

Real-Time Scheduling

Formal Model

[Some parts of this lecture are based on a real-time systems course
of Colin Perkins

<http://csp Perkins.org/teaching/rtes/index.html>]

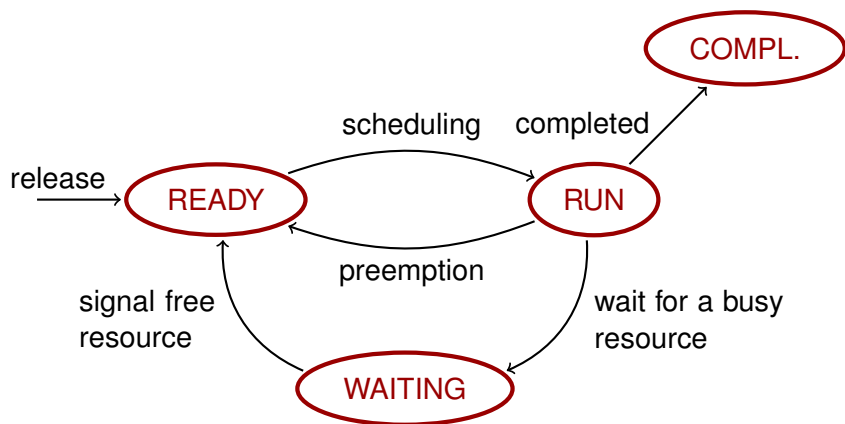
Real-Time Scheduling – Formal Model

- ▶ Introduce an abstract model of real-time systems
 - ▶ abstracts away unessential details
 - ▶ sets up consistent terminology
- ▶ Three components of the model
 - ▶ A workload model that describes applications supported by the system
i.e. jobs, tasks, ...
 - ▶ A resource model that describes the system resources available to applications
i.e. processors, passive resources, ...
 - ▶ Algorithms that define how the application uses the resources at all times
i.e. scheduling and resource access protocols

Basic Notions

- ▶ A *job* is a unit of work that is scheduled and executed by a system
compute a control law, transform sensor data, etc.
- ▶ A *task* is a set of related jobs which jointly provide some system function
check temperature periodically, keep a steady flow of water
- ▶ A job executes on a *processor*
CPU, transmission link in a network, database server, etc.
- ▶ A job may use some (shared) passive *resources*
file, database lock, shared variable etc.

Life Cycle of a Job



Jobs – Parameters

We consider finite, or countably infinite number of jobs J_1, J_2, \dots

Each job has several parameters.

There are four types of job parameters:

- ▶ temporal
 - ▶ release time, execution time, deadlines
- ▶ functional
 - ▶ Laxity type: hard and soft real-time
 - ▶ preemptability, (criticality)
- ▶ interconnection
 - ▶ precedence constraints
- ▶ resource
 - ▶ usage of processors and passive resources

Job Parameters – Execution Time

Execution time e_i of a job J_i – the amount of time required to complete the execution of J_i when it executes alone and has all necessary resources

- ▶ Value of e_i depends upon complexity of the job and speed of the processor on which it executes; may change for various reasons:
 - ▶ Conditional branches
 - ▶ Caches, pipelines, etc.
 - ▶ ...
- ▶ **Execution times fall into an interval $[e_i^-, e_i^+]$** ; we assume that we know this interval (WCET analysis) but not necessarily e_i

We usually validate the system using only e_i^+ for each job
i.e. assume $e_i = e_i^+$

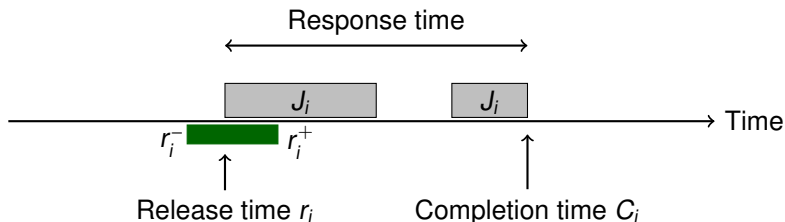
Job Parameters – Release and Response Time

Release time r_i – the instant in time when a job J_i becomes available for execution

- ▶ Release time may *jitter*, only an interval $[r_i^-, r_i^+]$ is known
- ▶ A job can be executed at any time at, or after, its release time, provided its processor and resource demands are met

Completion time C_i – the instant in time when a job completes its execution

Response time – the difference $C_i - r_i$ between the completion time and the release time

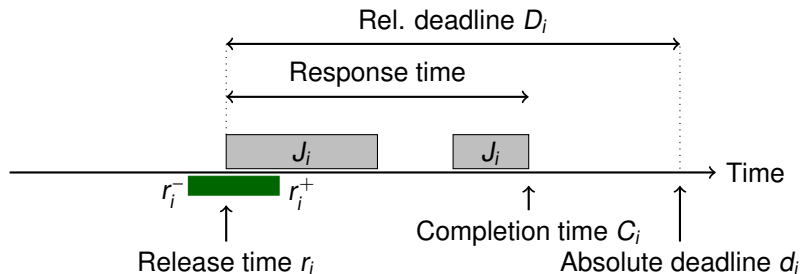


Job Parameters – Deadlines

Absolute deadline d_i – the instant in time by which a job must be completed

Relative deadline D_i – the maximum allowable response time
i.e. $D_i = d_i - r_i$

Feasible interval is the interval $(r_i, d_i]$



A *timing constraint* of a job is specified using release time together with relative and absolute deadlines.

Laxity Type – Hard Real-Time

A **hard real-time constraint** specifies that a job should never miss its deadline.

Examples: Flight control, railway signaling, anti-lock brakes, etc.

Several more precise definitions occur in literature:

- ▶ A timing constraint is hard if the failure to meet it is considered a fatal error
e.g. a bomb is dropped too late and hits civilians
- ▶ A timing constraint is hard if the usefulness of the results falls off abruptly (may even become negative) at the deadline
Here the nature of abruptness allows to soften the constraint

Definition 1

A *timing constraint is hard* if the user requires *formal validation* that the job meets its timing constraint.

Laxity Type – Soft Real-Time

A **soft real-time constraint** specifies that a job could occasionally miss its deadline

Examples: stock trading, multimedia, etc.

Several more precise definitions occur in literature:

- ▶ A timing constraint is soft if the failure to meet it is undesirable but acceptable if the probability is low
- ▶ A timing constraint is soft if the usefulness of the results decreases at a slower rate with *tardiness* of the job
e.g. the probability that a response time exceeds 50 ms is less than 0.2

Definition 2

A *timing constraint is soft* if either validation is not required, or only a demonstration that a *statistical constraint* is met suffices.

Jobs – Preemptability

Jobs may be interrupted by higher priority jobs

- ▶ A job is *preemptable* if its execution can be interrupted
- ▶ A job is *non-preemptable* if it must run to completion once started
(Some preemptable jobs have periods during which they cannot be preempted)
- ▶ The *context switch time* is the time to switch between jobs
(Most of the time we assume that this time is negligible)

Reasons for preemptability:

- ▶ Jobs may have different levels of criticality
e.g. brakes vs radio tuning
- ▶ Priorities may make part of scheduling algorithm
e.g. resource access control algorithms

Jobs – Precedence Constraints

Jobs may be constrained to execute in a particular order

- ▶ This is known as a *precedence constraint*
- ▶ A job J_i is a *predecessor* of another job J_k and J_k a *successor* of J_i (denoted by $J_i < J_k$) if J_k cannot begin execution until the execution of J_i completes
- ▶ J_i is an *immediate predecessor* of J_k if $J_i < J_k$ and there is no other job J_j such that $J_i < J_j < J_k$
- ▶ J_i and J_k are *independent* when neither $J_i < J_k$ nor $J_k < J_i$

A job with a precedence constraint becomes ready for execution when its release time has passed and when all predecessors have completed.

Example: authentication before retrieving an information, a signal processing task in radar surveillance system precedes a tracker task

Tasks – Modeling Reactive Systems

Reactive systems – run for unlimited amount of time

A system parameter: number of tasks

- ▶ may be known in advance (flight control)
- ▶ may change during computation (air traffic control)

We consider three types of tasks

- ▶ Periodic – jobs executed at regular intervals, hard deadlines
- ▶ Aperiodic – jobs executed in random intervals, soft deadlines
- ▶ Sporadic – jobs executed in random intervals, hard deadlines

... precise definitions later.

A **processor**, P , is an **active** component on which jobs are scheduled

The general case considered in literature:

m processors P_1, \dots, P_m , each P_i has its *type* and *speed*.

We mostly concentrate on **single processor** scheduling

- ▶ Efficient scheduling algorithms
- ▶ In a sense subsumes multiprocessor scheduling where tasks are assigned *statically* to individual processors
i.e. all jobs of every task are assigned to a single processor

Multi-processor scheduling is a rich area of current research, we touch it only lightly (later).

Resources

A **resource**, R , is a *passive* entity upon which jobs may depend

In general, **we consider n resources R_1, \dots, R_n of distinct types**

Each R_i is used in a mutually exclusive manner

- ▶ A job that acquires a free resource locks the resource
- ▶ Jobs that need a busy resource have to wait until the resource is released
- ▶ Once released, the resource may be used by another job (i.e. it is not consumed)

(More generally, each resource may be used by k jobs concurrently, i.e., there are k units of the resource)

Resource requirements of a job specify

- ▶ which resources are used by the job
- ▶ the time interval(s) during which each resource is required (precise definitions later)

Scheduling

Schedule assigns, in every time instant, processors and resources to jobs.

More formally, a schedule is a function

$$\sigma : \{J_1, \dots\} \times \mathbb{R}_0^+ \rightarrow \mathcal{P}(\{P_1, \dots, P_m, R_1, \dots, R_n\})$$

so that for every $t \in \mathbb{R}_0^+$ there are rational $0 \leq t_1 \leq t \leq t_2$ such that $\sigma(J_i, \cdot)$ is constant on $[t_1, t_2)$.

(We also assume that there is the least time quantum in which scheduler does not change its decisions, i.e. each of the intervals $[t_1, t_2)$ is larger than a fixed $\varepsilon > 0$.)

Valid and Feasible Schedule

A schedule is *valid* if it satisfies the following conditions:

- ▶ Every processor is assigned to at most one job at any time
- ▶ Every job is assigned to at most one processor at any time
- ▶ No job is scheduled before its release time
- ▶ The total amount of processor time assigned to a given job is equal to its actual execution time
- ▶ *All the precedence and resource usage constraints are satisfied*

A schedule is *feasible* if *all jobs with hard real-time constraints* complete before their deadlines

A set of jobs is *schedulable* if there is a feasible schedule for the set.

Scheduling – Algorithms

Scheduling algorithm computes a schedule for a set of jobs

A set of jobs is *schedulable according to a scheduling algorithm* if the algorithm produces a feasible schedule

Sometimes efficiency of scheduling algorithms is measured using a *cost function*:

- ▶ the maximum/average response time
- ▶ the maximum/average lateness – the difference between the completion time and the absolute deadline, i.e. $C_i - d_i$
- ▶ miss rate – the percentage of jobs that are executed but completed too late
- ▶ ...

Definition 3

A scheduling algorithm is *optimal* if it always produces a feasible schedule whenever such a schedule exists, and if a cost function is given, minimizes the cost.

Real-Time Scheduling

Individual Jobs

Scheduling of Individual Jobs

We start with scheduling of finite sets of jobs $\{J_1, \dots, J_m\}$ for execution on **single processor** systems

Each J_i has a release time r_i , an execution time e_i and a relative deadline D_i .

We assume hard real-time constraints

The question: Is there an optimal scheduling algorithm?

We proceed in the direction of growing generality:

1. No resources, independent, synchronized (i.e. $r_i = 0$ for all i)
2. No resources, independent but not synchronized
3. No resources but possibly dependent
4. The general case

No resources, Independent, Synchronized

	J_1	J_2	J_3	J_4	J_5
e_j	1	1	1	3	2
d_j	3	10	7	8	5

Is there a feasible schedule? Minimize maximal lateness.

Note: Preemption does not help in synchronized case

Theorem 4

If there are no resource contentions, then executing independent jobs in the order of non-decreasing deadline (EDD) produces a feasible schedule (if it exists) and minimizes the maximal lateness (always).

Proof.

Any feasible schedule σ can be transformed in finitely many steps to EDD schedule whose lateness is \leq the lateness of σ (whiteboard). \square

Is there any simple schedulability test?

$\{J_1, \dots, J_n\}$ where $d_1 \leq \dots \leq d_n$ is schedulable iff

$\forall i \in \{1, \dots, n\} : \sum_{k=1}^i e_k \leq d_i$

No resources, Independent (No Synchro)

	J_1	J_2	J_3
r_i	0	0	2
e_i	1	2	2
d_i	2	5	4

- ▶ find a (feasible) schedule (with and without preemption)
- ▶ determine response time of each job in your schedule
- ▶ determine lateness of each job in your schedule (is the maximal lateness minimized?)

Recall that lateness of J_i is equal to $C_i - d_i$

Preemption makes a difference

No resources, Independent (No Synchro)

Earliest Deadline First (EDF) scheduling:

At any time instant, a job with the earliest absolute deadline is executed

Here EDF works in the preemptive case but not in the non-preemptive one.

	J_1	J_2
r_i	0	1
e_i	4	2
d_i	7	5

No Resources, Dependent (No Synchro)

Theorem 5

If there are no resource contentions, jobs are independent and preemption is allowed, the EDF algorithm finds a feasible schedule (if it exists) and minimizes the maximal lateness.

Proof.

We show that any feasible schedule σ can be transformed in finitely many steps to EDF schedule which is feasible and whose lateness is \leq the lateness of σ .

Let σ be a feasible schedule but not EDF. Assume, w.l.o.g., that for every $k \in \mathbb{N}$ at most one job is executed in the interval $[k, k + 1)$ and that all release times and deadlines are in \mathbb{N} .

(Otherwise rescale by the least common multiple.)

No Resources, Dependent (No Synchro)

Proof cont.

We say that σ **violates** EDF at k if there are two jobs J_a and J_b that satisfy:

- ▶ J_b is executed in $[k, k + 1)$
- ▶ J_a is executed in $[\ell, \ell + 1)$ for $\ell > k$
- ▶ $d_a < d_b$

We say that J_a and J_b **witness** the violation.

Let $k \in \mathbb{N}$ be the *least* time instant such that σ violates EDF at k as witnessed by jobs J_a and J_b .

Assume, w.l.o.g. that J_a has the minimum deadline among all jobs ready for execution at k .

Let us define a new schedule σ' which is the same as σ except:

- ▶ executes J_a in $[k, k + 1)$
- ▶ executes J_b in $[\ell, \ell + 1)$

Then σ' does not violate EDF at any $k' \leq k$.

So finitely many transformations transform any feasible schedule to EDF.

No resources, Independent (No Synchro)

The **non-preemptive** case is **NP-hard**.

Heuristics are needed, such as the **Spring algorithm**, that usually work in much more general setting (with resources etc.)

Use the notion of *partial schedule* where only a subset of tasks has been scheduled.

Exhaustive search through partial schedules

- ▶ start with an empty schedule
- ▶ in every step either
 - ▶ add a job which maximizes a *heuristic function* H among jobs that have not yet been tried in this partial schedule
 - ▶ or backtrack if there is no such a job
- ▶ After failure, backtrack to previous partial schedule

Heuristic function identifies plausible jobs to be scheduled (earliest release, earliest deadline, etc.)

No resources, Dependent (No Synchro)

Theorem 6

Assume that there are no resource contentions and jobs are preemptable. There is a polynomial time algorithm which decides whether a feasible schedule exists and if yes, then computes one.

Idea: Reduce to independent jobs by changing release times and deadlines. Then use EDF.

Observe that if $J_i < J_k$ then replacing

- ▶ r_k with $\max\{r_k, r_i + e_i\}$
(J_k cannot be scheduled for execution before $r_i + e_i$ because J_i cannot be finished before $r_i + e_i$)
- ▶ d_i with $\min\{d_i, d_k - e_k\}$
(J_i must be finished before $d_k - e_k$ so that J_k can be finished before d_k)

does not change feasibility.

Replace systematically according to the precedence relation.

No Resources, Dependent (No Synchro)

Define r_k^* , d_k^* systematically as follows:

- ▶ Pick J_k whose all predecessors have been processed and compute $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$. Repeat for all jobs.
- ▶ Pick J_k whose all successors have been processed and compute $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$. Repeat for all jobs.

This gives a new set of jobs J_1^*, \dots, J_m^* where each J_k^* has the release time r_k^* and the absolute deadline d_k^* .

We impose **no precedence constraints** on J_1^*, \dots, J_m^* .

Lemma 7

$\{J_1, \dots, J_m\}$ is feasible iff $\{J_1^, \dots, J_m^*\}$ is feasible. If EDF schedule is feasible on $\{J_1^*, \dots, J_m^*\}$, then the same schedule is feasible on $\{J_1, \dots, J_m\}$.*

The same schedule means that whenever J_i^ is scheduled at time t , then J_i is scheduled at time t .*

No Resources, Dependent (No Synchro)

Recall: $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$ and
 $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$

Proof of Lemma 7.

\Rightarrow : It is easy to show that in *no feasible schedule* on $\{J_1, \dots, J_m\}$ any job J_k can be released before r_k^* and completed before d_k^* (otherwise, precedence constraints would be violated). \Leftarrow : Assume that EDF σ is feasible on $\{J_1^*, \dots, J_m^*\}$. Let us use σ on $\{J_1, \dots, J_m\}$.

i.e. J_i is executed iff J_i^ is executed.*

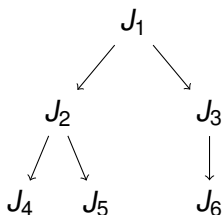
Timing constraints of $\{J_1, \dots, J_m\}$ are satisfied since $r_k \leq r_k^*$ and $d_k \geq d_k^*$ for all k .

Precedence constraints: Assume that $J_s < J_t$. Then J_s^* executes completely before J_t^* since $r_s^* < r_s^* + e_s \leq r_t^*$ and $d_s^* \leq d_t^* - e_t < d_t^*$ and σ is EDF on $\{J_1^* \dots, J_m^*\}$.

No Resources, Dependent (No Synchro)

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

Dependencies:



Find a feasible schedule.

Recall: $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$ and
 $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$

Resources, Dependent, Not Synchronized

Even the preemptive case is NP-hard

- ▶ reduce the non-preemptive case without resources to the preemptive with resources
- ▶ Use a common resource R .
 - ▶ Whenever a job starts its execution it locks the resource R .
 - ▶ Whenever a job finishes its execution it releases the resource R .

Could be solved using heuristics, e.g. the Spring algorithm.

Real-Time Scheduling

Scheduling of Reactive Systems

[Some parts of this lecture are based on a real-time systems course
of Colin Perkins

<http://csparks.org/teaching/rtes/index.html>]

Reminder of Basic Notions

- ▶ Jobs are executed on processors and need resources
- ▶ Parameters of jobs
 - ▶ temporal:
 - ▶ release time – r_i
 - ▶ execution time – e_i
 - ▶ absolute deadline – d_i
 - ▶ derived params: relative deadline (D_i), completion time, response time, ...
 - ▶ functional:
 - ▶ laxity type: hard vs soft
 - ▶ preemptability
 - ▶ interconnection
 - ▶ precedence constraints (independence)
 - ▶ resource
 - ▶ what resources and when are used by the job
- ▶ Tasks = sets of jobs

Reminder of Basic Notions

- ▶ Schedule assigns, in every time instant, processors and resources to jobs
- ▶ valid schedule = correct (common sense)
- ▶ Feasible schedule = valid and all hard real-time tasks meet deadlines
- ▶ Set of jobs is schedulable if there is a feasible schedule for it

- ▶ Scheduling algorithm computes a schedule for a set of jobs
- ▶ Scheduling algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists, and if a cost function is given, minimizes the cost

Scheduling Reactive Systems

We have considered scheduling of individual jobs

From this point on we concentrate on reactive systems

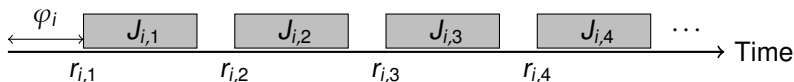
i.e. systems that run for unlimited amount of time

Recall that a task is a set of related jobs that jointly provide some system function.

- ▶ We consider various types of tasks
 - ▶ Periodic
 - ▶ Aperiodic
 - ▶ Sporadic
- ▶ Differ in execution time patterns for jobs in the tasks
- ▶ Must be modeled differently
 - ▶ Differing scheduling algorithms
 - ▶ Differing impact on system performance
 - ▶ Differing constraints on scheduling

Periodic Tasks

- ▶ A set of jobs that are executed repeatedly at regular time intervals can be modeled as a *periodic task*



- ▶ Each periodic task T_i is a sequence of jobs $J_{i,1}, J_{i,2}, \dots, J_{i,n}, \dots$
 - ▶ The *phase* φ_i of a task T_i is the release time $r_{i,1}$ of the first job $J_{i,1}$ in the task T_i ;
tasks are *in phase* if their phases are equal
 - ▶ The *period* p_i of a task T_i is the minimum length of all time intervals between release times of consecutive jobs in T_i
 - ▶ The *execution time* e_i of a task T_i is the maximum execution time of all jobs in T_i
 - ▶ The *relative deadline* D_i is relative deadline of all jobs in T_i(The period and execution time of every periodic task in the system are known with reasonable accuracy at all times)

Periodic Tasks – Notation

The 4-tuple $T_i = (\varphi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase φ_i , period p_i , execution time e_i , and relative deadline D_i

For example: jobs of $T_1 = (1, 10, 3, 6)$ are

- ▶ released at times 1, 11, 21, ...,
- ▶ execute for 3 time units,
- ▶ have to be finished in 6 time units (the first by 7, the second by 17, ...)

Default phase of T_i is $\varphi_i = 0$ and default relative deadline is $d_i = p_i$

$T_2 = (0, 10, 3, 6)$ satisfies $\varphi = 0$, $p_i = 10$, $e_i = 3$, $D_i = 6$, i.e. jobs of T_2 are

- ▶ released at times 0, 10, 20, ...,
- ▶ execute for 3 time units,
- ▶ have to be finished in 6 time units (the first by 6, the second by 16, ...)

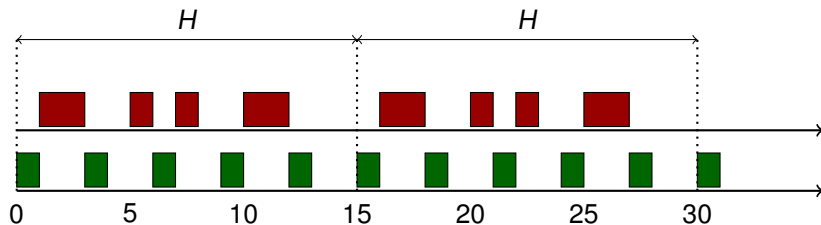
$T_3 = (0, 10, 3, 10)$ satisfies $\varphi = 0$, $p_i = 10$, $e_i = 3$, $D_i = 10$, i.e. jobs of T_3 are

- ▶ released at times 0, 10, 20, ...,
- ▶ execute for 3 time units,
- ▶ have to be finished in 10 time units (the first by 10, the second by 20, ...)

