‘‘preemption costs’’, simple:
      - requires just a hardware timer;
      - no need for shared resource access protocols.
  - Disadvantages
    - Limited to **periodic** events or **polling** sporadic events
    - Does **not support dynamic** systems where tasks may join and leave during the runtime
    - **Not robust** against overload
    - **Hard to maintain** when (see [Buttazzo], Section 4.2)
      - the frequency of a task must be updated
      - a task added
      - the execution time of a task increases
  - **Examples of use**
    - ROS/ROS2 (the Robot Operating System)
    - Lupo EL (developed by ME in 2009)
    - Signify LED drivers

A closer look at other known online scheduling policies

Online scheduler in an operating system

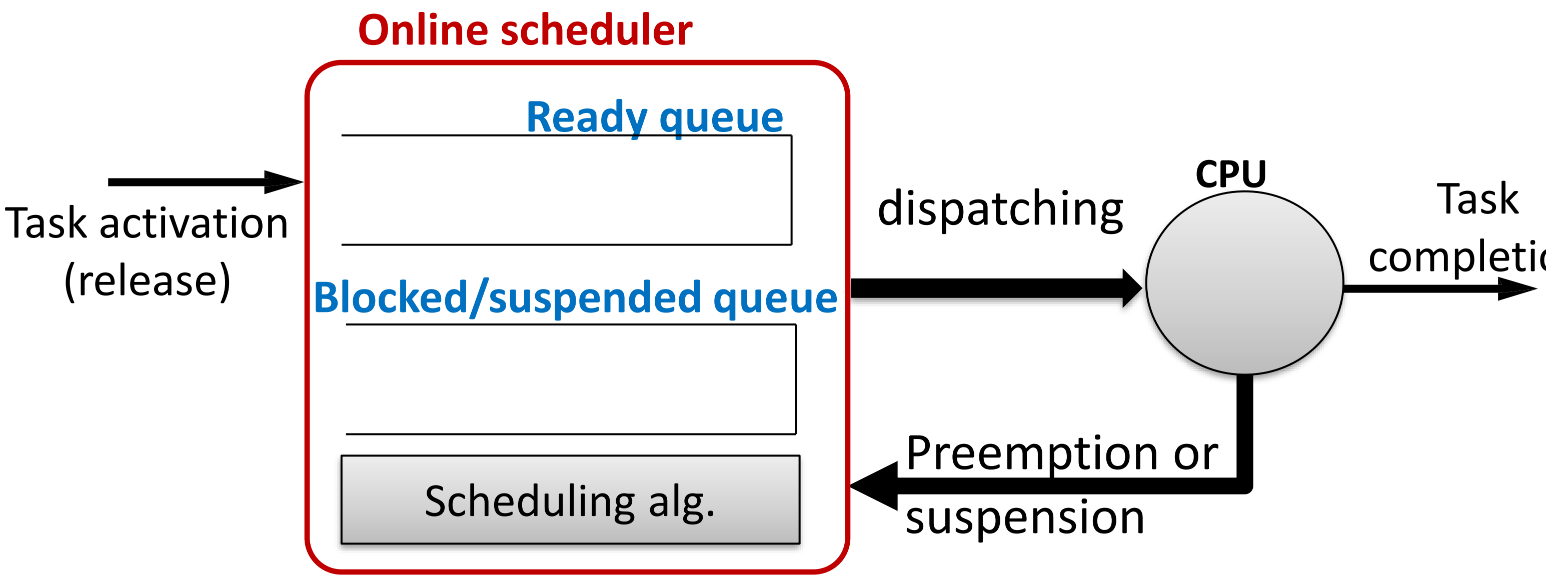
- In a concurrent system with one processor, several tasks can be simultaneously active, but only one can be in execution (running).

- An active task that is **not** in execution, blocked or suspended is said to be ready.

