

Resources and tasks management in ES

(up to/from here: session 19/20)

Amir Mahdi Hosseini Monazzah

Room 332,
School of Computer Engineering,
Iran University of Science and Technology,
Tehran, Iran.

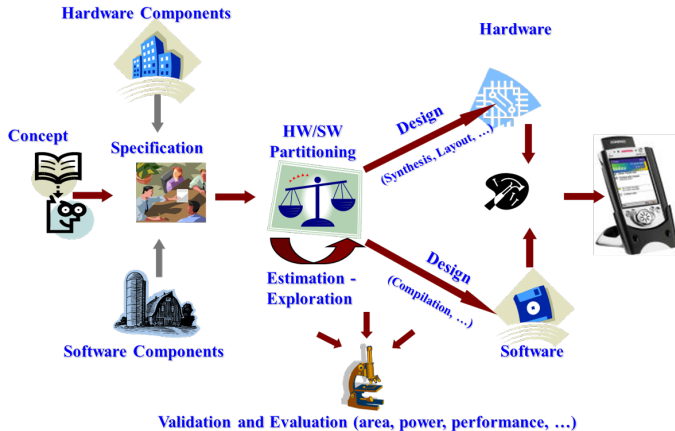
monazzah@iust.ac.ir

Fall 2024

Outline

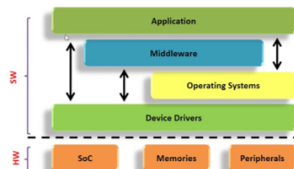
- Preliminaries on scheduling
 - A deeper view on ES software
 - Definitions
 - Classifications of scheduling algorithms
- Real-time scheduling
 - Definitions
 - Different types of real-time scheduling algorithms
 - Uniprocessor real-time scheduling
 - Equal arrival times algorithms
 - Different arrival times
 - Multi-processor real-time scheduling

Embedded system design flow



Reuse of standard software components

- Knowledge from previous designs to be made available in the form of intellectual property (IP, for SW and HW).
 - Operating systems (real-time)
 - Middleware
 - Real-time data bases
 - Standard software (MPEG-x, GSM-kernel, ...)
- Includes standard approaches for scheduling (requires knowledge about execution times)



Time-triggered systems (up to/from here: session 21/22)

- Entirely time-triggered system
 - The temporal control structure of all tasks is established a priori by off-line support-tools
 - Temporal control structure is encoded in a Task-Descriptor List (TDL) that contains the cyclic schedule for all activities of the node
 - This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an **explicit coordination** of the tasks by the operating system at run time is **not necessary**

Time-triggered systems

- The dispatcher is activated by the synchronized clock tick
 - It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz]
 - The only practical means of providing predictability
- It can be easily checked if timing constraints are met
 - Response to sporadic events may be poor

Worst case execution time

- **Definition:** The worst case execution time (WCET) is an upper bound on the execution times of tasks
 - Computing such a bound is undecidable
 - Pipeline hazards, interrupts, caches → serious overestimates
- **Approaches:**
 - For hardware: Typically requires hardware synthesis
 - For software: Requires availability of machine programs; complex analysis

Average execution time

- Estimated cost and performance values
 - Difficult to generate sufficiently precise estimates; Balance between run-time and precision
- Accurate cost and performance values
 - Can be done with normal tools (such as compilers)
 - As precise as the input data is

Which timing metric is more important in real-time ES?

Schedulability

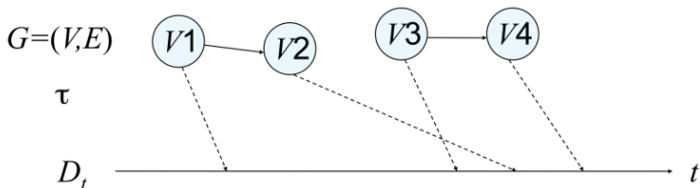
- A set of tasks is said to be schedulable under a given set of constraints, if a schedule exists for that set of tasks and constraints
 - Exact tests are NP-hard in many situations
 - Sufficient tests: Sufficient conditions for guaranteeing a schedule are checked
 - Necessary tests: Checking necessary conditions. Can be used to show that no schedule exists
 - Always not possible to prove even if no schedule exists

Scheduling classifications

- Centralized and distributed scheduling
 - Multiprocessor scheduling either locally on one or distributed on several processors
- Mono- and multi-processor scheduling
 - Simple scheduling algorithms handle single processors, more complex algorithms handle multiple processors
- Online (dynamic) - and offline (static) scheduling:
 - Online scheduling is done at run-time based on the information about the tasks arrived so far
 - Offline scheduling assumes prior knowledge about arrival times, execution times, and deadlines

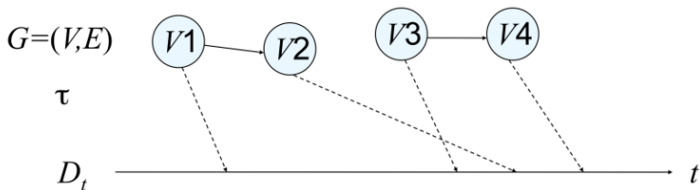
What is real-time scheduling?

- Assume that we have a task graph $G=(V,E)$
- A schedule of G is a mapping, $V \rightarrow T$, of a set of tasks V to start times from domain T



What is real-time scheduling?

