CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Parallel & Distributed Real-Time Systems

Lecture #4

Risat Pathan

Department of Computer Science and Engineering
Chalmers University of Technology

Student Representatives

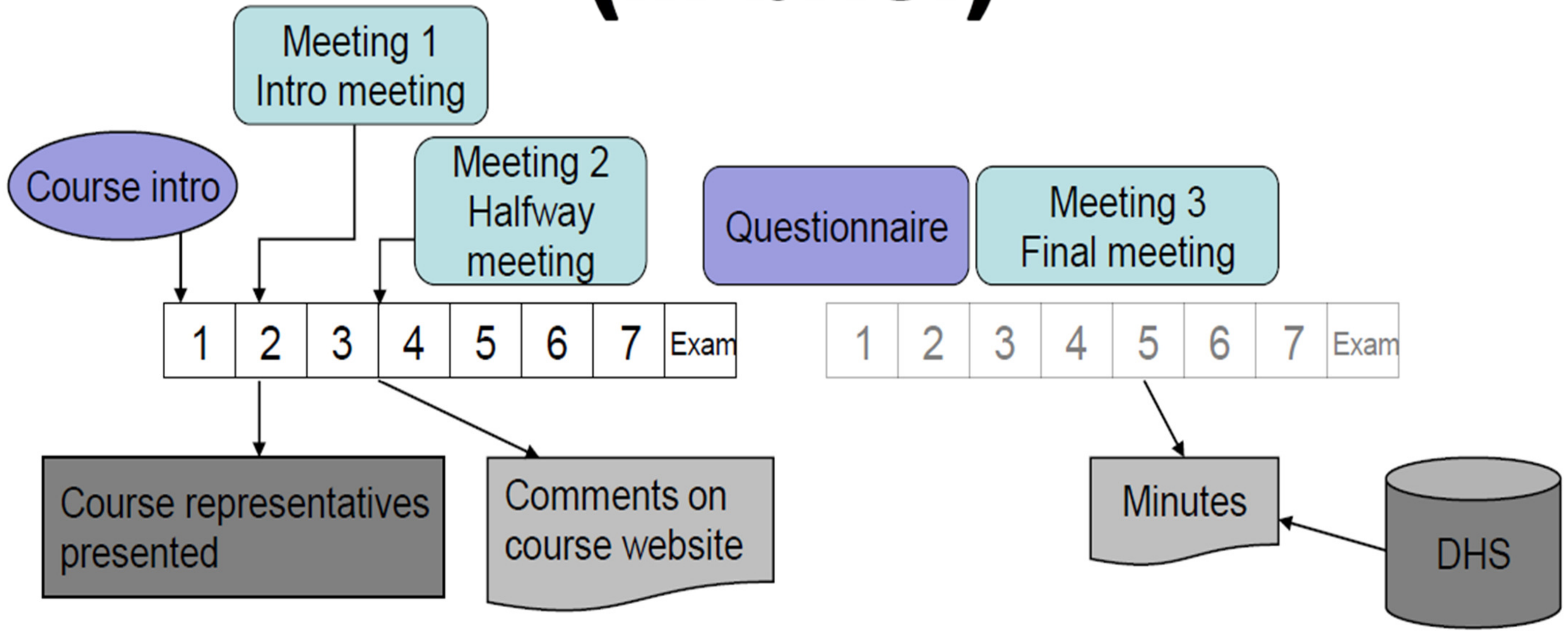
VIKTOR BOTEV	botev@student.chalmers.se
MICHAEL JASINSKI	jasinski@student.chalmers.se
VIKTOR DAHL	dviktor@student.chalmers.se