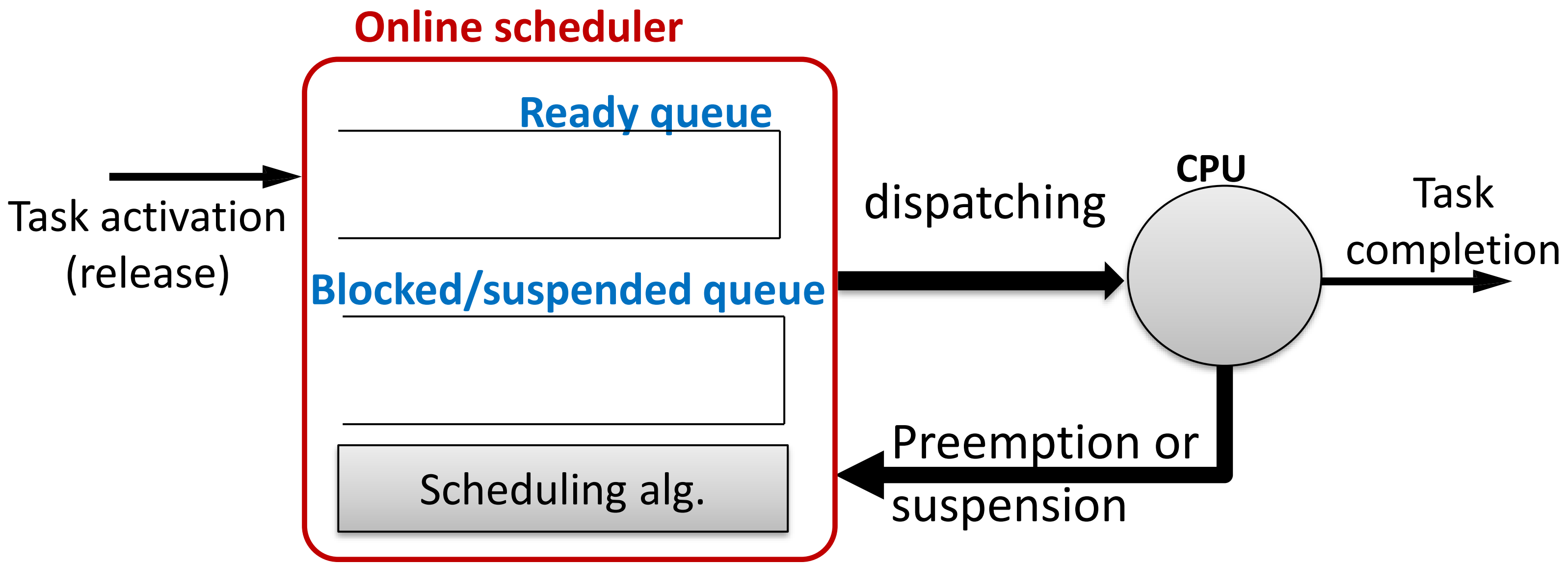


Online scheduler in an operating system

- Ready tasks are kept in a ready queue, managed by a scheduling policy.
- The scheduling policy determines which task is dispatched on the processor