Periodic Tasks – Hyperperiod

The *hyper-period* H of a set of periodic tasks is the least common multiple of their periods

If tasks are in phase, then H is the time instant after which the pattern of job release/execution times starts to repeat



Aperiodic and Sporadic Tasks

- ▶ Many real-time systems are required to respond to external events
- ▶ The tasks resulting from such events are *sporadic* and *aperiodic* tasks
 - ▶ *Sporadic* tasks – hard deadlines of jobs
e.g. autopilot on/off in aircraft
 - ▶ *Aperiodic* tasks – soft deadlines of jobs
e.g. sensitivity adjustment of radar surveillance system
- ▶ Inter-arrival times between consecutive jobs are identically and independently distributed according to a probability distribution $A(x)$
- ▶ Execution times of jobs are identically and independently distributed according to a probability distribution $B(x)$
- ▶ In the case of sporadic tasks, the usual goal is to decide, whether a newly released job can be feasibly scheduled with the remaining jobs in the system
- ▶ In the case of aperiodic tasks, the usual goal is to minimize the average response time

Scheduling – Classification of Algorithms

- ▶ Off-line vs Online
 - ▶ Off-line – sched. algorithm is executed on the whole task set before activation
 - ▶ Online – schedule is updated at runtime every time a new task enters the system
- ▶ Optimal vs Heuristic
 - ▶ Optimal – algorithm computes a feasible schedule and minimizes cost of soft real-time jobs
 - ▶ Heuristic – algorithm is guided by heuristic function; tends towards optimal schedule, may not give one

The main division is on

- ▶ Clock-Driven
- ▶ Priority-Driven

Scheduling – Clock-Driven

- ▶ Decisions about what jobs execute when are made at specific time instants
 - ▶ these instants are chosen before the system begins execution
 - ▶ Usually regularly spaced, implemented using a periodic timer interrupt
 - ▶ Scheduler awakes after each interrupt, schedules jobs to execute for the next period, then blocks itself until the next interrupt
E.g. the helicopter example with the interrupt every $1/180$ th of a second
- ▶ Typically in clock-driven systems:
 - ▶ All parameters of the real-time jobs are fixed and known
 - ▶ A schedule of the jobs is computed off-line and is stored for use at runtime; thus scheduling overhead at run-time can be minimized
 - ▶ Simple and straight-forward, not flexible

Scheduling – Priority-Driven

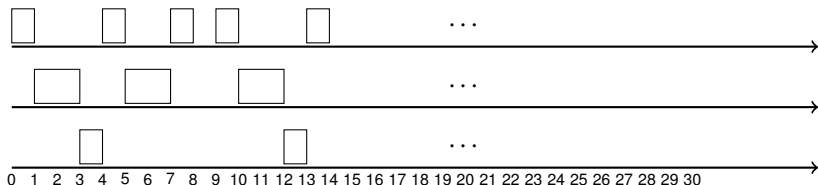
- ▶ Assign priorities to jobs, based on some algorithm
- ▶ Make scheduling decisions based on the priorities, when events such as releases and job completions occur
 - ▶ Priority scheduling algorithms are *event-driven*
 - ▶ Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed

(The assignment of jobs to priority queues, along with rules such as whether preemption is allowed, completely defines a priority-driven alg.)
- ▶ Priority-driven algs. make *locally optimal* scheduling decisions
 - ▶ Locally optimal scheduling is often *not* globally optimal
 - ▶ Priority-driven algorithms *never* intentionally leave idle processors
- ▶ Typically in priority-driven systems:
 - ▶ Some parameters do not have to be fixed or known
 - ▶ A schedule is computed online; usually results in larger scheduling overhead as opposed to clock-driven scheduling
 - ▶ Flexible – easy to add/remove tasks or modify parameters

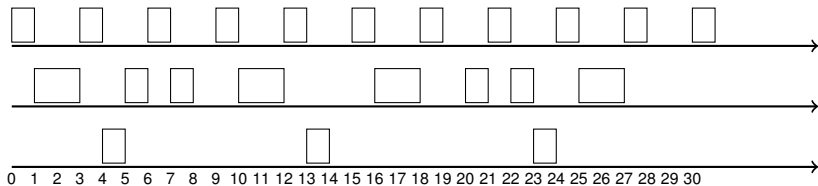
Clock-Driven & Priority-Driven Example

	T_1	T_2	T_3
p_i	3	5	10
e_i	1	2	1

Clock-Driven:



Priority-driven: $T_1 > T_2 > T_3$



Real-Time Scheduling

Scheduling of Reactive Systems

Clock-Driven Scheduling

Current Assumptions

- ▶ Fixed number, n , of periodic tasks T_1, \dots, T_n
- ▶ Parameters of periodic tasks are known a priori
 - ▶ Execution time $e_{i,k}$ of each job $J_{i,k}$ in a task T_i is fixed
 - ▶ For a job $J_{i,k}$ in a task T_i we have
 - ▶ $r_{i,1} = \varphi_i = 0$ (i.e., synchronized)
 - ▶ $r_{i,k} = r_{i,k-1} + p_i$
- ▶ We allow aperiodic jobs
 - ▶ assume that the system maintains a single queue for aperiodic jobs
 - ▶ Whenever the processor is available for aperiodic jobs, the job at the head of this queue is executed
- ▶ We treat sporadic jobs later

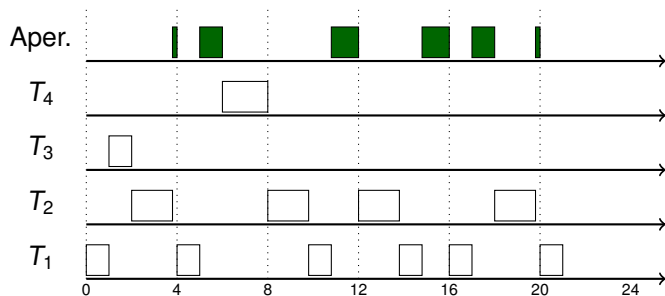
Static, Clock-Driven Scheduler

- ▶ Construct a *static schedule* offline
 - ▶ The schedule specifies exactly when each job executes
 - ▶ The amount of time allocated to every job is equal to its execution time
 - ▶ The schedule repeats each hyperperiod
i.e. it suffices to compute the schedule up to hyperperiod
- ▶ Can use complex algorithms offline
 - ▶ Runtime of the scheduling algorithm is not relevant
 - ▶ Can compute a schedule that optimizes some characteristics of the system
e.g. a schedule where the idle periods are nearly periodic (useful to accommodate aperiodic jobs)

Example

$$T_1 = (4, 1), T_2 = (5, 1.8), T_3 = (20, 1), T_4 = (20, 2)$$

Hyperperiod $H = 20$



Implementation of Static Scheduler

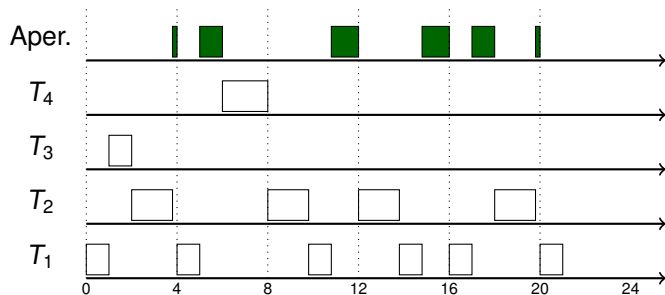
- ▶ Store pre-computed schedule as a table
 - ▶ Each entry $(t_k, T(t_k))$ gives
 - ▶ a decision time t_k
 - ▶ scheduling decision $T(t_k)$ which is either a task to be executed, or idle (denoted by I)
- ▶ The system creates all tasks that are to be executed:
 - ▶ Allocates memory for the code and data
 - ▶ Brings the code into memory
- ▶ Scheduler sets the hardware timer to interrupt at the first decision time $t_0 = 0$
- ▶ On receipt of an interrupt at t_k :
 - ▶ Scheduler sets the timer interrupt to t_{k+1}
 - ▶ If previous task overrunning, handle failure
 - ▶ If $T(t_k) = I$ and aperiodic job waiting, start executing it
 - ▶ Otherwise, start executing the next job in $T(t_k)$

k	t_k	$T(t_k)$
0	0.0	T_1
1	1.0	T_3
2	2.0	T_2
3	3.8	I
4	4.0	T_1
5	5.0	I
6	6.0	T_4
7	8.0	T_2
8	9.8	T_1
9	10.8	I
10	12.0	T_2
11	13.8	T_1
12	14.8	I
13	17.0	T_1
14	17.0	I
15	18.0	T_2
16	19.8	I

Example

$$T_1 = (4, 1), T_2 = (5, 1.8), T_3 = (20, 1), T_4 = (20, 2)$$

Hyperperiod $H = 20$



t_k	0.0	1.0	2.0	3.8	4.0	5.0	6.0	...
$T(t_k)$	T_1	T_3	T_2	I	T_1	I	T_4	...

Frame Based Scheduling

- ▶ Arbitrary table-driven cyclic schedules flexible, but inefficient
 - ▶ Relies on accurate timer interrupts, based on execution times of tasks
 - ▶ High scheduling overhead
- ▶ Easier to implement if a structure is imposed
 - ▶ Make scheduling decisions at periodic intervals (*frames*) of length f
 - ▶ Execute a fixed list of jobs within each frame;
no preemption within frames
- ▶ Gives two benefits:
 - ▶ Scheduler can easily check for overruns and missed deadlines at the end of each frame.
 - ▶ Can use a periodic clock interrupt, rather than programmable timer.

Frame Based Scheduling – Cyclic Executive

- ▶ Modify previous table-driven scheduler to be frame based
- ▶ Table that drives the scheduler has F entries, where $F = H/f$
 - ▶ The k -th entry $L(k)$ lists the names of the jobs that are to be scheduled in frame k ($L(k)$ is called *scheduling block*)
 - ▶ Each job is implemented by a procedure
- ▶ Cyclic executive executed by the clock interrupt that signals the start of a frame:
 - ▶ If an aperiodic job is executing, preempts it; if a periodic overruns, handles the overrun
 - ▶ Determines the appropriate scheduling block for this frame
 - ▶ Executes the jobs in the scheduling block
 - ▶ Executes jobs from the head of the aperiodic job queue for the remainder of the frame
- ▶ Less overhead than pure table driven cyclic scheduler, since only interrupted on frame boundaries, rather than on each job

Frame Based Scheduling – Frame Size

How to choose the frame length?

(Assume that periods are in \mathbb{N} and choose frame sizes in \mathbb{N} .)

1. Necessary condition for avoiding preemption of jobs is

$$f \geq \max_i e_i$$

(i.e. we want each job to have a chance to finish within a frame)

2. To minimize the number of entries in the cyclic schedule, the hyper-period should be an integer multiple of the frame size, i.e.

$$\exists i : p_i \bmod f = 0$$

3. To allow scheduler to check that jobs complete by their deadline, at least one frame should lie between release time of a job and its deadline, which is equivalent to

$$\forall i : 2 * f - \gcd(p_i, f) \leq D_i$$

All three constraints should be satisfied.

Frame Based Scheduling – Frame Size – Example

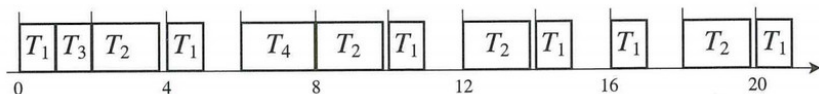
1. $f \geq \max_i e_i$
2. $\exists i : p_i \bmod f = 0$
3. $\forall i : 2 * f - \gcd(p_i, f) \leq D_i$

Example 8

$T_1 = (4, 1.0)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1.0)$, $T_4 = (20, 2.0)$

Then $f \in \mathbb{N}$ satisfies 1.–3. iff $f = 2$.

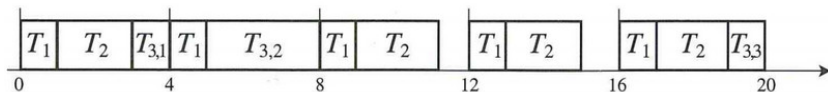
With $f = 2$ is schedulable:



Frame Based Scheduling – Job Slices

- ▶ Sometimes a system cannot meet all three frame size constraints simultaneously (and even if it meets the constraints, no non-preemptive schedule is feasible)
- ▶ Can be solved by partitioning a job with large execution time into slices with shorter execution times
This, in effect, allows preemption of the large job

- ▶ Consider $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$, $T_3 = (20, 5)$
- ▶ Cannot satisfy constraints: 1. $\Rightarrow f \geq 5$ but 3. $\Rightarrow f \leq 4$
- ▶ Solve by splitting T_3 into $T_{3,1} = (20, 1)$, $T_{3,2} = (20, 3)$, and $T_{3,3} = (20, 1)$
(Other splits exist)
- ▶ Result can be scheduled with $f = 4$



Building a Structured Cyclic Schedule

To construct a schedule, we have to make three kinds of design decisions (that cannot be taken independently):

- ▶ Choose a frame size based on constraints
- ▶ Partition jobs into slices
- ▶ Place slices into frames

There are efficient algorithms for solving these problems based e.g. on a reduction to the network flow problem.

Scheduling Aperiodic Jobs

So far, aperiodic jobs scheduled in the background after all jobs with hard deadlines

This may unnecessarily delay aperiodic jobs

Note: There is no advantage in completing periodic jobs early
Ideally, finish periodic jobs by their respective deadlines.

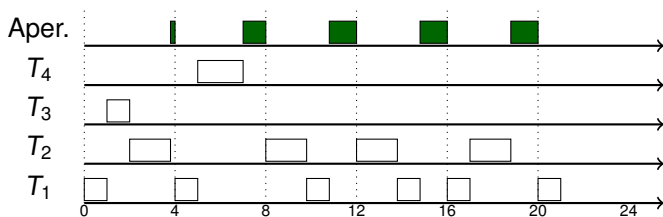
Slack Stealing:

- ▶ Slack time in a frame = the time left in the frame after all (remaining) slices execute
- ▶ Schedule aperiodic jobs ahead of periodic in the slack time of periodic jobs
 - ▶ The cyclic executive keeps track of the slack time left in each frame as the aperiodic jobs execute, preempts them with periodic jobs when there is no more slack
 - ▶ As long as there is slack remaining in a frame and the aperiodic jobs queue is non-empty, the executive executes aperiodic jobs, otherwise executes periodic
- ▶ Reduces resp. time for aper. jobs, but requires accurate timers

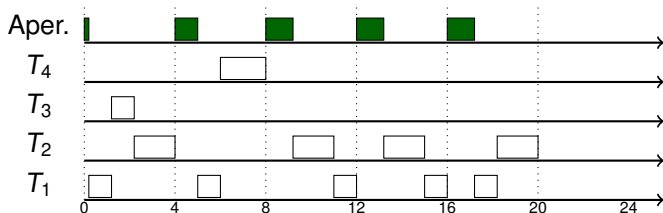
Example

Assume that the aperiodic queue is never empty.

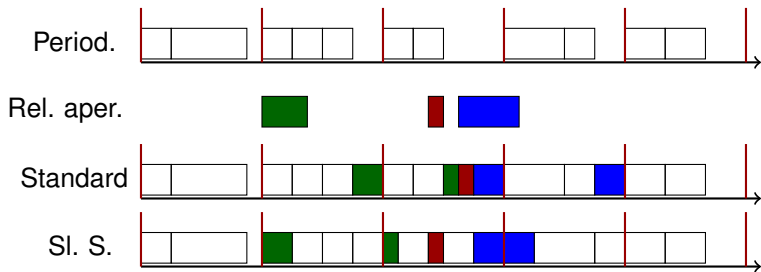
Aperiodic at the ends of frames:



Slack stealing:



Slack Stealing – cont.



Frame Based Scheduling – Sporadic Jobs

Let us allow **sporadic jobs**

i.e. hard real-time jobs whose release and exec. times are not known a priori

The scheduler determines whether to accept a sporadic job when it arrives (and its parameters become known)

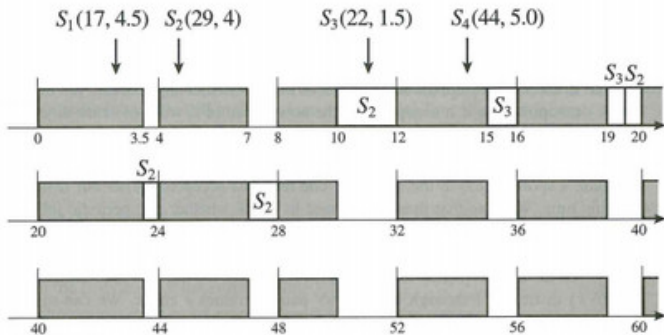
- ▶ Perform **acceptance test** to check whether the new sporadic job can be feasibly scheduled with all the jobs (periodic and sporadic) in the system at that time

Acceptance check done at the beginning of the next frame; has to keep execution times of the parts of sporadic jobs that have already executed

- ▶ If there is sufficient slack time in the frames before the new job's deadline, the new sporadic job is accepted; otherwise, rejected
- ▶ Among themselves, sporadic jobs scheduled according to EDF
This is optimal for sporadic jobs

Note: rejection is often better than missing deadline

e.g. a robotic arm taking defective parts off a conveyor belt: if the arm cannot meet deadline, the belt may be slowed down or stopped



- ▶ $S_1(17, 4.5)$ released at 3 with abs. deadline 17 and execution time 4.5; acceptance test at 4; must be scheduled in frames 2, 3, 4; total slack in these frames is 4, i.e. rejected
- ▶ $S_2(29, 4)$ released at 5 with abs. deadline 29 and exec. time 4; acc. test at 8; total slack in frames 3-7 is 5.5, i.e. accepted
- ▶ $S_3(22, 1.5)$ released at 11 with abs. deadline 22 and exec. time 1.5; acc. test at 12;
2 units of slack in frames 4, 5 as S_3 will be executed *ahead of the remaining parts of S_2* by EDF – check whether there will be enough slack for the remaining parts of S_2 , accepted
- ▶ $S_4(44, 5.0)$ is rejected (only 4.5 slack left)

Handling Overruns

Overruns may happen due to failures

e.g. unexpectedly large data over which the system operates, hardware failures, etc.

Ways to handle overruns:

- ▶ Abort the overrun job at the beginning of the next frame; log the failure; recover later
e.g. control law computation of a robust digital controller
- ▶ Preempt the overrun job and finish it as an aperiodic job
use this when aborting job would cause “costly” inconsistencies
- ▶ Let the overrun job finish – start of the next frame and the execution jobs scheduled for this frame are delayed

This may cause other jobs to be delayed
depends on application

Clock-drive Scheduling: Conclusions

Advantages:

- ▶ Conceptual simplicity
 - ▶ Complex dependencies, communication delays, and resource contention among jobs can be considered when constructing the static schedule
 - ▶ Entire schedule in a static table
 - ▶ No concurrency control or synchronization needed
- ▶ Easy to validate, test and certify

Disadvantages:

- ▶ Inflexible
 - ▶ If any parameter changes, the schedule must be usually recomputed
Best suited for systems which are rarely modified (e.g. controllers)
 - ▶ Parameters of the jobs must be fixed
As opposed to most priority-driven schedulers

Real-Time Scheduling

Scheduling of Reactive Systems

Priority-Driven Scheduling

Current Assumptions

- ▶ Single processor
- ▶ Fixed number, n , of *independent periodic* tasks
i.e. there is no dependency relation among jobs
 - ▶ Jobs can be preempted at any time and never suspend themselves
 - ▶ No aperiodic and sporadic jobs
 - ▶ No resource contentions

Moreover, unless otherwise stated, we assume that

- ▶ **Scheduling decisions take place precisely at**
 - ▶ release of a job
 - ▶ completion of a job

(and nowhere else)

- ▶ Context switch overhead is negligibly small
i.e. assumed to be zero
- ▶ There is an unlimited number of priority levels

Fixed-Priority vs Dynamic-Priority Algorithms

A priority-driven scheduler is on-line

i.e. it does not precompute a schedule of the tasks

- ▶ It assigns priorities to jobs after they are released and places the jobs in a ready job queue in the priority order with the highest priority jobs at the head of the queue
- ▶ At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue
i.e. one of the jobs with the highest priority

Fixed-priority = *all jobs in a task* are assigned the same priority

Dynamic-priority = jobs in a task may be assigned different priorities

Note: In our case, a priority assigned to a job does not change. There are *job-level dynamic priority* algorithms that vary priorities of individual jobs – we won't consider such algorithms.

Fixed-priority Algorithms – Rate Monotonic

Best known fixed-priority algorithm is *rate monotonic (RM)* scheduling that assigns priorities to tasks based on their periods

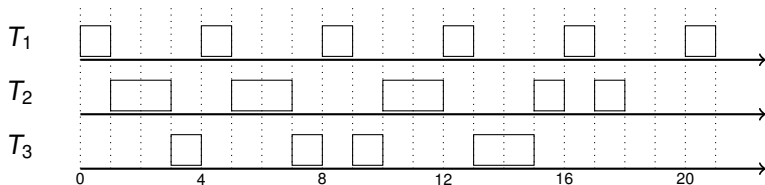
- ▶ The shorter the period, the higher the priority
- ▶ The *rate* is the inverse of the period, so jobs with higher rate have higher priority

RM is very widely studied and used

Example 9

$T_1 = (4, 1)$, $T_2 = (5, 2)$, $T_3 = (20, 5)$
with rates $1/4$, $1/5$, $1/20$, respectively

The priorities: $T_1 > T_2 > T_3$



Fixed-priority Algorithms – Deadline Monotonic

The *deadline monotonic (DM)* algorithm assigns priorities to tasks based on their *relative deadlines*

- ▶ the shorter the deadline, the higher the priority

Observation: When relative deadline of every task matches its period, then RM and DM give the same results

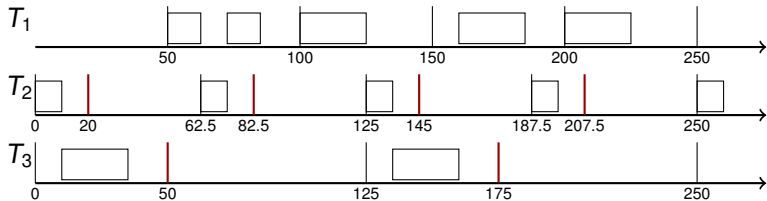
Proposition 1

When the relative deadlines are arbitrary DM can sometimes produce a feasible schedule in cases where RM cannot.

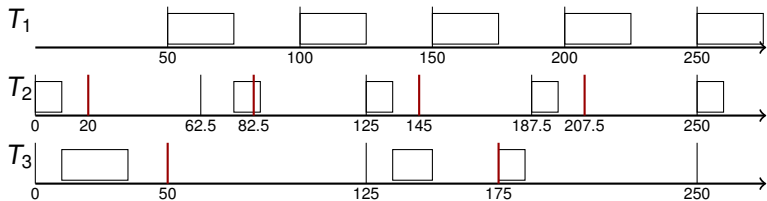
Rate Monotonic vs Deadline Monotonic

$$T_1 = (50, 50, 25, 100), T_2 = (0, 62.5, 10, 20), T_3 = (0, 125, 25, 50)$$

DM is optimal (with priorities $T_2 > T_3 > T_1$):



RM is not optimal (with priorities $T_1 > T_2 > T_3$):



Dynamic-priority Algorithms

Best known is *earliest deadline first (EDF)* that assigns priorities based on *current* (absolute) deadlines

- ▶ At the time of a scheduling decision, the job queue is ordered by earliest deadline

Another one is the *least slack time (LST)*

- ▶ The job queue is ordered by least slack time

Recall that the *slack time* of a job J_i at time t is equal to $d_i - t - x$ where x is the remaining computation time of J_i at time t

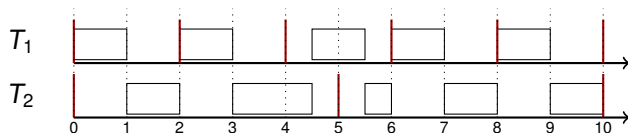
Comments:

- ▶ There is also a strict LST which reassigns priorities to jobs whenever their slacks change relative to each other – won't consider
- ▶ Standard “non real-time” algorithms such as FIFO and LIFO are also dynamic-priority algorithms

We focus on EDF here, leave some LST for homework

EDF – Example

$T_1 = (2, 1)$ and $T_2 = (5, 2.5)$



Note that the processor is 100% “utilized”, not surprising :-)

Summary of Priority-Driven Algorithms

We consider:

Dynamic-priority:

- ▶ **EDF** = at the time of a scheduling decision, the job queue is ordered by the earliest deadline

Fixed-priority:

- ▶ **RM** = assigns priorities to tasks based on their periods
- ▶ **DM** = assigns priorities to tasks based on their relative deadlines

(In all cases, ties are broken arbitrarily.)

