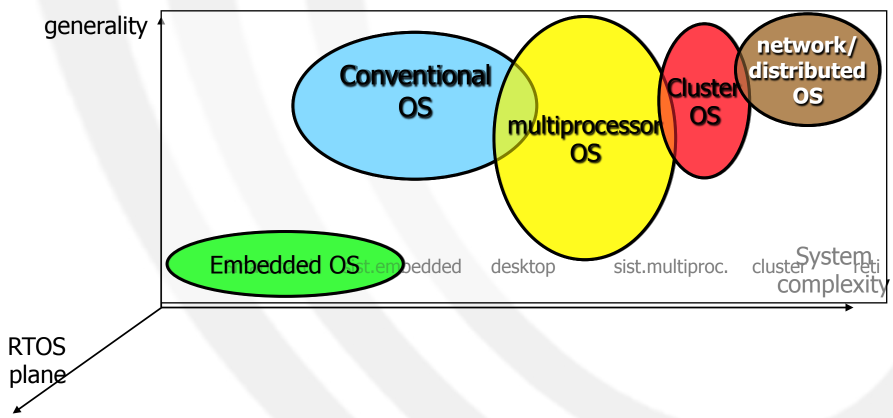
Real-Time Operating Systems

OS space: where are we?



Summary

- Introduction
- Basic Concepts
- Real Time Scheduling
  - Aperiodic Task Scheduling
  - Periodic Task Scheduling
  - Mixed Task Scheduling
    - Priority Servers
- Reference:
  - G.Buttazzo, "Hard Real-Time Computing Systems" Kluwer Academic Publishers, 2002

Introduction

- Real-time system:
  - "A real-time system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced"
  - "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment"

“Real” and “Time”

- Time
  - main difference to other classes of computation
- Real
  - reaction to external events must occur during their evolution

Concept of deadline

- Maximum time within which the task must be completed
- After deadline, a computation is not just late, it is wrong!
- System time (internal time) has to be measured with the same time scale used to measure the controlled environment (external time)
  - Real time does not mean fast but *predictable*

Examples of real time systems

- plant control
- control of production processes / industrial automation
- environmental acquisition and monitoring
- railway switching systems
- automotive applications
- flight control systems
- telecommunication systems
- robotics
- military systems
- space missions
- household appliances
- virtual / augmented reality

Hard vs. soft time

- Hard RT task
  - if missing its deadline may cause catastrophic consequences on the environment under control
- Soft RT task
  - if meeting its deadline is desirable (e.g. for performance reasons) but missing does not cause serious damage
- OS that is able to handle hard RT tasks is called hard real-time OS

Hard vs. soft time

- Typical hard real time activities
  - sensory data acquisition
  - detection of critical conditions
  - low-level control of critical system components
- Areas of application
  - Automotive
    - power-train control, air-bag control, steer by wire, brake by wire
  - Aircraft
    - engine control, aerodynamic control

Hard vs. soft time

- Typical soft real time activities
  - command interpreter of user interface
  - keyboard handling
  - displaying messages on screen
  - transmitting streaming data
- Areas of application
  - Communication systems
    - voice over IP, cellular telephony
    - user interaction
    - comfort electronics (body electronics in cars)

RTOS

Conventional operating systems

- Conventional OS kernels are inadequate w.r.t. RT requirements
  - Multitasking/scheduling
    - provided through system calls
    - does not take time into account (introduces unbounded delays)
  - Interrupt management
    - achieved by setting interrupt priority > than process priority
    - increase system reactivity but may cause unbounded delays on process execution even due to unimportant interrupts
  - Basic IPC and synchronization primitives
    - may cause priority inversion (high priority task blocked by a low priority task)
  - No concept of RT clock/deadline

Aim: minimal response time

Real-time operating systems

- Desirable features of a RTOS
  - Timeliness
    - OS has to provide mechanisms for
      - time management
      - handling tasks with explicit time constraints
  - Predictability
    - to guarantee in advance the deadline satisfaction
    - to notify when deadline cannot be guaranteed
  - Fault tolerance
    - HW/SW failures must not cause a crash
  - Design for peak load
    - All scenarios must be considered
  - Maintainability

13

Real-time operating systems

- Timeliness
  - Achieved through proper scheduling algorithms
    - Core of an RTOS!
- Predictability
  - Affected by several issues
    - Characteristics of the processor (pipelinig, cache, DMA, ...)
    - I/O & interrupts
    - Synchronization & IPC
    - Architecture
    - Memory management
    - Applications
    - Scheduling!

14

Achieving predictability: DMA

- Direct Memory Access
  - to transfer data between a device and the main memory
  - Problem: I/O device and CPU share the same bus
- 2 solution
  - Cycle stealing
    - The DMA steals a CPU memory cycle to execute a data transfer
    - The CPU waits until the transfer is completed
    - Source of non-determinism!
  - Time-slice method
    - Each memory cycle is split in two adjacent time slots
      - One for the CPU
      - One for the DMA
    - More costly, but more predictable!

Achieving predictability: cache

- To obtain a high predictability it is better to have processors without cache
- Source of non-determinism
  - cache miss vs. cache hit
  - writing vs. reading

It is necessary to consider the worst case

Achieving predictability: interrupts

- One of the biggest problem for predictability
  - Typical device driver
    ```
    <enable device interrupt>
    <wait for interrupt>
    <transfer data>
    ```
  - In most OS
    - interrupts served with respect to fixed priority scheme
    - interrupts have higher priorities than processes
    - How much is the delay introduced by interrupts?
      - How many interrupts occur during a task?
  - ➡ problem in real-time systems
    - processes may be of higher importance than I/0 operation!

Interrupts: 1° solution

RK

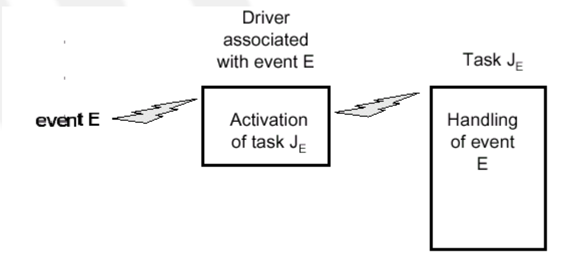
- Disable all interrupts, but timer interrupts
- Characteristics
  - All peripheral devices have to be handled by tasks
  - Data transfer by polling
  - Great flexibility, time for data transfers can be estimated precisely
  - No change of kernel needed when adding devices
- Problems
  - Degradation of processor performance (busy wait)
  - Task must know low level details of the drive

Interrupts: 2° solution

- Disable all interrupts but timer interrupts, and handle devices by special, timer-activated kernel routines
- Advantages
  - unbounded delays due to interrupt driver eliminated
  - periodic device routines can be estimated in advance
  - hardware details encapsulated in dedicated routines
- Problems
  - degradation of processor performance (still busy waiting - within I/0 routines)
  - more inter-process communication than first solution
  - kernel has to be modified when adding devices

Interrupts: 3° solution

- Enable external interrupts and reduce the drivers to the least possible size
  - Driver only activates proper task to take care of device
  - The task executes under direct control of OS, just like any other task
  - Control tasks then have higher priority than device task



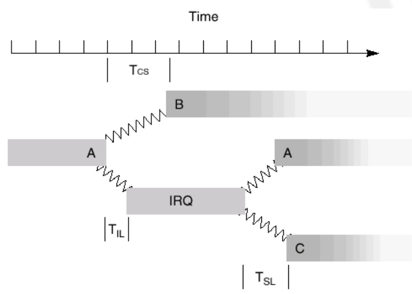
Interrupts: 3° solution

- Advantages
  - busy wait eliminated
  - unbounded delays due to unexpected device handling dramatically reduced (not eliminated !)
  - remaining unbounded overhead may be estimated relatively precisely
- State of the art!

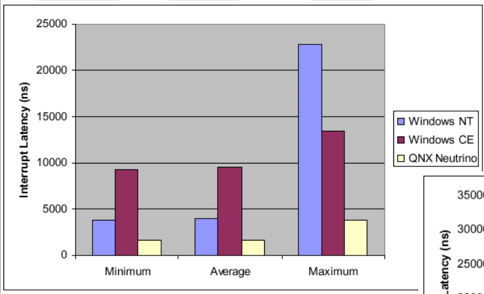
ARTS, HARTIK, SPRING

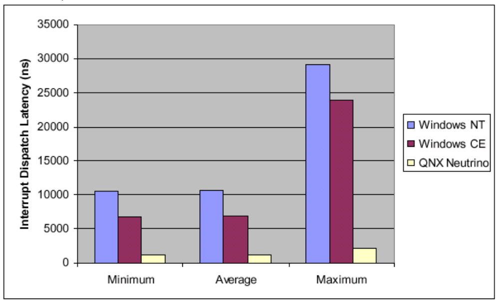
RTOS timing figures

- Interrupt latency (TIL)
  - the time from the start of the physical interrupt to the execution of the first instruction of the interrupt service routine
- Scheduling latency (interrupt dispatch latency) (TSL)
  - the time from the execution of the last instruction of the interrupt handler to the first instruction of the task made ready by that interrupt
- Context-switch time (TCS)
  - the time from the execution of the last instruction of one user-level process to the first instruction of the next user-level process
- Maximum system call time
  - should be predictable & independent of the # of objects in the system



