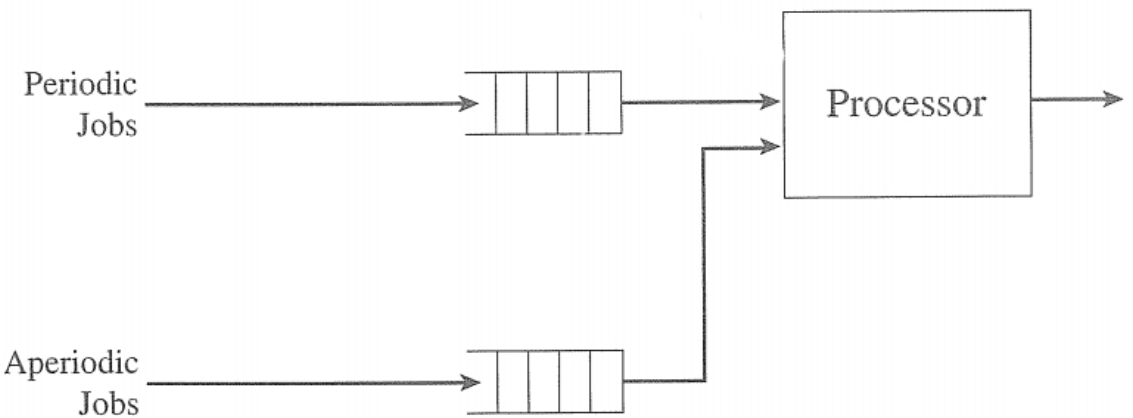
Real-Time Scheduling

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

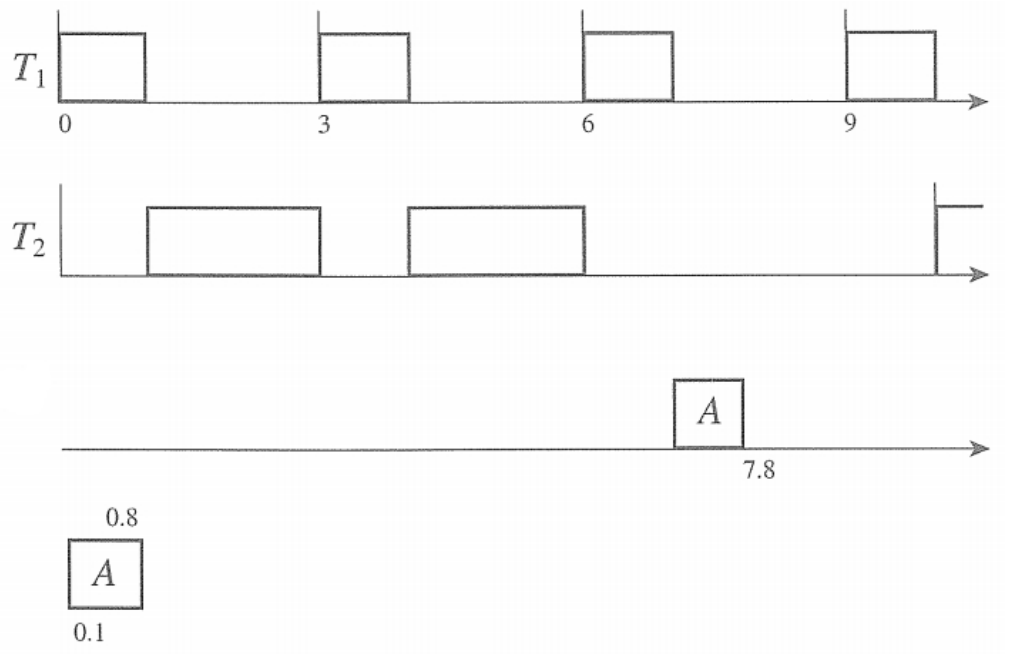
  $\Rightarrow$ This includes all periodic jobs
- A scheduling algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists, and if a cost function is given, minimizes the cost