- Schedules have to respect a set of constraints, such as resource, dependency, and **deadlines**
- Scheduling is the process of finding such a mapping
- During the design of embedded systems, scheduling has to be performed several times
 - Early rough scheduling as well as late precise scheduling



Simple tasks

- Tasks without any inter-process communication are called simple tasks (S-tasks)
 - S-tasks can be either **ready** or **running**
- API of an S-task in a TT system: Two OS calls
 - TERMINATE TASK and ERROR
 - The TERMINATE TASK system call is executed whenever the task has reached its termination point
 - In case of an error that cannot be handled within the application task, the task terminates its operation with the ERROR system call [Kopetz, 1997]

Cost functions

- Cost function: Different algorithms aim at minimizing different functions
 - For example **Maximum lateness** is one of the main metrics that are used in cost functions
- **Definition:** Maximum lateness is defined as the difference between the completion time and the deadline, maximized over all tasks
 - Maximum lateness is **negative** if all tasks complete before their deadline

Basic parameters in real-time scheduling

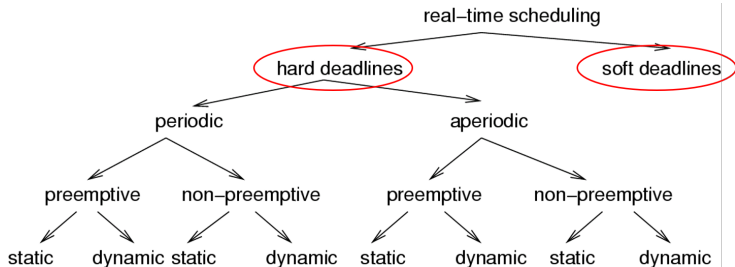
- Arrival time a_i
 - Time when a task becomes ready for execution
- Computation time c_i
 - Time necessary for completion of a task
- Absolute deadline d_i
 - Time before which a task should be finished
- Finishing time f_i
 - Time at which a task finishes its execution

Basic parameters in real-time scheduling

- Derived Parameters

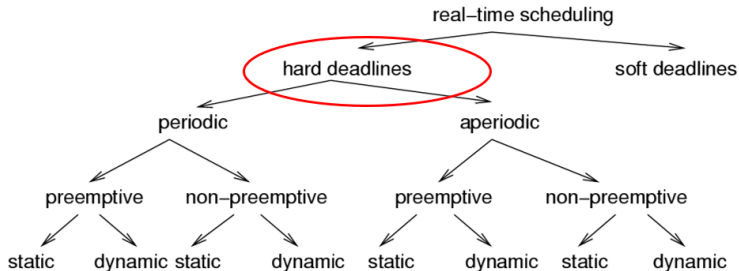
- Relative deadline $D_i = d_i - a_i$
- Response time $R_i = f_i - a_i$
- Lateness $L_i = f_i - d_i$ delay of a task (can be negative)
- Slack time (laxity) $s_i = d_i - a_i - c_i$
 - Maximum time a task can be delayed on its activation to complete within deadline

Classification of real-time scheduling algorithms

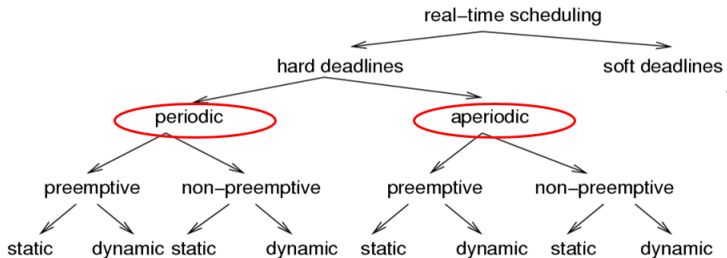


Hard and soft deadlines

- **Definition:** A time-constraint (deadline) is called hard if not meeting that constraint could result in a catastrophe [Kopetz, 1997]
 - All other time constraints are called soft
- We will focus on hard deadlines!

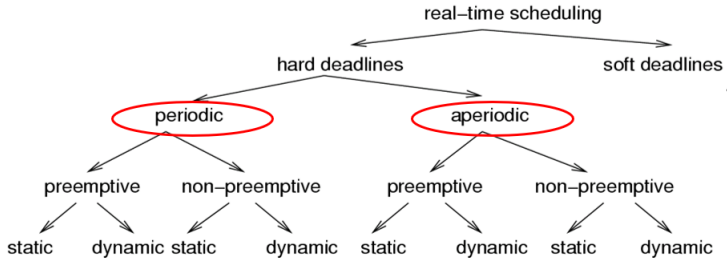


Periodic and aperiodic tasks



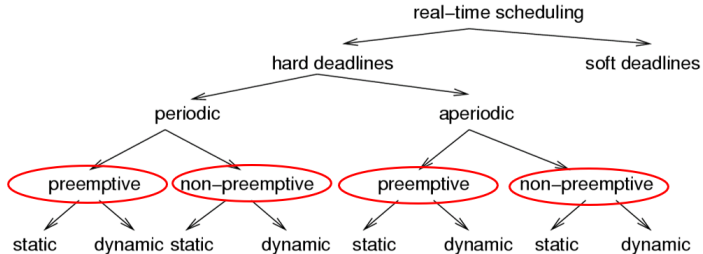
- **Definition:** Tasks which must be executed once every p units of time are called **periodic tasks**
 - p is called their period
 - Each execution of a periodic task is called a job
 - All other tasks are called aperiodic

Periodic and aperiodic tasks



- **Definition:** Aperiodic tasks requesting the processor at unpredictable times are called sporadic if there is a minimum separation between the times at which they request the processor

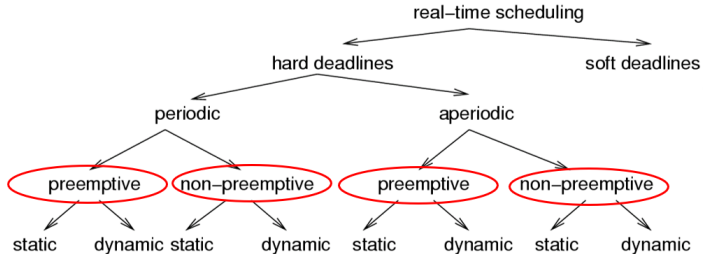
Preemptive and non-preemptive scheduling



- **Definition: Preemptive and non-preemptive scheduling**

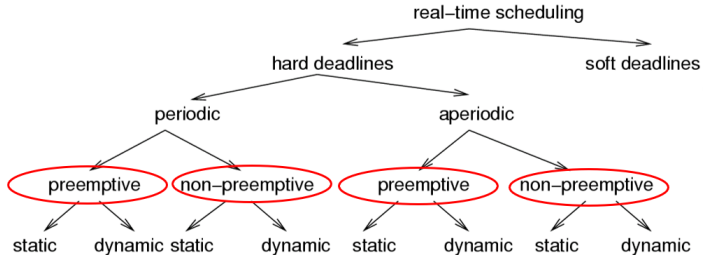
- Non-preemptive schedulers are based on the assumption that tasks are executed until they are done

