CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Dependable Real-Time Systems

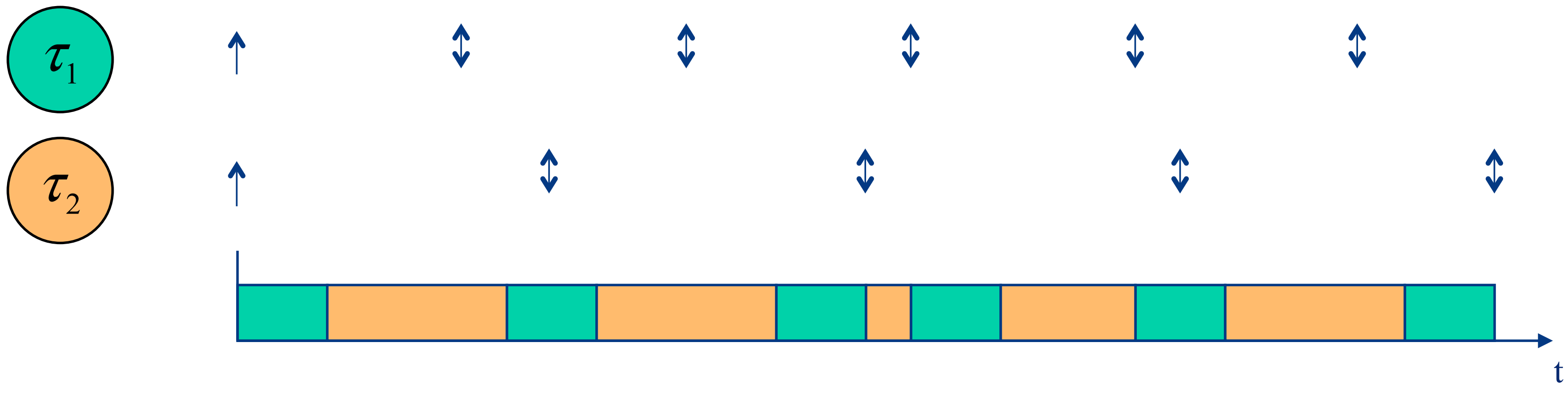
Lecture #3

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Scheduling

Attempts to meet application constraints should be done in a proactive way through <u>scheduling</u>.



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 + courses
  Goal: generate periodic schedules within 7-week blocks
  Technique: branch-and-bound algorithms

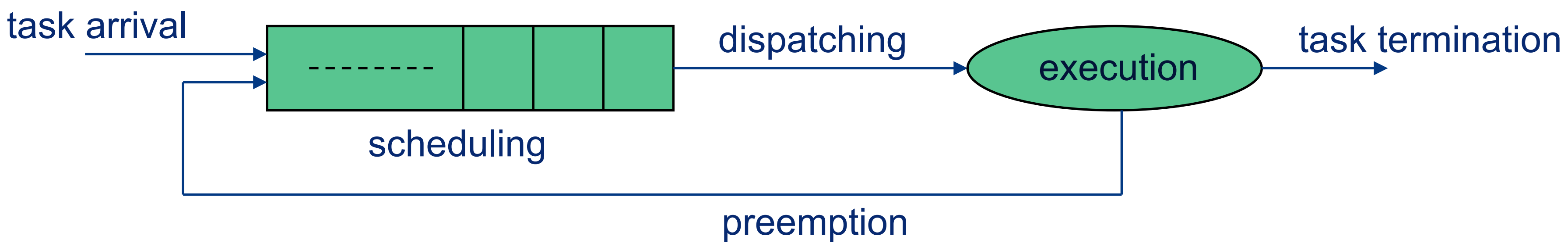
- Airline crew scheduling

  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

Scheduling:

- Implementation:
  - 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 uses a ready queue, where tasks are sorted according to desired execution order.
  - A dispatcher starts the execution of the task in the front of the ready queue, whenever a task switch is possible.



Scheduling

Classification of scheduling constraints:

- Processor-related constraints:
  - How may tasks be executed when multiple processors are available?
- Dispatch-related constraints:
  - What information is known regarding the current and future task set, and how may the dispatcher act based on that information?
- Preemption-related constraints:
  - What other tasks, if any, may preempt the currently-executing task?