Contact information will be available in homepage

Course Evaluation Procedure

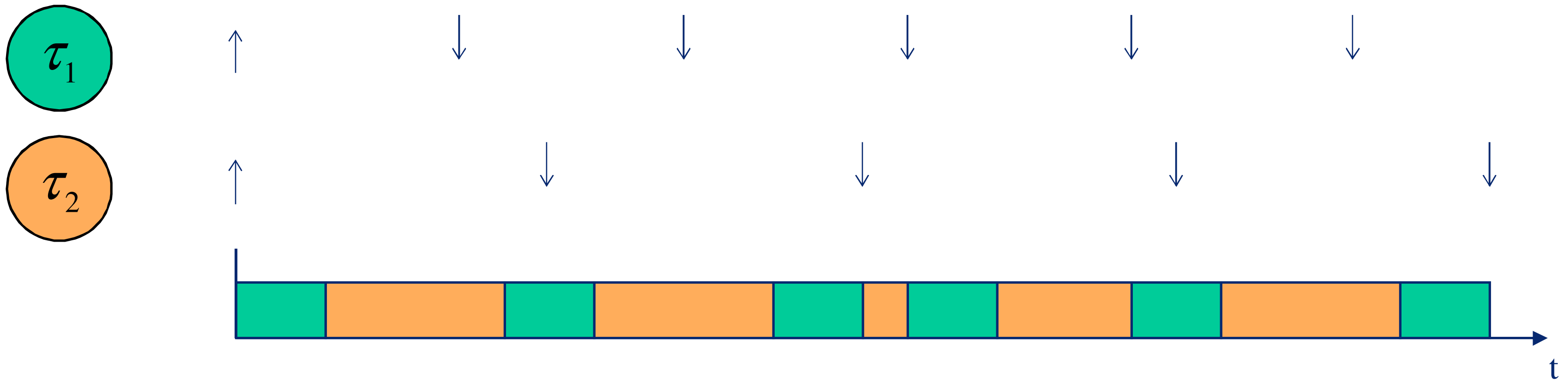
- Each course at Chalmers is evaluated with an **intro meeting**, a recommended **half-way meeting** and **a final meeting**.
- Questionnaires are sent out to all courses by the end of the exam week. And, can be answered until the end of Sw2 in the following Sp.
- When the questionnaires have closed, the final meeting is held.
- The final meeting is led by a programme representative.

What's the procedure? (in brief)



Scheduling

Attempts to meet application constraints should be done in a proactive way through scheduling.



Schedule = resources + operations on a time line

Scheduling

Scheduling is used in many disciplines:
(a.k.a. ”operations research”)

- Production pipelines (“Ford’s automotive assembly line”)

  Actors: workers + car parts
  Goal: generate schedules that maximizes system throughput (cars per time unit)
  Technique: job- and flow-shop scheduling

- Real-time systems

  Actors: processors, data structures, I/O hardware + tasks
  Goal: generate schedules that meet timing constraints (deadlines, periods, jitter)
  Technique: priority-based task scheduling

Scheduling

Scheduling is used in many disciplines:
(a.k.a. ”operations research”)

- Classroom scheduling

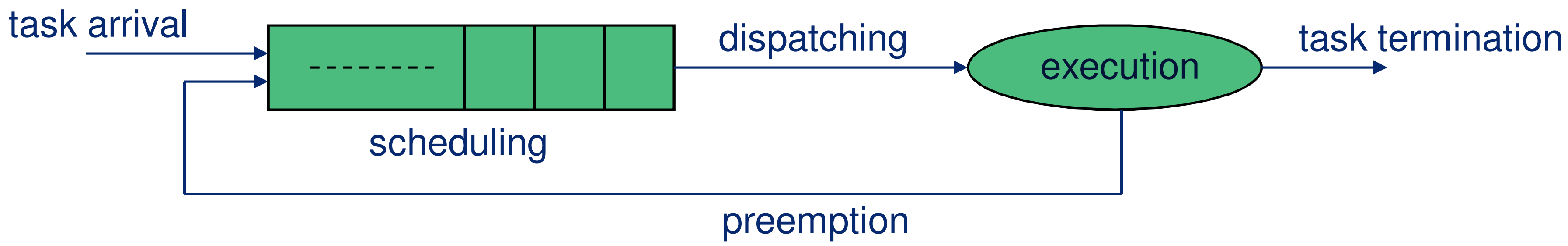
  Actors: classrooms, teachers, projectors + courses
  Goal: generate periodic schedules within 7-week blocks
  Technique: branch-and-bound algorithms

- Airline crew scheduling

  Actors: aircraft, staff + routes
  Goal: generate periodic schedules that minimizes the number of aircraft and staff used and fulfill union regulations for staff
  Technique: advanced branch-and-bound algorithms

Scheduling

- A scheduling algorithm generates a schedule for a given set of tasks and a certain type of run-time system.
- The scheduling algorithm is implemented by a scheduler that decides in which order the tasks should be executed.
- Observe that the scheduler selects which task should be executed next, while the dispatcher starts the execution of the selected task.



Scheduling

A schedule is said to be feasible if it fulfills all application constraints for a given set of tasks.

A set of tasks is said to be schedulable if there exists at least one scheduling algorithm that can generate a feasible schedule.

Scheduling

A scheduling algorithm is said to be optimal with respect to schedulability if it can always find a feasible schedule whenever any other scheduling algorithm can do so.

A scheduling algorithm is said to be optimal with respect to a performance metric if it can always find a schedule that maximizes/minimizes that metric value.