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

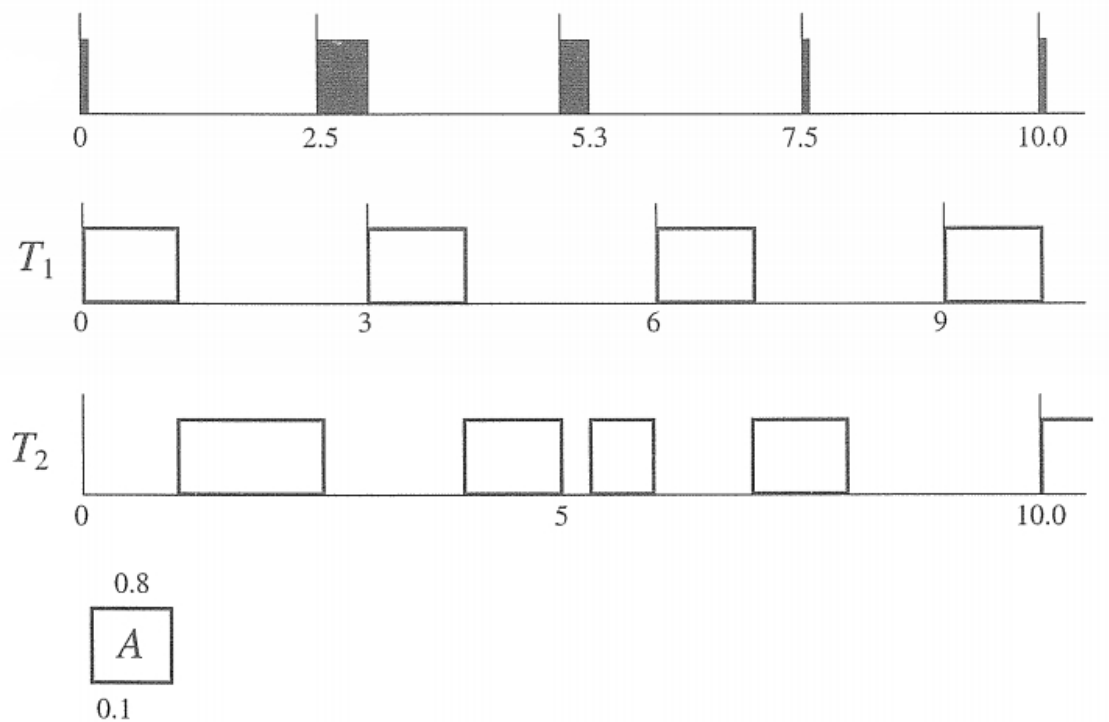


Polled Execution of Aperiodic Jobs

- 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 Severs – 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)