Scheduling constraints

Processor-related constraints:

- No processor sharing:
  - A processor can only execute one task at a time
  - Each core in multi-core processor viewed as separate processor
- No dynamic task parallelism:
  - A task can only execute on one processor at a time
  - Realistic assumption for any practical programming model
- No task migration:
  - A task can only execute on one given processor, or cannot change processor once it has started its execution
  - Assumption made for distributed systems, and also for some AUTOSAR multi-core processor designs

Scheduling constraints

Dispatch-related constraints:

- Myopic scheduling:
  - Scheduling algorithm only knows about currently ready tasks
- Clairvoyant scheduling:
  - Scheduling algorithm knows all future arrival times of all tasks
- Work-conserving scheduling:
  - As long as there are tasks in the ready queue the dispatcher must execute a task on a processor
- Non-work-conserving scheduling:
  - Although there are tasks in the ready queue the dispatcher may choose not to execute a task ("inserted idle time")

Scheduling constraints

Preemption-related constraints:

- Fully preemptive scheduling:
  - An executing task can be preempted by other tasks at any time
- Non-preemptive scheduling:
  - Once a task has started executing, it cannot be preempted by any other task
- Greedy scheduling:
  - Once a task has started executing, it cannot be preempted by a lower-priority task
- Fair scheduling:
  - Although a task has started executing, lower-priority tasks receive a guaranteed time quantum per time unit for execution

Preemption constraints

Fully preemptive scheduling:

- Advantages:
  - Gives highest flexibility in making scheduling decisions
- Disadvantages:
  - Guaranteeing mutual exclusion requires special run-time support (e.g., semaphores)
  - Typically incurs higher number of ready queue operations (e.g., insert, remove, dispatch) than for non-preemptive scheduling
  - WCET analysis becomes more complicated since cache and pipeline contents will be affected by a task switch

Preemption constraints

Non-preemptive scheduling:

- Advantages:
  - The only practical approach to implement scheduling of messages on communication networks
  - Guaranteeing mutual exclusion becomes a trivial problem, and can be solved without special run-time support
  - Results from WCET analysis correspond very well with real WCET behavior ("undisturbed execution" assumption)
- Disadvantages:
  - Once a task starts executing, all other tasks on the same processor will be blocked until execution is complete

Preemption constraints

Greedy scheduling:

- Approach: ”traditional” priority scheduling
  - Once a task has started executing, it cannot be preempted by tasks with priorities lower than the currently executing task
  - Note: this is a fundamental assumption in all single- and multi-processor feasibility tests presented so far
- Advantages:
  - Run-time scheduler relatively simple to implement
- Disadvantages:
  - Lower-priority tasks may starve and hence miss their deadlines

Preemption constraints

Fair scheduling:

- Approach: p-fair scheduling (Baruah et al. 1996)
  - Although a task has started executing, lower-priority tasks receive a guaranteed time quantum per time unit for execution
  - Hence: all tasks make some kind of progress per time unit
- Advantages:
  - Multiprocessor schedulability guaranteed for 100% task load (assuming that task-switch cost is negligible)
- Disadvantages:
  - Requires a more advanced run-time scheduler
  - Requires a more advanced approach to feasibility testing
  - Incurs significantly more task switches than greedy scheduling

Preemption constraints (recent results)

Limited preemption scheduling: (see Buttazzo et al. 2013)

- Preemption thresholds:
  - Allows a task to disable preemption by tasks with priorities lower than a specified threshold
  - Special case: “traditional” (greedy) priority scheduling, where the threshold is the priority of the currently executing task
- Deferred preemption:
  - Allows a task to postpone preemption for a given amount of time
  - Special case: non-preemptive scheduling (“postpone until done”)
- Fixed preemption points:
  - Allows a task to specify that preemption can only occur at given places in the program code (a k a “cooperative scheduling”)

Scheduling algorithms revisited

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 algorithms revisited

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 algorithms revisited

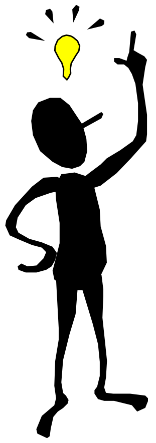
Methods for generating schedules:

- Cyclic executives:
  - Schedule generated ”off-line” before the tasks becomes ready, sometimes even before the system is in mission.
  - Schedule is generated by (i) simulating a pseudo-parallel scheduler or (ii) applying a search algorithm that finds a feasible schedule (whenever one exists) by considering all possible execution scenarios.
- Pseudo-parallel execution:
  - Schedule generated ”on-line” as a side effect of tasks being executed by the run-time system.
  - Resource conflicts at run-time are resolved by using priorities, possible combined with time quanta.