Scheduling constraints

Examples of scheduling constraints:

- No processor sharing:
  - A processor can only execute one task at a time
  - This is a realistic assumption for any processor type being used in practice
  - Note: in case of multi-core processors, each core is viewed as a separate processor
- No dynamic task parallelism:
  - A task can only execute on one processor at a time
  - This is a realistic assumption for any programming model being used in practice

Scheduling constraints

Examples of scheduling constraints:

- Non-preemptive scheduling:
  - Once started, a task cannot be preempted by another task
  - This assumption is not so common in priority-based scheduling
- Greedy scheduling:
  - Once started, a task cannot be preempted by a lower-priority task
  - This assumption applies for all run-time systems used in practice
- No task migration:
  - A task can only execute on one given processor, or cannot change processor once it has started its execution
  - This is a realistic assumption for distributed systems, and is also enforced for some multi-core processor designs (e.g. AUTOSAR)

Scheduling constraints

Non-preemptive scheduling:

- Advantages:
  - Mutual exclusion can be automatically guaranteed
  - Results from WCET analysis correspond well with real WCET behavior
- Disadvantages:
  - Negative effect on schedulability
    - Scheduling decision takes effect only after a task has completed its execution
    - Once a task starts executing, all other tasks on the same processor will be blocked until execution is complete

Scheduling constraints

Preemptive scheduling:

- Advantages:
  - Schedulability is not negatively affected
    - Scheduling decisions can take effect as soon as the system state changes (even in the middle of task execution)
    - The capacities of task priorities can be used in full
- Disadvantages:
  - Mutual exclusion has to be guaranteed by e.g. semaphores (or similar constructs)
  - WCET analysis is more complicated since cache and pipeline contents will be affected by a task switch
  - Program security may be compromised (through so-called *covert channels*) if full preemption is allowed

Scheduling constraints

Greedy scheduling:

- Example: ”traditional” static-priority scheduling (RM, DM)
  - Once a task starts executing, lower-priority tasks cannot grab the processor until execution is complete
- Advantages:
  - Scheduler relatively simple to implement
  - Supported by all run-time systems used in practice
- Disadvantages:
  - Schedulability is negatively affected:
    - Lower-priority tasks can starve and hence miss their deadlines

Scheduling constraints

Fair scheduling:

- Example: p-fair scheduling (Baruah et al. 1995)
  - Although a task has started executing, lower-priority tasks receive a guaranteed time quantum per time unit for execution
  - All tasks hence make some kind of progress per time unit
- Advantages:
  - Schedulability can be maximized on a multiprocessor system (assuming that task switch cost is negligible)
- Disadvantages:
  - Not supported by run-time systems used in practice
  - Poor schedulability when task switch cost is non-negligible
    - Fairness implies significantly more task switches than greediness

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Scheduling algorithms

How much an oracle is the scheduling algorithm?

- Myopic scheduler:
  - Scheduling algorithm only knows about currently ready tasks.
  - Scheduling decisions are only taken whenever a new task instance arrives or a running task instance terminates.
- Clairvoyant scheduler:
  - Scheduling algorithm ”knows the future”; that is, it knows in advance the arrival times of the tasks.
  - On-line clairvoyant scheduling is difficult to realize in practice.

”Predictions are always hard to make. In particular about the future.”
(Yogi Berra)

Scheduling algorithms

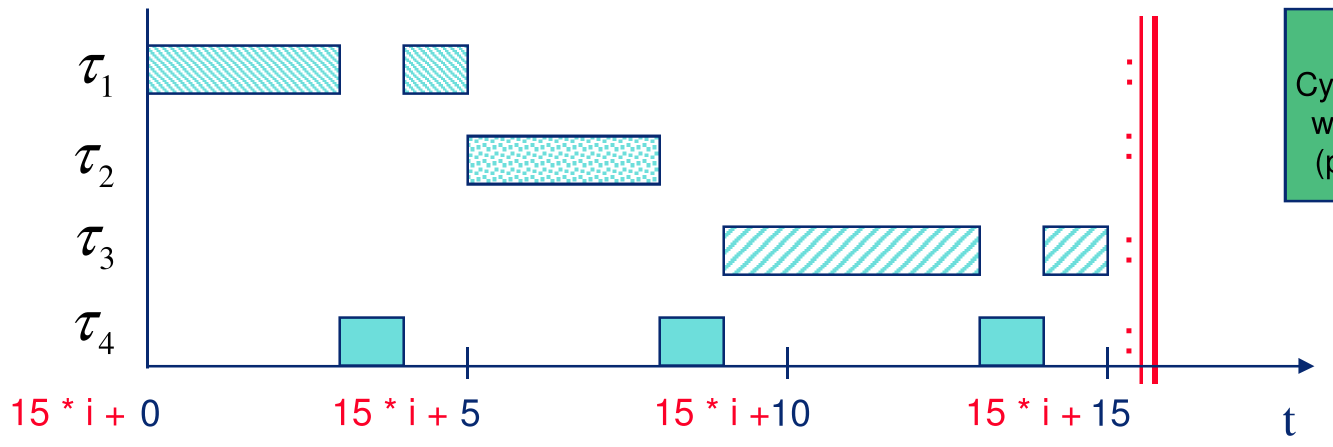
When are schedules generated?