RTOS and interrupts - example





Achieving predictability: system calls

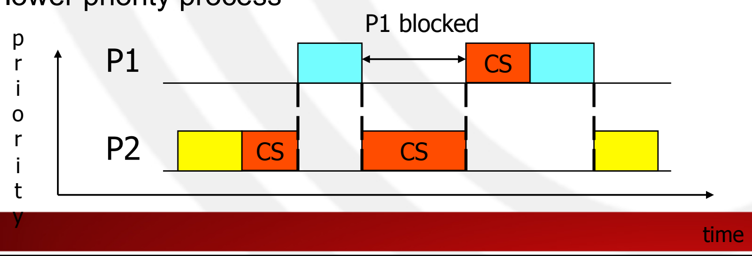
- All system calls have to be characterized by bounded execution time
  - each kernel primitive should be preemptable!
  - non-preemtable calls could delay the execution of critical activities → fault to hard deadline

Achieving predictability: semaphore

- Usual semaphore mechanism not suited for real time applications
  - Priority inversion problem
    - High priority task is blocked by low priority task for unbounded time
  - Solution: use special protocols
    - Priority Inheritance
    - Priority ceiling

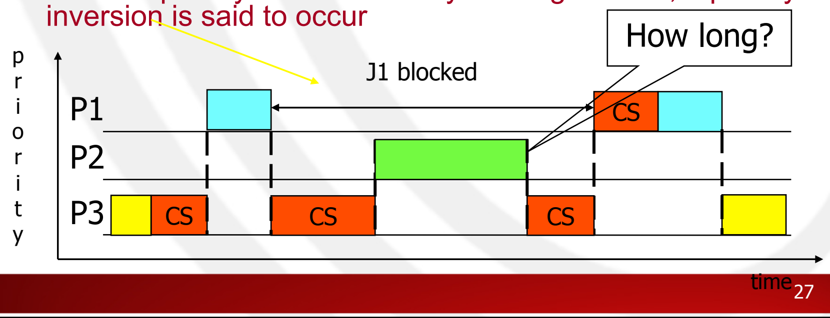
Priority inversion

- Priority(P1) > Priority (P2)
- P1, P2 share a critical section (CS)
- P1 must wait until P2 exits CS even if P(P1) > P(P2)
- Maximum blocking time equals the time needed by P2 to execute its CS
  - It is a direct consequence of mutual exclusion
- In general the blocking time cannot be bounded by CS of the lower priority process



Priority inversion

- Typical characterization of priority inversion
  - A medium-priority task preempts a lower-priority task that is using a shared resource on which a higher-priority task is pending
  - If the higher-priority task is otherwise ready to run, but a medium-priority task is currently running instead, a priority inversion is said to occur



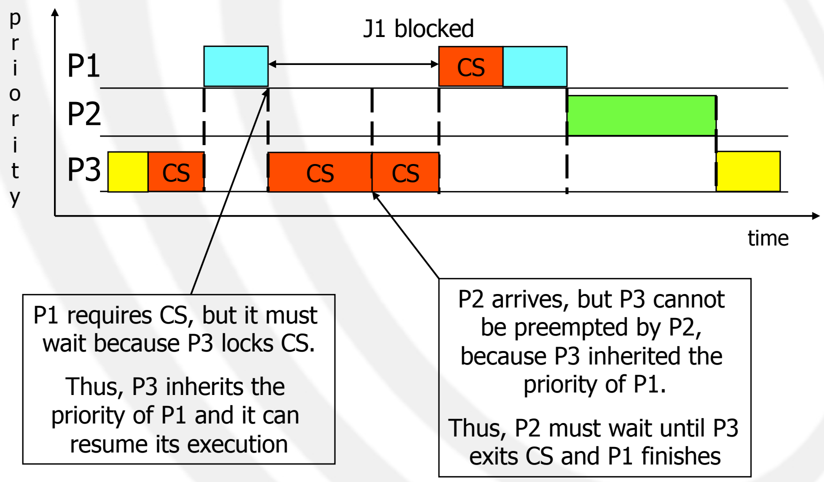
Priority inheritance [Sha 90]

- A job J uses its assigned priority,
  - unless it is in its CS and blocks higher priority jobs
    - In which case, J inherits PH , the highest priority of the jobs blocked by J
    - When J exits the CS, it resumes the priority it had at the point of entry into the CS
- Priority inheritance is transitive

Priority inheritance

- Advantage
  - Transparent to scheduler
- Disadvantage
  - Deadlock possible in the case of bad use of semaphores
  - Chained blocking: if P accesses n resources locked by processes with lower priorities, P must wait for n CS

Priority inheritance - example



Priority ceiling [Sha 90]

- Each resource S_k has a priority ceiling $C(S_k)$ equal to the priority of the highest-priority job that can lock it
- Let J_i the job with the highest priority among jobs ready to run, J_i is assigned to the CPU
- Let S* the resource such that $C(S^*) > C(S_j)$ for all S_j locked by $J_n \neq J_i$
- J_j acquires S_k iff $P(J_i) > C(S^*)$
  If $P(J_i) <= C(S^*)$, J_i is blocked on S* and it cannot acquire S^k

31

Priority ceiling

- When J_i is blocked on a resource, it transmits its priority to the job that locks the resource, let it J_k
  Then, J_k resumes and executes its CS with $P(J_i)$
- When J_k exits CS, it unlocks the resource and the highest priority job blocked on it is awakened
  - The priority of J_k is updated as follows
    - if no other jobs are blocked by J_k, the priority of J_k is set to the nominal one
    - otherwise the priority is set to the highest priority of the jobs blocked by J_k
- Priority inheritance is transitive

32

Priority ceiling

- Properties
  - A high-priority process can be blocked at most once during its execution by lower-priority processes
  - Deadlocks are prevented
  - Transitive blocking is prevented
- Advantage
  - Mutual exclusive access to resources is ensured, by the protocol itself (no semaphores etc. required)
  - Tasks can share resources simply by changing their priorities, thus eliminating the need for semaphores

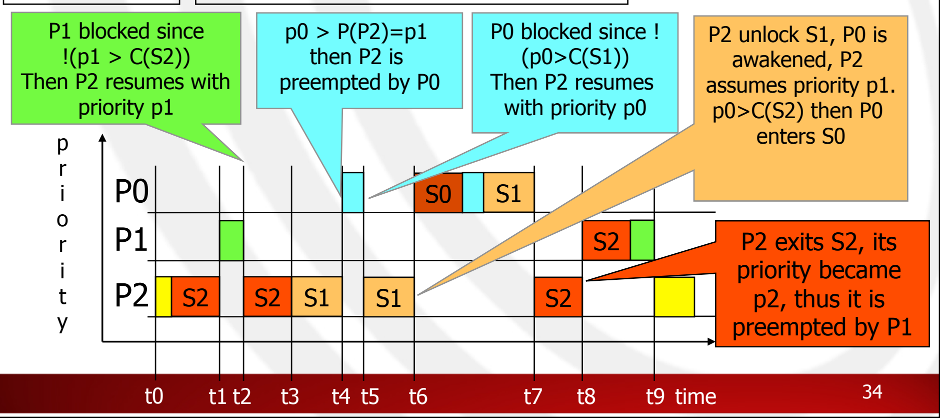
Priority ceiling - example

P(P0) = p0
P(P1) = p1
P(P2) = p2

P0 needs S0 e S1
P1 needs S2
P2 needs S2 and S1(nested in S2)

⇒

C(S0) = P(P0)
C(S1) = P(P0)
C(S2) = P(P1)



Achieving predictability: memory management

- Avoid non-deterministic delays
  - No conventional demand paging (page fault handling!)
    - Page fault & page replacement may cause unpredictable delays
    - May use selective page locking to increase determinism
- Typically used
  - Memory segmentation
  - Static partitioning
    - if applications require similar amounts of memory
- Problems
  - flexibility reduced in dynamic environment
    - careful balancing required between predictabiliy and flexibility

Achieving predictability: applications

- Current programming languages not expressive enough to prescribe precise timing
  - Need of specific RT languages
- Desirable features
  - no dynamic data structures
    - prevent the possibility of correctly predict time needed to create and destroy dynamic structures
  - no recursion
    - impossible estimation of execution time for recursive programs
  - only time-bound loops
    - to estimate the duration of cycles
- Example of RT programming language
  - Real-Time Concurrent C
  - Real-Time Euclid

What RTOS?

- Proprietary
  - VxWorks by WindRiver
  - LynxOS by Lynx
  - Windows CE
- Free/Academical/Open-source
  - RedHat' s eCos
  - RTLinux
  - QNX Neutrino
  - Spring
  - RTX
  - CoCoOS
  - …

http://en.wikipedia.org/wiki/List_of_real-time_operating_systems

REAL-TIME PROCESS MANAGEMENT & SCHEDULING

Processes

- Called tasks in the RT community
- Basic concepts
  - Task scheduling
  - Scheduling problems & anomalies

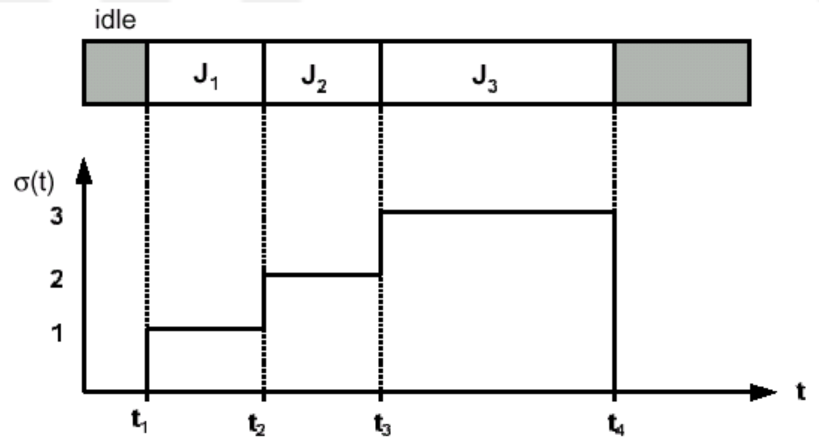
Scheduling – preliminaries

- Key fact:
  - Any RT scheduling policy must be preemptive
    - Tasks performing exception handling may need to preempt running tasks to ensure timely reaction
    - Tasks may have different levels of criticalness. This can be mapped to a preemption scheme
    - More efficient schedules can be produced with preemption