NP-completeness revisited

NP-complete problems:

Problems that are “just as hard” as a large number of other problems that are widely recognized as being difficult by algorithmic experts.



NP-completeness revisited

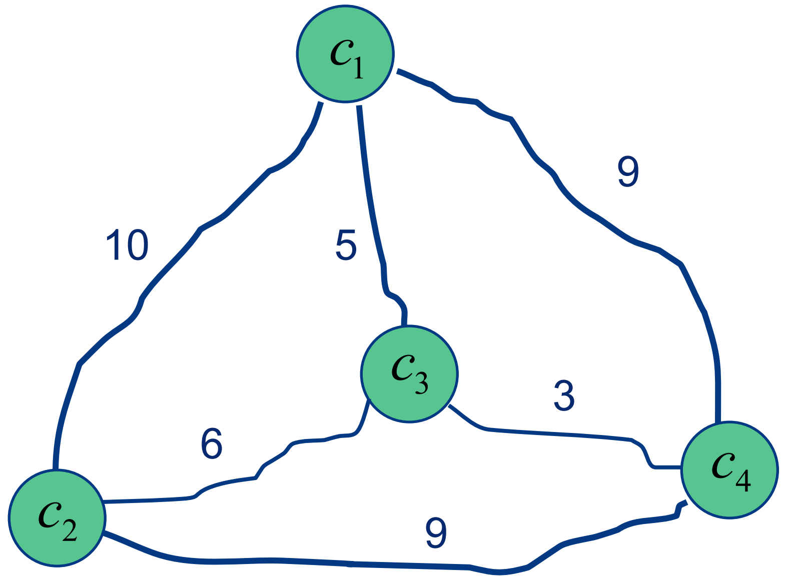
The theory of NP-completeness applies only to decision problems, where the solution is either a “Yes” or a “No”.

↓

If an optimization problem asks for a solution that has minimum “cost”, we can associate with that problem a decision problem that includes a numerical bound B as an additional parameter and that asks whether there exists a solution having cost no more than B.

NP-completeness revisited

Example: The Traveling Salesman Problem:



Is there a "tour" of all the cities in C having a total length of no more than B?

NP-completeness revisited

Deterministic algorithm: (Deterministic Turing Machine)

- Finite-state control:
  - The algorithm can pursue only one computation at a time
  - Given a problem instance I, some solution S is derived by the algorithm
  - The correctness of S is inherent in the algorithm

The class P is the class of all decision problems Π that can be solved by polynomial-time deterministic algorithms.

NP-completeness revisited

Non-deterministic algorithm: (Non-Deterministic Turing Machine)

1. Guessing stage:
   - Given a problem instance I, some solution S is "guessed".
   - The algorithm can pursue an unbounded number of independent computational sequences in parallel.
2. Checking stage:
   - The correctness of S is verified in a normal deterministic manner

The class NP is the class of all decision problems Π that can be solved by polynomial-time non-deterministic algorithms.

NP-completeness revisited

Reducibility:

- A problem Π' is <u>reducible</u> to problem Π if, for any instance of Π', an instance of Π can be constructed in polynomial time such that solving the instance of Π will solve the instance of Π' as well.

When Π' is <u>reducible</u> to Π, we write $\Pi' \propto \Pi$

A <u>decision</u> problem Π is said to be <u>NP-complete</u> if $\Pi \in$ NP and, for all other decision problems $\Pi' \in$ NP, Π' reduces to Π in polynomial time.

Relationship between P and NP

Observations:

1. P $\subseteq$ NP
   - Proof: use a polynomial-time deterministic algorithm as the checking stage and ignore the guess ....
2. P $\neq$ NP
   - This is a wide-spread belief, but …
   - … no proof of this conjecture exists!

The question of whether or not the NP-complete problems are intractable is now considered to be one of the foremost open questions of contemporary mathematics and computer science!