- Static scheduling:
  - Schedule generated ”off-line” before the tasks becomes ready, sometimes even before the system is in mission.
  - Schedule consists of a ”time table”, containing explicit start and completion times for each task instance, that controls the order of execution at run-time.
- Dynamic scheduling:
  - Schedule generated ”on-line” as a <u>side effect</u> of tasks being executed, that is, when the system is in mission.
  - Ready tasks are sorted in a queue and receive access to the processor and shared resources at run-time using conflict-resolving mechanisms.

Static scheduling

Off-line schedule generation:

- Simulate dynamic scheduling
  - Record a run-time behavior (linear time complexity)
- Apply a search heuristic (e.g., a branch-and-bound algorithm)
  - Find a feasible schedule (if one exists) by considering all possible execution scenarios (NP-complete problem)

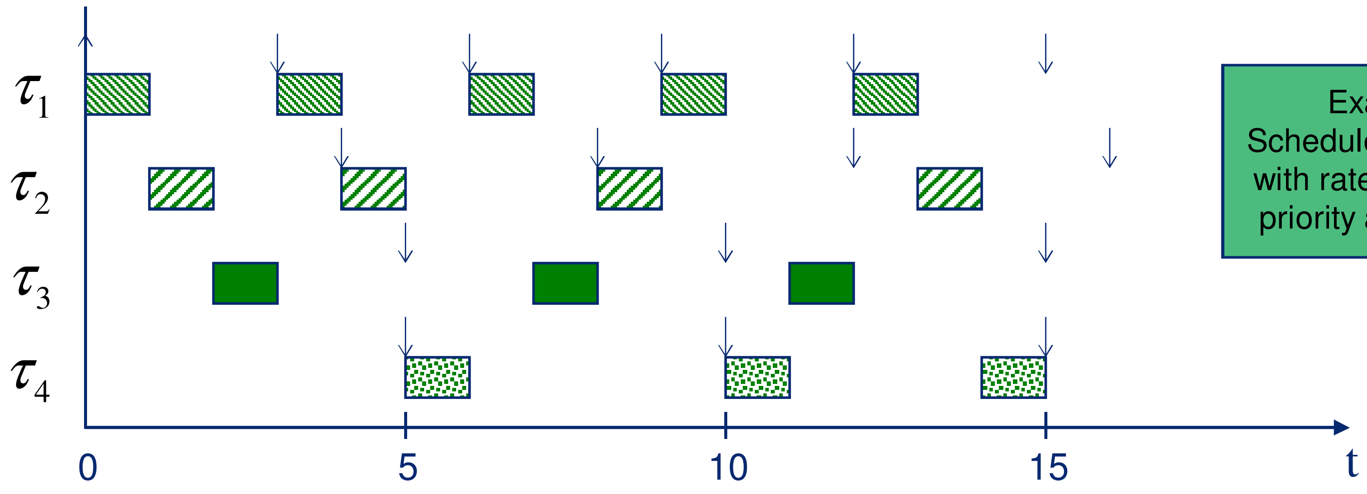


Example:
Cyclic static schedule with a hyper-period (period LCM) of 15.

Dynamic scheduling

On-line schedule generation:

- Mechanisms for resolving conflicts
  - Priorities possibly combined with time quanta
  - Feasibility of schedule must be checked off-line by making <u>predictions</u> on how the conflicts are resolved at run-time



Example:
Schedule generated with rate-monotonic priority assignment

Dynamic scheduling

Rate-monotonic scheduling (RM):

- Uses static priorities
  - Priority is determined by task frequency (rate)
  - Tasks with higher rates (i.e., shorter periods) are assigned higher priorities
- Theoretically well-established (for single-processor systems)
  - Sufficient schedulability test can be performed in linear time (under certain simplifying assumptions)
  - Exact schedulability test is an NP-complete problem
  - RM is optimal among all scheduling algorithms that uses static priorities under the assumption that $D_i = T_i$ for all tasks
    (shown by C. L. Liu & J. W. Layland in 1973)

Dynamic scheduling

Deadline-monotonic scheduling (DM):