Periodic Severs

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

Periodic Severs

Deferrable sever

- *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 $k = 0, 1, 2, \ldots$

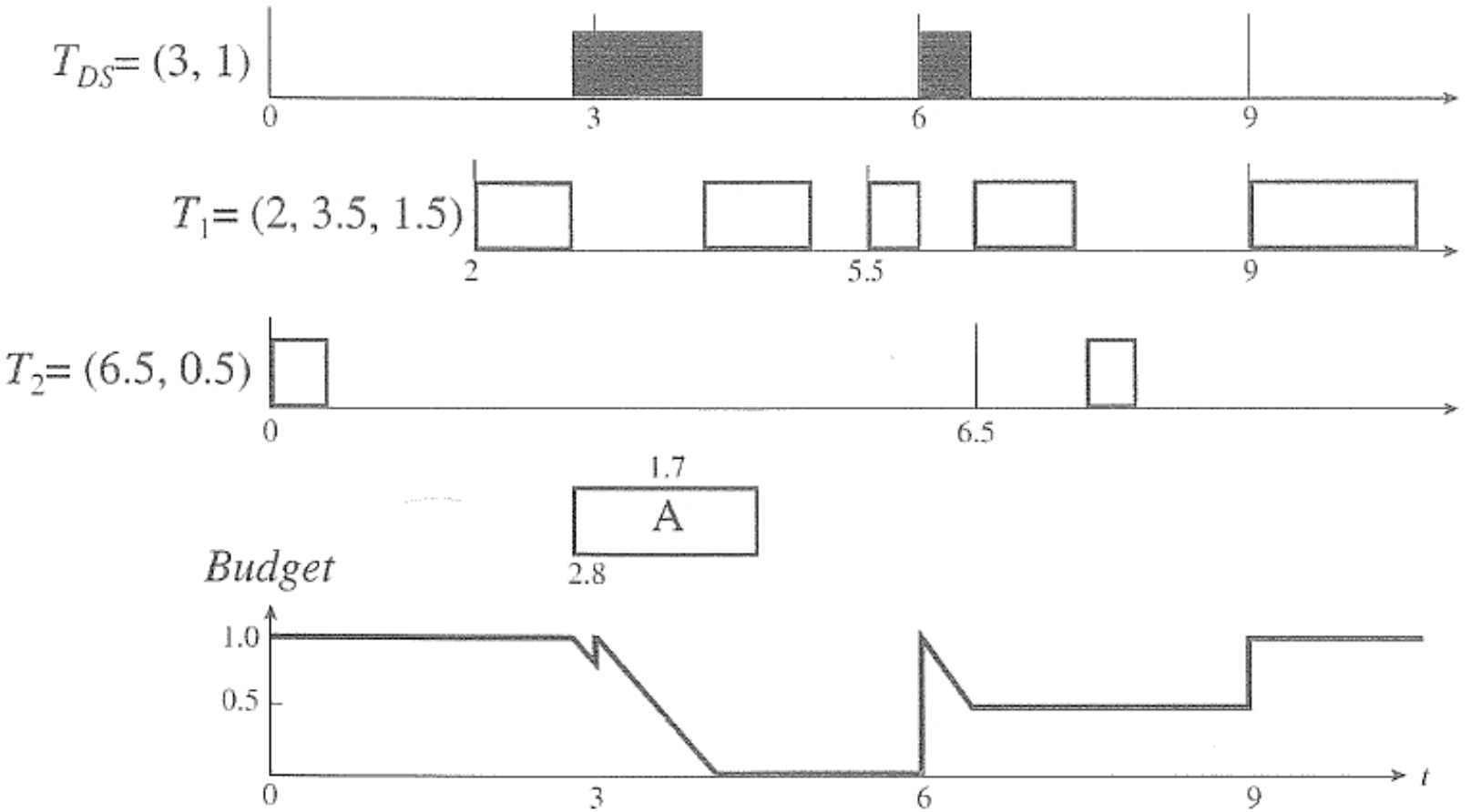
    (Note that the server is not able tu cumulate the budget over periods)

We consider both

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

Deferrable Server – RM

Here the tasks are scheduled using RM.