Strong NP-completeness

Pseudo-polynomial time complexity:

- Number problems
  - This is a special type of NP-complete problems for which the largest number (parameter value) in a problem instance is not bounded by the input length (size) of the problem.
- Number problems are often quite tractable
  - If the time complexity of a number problem can be shown to be a polynomial-time function of both the input length and the largest number, that number problem is said to have pseudo-polynomial time complexity.

That is, the time-complexity function is proportional to $p(max,n)$ for some polynomial function p, where max is the largest number and n is the input length.

Strong NP-completeness

If a decision problem Π is NP-complete and is <u>not</u> a number problem, then it cannot be solved by a pseudo-polynomial-time algorithm unless P = NP.

↓

Assuming P ≠ NP, the only NP-complete problems that are potential candidates for being solved by pseudo-polynomial-time algorithms are those that are number problems.

↓

A decision problem Π which cannot be solved by a pseudo-polynomial-time algorithm, unless P = NP, is said to be <u>NP-complete in the strong sense</u>.

Strong NP-completeness

NP-complete problems that are <u>number</u> problems ...

- ... but are NP-complete in the <u>strong</u> sense regardless
  - Multiprocessor scheduling (partitioned and global)
  - Uniprocessor scheduling of asynchronous tasks, or synchronous tasks with dynamic task priorities
  - 3-Partition, Simultaneous Congruences, Traveling Salesman
- ... and that do have pseudo-polynomial time complexity
  - Uniprocessor scheduling of synchronous constrained-deadline tasks with static priorities (using response-time analysis)
  - Uniprocessor scheduling of synchronous constrained-deadline tasks with dynamic task priorities and total utilization U < 1 (using processor-demand analysis)

Co-NP-complete problems

Class co-NP:

- Complement problem:
  - The complement of a decision problem Π is the problem Π^C having the same solution domain as Π, but with the outcome from solving the problem logically reversed.
  - That is, given the same problem instance, a "yes" outcome from solving problem Π would imply a "no" outcome from solving problem Π^C (and vice versa)

A decision problem $\Pi \in$ co-NP if and only if its complement problem $\Pi^C \in$ NP.

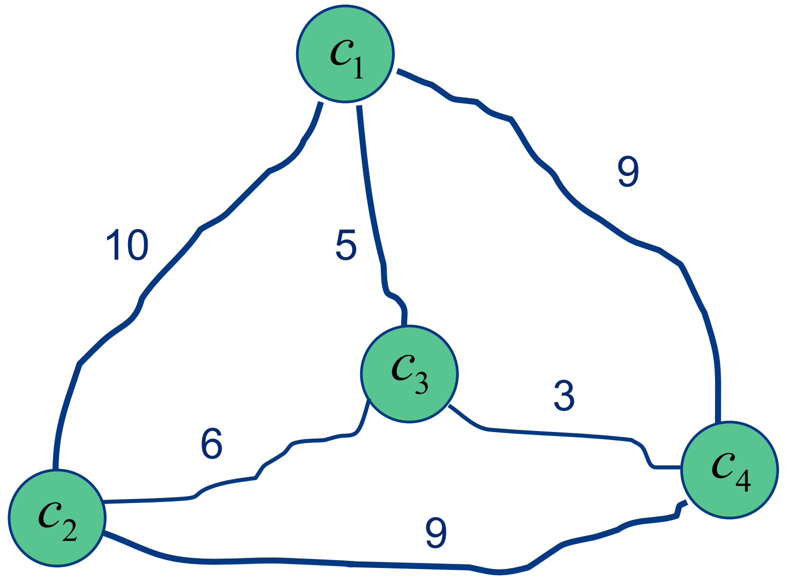
Co-NP-complete problems

NP vs co-NP:

- Problems in NP
  - The class of problems for which there exists a polynomial-time algorithm that can verify a solution that makes the binary problem statement true ("yes" outcome).
- Problems in co-NP
  - The class of problems for which there exists a polynomial-time algorithm that can verify a <u>counterexample</u> solution that makes the binary problem statement false ("no" outcome).
- Co-NP-complete problems
  - Decision problems for which it applies that their complement problem is an NP-complete problem.

Co-NP-complete problems

The Complement Traveling Salesman Problem:



Does every "tour" of all the cities in C have a total length that exceeds B?