We consider the following questions:

- ▶ Are the algorithms optimal?
- ▶ How to efficiently (or even online) test for schedulability?

To measure abilities of scheduling algorithms and to get fast online tests of schedulability we use a notion of **utilization**

Utilization

- ▶ *Utilization u_i of a periodic task T_i* with period p_i and execution time e_i is defined by $u_i := e_i/p_i$
 u_i is the fraction of time a periodic task with period p_i and execution time e_i keeps a processor busy
- ▶ *Total utilization $U^{\mathcal{T}}$ of a set of tasks $\mathcal{T} = \{T_1, \dots, T_n\}$* is defined as the sum of utilizations of all tasks of \mathcal{T} , i.e. by

$$U^{\mathcal{T}} := \sum_{i=1}^n u_i$$

- ▶ U is a *schedulable utilization* of an algorithm ALG if all sets of tasks \mathcal{T} satisfying $U^{\mathcal{T}} \leq U$ are schedulable by ALG.
Maximum schedulable utilization U_{ALG} of an algorithm ALG is the *supremum of schedulable utilizations of ALG*.
 - ▶ If $U^{\mathcal{T}} < U_{\text{ALG}}$, then \mathcal{T} is schedulable by ALG.
 - ▶ If $U > U_{\text{ALG}}$, then there is \mathcal{T} with $U^{\mathcal{T}} \leq U$ that is not schedulable by ALG.

Utilization – Example

- ▶ $T_1 = (2, 1)$ then $u_1 = \frac{1}{2}$
- ▶ $T_1 = (11, 5, 2, 4)$ then $u_1 = \frac{2}{5}$
(i.e., the phase and deadline do not play any role)
- ▶ $\mathcal{T} = \{T_1, T_2, T_3\}$ where $T_1 = (2, 1)$, $T_2 = (6, 1)$, $T_3 = (8, 3)$
then

$$U^{\mathcal{T}} = \frac{1}{2} + \frac{1}{6} + \frac{3}{8} = \frac{25}{24}$$

Real-Time Scheduling

Priority-Driven Scheduling

Dynamic-Priority

Optimality of EDF

Theorem 10

Let $\mathcal{T} = \{T_1, \dots, T_n\}$ be a set of independent, preemptable periodic tasks with $D_i \geq p_i$ for $i = 1, \dots, n$. The following statements are equivalent:

1. \mathcal{T} can be feasibly scheduled on one processor
2. $U^{\mathcal{T}} \leq 1$
3. \mathcal{T} is schedulable using EDF

(i.e., in particular, $U_{EDF} = 1$)

Proof.

1. \Rightarrow 2. We prove that $U^{\mathcal{T}} > 1$ implies that \mathcal{T} is not schedulable
2. \Rightarrow 3. Next slides and whiteboard ...
3. \Rightarrow 1. Trivial



Proof of 1. \Rightarrow 2.

Assume that $U^{\mathcal{T}} = \sum_{i=1}^N \frac{e_i}{p_i} > 1$.

Consider a time instant $t > \max_i \varphi_i$

(i.e. a time when all tasks are already "running")

Observe that the number of jobs of T_i that are released in the time interval $[0, t]$ is $\left\lceil \frac{t - \varphi_i}{p_i} \right\rceil$. Thus a single processor needs $\sum_{i=1}^n \left\lceil \frac{t - \varphi_i}{p_i} \right\rceil \cdot e_i$ time units to finish all jobs *released before or at t* .

However,

$$\sum_{i=1}^n \left\lceil \frac{t - \varphi_i}{p_i} \right\rceil \cdot e_i \geq \sum_{i=1}^n (t - \varphi_i) \cdot \frac{e_i}{p_i} = \sum_{i=1}^n t u_i - \varphi_i u_i = \sum_{i=1}^n t u_i - \sum_{i=1}^n \varphi_i u_i = t \cdot U^{\mathcal{T}} - \sum_{i=1}^n \varphi_i u_i$$

Here $\sum_{i=1}^n \varphi_i u_i$ does not depend on t .

Note that $\lim_{t \rightarrow \infty} (t \cdot U^{\mathcal{T}} - \sum_{i=1}^n \varphi_i u_i) - t = \infty$. So there exists t such that $t \cdot U^{\mathcal{T}} - \sum_{i=1}^n \varphi_i u_i > t + \max_i D_i$.

So in order to complete all jobs released before time t we need more time than $t + \max_i D_i$. However, the latest deadline of a job released before t is $t + \max_i D_i$. So at least one job misses its deadline.

Proof of 2. \Rightarrow 3. – Simplified

Let us start with a proof of a special case (see the assumptions A1 and A2 below). Then a complete proof will be presented.

We prove $\neg 3. \Rightarrow \neg 2.$ assuming that $D_i = p_i$ for $i = 1, \dots, n.$

(Note that the general case immediately follows.)

Assume that \mathcal{T} is not schedulable according to EDF.

(Our goal is to show that $U^{\mathcal{T}} > 1.$)

This means that there must be at least one job that misses its deadline when EDF is used.

Simplifying assumptions:

A1 Suppose that all tasks are in phase, i.e. the phase $\varphi_\ell = 0$ for every task $T_\ell.$

A2 Suppose that *the first job* $J_{i,1}$ of a task T_i misses its deadline.

By A1, $J_{i,1}$ is released at 0 and misses its deadline at $p_i.$ Assume w.l.o.g. that this is the first time when a job misses its deadline.

(To simplify even further, you may (privately) assume that no other job has its deadline at $p_i.$)

Proof of 2. \Rightarrow 3. – Simplified

Let G be the set of all jobs that are released in $[0, p_i]$ and have their deadlines in $[0, p_i]$.

Crucial observations:

- ▶ G contains $J_{i,1}$ and all jobs that preempt $J_{i,1}$.
(If there are more jobs with deadline p_i , then these jobs do not have to preempt $J_{i,1}$. Assume the worst case: all these jobs preempt $J_{i,1}$.)
- ▶ The processor is never idle during $[0, p_i]$ and executes *only* jobs of G .

Denote by E_G the total execution time of G , that is, the sum of execution times of all jobs in G .

Corollary of the crucial observation: $E_G > p_i$ because otherwise $J_{i,1}$ (and all jobs that preempt it) would complete by p_i .

Let us compute E_G .

Proof of 2. \Rightarrow 3. – Simplified

Since we assume $\varphi_\ell = 0$ for every T_ℓ , the first job of T_ℓ is released at 0, and thus $\lfloor \frac{p_i}{p_\ell} \rfloor$ jobs of T_ℓ belong to G .

E.g., if $p_\ell = 2$ and $p_i = 5$ then three jobs of T_ℓ are released in $[0, 5]$ (at times 0, 2, 4) but only $2 = \lfloor \frac{5}{2} \rfloor = \lfloor \frac{p_i}{p_\ell} \rfloor$ of them have their deadlines in $[0, p_i]$.

Thus the total execution time E_G of all jobs in G is

$$E_G = \sum_{\ell=1}^n \left\lfloor \frac{p_i}{p_\ell} \right\rfloor e_\ell$$

But then

$$p_i < E_G = \sum_{\ell=1}^n \left\lfloor \frac{p_i}{p_\ell} \right\rfloor e_\ell \leq \sum_{\ell=1}^n \frac{p_i}{p_\ell} e_\ell \leq p_i \sum_{\ell=1}^n u_\ell \leq p_i \cdot U^T$$

which implies that $U^T > 1$.

Proof of 2. \Rightarrow 3. – Complete

Now let us drop the simplifying assumptions A1 and A2 !

Notation: Given a set of tasks \mathcal{L} , we denote by $\cup \mathcal{L}$ the set of all jobs of the tasks in \mathcal{L} .

We prove $\neg 3. \Rightarrow \neg 2.$ assuming that $D_i = p_i$ for $i = 1, \dots, n$ (note that the general case immediately follows).

Assume that \mathcal{T} is not schedulable by EDF. We show that $U^{\mathcal{T}} > 1$.

Suppose that a job $J_{i,k}$ of T_i misses its deadline at time $t = r_{i,k} + p_i$.
Assume that this is the earliest deadline miss.

Let \mathcal{T}' be the set of all tasks whose jobs have deadlines (and thus also release times) in $[r_{i,k}, t]$
(i.e., a task belongs to \mathcal{T}' iff at least one job of the task is released in $[r_{i,k}, t]$).

Let t_- be the end of the *latest* interval before t in which either jobs of $\cup(\mathcal{T} \setminus \mathcal{T}')$ are executed, or the processor is idle.

Then $r_{i,k} \geq t_-$ since all jobs of $\cup(\mathcal{T} \setminus \mathcal{T}')$ waiting for execution during $[r_{i,k}, t]$ have deadlines later than t (thus have lower priorities than $J_{i,k}$).

Proof of 2. \Rightarrow 3. – Complete (cont.)

It follows that

- ▶ no job of $\cup(\mathcal{T} \setminus \mathcal{T}')$ is executed in $[t_-, t]$,
(by definition of t_-)
- ▶ the processor is fully utilized in $[t_-, t]$.
(by definition of t_-)
- ▶ *all jobs* (that all must belong to $\cup \mathcal{T}'$) *executed in $[t_-, t]$ are released in $[t_-, t]$ and have their deadlines in $[t_-, t]$* since
 - ▶ no job of $\cup \mathcal{T}'$ executes just before t_- ,
 - ▶ all jobs of $\cup \mathcal{T}'$ released in $[t_-, r_{i,k}]$ have deadlines before t ,
 - ▶ jobs of $\cup \mathcal{T}'$ released in $[r_{i,k}, t]$ with deadlines after t are not executed in $[r_{i,k}, t]$ as they have lower priorities than $J_{i,k}$.

Let G be the set of all jobs that are released in $[t_-, t]$ and have their deadlines in $[t_-, t]$.

Note that $J_{i,k} \in G$ since $r_{i,k} \geq t_-$.

Denote by E_G the sum of all execution times of all jobs in G (the total execution time of G).

Proof of 2. \Rightarrow 3. – Complete (cont.)

Now $E_G > t - t_-$ because otherwise $J_{i,k}$ would complete in $[t_-, t]$.

How to compute E_G ?

For $T_\ell \in \mathcal{T}'$, denote by R_ℓ the earliest release time of a job in T_ℓ during the interval $[t_-, t]$.

For every $T_\ell \in \mathcal{T}'$, exactly $\left\lfloor \frac{t - R_\ell}{p_\ell} \right\rfloor$ jobs of T_ℓ belong to G . (For every $T_\ell \in \mathcal{T} \setminus \mathcal{T}'$, exactly 0 jobs belong to G .)

Thus

$$E_G = \sum_{T_\ell \in \mathcal{T}'} \left\lfloor \frac{t - R_\ell}{p_\ell} \right\rfloor e_\ell$$

As argued above:

$$t - t_- < E_G = \sum_{T_\ell \in \mathcal{T}'} \left\lfloor \frac{t - R_\ell}{p_\ell} \right\rfloor e_\ell \leq \sum_{T_\ell \in \mathcal{T}'} \frac{t - t_-}{p_\ell} e_\ell \leq (t - t_-) \sum_{T_\ell \in \mathcal{T}'} u_\ell \leq (t - t_-) U^{\mathcal{T}}$$

which implies that $U^{\mathcal{T}} > 1$.

Density and EDF

What about tasks with $D_i < p_i$?

Density of a task T_i with period p_i , execution time e_i and relative deadline D_i is defined by

$$e_i / \min(D_i, p_i)$$

Total density $\Delta^{\mathcal{T}}$ of a set of tasks \mathcal{T} is the sum of densities of tasks in \mathcal{T}

Note that if $D_i < p_i$ for some i , then $\Delta^{\mathcal{T}} > U^{\mathcal{T}}$

Theorem 11

A set \mathcal{T} of independent, preemptable, periodic tasks can be feasibly scheduled on one processor if $\Delta^{\mathcal{T}} \leq 1$.

Note that this is NOT a necessary condition! (Example whiteb.)

Schedulability Test For EDF

The problem: Given a set of independent, preemptable, periodic tasks $\mathcal{T} = \{T_1, \dots, T_n\}$ where each T_i has a period p_i , execution time e_i , and relative deadline D_i , decide whether \mathcal{T} is schedulable by EDF.

Solution using utilization and density:

If $p_i \leq D_i$ for each i , then it suffices to decide whether $U^{\mathcal{T}} \leq 1$.

Otherwise, decide whether $\Delta^{\mathcal{T}} \leq 1$:

- ▶ If yes, then \mathcal{T} is schedulable with EDF
- ▶ If not, then \mathcal{T} does not have to be schedulable

Note that

- ▶ Phases of tasks do not have to be specified
- ▶ Parameters may vary: increasing periods or deadlines, or decreasing execution times does not prevent schedulability

Schedulability Test for EDF – Example

Consider a digital robot controller

- ▶ A control-law computation
 - ▶ takes no more than 8 ms
 - ▶ the sampling rate: 100 Hz, i.e. computes every 10 ms

Feasible? Trivially yes

- ▶ Add Built-In Self-Test (BIST)
 - ▶ maximum execution time 50 ms
 - ▶ want a minimal period that is feasible (max one second)

With 250 ms still feasible

- ▶ Add a telemetry task
 - ▶ maximum execution time 15 ms
 - ▶ want to minimize the deadline on telemetry
period may be large

Reducing BIST to once a second, deadline on telemetry
may be set to 100 ms

Real-Time Scheduling

Priority-Driven Scheduling

Fixed-Priority

Fixed-Priority Algorithms

Recall that we consider a set of n tasks $\mathcal{T} = \{T_1, \dots, T_n\}$

Any fixed-priority algorithm schedules tasks of \mathcal{T} according to fixed (distinct) priorities *assigned to tasks*.

We write $T_i \sqsupset T_j$ whenever T_i has a higher priority than T_j .

To simplify our reasoning, assume that

all tasks are in phase, i.e. $\varphi_k = 0$ for all T_k .

We will remove this assumption at the end.

Fixed-Priority Algorithms – Reminder

Recall that Fixed-Priority Algorithms do not have to be optimal.

Consider $\mathcal{T} = \{T_1, T_2\}$ where $T_1 = (2, 1)$ and $T_2 = (5, 2.5)$

$U^{\mathcal{T}} = 1$ and thus \mathcal{T} is schedulable by EDF

If $T_1 \sqsupset T_2$, then $J_{2,1}$ misses its deadline

If $T_2 \sqsupset T_1$, then $J_{1,1}$ misses its deadline

We consider the following algorithms:

- ▶ **RM** = assigns priorities to tasks based on their periods
the priority is inversely proportional to the period p_i
- ▶ **DM** = assigns priorities to tasks based on their relative deadlines
the priority is inversely proportional to the relative deadline D_i

(In all cases, ties are broken arbitrarily.)

We consider the following questions:

- ▶ Are the algorithms optimal?
- ▶ How to efficiently (or even online) test for schedulability?

Maximum Response Time

Which job of a task T_i has the maximum response time?

As all tasks are in phase, the first job of T_i is released together with (first) jobs of all tasks that have higher priority than T_i .

This means, that $J_{i,1}$ is the most preempted of jobs in T_i .

It follows, that $J_{i,1}$ has the maximum response time.

Note that this relies heavily on the assumption that tasks are in phase!

Thus in order to decide whether \mathcal{T} is schedulable, it suffices to test for schedulability of the first jobs of all tasks.

Optimality of RM for Simply Periodic Tasks

Definition 12

A set $\{T_1, \dots, T_n\}$ is **simply periodic** if for every pair T_i, T_ℓ satisfying $p_i > p_\ell$ we have that p_i is an integer multiple of p_ℓ

Example 13

The helicopter control system from the first lecture.

Theorem 14

A set \mathcal{T} of n simply periodic, independent, preemptable tasks with $D_i = p_i$ is schedulable on one processor according to RM iff $U^{\mathcal{T}} \leq 1$.

i.e. on simply periodic tasks RM is as good as EDF

Note: Theorem 14 is true in general, no "in phase" assumption is needed.

Proof of Theorem 14

By Theorem 10, every schedulable set \mathcal{T} satisfies $U^{\mathcal{T}} \leq 1$.

We prove that if \mathcal{T} is **not** schedulable according to RM, then $U^{\mathcal{T}} > 1$.

Assume that a job $J_{i,1}$ of T_i misses its deadline at $D_i = p_i$. W.l.o.g., we assume that $T_1 \supset \dots \supset T_n$ according to RM.

Let us compute the total execution time of $J_{i,1}$ and all jobs that preempt it:

$$E = e_i + \sum_{\ell=i+1}^n \left\lceil \frac{p_i}{p_\ell} \right\rceil e_\ell = \sum_{\ell=i}^n \frac{p_i}{p_\ell} e_\ell = p_i \sum_{\ell=i}^n u_\ell \leq p_i \sum_{\ell=1}^n u_\ell = p_i U^{\mathcal{T}}$$

Now $E > p_i$ because otherwise $J_{i,1}$ meets its deadline. Thus

$$p_i < E \leq p_i U^{\mathcal{T}}$$

and we obtain $U^{\mathcal{T}} > 1$.

Optimality of DM (RM) among Fixed-Priority Algs.

Theorem 15

A set of independent, preemptable periodic tasks with $D_i \leq p_i$ that are in phase (i.e., $\varphi_i = 0$ for all $i = 1, \dots, n$) can be feasibly scheduled on one processor according to DM if it can be feasibly scheduled by some fixed-priority algorithm.

Proof.

Assume a fixed-priority feasible schedule with $T_1 \supset \dots \supset T_n$.

Consider the least i such that the relative deadline D_i of T_i is larger than the relative deadline D_{i+1} of T_{i+1} .

Swap the priorities of T_i and T_{i+1} .

The resulting schedule is still feasible.

DM is obtained by using finitely many swaps. □

Note: If the assumptions of the above theorem hold and all relative deadlines are equal to periods, then RM is optimal among all fixed-priority algorithms.

Note: no "in phase" assumption is needed here.

Fixed-Priority Algorithms: Schedulability

We consider two schedulability tests:

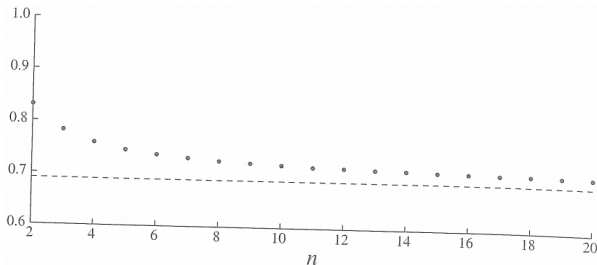
- ▶ Schedulable utilization U_{RM} of the RM algorithm.
- ▶ Time-demand analysis based on response times.

Schedulable Utilization for RM

Theorem 16

Let us fix $n \in \mathbb{N}$ and consider only independent, preemptable periodic tasks with $D_i = p_i$.

- ▶ If \mathcal{T} is a set of n tasks satisfying $U^{\mathcal{T}} \leq n(2^{1/n} - 1)$, then $U^{\mathcal{T}}$ is schedulable according to the RM algorithm.
- ▶ For every $U > n(2^{1/n} - 1)$ there is a set \mathcal{T} of n tasks satisfying $U^{\mathcal{T}} \leq U$ that is not schedulable by RM.



Note: Theorem 16 holds in general, no "in phase" assumption is needed.

Schedulable Utilization for RM

It follows that the maximum schedulable utilization U_{RM} over independent, preemptable periodic tasks satisfies

$$U_{RM} = \inf_n n(2^{1/n} - 1) = \lim_{n \rightarrow \infty} n(2^{1/n} - 1) = \ln 2 \approx 0.693$$

Note that $U^{\mathcal{T}} \leq n(2^{1/n} - 1)$ is a sufficient but not necessary condition for schedulability of \mathcal{T} using the RM algorithm (an example will be given later)

We say that a set of tasks \mathcal{T} is *RM-schedulable* if it is schedulable according to RM.

We say that \mathcal{T} is *RM-infeasible* if it is not RM-schedulable.

Proof – Special Case

To simplify, we restrict to two tasks and always assume $p_2 \leq 2p_1$.
(the latter condition is w.l.o.g., proof omitted)

Outline: Given p_1, p_2, e_1 , denote by \max_{e_2} the **maximum** execution time so that $\mathcal{T} = \{(p_1, e_1), (p_2, \max_{e_2})\}$ is RM-schedulable.

We define $U_{e_1}^{p_1, p_2}$ to be $U^{\mathcal{T}}$ where $\mathcal{T} = \{(p_1, e_1), (p_2, \max_{e_2})\}$.

We say that \mathcal{T} fully utilizes the processor, any increase in an execution time causes RM-infeasibility.

Now we find the (global) minimum $\min U$ of $U_{e_1}^{p_1, p_2}$.

Note that this suffices to obtain the desired result:

- ▶ Given a set of tasks $\mathcal{T} = \{(p_1, e_1), (p_2, e_2)\}$ satisfying $U^{\mathcal{T}} \leq \min U$ we get $U^{\mathcal{T}} \leq \min U \leq U_{e_1}^{p_1, p_2}$, and thus the execution time e_2 cannot be larger than \max_{e_2} . Thus, \mathcal{T} is RM-schedulable.
- ▶ Given $U > \min U$, there must be p_1, p_2, e_1 satisfying $\min U \leq U_{e_1}^{p_1, p_2} < U$ where $U_{e_1}^{p_1, p_2} = U^{\mathcal{T}}$ for a set of tasks $\mathcal{T} = \{(p_1, e_1), (p_2, \max_{e_2})\}$.

However, now increasing e_1 by a sufficiently small $\varepsilon > 0$ makes the set RM-infeasible without making utilization larger than U .

Proof – Special Case (Cont.)

Consider two cases depending on e_1 :

1. $e_1 < p_2 - p_1$:

Maximum RM-feasible max_e_2 (with p_1, p_2, e_1 fixed) is $p_2 - 2e_1$.

Which gives the utilization

$$U_{e_1}^{p_1, p_2} = \frac{e_1}{p_1} + \frac{max_e_2}{p_2} = \frac{e_1}{p_1} + \frac{p_2 - 2e_1}{p_2} = \frac{e_1}{p_1} + \frac{p_2}{p_2} - \frac{2e_1}{p_2} = 1 + \frac{e_1}{p_2} \left(\frac{p_2}{p_1} - 2 \right)$$

As $\frac{p_2}{p_1} - 2 \leq 0$, the utilization $U_{e_1}^{p_1, p_2}$ is minimized by maximizing e_1 .

2. $e_1 \geq p_2 - p_1$:

Maximum RM-feasible max_e_2 (with p_1, p_2, e_1 fixed) is $p_1 - e_1$. Which gives the utilization

$$U_{e_1}^{p_1, p_2} = \frac{e_1}{p_1} + \frac{max_e_2}{p_2} = \frac{e_1}{p_1} + \frac{p_1 - e_1}{p_2} = \frac{e_1}{p_1} + \frac{p_1}{p_2} - \frac{e_1}{p_2} = \frac{p_1}{p_2} + \frac{e_1}{p_2} \left(\frac{p_2}{p_1} - 1 \right)$$

As $\frac{p_2}{p_1} - 1 \geq 0$, the utilization $U_{e_1}^{p_1, p_2}$ is minimized by minimizing e_1 .

The minimum of $U_{e_1}^{p_1, p_2}$ is attained at $e_1 = p_2 - p_1$.

(Both expressions defining $U_{e_1}^{p_1, p_2}$ give the same value for $e_1 = p_2 - p_1$.)

Proof – Special Case (Cont.)

