

Real-Time Embedded Systems

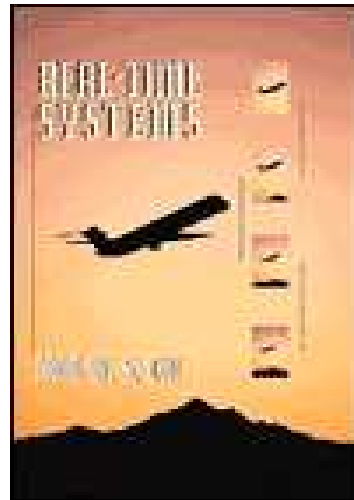
Wan Fokkink

Vrije Universiteit Amsterdam & CWI

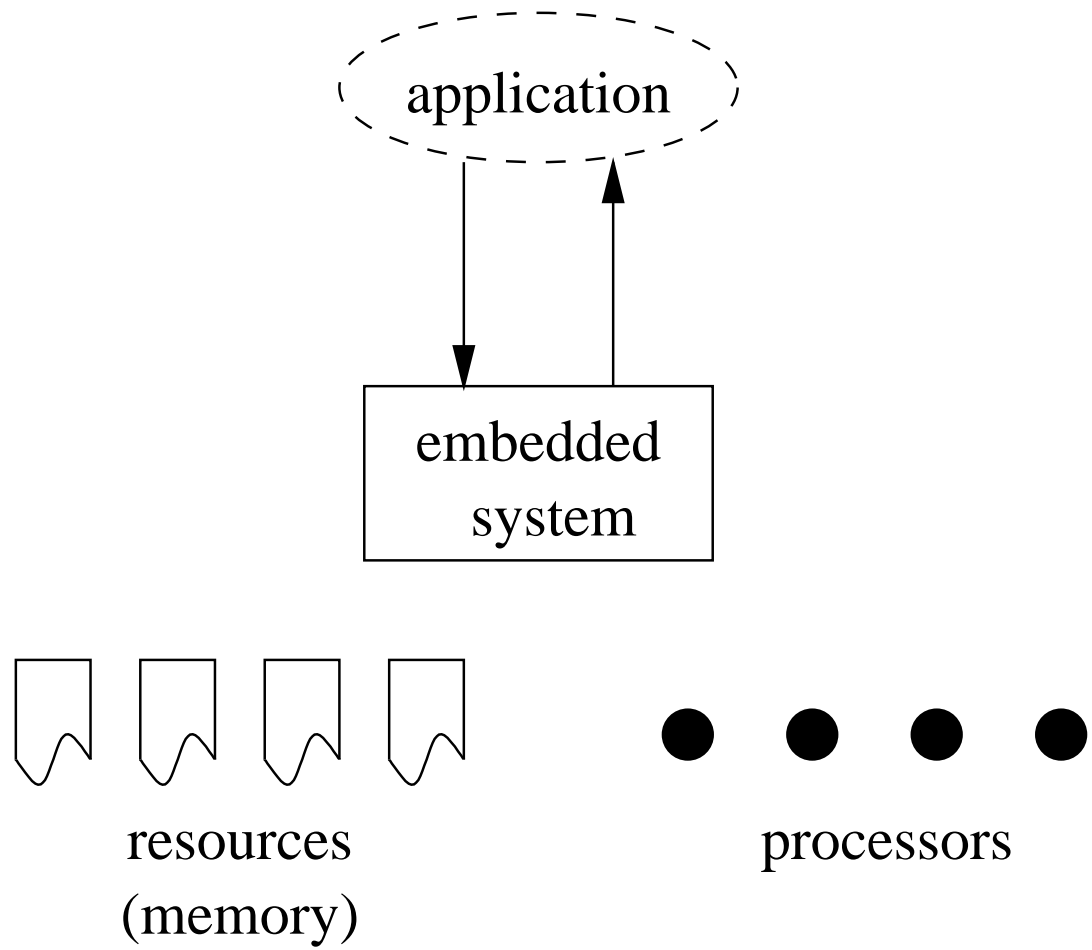
Jane W.S. Liu

Real-Time Systems

Prentice Hall, 2000



General Picture



Resources are allocated to processors.

Jobs

A **job** is a unit of work, scheduled and executed by the system.

Parameters of jobs are:

- functional behavior
- time constraints
- resource requirements

Jobs are divided over **processors**, and they are competing for

A **scheduler** decides in which order jobs are performed on a processor and which resources they can claim.

Terminology

release time: when a job becomes available for execution

execution time: amount of processor time needed to perform (assuming it executes alone and all resources are available)

response time: length of time from arrival until completion of

(absolute) deadline: when a job is required to be completed

relative deadline: maximum allowed response time

hard deadline: late completion not allowed

soft deadline: late completion allowed

jitter: imprecise release and/or execution time

A **preemptive** job can be suspended at any time of its execution

Out of scope:

- use of distant resources
- communication between jobs
- migration of jobs
- overrun management
- penalty for missing a soft deadline
- performance
- different processor and resource types

Types of Tasks

A **task** is a set of related jobs.

A processor distinguishes three types of tasks:

- **periodic**: known input before the start of the system, and deadlines.

Execution and interarrival times are fixed.

- **aperiodic**: executed in response to some external event, deadlines.

Execution and interarrival times are according to some *p* *distribution*.

- **sporadic**: executed in response to some external event, deadlines.

Execution times are according to some *probability distribution* interarrival times are *random*.

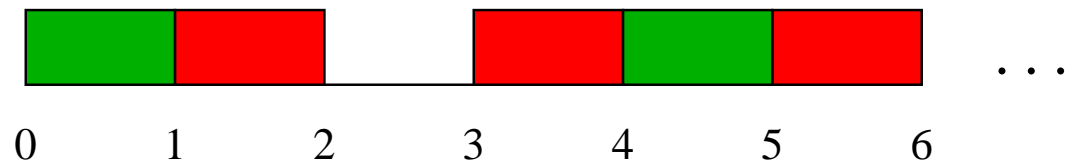
Periodic Tasks

A periodic task is defined by:

- **release time** r (of the first periodic job)
- **period** p (regular time interval, at the start of which a job is released)
- **execution time** e

For simplicity we assume that the relative deadline of each job is equal to its period.

Example: $T_1 = (1, 2, 1)$ and $T_2 = (0, 3, 1)$.