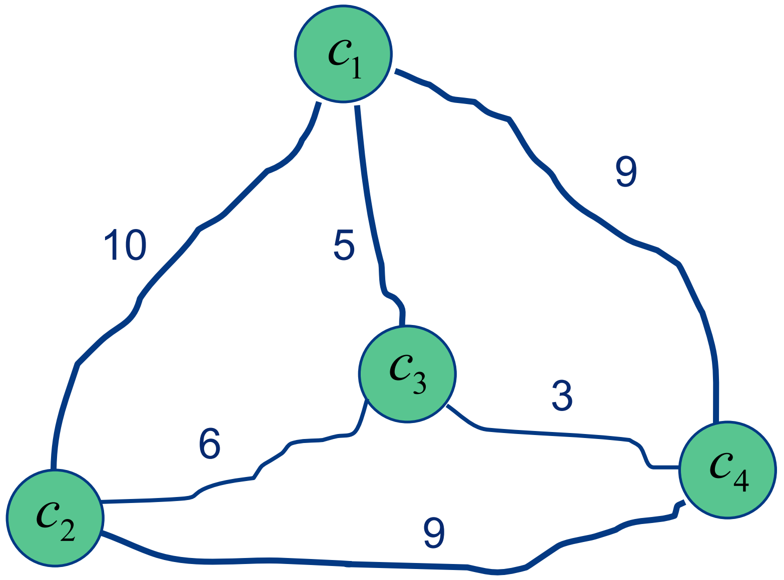
Co-NP-complete problems

The Complement Traveling Salesman Problem:

- Verifying a “yes” outcome
  - Requires checking that all possible solutions (”tours”) to the problem instance fulfills the problem statement. Can in general only be done in exponential time (need to show that every possible “tour” length > B).
- Verifying a “no” outcome
  - Requires checking that one solution (the counterexample “tour”) to the problem instance does not fulfil the problem statement. Can be done in polynomial time (only need to show that the counterexample “tour” length ≤ B).
  - This corresponds exactly to verifying a “yes” outcome in the original Traveling Salesman Problem (which is NP-complete).

Co-NP-complete problems

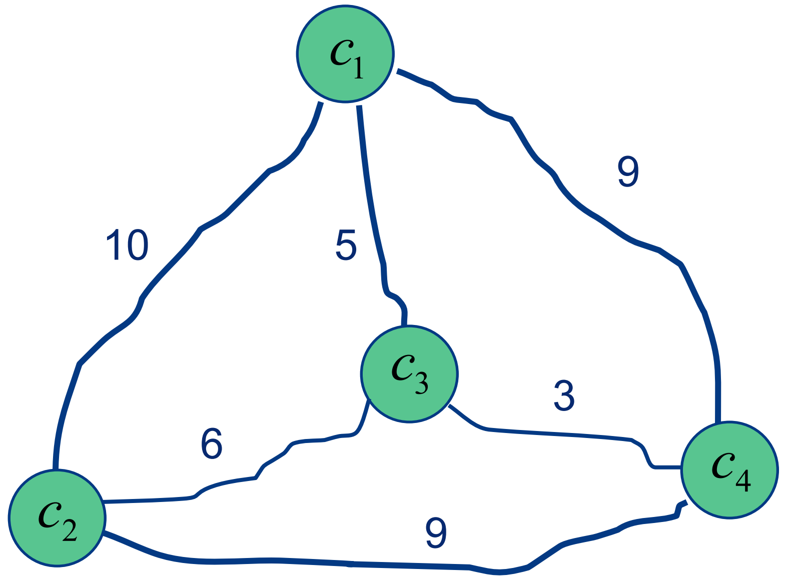
The Complement Traveling Salesman Problem:



Does every "tour" of all the cities in C have a total length that exceeds **30**?

Co-NP-complete problems

The Original Traveling Salesman Problem:



Is there a "tour" of all the cities in C having a total length of no more than **30**?

Proving NP-completeness

Proving NP-completeness for a decision problem Π:

1. Show that Π is in NP
2. Select a known NP-complete problem Π'
3. Construct a transformation $\propto$ from Π' to Π
4. Prove that $\propto$ is a (polynomial) transformation

Highlighted article:
Read the paper by Jeffay, Stanat and Martel (RTSS'91)
Study particularly how the transformation from 3-PARTITION is used for proving strong NP-completeness (Theorem 5.2)