Substitute $e_1 = p_2 - p_1$ into the expression for $U_{e_1}^{p_1, p_2}$:

$$\begin{aligned}U_{p_2-p_1}^{p_1, p_2} &= \frac{p_1}{p_2} + \frac{p_2 - p_1}{p_2} \left(\frac{p_2}{p_1} - 1 \right) = \frac{p_1}{p_2} + \left(1 - \frac{p_1}{p_2} \right) \left(\frac{p_2}{p_1} - 1 \right) \\ &= \frac{p_1}{p_2} + \frac{p_1}{p_2} \left(\frac{p_2}{p_1} - 1 \right) \left(\frac{p_2}{p_1} - 1 \right) = \frac{p_1}{p_2} \left(1 + \left(\frac{p_2}{p_1} - 1 \right)^2 \right)\end{aligned}$$

Denoting $G = \frac{p_2}{p_1} - 1$ we obtain

$$U_{p_2-p_1}^{p_1, p_2} = \frac{p_1}{p_2} (1 + G^2) = \frac{1 + G^2}{p_2/p_1} = \frac{1 + G^2}{1 + G}$$

Differentiating w.r.t. G we get

$$\frac{G^2 + 2G - 1}{(1 + G)^2}$$

which attains minimum at $G = -1 \pm \sqrt{2}$. Here only $G = -1 + \sqrt{2} > 0$ is acceptable since the other root is negative.

Proof – Special Case (Cont.)

Thus the **minimum value** of $U_{e_1}^{p_1, p_2}$ is

$$\frac{1 + (\sqrt{2} - 1)^2}{1 + (\sqrt{2} - 1)} = \frac{4 - 2\sqrt{2}}{\sqrt{2}} = 2(\sqrt{2} - 1)$$

It is attained at periods satisfying

$$G = \frac{p_2}{p_1} - 1 = \sqrt{2} - 1 \quad \text{i.e. satisfying } p_2 = \sqrt{2}p_1.$$

The execution time e_1 which at full utilization of the processor (due to \max_e_2) gives the minimum utilization is

$$e_1 = p_2 - p_1 = (\sqrt{2} - 1)p_1$$

and the corresponding $\max_e_2 = p_1 - e_1 = p_1 - (p_2 - p_1) = 2p_1 - p_2$.

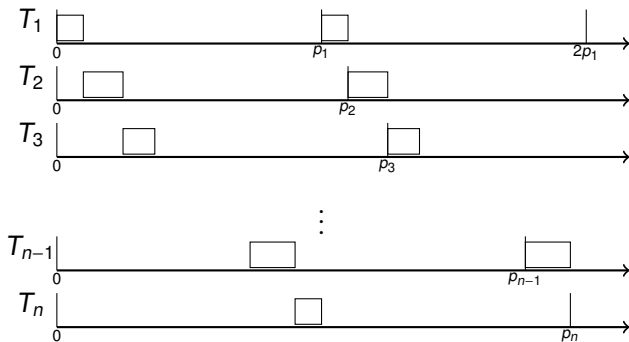
Scaling to $p_1 = 1$, we obtain a completely determined example

$$p_1 = 1 \quad p_2 = \sqrt{2} \approx 1.41 \quad e_1 = \sqrt{2} - 1 \approx 0.41 \quad \max_e_2 = 2 - \sqrt{2} \approx 0.59$$

that fully utilizes the processor (no execution time can be increased) but has the minimum utilization $2(\sqrt{2} - 1)$.

Proof Idea of Theorem 16

Fix periods $p_1 < \dots < p_n$ so that (w.l.o.g.) $p_n \leq 2p_1$. Then the following set of tasks has the smallest utilization among all task sets that fully utilize the processor (i.e., any increase in any execution time makes the set unschedulable).



$$e_k = p_{k+1} - p_k \quad \text{for } k = 1, \dots, n-1$$

$$e_n = p_n - 2 \sum_{k=1}^{n-1} e_k = 2p_1 - p_n$$

Time-Demand Analysis

Consider a set of n tasks $\mathcal{T} = \{T_1, \dots, T_n\}$.

Recall that we consider only independent, preemptable, in phase (i.e. $\varphi_i = 0$ for all i) tasks without resource contentions.

Assume that $D_i \leq p_i$ for every i , and consider an arbitrary fixed-priority algorithm. W.l.o.g. assume $T_1 \supset \dots \supset T_n$.

Idea: For every task T_i and every time instant $t \geq 0$, compute the total execution time $w_i(t)$ (the time demand) of the first job $J_{i,1}$ and of all higher-priority jobs released up to time t .

If $w_i(t) \leq t$ for some time $t \leq D_i$, then $J_{i,1}$ is schedulable, and hence all jobs of T_i are schedulable.

Time-Demand Analysis

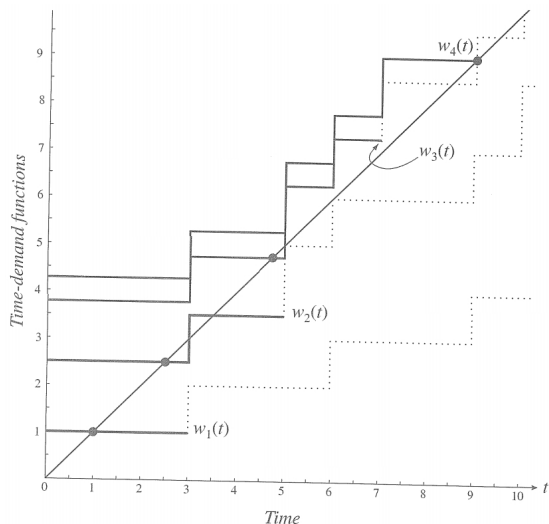
- ▶ Consider one task T_i at a time, starting with highest priority and working to lowest priority.
- ▶ Focus on the first job $J_{i,1}$ of T_i .
If $J_{i,1}$ makes it, all jobs of T_i will make it due to $\varphi_i = 0$.
- ▶ At time t for $t \geq 0$, the processor time demand $w_i(t)$ for this job and all higher-priority jobs released in $[0, t]$ is bounded by

$$w_i(t) = e_i + \sum_{\ell=1}^{i-1} \left\lceil \frac{t}{p_\ell} \right\rceil e_\ell \quad \text{for } 0 < t \leq p_i$$

(Note that the smallest t for which $w_i(t) \leq t$ is the response time of $J_{i,1}$, and hence the maximum response time of jobs in T_i).

- ▶ If $w_i(t) \leq t$ for some $t \leq D_i$, the job $J_{i,1}$ meets its deadline D_i .
- ▶ If $w_i(t) > t$ for all $0 < t \leq D_i$, then the first job of the task cannot complete by its deadline.

Time-Demand Analysis – Example



Example: $T_1 = (3, 1)$, $T_2 = (5, 1.5)$, $T_3 = (7, 1.25)$, $T_4 = (9, 0.5)$

This is schedulable by RM even though

$$U(T_1, \dots, T_4) = 0.85 > 0.757 = U_{RM}(4)$$

Time-Demand Analysis

- ▶ The time-demand function $w_i(t)$ is a staircase function
 - ▶ Steps in the time-demand for a task occur at multiples of the period for higher-priority tasks
 - ▶ The value of $w_i(t) - t$ linearly decreases from a step until the next step
- ▶ If our interest is the schedulability of a task, it suffices to check if $w_i(t) \leq t$ at the time instants when a higher-priority job is released and at D_i
- ▶ Our schedulability test becomes:
 - ▶ Compute $w_i(t)$
 - ▶ Check whether $w_i(t) \leq t$ for some t equal either to D_i , or to $j \cdot p_k$ where $k = 1, 2, \dots, i$ and $j = 1, 2, \dots, \lfloor D_i/p_k \rfloor$

Time-Demand Analysis – Comments

- ▶ Time-demand analysis schedulability test is more complex than the schedulable utilization test but more general:
 - ▶ Works for *any* fixed-priority scheduling algorithm, provided the tasks have short response time ($D_i \leq p_i$)
Can be extended to tasks with arbitrary deadlines
- ▶ Still more efficient than exhaustive simulation.
- ▶ Assuming that the tasks are in phase the time demand analysis is complete.

We have considered the time demand analysis for tasks in phase. In particular, we used the fact that the first job has the maximum response time.

This is not true if the jobs are not in phase, we need to identify the so called *critical instant*, the time instant in which the system is most loaded, and has its worst response time.

Critical Instant – Formally

Definition 17

A **critical instant** t_{crit} of a task T_i is a time instant in which a job $J_{i,k}$ in T_i is released so that $J_{i,k}$ either does not meet its deadline, or has the maximum response time of all jobs in T_i .

Theorem 18

In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant of a task T_i occurs when one of its jobs $J_{i,k}$ is released at the same time with a job from every higher-priority task.

Note that the situation described in the theorem does not have to occur if tasks are not in phase!

Critical Instant and Schedulability Tests

We use critical instants to get upper bounds on schedulability as follows:

- ▶ Set phases of all tasks to zero, which gives a new set of tasks $\mathcal{T}' = \{T'_1, \dots, T'_n\}$

By Theorem 18, the response time of the first job $J'_{i,1}$ of T'_i in \mathcal{T}' is at least as large as the response time of every job of T_i in \mathcal{T} .

- ▶ Decide schedulability of \mathcal{T}' , e.g. using the timed-demand analysis.
 - ▶ If \mathcal{T}' is schedulable, then also \mathcal{T} is schedulable.
 - ▶ If \mathcal{T}' is not schedulable, then \mathcal{T} does not have to be schedulable.

But may be schedulable, which makes the time-demand analysis incomplete in general for tasks not in phase.

Dynamic vs Fixed Priority

- ▶ EDF
 - ▶ pros:
 - ▶ optimal
 - ▶ very simple and complete test for schedulability
 - ▶ cons:
 - ▶ difficult to predict which job misses its deadline
 - ▶ strictly following EDF in case of overloads assigns higher priority to jobs that missed their deadlines
 - ▶ larger scheduling overhead
- ▶ DM (RM)
 - ▶ pros:
 - ▶ easier to predict which job misses its deadline (in particular, tasks are not blocked by lower priority tasks)
 - ▶ easy implementation with little scheduling overhead
 - ▶ (optimal in some cases often occurring in practice)
 - ▶ cons:
 - ▶ not optimal
 - ▶ incomplete and more involved tests for schedulability

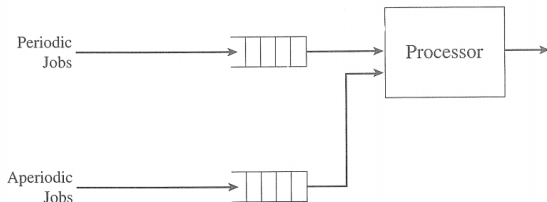
Real-Time Scheduling

Priority-Driven Scheduling

Aperiodic Tasks

Current Assumptions

- ▶ Single processor
- ▶ Fixed number, n , of *independent periodic* tasks
 - ▶ Jobs can be preempted at any time and never suspend themselves
 - ▶ No resource contentions
- ▶ Aperiodic jobs exist
 - ▶ They are independent of each other, and of the periodic tasks
 - ▶ They can be preempted at any time
- ▶ There are no sporadic jobs (for now)
- ▶ Jobs are scheduled using a priority driven algorithm



Scheduling Aperiodic Jobs

Consider:

- ▶ A set $\mathcal{T} = \{T_1, \dots, T_n\}$ of periodic tasks
- ▶ An aperiodic task A

Recall that:

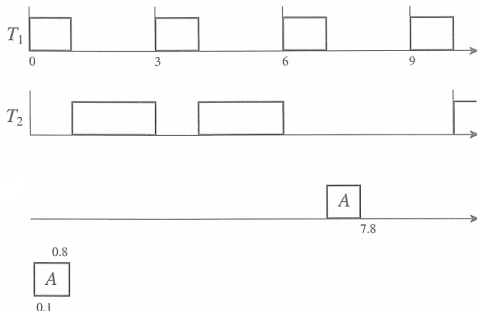
- ▶ A schedule is feasible if all jobs with hard real-time constraints complete before their deadlines
⇒ This includes all periodic jobs
- ▶ A scheduling algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists, and if a cost function is given, minimizes the cost

We assume that the periodic tasks are scheduled using a priority-driven algorithm

Background Scheduling of Aperiodic Jobs

- ▶ Aperiodic jobs are scheduled and executed only at times when there are no periodic jobs ready for execution
- ▶ Advantages
 - ▶ Clearly produces feasible schedules
 - ▶ Extremely simple to implement
- ▶ Disadvantages
 - ▶ Not optimal since the execution of aperiodic jobs may be unnecessarily delayed

Example: $T_1 = (3, 1)$, $T_2 = (10, 4)$

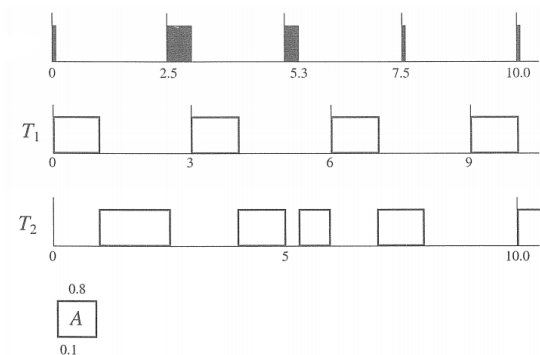


Polled Execution of Aperiodic Jobs

- ▶ We may use a *polling server*
 - ▶ A periodic job (p_s, e_s) scheduled according to the periodic algorithm, generally as the highest priority job
 - ▶ When executed, it examines the aperiodic job queue
 - ▶ If an aperiodic job is in the queue, it is executed for up to e_s time units
 - ▶ If the aperiodic queue is empty, the polling server self-suspends, giving up its execution slot
 - ▶ The server does not wake-up once it has self-suspended, aperiodic jobs which become active during a period are not considered for execution until the next period begins
- ▶ Simple to prove correctness, performance less than ideal – executes aperiodic jobs in particular timeslots

Polled Execution of Aperiodic Jobs

Example: $T_1 = (3, 1)$, $T_2 = (10, 4)$, $poller = (2.5, 0.5)$



Can we do better?

Yes, polling server is a special case of *periodic-server* for aperiodic jobs.

Periodic Servers – Terminology

periodic server = a task that behaves much like a periodic task, but is created for the purpose of executing aperiodic jobs

- ▶ A periodic server, $T_S = (p_S, e_S)$
 - ▶ p_S is a period of the server
 - ▶ e_S is the (maximal) *budget* of the server
- ▶ The budget can be *consumed* and *replenished*; the budget is *exhausted* when it reaches 0
(Periodic servers differ in how they consume and replenish the budget)
- ▶ A periodic server is
 - ▶ *backlogged* whenever the aperiodic job queue is non-empty
 - ▶ *idle* if the queue is empty
 - ▶ *eligible* if it is backlogged and the budget is not exhausted
- ▶ When a periodic server is eligible, it is scheduled as any other periodic task with parameters (p_S, e_S)

Each periodic server is thus specified by

- ▶ *consumption rules*: How the budget is consumed
- ▶ *replenishment rules*: When and how the budget is replenished

Polling server

- ▶ *consumption rules*:
 - ▶ Whenever the server executes, the budget is consumed at the rate one per unit time.
 - ▶ Whenever the server becomes idle, the budget gets immediately exhausted
- ▶ *replenishment rule*: At each time instant $k \cdot p_S$ replenish the budget to e_S

Deferrable server

- ▶ *Consumption rules:*
 - ▶ The budget is consumed at the rate of one per unit time whenever the server executes
 - ▶ Unused budget is retained throughout the period, to be used whenever there are aperiodic jobs to execute (i.e. instead of discarding the budget if no aperiodic job to execute at start of period, keep in the hope a job arrives)
- ▶ *Replenishment rule:*
 - ▶ The budget is set to e_S at multiples of the period
 - ▶ i.e. time instants $k \cdot p_S$ for $k = 0, 1, 2, \dots$

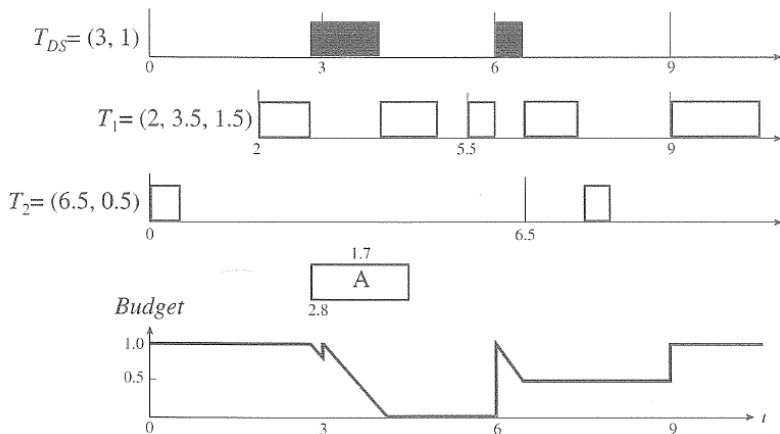
(Note that the server is not able to cumulate the budget over periods)

We consider both

- ▶ Fixed-priority scheduling
- ▶ Dynamic-priority scheduling (EDF)

Deferrable Server – RM

Here the tasks are scheduled using RM.

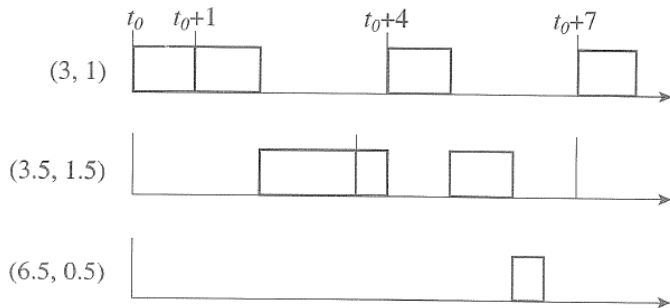


Is it possible to increase the budget of the server to 1.5 ?

Deferrable Server – RM

Consider $T_1 = (3.5, 1.5)$, $T_2 = (6.5, 0.5)$ and $T_{DS} = (3, 1)$

A **critical instant** for $T_1 = (3.5, 1.5)$ looks as follows:



i.e. increasing the budget above 1 may cause T_1 to miss its deadline

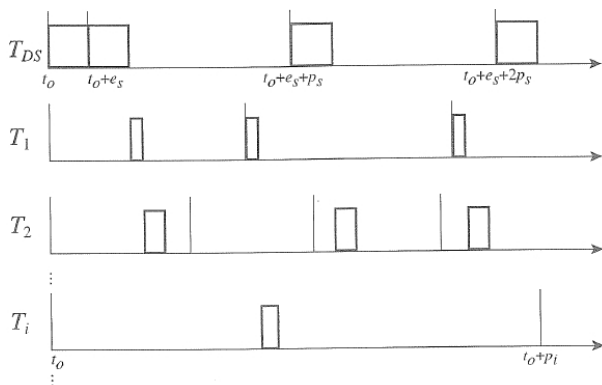
Lemma 19

Assume a fixed-priority scheduling algorithm. Assume that $D_i \leq p_i$ and that the deferrable server (p_S, e_S) has the highest priority among all tasks. Then a critical instant of every periodic task T_i occurs at a time t_0 when all of the following are true:

- ▶ *One of its jobs $J_{i,c}$ is released at t_0*
- ▶ *A job in every higher-priority periodic task is released at t_0*
- ▶ *The budget of the server is e_S at t_0 , one or more aperiodic jobs are released at t_0 , and they keep the server backlogged hereafter*
- ▶ *The next replenishment time of the server is $t_0 + e_S$*

Deferrable Server – Critical Instant

Assume $T_{DS} \supset T_1 \supset T_2 \supset \dots \supset T_n$
(i.e. T_1 has the highest priority and T_n lowest)



Deferrable Server – Time Demand Analysis

Assume that the deferrable server has the highest priority

- ▶ The definition of critical instant is identical to that for the periodic tasks without the deferrable server + the worst-case requirements for the server
- ▶ Thus the expression for the time-demand function becomes

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k + e_S + \left\lceil \frac{t - e_S}{p_S} \right\rceil e_S \quad \text{for } 0 < t \leq p_i$$

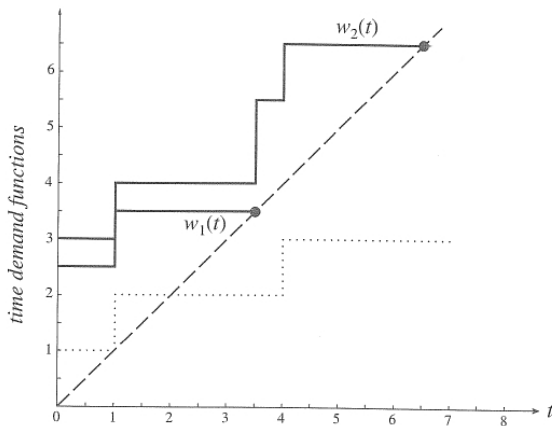
- ▶ To determine whether the task T_i is schedulable, we simply check whether $w_i(t) \leq t$ for some $t \leq D_i$

Note that this is a *sufficient condition*, not necessary.

- ▶ Check whether $w_i(t) \leq t$ for some t equal either
 - ▶ to D_i , or
 - ▶ to $j \cdot p_k$ where $k = 1, 2, \dots, i$ and $j = 1, 2, \dots, \lfloor D_i/p_k \rfloor$, or
 - ▶ to $e_S, e_S + p_S, e_S + 2p_S, \dots, e_S + \lfloor (D_i - e_i)/p_S \rfloor p_S$

Deferrable Server – Time Demand Analysis

$$T_{DS} = (3, 1.0), T_1 = (3.5, 1.5), T_2 = (6.5, 0.5)$$



Deferrable Server – Schedulable Utilization

- ▶ No maximum schedulable utilization is known in general
- ▶ A special case:
 - ▶ A set T of n independent, preemptible periodic tasks whose periods satisfy $p_S < p_1 < \dots < p_n < 2p_S$ and $p_n > p_S + e_S$ and whose relative deadlines are equal to their respective periods, can be scheduled according to RM with a deferrable server provided that

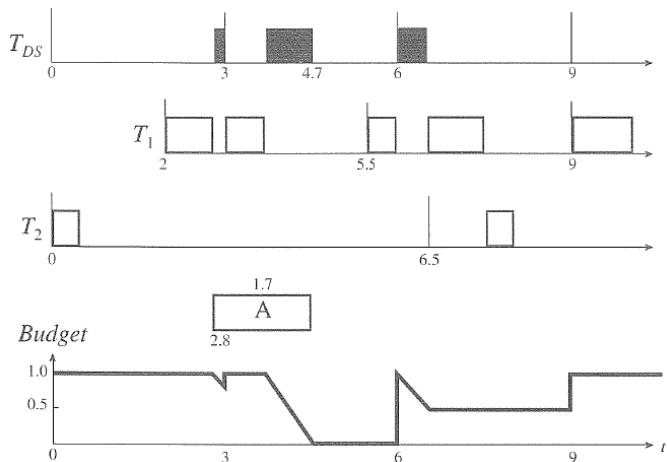
$$U^T \leq U_{RM/DS}(n) := (n-1) \left[\left(\frac{u_S + 2}{u_S + 1} \right)^{\frac{1}{n-1}} - 1 \right]$$

where $u_S = e_S/p_S$

Deferrable Server – EDF

Here the tasks are scheduled using EDF.

$$T_{DS} = (3, 1), T_1 = (2, 3.5, 1.5), T_2 = (6.5, 0.5)$$



Theorem 20

A set of n independent, preemptable, periodic tasks satisfying $p_i \leq D_i$ for all $1 \leq i \leq n$ is schedulable with a deferrable server with period p_S , execution budget e_S and utilization $u_S = e_S/p_S$ according to the EDF algorithm if:

$$\sum_{k=1}^n u_k + u_S \left(1 + \frac{p_S - e_S}{\min_i D_i} \right) \leq 1$$

Sporadic Server – Motivation

- ▶ Problem with polling server: $T_{PS} = (p_S, e_S)$ executes aperiodic tasks at the multiples of p_S
- ▶ Problem with deferrable server: $T_{DS} = (p_S, e_S)$ may delay lower priority jobs longer than periodic task (p_S, e_S)
Therefore special version of time-demand analysis and utilization bounds were needed.
- ▶ **Sporadic server** $T_{SS} = (e_S, p_S)$
 - ▶ may execute jobs “in the middle” of its period
 - ▶ *never* delays periodic tasks for longer time than the periodic task (p_S, e_S)
Thus can be tested for schedulability as an ordinary periodic task.