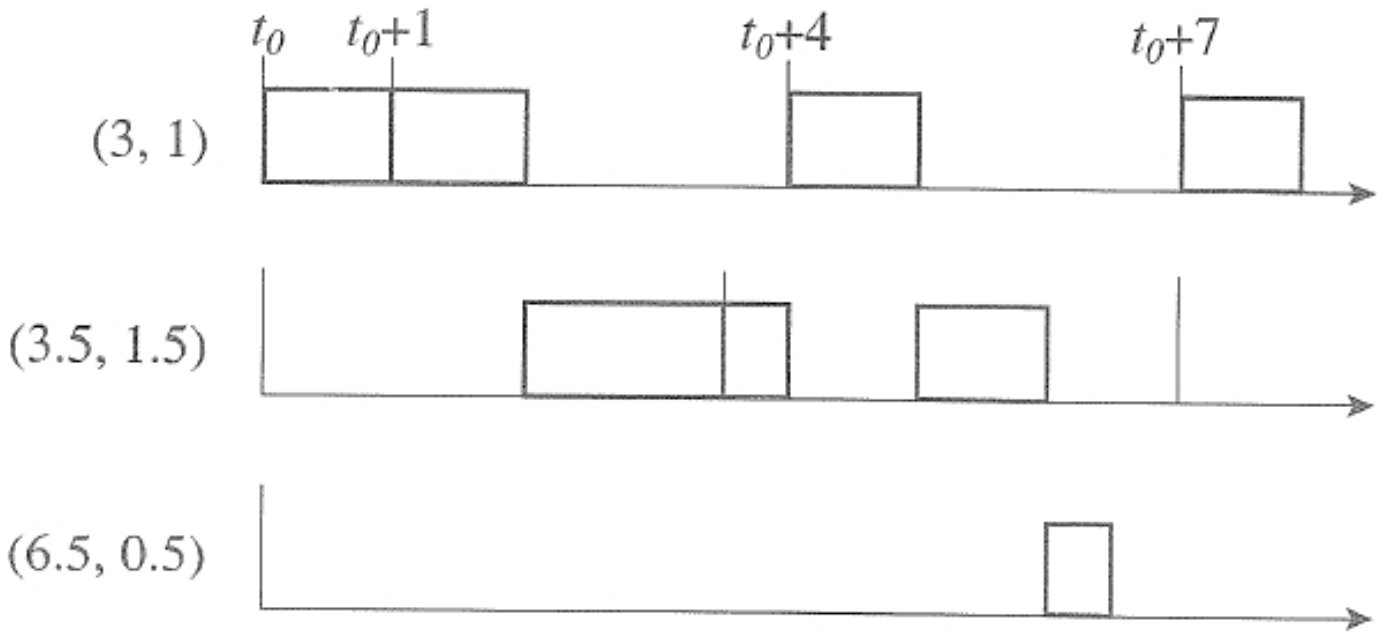


Is it possible to increase the budget of the server to 1.5 ?

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

Deferrable Server – Critical Instant

Lemma 23

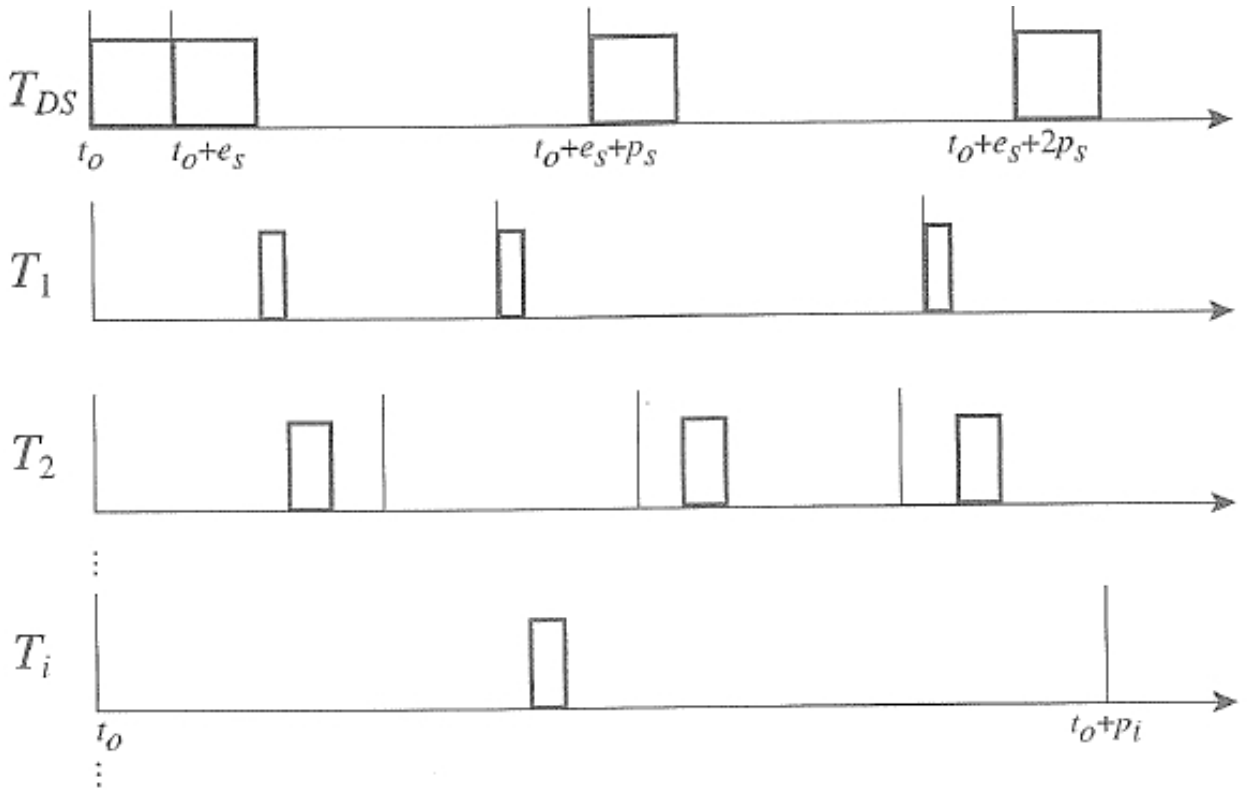
Assume a fixed-priority scheduling algorithm. Assume that $D_i \leq p_i$ and that the deferrable server (p_S, e_S) has the highest priority among all tasks. Then a critical instant of every periodic task T_i occurs at a time t_0 when all of the following are true:

- *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 $T_{DS} \sqsupset T_1 \sqsupset T_2 \sqsupset \cdots \sqsupset T_n$

(i.e. T_1 has the highest pririty 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

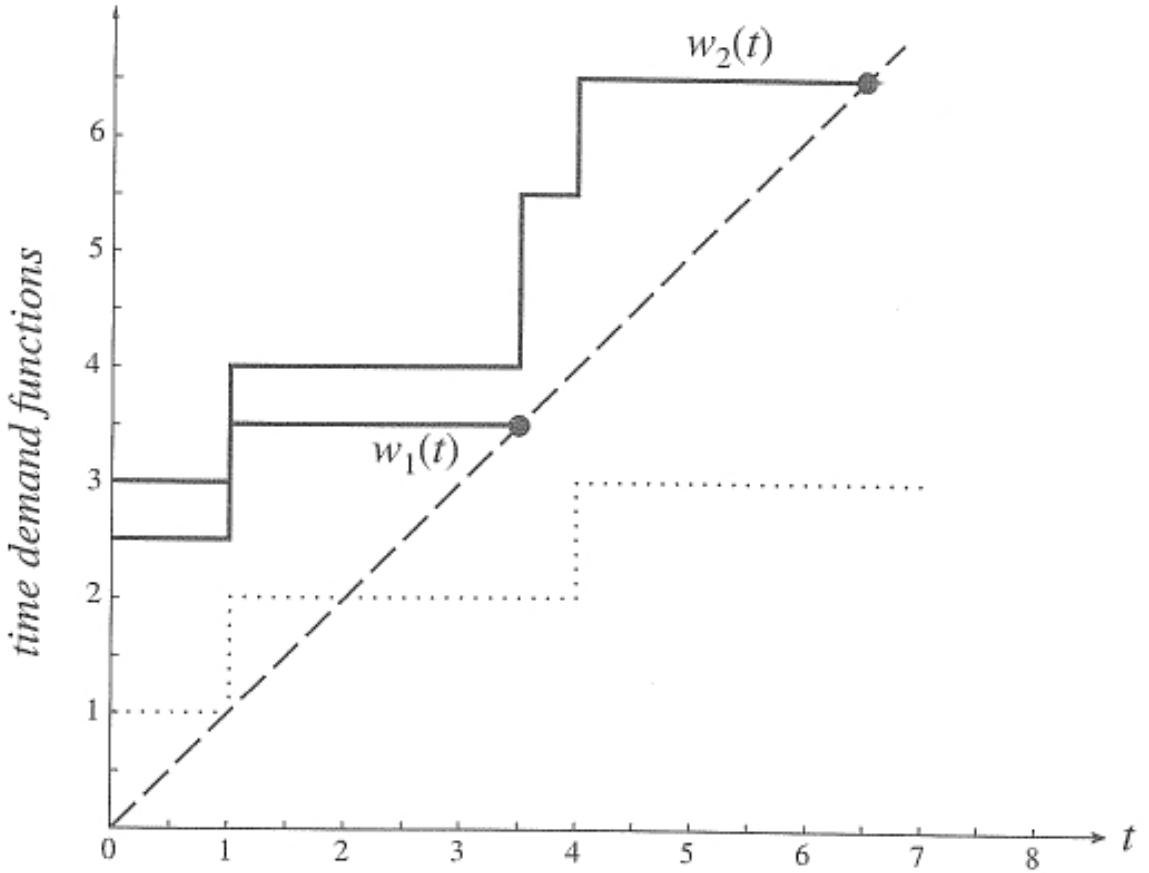
$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k + e_S + \left\lceil \frac{t - e_S}{p_S} \right\rceil e_S \qquad \text{for } 0 < t \leq p_i$$