- Uses <u>static</u> priorities
  - Priority is determined by task deadline
  - Tasks with shorter (relative) deadlines are assigned higher priorities
  - Note: RM is a special case of DM, with $D_i = T_i$
- Theoretically well-established (for single-processor systems)
  - Exact schedulability test is an NP-complete problem
  - DM is optimal among all scheduling algorithms that uses static priorities under the assumption that $D_i \leq T_i$ for all tasks
    (shown by J. Y.-T. Leung & J. Whitehead in 1982)

Dynamic scheduling

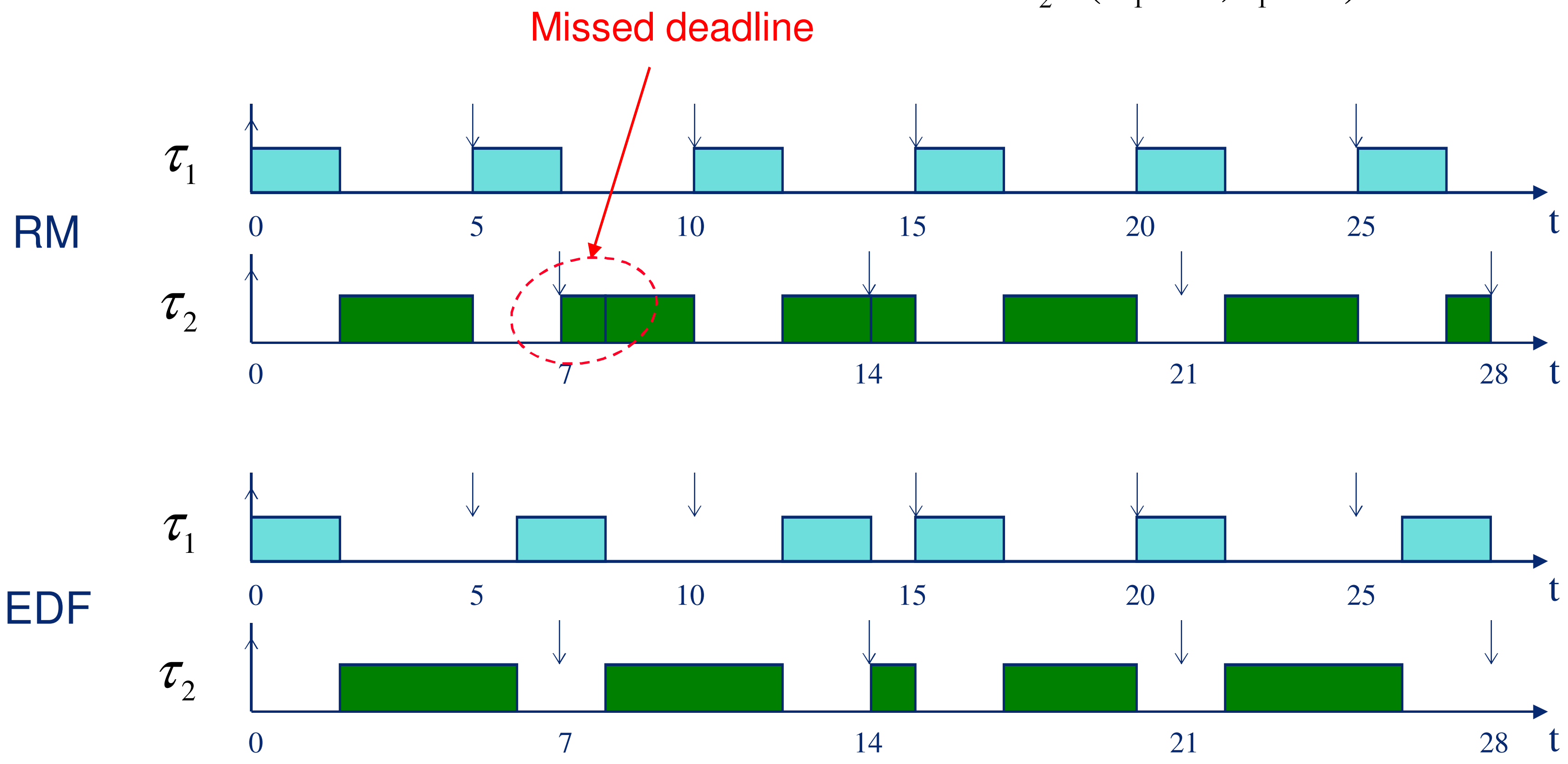
Earliest-deadline-first scheduling (EDF):

- Uses dynamic priorities
  - Priority is determined by how critical the process is at a given time instant
  - The task whose absolute deadline is closest in time receives the highest priority
- Theoretically well-established (for single-processor systems)
  - Exact schedulability test can be performed in linear time (under certain simplifying assumptions)
  - EDF is optimal among all scheduling algorithms that uses dynamic priorities under the assumption that $D_i = T_i$ for all tasks (shown by C. L. Liu & J. W. Layland in 1973)

