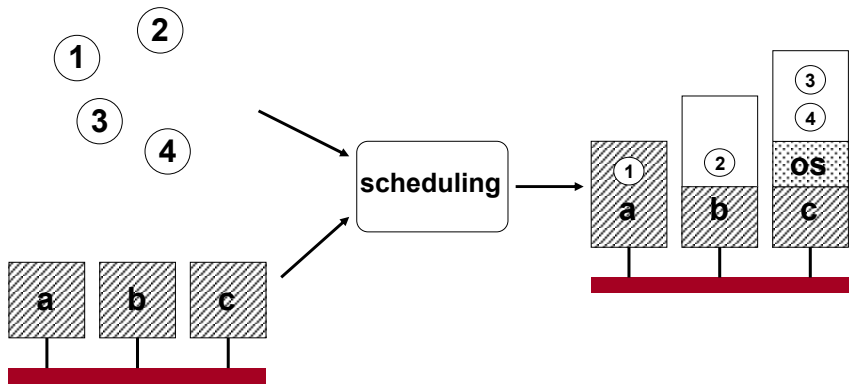
Scheduling of Multi-Task System Specifications

Prof. Jan Madsen

Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, Building 321
DK2800 Lyngby, Denmark
jan@imm.dtu.dk
http://www.imm.dtu.dk/~jan

DTU

Scheduling



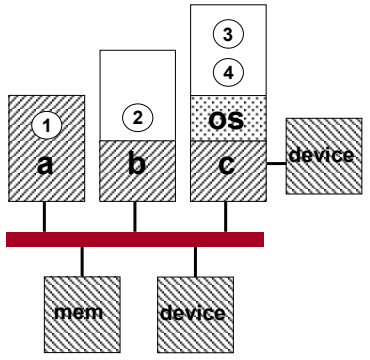
DTU

About this lecture

- Huge literature on scheduling theory
- Aim is to give an overview of problems and possible solutions to the multi-task scheduling problem related to RTOS
- Structure of lecture:
  - Some basic concepts and teminology
  - Uni-processor scheduling
    - Rate-monotonic scheduling
    - Earlies-deadlines-first scheduling
  - Multi-processor scheduling

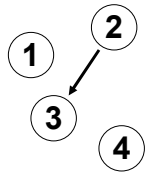
Architecture

- Components
  - Processors
    - CPU, DSP, ASIC, ...
  - Communication
    - Bus, network, ...
  - Devices
    - sensor, actuator, display, ...
- Architecture may be fixed or (re-)configurable

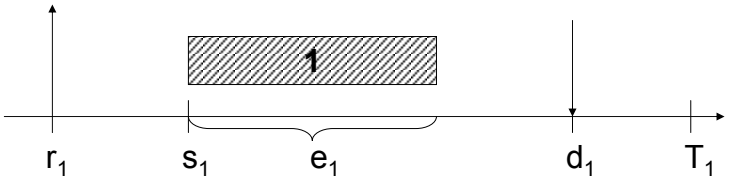


Application

- Task
  - A module which can be envoked to perform a particular function
  - A schedulable entity
- Characterized by its:
  - Timing constraints
  - Precedence contraints
  - Exclusion constraints
  - Resource requirements



Timing constraints



r_1 = time at which task becomes ***released (or active)***

s_1 = time at which task ***starts*** its execution

e_1 = worst case ***execution*** time (WCET)

d_1 = ***deadline***, task should complete before this!

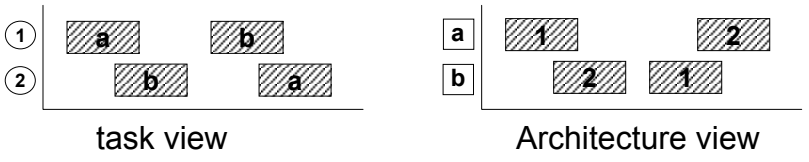
T_1 = ***period***, minimum time between process releases

Task execution

- Periodic
  - the period is the time between successive executions
- Aperiodic
  - non-periodic task, typical an event handler
- Sporadic
  - hard real-time aperiodic task
  - typical, a minimum interarrival constraint is imposed

Scheduling

- Allocation
  - Determine number and type of processors/resources
- Assignment
  - Binding tasks to processors
- Scheduling
  - Determine execution order

