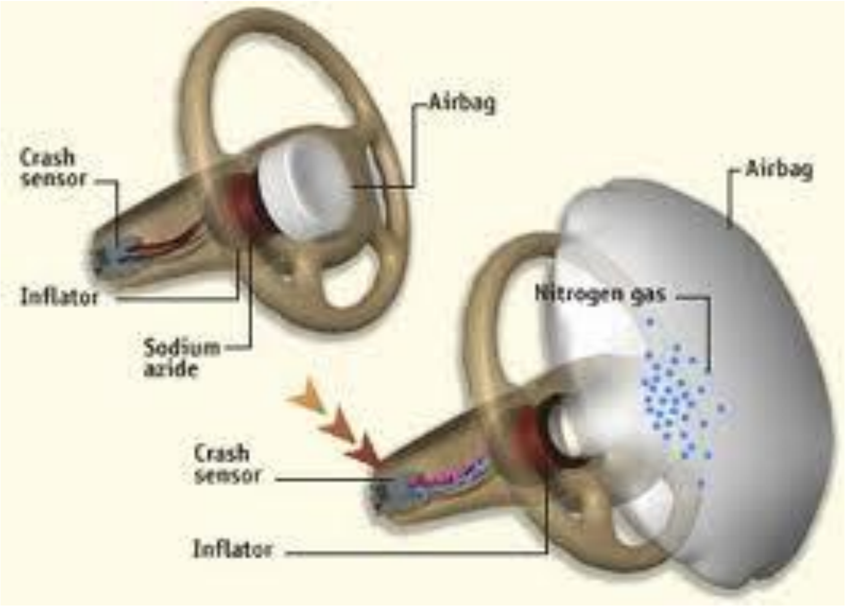
Real-Time Scheduling

Ramin Sadre
with material by Brian Nielsen, AAU

Real-time (RT) systems

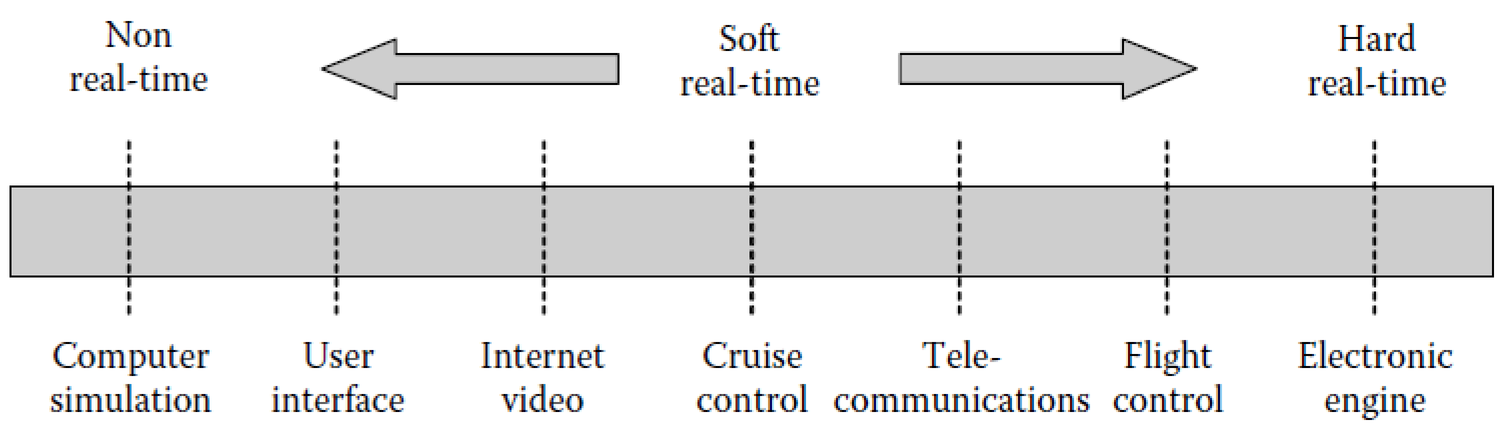
- Computer-based control and monitoring of physical equipment (via sensors and actuators)
- Requirements on timing. For example:

  Airbag must release 10ms-20ms after impact
- Requires RT-OS (= OS with ”predictable timing”)