- 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 $k = 1, 2, \ldots, i$ and $j = 1, 2, \ldots, \lfloor D_i/p_k \rfloor$, or
  - to $e_S, e_S + p_S, e_S + 2p_S, \ldots, e_S + \lfloor (D_i - e_i)/p_S \rfloor p_S$

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

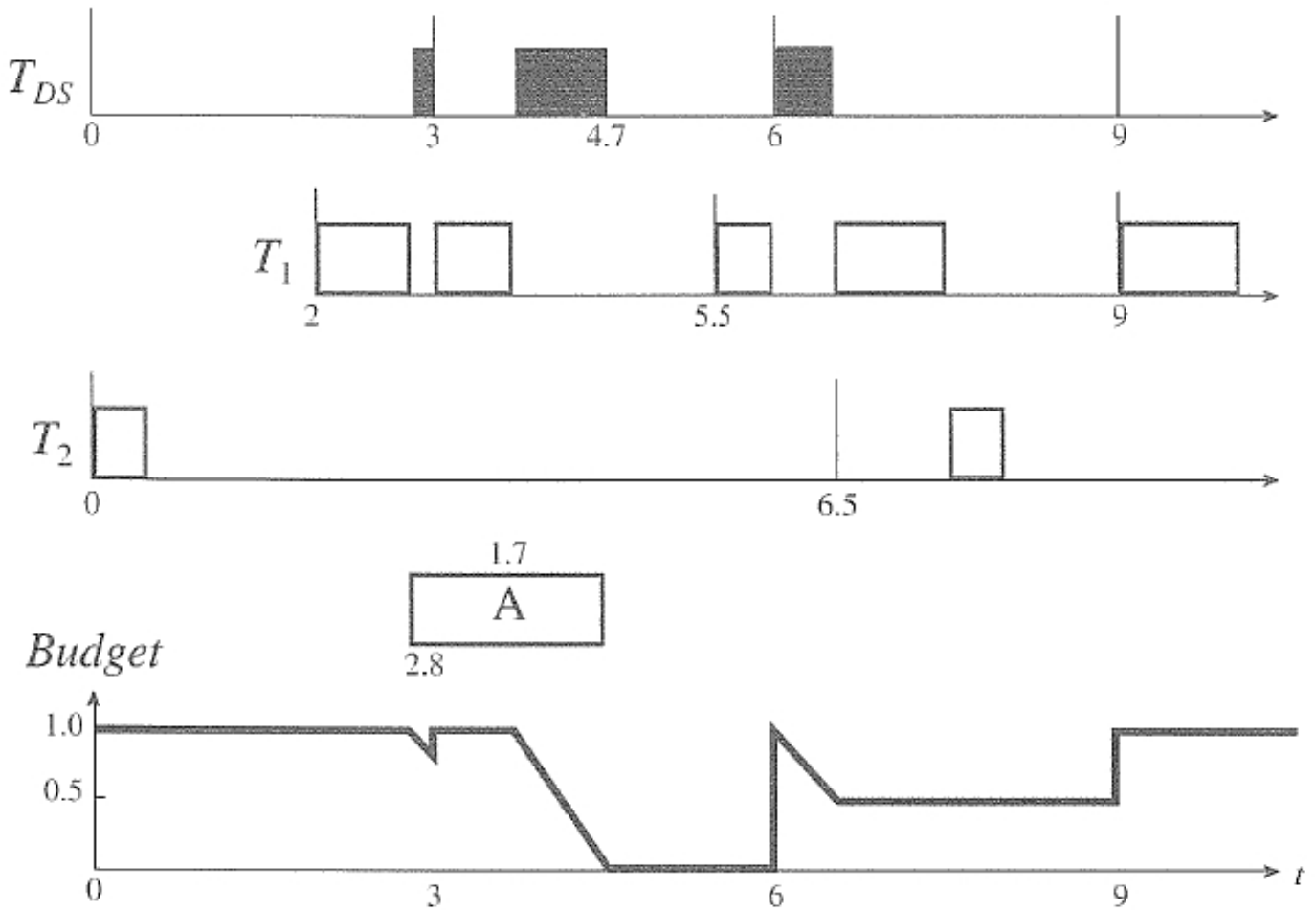
- No maximum schedulable utilization is known in general
- A special case:
  - A set T of n independent, preemptable periodic tasks whose periods satisfy $p_S < p_1 < \cdots < p_n < 2p_S$ and $p_n > p_S + e_S$ and whose relative deadlines are equal to their respective periods, can be scheduled according 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.
$T_{DS} = (3, 1)$, $T_1 = (2, 3.5, 1.5)$, $T_2 = (6.5, 0.5)$