Originally proposed by Sprunt, Sha, Lehoczky in 1989
original version contains a bug which allows longer delay of lower priority jobs

Part of POSIX standard

also incorrect as observed and (probably) corrected by Stanovich in 2010

Very Simple Sporadic Server

For simplicity, we consider only fixed priority scheduling, i.e. assume $T_1 \supset T_2 \supset \dots \supset T_n$ and consider a sporadic server $T_{SS} = (p_S, e_S)$ with the *highest priority*

Notation:

- ▶ t_r = the *latest* replenishment time
 - ▶ t_f = first instant after t_r at which server begins to execute
 - ▶ n_r = a variable representing the *next* replenishment
-
- ▶ *Consumption rule*: The budget is consumed (at the rate of one per unit time) whenever the current time t satisfies $t \geq t_f$
 - ▶ *Replenishment rules*: At the beginning, $t_r = n_r = 0$
 - ▶ Whenever the current time is equal to n_r , the budget is set to e_S and t_r is set to the current time
 - ▶ At the first instant t_f after t_r at which the server starts executing, n_r is set to $t_f + p_S$

(Note that such server resembles a periodic task with the highest priority whose jobs are released at times t_f and execution times are at most e_S)

Very Simple Sporadic/Background Server

New notation:

- ▶ t_r = the *latest* replenishment time
 - ▶ t_f = first instant after t_r at which server begins to execute and *at least one task of \mathcal{T} is not idle*
 - ▶ n_r = a variable representing the *next* replenishment
-
- ▶ *Consumption rule*: The budget is consumed (at the rate of one per unit time) whenever the current time t satisfies $t \geq t_f$ and *at least one task of \mathcal{T} is not idle*
 - ▶ *Replenishment rules*: At the beginning, $t_r = n_r = 0$
 - ▶ Whenever the current time is equal to n_r , the budget is set to e_S and t_r is set to the current time
 - ▶ *At the beginning of an idle interval of \mathcal{T} , the budget is set to e_S and n_r is set to the end of this interval*
 - ▶ At the first instant t_f after t_r at which the server starts executing and *\mathcal{T} is not idle*, n_r is set to $t_f + p_S$

This combines the very simple sporadic server with background scheduling.

Very Simple Sporadic Server

Correctness (informally):

Assuming that \mathcal{T} never idles, the sporadic server resembles a periodic task with the highest priority whose jobs are released at times t_f and execution times are at most e_S

Whenever \mathcal{T} idles, the sporadic server executes in the background, i.e. does not block any periodic task, hence does not consume the budget

Whenever an idle interval of \mathcal{T} ends, we may treat this situation as a restart of the system with possibly different phases of tasks (so that it is safe to have the budget equal to e_S)

Note that in both versions of the sporadic server, e_S units of execution time are available for aper. jobs every p_S units of time
This means that if the server is always backlogged, then it executes for e_S time units every p_S units of time

Real-Time Scheduling

Priority-Driven Scheduling

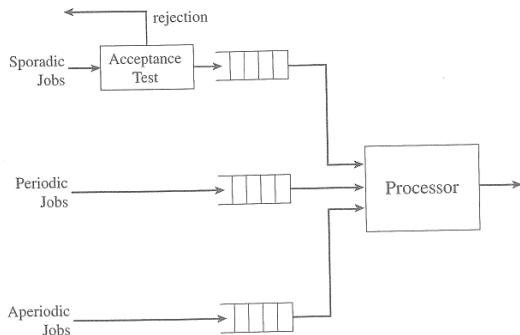
Sporadic Tasks

Current Assumptions

- ▶ Single processor
- ▶ Fixed number, n , of *independent periodic* tasks, T_1, \dots, T_n where $T_i = (\varphi_i, p_i, e_i, D_i)$
 - ▶ Jobs can be preempted at any time and never suspend themselves
 - ▶ No resource contentions
- ▶ Sporadic tasks
 - ▶ Independent of the periodic tasks
 - ▶ Jobs can be preempted at any time
- ▶ Aperiodic tasks
For simplicity scheduled in the background – i.e. we may ignore them
- ▶ Jobs are scheduled using a priority driven algorithm

A sporadic job = a job of a sporadic task

Our situation



- ▶ Based on the execution time and deadline of each newly arrived sporadic job, decide whether to accept or reject the job
- ▶ Accepting the job implies that the job will complete within its deadline, without causing any periodic job or previously accepted sporadic job to miss its deadline
- ▶ Do not accept a sporadic job if cannot guarantee it will meet its deadline

Scheduling Sporadic Jobs – Correctness and Optimality

- ▶ A *correct* schedule is one where all periodic tasks, and all sporadic jobs that have been accepted, meet their deadlines
- ▶ A scheduling algorithm supporting sporadic jobs is a *correct* algorithm if it only produces correct schedules for the system
- ▶ A sporadic job scheduling algorithm is *optimal* if it accepts a new sporadic job, and schedules that job to complete by its deadline, iff the new job can be correctly scheduled to complete in time

Model for Scheduling Sporadic Jobs with EDF

- ▶ Assume that all jobs in the system are scheduled by EDF
- ▶ if more sporadic jobs are released at the same time their acceptance test is done in the EDF order
- ▶ Definitions:
 - ▶ Sporadic jobs are denoted by $S(r, d, e)$ where r is the release time, d the (absolute) deadline, and e is the maximum execution time
 - ▶ The **density** of $S(r, d, e)$ is defined by $e/(d - r)$
 - ▶ The **total density** of a set of sporadic jobs is the sum of densities of these jobs
 - ▶ The sporadic job $S(r, d, e)$ is *active at time t* iff $t \in (r, d]$

Note that each job of a periodic task (φ, p, e, D) can be seen as a sporadic job; to simplify, we **assume that always** $D \leq p$.

This in turn means that there is always at most one job of a given task active at a given time instant.

For every job of this task released at r with abs. deadline d , we obtain the density $e/(d - r) = e/D$

Schedulability of Sporadic Jobs with EDF

Theorem 21

A set of independent preemptable sporadic jobs is schedulable according to EDF if at every time instant t the total density of all jobs active at time t is at most one.

Proof.

By contradiction, suppose that a job misses its deadline at t , no deadlines missed before t

Let t_{-1} be the supremum of time instants before t when either the system idles, or a job with a deadline after t executes

Suppose that jobs J_1, \dots, J_k execute in $[t_{-1}, t]$ and that they are ordered w.r.t. increasing deadline (J_k misses its deadline at t)

Let L be the number of releases and completions in $[t_{-1}, t]$, denote by t_i the i -th time instant when i -th such event occurs (then $t_{-1} = t_1$, we denote by t_{L+1} the time instant t)

Denote by X_i the set of all jobs that are active during the interval $(t_i, t_{i+1}]$ and let Δ_i be their total density

The rest on whiteboard



Sporadic Jobs with EDF – Example

Note that the above theorem includes both the periodic as well as sporadic jobs

This test is sufficient but not necessary

Example 22

Three sporadic jobs: $S_1(0, 2, 1)$, $S_2(0.5, 2.5, 1)$, $S_3(1, 3, 1)$

Total density at time 1.5 is 1.5

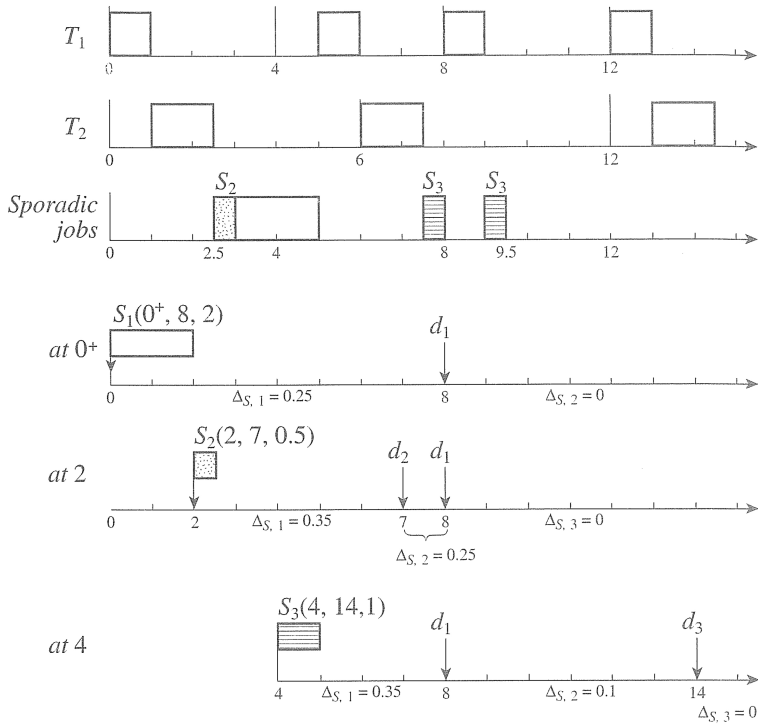
Yet, the jobs are schedulable by EDF

Admission Control for Sporadic Jobs with EDF

Let Δ be the total density of *periodic tasks*.

Assume that a new sporadic job $S(t, d, e)$ is released at time t .

- ▶ At time t there are n active sporadic jobs in the system
- ▶ The EDF scheduler maintains a list of the jobs, in non-decreasing order of their deadlines
 - ▶ The deadlines partition the time from t to ∞ into $n + 1$ discrete intervals I_1, I_2, \dots, I_{n+1}
 - ▶ I_1 begins at t and ends at the earliest sporadic job deadline
 - ▶ For each $1 \leq k \leq n$, each I_{k+1} begins when the interval I_k ends, and ends at the next deadline in the list (or ∞ for I_{n+1})
 - ▶ The scheduler maintains the total density $\Delta_{S,k}$ of sporadic jobs active in each interval I_k
- ▶ Let I_ℓ be the interval containing the deadline d of the new sporadic job $S(t, d, e)$
 - ▶ The scheduler accepts the job if $e/(d - t) + \Delta_{S,k} \leq 1 - \Delta$ for all $k = 1, 2, \dots, \ell$
 - ▶ i.e. accept if the new sporadic job can be added, without increasing density of any intervals past 1



This acceptance test is not optimal: a sporadic job may be rejected even though it could be scheduled.

- ▶ The test is based on the density and hence is sufficient but not necessary.
- ▶ It is possible to derive a – much more complex – expression for schedulability which takes into account slack time, and is optimal. Unclear if the complexity is worthwhile.

Sporadic Jobs with EDF

- ▶ One way to schedule sporadic jobs in a **fixed-priority** system is to use a sporadic server to execute them
- ▶ Because the server (p_S, e_S) has e_S units of processor time every p_S units of time, the scheduler can compute the least amount of time available to every sporadic job in the system
 - ▶ Assume that sporadic jobs are ordered among themselves according to EDF
 - ▶ When first sporadic job $S_1(t, d_{S,1}, e_{S,1})$ arrives, there is at least

$$\lfloor (d_{S,1} - t) / p_S \rfloor e_S$$

units of processor time available to the server before the deadline of the job

- ▶ Therefore it accepts S_1 if the slack of the job

$$\sigma_{S,1}(t) = \lfloor (d_{S,1} - t) / p_S \rfloor e_S - e_{S,1} \geq 0$$

Sporadic Jobs with EDF

- ▶ To decide if a new job $S_i(t, d_{S,i}, e_{S,i})$ is acceptable when there are n sporadic jobs in the system, the scheduler first computes the slack $\sigma_{S,i}(t)$ of S_i :

$$\sigma_{S,i}(t) = \lfloor (d_{S,i} - t) / p_S \rfloor e_S - e_{S,i} - \sum_{d_{S,k} < d_{S,i}} (e_{S,k} - \xi_{S,k})$$

where $\xi_{S,k}$ is the execution time of the completed part of the existing job S_k

Note that the sum is taken over sporadic jobs with earlier deadline as S_i since sporadic jobs are ordered according to EDF

- ▶ The job cannot be accepted if $\sigma_{S,i}(t) < 0$
- ▶ If $\sigma_{S,i}(t) \geq 0$, the scheduler checks if any existing sporadic job S_k with deadline equal to, or after $d_{S,i}$ may be adversely affected by the acceptance of S_i , i.e. check if $\sigma_{S,k}(t) \geq e_{S,i}$

Real-Time Scheduling

Resource Access Control

[Some parts of this lecture are based on a real-time systems course
of Colin Perkins

<http://csperkins.org/teaching/rtes/index.html>]

Current Assumptions

- ▶ Single processor
- ▶ Individual jobs
(that possibly belong to periodic/aperiodic/sporadic tasks)
 - ▶ Jobs can be preempted at any time and never suspend themselves
- ▶ Jobs are scheduled using a priority-driven algorithm
i.e., jobs are assigned priorities, scheduler executes jobs according to these priorities
- ▶ n resources R_1, \dots, R_n of distinct types
 - ▶ used in non-preemptable and mutually exclusive manner;
serially reusable

Motivation & Notation

Resources may represent:

- ▶ Hardware devices such as sensors and actuators
- ▶ Disk or memory capacity, buffer space
- ▶ Software resources: locks, queues, mutexes etc.

Assume a lock-based concurrency control mechanism

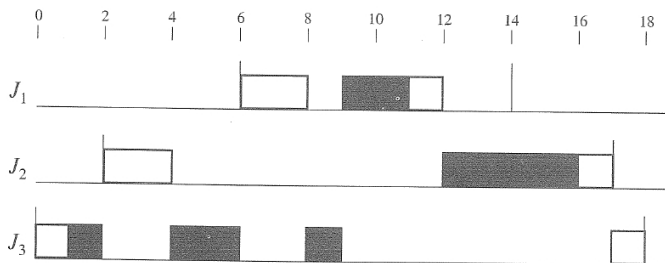
- ▶ A job wanting to use a resource R_k executes $L(R_k)$ to lock the resource R_k
- ▶ When the job is finished with the resource R_k , unlocks this resource by executing $U(R_k)$
- ▶ If lock request fails, the requesting job is **blocked** and has to wait, when the requested resource becomes available, it is unblocked

In particular, a job holding a lock cannot be preempted by a higher priority job needing that lock

The segment of a job that begins at a lock and ends at a matching unlock is a *critical section* (CS)

- ▶ CS must be properly nested if a job needs multiple resources

Example



J_1, J_2, J_3 scheduled according to EDF.

- ▶ At 0, J_3 is ready and executes
- ▶ At 1, J_3 executes $L(R)$ and is granted R
- ▶ J_2 is released at 2, preempts J_3 and begins to execute
- ▶ At 4, J_2 executes $L(R)$, becomes blocked, J_3 executes
- ▶ At 6, J_1 becomes ready, preempts J_3 and begins to execute
- ▶ At 8, J_1 executes $L(R)$, becomes blocked, and J_3 executes
- ▶ At 9, J_3 executes $U(R)$ and both J_1 and J_2 are unblocked. J_1 has higher priority than J_2 and executes
- ▶ At 11, J_1 executes $U(R)$ and continues executing

Priority Inversion

Definition 23

Priority inversion occurs when

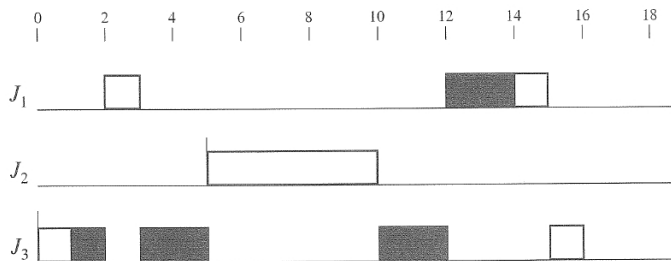
- ▶ a **high** priority job
- ▶ is blocked by a **low** priority job
- ▶ which is subsequently preempted by a **medium** priority job

Then effectively the **medium** priority job executes with higher priority than the **high** priority job even though they do not contend for resources

There may be arbitrarily many medium priority jobs that preempt the low priority job \Rightarrow uncontrolled priority inversion

Priority Inversion – Example

Uncontrolled priority inversion:



High priority job (J_1) can be blocked by low priority job (J_3) for unknown amount of time depending on middle priority jobs (J_2)

Definition 24 (suitable for resource access control)

A deadlock occurs when there is a set of jobs \mathcal{D} such that each job of \mathcal{D} is waiting for a resource previously allocated by another job of \mathcal{D} .

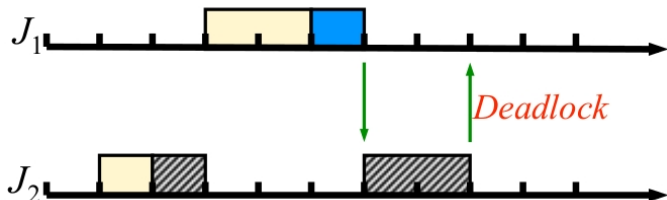
Deadlocks can be

- ▶ *detected*: regularly check for deadlock, e.g. search for cycles in a resource allocation graph regularly
- ▶ *avoided*: postpone unsafe requests for resources even though they are available (banker's algorithm, priority-ceiling protocol)
- ▶ *prevented*: many methods invalidating sufficient conditions for deadlock (e.g., impose locking order on resources)

See your operating systems course for more information

Deadlock – Example

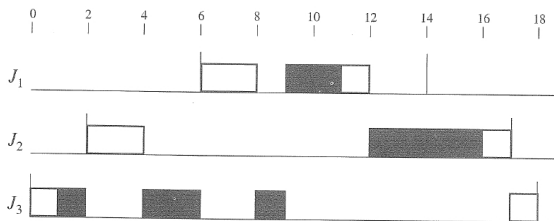
Deadlock can result from piecemeal acquisition of resources: classic example of two jobs J_1 and J_2 both needing both resources R and R'



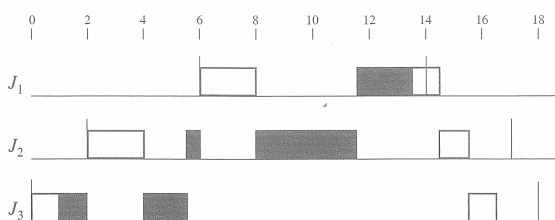
- ▶ J_2 locks R' and J_1 locks R
- ▶ J_1 tries to get R' and is blocked
- ▶ J_2 tries to get R and is blocked

Timing Anomalies due to Resources

Previous example, the critical section of J_3 has length 4



... the critical section of J_3 shortened to 2.5



... but response of J_1 becomes longer!

Controlling Timing Anomalies

Contention for resources causes timing anomalies, priority inversion and deadlock

Several protocols exist to control the anomalies

- ▶ Non-preemptive CS
- ▶ Priority inheritance protocol
- ▶ Priority ceiling protocol
- ▶

Terminology:

- ▶ A job J_h is *blocked* by a job J_k when
 - ▶ the priority of J_k is lower than the priority of J_h and
 - ▶ J_k holds a resource R and
 - ▶ J_h executes $L(R)$.

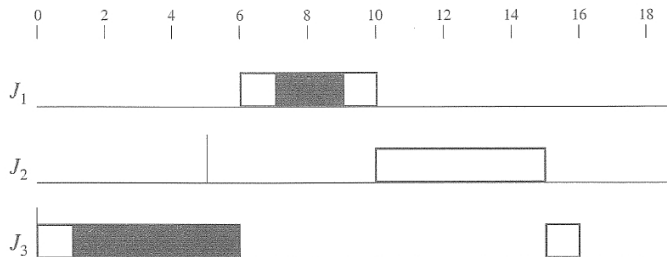
In such situation we sometimes say that J_h is blocked by the corresponding critical section of J_k .

Non-preemptive Critical Sections

The **protocol**: when a job locks a resource, it is scheduled with priority higher than all other jobs (i.e., is non-preemptive)

Example 25

Jobs J_1, J_2, J_3 with release times 2, 5, 0, resp., and with execution times 4, 5, 7, resp.



Non-preemptive Critical Sections – Features

- ▶ no deadlock as no job holding a resource is ever preempted
- ▶ no priority inversion:
 - ▶ A job J_h can be blocked (by a lower priority job) *only at release time*.
(Indeed, if J_h is not blocked at the release time r_h , it means that no lower priority job holds any resource at r_h . However, no lower priority job can be executed before completion of J_h , and thus no lower priority job may block J_h .)
 - ▶ If J_h is blocked at release time, then once the blocking critical section completes, no lower priority job can block J_h .
 - ▶ It follows that any job can be blocked only once, at release time, blocking time is bounded by duration of one critical section of a lower priority job.

Advantage: very simple; easy to implement both in fixed and dynamic priority; no prior knowledge of resource demands of jobs needed

Disadvantage: every job can be blocked by every lower-priority job with a critical section, even if there is no resource conflict

Priority-Inheritance Protocol

Idea: adjust the scheduling priorities of jobs during resource access, to reduce the duration of timing anomalies

(As opposed to non-preemptive CS protocol, this time the priority is not always increased to maximum)

Notation:

- ▶ *assigned priority* = priority assigned to a job according to a standard scheduling algorithm
- ▶ At any time t , each ready job J_k is scheduled and executes at its *current priority* $\pi_k(t)$ which may differ from its assigned priority and may vary with time
 - ▶ The current priority $\pi_k(t)$ of a job J_k may be raised to the higher priority $\pi_h(t)$ of another job J_h
 - ▶ In such a situation, the lower-priority job J_k is said to *inherit* the priority of the higher-priority job J_h , and J_k executes at its inherited priority $\pi_h(t)$

Priority-Inheritance Protocol

▶ Scheduling rules:

- ▶ Jobs are scheduled in a preemptable priority-driven manner *according to their current priorities*
- ▶ At release time, the current priority of a job is equal to its assigned priority
- ▶ The current priority remains equal to the assigned priority, except when the priority-inheritance rule is invoked

▶ Priority-inheritance rule:

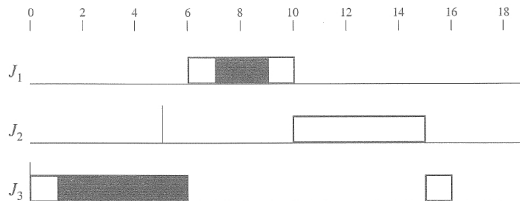
- ▶ When a job J_h becomes blocked on a resource R , the job J_k which blocks J_h inherits the current priority $\pi_h(t)$ of J_h ;
- ▶ J_k executes at its inherited priority until it releases R ;
at that time, the priority of J_k is *set to the highest priority of all jobs still blocked by J_k after releasing R* .
(the resulting priority may still be an inherited priority)

- ▶ **Resource allocation:** When a job J requests a resource R at t :
 - ▶ If R is free, R is allocated to J until J releases it
 - ▶ If R is not free, the request is denied and J is blocked

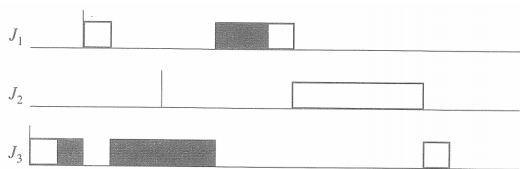
(Note that J is only denied R if the resource is held by another job.)