Scheduling – definition

- Given a set of tasks $J = \{J_1, \dots J_n\}$ a schedule is an assignment of tasks to the processor so that each task is executed until completion
- Formally
  - A schedule is a function $s : R^+ \rightarrow N$ such that
    - $\forall\, t \in R^+, \exists\, t1, t2 \in R^+ \mid \forall\, t' \in [t1, t2)\ \ s(t) = s(t')$
- In practice, s is an integer step function
  - $s(t) = k$ means task J_k is executing at time t
  - $s(t) = 0$ means CPU is idle
- Each interval $[\, t_i, t_{i+1}\,)$ with $s(t)$ constant for $t \in [t_i, t_{i+1})$ is called a time slice

Scheduling – example



Scheduling – properties

- A schedule is called *feasible* if all tasks can be completed according to a set of specified constraints
- A set of tasks is called *schedulable* if there exist at least one algorithm that can produce a feasible schedule

43

Scheduling constraints

- The following types of constraints are considered
  - Timing constraints
    - meet your deadline
  - Precedence constraints
    - respect prerequisites
  - Resource constraints
    - access only available resources

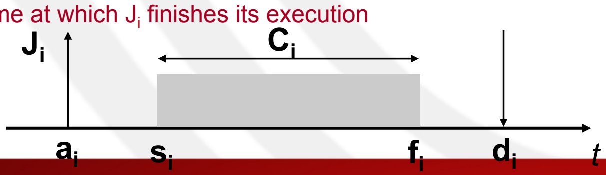
44

Timing constraints

- Real-time systems are characterized mostly by timing constraints
  – Typical timing constraint: deadline
- Deadline missing separates two classes of RT systems
  – Hard
    - missing of deadline can cause catastrophic consequences
  – Soft
    - missing of deadline decreases performance of system

Task characterization

- Arrival time a_i
  – the time J_i becomes ready for execution
  – Also called release time r_i
- Computation time C_i
  – time necessary for execution without interruption
- Deadline d_i
  – time before which task has to complete its execution
- Start time s_i
  – time at which J_i start its execution
- Finish time f_i
  – time at which J_i finishes its execution



ESD

Task characterization

- Lateness L_i
  - $L_i = f_i - d_i$ (it is < 0 if task finishes before deadline)
  - delay of task completion with respect to di
- Laxity or slack time X_i
  - $X_i = d_i - a_i - C_i$
  - maximum time a task can be delayed on first activation to complete before its deadline

47

ESD

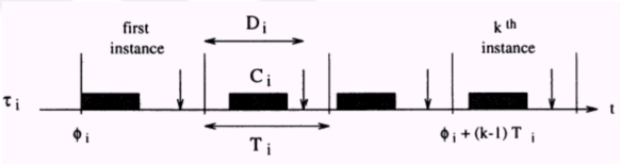
Task models

- Time-driven activation
  - Periodic tasks
- Event-driven activation
  - Aperiodic tasks
  - Sporadic tasks

48

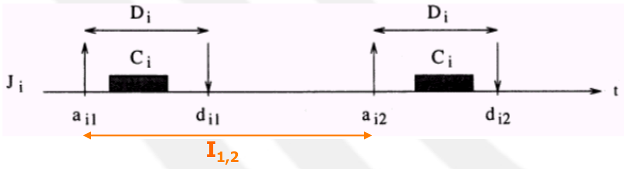
Periodic tasks

- Periodic task τ_i consists of infinite sequence of identical activities, called instances
  - Regularly activated at a constant rate
  - Activation time of first instance of τ is called phase (ϕ_i)
  - T_i = period of the task
  - Each task ti can be characterized by C_i , T_i, , D_i (deadline)
    - C_i, T_i , D_i constant for each instance
    - In most cases: $T_i = D_i$



Aperiodic tasks

- Aperiodic task J_i consists of infinite sequence of identical activities (instances)
  - Their activations are not regular
  - Usually small number of instances
- Sporadic tasks similar to aperiodic, but inter-arrival time is bounded



Aperiodic: $I_{1,2}$ unknown
Sporadic: $I_{1,2} > IT_{min}$

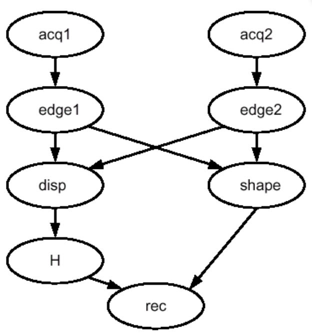
Precedence constraints

- Task have often to respect some precedence relations
  - Described by a DAG G
    - Nodes N(G) = tasks
    - Edges E(G) = precedence relations
  - G induces partial order on task set
- Notation
  - $J_a < J_b$ means J_a is a predecessor of J_b
    - There exists a path from task (node) J_a to task J_b in G
  - $J_a \rightarrow J_b$ means J_a is an immediate predecessor of J_b
    - There exist an edge (J_a, J_b) in E(G)

51

Precedence constraints – example

- System for recognizing object on a conveyer belt through two cameras
- Tasks
  - For each camera
    - image acquisition acq1 and acq2
    - low level image processing edge1 and edge2
  - Task shape to extract two-dimensional features from object contours
  - Task disp to compute pixel disparities from the two images
  - Task H that calculates object height from results of disp
  - Task rec that performs final recognition based on H and shape



52

Resource constraints

- Process view
  - Resource
    - Any SW structure that can be used by process to advance execution
      - Data structure, set of variables, memory area, files, registers of a peripheral, …
  - Distinction between private resources, shared resources and exclusive resources
- Critical section as for conventional systems
  - Conventional semaphore-like structure suffer from priority inversion problem

REAL-TIME SCHEDULING

Scheduling – problem formulation

- Given
  - a set of n tasks $J = \{J_1 , ..., J_n\}$
  - a set of m processor $P = \{P_1, ..., P_m\}$
  - a set of s resources $R = \{R_1, ..., R_s\}$
  - precedences specified by using a precedence graph
  - timing constraints associated to each task
- Scheduling means to assign processors from P and resources from R to tasks from J in order to complete all tasks under the imposed constraints
  - NP-complete!

Scheduling – classification

- Preemptive/non-preemptive
- Static
  - scheduling decisions based on fixed parameters assigned before activation
- Dynamic
  - scheduling decisions based on parameters that change during system evolution
- Off-line
  - scheduling algorithm is preformed on the entire task set before start of system
- On-line
  - scheduling decisions are taken at run-time every time a task enters or leaves the system

Scheduling – guarantee-based algorithms

- Hard RT systems require that
  - feasibility of schedule has to be guaranteed in advance
- Solutions
  - Static RT systems
  - Dynamic RT systems

57

Static RT systems

- Static RT systems
  - All task activations can be pre-calculated off-line
  - Entire schedule can be stored in a table
  - Simple
  - Overhead for dispatching does not depend on the scheduling algorithm → sophisticated algorithm can be used to find optimal scheduling
  - Not flexible

58

Dynamic RT systems

- Activation of new (sporadic) tasks subject to *acceptance test*
  - J = current task set, previously guaranteed
  - J_{new} = newly arriving task
  - J_{new} is accepted iff task set $J' = J \cup \{ J_{new} \}$ is schedulable
- Guarantee mechanism based on worst case assumptions ➡ pessimistic (task could unnecessarily rejected, but potential overload are known in advance)

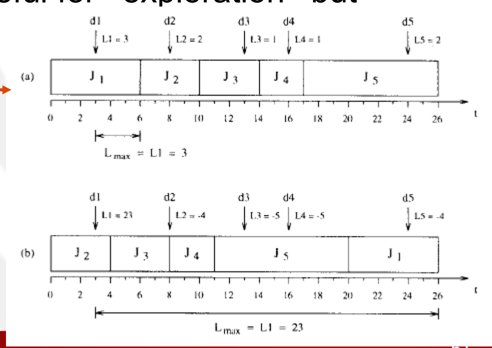
Scheduling metrics

- n tasks
  - Maximum lateness
    - $Lmax = max_{i=1...N} (f_i - d_i)$
  - Maximum number of late tasks
    - $Nlate = \Sigma_{i=1...N}$ miss (f_i)
    - miss (f_i)= 0 if $f_i \leq d_i$, 1 otherwise
  - Average response time
    - $\bar{t_r} = 1/n * \Sigma_{i=1...N} (f_i - a_i)$
  - Total completion time
    - $t_c = max_{i=1...N} (f_i) - min_{i=1...N} (a_i)$
  - Weighted sum of completion times
    - $t_w = \Sigma_{i=1...N} w_i * f_i$

$\Bigg\}$ minimize

Scheduling metrics

- Average response time/total completion time not appropriate for hard real time tasks → loses information about deadline satisfaction
- Maximum lateness: useful for "exploration" but minimizing maximum lateness does not minimize number of tasks that miss their deadlines