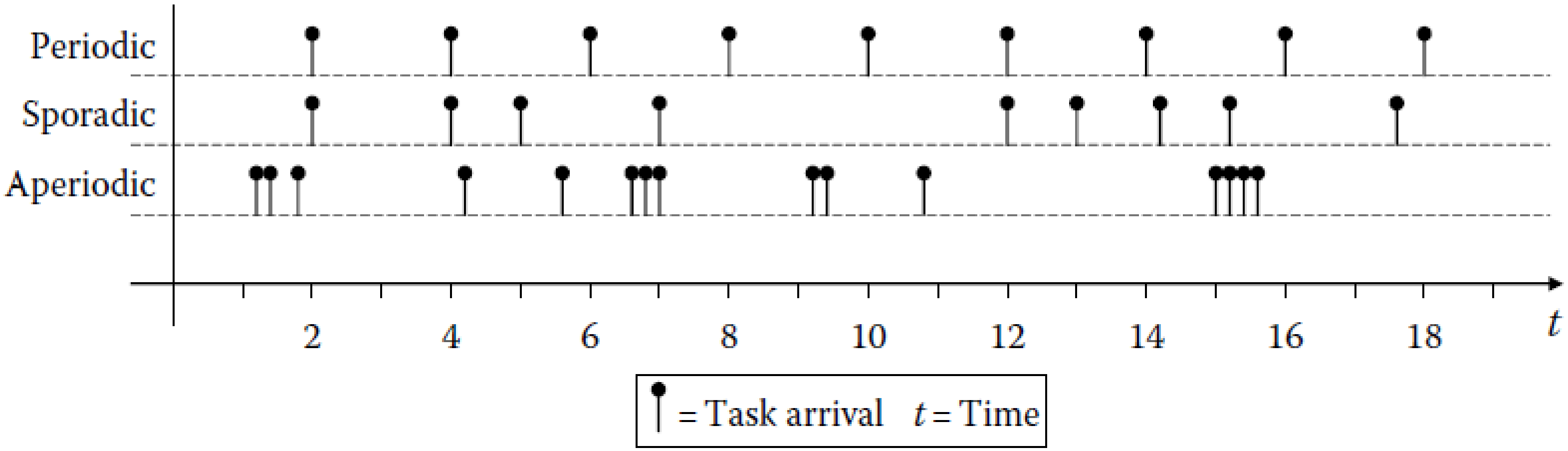


Tasks can have different requirements

- **Hard**
  - Potentially severe consequences if reaction/result is produced after a specified **deadline**
  - Results has no value (or even negative value) after deadline
- **Firm**
  - Occasional miss tolerated, but degrades QoS
  - No or little value after deadline
- **Soft**
  - Result still has value after deadline



Types of task behavior



- **Periodic tasks**: have fixed interarrival time (= period)
- **Sporadic tasks**: Have known minimum interarrival time
- **Aperiodic tasks:** No known interarrival time

Example

- Imagine an embedded system with three tasks:
  - Task 1: Every 5 seconds, the red LED should be toggled. This task should not be delayed more than 150ms. This task takes around 50ms.
  - Task 2: Sometimes (not more often than 10 times per second), a network packet arrives. This task should not be delayed more than 500ms. This task takes around 200ms.
  - Task 3: The user can push the emergency “STOP” button at any time to stop the device. The allowed delay to execute this task is 50ms.

Scheduling

- Tasks are executed by threads or processes on the CPU
- Problem: We have a limited resource here, the CPU. We cannot execute all tasks at the same time.
- Imagine the following situation:
  - At time t=15s, the system should toggle the LED. At the same time, a network packet arrives. Which task should the CPU execute first?
  - Answer: Task 1 should be executed first. If we executed task 2 first, task 1 would not be able to meet its deadline.
- This decision is called *scheduling*. In an OS, this is done by the *scheduler.*

Scheduling in general-purpose systems

- In general-purpose OS (Linux, Windows,...), the scheduler has the following goals:
  - Fairness: no starvation of processes
  - Optimize average response time: should be as short as possible
    (Response time = waiting time + execution time)
  - Optimize throughput: serve as many jobs as possible, *on average*

- Typically, such schedulers use some variation of *Round-Robin* (RR) scheduling with time slices