Priority-Inheritance Simple Example

non-preemptive CS:

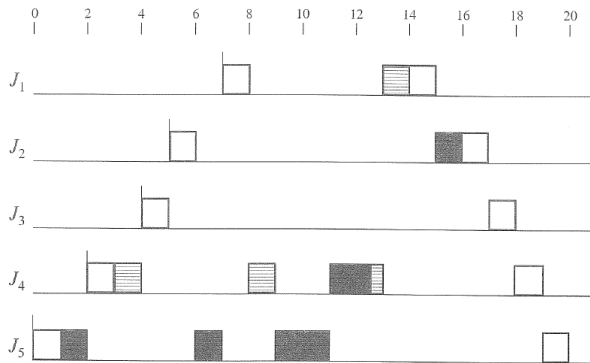


priority-inheritance:



- ▶ At 3, J_1 is blocked by J_3 , J_3 inherits priority of J_1
- ▶ At 5, J_2 is released but cannot preempt J_3 since the inherited priority of J_3 is higher than the (assigned) priority of J_2

Priority-Inheritance Example



- ▶ At 0, J_5 starts executing at priority 5, at 1 it executes L (Black)
- ▶ At 2, J_4 preempts J_5 and executes
- ▶ At 3, J_4 executes L (Shaded), J_4 continues to execute
- ▶ At 4, J_3 preempts J_4 ; at 5, J_2 preempts J_3
- ▶ At 6, J_2 executes L (Black) and is blocked by J_5 . Thus J_5 inherits the priority 2 of J_2 and executes
- ▶ At 8, J_1 executes L (Shaded) and is blocked by J_4 . Thus J_1 inherits the

Properties of Priority-Inheritance Protocol

- ▶ Simple to implement, does not require prior knowledge of resource requirements
- ▶ Jobs exhibit two types of "blocking"
 - ▶ **(Direct) blocking** due to resource locks
i.e., a job J_ℓ locks a resource R , J_h executes $L(R)$ is directly blocked by J_ℓ on R
 - ▶ **Priority-inheritance "blocking"**
i.e., a job J_h is preempted by a lower-priority job that inherited a higher priority
- ▶ Jobs may exhibit **transitive blocking**
In the previous example, at 9, J_5 blocks J_4 and J_4 blocks J_1 , hence J_5 inherits the priority of J_1
- ▶ Deadlock is *not* prevented
In the previous example, let J_5 request *shaded* at 6.5, then J_4 and J_5 become deadlocked
- ▶ Can reduce blocking time (see next slide) compared to non-preemptable CS but does not guarantee to minimize blocking

Priority-Inheritance – Blocking Time (Optional)

$z_{\ell,k}$ = the k -th critical section of J_ℓ

A job J_h is blocked by $z_{\ell,k}$ if J_h has higher assigned priority than J_ℓ but has to wait for J_ℓ to exit $z_{\ell,k}$ in order to continue

$\beta_{h,\ell}^*$ = the set of all maximal critical sections $z_{\ell,k}$ that *may* block J_h , i.e., which correspond to resources that are (potentially) used by jobs with priorities equal or higher than J_h .

(recall that CS are properly nested, maximal CS which may block J_h is the one which is not contained within any other CS which may block J_h)

Theorem 26

Let J_h be a job and let J_{h+1}, \dots, J_{h+m} be jobs with lower priority than J_h . Then J_h can be blocked for at most the duration of one critical section in each of $\beta_{h,\ell}^$ where $\ell \in \{h+1, \dots, h+m\}$.*

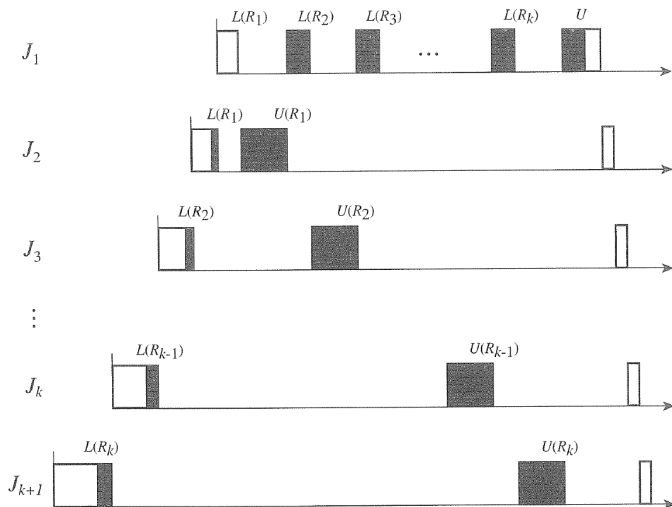
The theorem is a direct consequence of the next lemma.

Lemma 27

J_h can be blocked by J_ℓ only if J_ℓ is executing within a critical section $z_{\ell,k}$ of $\beta_{h,\ell}^*$ when J_h is released

- ▶ Assume that J_h is released at t and J_ℓ is in no CS of $\beta_{h,\ell}^*$ at t . We show that J_ℓ never executes between t and completion of J_h :
 - ▶ If J_ℓ is not in any CS at t , then its current priority at t is equal to its assigned priority and cannot increase. Thus, J_ℓ has to wait for completion of J_h as the current priority of J_h is always higher than the assigned priority of J_ℓ .
 - ▶ If J_ℓ is still in a CS at t , then this CS does not belong to $\beta_{h,\ell}^*$ and thus cannot block J_h before completion and cannot execute before completion of J_h .
- ▶ Assume that J_ℓ leaves $z_{\ell,k} \in \beta_{h,\ell}^*$ at time t . We show that J_ℓ never executes between t and completion of J_h :
 - ▶ If J_ℓ is not in any CS at t , then, as above, J_ℓ never executes before completion of J_h and cannot block J_h .
 - ▶ If J_ℓ is still in a CS at t , then this CS does not belong to $\beta_{h,\ell}^*$ because otherwise $z_{\ell,k}$ would not be maximal. Thus J_ℓ cannot block J_h , and thus J_ℓ is never executed before completion of J_h .

Priority-Inheritance – The Worst Case



J_1 is blocked for the total duration of all critical sections in all lower priority jobs.

Priority-Ceiling Protocol

The goal: to further reduce blocking times due to resource contention and to prevent deadlock

- ▶ in its basic form priority-ceiling protocol works under the assumption that the priorities of jobs and resources required by all jobs are known a priori
- can be extended to dynamic priority (job-level fixed priority), see later

Notation:

- ▶ The *priority ceiling* of any resource R_k is the highest priority of all the jobs that require R_k and is denoted by $\Pi(R_k)$
- ▶ At any time t , the current priority ceiling $\Pi(t)$ of the system is equal to the highest priority ceiling of the resources that are in use at the time
- ▶ If all resources are free, $\Pi(t)$ is equal to Ω , a newly introduced priority level that is lower than the lowest priority level of all jobs

Priority-Ceiling Protocol

The scheduling and priority-inheritance rules are the same as for priority-inheritance protocol

- ▶ **Scheduling rules:**

- ▶ Jobs are scheduled in a preemptable priority-driven manner *according to their current priorities*
- ▶ At release time, the current priority of a job is equal to its assigned priority
- ▶ The current priority remains equal to the assigned priority, except when the priority-inheritance rule is invoked

- ▶ **Priority-inheritance rule:**

- ▶ When job J_h becomes blocked on a resource R , the job J_k which blocks J_h inherits the current priority $\pi_h(t)$ of J_h ;
- ▶ J_k executes at its inherited priority until it releases R ;
at that time, the priority of J_k is *set to the highest priority of all jobs still blocked by J_k after releasing R .*
(which may still be an inherited priority)

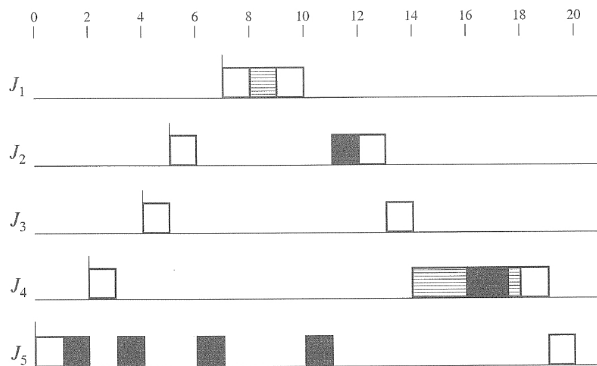
Priority-Ceiling Protocol

Resource allocation rules:

- ▶ When a job J requests a resource R held by another job, the request fails and the requesting job blocks
- ▶ When a job J requests a resource R at time t , and that resource is free:
 - ▶ If J 's priority $\pi(t)$ is *strictly higher* than current priority ceiling $\Pi(t)$, R is allocated to J
 - ▶ If J 's priority $\pi(t)$ is not higher than $\Pi(t)$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\Pi(t)$, otherwise J is blocked
(Note that only one job may hold the resources whose priority ceiling is equal to $\Pi(t)$)

Note that unlike priority-inheritance protocol, the priority-ceiling protocol can deny access to an available resource.

Priority-Ceiling Protocol



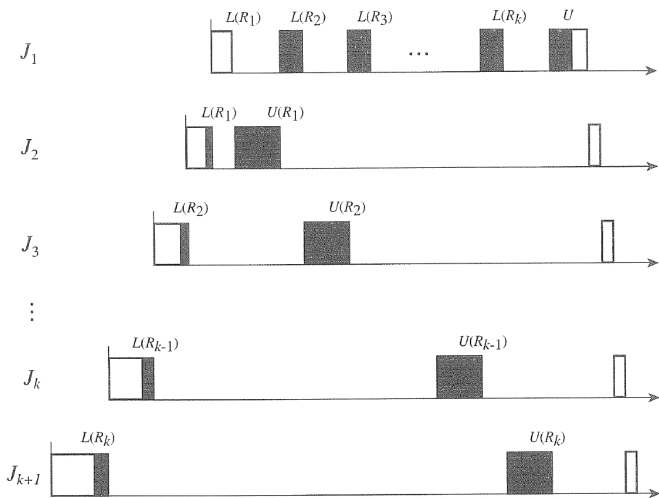
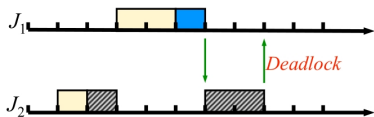
- ▶ At 1, $\Pi(t) = \Omega$, J_5 executes $L(\text{Black})$, continues executing
- ▶ At 3, $\Pi(t) = 2$, J_4 executes $L(\text{Shaded})$; because the ceiling of the system $\Pi(t)$ is higher than the current priority of J_4 , job J_4 is blocked, J_5 inherits J_4 's priority and executes at priority 4
- ▶ At 4, J_3 preempts J_5 ; at 5, J_2 preempts J_3 . At 6, J_2 requests Black and is directly blocked by J_5 . Consequently, J_5 inherits priority 2 and executes until preempted by J_1

Theorem 28

Assume a system of preemptable jobs with fixed assigned priorities. Then

- ▶ *deadlock may never occur,*
- ▶ *a job can be blocked for at most the duration of one critical section.*

These situations cannot occur with priority ceiling protocol:



Differences between the priority-inheritance and priority-ceiling

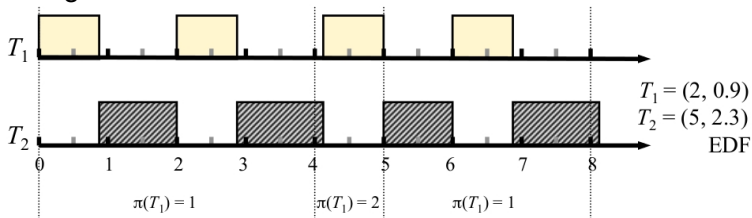
- ▶ Priority-inheritance is greedy, while priority ceiling is not
The priority-ceiling protocol may withhold access to a free resource, i.e., a job can be prevented from execution by a lower-priority job which does not hold the requested resource – *avoidance "blocking"*
- ▶ The priority ceiling protocol forces a fixed order onto resource accesses thus eliminating deadlock

Resources in Dynamic Priority Systems

The priority ceiling protocol assumes fixed and known priorities

In a dynamic priority system, the priorities of the periodic tasks change over time, while the set of resources is required by each task remains constant

- ▶ As a consequence, the priority ceiling of each resource changes over time



What happens if T_1 uses resource X , but T_2 does not?

- ▶ Priority ceiling of X is 1 for $0 \leq t \leq 4$, becomes 2 for $4 \leq t \leq 5$, etc. even though the set of resources is required by the tasks remains unchanged

Resources in Dynamic Priority Systems

- ▶ If a system is job-level fixed priority, but task-level dynamic priority, a priority ceiling protocol can still be applied
 - ▶ Each job in a task has a fixed priority once it is scheduled, but may be scheduled at different priority to other jobs in the task (e.g. EDF)
 - ▶ Update the priority ceilings of all resources each time a new job is introduced; use until updated on next job release
- ▶ Has been proven to prevent deadlocks and no job is ever blocked for longer than the length of one critical section
 - ▶ But: very inefficient, since priority ceilings updated frequently
 - ▶ May be better to use priority inheritance, accept longer blocking

Schedulability Tests with Resources

How to adjust schedulability tests?

Add the blocking times to execution times of jobs; then run the test as normal

The blocking time b_i of a job J_i can be determined for all three protocols:

- ▶ non-preemptable CS $\Rightarrow b_i$ is bounded by the maximum length of a critical section in lower priority jobs
- ▶ priority-inheritance $\Rightarrow b_i$ is bounded by the total length of the m longest critical sections where m is the number of jobs that may block J_i
(For a more precise formulation see Theorem 2.)
- ▶ priority-ceiling $\Rightarrow b_i$ is bounded by the maximum length of a critical section

Mars Pathfinder vs Priority Inversion

- ▶ Mars Pathfinder = a US spacecraft that landed on Mars in July 4th, 1997.
 - ▶ Consisted of a lander and a lightweight wheeled robotic Mars rover called Sojourner
- 
- ▶ What Happened:
 - ▶ Few days in to the mission, not long after Pathfinder started gathering meteorological data, it began experiencing total system resets, each resulting in losses of data.
 - ▶ Apparently a software problem caused these resets.
 - ▶ The system:
 - ▶ Pathfinder used the well-known real-time embedded systems kernel VxWorks by Wind River.
 - ▶ VxWorks uses preemptive priority-based scheduling, in this case a deadline monotonic algorithm.
 - ▶ Pathfinder contained an "information bus" (a shared memory) used for communication, synchronized by locks.

Mars Pathfinder – The Problem

- ▶ Problematic tasks:
 - ▶ A **bus management** task ran frequently with high priority to move data in/out of the bus. If the bus has been locked, then this thread itself had to wait.
 - ▶ A **meteorological data gathering** task ran as an infrequent, low priority thread, and used the bus to publish its data.
 - ▶ The bus was also used by a **communication** task that ran with medium priority.
- ▶ Occasionally the **communication** task (medium priority) was invoked at the precise time when the **bus management** task (high priority) was blocked by the **meteorological data gathering** task (low priority) – priority inversion!
- ▶ The **bus management** task was blocked for considerable amount of time by the **communication** task, which caused a watchdog timer to go off, notice that the bus management task has not been executed for some time, which typically means that something had gone drastically wrong, and initiate a total system reset.

Mars Pathfinder – Solution

- ▶ JPL (Jet Propulsion Laboratory) engineers spent hours and hours running the system on a spacecraft replica.
- ▶ Early in the morning, after all but one engineer had gone home, the engineer finally reproduced a system reset on the replica.

Solution: Turn the priority inheritance on!

This was done online using a C language interpreter which allowed to execute C functions on-the-fly.

A short code changed a mutex initialization parameter from FALSE to TRUE.

Real-Time Scheduling

Multiprocessor Real-Time Systems

Multiprocessor Real-time Systems

- ▶ Many embedded systems are composed of many processors (control systems in cars, aircraft, industrial systems etc.)
- ▶ Today most processors in computers have multiple cores
The main reason is that increasing frequency of a single processor is no more feasible (mostly due to power consumption problems, growing leakage currents, memory problems etc.)

Applications must be developed specifically for multiprocessor systems.

Multiprocessor Frustration

In case of real-time systems, multiple processors bring serious difficulties concerning correctness, predictability and efficiency.

The “root of all evil” in global scheduling: (Liu, 1969)

Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.

The Model

- ▶ A *job* is a unit of work that is scheduled and executed by a system
(Characterised by the release time r_i , execution time e_i and deadline d_i)
- ▶ A *task* is a set of related jobs which jointly provide some system function
- ▶ Jobs execute on *processors*

In this lecture we consider *m processors*

- ▶ Jobs may use some (shared) passive *resources*

Schedule

Schedule assigns, in every time instant, processors and resources to jobs.

A schedule is *feasible* if *all jobs with hard real-time constraints* complete before their deadlines.

A set of jobs is *schedulable* if there is a feasible schedule for the set.

A scheduling algorithm is *optimal* if it always produces a feasible schedule whenever such a schedule exists.
(and if a cost function is given, minimizes the cost)

We also consider *online* scheduling algorithms that do not use any knowledge about jobs that will be released in the future but are given a complete information about jobs that have been released.
(e.g. EDF is online)

Multiprocessor Taxonomy

- ▶ **Identical processors:** All processors identical, have the same computing power
- ▶ **Uniform processors:** Each processor is characterized by its own computing capacity κ , completes κt units of execution after t time units
- ▶ **Unrelated processors:** There is an execution rate ρ_{ij} associated with each job-processor pair (J_i, P_j) so that J_i completes $\rho_{ij}t$ units of execution by executing on P_j for t time units

In addition, cost of communication can be included etc.

Assumptions – Priority Driven Scheduling

Throughout this lecture we assume:

- ▶ Unless otherwise stated, consider *m identical* processors
- ▶ Jobs can be preempted at any time and never suspend themselves
- ▶ Context switch overhead is negligibly small
i.e. assumed to be zero
- ▶ There is an unlimited number of priority levels

- ▶ For simplicity, we assume *independent* jobs that do not contend for resources

Unless otherwise stated, we assume that scheduling decisions take place only when a job is released, or completed.

Multiprocessor Scheduling Taxonomy

Multiprocessor scheduling attempts to solve two problems:

- ▶ the *allocation problem*, i.e., on which processor a given job executes
- ▶ the *priority problem*, i.e., when and in what order the jobs execute

What results from single processor scheduling remain valid in multiprocessor setting?

- ▶ Are there simple optimal scheduling algorithms?
- ▶ Are there optimal *online* scheduling algorithms (i.e. those that do not know what jobs come in future)
- ▶ Are there efficient tests for schedulability?

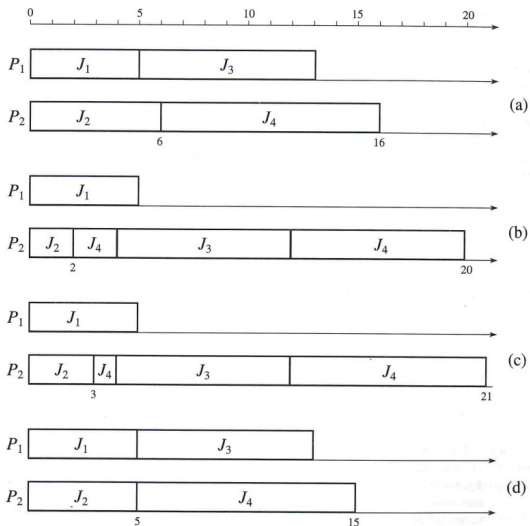
In this lecture we consider:

- ▶ Individual jobs
- ▶ Periodic tasks

Start with n individual jobs $\{J_1, \dots, J_n\}$

Individual Jobs – Timing Anomalies

Priority order: $J_1 \sqsupset \dots \sqsupset J_4$



Individual Jobs – EDF

EDF on m identical processors: At any time instant, jobs with the earliest absolute deadlines are executed on available processors.
(Recall: no job can be executed on more than one processor at a given time!)

Is this optimal? NO!

Example:

J_1, J_2, J_3 where

- ▶ $r_i = 0$ for $i \in \{1, 2, 3\}$
- ▶ $e_1 = e_2 = 1$ and $e_3 = 5$
- ▶ $d_1 = 1, d_2 = 2, d_3 = 5$

Individual Jobs – Online Scheduling

Theorem 29

No optimal **on-line** scheduler can exist for a set of jobs with two or more distinct deadlines on any $m > 1$ processor system.

Proof.

Assume $m = 2$ and consider three jobs J_1, J_2, J_3 are released at time 0 with the following parameters:

- ▶ $e_1 = e_2 = 2$ and $e_3 = 4$
- ▶ $d_1 = d_2 = 4$ and $d_3 = 8$

Depending on scheduling in $[0, 2]$, new tasks T_4, T_5 are released at 2:

- ▶ If J_3 is executed in $[0, 2]$, then at 2 release J_4, J_5 with $d_4 = d_5 = 4$ and $e_4 = e_5 = 2$.
- ▶ If J_3 is not executed in $[0, 2]$, then at 4 release J_4, J_5 with $d_4 = d_5 = 8$ and $e_4 = e_5 = 4$.

In either case the schedule produced is not feasible. However, if the scheduler is given either of the sets $\{J_1, \dots, J_5\}$ at the beginning, then there is a feasible schedule. □

Individual Jobs – Speedup Helps(?)

Theorem 30

If a set of jobs is feasible on m identical processors, then the same set of jobs will be scheduled to meet all deadlines by EDF on identical processors in which the individual processors are $(2 - \frac{1}{m})$ times as fast as in the original system.

The result is tight for EDF (assuming dynamic job priority):

Theorem 31

There are sets of jobs that can be feasibly scheduled on m identical processors but EDF cannot schedule them on m processors that are only $(2 - \frac{1}{m} - \varepsilon)$ faster for every $\varepsilon > 0$.

... there are also general lower bounds for online algorithms:

Theorem 32

*There are sets of jobs that can be feasibly scheduled on m (here m is even) identical processors but **no online** algorithm can schedule them on m processors that are only $(1 + \varepsilon)$ faster for every $\varepsilon < \frac{1}{5}$.*

Reactive Systems

Consider fixed number, n , of *independent periodic* tasks

$$\mathcal{T} = \{T_1, \dots, T_n\}$$

i.e. there is no dependency relation among jobs

- ▶ Unless otherwise stated, assume no phase and deadlines equal to periods
- ▶ Ignore aperiodic tasks
- ▶ No sporadic tasks unless otherwise stated

Utilization u_i of a periodic task T_i with period p_i and execution time e_i is defined by $u_i := e_i/p_i$

u_i is the fraction of time a periodic task with period p_i and execution time e_i keeps a processor busy

Total utilization $U^{\mathcal{T}}$ of a set of tasks $\mathcal{T} = \{T_1, \dots, T_n\}$ is defined as the sum of utilizations of all tasks of \mathcal{T} , i.e. by $U^{\mathcal{T}} := \sum_{i=1}^n u_i$

Given a scheduling algorithm ALG , the *schedulable utilization U_{ALG}* of ALG is the maximum number U such that for all \mathcal{T} : $U_{\mathcal{T}} \leq U$ implies \mathcal{T} is schedulable by ALG .

Multiprocessor Scheduling Taxonomy

Allocation (migration type)

- ▶ **No migration**: each **task** is allocated to a processor
- ▶ (Task-level migration: **jobs** of a task may execute on different processors; however, each job is assigned to a single processor)
- ▶ **Job-level migration**: A single job can migrate and execute on different processors
(however, parallel execution of one job is not permitted and migration takes place only when the job is rescheduled)

Priority type

- ▶ **Fixed task-level priority** (e.g. EDF)
- ▶ **Fixed job-level priority** (e.g. RM)
- ▶ (Dynamic job-level priority)

Partitioned scheduling = No migration

Global scheduling = job-level migration

Fundamental Limit – Fixed Job-Level Priority

Consider m processors and $m + 1$ tasks $\mathcal{T} = \{T_1, \dots, T_{m+1}\}$, each $T_i = (L, 2L - 1)$.

Then

$$U_{\mathcal{T}} = \sum_{i=1}^{m+1} L/(2L - 1) = (m + 1)(L/(2L - 1)) = (m + 1)/2 \cdot L/(L - 1)$$

For very large L , this number is close to $(m + 1)/2$.