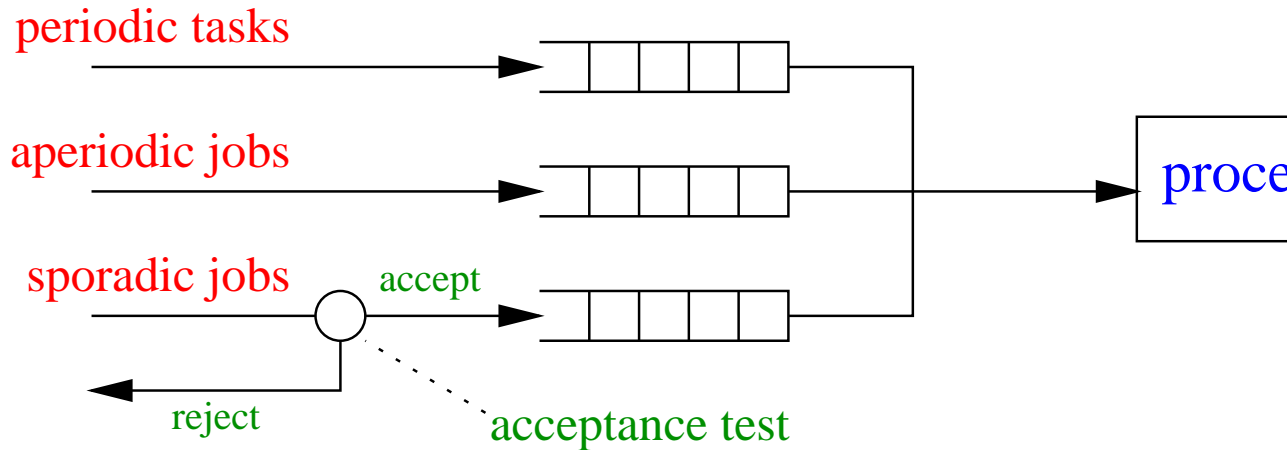


The conflict at time 3 is resolved by some scheduler.

The **hyperperiod** is 6.

Job Queues at a Processor

We focus on individual aperiodic and sporadic *jobs*.



- *Sporadic jobs* are only accepted when they can be completed before their deadline.
- *Aperiodic jobs* are always accepted, and performed such that periodic and accepted sporadic jobs do not miss their deadlines.

Average Response Time

The queueing discipline of aperiodic jobs tries to minimize e.g. **tardiness** (completion time minus deadline) or the number of soft deadlines.

The **average response time** of aperiodic jobs can be analyzed

- simulation and measurement
- Queueing Theory
- Integer Linear Programming

In the last two cases, values for e.g. average execution time of aperiodic jobs must in general still be estimated using simulation or measurement.

Scheduler

The **scheduler** of a processor schedules and allocates resources (according to some **scheduling algorithms** and **resource access protocols**).

A schedule is **valid** if:

- jobs are not scheduled before their release times
- the total amount of processor time assigned to a job equals its (maximum) execution time

A (valid) schedule is **feasible** if all hard deadlines are met.

A scheduler is **optimal** if it produces a feasible schedule whenever possible.

Clock-Driven Scheduler

Off-line scheduling: the schedule for periodic tasks is computed beforehand (typically with an algorithm for an NP-complete problem).

Time is divided into regular time intervals called **frames**.

In each frame, a predetermined set of periodic tasks is executed.

Jobs may be sliced into subjobs, to accommodate frame length.

Clock-driven scheduling is conceptually simple, but cannot cope with:

- jitter
- system modifications
- nondeterminism

Slack

Idle time in a frame can be used to execute aperiodic and sporadic jobs.

Slack of a frame $[s, c]$ at time t is $t - c$ minus the total execution of periodic and accepted sporadic jobs in the frame after t .

Slack stealing: execution of aperiodic jobs until the slack of current frame is zero.

Acceptance test: straightforward check (in absence of jitter) whether a newly arrived sporadic job can be completed before its deadline (whether there is sufficient slack).

Resources: are in general distributed according to some periodic cyclic schedule.

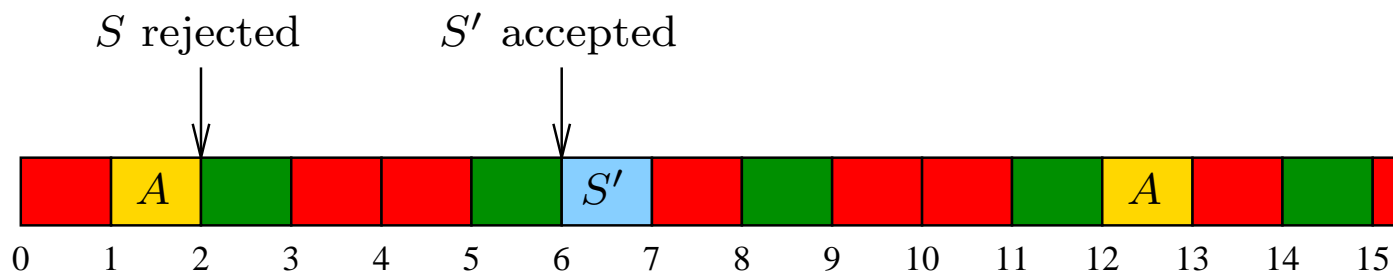
Example: Periodic jobs $T_1 = (0, 2, 1)$ and $T_2 = (0, 3, 1)$.

Frame length is 6.

Aperiodic job A , with execution time 2, arrives at 1.

Sporadic job S , with execution time 1, arrives at 2 with deadline 3.

Sporadic job S' , with execution time 1, arrives at 6 with deadline 7.



Priority-Driven Scheduling

On-line scheduling: the schedule is computed at run-time.

Scheduling decisions are taken when:

- periodic jobs are released or aperiodic/sporadic jobs arrive
- jobs are completed
- resources are required or released

Released jobs are placed in **priority queues**, e.g. ordered by:

- release time (FIFO, LIFO)
- execution time (SETF, LETF)
- period of the task (RM)
- deadline (EDF) or slack (LST)

We focus on EDF scheduling.