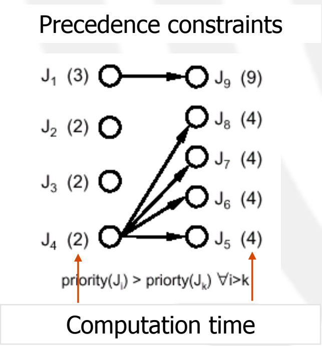
- Max # of late task more significant

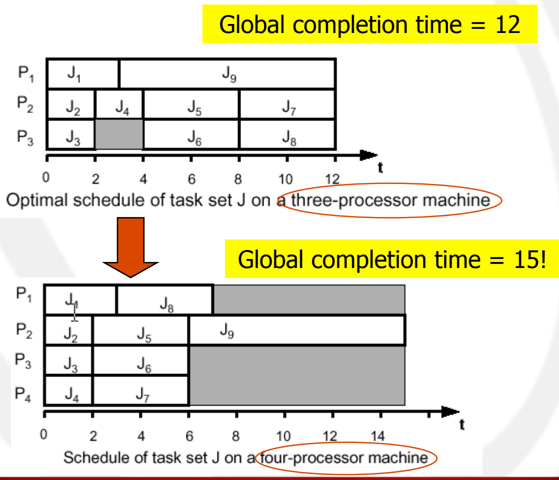
Scheduling anomalies

- [Graham 76]
  - If a task set is optimally scheduled on a multiprocessor with some priority assignment, a fixed number of processors, fixed execution times, and precedence constraints, then
    - increasing the number of processors
    - reducing execution times
    - weakening the precedence constraints
  - can increase the schedule length

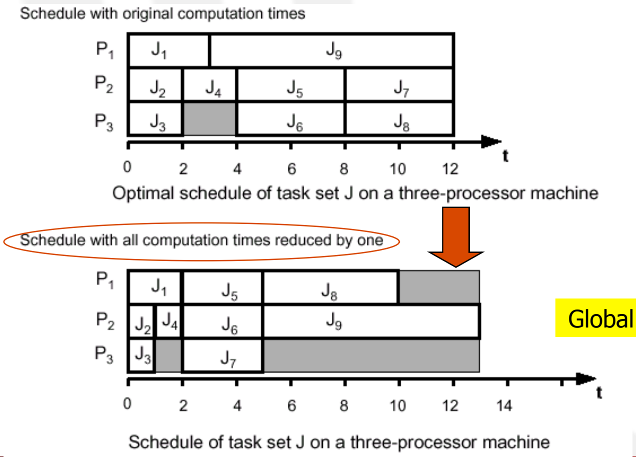
RT-computing is not equivalent to fast computing!

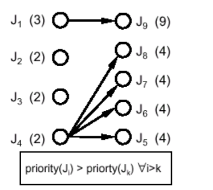
Increasing the number of processors





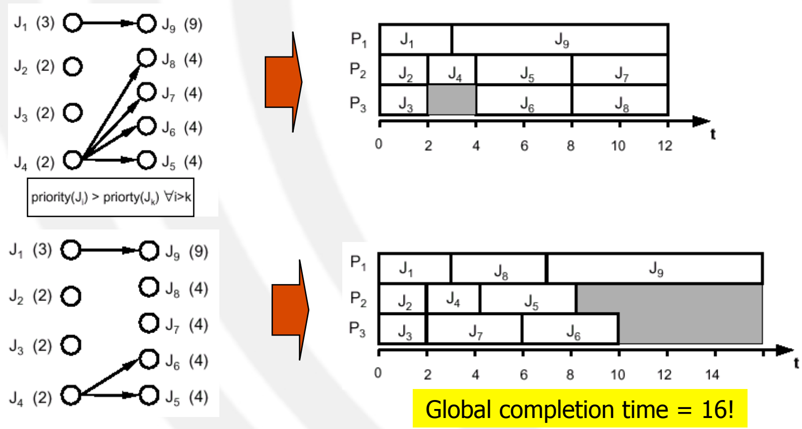
Decreasing computation times





Global completion time = 13!

Weakening precedences



APERIODIC TASK SCHEDULING

Aperiodic task scheduling

- Classification [Graham 79]
  - Triple (α, β, γ)
    - α = the environment on which the task set has to be scheduled (typically # of processors)
    - β = tasks and resource characteristics (preemptive, precedence, synchronous activations etc.)
    - γ = cost function to be optimized

Aperiodic task scheduling

- Examples:
  - 1 | prec | L_{max}
    - uniprocessor machine
    - task set with precedence constraints
    - minimize maximum lateness
  - 2 | sync | Σ_i $Late_i$
    - two processor machine
    - tasks have synchronous arrival time
    - minimize # of late tasks

Aperiodic task scheduling

- Typical scheduling space
  - Task activation times
    - Synchronous activations (a_i=0, $\forall$i)
    - Asynchronous activations ($\exists$i, s.t. $a_i \neq 0$)
  - Task relations
    - With/without precedence relations
  - Preemption
    - With/without preemption

Aperiodic task scheduling algorithms

- Without precedence constraints
  - Jackson's algorithm
  - Horn's algorithm

Jackson's algorithm [Jackson 55]

- To solve 1 | sync | Lmax
  - Uniprocessor, synchronous arrivals, minimize lateness
- No other constraints are considered
  - tasks are independent
    - no precedence relations
    - no shared resources
- Task set J = {J_i (C_i, D_i) | i = 1…n}
  - Computation time C_i
  - Deadline D_i
- Principle: Earliest Due Date (EDD)

P.S. Preemption is not a issue because of sync!

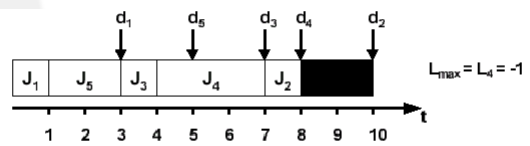
Jackson's algorithm

- It can be proved that:
  - given a set of *n* independent tasks, any algorithm that executes the tasks in order of non-decreasing deadlines is *optimal* with respect to minimize the maximum lateness
- Complexity
  - sorting n values (O (n*log n))
- EDD can not guarantee feasible schedule
  It only guarantees that if a feasible schedule exists it will find it

Jackson's algorithm - example

- Example of feasible schedule

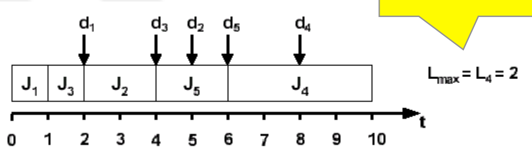
	J_1	J_2	J_3	J_4	J_5
C_i	1	1	1	3	2
d_i	3	10	7	8	5



- Example of unfeasible schedule

EDD minimizes L_{max} but the schedule is not feasible

	J_1	J_2	J_3	J_4	J_5
C_i	1	2	1	4	2
d_i	2	5	4	8	6



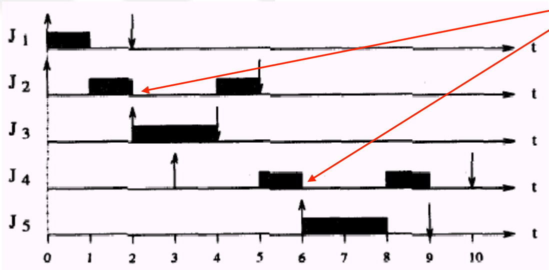
Horn's algorithm [Horn 74]

- To solve 1 | preem | L_{max}
- Principle: Earliest Deadline First (EDF)
- It can be proved that
  - given a set of n independent tasks with arbitrary arrival times, any algorithm that at any time executes the task with the earliest absolute deadline among all the ready tasks is *optimal* with respect to minimizing the maximum lateness
- Complexity
  - O(n) per task
    - inserting a newly arriving task into an ordered list properly
  - n tasks => total complexity O(n2)
- Non preemptive EDF is not optimal!

Horn's algorithm - example

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
c_i	1	2	2	2	2
d_i	2	5	4	10	9

preemption



$L_{max} = L_2 = L_3 = 0$

EDF can not guarantee feasible schedule

Scheduling with precedence constraints

- In General it is a NP-hard problem
  - For special cases polynomial time algorithms possible
- Two schemes
  - Latest Deadline First (LDF)
  - Modified EDF

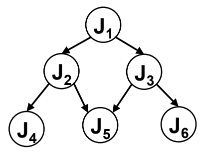
LDF algorithm [Lawler 73]

- To solve 1 | (prec, sync) | Lmax
  - Given:
    - set J of n tasks
    - a DAG describing their precedence relations
  - Arrival times assumed to be simultaneous
  - LDF builds the scheduling queue from tail to head
    - among the tasks without successors or with all successors already selected, LDF selects the one with latest deadline to be scheduled last
    - Iterate this scheme until all tasks are selected
- Complexity
  - O(n2)
    - for each job, the precedence graph has to be visited

LDF algorithm- example

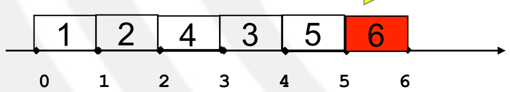
	J_1	J_2	J_3	J_4	J_5	J_6
C_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6

Precedence graph



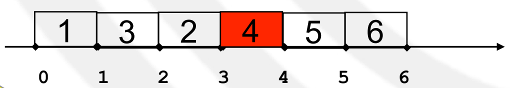
First to be inserted by LDF

LDF



$L_{max} = 0$

EDF



$L_{max} = L_4 = 1$

Not optimal when precedences are considered

EDF with precedence constraints [Chetto et al. 90]

- To solve 1 | (prec, preem) | Lmax
- Modified EDF
  - Transform set J of dependent tasks into set J* of independent ones by an adequate modification of timing parameters
  - Then apply EDF
- The transformation ensures
  - J* schedulable ⇔ J schedulable and prec constraints satisfied