Preemptive and non-preemptive scheduling



- For non-preemptive scheduler
 - The response time for external events may be quite long if some tasks have a large execution time

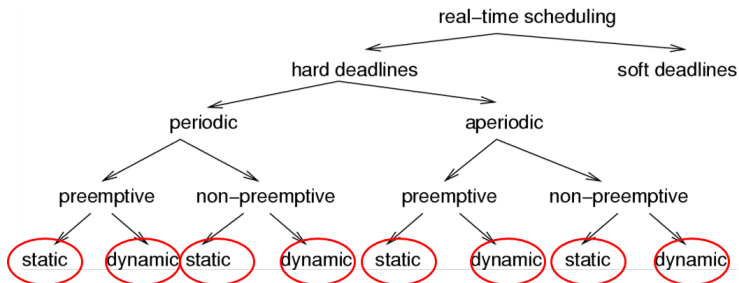
Preemptive and non-preemptive scheduling



- Preemptive schedulers have to be used if some tasks have long execution times or if the response time for external events is required to be short

Static and dynamic scheduling

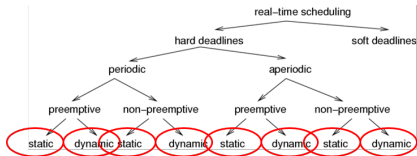
- **Definition: Dynamic scheduling**
 - **Processor allocation decisions (scheduling) done at run-time**




Static and dynamic scheduling

• Definition: Static scheduling

- Processor allocation decisions (scheduling) done at design-time
 - Dispatcher allocates processor when interrupted by a timer
 - The timer is controlled by a table generated at design time

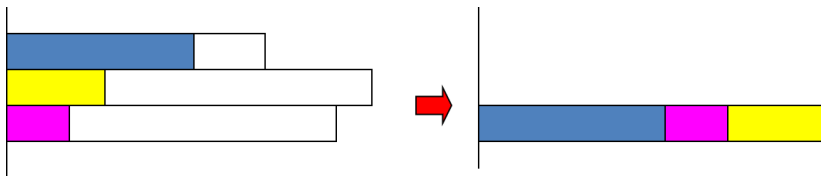


Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	

→ 

Earliest Due Date (EDD)

- Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness
 - Proof: See Dertouzos, M. L.: "Control Robotics: the Procedural Control of Physical Processes", Information Processing 74, North-Holland Publishing Company, 1974
- EDD requires all tasks to be sorted by their deadlines



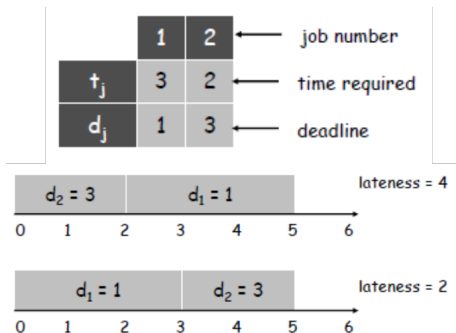
In-class assignment: Question

- Consider the following job's information table
 - Draw all possible scheduling Gantt chart
 - Which one is better in terms of minimizing maximum lateness?

	1	2	← job number
t_j	3	2	← time required
d_j	1	3	← deadline

In-class assignment: Answer

- Consider the following job's information table
 - Draw all possible scheduling Gantt chart
 - Which one is better in terms of minimizing maximum lateness?



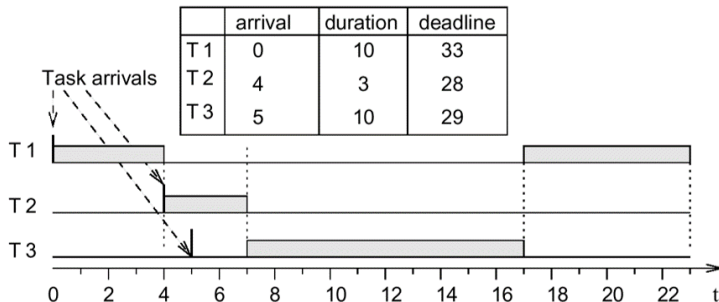
Earliest Deadline First (EDF)

- Theorem [Horn74]: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the **earliest absolute deadline** among all the ready tasks is optimal with respect to minimizing the maximum lateness

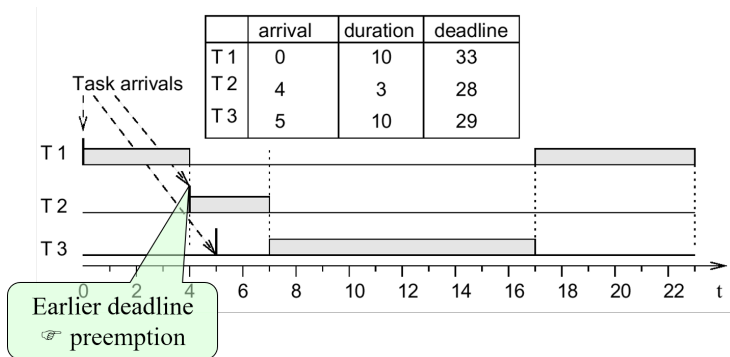
Earliest Deadline First (EDF)

- Each time a new ready task arrives, it is inserted into a queue of ready tasks, sorted by their deadlines
 - If a newly arrived task is inserted at the head of the queue, the currently executing task is preempted
 - Preemption potentially reduces lateness