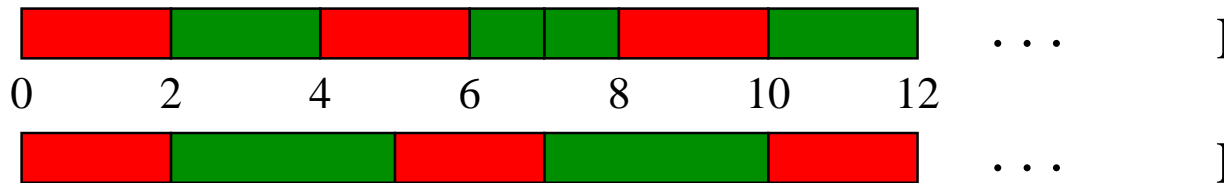
RM Scheduler

Rate Monotonic: Shorter period gives a higher priority.

Advantage: Priority on the level of tasks makes RM easier to implement than EDF/LST.

Non-optimality of the RM scheduler (one processor, preemptive, no competition for resources):

Let $T_1 = (0, 4, 2)$ and $T_2 = (0, 6, 3)$.



Remark: If for periods $p < p'$, p is always a divisor of p' , the scheduler is optimal.

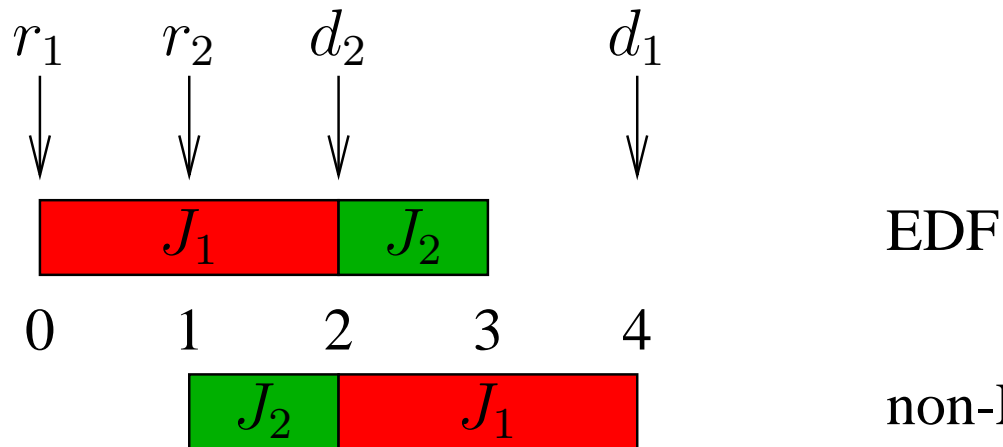
EDF Scheduler

Earliest Deadline First: the earlier the deadline, the higher the priority.

Consider a single processor, and preemptive jobs.

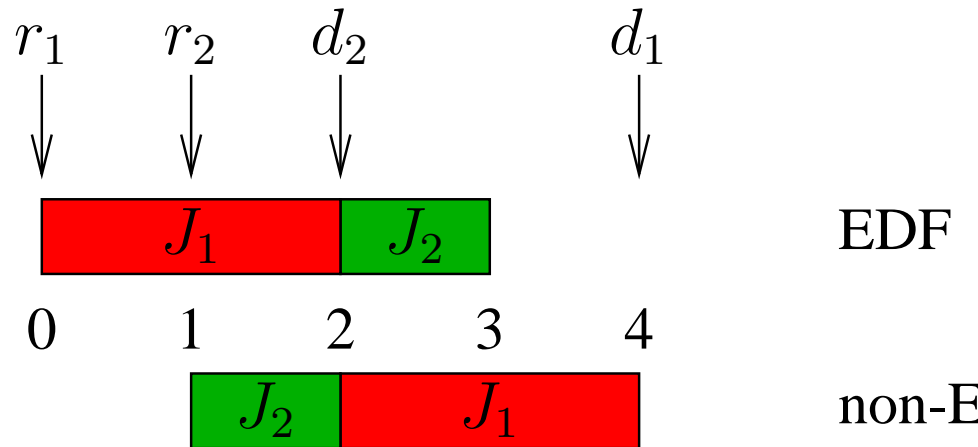
Theorem: When jobs do not compete for resources, the EDF scheduler is optimal.

Non-optimality in case of non-preemption:

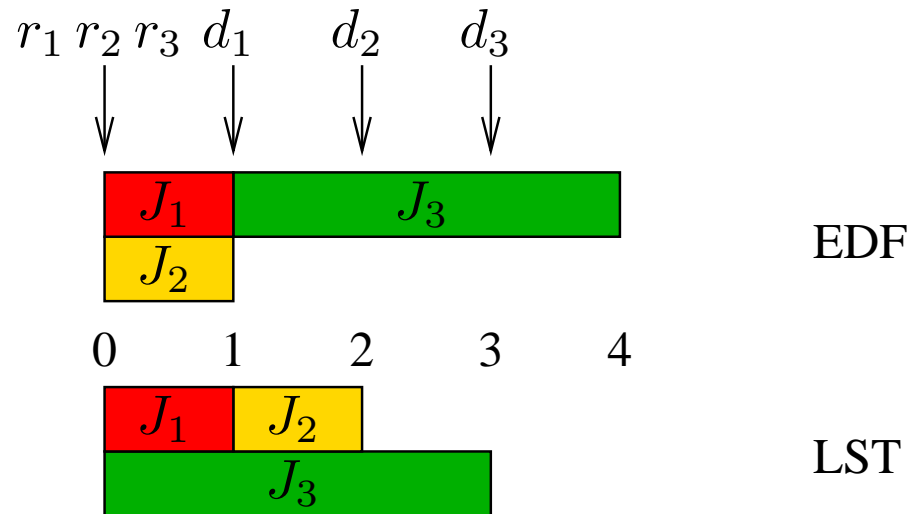


Non-optimality in case of resource competition:

Let J_1 and J_2 both require resource R .



Non-optimality in case of two processors (with migration):



Drawbacks of EDF:

- dynamic priority of periodic tasks makes it difficult to analyze which deadlines are met in case of **overloads**
- late jobs can cause other jobs to miss their deadlines (good **overrun management** is needed)

LST Scheduler

Least Slack Time first: less slack gives a higher priority.