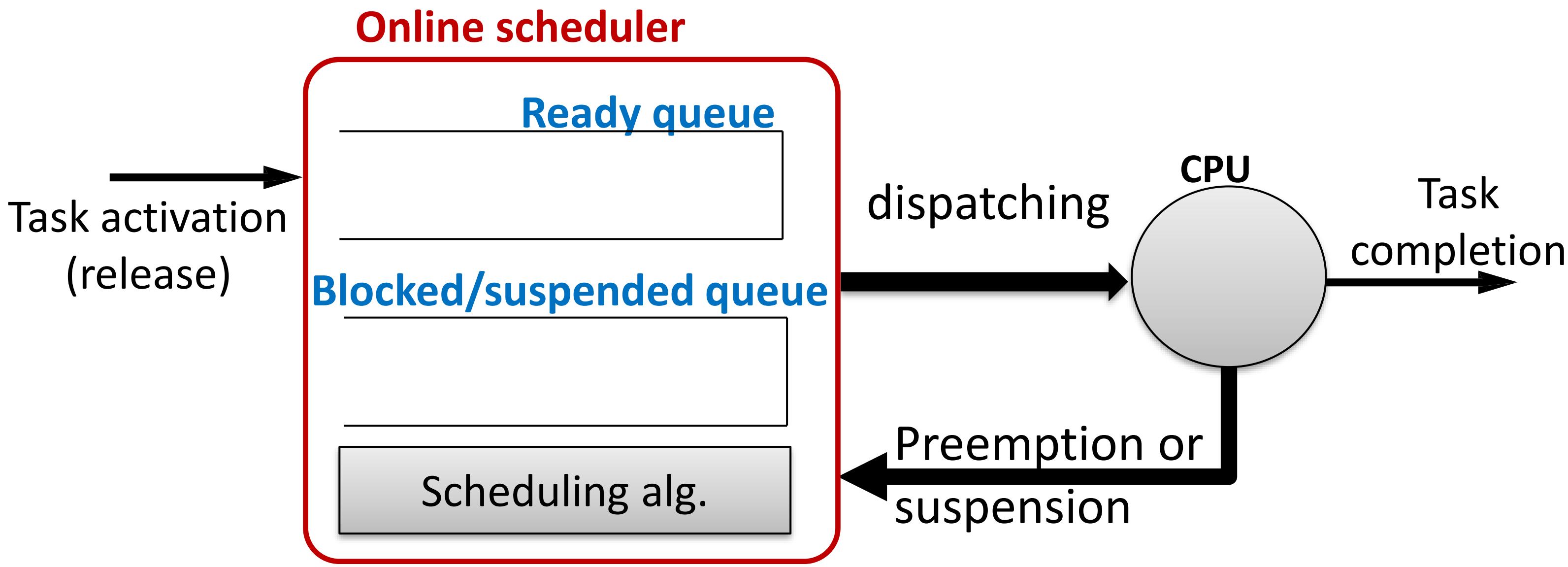
Preemption



Preemption is a kernel mechanism that allows to preempt the execution of the running task in favor of another task.
The preempted task goes back into the ready queue.

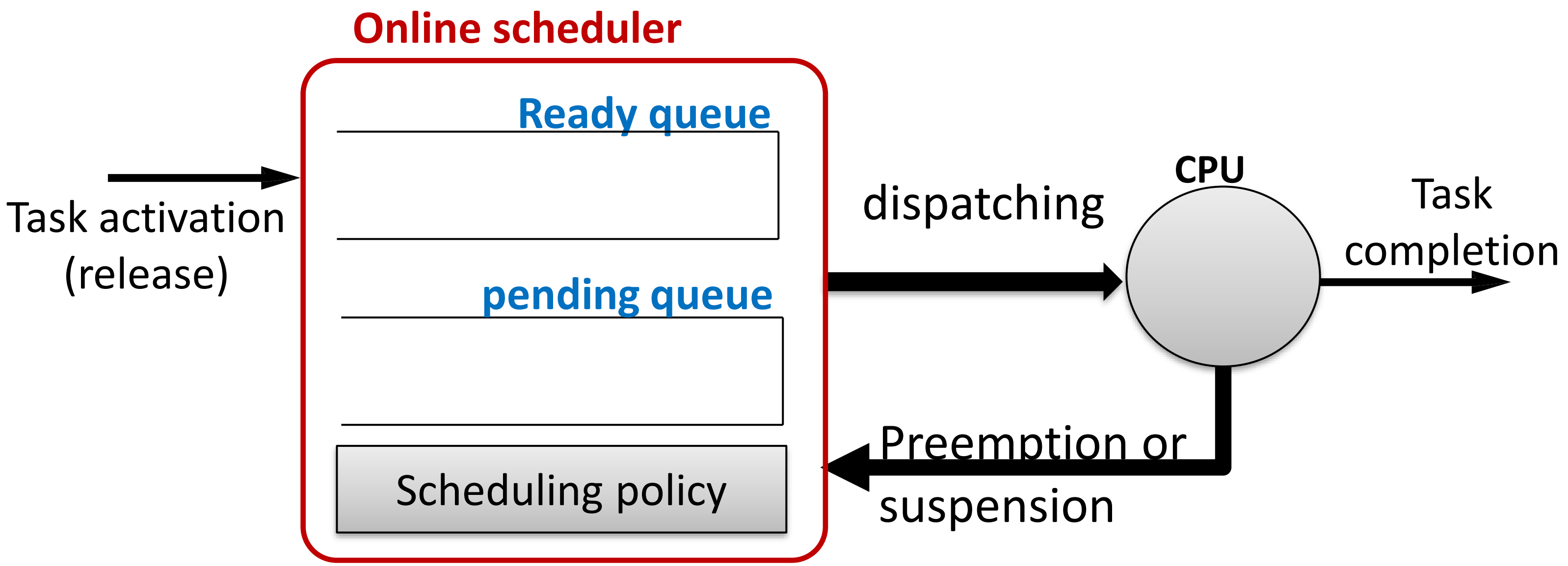
- Preemption enhances concurrency and may help reducing the response time of high priority tasks.
- It can be disabled (completely or temporarily) to ensure the consistency of certain critical operations.