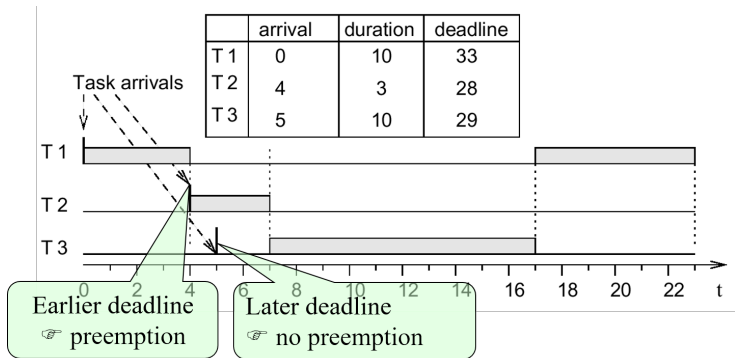
Earliest Deadline First (EDF)



Earliest Deadline First (EDF)



Earliest Deadline First (EDF)

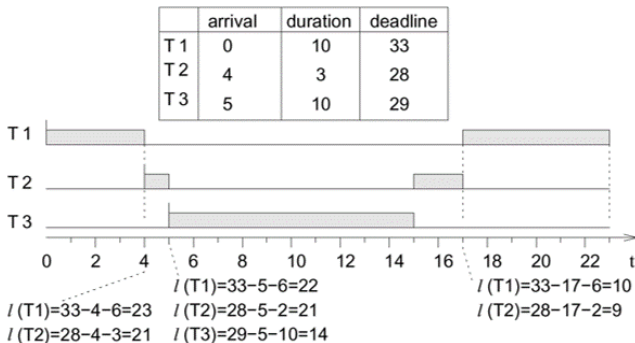


Scheduling with no precedence constraints (up to/from here: session 22/23)

- Let T_i be a set of tasks and
 - c_i be the execution time of T_i
 - d_i be the **deadline interval**, that is, the time between T_i becoming available and the time until which T_i has to finish execution
 - l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$

Least Laxity First (LLF) or Least Slack Time First (LSF)

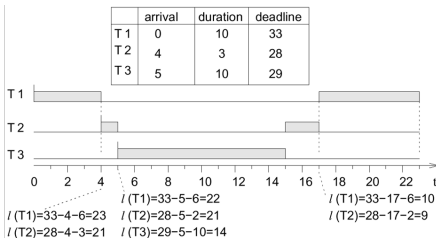
- Priorities = Decreasing function of the laxity
 - The less laxity, the higher the priority
 - Dynamically changing priority
 - Preemptive



Least Laxity First (LLF) or Least Slack Time First (LSF)

- Priorities = Decreasing function of the laxity
 - The less laxity, the higher the priority
 - Dynamically changing priority
 - Preemptive

- ✓ Requires calling the scheduler periodically, and to recompute the laxity. Overhead for many calls of the scheduler and many context switches.
- ✓ Detects missed deadlines early



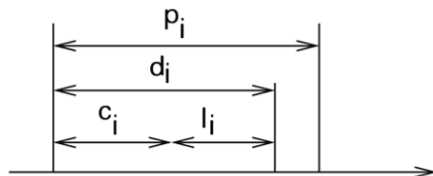
LLF/LSF properties

- LLF/LSF is also an optimal scheduling for uni-processor systems
 - BUT... uses dynamic priorities
 - Therefore cannot be used with a fixed priority OS
 - Fixed-priority preemptive scheduling is a scheduling system commonly used in real-time systems
- LLF/LSF scheduling requires the knowledge of the execution time
 - May not know this in advance!

Periodic scheduling

- Let

- p_i be the period of task T_i ,
- c_i be the execution time of T_i ,
- d_i be the deadline interval, that is, the time between a job of T_i becoming available and the time after which the same job T_i has to finish execution
- l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$



Periodic scheduling

- Accumulated utilization
 - Accumulated execution time divided by period

$$\mu = \sum_{i=1}^n \frac{c_i}{p_i}$$

Necessary condition for schedulability
(with m =number of processors):

$$\mu \leq m$$

Rate Monotonic (RM)

- Well-known technique for scheduling independent periodic tasks [Liu, 1973]
- Assumptions:
 - All tasks that have hard deadlines are periodic
 - All tasks are independent
 - $d_i = p_i$, for all tasks
 - c_i is constant and is known for all tasks
 - The time required for context switching is negligible

Rate Monotonic (RM)

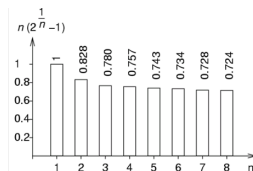
- RM schedulability condition
 - For a single processor with n tasks, the following equation must hold for the accumulated utilization μ

$$\mu = \sum_{i=1}^n \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$$

Rate Monotonic (RM)

- The priority of a task is a monotonically decreasing function of its period
 - Low period: High priority
- At any time, a highest priority task among all those that are ready for execution is allocated
- If all assumptions are met, schedulability is guaranteed