Deferrable Server – EDF – Schedulability

Theorem 24

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 $T_1 \sqsupset T_2 \sqsupset \cdots \sqsupset T_n$ 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

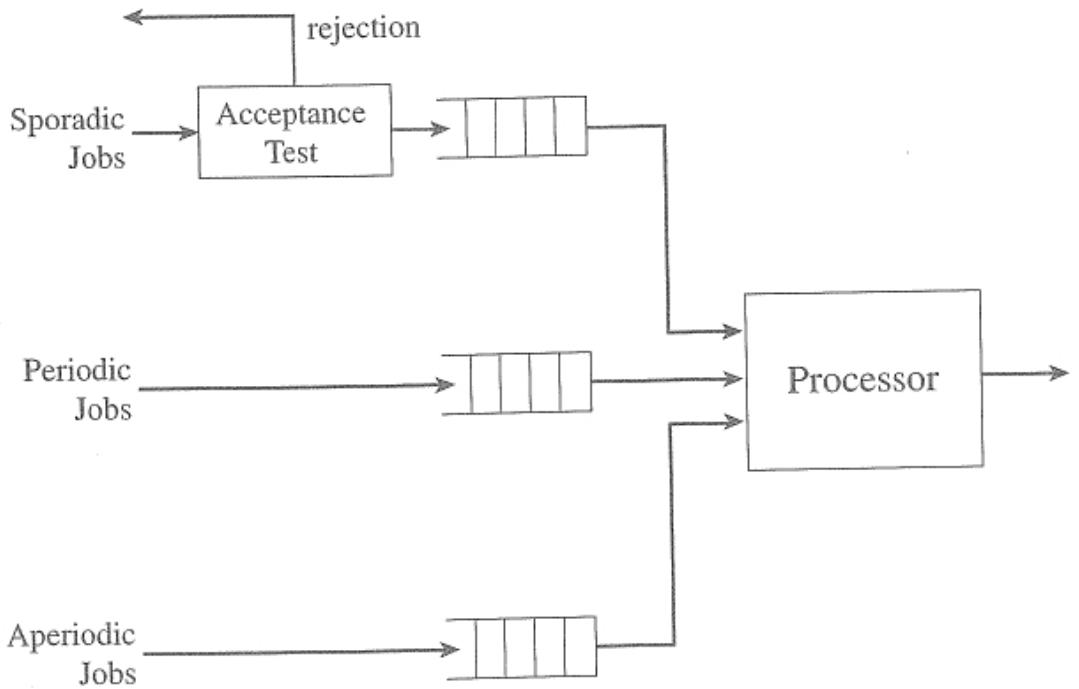
Current Assumptions

- Single processor
- Fixed number, n, of *independent periodic* tasks, $T_1, \ldots, T_n$ 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 tasks 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 25

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 $J_1, \ldots, J_k$ 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 26