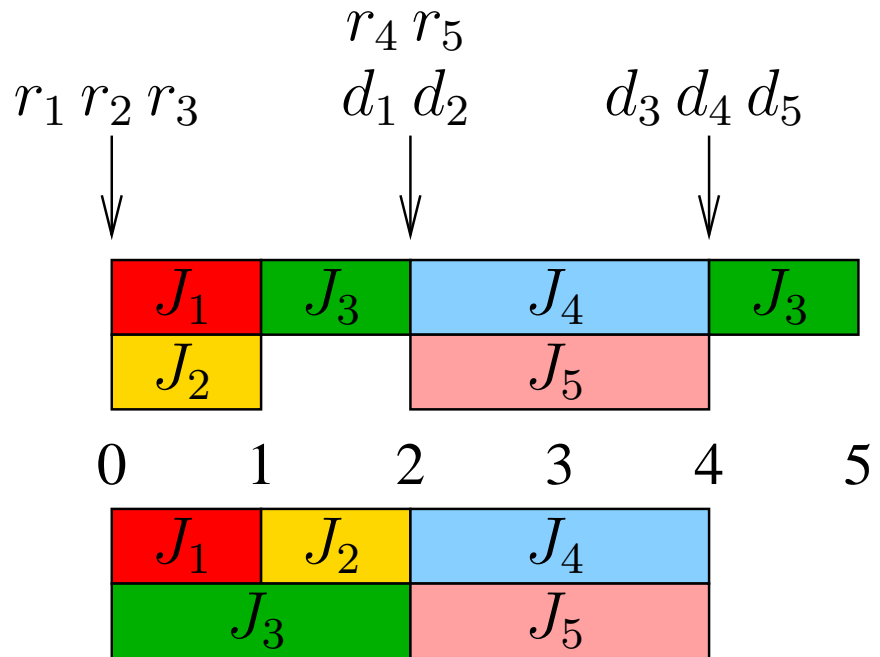
Slack of a job at time t is the idle time of the job until its d

Theorem: When jobs do not compete for resources, the LST is optimal.

Remarks for the LST scheduler:

- Priorities of jobs change dynamically.
- Continuous scheduling decisions would lead to **context s** overhead in case of two jobs with the same slack.

Non-optimality of the LST scheduler in case of two processors (with migration):

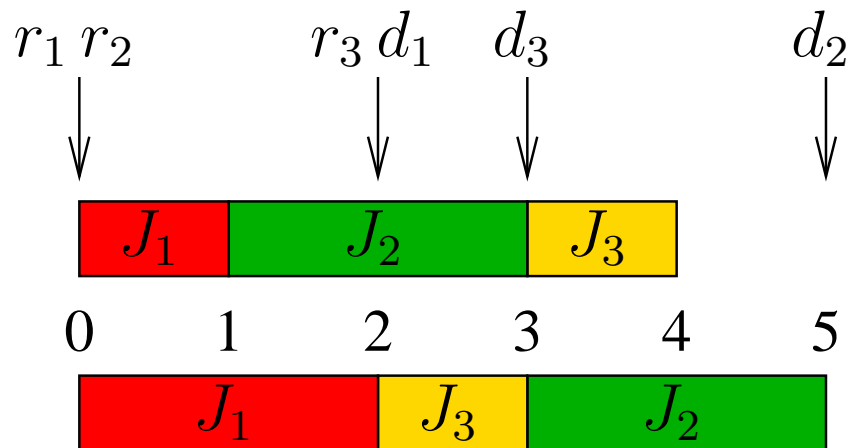


Drawback of LST: computationally expensive

Scheduling Anomaly

Let jobs be non-preemptive. Then shorter execution times can cause a violation of deadlines.

Consider the EDF (or LST) scheduler:



If jobs are preemptive, and there is no competition for resources, there is no scheduling anomaly.

Utilization

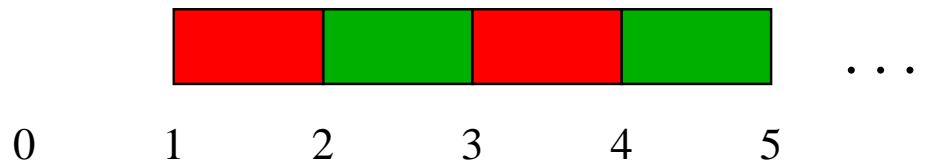
Utilization of a *periodic task* $T = (r, p, e)$ is $\frac{e}{p}$.

Utilization of a *processor* is the sum of utilizations of its periodic tasks.

Assumptions: jobs preemptive, no resource competition.

Theorem: Utilization of a processor is ≤ 1 if and only if scheduling of its periodic tasks is feasible.

Example: $T_1 = (1, 2, 1)$ and $T_2 = (1, 2, 1)$.



Assignment of Periodic Tasks to Processors

Goal: To fit periodic tasks on a minimal number of processors.

Remark: **Load balancing** is not taken into account here.

Simple approach: Assume processors P_1, \dots, P_k .

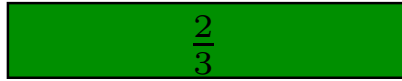
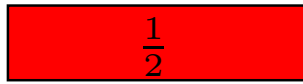
Periodic tasks T_1, \dots, T_ℓ are assigned to processors as follows.

Let T_1, \dots, T_{i-1} have been assigned. T_i is assigned to P_j if it does not fit on P_1, \dots, P_{j-1} (i.e., utilization of these processors is beyond 1) and does fit on P_j .

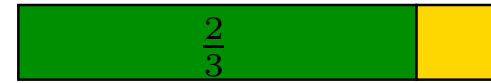
Smart approach: First sort periodic tasks by their **utilization** (T_1 has largest utilization, T_ℓ smallest utilization).

This improves worst-case and average complexity (in number of required processors).

Example: Utilizations $\frac{1}{3}$ $\frac{1}{3}$ $\frac{1}{2}$ $\frac{2}{3}$.



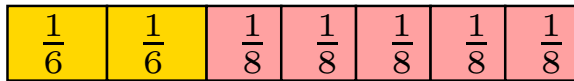
simple



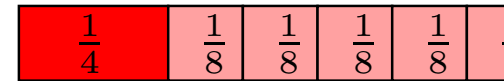
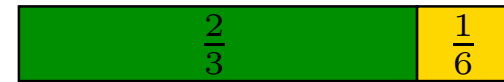
smart

However, the smart approach is not optimal. Fitting periodic a minimal number of processors is NP-complete.