Suspension



Suspension happens when a task decides to suspend itself (e.g., by “delay( )” or “sleep( )”), or the task makes a non-blocking call to an I/O or message queue or waits for results from another tasks or co-processor.
The suspended task goes in the suspended queue.

Online scheduling



What online scheduling policies (or algorithms) do you know?

Some well-known scheduling algorithms

- **First-in-first-out scheduling (FIFO)**

  = First-come-first-serve (FCFS)

- **Round robin**
- **Shortest-job first**
- **Earliest deadline first (EDF)**
- **Fixed-priority scheduling (FP)**

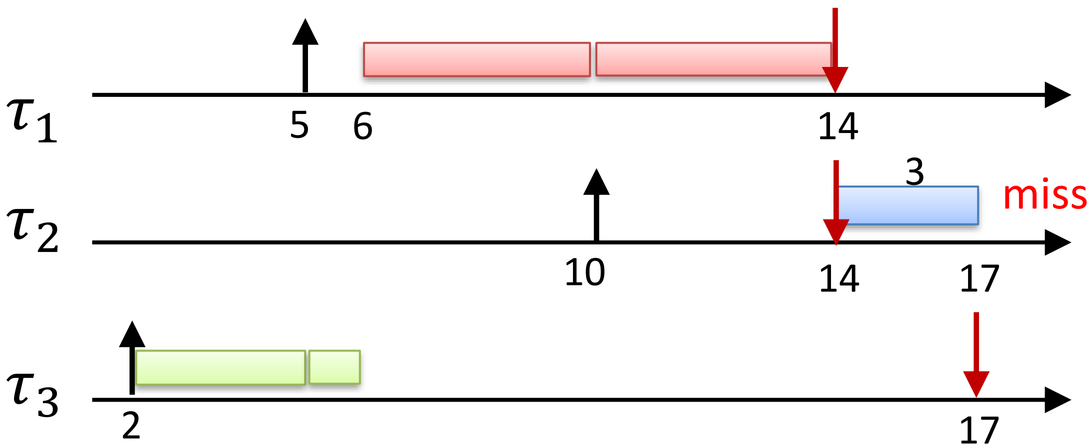
Online scheduling policies:
Static vs. dynamic priorities

- **Task-level static priorities**
  - Scheduling decisions are taken based on task’s fixed parameters that are known beforehand.
  - Examples: Rate Monotonic (RM), Deadline Monotonic (DM)
- **Job-level static priorities**
  - Scheduling decisions are taken based on parameters of the job known only at its release time.
  - Examples: FIFO or earliest-deadline first (EDF), which uses the absolute deadline of the jobs in order to decide which job has the highest-priority.

    Note: a job-level static scheduling policy such as EDF or FIFO is a task-level dynamic priority scheduling policy.
- **Job-level dynamic priorities**
  - Scheduling decisions are taken based on parameters that can change with time.
  - Example: scheduling policy with ageing, least-laxity first or shortest remaining execution time first policy

Example: FIFO (non-preemptive)