Maximum utilization as a function of the number of tasks \Rightarrow



Example: RM-generated schedule

- T1 preempts T2 and T3
- T2 and T3 do not preempt each other



Case of failing RM scheduling

Task 1: period 5, execution time 2

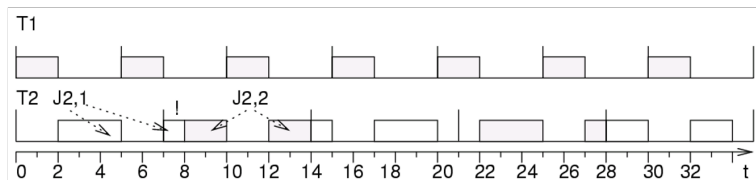
Task 2: period 7, execution time 4

$$\mu = 2/5 + 4/7 = 34/35 \approx 0.97$$

$$2(2^{1/2} - 1) \approx 0.828$$

$$\mu = \sum_{i=1}^n \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$$

Not enough idle time



Case of failing RM scheduling

Task 1: period 5, execution time 2

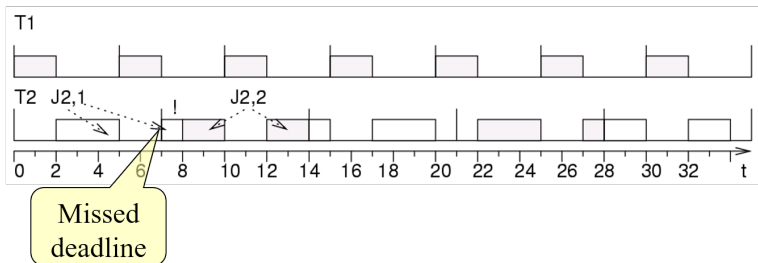
Task 2: period 7, execution time 4

$$\mu = 2/5 + 4/7 = 34/35 \approx 0.97$$

$$2(2^{1/2} - 1) \approx 0.828$$

$$\mu = \sum_{i=1}^n \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$$

Not enough idle time



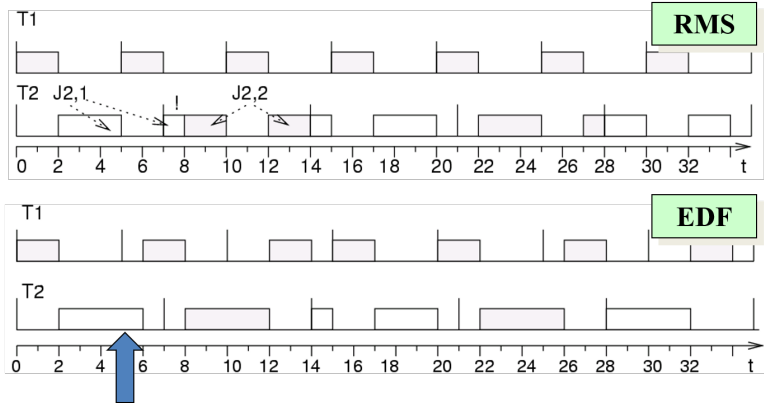
Properties of RM scheduling

- RM scheduling is based on static priorities
 - This allows RM scheduling to be used in standard OS
 - Such as Windows NT
 - A huge number of variations of RM scheduling exists
- In the context of RM scheduling, many formal proofs exist
- The **idle capacity** is not required if the period of all tasks is a multiple of the period of the highest priority task
 - Necessary condition for schedulability: $\mu \leq 1$

EDF in periodic scheduling

- EDF can also be applied to periodic scheduling
- EDF optimal for every period
 - Optimal for periodic scheduling
 - Trivially!
 - EDF must be able to schedule the example in which RMS failed
- EDF requires dynamic priorities
 - EDF cannot be used with a standard operating system just providing static priorities

Comparison of EDF and RMS



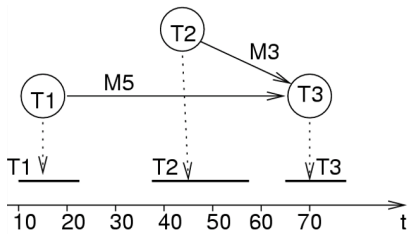
T2 not preempted, due to its earlier deadline

Scheduling without preemption

- Optimal schedules may leave processor idle to finish tasks with early deadlines arriving late
 - Knowledge about the future is needed for optimal scheduling algorithms
 - No online algorithm can decide whether or not to keep idle
- EDF is optimal among all scheduling algorithms not keeping the processor idle at certain times

Scheduling with precedence constraints

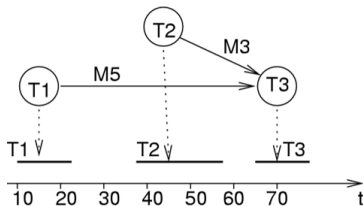
- Task graph and possible schedule



Schedule can be stored in a table
(can be used by dispatcher/OS)

Synchronous arrival times

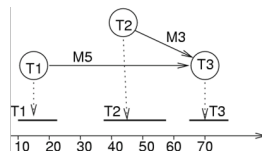
- Optimal algorithm for minimum latency
 - Latest Deadline First (LDF)
- LDF [Lawler, 1973]: Generation of total order compatible with the partial order described by the task graph (LDF performs a **topological sort**)



Synchronous arrival times

- LDF reads the task graph from tail to head and inserts tasks with no successors into a queue. It then repeats this process, putting tasks whose successor have all been selected into the queue
 - At run-time, the tasks are executed in from head to tail
 - LDF is non-preemptive and is optimal for uni-processors

Latest Deadline First (LDF)



Asynchronous arrival times

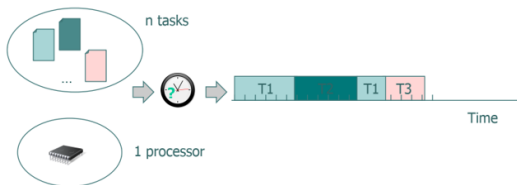
- This case can be handled with a modified EDF algorithm
 - The key idea is to transform the problem from a given set of dependent tasks into a set of independent tasks with different timing parameters [Chetto90]
- This algorithm is optimal for uni-processor systems
- If preemption is not allowed, the heuristic algorithm developed by [Stankovic and Ramamritham 1991] can be used

Sporadic tasks

- If sporadic tasks were connected to interrupts, the execution time of other tasks would become very unpredictable
 - Introduction of a sporadic task server, periodically checking for ready sporadic tasks
 - Sporadic tasks are essentially turned into periodic tasks

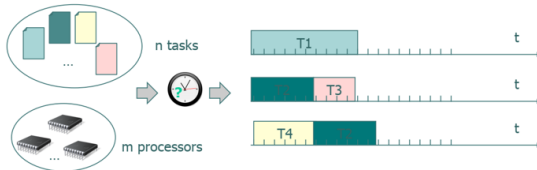
Introduction

- Mono-processor scheduling: One-dimension problem
 - **Temporal** organization



Introduction

- Multi-processor (multi-core) scheduling: Two dimension problem
 - **Temporal** organization +
 - **Spatial** organization
 - On which processor execute every task?