Example: Utilizations $\frac{1}{8}$ $\frac{1}{8}$ $\frac{1}{8}$ $\frac{1}{8}$ $\frac{1}{8}$ $\frac{1}{8}$ $\frac{1}{6}$ $\frac{1}{6}$ $\frac{1}{4}$ $\frac{2}{3}$.



smart



optimal

Remark: Communication overhead between jobs on different processors, or between a job and a remote resource, may be account.

The problem of assigning periodic tasks to processors with m communication overhead can be reformulated into Integer Linear Programming.

Scheduling Aperiodic Jobs

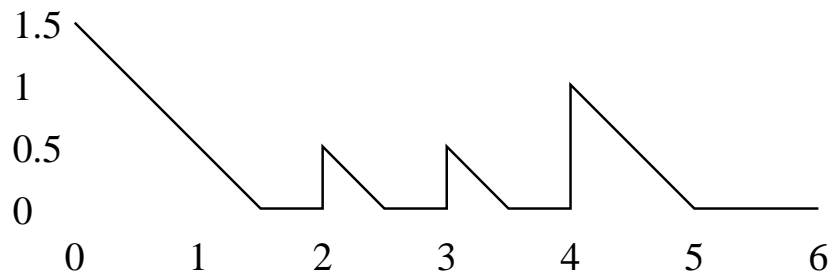
Background: aperiodic jobs are only scheduled in idle time.

Drawback: needless delay of aperiodic jobs.

Slack stealing: periodic tasks and accepted sporadic jobs may be interrupted if there is sufficient slack.

Example: $T_1 = (0, 2, \frac{1}{2})$ and $T_2 = (1, 3, \frac{1}{2})$.

Aperiodic jobs available in $[0, 6]$.



Drawback: difficult to compute in case of jitter.

Polling: gives a **period** p_s , and an **execution time** e_s for aperiodic jobs in such a period.

At the start of a new period, *the first e_s time units* can be used to execute aperiodic jobs.

Consider periodic tasks $T_k = (r_k, p_k, e_k)$ for $k = 1, \dots, n$. The server works if

$$\sum_{k=1}^n \frac{e_k}{p_k} + \frac{e_s}{p_s} \leq 1$$

Drawback: aperiodic jobs released just after a polling may be served needlessly.

We proceed to present two servers based on polling that try to avoid this drawback.

For the moment, we ignore sporadic jobs.

Deferrable Server

Allows a polling server to save its execution time within a period p_s (but not after this period!) if the aperiodic queue is empty.

In case of an EDF scheduler, the deadline of a deferrable server at the end of a period p_s can be treated as a hard deadline.

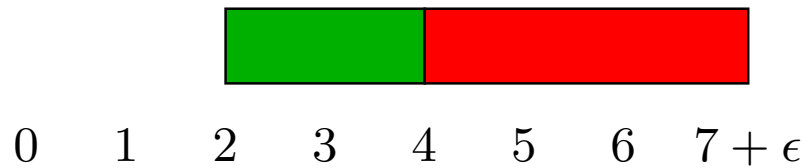
Then the deferrable server works if

$$\sum_{k=1}^n \frac{e_k}{p_k} + \frac{e_s}{p_s} \left(1 + \frac{p_s - e_s}{p_i}\right) \leq 1$$

for $i = 1, \dots, n$ (Ghazalie & Baker, 1995).

Remark: $\sum_{k=1}^n \frac{e_k}{p_k} + \frac{e_s}{p_s} \leq 1$ is not good enough.

Example: $T_1 = (2, 5, 3 + \epsilon)$ and $p_s = 3, e_s = 1$.



T_1 misses its deadline at 7

Drawback: only partial use of available bandwidth.

Total Bandwidth Server

Fix an allowed utilization rate \tilde{u}_s for the server, such that

$$\sum_{k=1}^n \frac{e_k}{p_k} + \tilde{u}_s \leq 1$$

When the aperiodic queue is non-empty, a **deadline** d is determined for the head of the queue, according to the following rules.

(Let the head of the aperiodic queue have **execution time** e .)

- When a job arrives at the *empty* aperiodic queue at time t :

$$d := \max(d, t) + \frac{e}{\tilde{u}_s}$$

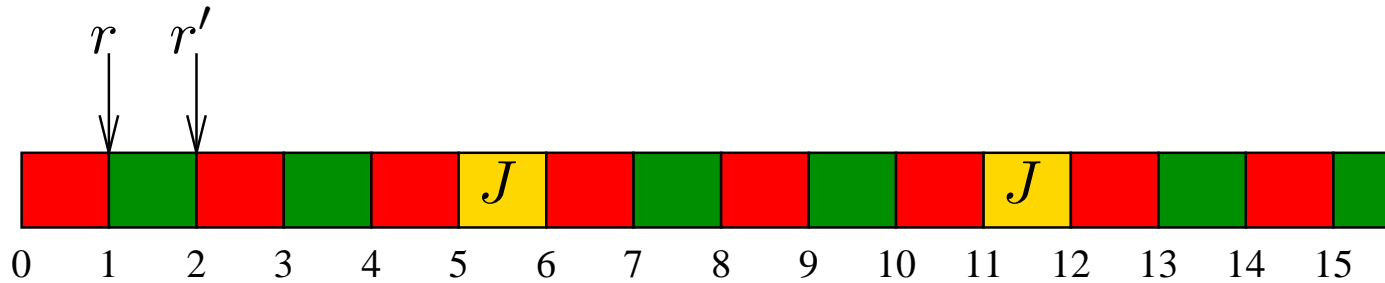
- When an aperiodic job *completes* and the *tail* of the aperiodic queue is *non-empty*:

$$d := d + \frac{e}{\tilde{u}_s}$$

Initially, $d = 0$.

Aperiodic jobs can now be treated as periodic jobs by the EDF scheduler.

Example: $T_1 = (0, 2, 1)$ and $T_2 = (0, 3, 1)$. We fix $\tilde{u}_s = \frac{1}{6}$.



J released at 1 with $e = 2$ gets (at 1) deadline $1+12=13$.

J' released at 2 with $e' = 1$ gets (at 12) deadline $13+6=19$.

Drawback: unfair in case of multiple servers; and computationally expensive compared to the deferrable server.

Generalized processor sharing divides available time over servers in a round-robin fashion.