Dynamic scheduling

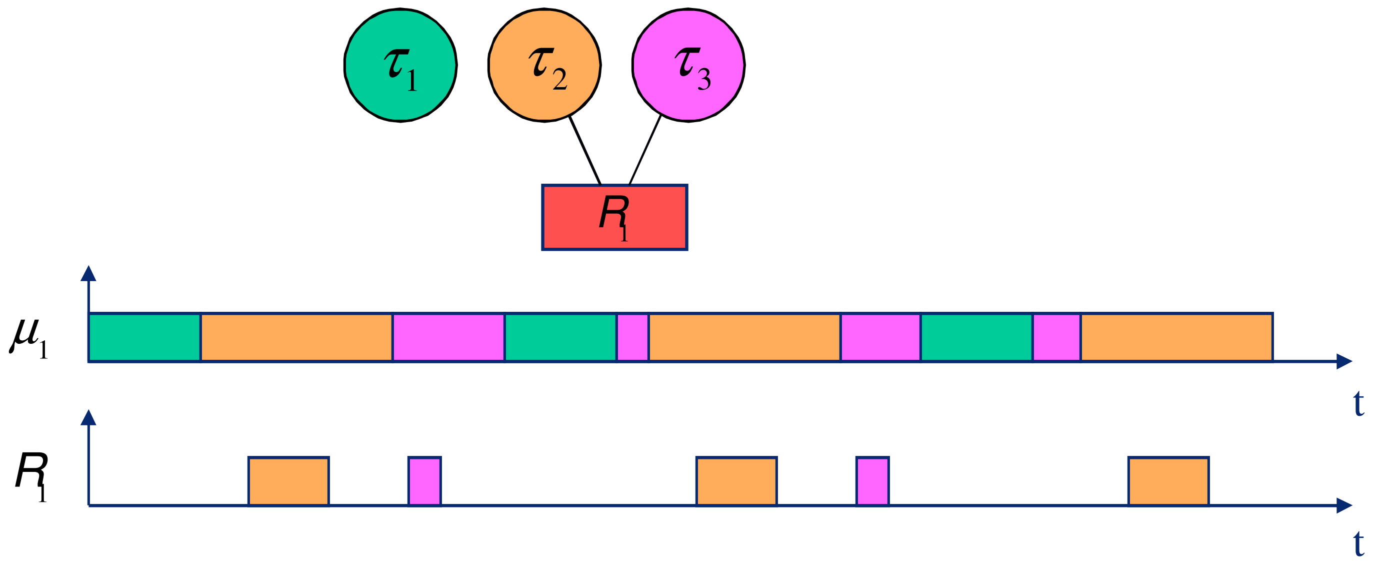
Example: RM versus EDF

$$\tau_1 : (C_1 = 2, T_1 = 5)$$
$$\tau_2 : (C_1 = 4, T_1 = 7)$$



Handling shared resources

When tasks are no longer independent (i.e., they access shared software/hardware objects for which mutual exclusion is enforced) the scheduler must be extended with special mechanisms.



Handling shared resources

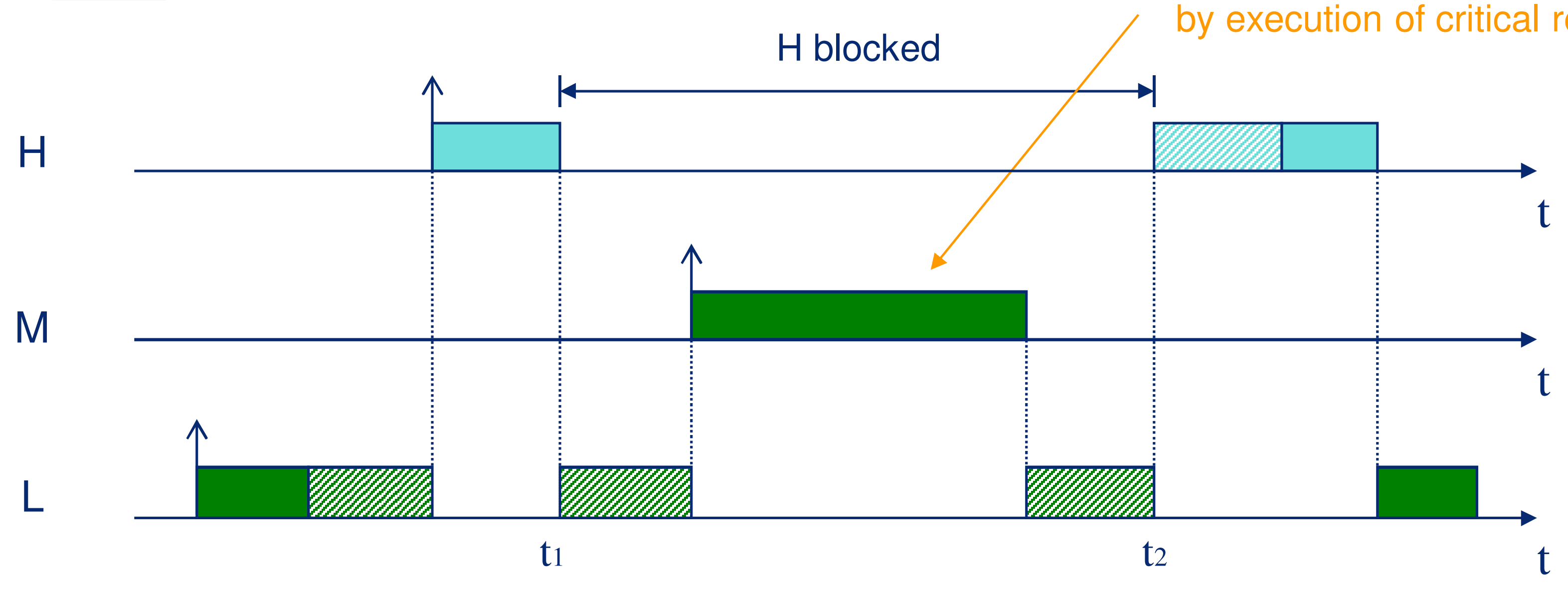
Priority inversion phenomenon:

priority (H) > priority (M) > priority (L)
H and L share mutex resource R

▨ normal execution

▨ critical region

Blocking time for H not bounded by execution of critical region



Handling shared resources

Resolving resource conflicts:
(while also avoiding priority/deadline inversion)

- Off-line resource scheduling:
  - Intelligent algorithms that are configured to generate schedules with no need for conflict resolution at run-time.
    Examples: branch-and-bound (B&B) algorithms
- On-line resource access protocols:
  - Blocking protocols using dynamic adjustments of task priorities.
    Examples: Priority Inheritance Protocol, Deadline Inheritance Protocol, Priority Ceiling Protocol, Immediate Ceiling Priority Protocol, Stack Resource Policy
  - Non-blocking protocols using retry loops.
    Examples: lock-free and wait-free object sharing

Handling shared resources

Priority Inheritance Protocol: (Sha, Rajkumar & Lehoczky, 1990)

- Basic idea: When a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

  Advantage:

  – Prevents medium-priority tasks from preempting τ_i and prolonging the blocking duration experienced by higher-priority tasks.

  Disadvantage:

  – May deadlock: priority inheritance can cause deadlock

  – Chained blocking: the highest-priority task may be blocked once by every other task executing on the same processor.

Handling shared resources

Priority Ceiling Protocol: (Sha, Rajkumar & Lehoczky, 1990)

- Basic idea: Each resource is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it. Then, a task τ_i is allowed to enter a critical region only if its priority is higher than all priority ceilings of the resources currently locked by tasks other than τ_i.
  When the task τ_i blocks one or more higher-priority tasks, it temporarily inherits the highest priority of the blocked tasks.

  Advantage:
  - No deadlock: priority ceilings prevent deadlocks
  - No chained blocking: a task can be blocked at most the duration of one critical region.