- The job with the earliest released is scheduled first

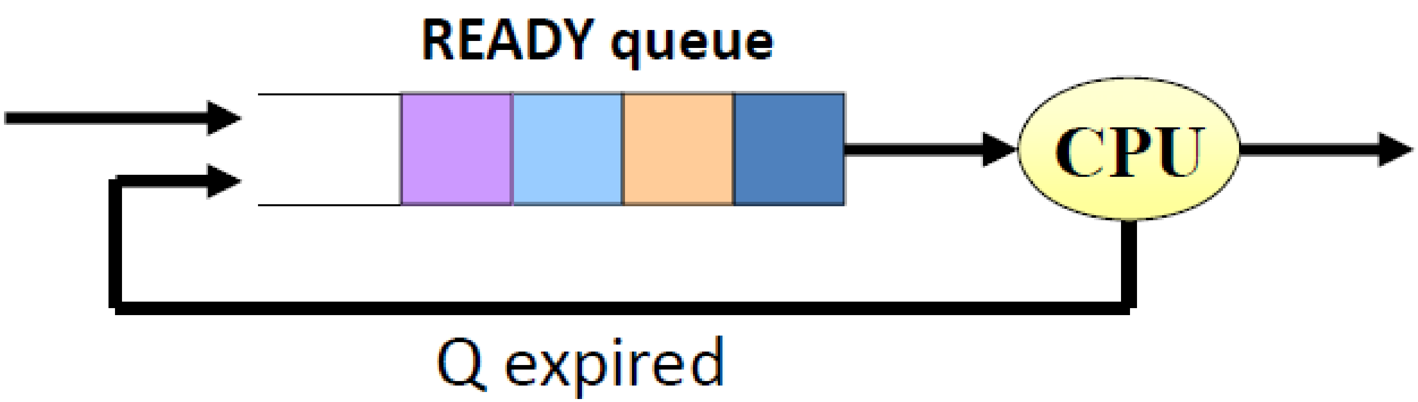


τ_i	C_i	$r_{i,1}$	$d_{i,1}$
τ_1	8	5	14
τ_2	3	10	14
τ_3	4	2	17

Assume that each task releases a single job

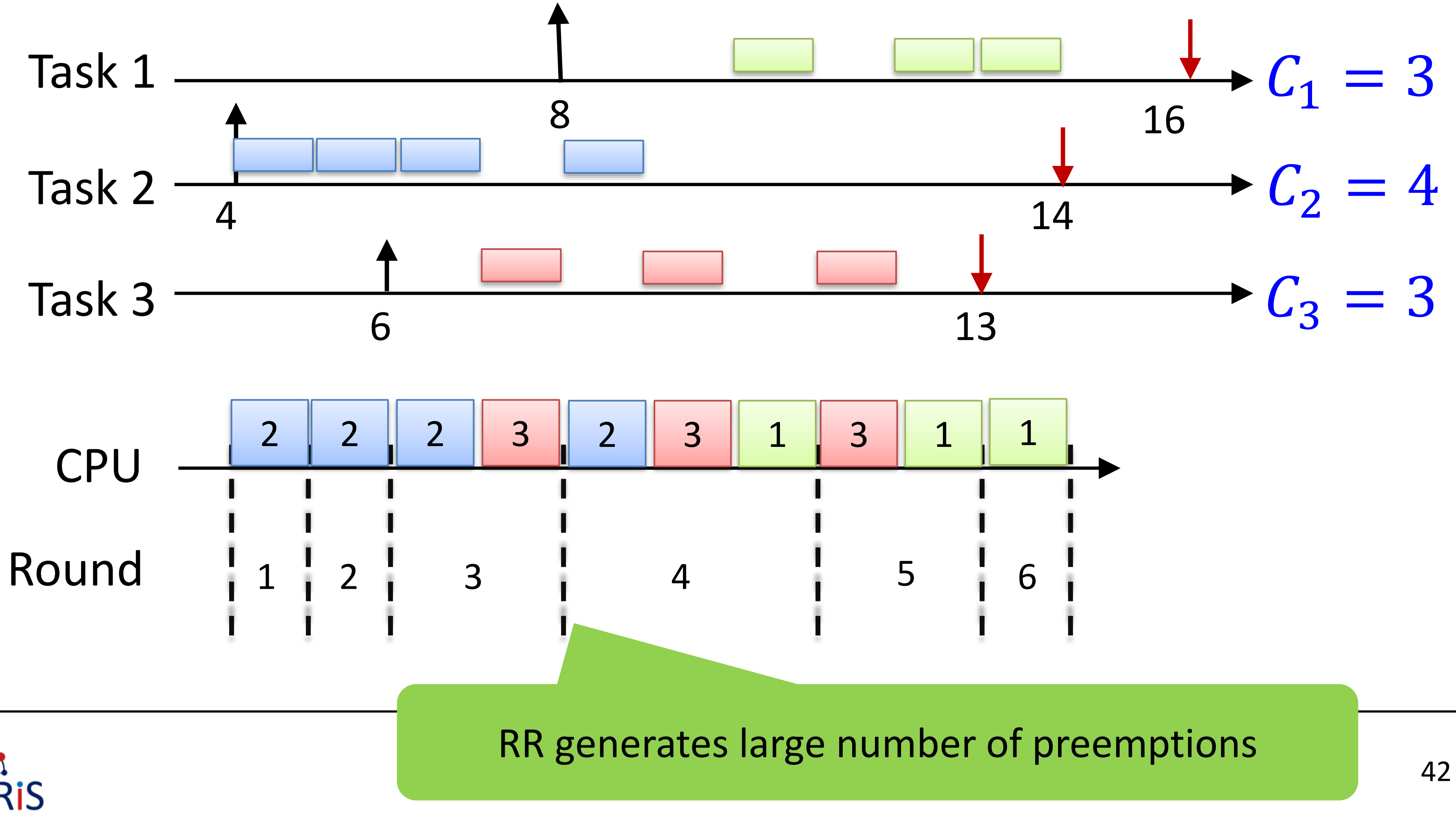
Round robin

- The ready queue is served with FIFO, but ...
- Each task τ_i cannot execute for more than **Q** time units (**Q** = time quantum).
- When **Q** expires, $\boldsymbol{\tau_i}$ is put back in the queue.