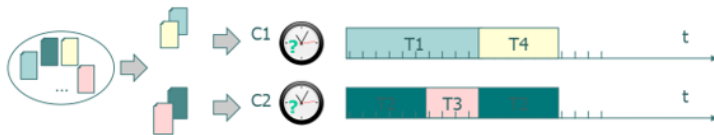


Classification

- **Partitioned** scheduling
 - Each of the two dimensions is dealt with separately
- **Global** scheduling
 - Temporal and spatial dimensions are deal with jointly
- **Semi-partitioned** scheduling
 - Hybrid

Classification: partitioned scheduling

- Each of the two dimensions is dealt with separately
 - Spatial organization: the n tasks are partitioned onto the m cores
 - No task migration at run-time
 - Temporal organization: Mono-processor scheduling is used on each core



Classification: partitioned scheduling

- Two points of view
 - Number of processors to be determined: Optimization problem (bin-packing problem)
 - Bin = task, size = utilization (or other expression obtained from the task temporal parameters)
 - Boxes = processors, size = ability to host tasks
 - Fixed number of processors: search problem (knapsack problem)
- Both problems are NP-hard

Classification: partitioned scheduling

- Optimal mono-processor scheduling strategies: XX
 - RM, DM (Deadline Monotonic: Like rate monotonic but consider **task's deadlines** instead of task's periods)
 - EDF, LLF (see uni-processor scheduling section)
- Bin-packing heuristics: YY
 - FF: First-Fit
 - BF: Best-Fit
 - WF: Worst-Fit
 - NF: Next-Fit
 - FFD, BFD, WFD:
First/Best/Worst-Fit Decreasing

Partitioning algorithms
XX-YY

Classification: partitioned scheduling

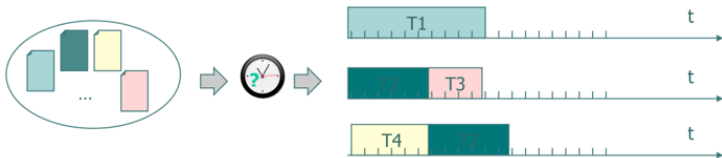
- Benefits
 - Implementation: local schedulers are independent
 - No migration costs
 - Direct reuse of mono-processor schedulability tests
 - Isolation between processors in case of overload
- Limits

Classification: partitioned scheduling

- Benefits
- Limits
 - Rigid: suited to static configurations
 - NP-hard task partitioning
 - Largest utilization bound for **any** partitioning algorithm [Andersson, 2001]
 - $m+1$ tasks of execution time " $1 + \epsilon$ " and period 2: $\frac{m+1}{2}$

Classification: global scheduling

- Temporal and spatial dimensions are dealt with jointly
 - Global unique scheduler and run queue
 - At each scheduling point, the scheduler decides when and where to schedule at most m tasks
 - Task migration allowed



Classification: global scheduling (up to/from here: session 23/24)

- Benefits
 - Suited to dynamic configurations
 - Dominates all other scheduling policies
 - If we consider unconstrained migrations + dynamic priorities
 - Optimal schedulers exist
 - Overloads/underloads spread on all processors
- Drawbacks
 - System overheads: migrations, mutual exclusion for sharing the run queue

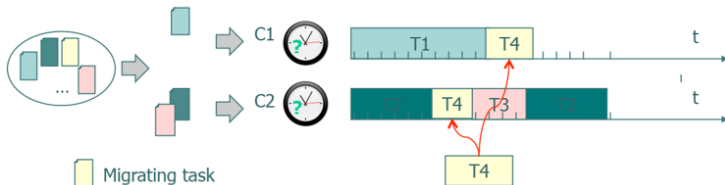
Classification: global scheduling

- Global RM/DM/EDF (preemptive): definition
 - Task priorities assigned according to RM/DM/EDF
 - Scheduling algorithm: The m higher priority tasks are executed on the m processors



Classification: semi-partitioned scheduling

- Partitioned scheduling as far as possible
- Some statically determined tasks may migrate
 - Constraint: Migrating tasks (T4 on the example) must execute on a single processor at a time



Overview of global scheduling policies

- Assumptions

- Tasks

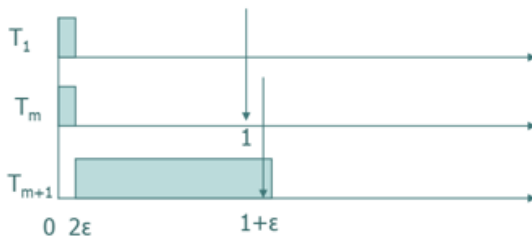
- Periodic tasks (p_i)
 - Implicit deadlines ($d_i = p_i$)
 - Synchronous tasks
 - Independent tasks
 - A single job of a task can be active at a time

- Architecture

- Identical processors
 - Costs are neglected (preemption, migration, scheduling policy)

Scheduling anomalies

- Dhall's effect [Dhall Liu, 1978]
 - Periodic task sets with utilization close to 1 are unschedulable using global RM/EDF
 - $n=m+1$, $p_i=1$, $c_i=2\epsilon$, $u_i=2\epsilon$ for all $1 \leq i \leq m$
 - $P_{m+1}=1+\epsilon$, $C_{m+1}=1$, $u_{m+1}=\frac{1}{1+\epsilon}$
 - Task $m+1$ misses its deadline although U very close to 1
 - We assumed m processor cores

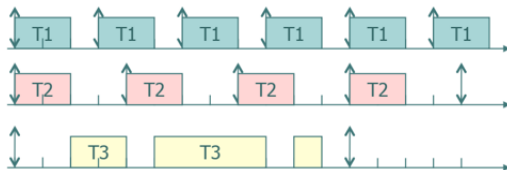


Scheduling anomalies

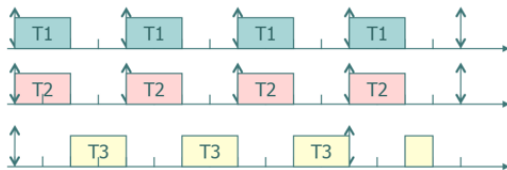
- For G-RM, there may be situations in which schedules exist for a certain task system, but deadlines are violated if periods are extended [Anderson, 2003]
 - $n = 3$, $m = 2$
 - $(P_1 = 3, C_1 = 2), (P_2 = 4, C_2 = 2), (P_3 = 12, C_3 = 7)$
 - Schedulable under global RM
 - If P_1 is increased to 4 and priorities stay the same, T3 misses its deadline