Refresher: Round-Robin Scheduling

- RR = Preemptive scheduling with time slices:
  1. All jobs wait in a queue for the CPU
  2. Job at the head of the queue receives service for time quantum *Q*
  3. Then it is interrupted (=*preempted*) by the next waiting job and goes back to the end of the queue
  4. Repeat

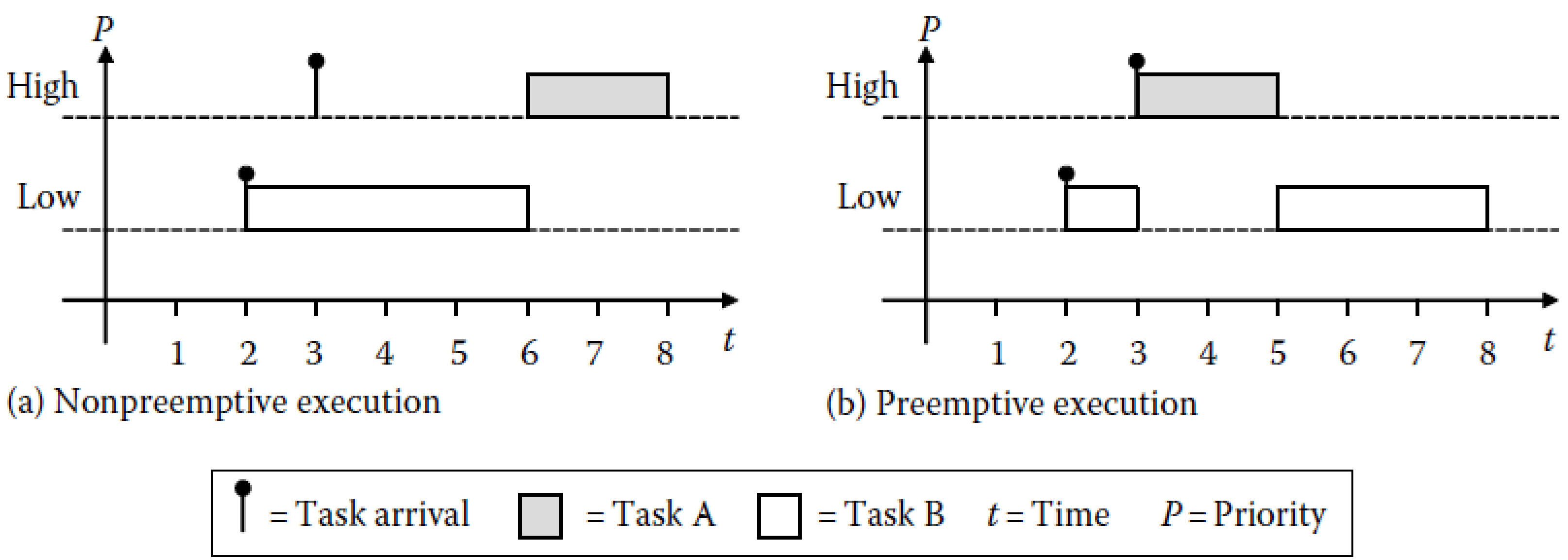
- In practice, one chooses a small *Q*
  - This gives the user the feeling that all processes run at the same time
  - However, *Q* should not be too small: overhead would become too high! (= context switch, cache misses,…)

Scheduling in real-time systems

- RT systems don't have the same requirements as general-purpose systems:
  - Real-time execution: tasks must meet their deadlines!
  - *Worst-case* performance is more important than average performance: 100ms average response time is nice, but can the scheduler *guarantee* that the response time is *never* longer than 150ms?
  - Fairness not so important: In an airplane, the engine-management process is more important than the coffee-machine process

Scheduling algorithms

- There are many possible scheduling algorithms
  - Round-Robin
  - Shortest-Job Next
  - With/without priorities
  - Preemptive vs non-preemptive
  - …



Schedulability

- Given: a set of tasks with their periods, deadlines, etc.
- Given: a scheduling algorithm
- Definition: The task set is *schedulable* if the scheduler can schedule the tasks such that all tasks meet their deadlines