Handling shared resources

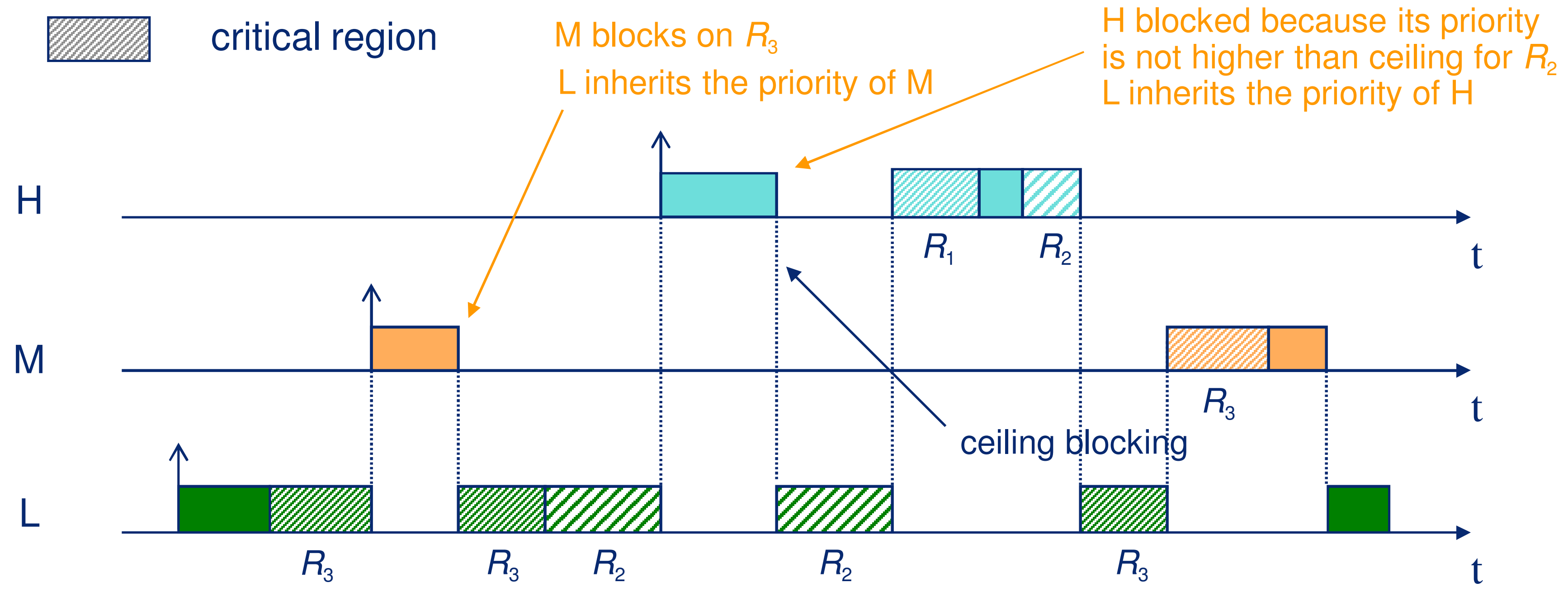
Priority Ceiling Protocol:

priority (H) > priority (M) > priority (L)

H sequentially accesses resources R_1 and R_2
M accesses resource R_3
L accesses resource R_3 and nests R_2

▆ normal execution

▨ critical region


M blocks on R_3
L inherits the priority of M
H blocked because its priority is not higher than ceiling for R_2
L inherits the priority of H
H
M
L
t
t
t
R_1
R_2
R_3
R_3
R_3
R_2
R_2
R_3
ceiling blocking

Handling shared resources

Distributed PCP: (Rajkumar, Sha & Lehoczky, 1988)

- All critical regions associated with the same global resource are bound to a specified synchronization processor.
- A task ”migrates” to the synchronization processor to execute the critical region (using remote-procedure calls)
  - Advantage: deadlock-free algorithm
  - Disadvantage: large overhead for message-passing protocol
- All critical regions associated with the same global resource are executed at a priority equal to the semaphore’s priority ceiling
  - short blocking times

Handling shared resources

Lock-Free and Wait-Free Object Sharing:

If several tasks attempt to access a <u>lock-free</u> object concurrently, and if a subset of these tasks stop taking steps, then <u>one</u> of the remaining tasks completes its access in a finite number of steps.

If several tasks attempt to access a <u>wait-free</u> object concurrently, and if a subset of these tasks stop taking steps, then <u>each</u> of the remaining tasks complete their access in a finite number of steps.

Handling shared resources

Lock-Free Object Sharing: (Anderson et al., 1996)

- Basic idea: The lock-free object sharing scheme is implemented using ”retry loops”. Object accesses are implemented using *test-and-set* or *compare-and-swap* instructions typically found in modern RISC processors.
- Advantage:
  - Resource accesses are non-blocking
  - Deadlock-free
  - Avoids priority inversion
  - Requires no kernel-level support
- Disadvantage:
  - Potentially unbounded retry loops

Handling shared resources

Wait-Free Object Sharing: (Anderson et al., 1997)

- Basic idea: The wait-free object sharing scheme is implemented using a ”helping” strategy where one task ”helps” one or more other tasks to complete an operation.

  Before beginning an operation, a task must announce its intentions in an ”announce variable”.

  While attempting to perform its own operations, a task must also help any previously-announced operation (on its processor) to complete execution.

- Advantage:
  - Non-blocking, deadlock-free, and priority-inversion-free
  - Requires no kernel-level support
  - Precludes waiting dependencies among tasks

Handling shared resources

Non-existence of optimal on-line shared-resource scheduler: (Mok, 1983)

When there are mutual exclusion constraints in a system, it is impossible to find an optimal on-line scheduling algorithm (unless it is clairvoyant).

Complexity of shared-resource feasibility test: (Mok, 1983)

The problem of deciding feasibility for a set of periodic tasks which use semaphores to enforce mutual exclusion is NP-hard.

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

End of lecture #4