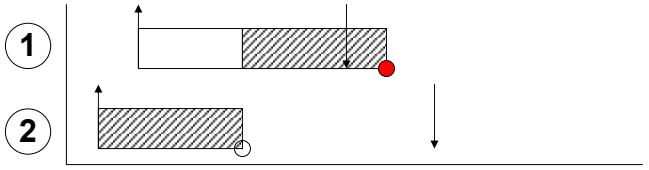


Scheduling principles

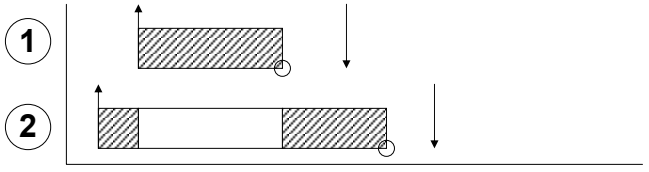
- Off-line
  - A scheduler performing its job before the scheduled system is put into operation
- On-line
  - A scheduler performing its job at run-time, when the system is running
- Typically list-based
  - When a task is released (ready) it is placed in a list
  - Scheduler select which task from the list to execute next
  - Selection based on some criteria – ***scheduling policy***

Preemption vs. Non-preemption

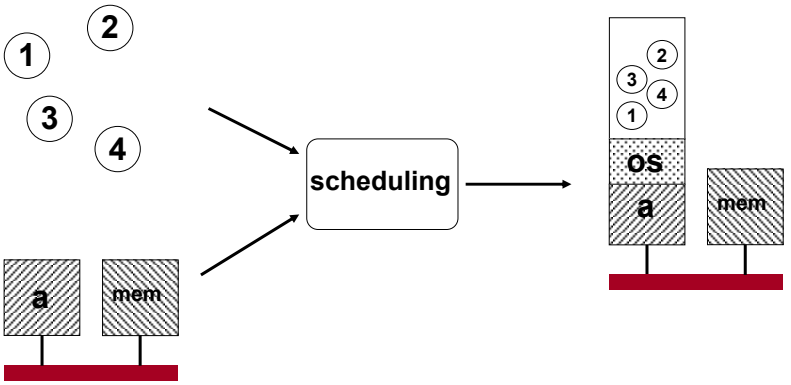
- Preemptive scheduling



- Non-preemptive scheduling



Uni-processor scheduling



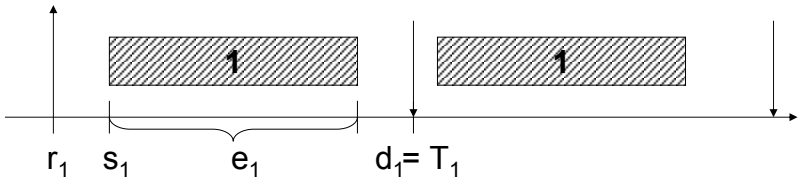
DTU

Rate-monotonic scheduling

- Real-time scheduling with provable properties [Liu and Layland 73]
- Assumptions:
  - Each task is independent and given a unique priority
  - A task may be preempted by a higher-priority process
  - All tasks have a deadline equal to their period and a fixed WCET
  - Context switching overhead is ignored

DTU

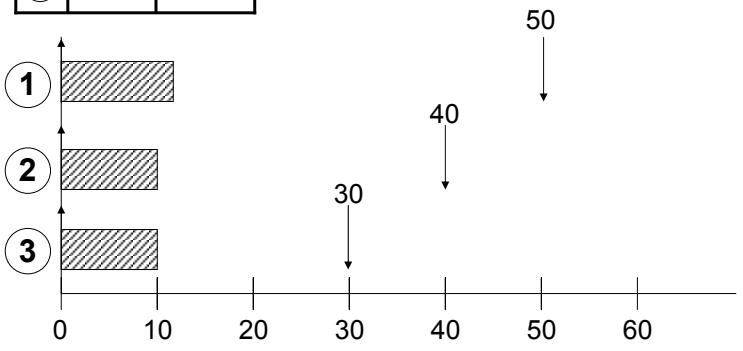
RMS



- The ***critical instance*** for a task occurs when the task and all higher-priority tasks are released simultaneously
- Must guarantee that all tasks meet their deadlines, independent of order of releases

RMS example

	T_i	e_i
1	50	12
2	40	10
3	30	10



Rate-monotonic priority assignment

- Each task is given a unique priority where 1 is the highest
- Priority policy:
  - Task with shortest period gets highest priority
  - Fixed-priority scheme
  - Independent of e_i's
- This priority assignment is optimal

No fixed priority scheme does it better!