- How can we know whether a task set is schedulable for a given scheduling algorithm?
  1. Measure on the real system: expensive! We have to implement and deploy it
  2. Simulate: TOSSIM (TinyOS), Cooja (Contiki),…
  3. <u>Mathematical analysis</u>

Rate Monotonic (RM) scheduling

- Let's assume we have a system with N periodic tasks with periods $T_1, \dots, T_N$ and deadlines $D_1, \dots, D_N$
- RM assigns a static priority to each task:
  - Task with shortest period has highest priority
- RM schedules tasks with preemption: a task can preempt other tasks with lower priority

- Example: System with three tasks

Task	T_i	Priority automatically assigned by RM
X	30ms	High
Y	40ms	Medium
Z	52ms	Low

Schedulability Analysis for RM

- Given a set of periodic tasks with periods $T_1, \dots, T_N$ and deadlines $D_1, \dots, D_N$, is the task set schedulable with RM?
- In general, difficult to answer! To simplify the problem we assume:
  - System has only one CPU
  - *No priority inversion* (we will see that later)
  - Zero overhead for context switching
  - Number of tasks is constant
  - $D_i = T_i$ for all tasks
- Under the above conditions, it holds:
  - RM is an optimal preemptive static priority scheduling algorithm. That means: If a task set is not schedulable with RM, *no other preemptive static priority scheduling algorithm can schedule it*!

A sufficient schedulability test for RM

- Let's assume we know for each task its worst-case execution time C_i, i.e., its execution time is always $\leq C_i$
- If we assume $D_i = T_i$ for all tasks, a simple test for schedulability with RM is

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1)$$

Total CPU utilization

- This is a *sufficient but not necessary* test

Schedulability test for RM: Example

Task	T_i	C_i	$D_i = T_i$	Priority	C_i/T_i
X	30	10	30	High	0.33
Y	40	10	40	Medium	0.25
Z	52	10	52	Low	0.23
					$\sum_{i=1}^{N} \frac{C_i}{T_i} = 0.81$
					$N\left(2^{\frac{1}{N}} - 1\right) = 0.78$