EDF with precedence constraints

- Modification
  - Change arrival times & deadlines such that each task
    - cannot start before its predecessors
    - cannot preempt their successors (other tasks, however, may be preempted)

EDF with precedence constraints

- Modification of arrival (release) times:
  - Try to postpone release time
  - Given two tasks J_a and J_b, $J_a \rightarrow J_b$, the following two conditions must be satisfied
    - $s_b \geq r_b$
      - J_b cannot start earlier than its arrival time
    - $s_b \geq r_a + c_a$
      - J_b cannot start earlier than minimum finish time of J_a
  - Then the new release time for J_b is
    - $r^*_b = \max(r_b, r_a + c_a)$

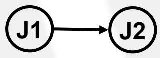
Complexity = O (n^2) 81

EDF with precedence constraints

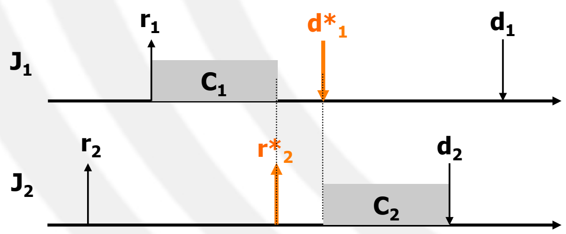
- Modification of deadlines
  - Try to anticipate the deadline
  - Given two tasks J_a and J_b , $J_a \rightarrow J_{b,}$ the following two conditions must be satisfied
    - $f_a \leq d_a$
      - J_a must finish before its deadline
    - $f_a \leq d_b - c_b$
      - J_a must finish before latest start time of b
  - The new deadline for J_a is
    - $d^*_a = \min(d_a, d_b - C_b)$

Complexity = O (n^2) 82

EDF with precedence constraints - example



$r^*_1 = r_1$
$r^*_2 = r_1 + c_1$
$d^*_1 = d_2 - c_2$
$d^*_2 = d_2$



Aperiodic task scheduling: summary

	Sync.activation	Preemtive async. activation	non-preemtive async. activation
independent	EDD (Jackson'55) O(n logn) Optimal	EDF (Horn '74) O (n²) Optimal	Tree search (Bartly '71) O (n n!) Optimal
Precedence constraints	LDF (Lawler '73) O(n²) Optimal	EDF* (Chetto et al. '90) O(n²) Optimal	Spring (Stankovic& Ramamritham '87) O (n²) Heuristic

PERIODIC TASK SCHEDULING

Introduction

- Periodic activities represent the major computational demand in many applications
  - sensory data acquisition
  - control loops
  - system monitoring
- Usually several periodic tasks running concurrently

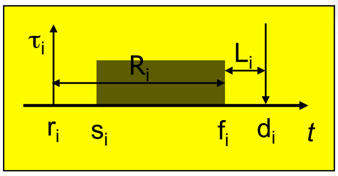
Assumptions

1. Instances of a task are regularly activated at constant rate. Interval between two consecutive activations is the period of the task
2. All instances of a task have the same worst case execution time C_i
3. All instances of a task have the same deadline D_i, and $D_i = T_i$ (deadline = period)
4. All periodic tasks are independent (no precedence relations, no resource constraints)
5. No task can suspend itself (e.g. for I/O)
6. All tasks are released as soon as they arrive
7. All overheads due to the RTOS are assumed to be zero

3,4: can be too tight for practical application

Characterization of periodic tasks

- A periodic task τ_i can be characterized (see assumptions 1-4) by
  - phase f_i
  - period T_i
  - worst case computation time C_i
- Additional parameters
  - Response time $R_i = f_i - r_i$
  - Critical instant (of a task)
    - Release time of a task instance resulting in the largest response time



Scheduling of periodic tasks

- Static scheduling
- Dynamic (process-based) scheduling

Static scheduling

- Cyclic executive approach:
  - With a fixed set of purely periodic tasks it is possible to layout a schedule such that the repeated execution of this schedule will cause all processes to run at their correct rate
  - Essentially a *table of procedure calls*, where each procedure represents part of a code for a "process"
    - Off-line

Cyclic executive approach

- Schedule structure:
  - Tasks are mapped onto a set of *minor cycles*
  - The set of minor cycles constitute a *major cycle* (the complete schedule)
- Cycle durations:
  - Minor cycle m = $\min_i (T_i)$
  - Major cycle M = $LCM(T_i)$ ➡ M = k·m
- Example:
  - T = (7, 10, 21, 35)
    - m = 7
    - M = 2x3x5x7 = 210

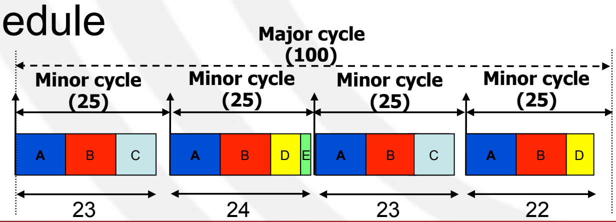
LCM = Least common multiple

Cyclic executive approach: example

- Task set

Process	period,T	Computation Time, C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- Schedule



Cyclic executive approach: example

- Actual code that implements the above cyclic executive schedule

```
loop
  wait_for_interrupt
  Procedure_For_A
  Procedure_For_B
  Procedure_For_C
  wait_for_interrupt
  Procedure_For_A
  Procedure_For_B
  Procedure_For_D
  Procedure_For_E
  wait_for_interrupt
  Procedure_For_A
  Procedure_For_B
  Procedure_For_C
  wait_for_interrupt
  Procedure_For_A
  Procedure_For_B
  Procedure_For_D
end loop
```


Cyclic executive approach

- Advantages
  - No actual process exists at run-time; each minor cycle is just a sequence of procedure calls
  - The procedures share common address space and can pass data between themselves
    - No need for data protection, no concurrency

Cyclic executive approach

- Disadvantages
  - Task periods must be multiple of minor cycle time
    (to make this manageable)
  - It only handles periodic tasks
    - Difficult to incorporate sporadic processes (major cycle time)
  - Difficult to construct cyclic executive
    (equivalent to bin packing problem, NP-hard)

Dynamic (process-based) scheduling

- Fixed-priority scheduling
  - Rate-monotonic (RM) scheduling
  - Deadline-monotonic (DM) scheduling
- Dynamic-priority scheduling
  - EDF

Processor utilization factor

- Given a set Γ of periodic tasks the utilization factor U:
  - is the fraction of processor time spent in the execution of the task set
  - determines the load of the CPU
- C_i/T_i is the fraction of processor time spent in executing t_i
- $U = \Sigma_{i=1\ldots n} C_i/T_i$

97

Processor utilization factor

- U can be improved by:
  - Increasing computation times of the tasks
  - Decreasing the periods of the tasks
- up to a limit below which Γ is not schedulable
- Limit depends on:
  - task set (particular relations among task's periods)
  - algorithm used to schedule the tasks

98

Processor utilization factor

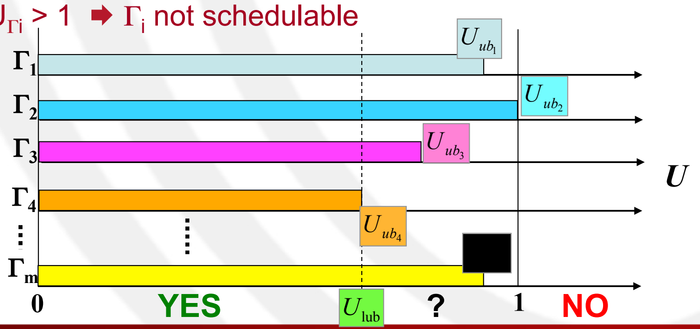
- Upper bound of U, $U_{ub}(\Gamma, A)$
  - Value of U (for a given task set and scheduling algorithm) for which the processor is fully utilized
  - Task set Γ is schedulable using A but any increase of computation time in one of the tasks may make the set infeasible
- Least upper bound of U, U_{lub}
  - Minimum of U_{ub} over all task sets Γ that fully utilize the processor for a given algorithm

$$U_{\text{lub}}(A) = \min(U_{ub}(\Gamma, A)), \forall \Gamma.$$

- U_{lub} allows to easily test for schedulability of set

Processor utilization factor

- Schedulability test
  - $U_{\Gamma i} \le U_{lub}(A) \Rightarrow \Gamma_i$ schedulable
  - $U_{\Gamma i} > U_{lub}(A) \Rightarrow \Gamma_i$ may be schedulable, if the periods of the tasks are suitable related
  - $U_{\Gamma i} > 1 \Rightarrow \Gamma_i$ not schedulable



Rate Monotonic (RM) Scheduling

- Static priority scheduling
- Rate monotonic ➡ priorities are assigned to tasks according to their request rates
- Each process is assigned a (unique) priority based on its period
  - The shorter the period, the higher the priority
  - Given tasks t_i and t_j, $T_i < T_j$ ➡ $P_i > P_j$
- Intrinsically preemptive
  - Currently executing task is preempted by a newly released task with shorter period

RM scheduling

- RM is proven to be optimal
  - If a set of processes can be scheduled (using preemptive priority-based scheduling) with a fixed priority-based assignment scheme, then RM can also schedule the set of processes