RMS example (cont.)

	T_i	e_i	priority
①	50	12	3
②	40	10	2
③	30	10	1

CPU utilization under RMS

- Given N independent tasks, utilization U

$$U = \sum_{i=1}^{N} \frac{e_i}{T_i}$$

- Least upper bound for utilization

$$U_{RM} < N(2^{1/N} - 1)$$

$$\lim_{N \to \infty}(N(2^{1/N} - 1) = \mathbf{69.3}\%$$

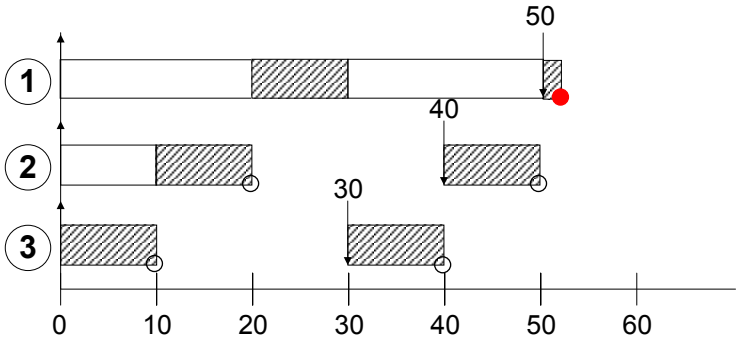
RMS example (cont.)

	T_i	e_i	priority	U
①	50	12	1	0.24
②	40	10	2	0.25
③	30	10	3	0.33
				0.82

$$U_{RM} = 3\,(2^{1/3} - 1) = 0.78$$

$$U \not\leq U_{RM}$$

RMS example (cont.)



RMS

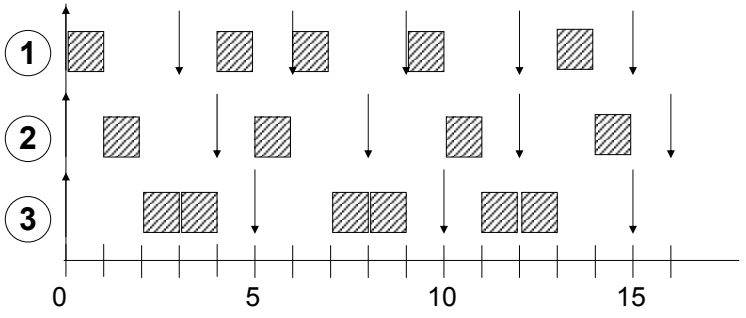
- The utilization-based schedulability test is ***sufficient*** but not ***necessary***
  - If the test is passed it is schedulable
  - If the test is not passed, it may be schedulable
- If all tasks meet their first deadline then they will meet all future ones

Earliest-Deadline-First Scheduling

- Dynamic priority scheme
- Changes priorities during execution based on release times
- Can achieve higher CPU utilization than RMS, i.e. 100%
- Priority policy:
  - task with closest deadline gets highest-priority