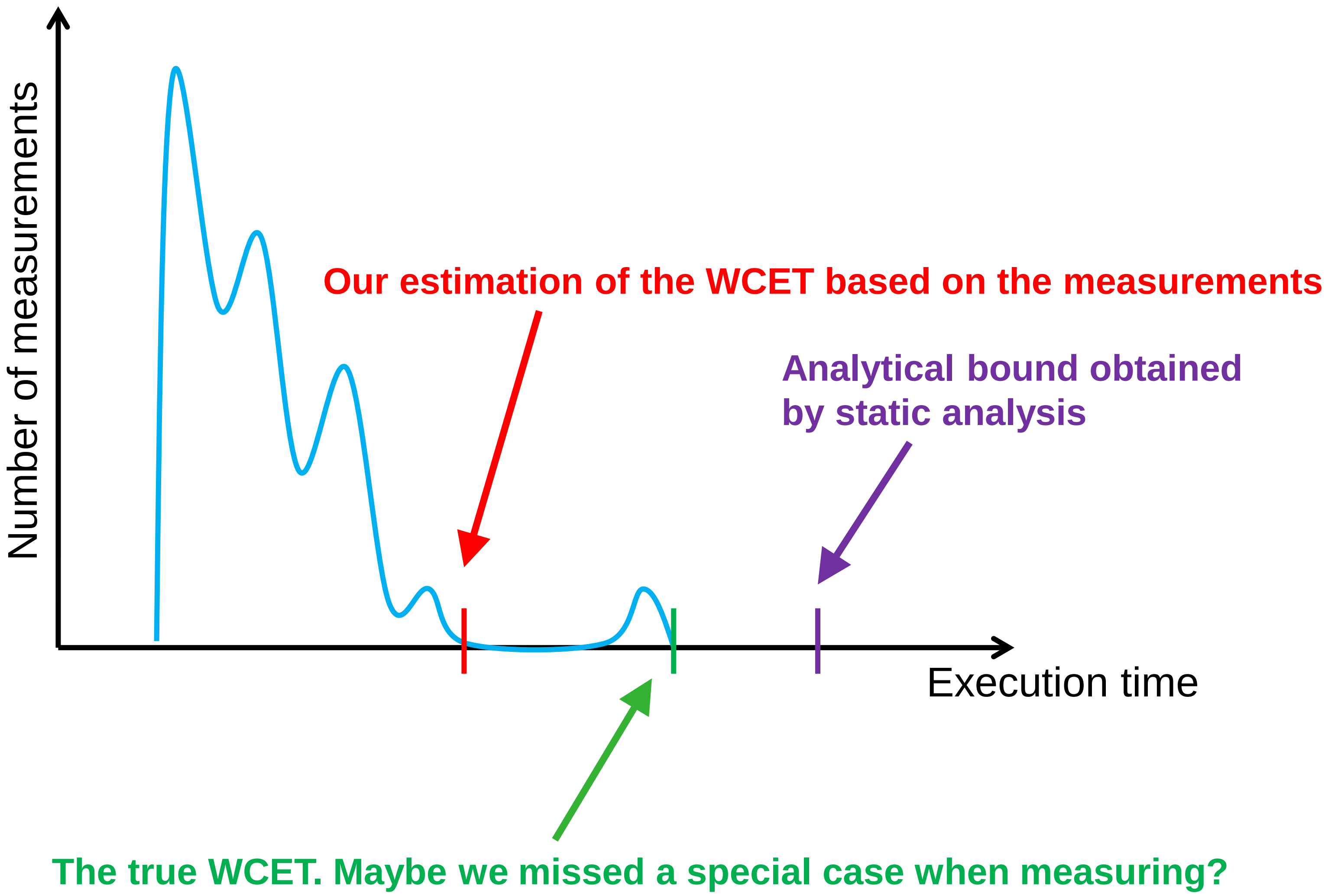
0.81 > 0.78

→ this task set *might* not be schedulable with RM
(the test is sufficient but not necessary!)

Worst-Case Execution Time (WCET)

- How do we know C_i of a task?
  - We could measure it
  - Static analysis based on source code or design specification
- Static analysis:
  - Compute maximum CPU cycles for each path in the program under worst-case variable assignment
  - Problem 1: Halting problem
  - Problem 2: Difficult to make a good estimation of number of cycles on complex CPUs (cache, piplelines, DMA,...)

Worst-Case Execution Time Analysis



Deadline Monotonic (DM) scheduling

- RM assigns priorities based on task periods T_i. It completely ignores the deadlines D_i.
- Maybe not a good idea! That means a task with period 100ms and deadline 100ms will get a higher priority than a task with period 110ms and deadline 5ms!

- DM scheduling = preemptive static priority scheduling and
  task with *shortest deadline* has highest priority

- Note that DM is identical to RM if all tasks have $D_i = T_i$

Schedulability analysis for DM

- It can be shown that DM is an optimal preemptive static scheduling algorithm for tasks with $D_i < T_i$.

That means: If a task set with tasks $D_i < T_i$ is not schedulable with DM, *no other preemptive static priority scheduling algorithm can schedule it*.

- Sufficient but not necessary test for schedulability with DM:

$$\sum_{i=1}^{N} \frac{C_i}{D_i} \leq N(2^{\frac{1}{N}} - 1)$$

Response Time Analysis (RTA)

- RTA provides a sufficient and necessary test for all fixed-priority preemptive scheduling algorithms (RM, DM,…)
  - More complicated than the other tests we have seen

- A task set is schedulable if worst-case response time R_i

$$R_i \leq D_i$$

  for all tasks.

- Response time R_i consists of:

$$R_i = C_i + I_i$$

C_i = worst case execution time

I_i= worst case interference time = amount of time a task is delayed by execution of higher priority tasks

Calculating R_i

$$R_i = C_i + I_i$$

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of tasks with higher priority than task i

$\left\lceil \frac{R_i}{T_j} \right\rceil$ = number of preemptions of task i by task j

Calculating R_i

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

is a recursive formula.