RM scheduling: Example

- 1 = lower priority
- 5 = higher priority

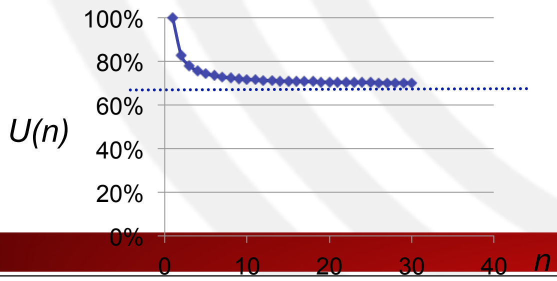
Process	Period, *T*	Priority, *P*
A	25	5
B	60	3
C	42	4
D	105	1
E	75	2

RM schedulability test

- Considers only the utilization of the process set [Liu&Layland 73]

$$U_{lub} = \sum_{i=1}^{n} \left(\frac{C_i}{T_i} \right) \leq n(2^{1/n} - 1)$$

- Total utilization of the process set



N	U(n)
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718

RM schedulability test

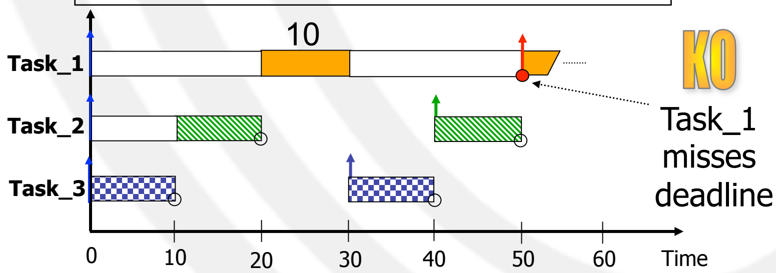
- For large values of n, the bound asymptotically reaches 69.3% (ln2)
- Any process set with a combined utilization of less than 69.3% will always be schedulable under RM
- NOTE
  - This schedulability test is sufficient, but not necessary
    - If a process set passes the test, it will meet all deadlines; if it fails the test, it may or may not fail at run-time
  - The utilization-based test only gives a yes/no answer
    - No indication of actual response times of processes!

RM schedulability test – example

Process	Period, *T*	Computation time, *C*	Priority, *P*	Utilization, *U*
Task_1	50	12	1	0.24
Task_2	40	10	2	0.25
Task_3	30	10	3	0.33

$U = 12/50 + 10/40 + 10/30 = 0.24 + 0.25 + 0.33 = 0.82$

$U > U(3) = 3\,(2^{1/3} - 1) = 0.78$

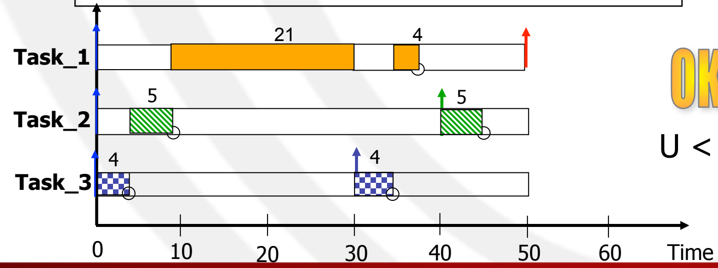


RM schedulability test – example

Process	Period, *T*	Computation time, *C*	Priority, *P*	Utilization, *U*
Task_1	50	25	1	0.5
Task_2	40	5	2	0.125
Task_3	30	4	3	0.133

$U = 25/50 + 5/40 + 4/30 = 0.5 + 0.125 + 0.133 = 0.758$

$U < U(3) = 3\,(2^{1/3} - 1) = 0.78$



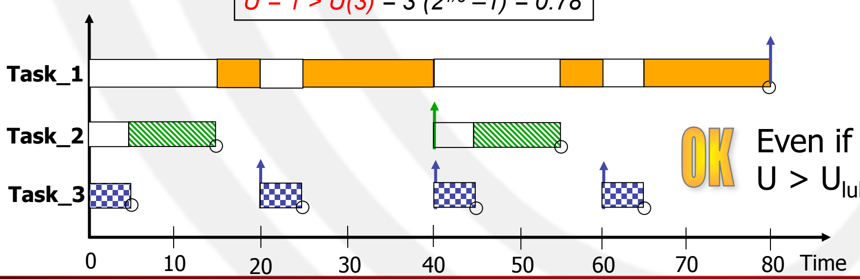
OK

$U < U_{lub}$

RM schedulability test - example

Process	Period, *T*	Computation time, *C*	Priority, *P*	Utilization, *U*
Task_1	80	40	1	0.500
Task_2	40	10	2	0.250
Task_3	20	5	3	0.250

$U = 1 > U(3) = 3\,(2^{1/3} - 1) = 0.78$



OK Even if $U > U_{lub}$

Response time analysis

- Drawbacks of utilization-based tests:
  - Not exact
  - Overestimation of the processor load
- Solution
  - Response time analysis

Response time analysis

- The analysis has two stages:
  1. The worst-case response time of each process is obtained analytically
  2. The response times are then individually compared with the process deadlines
- Response time analysis provides sufficient and necessary conditions for schedulability

Response time analysis

- For any process i, the worst-case response time is given by: $R_i = C_i + I_i$
  - I_i is the maximum interference that process i can experience in any time during the interval $[t, t+R_i)$
  - Interference = preemption
    - For the highest priority process, its worst-case response time will equal its own computation time (that is, R = C)
    - Other processes will suffer interference from higher-priority processes

Response time analysis

- Let i, j be two processes where:
  - priority(j) > priority(i)
- During the interval $[0,R_i)$ we have:
  - Number of releases of j instances = $\lceil Ri / Tj \rceil$
  - Max interference of j = $\lceil Ri / Tj \rceil$ Cj

hp(i) **= set of tasks with higher priority than i**

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

Response time analysis

- Solving by forming a recurrence equation
  - Where the set $\{w_i^0, w_i^1, w_i^2, \ldots\ldots, w_i^n, \ldots.\}$ is monotonically non-decreasing

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

113

Response time analysis

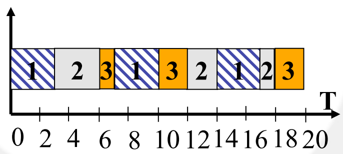
- The equation is solved when $w_{n+1} = w_n$
- If the equation does not have a solution, then the w values will continue to rise
  - Stop when w > D → not schedulable
- Value of w_0 ?
  - The smallest possible value for Ri is Ci

114

Response time analysis - example

Process	Period, *T*	Computation time, *C*	Priority, *P*
Task_1	7	3	3
Task_2	12	3	2
Task_3	20	5	1

- Task 1 => R1 = C1 = 3 <= 7 OK
- Task 2 =>
  - w20 = C2 = 3
  - w21 = 3 + ⌈3/7⌉ 3 = 6
  - w22 = 3 + ⌈6/7⌉ 3 = 6 = w21 => R2 = 6 <= 12 OK
- Task 3 =>
  - w30 = C3 = 5
  - w31 = 5 + ⌈5/12⌉ 3 + ⌈5/7⌉ 3 = 11
  - w32 = 5 + ⌈11/12⌉ 3 + ⌈11/7⌉ 3 = 14
  - w33 = 5 + ⌈14/12⌉ 3 + ⌈14/7⌉ 3 = 17
  - w34 = 5 + ⌈17/12⌉ 3 + ⌈17/7⌉ 3 = 20
  - w35 = 5 + ⌈20/12⌉ 3 + ⌈20/7⌉ 3 = 20
  - => R3 = 20 <= 20 OK



Response time analysis - example

Process	Period, *T*	Computation time, *C*	Priority, *P*
Task_1	80	40	1
Task_2	40	10	2
Task_3	20	5	3

- Process set that failed the utilization-based test
- U = 40/80+10/40+5/20 = 1/2+1/4+1/4 = 1 > 0.78
- Response time test ok
  - R1 = 80 <= 80 OK
  - R2 = 15 <= 40 OK
  - R3 = 5 <= 20 OK

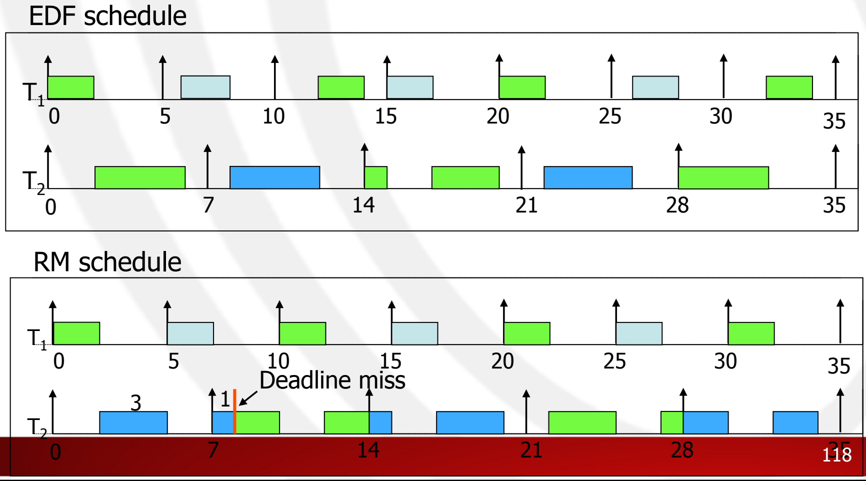
EDF algorithm

- Dynamic scheduling algorithm
  - Dynamic priority assignment
- Idea as for aperiodic tasks
  - Tasks are selected according to their absolute deadlines
  - Tasks with earlier deadlines are given higher priorities
  - It is intrinsically preemptive
    - The currently executing task is preempted whenever another instance with earlier deadline becomes active
- More powerful than RM!
- It works for periodic as well as aperiodic tasks
  - Optimality holds for periodic as well aperiodic tasks