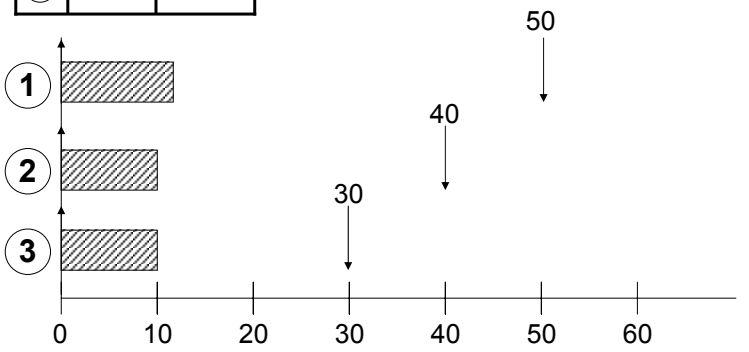
EDF example

	T_i	e_i
1	3	1
2	4	1
3	5	2



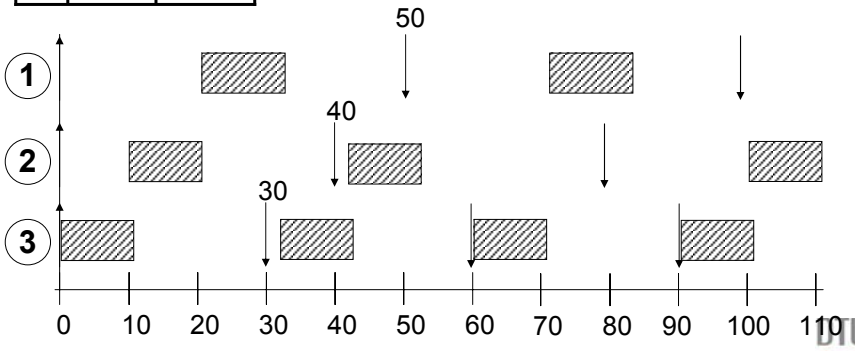
RMS example

	T_i	e_i
1	50	12
2	40	10
3	30	10



RMS example scheduled using EDF

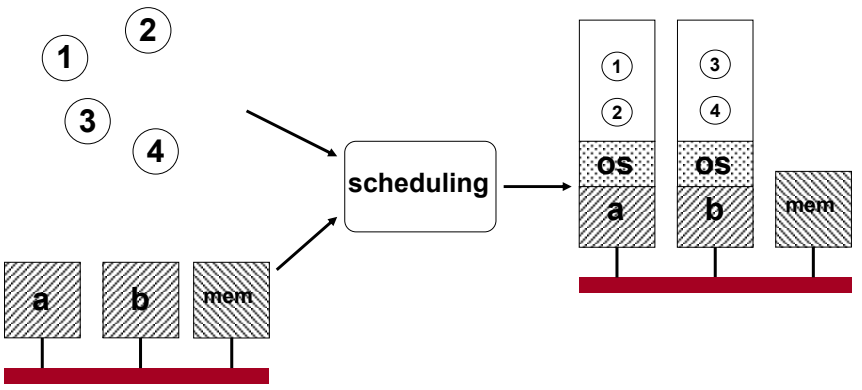
	T_i	e_i
p_1	50	12
p_2	40	10
p_3	30	10



Comparing RMS and EDF

- Higher CPU utilization with EDF
- Easier to ensure that deadlines will be satisfied with RMS
- RMS easier to implement

Multi-processor scheduling



Multi-processor scheduling

- Primary objective is to ensure that all timing constraints are met
- Serious difficulties in validating hard timing constraints
- Assumptions:
  - Each task has its own scheduler
  - The scheduler uses a uni-processor scheduling algorithm
  - Schedulers on different processors need not use the same algorithm

DTU

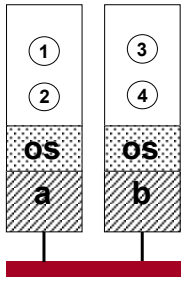
Multi-processor scheduling

- Task assignment
  - Most real-time systems are static, tasks are partitioned and statically bound to processors
- Inter-processor synchronization
  - Ensuring that precedence constraints of tasks on different processors are always satisfied

DTU

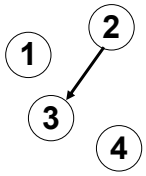
Task assignment

- Often done off-line
- NP-hard problem
- Assignment based on:
  - Execution times
  - Resource requirements
  - Data dependencies
  - Timing constraints
  - Communication cost
  - ...



Task assignment - RMFF

- Rate-Monotonic First-Fit algorithm:
  1. Sort tasks in increasing order according to their period
  2. Assign tasks one by one
     - (1) is assigned to [a]
     - (i) is assigned to [k] if the total utilization of (i) and the tasks already assigned to [k] $\leqslant U_{RM}$
- Example:

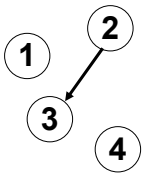


Assume zero communication overhead

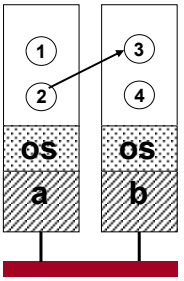
Scheduling based on fixed priorities

A synchronization signal makes sure that (3) is released as soon as (2) has completed