Solve by finding smallest fixed point iteratively:

$$R_i^0 = C_i$$

$$R_i^{m+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^m}{T_j} \right\rceil C_j$$

RTA: Example

- Let's assume we have the following task set and we are using RM scheduling:

Task	T	C	D	Priority assigned by RM
X	30	10	25	High
Y	40	10	20	Medium
Z	52	10	52	Low

- We use RTA:

Task	T	C	D	Priority	R	RTA test: $R \leq D$?
X	30	10	20	High	10	Yes
Y	40	10	30	Medium	20	Yes
Z	52	10	52	Low	52	Yes

- Conclusion: the task set is schedulable with RM, i.e. the tasks will always meet their deadlines with RM

Earliest Deadline First (EDF) scheduling

- RM and DM use static priorities
- EDF is an optimal preemptive dynamic priority scheduling algorithm
  - Task with earliest *absolute* deadline has the highest priority

  (Absolute deadline = time when the job is ready + D_i)
- Schedulability test if we assume that $D_i = T_i$:

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

Total CPU utilization

That means: As long as total CPU utilization is not more than 100%, EDF guarantees schedulability!

Problems of EDF scheduling

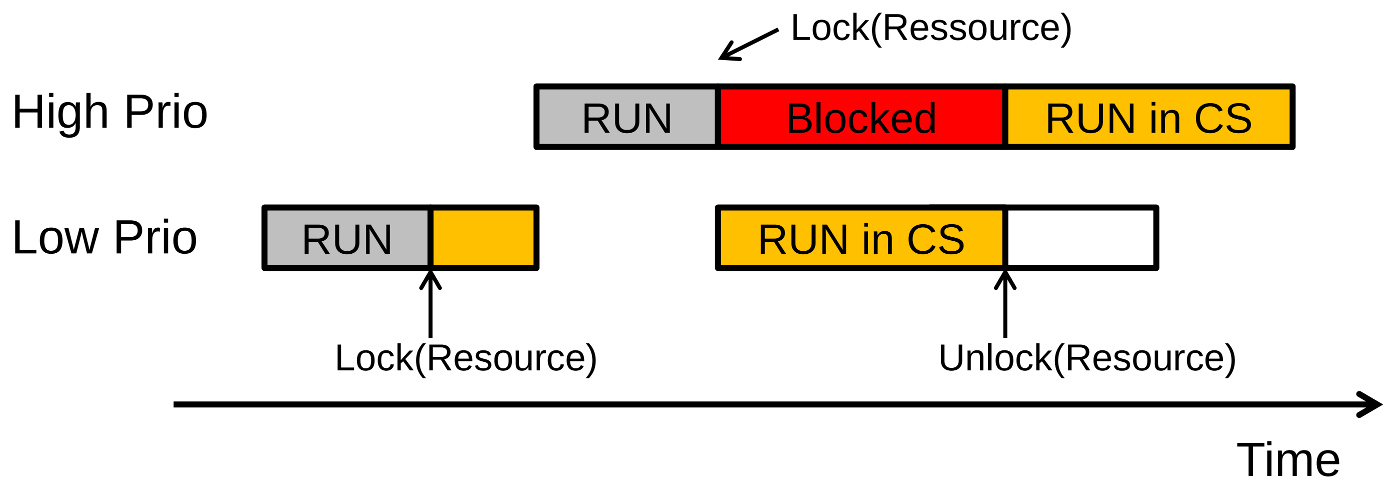
1. When utilization is higher than 100%, tasks will miss their deadlines. It's very hard to predict which tasks will be affected.
   - Not a nice property to build reliable systems
2. Not easy to implement
   - Requires a scheduler which recalculates the dynamic priorities during runtime
   - Small inaccuracies in the clocks can disturb the system

- For these reasons, fixed-priority schedulers are often preferred in real-time systems: they are easier to implement and easier to understand

Priority Inversion

Dependent tasks

- So far, we have assumed that all tasks are independent
- In a real system, tasks can be dependent because they access a shared resource in a critical section (CS)