EDF - example

Worst case execution time

Process	Period, T	WCET, C
T_1	5	2
T_2	7	4



EDF schedulability test

- Schedulability of a periodic task set scheduled by EDF can be verified through the processor utilization factor U
- Theorem [Liu&Layland 73]
  - A set of periodic tasks is schedulable with EDF iff

$$\sum_{i=1}^{n}\left(\frac{C_i}{T_i}\right)\leq 1$$

  - This is a sufficient and necessary condition

EDF schedulability test: example

Process	Period, T	WCET, C
T_1	5	2
T_2	7	4

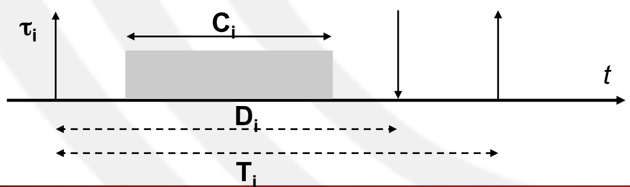
- Processor utilization of the task set
  - U = 2/5 + 4/7 = 34/35 = 0.97
  - U > 0.82
    - schedulability not guaranteed under RM
  - U <1
    - schedulability guaranteed under EDF

Deadline monotonic (DM) scheduling

- Assumption up to now
  - relative deadline = period
- DM scheduling weakens this assumption
  - Static algorithm with preemption
- For DM each periodic tasks τ_i is characterized by four parameters:
  - Relative deadline D_i (equal for all instances)
  - Worst case computation time C_i (equal for all instances)
  - Period T_i
  - Phase f_i

DM scheduling

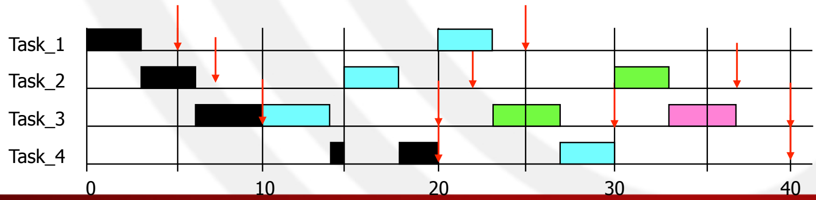
- DM = generalization of RM
  - RMA optimal for D = T
  - DMA extends this optimality for D < T
- Priority of a process inversely proportional to its deadline (but still static!)
  - Given tasks τ_i and τ_j, $D_i < D_j \Rightarrow P_i > P_j$



DM scheduling: example

- Task set not schedulable with RM but schedulable with DM

Process	Period, *T*	Deadline, D	Computation time, *C*	Priority, *P*	Response time, *R*
Task_1	20	5	3	4	3
Task_2	15	7	3	3	6
Task_3	10	10	4	2	10
Task_4	20	20	3	1	20



DM schedulability analysis

- Schedulability can be tested replacing the period with the deadlines in the definition of U

$$U = \Sigma_{i=1 \ldots n} C_i/D_i$$

  – Too pessimistic! (U overestimated)

DM schedulability analysis

- Actual guarantee test based on a modified response time analysis
  - Intuitively: for each τ_i , the sum of its processing time and the interference (preemption) imposed by higher priority tasks must be $\leq D_i$

$$\forall i: 1 \leq i \leq n$$

$$C_i + I_i \leq D_i \qquad I_i = \Sigma_{(j=1\ldots i-1)} \lceil R_i / T_j \rceil C_j$$

EDF for D<T

- EDF applies also to the case D < T
- Different schedulability test
  - Based on the processor demand criterion
- The processor demand of a task τ_i in any interval [t, t+L] is the amount of processing time required by τ_i in [t, t+L] that has to be completed at or before t+L
  - That is, that has to be executed with deadlines $\leq$ t+L

Processor demand for EDF

- Applicable also to the case D=T
- In general, the schedulability of the task set is guaranteed iff the cumulative processor demand in any interval [0, L] ≤ L (the interval length):

$$C_P(0,L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i. \le L$$

Processor demand for EDF

- In the case D<T

$$\forall L \ge 0 \qquad L \ge \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

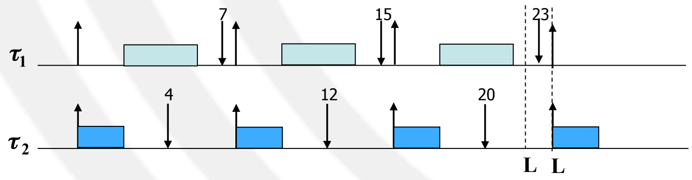
Number of checkpoints is actually limited

Number of completions between 0 and L-D_i

Processor demand for EDF - example

- Schedulability test (L=21)
  - ($\lfloor(21-7)/8\rfloor$+1) • 3 + ($\lfloor(21-4)/8\rfloor$+1) • 2 = 6 + 6 = 12 < 21 OK
- Schedulability test (L=24)
  - ($\lfloor(24-7)/8\rfloor$+1) • 3 + ($\lfloor(24-4)/8\rfloor$+1) • 2 = 9 + 6 = 15 < 24 OK

Task	T	D	C
τ_1	8	7	3
τ_2	8	4	2



Periodic task scheduling: summary

- Rate Monotonic (RM) is optimal among fixed priority assignments (with D=T)
- Earliest Deadline First (EDF) is optimal among dynamic priority assignments
- Deadlines = Periods
  - guarantee test in O(n) using processor utilization, applicable to EDF and RM (only sufficient condition)
- Deadlines < periods
  - polynomial time algorithms for guarantee test
  - fixed priority (DM): response time analysis
  - dynamic priority (EDF): processor demand

Periodic task scheduling: summary

	$D_i = T_i$	$D_i \le T_i$
Static Priority	**RMA** Processor utilization approach $\sum_{i=1}^{n}\left(\frac{C_i}{T_i}\right) \le n(2^{1/n}-1)$	**DMA** Response time approach $\forall i,\ R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \le D_i$
Dynamic Priority	**EDF** Processor utilization approach $\sum_{i=1}^{n}\left(\frac{C_i}{T_i}\right) \le 1$	**EDF** Processor demand approach $\forall L > 0,\quad L \ge \sum_{i=1}^{n}\left(\left\lfloor \frac{L-D_i}{T_i} \right\rfloor + 1\right) C_i$

PRIORITY SERVERS

Introduction

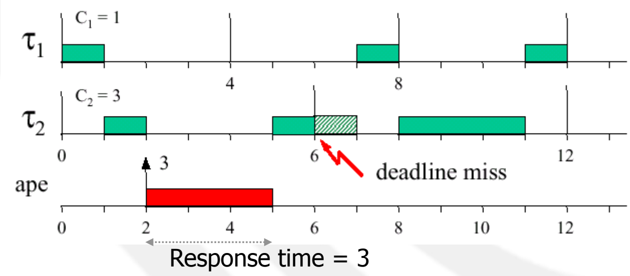
- In most real-time applications there are
  - Both periodic and aperiodic tasks
    - typically periodic tasks are time-driven, hard real-time
    - typically aperiodic tasks are event-driven, soft or hard RT
- Objectives:
  - Guarantee hard RT tasks
  - Provide good average response time for soft RT tasks

Handling periodic and aperiodic tasks

- Solutions
  - Immediate service
  - Background scheduling
  - Aperiodic servers
    - Static priority servers
    - Dynamic priority servers

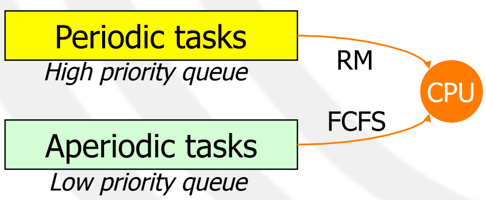
Immediate service

- Aperiodic requests are served as soon as they arrive in the system
- Minimum response times for aperiodic requests
- Low guarantee of periodic tasks



Background scheduling

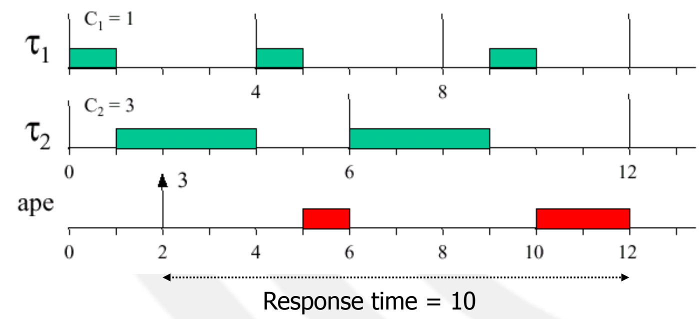
- Handle soft aperiodic tasks in the background behind periodic tasks, that is, in the processor time left after scheduling all periodic tasks
- Aperiodic tasks just get assigned a priority lower than any periodic one
- Organization of background scheduling:



Background scheduling - example

Task	C	T
$\tau 1$	1	4
$\tau 2$	3	6

EDF

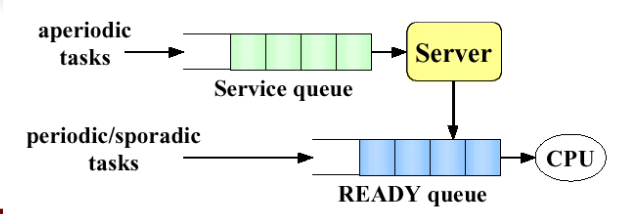


Background scheduling

- Utilization factor under RM < 1 ➡ some processor time is left, it can be used for aperiodic tasks
- High periodic load ➡ bad response time for aperiodic tasks
- Applicable only if no stringent timing requirements for aperiodic tasks
- Major advantage: simplicity