Scheduling anomalies

- $(P_1 = 3, C_1 = 2), (P_2 = 4, C_2 = 2), (P_3 = 12, C_3 = 7)$

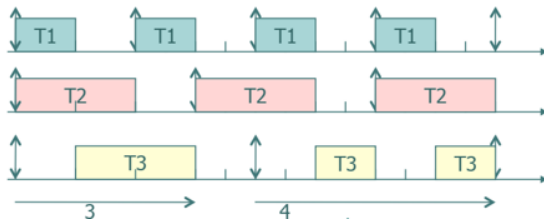


- $(P_1 = 4, C_1 = 2), (P_2 = 4, C_2 = 2), (P_3 = 12, C_3 = 7)$



Scheduling anomalies

- Critical instant not necessarily the simultaneous release of higher priority tasks
 - $n=3, m=2$
 - $(P_1 = 2, C_1 = 1), (P_2 = 3, C_2 = 2), (P_3 = 4, C_3 = 2)$
 - Under RM scheduling
 - Response time of T3 higher at time 4 than at time 0

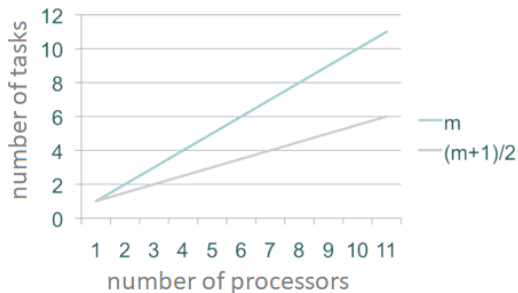


General properties of multiprocessor scheduling

- Exact schedulability condition
 - $U \leq m$ and $u_{max} \leq 1$
 - U = Total utilization
 - u_{max} = Maximum utilization
 - Does not tell for which scheduling algorithm!
- Schedule is cyclic on the hyperperiod H (PPCM(P_i)) for:
 - Deterministic tasks
 - Without memory scheduling algorithms

General properties of multiprocessor scheduling

- Theorem [Srinivasan Baruah, 2002]
 - Non existence of FPJ (FPJ+FPT) scheduling with utilization bound strictly larger than $\frac{m+1}{2}$ for implicit deadline periodic task sets!



Global multiprocessor scheduling: detailed outline

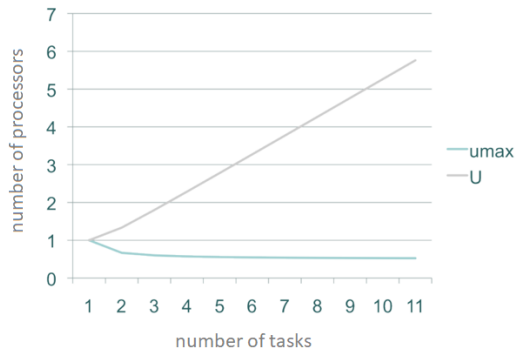
- Transposition of uni-processor algorithms
- Extensions of uni-processor algorithms
 - US (Utilization Threshold)
 - EDF(k)
 - ZL (Zero Laxity)
- Pfair approaches (Proportional Fair)

Transposition of uni-processor algorithms

- Main algorithms
 - RM (Rate Monotonic) \rightarrow G-RM, Global RM
 - EDF (Earliest Deadline First) \rightarrow G-EDF, Global EDF
 - Not optimal anymore
 - Sufficient schedulability tests (depend on u_{max})

G-RM	G-EDF
$u_{max} \leq m/(3m-2)$ and $U \leq m^2/(3m-2)$	$u_{max} \leq m/(2m+1)$ and $U \leq m^2/(2m+2)$
$u_{max} \leq 1/3$ and $U \leq m/3$	$u_{max} \leq 1/2$ and $U \leq (m+1)/2$
$U \leq m/2 * (1-u_{max}) + u_{max}$	$U \leq m - (m-1) u_{max}$

Transposition of uni-processor algorithms



Exten. of global RM/EDF: US (Utilization Threshold)

- Priority assignment depend on an utilization threshold ξ
 - If $u_{max} > \xi$ then T_i is assigned maximal priority
 - Else, T_i 's priority assigned as in original algorithm (RM/EDF)
- Remarks
 - Still non optimal
 - Outperforms the base policy that is used
 - Defies Dhall's effect

Exten. of global RM/EDF: US (Utilization Threshold)

- Example: RM-US[$\xi=\frac{1}{2}$]

	C_i	P_i	U_i	Prio
T1	4	10	2/5	2
T2	3	10	3/10	2
T3	8	12	2/3	∞
T4	5	12	5/12	1
T5	7	12	7/12	∞

Exten. of global RM/EDF: US (Utilization Threshold)

- Utilization bounds

RM-US		EDF-US	
$\xi = m/(3m-2)$	$U \leq m^2/(3m-2)$	$\xi = m/(2m-1)$	$U \leq m^2/(2m-1)$
$\xi = 1/3$	$U \leq (m+1)/3$	$\xi = 1/2$	$U \leq (m+1)/2$

- Remarks

- Utilization bounds do not depend on u_{max} more
- EDF-US $[\xi = \frac{1}{2}]$ attains the best utilization bound possible for FPJ ($\frac{m+1}{2}$)

Exten. of global RM/EDF: EDF(k)

- Task indices by decreasing utilization
 - $u_i \geq u_{i+1}$ for all i in $[1, n]$
- Priority assignment depends on a threshold on task index
 - $i < k$, then maximum priority
 - Else, priority assignment according to original algorithm

Exten. of global RM/EDF: EDF(k)

- Example, EDF(4)

	C_i	P_i	U_i	Prio
T1	4	10	2/5	EDF
T2	3	10	3/10	EDF
T3	8	12	2/3	∞
T4	5	12	5/12	∞
T5	7	12	7/12	∞