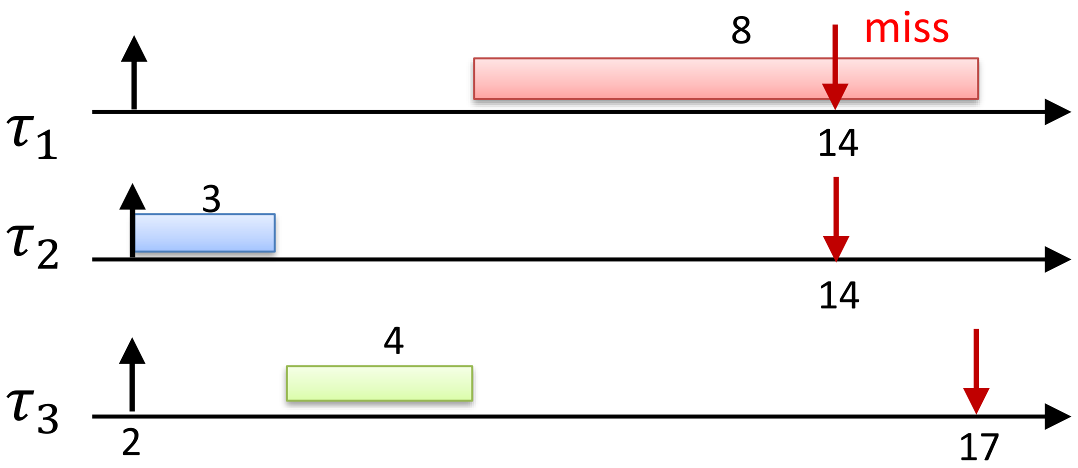


Round robin

- n = number of task in the ready queue
- Round robin creates "rounds" that are as long as $\boldsymbol{n \cdot Q}$
- **Assume $\boldsymbol{Q = 1}$**



Example: Shortest Job First (non-preemptive)



τ_i	C_i	$r_{i,1}$	$d_{i,1}$
τ_1	8	2	14
τ_2	3	2	14
τ_3	4	2	17

SJF is **difficult to implement** in practice because it **requires to estimate the execution time** of each job, which may be different from their worst-case execution times