The set \mathcal{T} is not schedulable using any *fixed job-level* priority algorithm.

In other words, the schedulable utilization of fixed job-level priority algorithms is at most $(m + 1)/2$, i.e., half of the processors capacity.

There are variants of EDF achieving this bound (see later slides).

Partitioned vs Global Scheduling

Most algorithms up to the end of 1990s based on *partitioned scheduling*

- ▶ no migration

From the end of 1990s, many results concerning *global scheduling*

- ▶ job-level migration

The task-level migration has not been much studied, so it is not covered in this lecture.

We consider fixed job-level priority (e.g. EDF) and fixed task-level priority (e.g. RM).

As before, we ignore dynamic job-level priority.

Partitioned Scheduling & Fixed Job-Level Priority

The algorithm proceeds in two phases:

1. Allocate tasks to processors, i.e., partition the set of tasks into m possibly empty *modules* M_1, \dots, M_m
2. Schedule tasks of each M_i on the processor i according to a given single processor algorithm

The quality of task assignment is determined by the number of assigned processors

- ▶ Use EDF to schedule modules
- ▶ Suffices to test whether the total utilization of each module is ≤ 1
(or, possibly, $\leq \hat{U}$ where $\hat{U} < 1$ in order to accommodate aperiodic jobs ...)

Finding an optimal schedule is equivalent to a simple *uniform-size bin-packing problem* (and hence is NP-complete)

Similarly, we may use RM for fixed task-level priorities (total utilization in modules $\leq \log 2$, etc.)

Global Scheduling

- ▶ All ready jobs are kept in a global queue
- ▶ When selected for execution, a job can be assigned to any processor
- ▶ When preempted, a job goes to the global queue (i.e., forgets on which processor it executed)

Global Scheduling – Fixed Job-Level Priority

Dhall's effect:

- ▶ Consider $m > 1$ processors
- ▶ Let $\varepsilon > 0$
- ▶ Consider a set of tasks $\mathcal{T} = \{T_1, \dots, T_m, T_{m+1}\}$ such that
 - ▶ $T_i = (2\varepsilon, 1)$ for $1 \leq i \leq m$
 - ▶ $T_{m+1} = (1, 1 + \varepsilon)$
- ▶ \mathcal{T} is schedulable
- ▶ Standard EDF and RM schedules are not feasible (whiteb.)

However,

$$U_{\mathcal{T}} = m \frac{2\varepsilon}{1} + \frac{1}{1 + \varepsilon}$$

which means that for small ε the utilization $U_{\mathcal{T}}$ is close to 1 (i.e., $U_{\mathcal{T}}/m$ is very small for $m \gg 0$ processors)

How to avoid Dhall's effect?

- ▶ Note that RM and EDF only account for task periods and ignore the execution time!
- ▶ (Partial) Solution: Dhall's effect can be avoided by giving high priority to tasks with high utilization

Then in the previous example, T_{m+1} is executed whenever it comes and the other tasks are assigned to the remaining processors – produces a feasible schedule

Global Scheduling – Fixed Job-Level Priority

Apparently there is a problem with long jobs due to Dhall's effect.

There is an improved version of EDF called EDF-US(1/2) which

- ▶ assigns the highest priority to tasks with $u_i \geq 1/2$
- ▶ assigns priorities to the rest according to deadlines

which reaches the generic schedulable utilization bound $(m + 1)/2$.

Partitioned vs Global

Advantages of the global scheduling:

- ▶ Load is automatically balanced
- ▶ Better average response time (follows from queueing theory)

Disadvantages of the global scheduling:

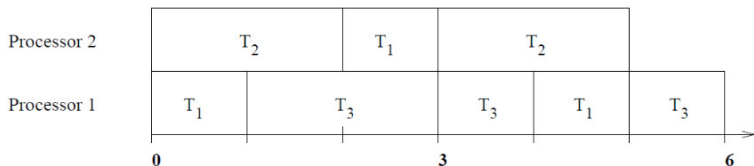
- ▶ Problems caused by migration (e.g. increased cache misses)
- ▶ Schedulability tests more difficult (active area of research)

Is either of the approaches better from the schedulability standpoint?

Global Beats Partitioned

There are sets of tasks schedulable only with global scheduler:

- ▶ $\mathcal{T} = \{T_1, T_2, T_3\}$ where $T_1 = (1, 2)$, $T_2 = (2, 3)$, $T_3 = (2, 3)$, can be scheduled using a global scheduler:

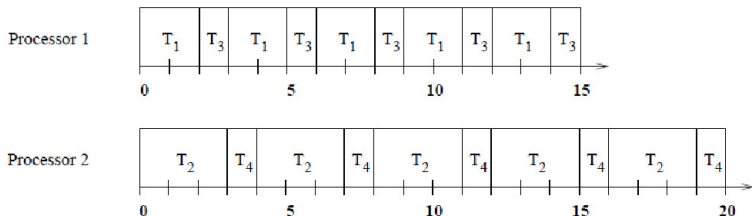


- ▶ No feasible partitioned schedule exists, always at least one processor gets tasks with total utilization higher than 1.

Partitioned Beats Global

There are task sets that can be scheduled only with partitioned scheduler (assuming fixed task-level priority assignment):

- ▶ $\mathcal{T} = \{T_1, \dots, T_4\}$ where $T_1 = (2, 3)$, $T_2 = (3, 4)$, $T_3 = (5, 15)$, $T_4 = (5, 20)$, can be scheduled using a fixed task-level priority partitioned schedule:



- ▶ Global scheduling (fixed job-level priority): There are 9 jobs released in the interval $[0, 12)$. Any of the $9!$ possible priority assignments leads to a deadline miss.

Optimal Algorithm?

There IS an optimal algorithm in the case of job-level migration & dynamic job-level priority. However, the algorithm is *time driven*.

The *priority fair* (PFair) algorithm is optimal for periodic systems with deadlines equal to periods

Idea (of PFair): In any interval $(0, t]$ jobs of a task T_i with utilization u_i execute for amount of time W so that $u_i t - 1 < W < u_i t + 1$

(Here every parameter is assumed to be a natural number)

This is achieved by cutting time into small quanta and scheduling jobs in these quanta so that the execution times are always (more or less) in proportion.

There are other optimal algorithms, all of them suffer from a large number of preemptions/migrations.

No optimal algorithms are known for more general settings: deadlines bounded by periods, arbitrary deadlines.

Recall, that no optimal *on-line* scheduling possible

Real-Time Programming & RTOS

Concurrent and real-time programming tools

Concurrent Programming

Concurrency in real-time systems

- ▶ typical architecture of embedded real-time system:
 - ▶ several input units
 - ▶ computation
 - ▶ output units
 - ▶ data logging/storing
- ▶ i.e., handling several concurrent activities
- ▶ concurrency occurs naturally in real-time systems

Support for concurrency in programming languages (Java, Ada, ...)
advantages: readability, OS independence, checking of interactions by compiler, embedded computer may not have an OS

Support by libraries and the operating system (C/C++ with POSIX)
advantages: multi-language composition, language's model of concurrency may be difficult to implement on top of OS, OS API standards imply portability

Processes and Threads

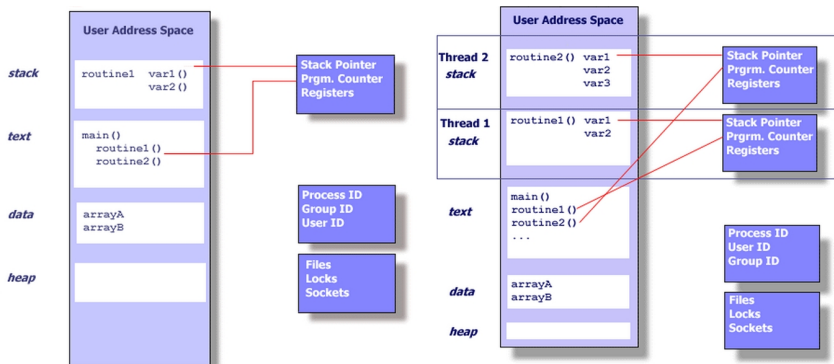
Process

- ▶ running instance of a program,
- ▶ executes its own virtual machine to avoid interference from other processes,
- ▶ contains information about program resources and execution state, e.g.:
 - ▶ environment, working directory, ...
 - ▶ program instructions,
 - ▶ registers, heap, stack,
 - ▶ file descriptors,
 - ▶ signal actions, inter-process communication tools (pipes, message boxes, etc.)

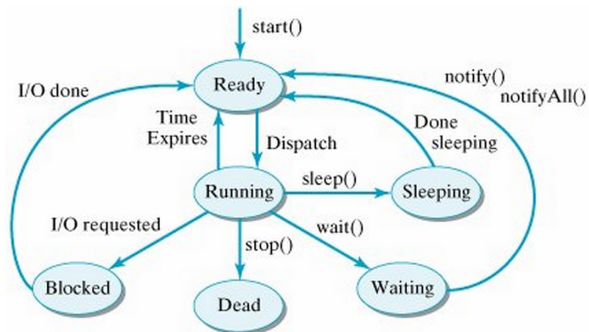
Thread

- ▶ exists *within a process*, uses process resources ,
- ▶ can be scheduled by OS and run as an independent entity,
- ▶ keeps its own: execution stack, local data, etc.
- ▶ share global data and resources with other threads of the same process

Processes and threads in UNIX



Process (Thread) States



Communication and Synchronization

Communication

- ▶ passing of information from one process (thread) to another
- ▶ typical methods: shared variables, message passing

Synchronization

- ▶ satisfaction of constraints on the interleaving of actions of processes
e.g. action of one process has to occur after an action of another one
- ▶ typical methods: semaphores, monitors

Communication and synchronization are linked:

- ▶ communication requires synchronization
- ▶ synchronization corresponds to communication without content

Communication: Shared Variables

Consistency problems:

- ▶ unrestricted use of shared variables is unreliable
- ▶ *multiple update problem*

example: shared variable X , assignment $X := X + 1$

- ▶ load the current value of X into a register
- ▶ increment the value of the register
- ▶ store the value of the register back to X
- ▶ two processes executing these instruction \Rightarrow certain interleavings can produce inconsistent results

Solution:

- ▶ parts of the process that access shared variables (i.e. critical sections) must be executed indivisibly with respect to each other
- ▶ required protection is called *mutual exclusion*

... one may use a special mutual ex. protocol (e.g. Peterson) or a synchronization mechanism – semaphores, monitors

Synchronization: Semaphores

A semaphore contains an integer variable that, apart from initialization, is accessed only through two standard operations: `wait()` and `signal()`.

- ▶ semaphore is initialized to a non-negative value (typically 1)
- ▶ `wait()` operation: decrements the semaphore value if the value is positive; otherwise, if the value is zero, the caller becomes blocked
- ▶ `signal()` operation: increments the semaphore value; if the value is not positive, then one process blocked by the semaphore is unblocked (usually in FIFO order)
- ▶ both `wait` and `signal` are atomic

Semaphores are elegant low-level primitive but error prone and hard to debug (deadlock, missing signal, etc.)

For more details see an operating systems course.

Synchronization: Monitors

- ▶ *encapsulation* and efficient condition synchronization
- ▶ critical regions are written as procedures; all encapsulated in a single object or module
- ▶ procedure calls into the module are guaranteed to be mutually exclusive
- ▶ shared resources accessible only by these procedures

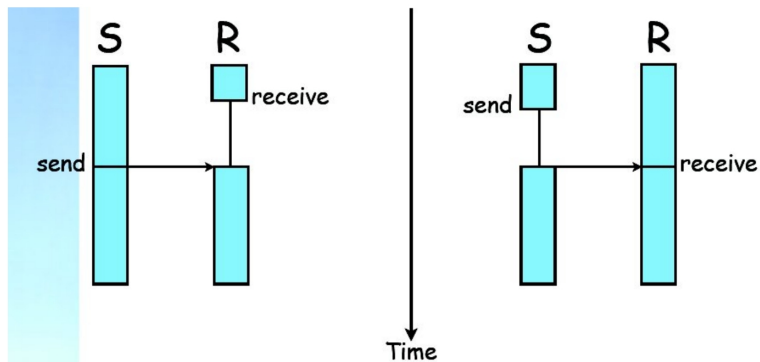
For more details (such as condition variables) see an operating systems course.

Communication: Message Passing

Communication among two, or more processes where there is no shared region between the two processes. Instead they communicate by passing messages.

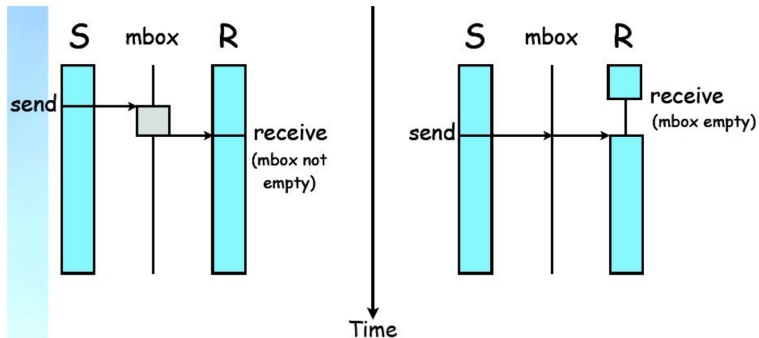
- ▶ **synchronous** (rendezvous): send and receive operations are blocking, no buffer required
- ▶ **asynchronous** (no-wait): send operation is not blocking, requires buffer space (mailbox)
- ▶ **remote invocation** (extended rendezvous): sender is blocked until reply is received

Synchronous Message Passing



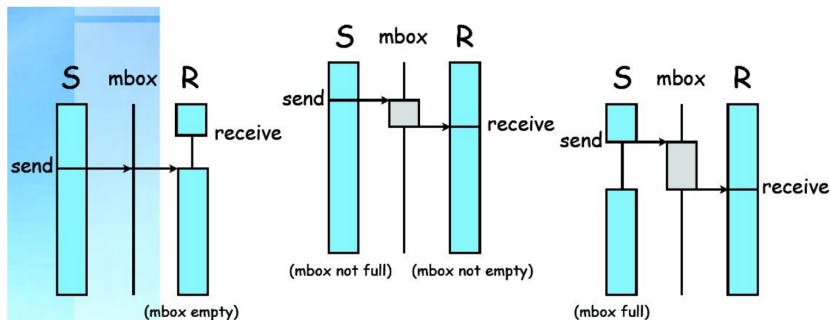
Both send and receive may block indefinitely!

Asynchronous Message Passing



Only the receiver might block indefinitely!

Asynch. Message Passing with Bounded Buffer



Receiver blocks if mbox is empty,
sender blocks if mbox is full

Note: size = 0 gives synchronous communication!

Concurrent Programming is Complicated

Multi-threaded applications with **shared data** may have numerous flaws.

- ▶ **Race condition**

Two or more threads try to access the same shared data, the result depends on the exact order in which their instructions are executed

- ▶ **Deadlock**

occurs when two or more threads wait for each other, forming a cycle and preventing all of them from making any forward progress

- ▶ **Starvation**

an indefinite delay or permanent blocking of one or more runnable threads in a multithreaded application

- ▶ **Livelock**

occurs when threads are scheduled but are not making forward progress because they are continuously reacting to each other's state changes

Usually difficult to find bugs and verify correctness.

Real-Time Aspects

- ▶ **time-aware systems** make explicit references to the time frame of the enclosing environment
 - e.g. a bank safe's door are to be locked from midnight to nine o'clock
 - ▶ the "real-time" of the environment must be available
- ▶ **reactive systems** are typically concerned with relative times
 - an output has to be produced within 50 ms of an associated input
 - ▶ must be able to measure intervals
 - ▶ usually must synchronize with environment: input sampling and output signalling must be done very regularly with controlled variability

The Concept of Time

Real-time systems must have a concept of time – but what is time?

- ▶ Measure of a time interval
 - ▶ Units?
seconds, milliseconds, cpu cycles, system "ticks"
 - ▶ Granularity, accuracy, stability of the clock source
 - ▶ Is "one second" a well defined measure?
"A second is the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium-133 atom."
 - ▶ ... temperature dependencies and relativistic effects
(the above definition refers to a caesium atom at rest, at mean sea level and at a temperature of 0 K)
 - ▶ Skew and divergence among multiple clocks
Distributed systems and clock synchronization
- ▶ Measuring time
 - ▶ external source (GPS, NTP, etc.)
 - ▶ internal – hardware clocks that count the number of oscillations that occur in a quartz crystal

Requirements for Interaction with "time"

For RT programming, it is desirable to have:

- ▶ access to clocks and representation of time
- ▶ delays
- ▶ timeouts
- ▶ (deadline specification and real-time scheduling)

Access to Clock and Representation of Time

- ▶ requires a hardware clock that can be read like a regular external device
- ▶ mostly offered by an OS service, if direct interfacing to the hardware is not allowed

Example of time representation: (POSIX high resolution clock, counting seconds and nanoseconds since 1970 with known resolution)

```
struct timespec {  
    time_t tv_sec;  
    long   tv_nsec;  
}
```

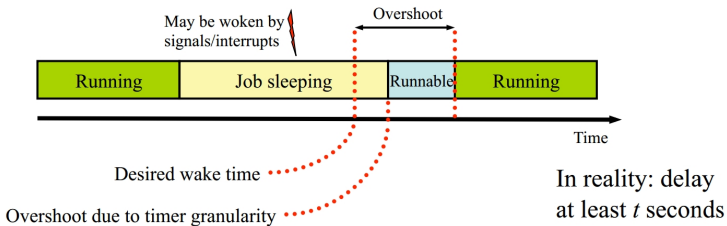
```
int clock_gettime( clockid_t clock_id,  
                  struct timespec * tp );  
int clock_settime( clockid_t id,  
                  const struct timespec * tp );
```

Time is often kept by incrementing an integer variable, need to take case of overflows (i.e. jumps to the past).

Delays

In addition to having access to a clock, need ability to

- ▶ Delay execution until an arbitrary calendar time
What about daylight saving time changes? Problems with leap seconds.
- ▶ Delay execution for a relative period of time
 - ▶ Delay for t seconds



- ▶ Delay for t seconds after event e begins

```
start = curr_time();  
do_action1();  
delay(10.0 - (curr_time() - start));  
do_action2();
```

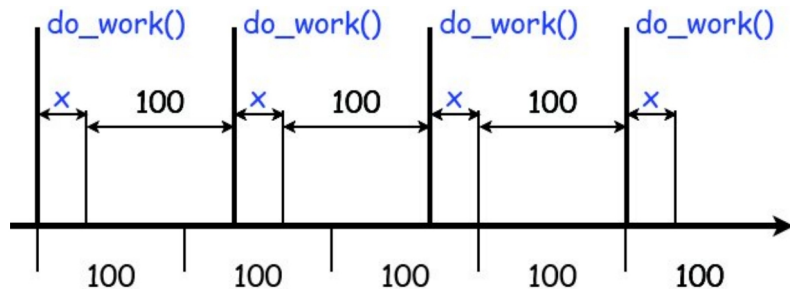
What if pre-empted between these?
Oversleep unless system has a function
`delay_until(start+10.0)`

A Repeated Task (An Attempt)

The goal is to do work repeatedly every 100 time units

```
while(1) {  
    delay(100);  
    do_work();  
}
```

Does it work as intended? No, accumulates drift ...



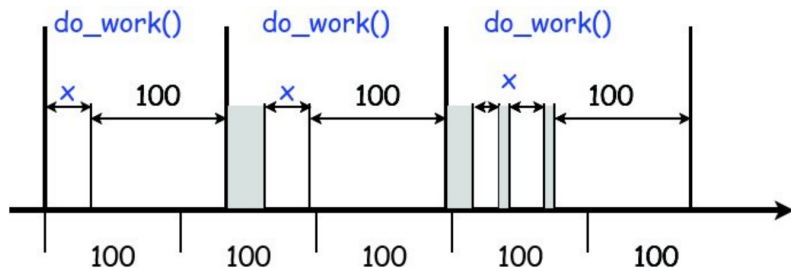
Each turn in the loop will take at least $100 + x$ milliseconds, where x is the time taken to perform `do_work()`

A Repeated Task (An Attempt)

The goal is to do work repeatedly every 100 time units

```
while(1) {  
    delay(100);  
    do_work();  
}
```

Does it work as intended? No, accumulates drift ...



Delay is just lower bound, a delaying process is not guaranteed access to the processor (the delay does not compensate for this)

Eliminating (Part of) The Drift: Timers

- ▶ Set an alarm clock, do some work, and then wait for **whatever time** is left before the alarm rings
- ▶ This is done with **timers**
- ▶ Thread is told to wait until the next ring – accumulating drift is eliminated
- ▶ Two types of timers
 - ▶ one-shot
After a specified interval call an associated function.
 - ▶ periodic (also called auto-reload timer in freeRTOS)
Call the associated function repeatedly, always after the specified interval.
- ▶ Even with timers, drift may still occur, but it does not accumulate (local drift)

Synchronous blocking operations can include timeouts

- ▶ Synchronization primitives

Semaphores, locks, etc.

... timeout usually generates an error/exception

- ▶ Networking and other I/O calls

E.g. `select()` in POSIX

Monitors readiness of multiple file descriptors, is ready when the corresponding operation with the file desc is possible without blocking.

Has a timeout argument that specifies the minimum interval that `select()` should block waiting for a file descriptor to become ready.

May also provide an asynchronous timeout signal

- ▶ Detect time overruns during execution of periodic and sporadic tasks

Deadline specification and real-time scheduling

Clock driven scheduling trivial to implement via cyclic executive.

Other scheduling algorithms need OS and/or language support:

- ▶ System calls create, destroy, suspend and resume tasks.
- ▶ Implement tasks as either threads or processes.
Threads usually more beneficial than processes (with separate address space and memory protection):
 - ▶ Processes not always supported by the hardware
 - ▶ Processes have longer context switch time
 - ▶ Threads can communicate using shared data (fast and more predictable)
- ▶ Scheduling support:
 - ▶ Preemptive scheduler with multiple priority levels
 - ▶ Support for aperiodic tasks (at least background scheduling)
 - ▶ Support for sporadic tasks with acceptance tests, etc.

Jobs, Tasks and Threads

- ▶ In theory, a system comprises a set of (abstract) *tasks*, each task is a series of *jobs*
 - ▶ tasks are typed, have various parameters, react to events, etc.
 - ▶ Acceptance test performed before admitting new tasks
- ▶ In practice, a *thread* (or a process) is the basic unit of work handled by the scheduler
 - ▶ Threads are the instantiation of tasks that have been admitted to the system

How to map *tasks* to *threads*?

Periodic Tasks

Real-time tasks defined to execute periodically $T = (\phi, p, e, D)$

It is clearly inefficient if the thread is created and destroyed repeatedly every period

- ▶ Some op. systems (funkOS) and programming languages (Real-time Java & Ada) support *periodic threads*
 - ▶ the kernel (or VM) reinitializes such a thread and puts it to sleep when the thread completes
 - ▶ The kernel releases the thread at the beginning of the next period
 - ▶ This provides clean abstraction but needs support from OS
- ▶ Thread instantiated once, performs job, sleeps until next period, repeats
 - ▶ Lower overhead, but relies on programmer to handle timing
 - ▶ Hard to avoid timing drift due to sleep overruns
(see the discussion of delays earlier in this lecture)
 - ▶ Most common approach

Sporadic and Aperiodic Tasks

Events trigger sporadic and aperiodic tasks

- ▶ Might be external (hardware) interrupts
- ▶ Might be signalled by another task

Usual implementation:

- ▶ OS executes periodic server thread
(background server, deferrable server, etc.)
- ▶ OS maintains a “server queue” = a list of pointers which give starting addresses of functions to be executed by the server
- ▶ Upon the occurrence of an event that releases an aperiodic or sporadic job, the event handler (usually an interrupt routine) inserts a pointer to the corresponding function to the list

Real-Time Programming & RTOS

Real-Time Operating systems

Operating Systems – What You Should Know ...