Priority servers

- Alternative scheme to achieve more predictable aperiodic task handling
  - A specific periodic task (server) services aperiodic requests
  - The server is assigned a period T_s and a computation time C_s (capacity of the server)
  - The server is scheduled like any other periodic task, not necessarily at lowest priority
- Conceptual scheme



Priority servers

- Priority server are classified according to the priority scheme (of the periodic scheduler)
  - Static priority servers
    - Polling Server
    - Deferrable server
    - Priority exchange
    - Sporadic server
    - Slack stealing
  - Dynamic priority servers
    - Dynamic Polling Server
    - Dynamic Deferrable Server
    - Dynamic Sporadic Server
    - Total Bandwidth Server
    - Constant Bandwidth Server

Polling server (PS)

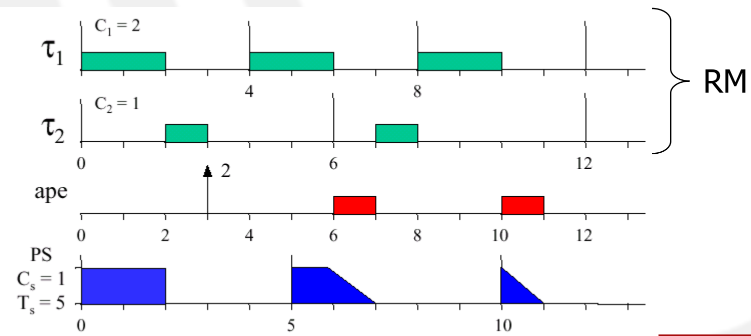
- At the beginning of its period
  - PS is (re)-charged at its full value C_s
  - PS becomes active and is ready to serve any pending aperiodic requests within the limits of its capacity C_s
- If no aperiodic request pending ➡ PS “suspends” itself until beginning of its next period
  - Processor time is used for periodic tasks
  - C_s is discharged to 0
  - If aperiodic task arrives just after suspension of PS it is served in the next period
- If there are aperiodic requests pending ➡ PS serves them until C_s>0

Polling server - example

Task	C	T
τ1	2	4
τ2	1	6

Server:
T_s = 5
C_s = 1

P(τ1)>P(S)>P(τ2)



Polling server analysis

- In the worst-case, the PS behaves as a periodic task with utilization $U_s = C_s/T_s$
- Usually associated to RM for periodic tasks
- Aperiodic tasks execute at the highest priority if
  - Ts = min (T1, … ,Tn)
- Utilization (For U_s=0, reduces to U^{RM})

$$U_{\text{lub}}^{RM+PS}(n) = U_s + n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]$$

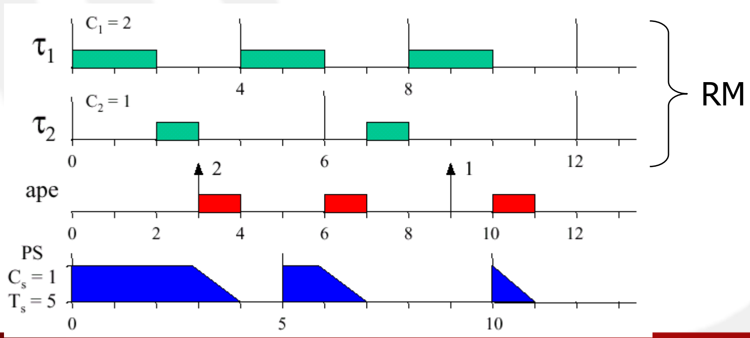
Deferrable server

- Basic approach like Polling Server
- Differences
  - DS preserves its capacity if no requests are pending at invocation of the server
  - Capacity is maintained until server period ➡ aperiodic requests arriving at any time are served as long as the capacity has not been exhausted
- At the beginning of any server period, the capacity is replenished at its full value (as in PS)
  - But no cumulation!

Deferrable server – example

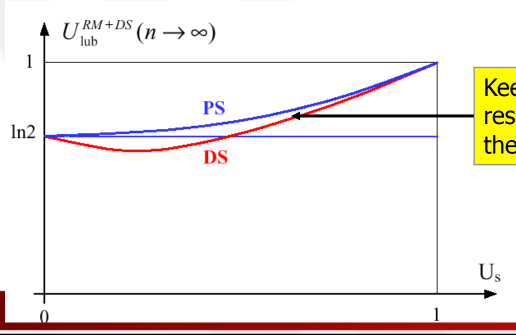
Task	C	T
$\tau 1$	2	4
$\tau 2$	1	6

Server:
$T_s = 5$
$C_s = 1$



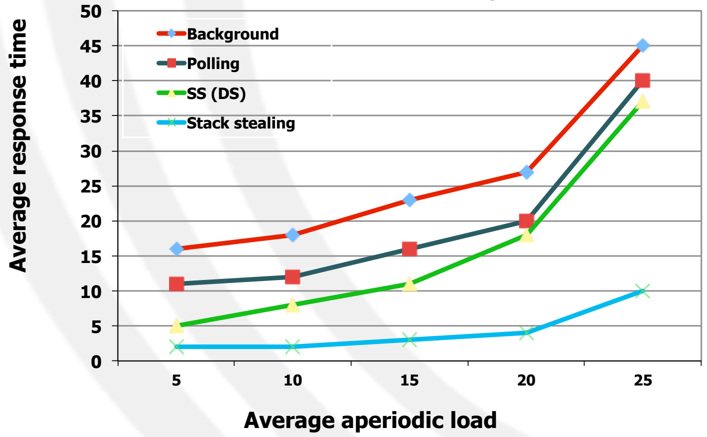
Deferrable server analysis

- Utilization $$U_{\text{lub}}^{RM+DS}(n) = U_s + n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n} - 1\right]$$
- Comparing PS and DS



Keeping the budget improves responsiveness, but decreases the utilization bound.

Schedulability analysis and comparison (U_p=0.69)



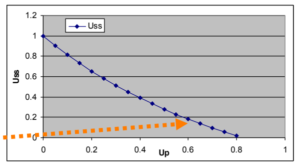
Comparison of fixed priority servers

	Performance	Computational complexity	Memory requirement	Implementation complexity
Background server	C	A	A	A
Polling Server	C	A	A	A
Deferrable Server	B	A	A	A
Priority Exchange	B	B	B	B
Slack Stealing	A	C	C	C

A=excellent B=good C=poor

Dynamic priority servers

- Dynamic scheduling algorithms have higher schedulability bounds than fixed priority ones
- This implies higher overall schedulability
- Example:
  - Suppose
    - Aperiodic server using Slack Stealing
      - $U_{ss} = 2(U_p/2 + 1)^{-2} - 1$
    - Utilization factor of periodic tasks Up = 0.6
    - Periodic task scheduling under RM
      - $U_{ss} = 0.18$
    - Periodic task scheduling under EDF
      - $U_{ss} = 1 - U_p = 0.4$



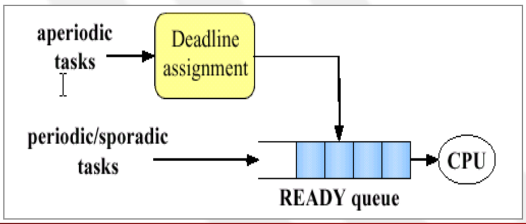
U_{ss} vs. U_p

Dynamic priority servers

- Goal
  - Decreasing average response time for aperiodic tasks and preserving the schedulability of periodic tasks
- Solutions
  - Adaptation of static servers (EDF instead of RM for periodic tasks)
    - Dynamic priority exchange server
    - Improved priority exchange server
    - Dynamic sporadic server
  - Total Bandwidth Server
    - Whenever an aperiodic request enters the system the total bandwidth of the server is immediately assigned to it, whenever possible

Total bandwidth server (TBS)

- Dynamic priority server, used with EDF
  - Each aperiodic request is assigned a deadline so that the server demand does not exceed a given bandwidth Us
  - Aperiodic jobs are inserted in the ready queue and scheduled together with the hard tasks
- Conceptual view:



Periodic tasks are guaranteed *if and only if* $U_p + U_s \leq 1$

Total bandwidth server

- Deadline assignment
  - Job J_k with computation time C_k arrives at time r_k is assigned a deadline

$$d_k = r_k + C_k / U_s$$

- To keep track of the bandwidth assigned to previous jobs, dk must be computed as

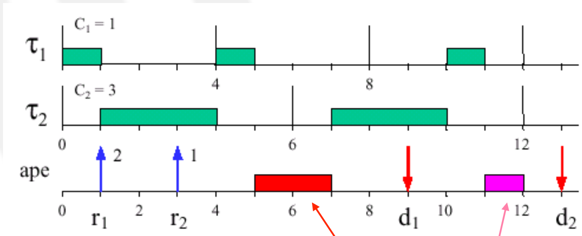
$$d_k = \max(r_k, d_{k-1}) + C_k / U_s$$

- Deadline used to assign priority

Total bandwidth server - example

Task	C	T
τ1	1	4
τ2	3	6

$U_s = 1\text{-}U_p = 1\text{-}0.75 = 0.25$



$d_1 = r_1 + C_1 / U_s = 1 + 2/0.25 = 9$

$d_2 = \max(r_2,d_1) + C_2 / U_s = 9 + 1/0.25 = 13$

153

THE END

THANK YOU

FOR YOUR ATTENTION!

If you liked the SOA class, please consider the possibility of:

1. attending to other ESD courses (PSE, SSE, …)
2. enjoying an exciting thesis/stage in the System Architecture area (ESD Group)

154