Assume that each task releases a single job

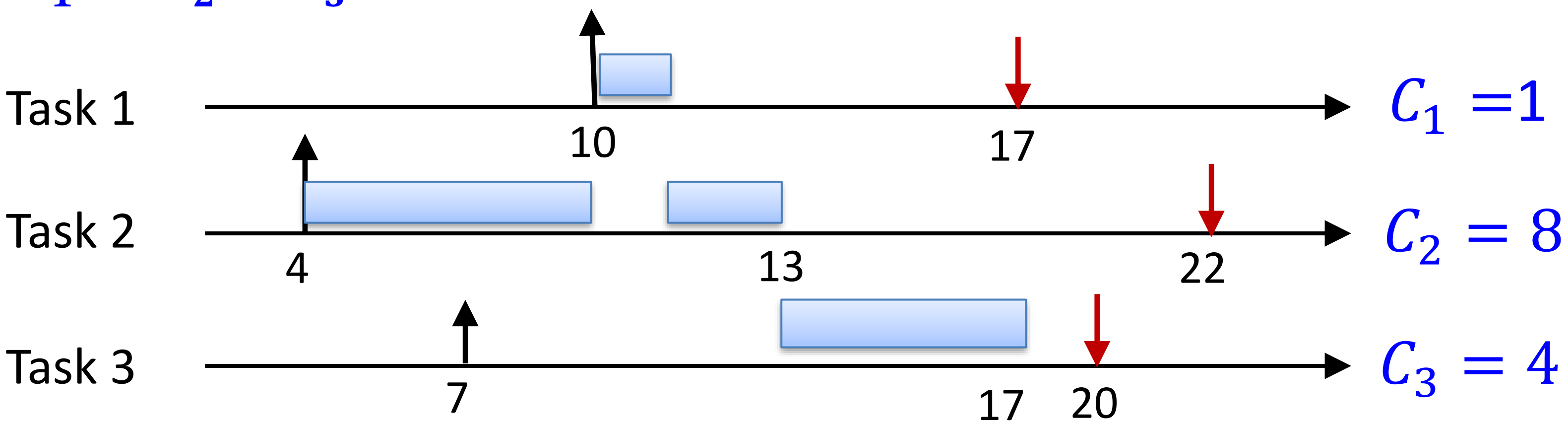
Fixed-priority scheduling (preemptive)

- Each task has a priority P_i, typically $P_i \in [0, 255]$

  ($P_i > P_j$ means that Task τ_i has a higher priority than task τ_j)

- The task with the highest priority is selected for execution.
- Tasks with the same priority are served in FIFO order

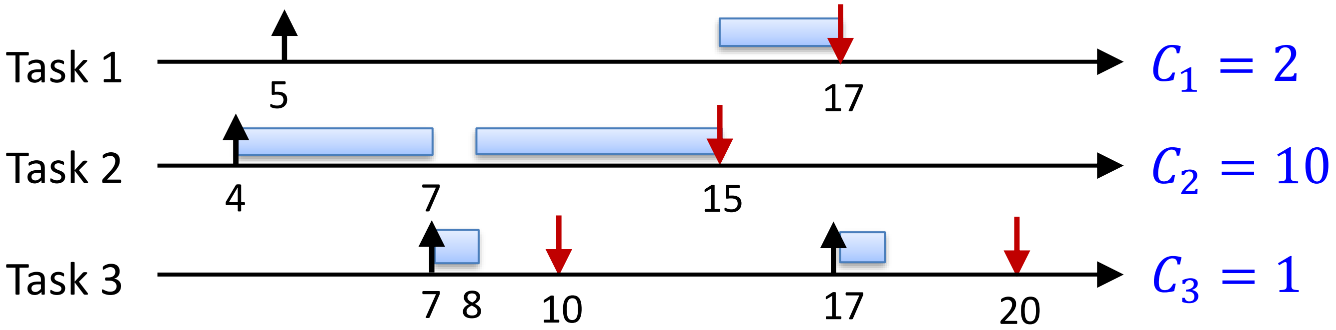
$\boldsymbol{P_1 > P_2 > P_3}$



Earliest deadline first (EDF)

- **Algorithm** [Horn 74]
  - Order the ready queue by **increasing absolute deadlines (job-level fixed priority)**.

Example of schedule with EDF



Earliest deadline first (EDF)

- **Algorithm** [Horn 74]

  - Order the ready queue by **increasing absolute deadline (job-level fixed priority)**.