Acceptance Test for Sporadic Jobs

A sporadic job with deadline d and execution time e is accepted if utilization (of the periodic and accepted sporadic jobs) in time interval $[t, d]$ is never more than $1 - \frac{e}{d-t}$.

If accepted, utilization in $[t, d]$ is increased with $\frac{e}{d-t}$.

Example: Periodic task $T_1 = (0, 2, 1)$.

Sporadic job with $r = 1$, $e = 2$ and $d = 6$ is accepted. Utilization in $[1, 6]$ is increased to $\frac{9}{10}$.

Sporadic job with $r = 2$, $e = 2$ and $d = 20$ is rejected.

Sporadic job with $r = 3$, $e = 1$ and $d = 13$ is accepted. Utilization in $[3, 6]$ is increased to 1, and utilization in $[6, 13]$ to $\frac{3}{5}$.

The acceptance test may reject schedulable sporadic jobs.

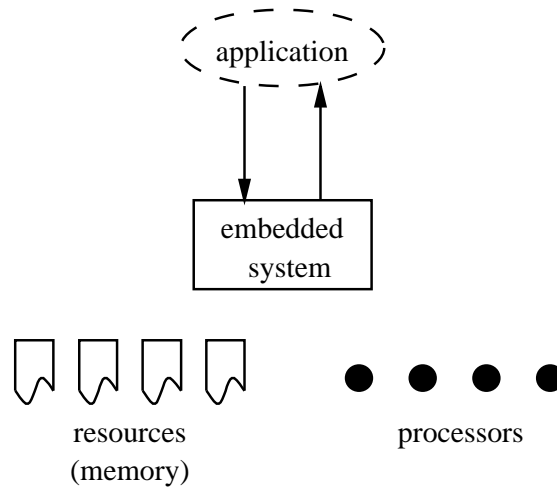
Example: Periodic task $T_1 = (0, 2, 1)$.

A sporadic job is released at time 0 with $e = 1$ and $d = 1$.

Utilization until time 1 is 1.5, but still the sporadic job could be scheduled.

Remark: The total bandwidth server can be integrated with the acceptance test for sporadic jobs (e.g. by making the allowed utilization rate \tilde{u}_s dynamic.)

Remote Access Control Protocols



Resource units can be requested by jobs during their execution. These units are allocated to jobs in a **mutually exclusive** fashion.

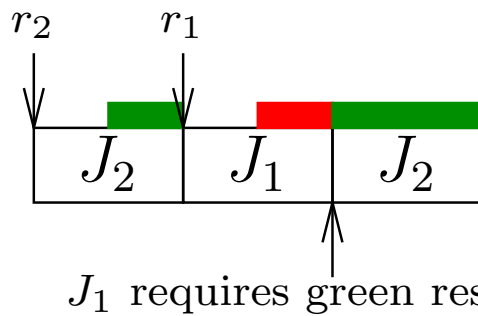
When a requested resource is refused, the job is preempted (

Remark: Resource sharing gives rise to **scheduling anomaly**.

Dangers of Resource Sharing

(1) Deadlock can occur.

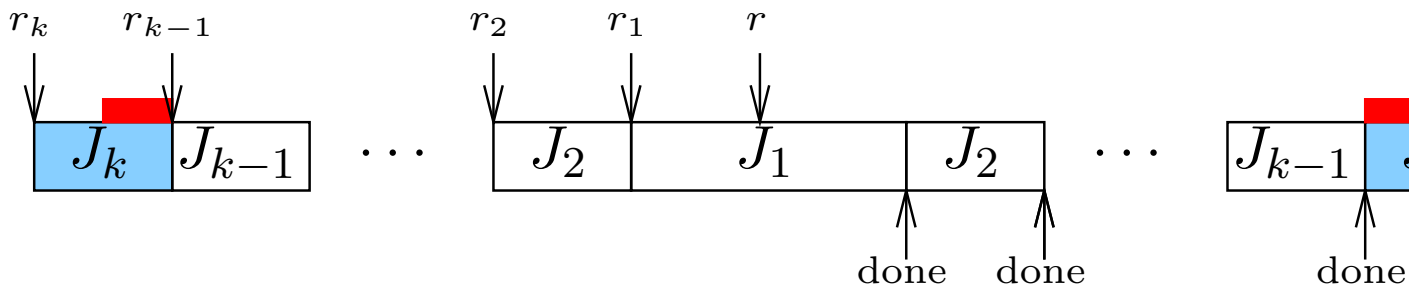
Example: $J_1 > J_2$.



J_2 requires red resource, which yields

(2) A job J can be blocked by subsequent lower-priority jobs.

Example: $J > J_1 > \dots > J_k$, and J, J_k require the red resource.

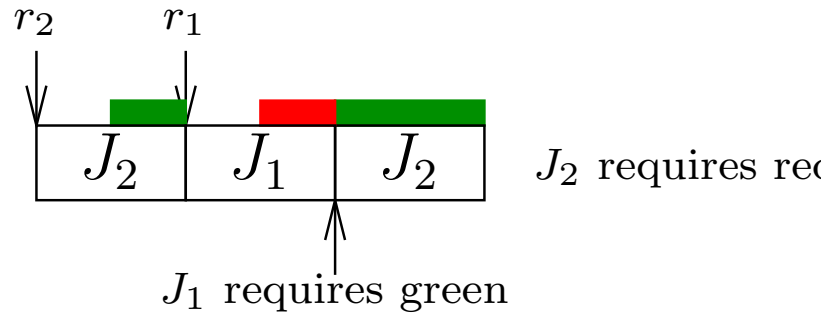


Priority Inheritance

When a job J requires a resource R and becomes blocked, the job holding R inherits the priority of J until it releases R .