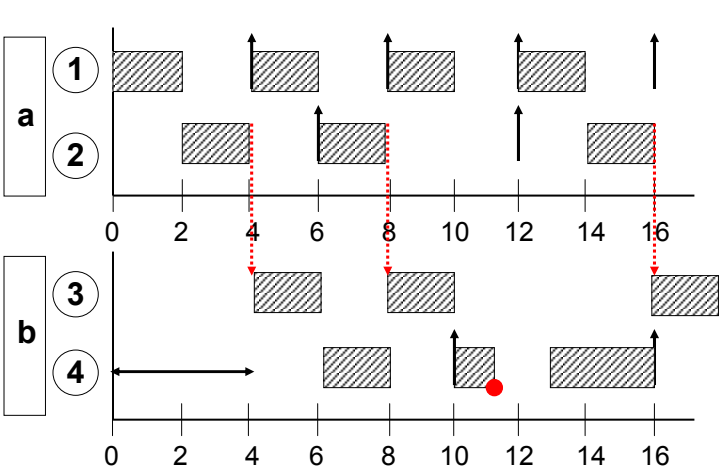
Example

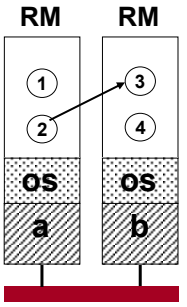


	r_i	T_i	e_i
1	0	4	2
2 → 3	0	6	2+2
4	4	6	3

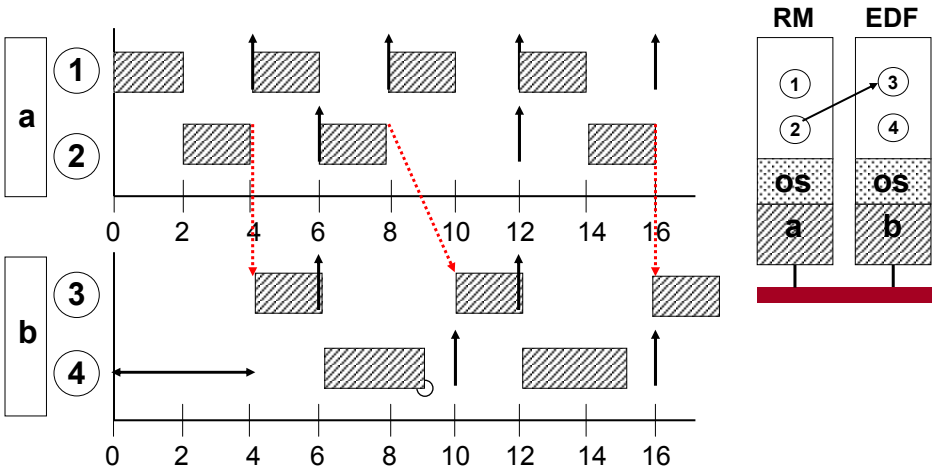


Dynamic scheduling





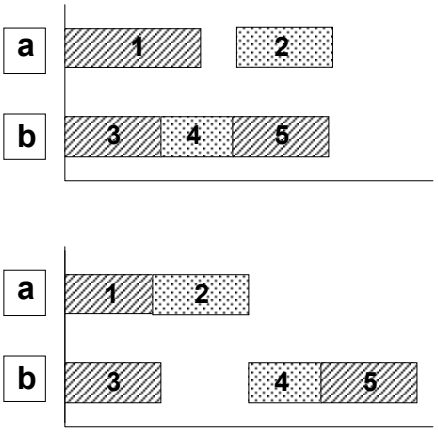
Changing synchronization protocol



Multi-processing anomalies

- Assume a set of tasks optimally scheduled on a multiprocessor system with:
  - fixed number of processors
  - fixed execution times (e_i)
  - precedence constraints
- Then
  - ***changing*** the priority list
  - ***increasing*** the number of processor
  - ***reducing*** execution times
  - ***weakening*** the precedence constraints
- May ***increase*** the scheduling length!

Example of anomalies



Task 2 and 4 are sharing a resource, i.e. mutually exclusion

Reduce e_1 of task 1

Consequences of anomalies

- Tasks may complete before their WCETs
- So most on-line scheduling algorithms are subject to experience anomalies
- Simple but inefficient solution:
  - Have tasks completing early idle

Summary

- Introduction to classical uni-processor multi-task scheduling
- Illustration of multi-processor scheduling
- Not covered:
  - Priority inversion when having shared resources
  - Context switching and cache overhead
  - Overload conditions
  - Low power scheduling,
    - increasing idle periods on processors
    - Voltage scaling
  - Communication scheduling

DTU