- **Assumptions**

  Horn's algorithm is **preemptive** and is for **independent tasks** executed on a **single core** platform

- **Property**

  - Under the assumptions above, EDF minimizes **the maximum lateness (L_{max})**

$$L_{max} = max\{L_{i,j} \mid \forall J_{i,j} \in job\ set\}$$

➔ EDF is optimal from a feasibility viewpoint

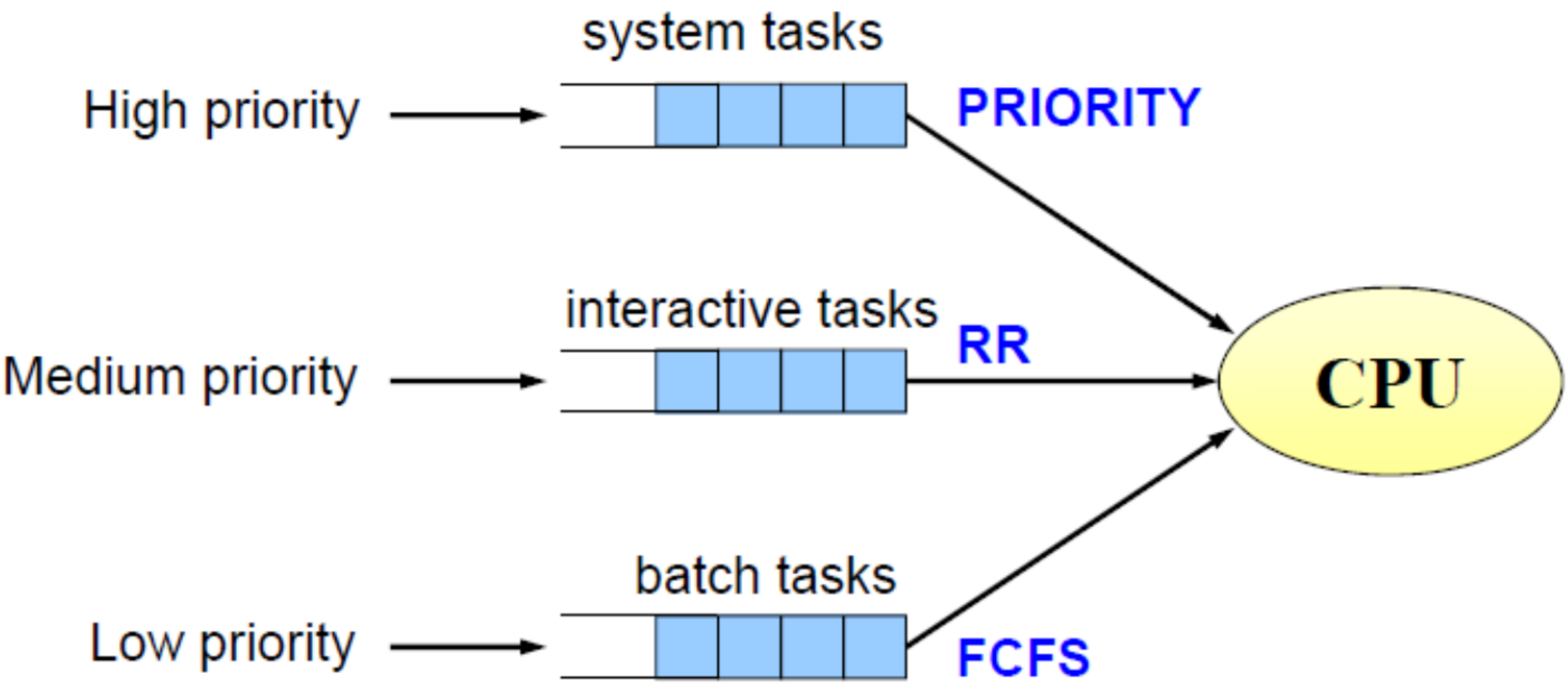
A property of optimal algorithms

If a task set is not schedulable by an optimal algorithm, then it cannot be schedulable by any other algorithm.

If an algorithm A **minimizes L_{max}** then A is also **optimal** in the sense of feasibility.
The opposite is not true.

➔ EDF is therefore optimal (w.r.t. feasibility) for the scheduling independent tasks on single core

Multi-level scheduling



Summary

- FIFO is fast and simple to implement, but bad at guaranteeing deadlines
- Round robin generates a large number of preemptions
- EDF is optimal w.r.t. feasibility (because it minimizes the maximum lateness)
- Table-driven scheduling is inflexible and trades computing complexity for memory complexity