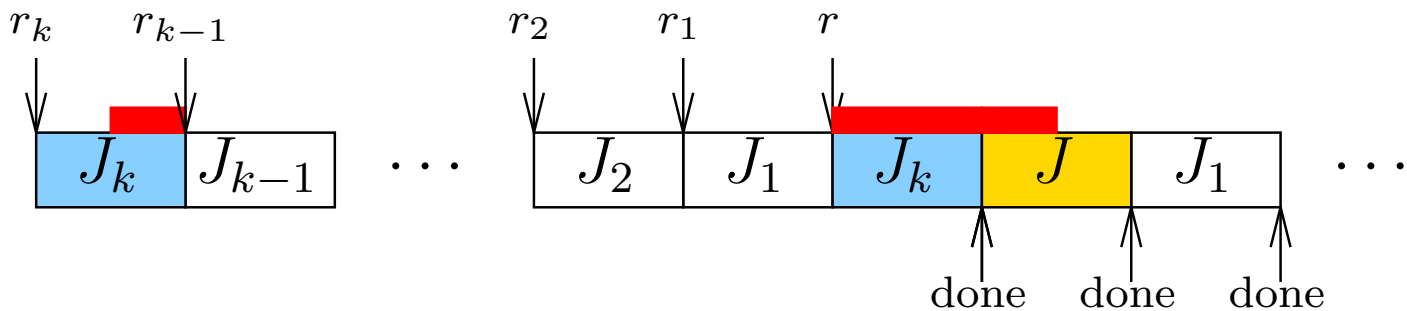
(1) Deadlock can still occur.

Example: $J_1 > J_2$.



(2) Blocking by subsequent lower-priority jobs becomes less

Example: $J > J_1 > \dots > J_k$, and J, J_k require red.



Priority Ceiling

The **priority ceiling** of a *resource* R at time t is the highest priority (known) jobs that require R at some time $\geq t$.

The **priority ceiling** of the *system* at time t is the highest priority ceiling of resources that are in use at time t .

(It has a special bottom value Ω when no resources are in use.)

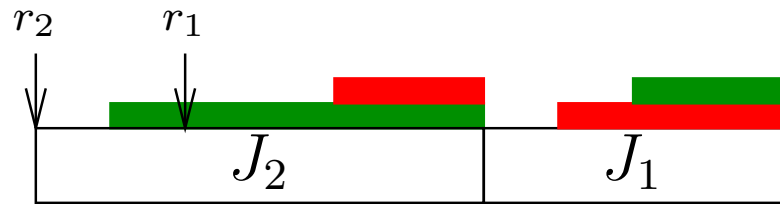
In a **priority ceiling protocol**, from the arrival of a job, this job is not preempted until its priority is higher than the priority ceiling of the system.

Assumption: The resources required by a job are known before its arrival.

Note: In the pictures to follow, r denotes the *arrival* of a job.

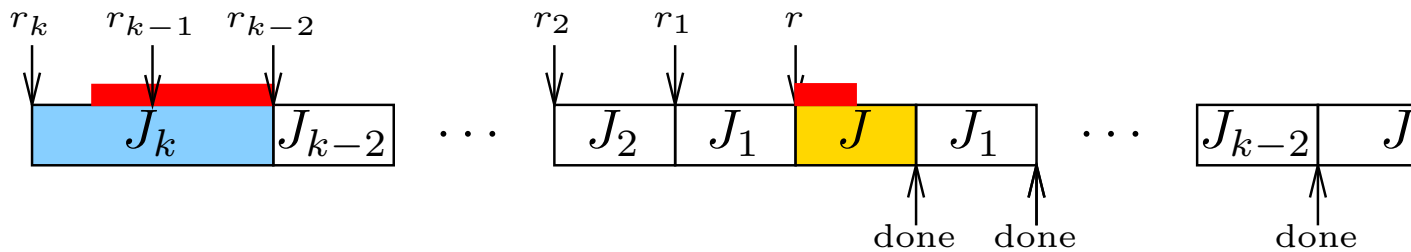
(1) No deadlocks. Because a job can only start executing w resources it will require are free.

Example: $J_1 > J_2$.



(2) Blocking by subsequent lower-priority jobs becomes less

Example: $J > J_1 > \dots > J_k$, and J, J_k require the red resou



This example assumes that the arrival of J is known at time

Question: What would happen if J were only known at its a

Preemption Ceiling

Motivation: with dynamic priorities of tasks, the overhead of computing priority ceilings is high.

The **preemption level** of jobs must be such that if $J > J'$, and J arrives after J' , then J has a higher preemption level than J' .

Idea: J' will never preempt J .

Example: The preemption level can coincide with *priorities*, *arrival times*.

Ideally, the preemption level can be defined for periodic *tasks* (or *jobs*). For instance,

EDF: preemption level can coincide with RM-priorities of tasks

FIFO: any preemption level is allowed

LIFO: in general, preemption level cannot be defined for tasks

The **preemption ceiling** of a resource R at time t is the highest preemption level of (known) jobs that require R at some time.

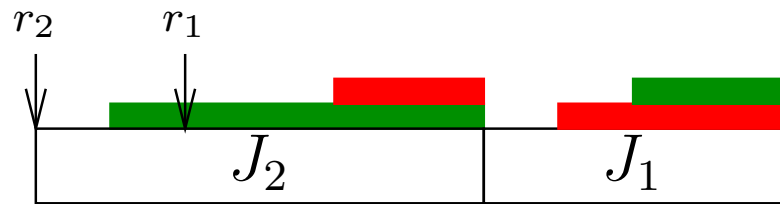
The **preemption ceiling** of the *system* at time t is the highest preemption level of resources that are in use at time t .

In a **preemption ceiling protocol**, from the arrival of a job, the job is not released until its preemption level is higher than the preemption ceiling of the system.

(Moreover, there is a priority inheritance rule.)

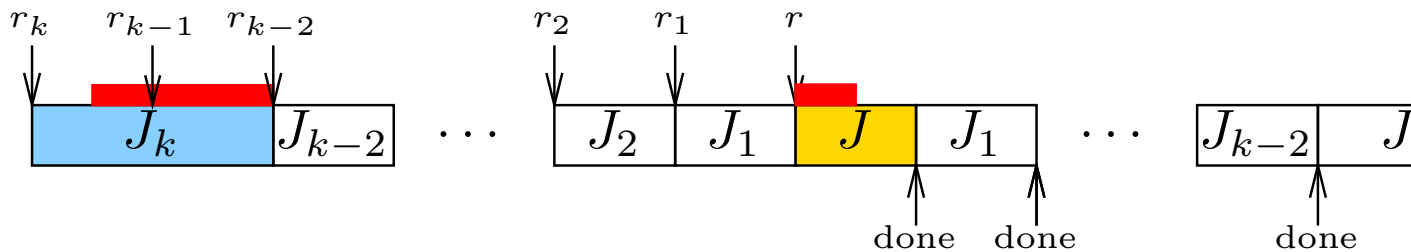
(1) No deadlocks. Because a job can only start executing w resources it will require are free.