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

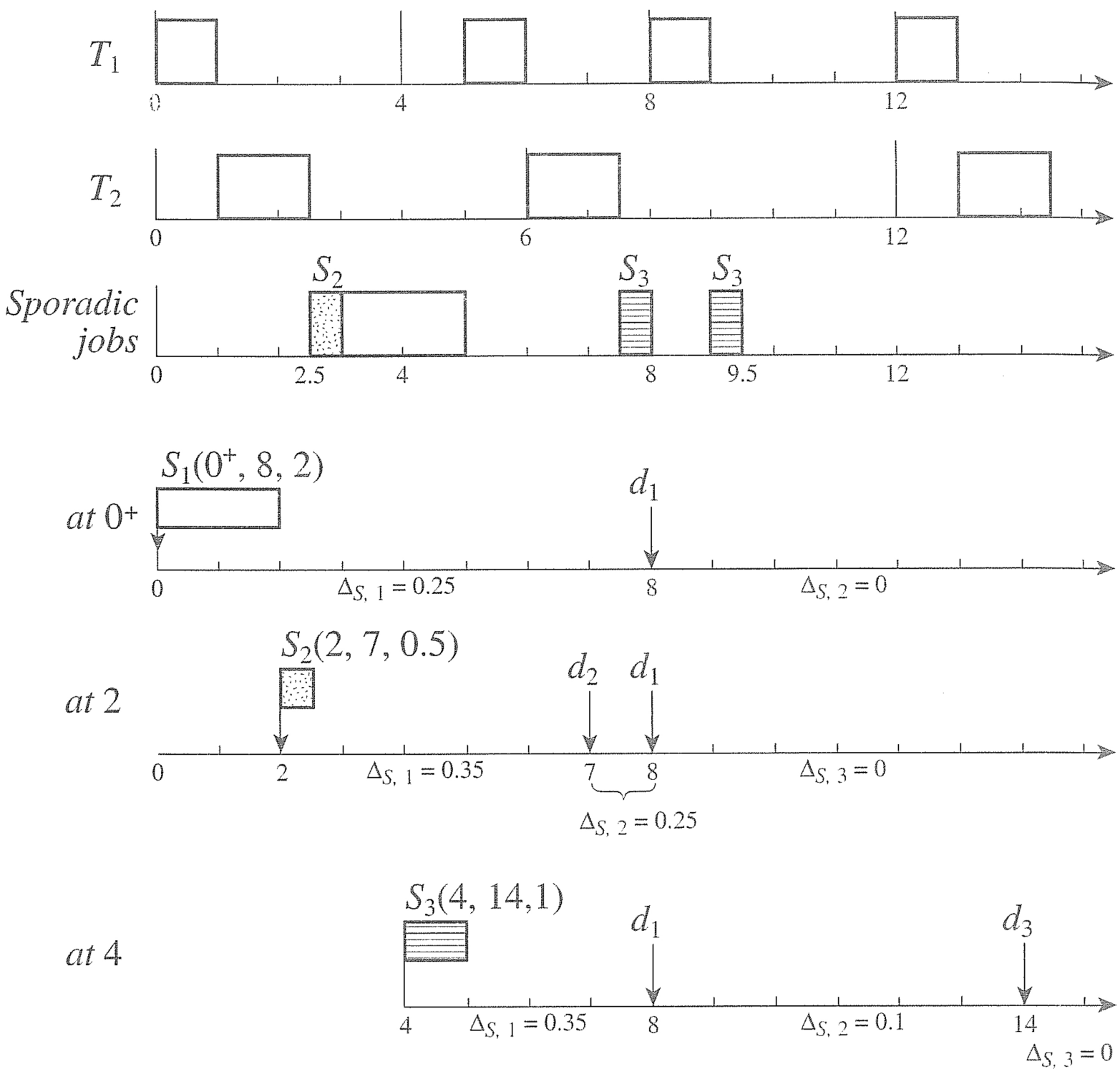
Total density at time 1.5 is 1.5

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 deadline
  - The deadlines partition the time from t to ∞ into $n + 1$ discrete intervals $I_1, I_2, \ldots, I_{n+1}$
    - 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 $k = 1, 2, \ldots, \ell$
  - i.e. accept if the new sporadic job can be added, without increasing density of any intervals past 1


T_1
0
4
8
12
T_2
0
6
12
Sporadic jobs
S_2
S_3
S_3
0
2.5
4
8
9.5
12
at 0+
$S_1(0^+, 8, 2)$
d_1
0
$\Delta_{S,1} = 0.25$
8
$\Delta_{S,2} = 0$
at 2
$S_2(2, 7, 0.5)$
d_2
d_1
0
2
$\Delta_{S,1} = 0.35$
7
8
$\Delta_{S,3} = 0$
$\Delta_{S,2} = 0.25$
at 4
$S_3(4, 14, 1)$
d_1
d_3
4
$\Delta_{S,1} = 0.35$
8
$\Delta_{S,2} = 0.1$
14
$\Delta_{S,3} = 0$

Admission Control for Sporadic Jobs with EDF

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 $\sigma_{S,k}(t) \geq e_{S,i}$