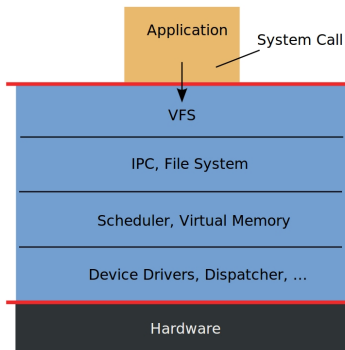
An operating system is a collection of software that manages computer hardware resources and provides common services for computer programs.

Basic components multi-purpose OS:

- ▶ Program execution & process management
processes (threads), IPC, scheduling, ...
- ▶ Memory management
segmentation, paging, protection ...
- ▶ Storage & other I/O management
files systems, device drivers, ...
- ▶ Network management
network drivers, protocols, ...
- ▶ Security
user IDs, privileges, ...
- ▶ User interface
shell, GUI, ...

Operating Systems – What You Should Know ...

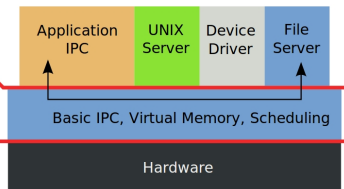
Monolithic Kernel based Operating System



Microkernel based Operating System

user mode

kernel mode



Implementing Real-Time Systems

- ▶ Key fact from scheduler theory: *need predictable behavior*
 - ▶ Raw performance less critical than consistent and predictable performance; hence focus on scheduling algorithms, schedulability tests
 - ▶ Don't want to fairly share resources – be unfair to ensure deadlines met
- ▶ Need to run on a wide range of – often custom – hardware
 - ▶ Often resource constrained:
limited memory, CPU, power consumption, size, weight, budget
 - ▶ Closed set of applications
(Do we need a wristwatches to play DVDs?)
 - ▶ Strong reliability requirements – may be safety critical
 - ▶ How to upgrade software in a car engine? A DVD player?

Implications on Operating Systems

- ▶ General purpose operating systems not well suited for real-time
 - ▶ Assume plentiful resources, fairly shared amongst untrusted users
 - ▶ Serve multiple purposes
 - ▶ Exactly opposite of an RTOS!
 - ▶ Instead want an operating system that is:
 - ▶ Small and light on resources
 - ▶ Predictable
 - ▶ Customisable, modular and extensible
 - ▶ Reliable
- ... and that can be *demonstrated* or *proven* to be so

Implications on Operating Systems

- ▶ Real-time operating systems typically either *cyclic executive* or *microkernel* designs, rather than a traditional monolithic kernel
 - ▶ Limited and well defined functionality
 - ▶ Easier to demonstrate correctness
 - ▶ Easier to customise
- ▶ Provide rich support for concurrency & real-time control
- ▶ Expose low-level system details to the applications
control of scheduling, interaction with hardware devices, ...

Cyclic Executive without Interrupts

- ▶ The simplest real-time systems use a “nanokernel” design
 - ▶ Provides a minimal time service: scheduled clock pulse with a fixed period
 - ▶ No tasking, virtual memory/memory protection etc.
 - ▶ Allows implementation of a static cyclic schedule, provided:
 - ▶ Tasks can be scheduled in a frame-based manner
 - ▶ All interactions with hardware to be done on a polled basis
- ▶ Operating system becomes a single task cyclic executive

```
setup timer
c = 0;
while (1) {
    suspend until timer expires
    c++;
    do tasks due every cycle
    if (((c+0) % 2) == 0) do tasks due every 2nd cycle
    if (((c+1) % 3) == 0) {
        do tasks due every 3rd cycle, with phase 1
    }
    ...
}
```

Microkernel Architecture

- ▶ Cyclic executive widely used in low-end embedded devices
 - ▶ 8 bit processors with kilobytes of memory
 - ▶ Often programmed in (something like) C via cross-compiler, or assembler
 - ▶ Simple hardware interactions
 - ▶ Fixed, simple, and static task set to execute
 - ▶ Clock driven scheduler
- ▶ But many real-time embedded systems are more complex, need a sophisticated operating system with priority scheduling
- ▶ Common approach: a *microkernel* with priority scheduler
Configurable and robust, since architected around interactions between cooperating system servers, rather than a monolithic kernel with ad-hoc interactions

- ▶ A microkernel RTOS typically provides:
 - ▶ Timing services, interrupt handling, support for hardware interaction
 - ▶ Task management, scheduling
 - ▶ Messaging, signals
 - ▶ Synchronization and locking
 - ▶ Memory management (and sometimes also protection)

Example RTOS: FreeRTOS

- ▶ RTOS for embedded devices (ported to many microcontrollers from more than 20 manufacturers)
- ▶ Distributed under GPL
- ▶ Written in C, kernel consists of 3+1 C source files (approx. 9000 lines of code including comments)
- ▶ Largely configurable

Example RTOS: FreeRTOS

- ▶ The OS is (more or less) a library of object modules; the application and OS modules are linked together in the resulting executable image
- ▶ Prioritized scheduling of tasks
 - ▶ tasks correspond to threads (share the same address space; have their own execution stacks)
 - ▶ highest priority executes; same priority ⇒ round robin
 - ▶ implicit *idle task* executing when no other task executes ⇒ may be assigned functionality of a background server
- ▶ Synchronization using semaphores
- ▶ Communication using message queues
- ▶ Memory management
 - ▶ no memory protection in basic version (can be extended)
 - ▶ various implementations of memory management
 - memory can/cannot be freed after allocation, best fit vs combination of adjacent memory block into a single one

That's (almost) all

Example RTOS: FreeRTOS

Tiny memory requirements: e.g. IAR STR71x ARM7 port, full optimisation, minimum configuration, four priorities ⇒

- ▶ size of the scheduler = 236 bytes
- ▶ each queue adds 76 bytes + storage area
- ▶ each task 64 bytes + the stack size

Details of FreeRTOS Scheduling

- ▶ The scheduler must be explicitly invoked by calling `void vTaskStartScheduler(void)` from `main()`.

The scheduler may also stop either due to error, or if one of the tasks calls `void vTaskEndScheduler(void)`.

- ▶ It is possible to create a new task by calling

```
portBASE_TYPE xTaskCreate(  
    pdTASK_CODE pvTaskCode,  
    const char * const pcName,  
    unsigned short usStackDepth,  
    void *pvParameters,  
    unsigned portBASE_TYPE uxPriority,  
    xTaskHandle *pvCreatedTask);
```

`pvTaskCode` is a pointer to a function that will be executed as the task, `pcName` is a human-readable name of the task, `usStackDepth` indicates how many words must be reserved for the task stack, `pvParameters` is a pointer to parameters of the task (without interpretation by the OS), `uxPriority` is the assigned priority of the task (see resource control lecture 7), `pvCreatedTask` is the task handle used by OS routines.

Details of FreeRTOS Scheduling

- ▶ A task can be deleted by means of `void vTaskDelete(xTaskHandle pxTaskToDelete)`
 - ▶ Like most other (non-POSIX-compliant) small real-time systems, does not provide a task cancellation mechanism, i.e. tasks cannot decline, or postpone deletion – the deletion is immediate.
 - ▶ Memory is not freed immediately, only the idle task can do it that must be executed occasionally.
 - ▶ A shared resource owned by a deleted task remains locked.
- ▶ Priorities are handled by means of `uxTaskPriorityGet` and `uxTaskPrioritySet`. FreeRTOS implements priority inheritance protocol, the returned priorities are the current ones.
- ▶ Tasks can be suspended `vTaskSuspend` or `vTaskSuspendAll` (suspends of but the calling one), and resumed by `vTaskResume` or `vTaskResumeAll`. Suspend/resume all can be used to implement non-preemptable critical sections.

Clocks & Timers in FreeRTOS

- ▶ `portTickType xTaskGetTickCount(void)`

Get current time, in ticks, since the scheduler was started.

The frequency of ticks is determined by `configTICK_RATE_HZ` set w.r.t. the HW port.

- ▶ `void vTaskDelay(portTickType xTicksToDelay)`

Blocks the calling task for the specified number of ticks.

- ▶ `void vTaskDelayUntil(
 portTickType *pxPreviousWakeTime,
 portTickType xTimeIncrement
);`

Blocks the calling process for `xTimeIncrement` ticks since the `pxPreviousWakeTime`.

(At the wakeup, the `pxPreviousWakeTime` is incremented by `xTimeIncrement` so that it can be readily used to implement periodic tasks.)

Real-Time Programming & RTOS

Real-Time Programming Languages

Brief Overview

IEEE 1003 POSIX

- ▶ "Portable Operating System Interface"
- ▶ Defines a subset of Unix functionality, various (optional) extensions added to support real-time scheduling, signals, message queues, etc.
- ▶ Widely implemented:
 - ▶ Unix variants and Linux
 - ▶ Dedicated real-time operating systems
 - ▶ Limited support in Windows

Several POSIX standards for real-time scheduling

- ▶ POSIX 1003.1b ("real-time extensions")
- ▶ POSIX 1003.1c ("pthreads")
- ▶ POSIX 1003.1d ("additional real-time extensions")
- ▶ Support a sub-set of scheduler features we have discussed

POSIX Scheduling API (Threads)

```
#include <unistd.h>
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_getschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *p);
int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *p);

int pthread_create(pthread_t      *thread,
                  pthread_attr_t *attr,
                  void *(*thread_func)(void*),
                  void  *thread_arg);

int pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
```

`struct sched_param` typically contains only `sched_priority`.

`pthread_join` suspends execution of the thread until termination of the thread; `retval` of the terminating thread is available to any successfully joined thread.

Threads: Example I

```
#include <pthread.h>

pthread_t id;
void *fun(void *arg) {
    // Some code sequence
}

main() {
    pthread_create(&id, NULL, fun, NULL);
    // Some other code sequence
}
```

Threads: Example II

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS    5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```


POSIX: Synchronization and Communication

- ▶ Synchronization:
 - ▶ mutexes
(variables that can be locked/unlocked by threads),
 - ▶ (counting) semaphores,
 - ▶ condition variables,
(Used to wait for some condition. The waiting thread is put into a queue until signaled on the condition variable by another thread.)
 - ▶ ...
- ▶ Communication:
 - ▶ signals (`kill(pid, sig)`),
 - ▶ message passing,
 - ▶ shared memory.

POSIX: Real-Time Support

Getting Time

- ▶ `time()` = seconds since Jan 1 1970
- ▶ `gettimeofday()` = seconds + nanoseconds since Jan 1 1970
- ▶ `tm` = structure for holding human readable time

```
struct tm {
    int    tm_sec;    // seconds (0 - 60)
    int    tm_min;    // minutes (0 - 59)
    int    tm_hour;   // hours (0 - 23)
    int    tm_mday;   // day of month (1 - 31)
    int    tm_mon;    // month of year (0 - 11)
    int    tm_year;   // year - 1900
    int    tm_wday;   // day of week (Sunday = 0)
    int    tm_yday;   // day of year (0 - 365)
    int    tm_isdst;  // is summer time in effect?
    char   *tm_zone;  // timezone name
    long   tm_gmtoff; // offset from UTC
};
struct tm localtime(time_t t);
time_t    mktime(struct tm *t);
```

- ▶ POSIX requires at least one clock of minimum resolution 50Hz (20ms)

POSIX: High Resolution Time & Timers

High resolution clock. Known clock resolution.

```
struct timespec {  
    time_t tv_sec;  
    long   tv_nsec;  
}
```

```
int clock_gettime( clockid_t clock_id,  
                  struct timespec * tp );  
int clock_settime( clockid_t id,  
                  const struct timespec * tp );
```

Simple waiting: sleep, or

```
int nanosleep(struct timespec *delay, struct timespec *remaining);
```

Sleep for the interval specified. May return earlier due to signal (in which case remaining gives the remaining delay).

Accuracy of the delay not known (and not necessarily correlated to clock_getres() value)

POSIX: Timers

- ▶ type `timer_t`; can be set:
 - ▶ relative/absolute time
 - ▶ single alarm time and an optional repetition period
- ▶ timer “rings” according to `sevp` (e.g. by sending a signal)

```
int timer_create(clockid_t clockid, struct sigevent *sevp,  
                timer_t *timerid);
```

```
int timer_settime(timer_t timerid, int flags,  
                  const struct itimerspec *new_value,  
                  struct itimerspec * old_value);
```

where

```
struct itimerspec {  
    struct timespec it_interval; /* Timer interval */  
    struct timespec it_value;    /* Initial expiration */  
};
```

POSIX Scheduling API

- ▶ Four scheduling policies:
 - ▶ SCHED_FIFO = Fixed priority, pre-emptive, FIFO on the same priority level
 - ▶ SCHED_RR = Fixed priority, pre-emptive, round robin on the same priority level
 - ▶ SCHED_SPORADIC = Sporadic server
 - ▶ SCHED_OTHER = Unspecified (often the default time-sharing scheduler)
- ▶ A process can `sched_yield()` or otherwise block at any time
- ▶ POSIX 1003.1b provides (largely) fixed priority scheduling
 - ▶ Priority can be changed using `sched_set_param()`, but this is high overhead and is intended for reconfiguration rather than for dynamic scheduling
 - ▶ No direct support for dynamic priority algorithms (e.g. EDF)
- ▶ Limited set of priorities:
 - ▶ Use `sched_get_priority_min()`, `sched_get_priority_max()` to determine the range
 - ▶ Guarantees at least 32 priority levels

Using POSIX Scheduling: Rate Monotonic

Rate monotonic and deadline monotonic schedules can be naturally implemented using POSIX primitives

1. Assign priorities to tasks in the usual way for RM/DM
2. Query the range of allowed system priorities
`sched_get_priority_min()` and
`sched_get_priority_max()`
3. Map task set onto system priorities
Care needs to be taken if there are large numbers of tasks, since some systems only support a few priority levels
4. Start tasks using assigned priorities and `SCHED_FIFO`

There is no explicit support for indicating deadlines, periods

EDF scheduling not supported by POSIX

Using POSIX Scheduling: Sporadic Server

POSIX 1003.1d defines a hybrid sporadic/background server

```
struct sched_param {
    int          sched_priority;
    int          sched_ss_low_priority;
    struct timespec sched_ss_repl_period;
    struct timespec sched_ss_init_budget;
};
```

When server has budget, runs at `sched_priority`, otherwise runs as a background server at `sched_ss_low_priority`

Set `sched_ss_low_priority` to be lower priority than real-time tasks, but possibly higher than other non-real-time tasks in the system

Also defines the replenishment period and the initial budget after replenishment

Examples of POSIX-compliant implementations:

- ▶ commercial:
 - ▶ VxWorks
 - ▶ QNX
 - ▶ OSE
- ▶ Linux-related:
 - ▶ RTLinux
 - ▶ RTAI

(Some) sources of hard to predict latency caused by the system:

- ▶ **Interrupts**
see next slide
- ▶ **System calls**
RTOS should characterise WCET; kernel should be preemptable
- ▶ **Memory management: paging**
avoid, either use segmentation with a fixed memory management scheme, or memory locking
- ▶ **Caches**
may introduce non-determinism; there are techniques for computing WCET with processor caches
- ▶ **DMA**
competes with processor for the memory bus, hard to predict who wins

The amount of time required to handle interrupt varies

Thus in most OS, interrupt handling is divided into two steps

- ▶ Immediate interrupt service
very short; invokes a scheduled interrupt handling routine
- ▶ Scheduled interrupt service
preemptable, scheduled as an ordinary job at a suitable priority

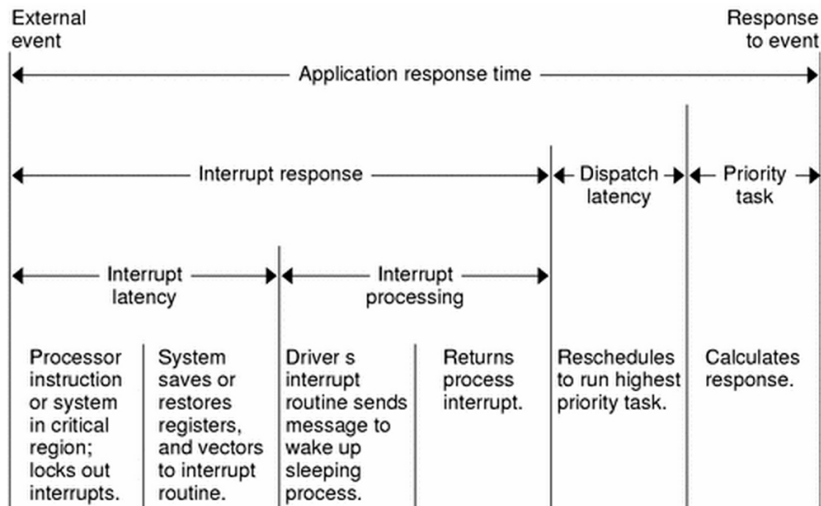
Immediate Interrupt Service

Interrupt latency is the time between interrupt request and execution of the first instruction of the interrupt service routine

The total delay caused by interrupt is the sum of the following factors:

- ▶ the time the processor takes to complete the current instruction, do the necessary chores (flush pipeline and read the interrupt vector), and jump to the trap handler and interrupt dispatcher
- ▶ the time the kernel takes to disable interrupts
- ▶ the time required to complete the immediate interrupt service routines with higher-priority interrupts (if any) that occurred simultaneously with this one
- ▶ the time the kernel takes to save the context of the interrupted thread, identify the interrupting device, and get the starting address of the interrupt service routine
- ▶ the time the kernel takes to start the interrupt service routine

Event Latency



- ▶ **object-oriented** programming language
- ▶ developed by Sun Microsystems in the early 1990s
- ▶ compiled to **bytecode** (for a *virtual machine*), which is compiled to native machine code at runtime
- ▶ syntax of Java is largely derived from C/C++

Concurrency: Threads

- ▶ predefined class `java.lang.Thread` – provides the mechanism by which threads are created
- ▶ to avoid all threads having to be child classes of `Thread`, it also uses a standard interface:

```
public interface Runnable {  
    public abstract void run();  
}
```

- ▶ any class which wishes to express concurrent execution must implement this interface and provide the `run()` method

Threads: Creation & Termination

Creation:

- ▶ **dynamic thread creation**, arbitrary data to be passed as parameters
- ▶ thread hierarchies and thread groups can be created

Termination:

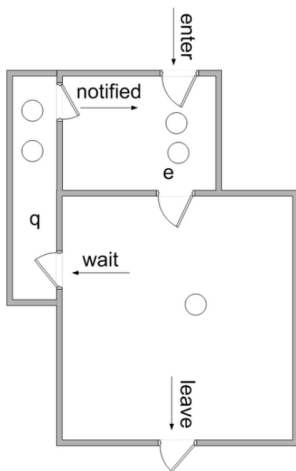
- ▶ one thread can wait for another thread (the target) to terminate by issuing the **join method** call on the target's thread object
- ▶ the **isAlive** method allows a thread to determine if the target thread has terminated
- ▶ garbage collection cleans up objects which can no longer be accessed
- ▶ main program terminates when all its user threads have terminated

Synchronized Methods

- ▶ **monitors** are implemented in the context of classes and objects
- ▶ **lock** associated with **each object**; lock cannot be accessed directly by the application but is affected by
 - ▶ the method modifier **synchronized**
 - ▶ block synchronization
- ▶ **synchronized method** – access to the method can only proceed once the lock associated with the object has been obtained
- ▶ **non-synchronized methods** do not require the lock, can be called at any time

Waiting and Notifying

- ▶ `wait()` always blocks the calling thread and releases the lock associated with the object
- ▶ `notify()` wakes up one waiting thread
which thread is woken is not defined
- ▶ `notifyAll()` wakes up all waiting threads
- ▶ if no thread is waiting, then `notify()` and `notifyAll()` have no effect



Real-Time Java

- ▶ Standard Java is not enough to handle real-time constraints
- ▶ Java (and JVM) lacks semantic for standard real-time programming techniques.
- ▶ Embedded Java Specification was there, but merely a subset of standard Java API.
- ▶ There is a gap for a language real-time capable and equipped with all Java's powerful advantages.

- ▶ IBM, Sun and other partners formed Real-time for Java Expert Group sponsored by NIST in 1998
- ▶ It came up with Real-Time Specification for Java (RTSJ) to fill this gap for real-time systems
- ▶ RTSJ proposed seven areas of enhancements to the standard Java

RTSJ – Areas of Enhancement

1. Thread scheduling and dispatching
see the next slides
2. Memory management
immortal memory (no garbage collection), threads not preemptable by garbage collector
3. Synchronization and resource sharing
priority inheritance and ceiling protocols
4. Asynchronous event handling, asynchronous transfer of control, asynchronous thread termination
reaction to OS-level signals (POSIX), hardware interrupts and custom events defined and fired by the application
5. Physical memory access

The resulting real-time extension needs a modified Java virtual machine due to changes to memory model, garbage collector and thread scheduling

- ▶ `java.lang.System.currentTimeMillis` returns the number of milliseconds since Jan 1 1970
- ▶ Real Time Java adds real time clocks with high resolution time types

Timers

- ▶ one shot timers (`javax.realtime.OneShotTimer`)
- ▶ periodic timers (`javax.realtime.PeriodicTimer`)

Constructor:

`Timer(HighResolutionTime t, Clock c, AsyncEventHandler handler)`
... create a timer that fires at time `t`, according to `Clock c` and is handled by the specified handler.

Real-Time Thread Scheduling

- ▶ Extends Java with a schedulable interface and `RealtimeThread` class, and numerous supporting libraries
- ⇒ Allows definition of timing and scheduling parameters, and memory requirements of threads
- ▶ `Abstract Scheduler` and `SchedulingParameters` classes defined
 - ▶ Allows a range of schedulers to be developed
 - ▶ Current standards only allow system-defined schedulers; cannot write a new scheduler without modifying the JVM
 - ▶ Current standards provide a pre-emptive fixed priority scheduler (`PriorityScheduler` class)
 - ▶ Allows monitoring of execution times; missed deadlines; CPU budgets
 - ▶ Allows thread priority to be changed programatically
 - ▶ Limited support for acceptance tests (`isFeasible()`)

Real-Time Thread Scheduling

```
abstract class ReleaseParameters
{
    RelativeTime      cost
    RelativeTime      deadline
    AsyncEventHandler overrunHandler
    AsyncEventHandler missHandler
    ...
}
```

Extends

```
class PeriodicParameters
{
    HighResolutionTime start
    RelativeTime          period
    ...
}
```

```
class AperiodicParameters
{
    ...
}
```

```
class SporadicParameters
{
    RelativeTime minInterarrival
    ...
}
```

- ▶ Class hierarchy to express release timing parameters
- ▶ Deadline monitoring: missHandler if deadline exceeded
- ▶ Execution time monitoring:
 - ▶ cost = needed CPU time
 - ▶ overrunHandler if execution time budget exceeded

Real-Time Thread Scheduling

```
class RealtimeThread extends java.lang.Thread
{
    // ...adds additional constructors to specify
    // ReleaseParameters and SchedulingParameters
    ...

    // ...adds additional methods:
    public void      setScheduler(Scheduler s);
    public void      schedulePeriodic();
    public boolean   waitForNextPeriod();
    ...
}
```

- ▶ The RealtimeThread class extends Thread with extra methods and parameters
 - ▶ Direct support for periodic threads
 - ▶ run() method will be a loop ending in a waitForNextPeriod() call
 - ▶ ... i.e. does not have to be implemented using sleep as e.g. with POSIX API

- ▶ designed for United States Department of Defense during 1977-1983
- ▶ **targeted at embedded and real-time systems**
- ▶ Ada 95 revision
- ▶ used in critical systems (avionics, weapon systems, spacecrafts)
- ▶ free compiler: gnat

... see the lecture of Petr Holub.



Ada Lovelace
(1815-1852)