Example: $J_1 > J_2$.



(2) Blocking by subsequent lower-priority jobs becomes less

Example: $J > J_1 > \dots > J_k$, and J, J_k require the red resou



Note that the preemption level of J is the highest of all jobs

This example assumes that the arrival of J is known at time

Multiple Resource Units

The notions of *priority* and *preemption ceiling* assumed only per resource type.

In case of multiple units of the same resource type, the definition of priority and preemption ceiling need to be adapted:

The **priority** (or **preemption**) **ceiling** of a resource R with k units at time t is the highest priority (or preemption level) of known jobs that require $> k$ units of R at some time $\geq t$.

Multiple Processors

In a **multiprocessor** setting, jobs that require a **global critical path** are given higher priority than jobs that only require local resources.

Greedy Synchronization Protocol

Consider a *multiprocessor* environment, where each processor runs some periodic tasks. There may be dependencies between jobs.

In the *greedy synchronization protocol*, jobs are released as soon as possible.

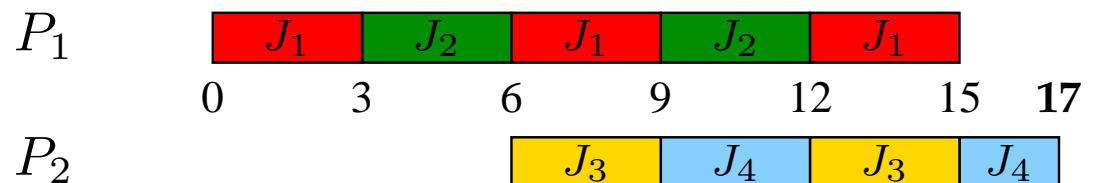
This schedule is not always optimal!

Example: Two processors P_1 and P_2 .

P_1 runs $T_1 = (0, 6, 3)$ and $T_2 = (0, 9, 3)$, with $T_1 > T_2$.

P_2 runs $T_3 = (0, 9, 3)$ and $T_4 = (6, 10, 5)$, with $T_3 > T_4$.

A job of T_3 can only be released if a job of T_2 completed.



Example: Consider the same example, but now with the EDF scheduler.

P_1 runs $T_1 = (0, 6, 3)$ and $T_2 = (0, 9, 3)$.

P_2 runs $T_3 = (0, 9, 3)$ and $T_4 = (6, 10, 5)$.

A job of T_3 can only be released if a job of T_2 completed.



All deadlines are met.

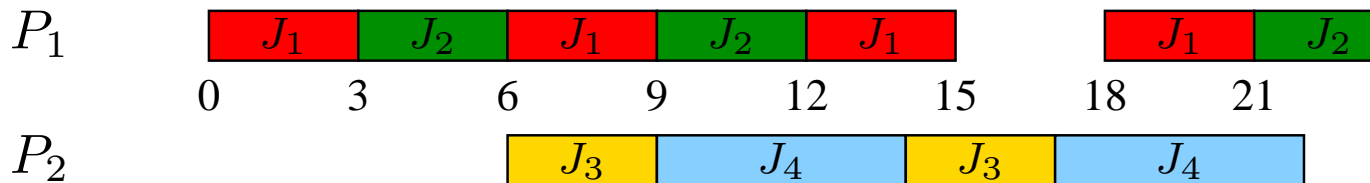
But the response time of a pair of jobs from (T_2, T_3) can be large, and the completion time can become unpredictable.

Release-Guard Protocol

Suppose a job of periodic task $T_2 = (r, p, e)$ can only be released after a job of periodic task T_1 is completed.

In the **release-guard protocol**, the k -th job of T_2 is released at time t with:

- $t \geq r + (k-1)p$;
- the k -th job of T_1 is completed before t ; and
- either T_2 's processor is idle at t , or t is at least p time units after the release of the $(k-1)$ -th job of T_2 .



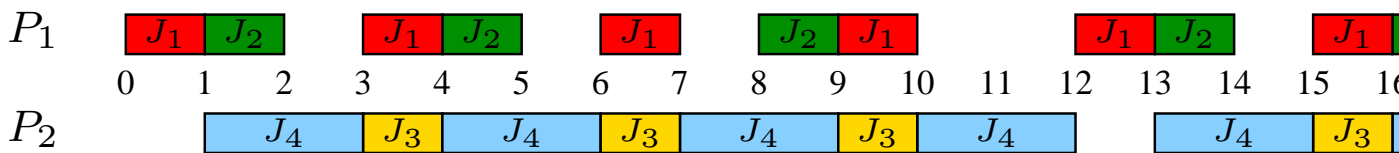
Example: Two processors P_1 and P_2 .

P_1 runs $T_1 = (0, 3, 1)$ and $T_2 = (0, 4, 1)$.

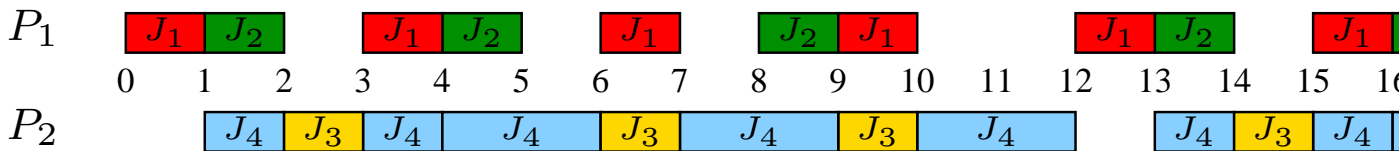
P_2 runs $T_3 = (0, 4, 1)$ and $T_4 = (1, 3, 2)$.

A job of T_3 can only be released if a job of T_2 completed.

First consider the greedy synchronization protocol, with EDF



Now consider the release-guard protocol, with $T_1 > T_2$ and $T_3 > T_4$



Summary:

- clock-driven vs. priority-driven scheduling
- fixed priority of RM vs. dynamic priority of EDF/LST
- optimality of EDF/LST vs. non-optimality of RM
- divide periodic tasks over processors
- scheduling of aperiodic jobs
(*slack stealing / polling / total bandwidth*)
- acceptance test for sporadic jobs (*based on utilization*)
- resource sharing
(*priority inheritance / priority ceiling / preemption c*)
- dependencies between jobs
(*greedy synchronization / release-guard*)