Exten. of global RM/EDF: EDF(k)

- Sufficient schedulability test

- $m \geq (k-1) - \lceil \frac{\sum_{i=k+1}^n u_i}{1-u_k} \rceil$
- k_{min} = value minimizing right side of the equation
- With $k = k_{min}$, utilization bound of $\frac{m+1}{2}$ (the best possible for FPJ)
- Comparison with EDF[$\xi = \frac{1}{2}$]
 - Same utilization bound
 - EDF(k_{min}) dominates EDF[$\xi = \frac{1}{2}$]

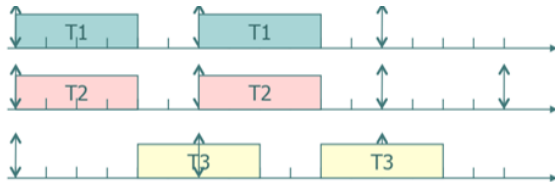
Exten. of global RM/EDF: ZL (Zero Laxity) policies

- XX-ZL: Apply policy XX until Zero Laxity
 - Maximal priority when laxity reaches zero (regardless of the currently running job), original priority assignment for the others
 - In category DPJ (dynamic job scheduling)
 - Policies: EDZL [Lee, 1994], RMZL [Kato et al, 2009], FPZL [Davis et al, 2010]
 - Utilization bound: $\frac{m+1}{2}$
 - Dominates G-EDF

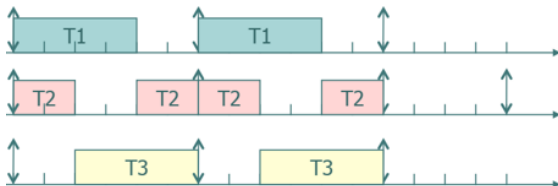
Exten. of global RM/EDF: ZL (Zero Laxity) policies

- Example: $n=3, m=2$; all P_i to 6, all C_i to 4

- G-EDF: T3 misses its deadline



- EDZL: OK



Pfair algorithms: principle (up to/from here: session 24/25)

- Pfair: “Proportionate Fair” [Baruah et al, 1996]
 - Allocate time slots to tasks as close as possible to a “fluid” system, proportional to their utilization factor
- Example
 - $C_1=C_2=3, P_1=P_2=6$ ($u_1=u_2=\frac{1}{2}$)
 - Each task will be “approximately” allocated 1 slot out of 2 (whatever the processor)

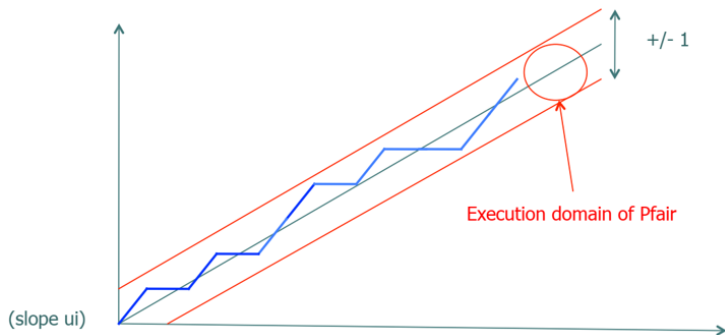
Pfair algorithms: principle

- Lag function: Difference between real and fluid execution
 - Discrete time, successive time slots $[t, t+1]$
 - Weight of a task: $\omega_i = u_i$
- Lag
 - $lag(T_i, t) = \omega_i t - \sum_{u=0}^{t-1} S(T_i, u)$
 - First term: Fluid execution
 - Second term: real execution, with $S(T_i, u) = 1$ if T_i executed in slot u , else 0

Pfair schedule: for all time t , lag in interval $[-1, 1]$

Pfair algorithms: principle

- Example



Pfair algorithms: principle

- Property
 - If a Pfair schedule exists, deadlines are met
- Exact test of existence of a Pfair schedule
 - $\sum_{i=1}^n u_i < m$

Full processor utilization!

Pfair algorithms: construction of a Pfair schedule

- Divide tasks in unity-length sub-tasks
 - Pfair condition: each subtask j executes in a time window between a pseudo-arrival and a pseudo deadline
 - Pseudo-arrival
 - $r(T_i^j) = \lfloor \frac{j-1}{\omega_i} \rfloor$
 - Pseudo-deadline
 - $d(T_i^j) = \lceil \frac{j}{\omega_i} \rceil$

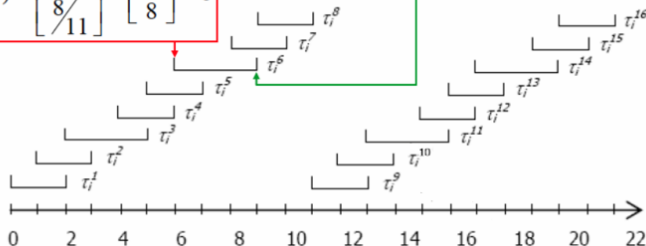
Pfair algorithms: construction of a Pfair schedule

• Example (to be fixed)

■ $(C_i = 8, T_i = 11) \rightarrow \omega_i = u_i = 8/11$

$$r(\tau_i^6) = \left\lfloor \frac{6-1}{8/11} \right\rfloor = \left\lfloor \frac{55}{8} \right\rfloor = 6$$

$$d(\tau_i^6) = \left\lceil \frac{6}{8/11} \right\rceil = \left\lceil \frac{33}{4} \right\rceil = 9$$



Pfair algorithms: scheduling algorithms

- EPDF (Earliest Pseudo-Deadline First)
 - Apply EDF to pseudo-deadlines
 - Optimal only for $m=2$ (2 processors)
- Ongoing works
 - Reduce numbers of context switches and migrations while maintaining optimality

Conclusion

- Multi-processor scheduling is an active research area
- Ongoing works
 - Global multi-core scheduling
 - Semi-partitioned scheduling
 - Determining upper bounds of practical factors (preemption, migration, ...)
 - Implementation in real-time operating systems

The End