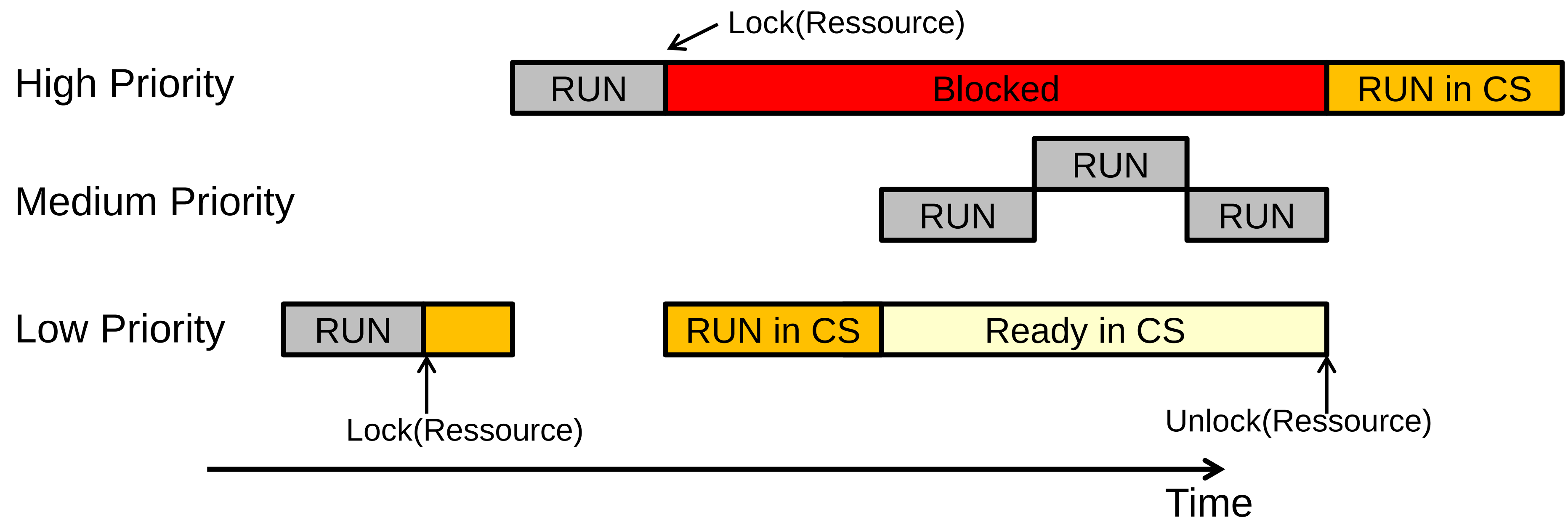


- Response time becomes:

$$R_i = C_i + I_i + B_i$$

(Warning: RTA test only sufficient, not necessary, with blocking!)

Priority Inversion



Pretty bad! The task with highest priority can be unboundedly blocked by tasks with lower priority

Priority Inversion in practice…

Read how the priority inversion problem nearly caused the loss of the Mars Pathfinder mission in 1997 (running VxWorks):

https://www.microsoft.com/en-us/research/people/mbj/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Fmbj%2Fmars_pathfinder%2Fauthoritative_account.html#!just-for-fun

Priority Inheritance Protocol

- Idea to avoid the problem:
  While in the CS, the low-priority task inherits the priority of the blocked high-priority task

  → no preemption by medium-priority tasks

- Protocol:
  - When task i is blocked by a CS held by task k and prio(i)>prio(k) → prio(k):=prio(i)
  - When task k leaves the CS:
    - If task k no longer blocks any tasks, it returns to its old priority
    - If task k still blocks other tasks, it inherits their highest priority

(Immediate) Priority Ceiling Protocol

- Alternative to priority inheritance
- Let's assume a shared resource R can be accessed only by tasks $S_R = \{t_1, \ldots, t_m\}$
- Assign a priority ceiling C_R to that resource

$$C_R = \max_{t_i \in S}(prio(t_i))$$

- When a task locks that resource, its priority is immediately boosted to C_R

- Note that priority inheritance and priority ceiling require a scheduler that can handle dynamic priorities

Scheduling of Non-Periodic Tasks

Non-periodic tasks

- So far, we have assumed that all tasks are periodic
- What about aperiodic/sporadic tasks?
- Different solutions possible. We will not discuss them in detail here.
  - We could treat aperiodic/sporadic tasks as periodic tasks with very small period

    → not very efficient
  - We could treat aperiodic/sporadic tasks as low-priority tasks that are executed when no periodic task needs the CPU

    → aperiodic/sporadic tasks might starve if periodic tasks consume all the CPU
  - We could replace all aperiodic/sporadic tasks by <u>one</u> periodic "server" task: aperiodic/sporadic events are queued