Scheduling aperiodic and sporadic jobs in priority-driven systems

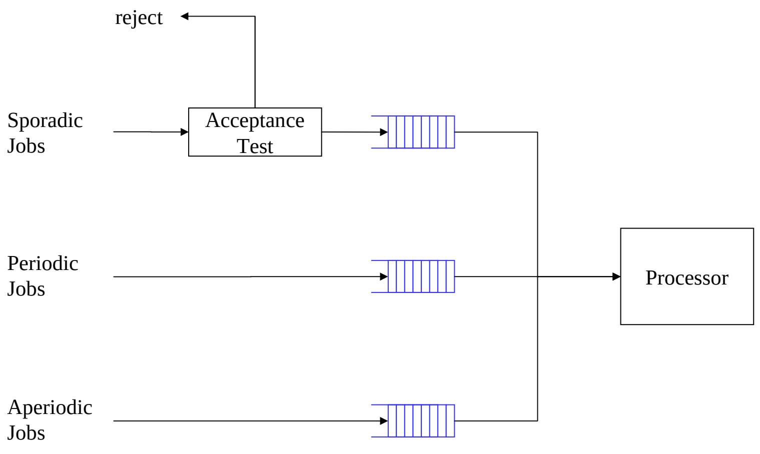
OUTLINE

Ref: [Liu]
- Ch. 7 (pg. 190 – 276)

Aperiodic and sporadic jobs

- When variations in inter-release times and execution times of a task T_s are small, we can treat the task as a periodic task $T_s = (p_s, e_s)$, and schedule it accordingly.
- What to do when variations are not small? sporadic and aperiodic jobs!
  - sporadic jobs: have a hard deadline
  - aperiodic jobs: have a soft deadline
    - can arrive at any time,
    - execution times vary widely,
    - deadlines are unknown a priori.
- Problem: given a set of *n* periodic tasks $T_1, \dots, T_i = (p_i, e_i), \dots, T_n$ and a priority-driven scheduler, assuming that all periodic tasks meet their deadlines,
  - determine if and when to execute aperiodic and sporadic jobs:
    - sporadic jobs: acceptance test + scheduling of accepted job
    - aperiodic jobs: schedule them to complete ASAP.

Acceptance test for sporadic jobs



Priority Queues for Periodic/Sporadic/Aperiodic Jobs

For the moment, we ignore sporadic jobs and consider the scheduling of aperiodic jobs in the midst of periodic tasks

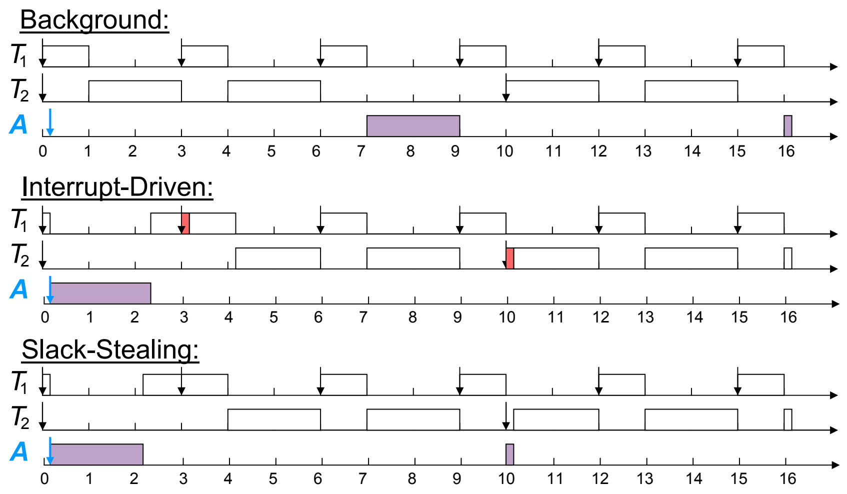
Aperiodic job scheduling

4 COMMON APPROACHES:

- Background:
  - Aperiodic job queue has always lowest priority among all queues.
  - Periodic tasks (and accepted sporadic jobs) always meet deadlines.
  - Simple to implement.
  - Problem: execution of aperiodic jobs may be unduly delayed.
- Interrupt-Driven:
  - Response time as short as possible.
  - Periodic tasks may miss some deadlines.
- Slack-Stealing:
  - Postpone execution of periodic tasks only when it is safe to do so:
    - well-suited for clock-driven environments.
    - what about priority-driven environments? (quite complicated).
- Polling Server:
  - a periodic task that is scheduled with the other periodic tasks,
  - executes the first job in the aperiodic job queue (when not empty).

Background / Interrupt-Driven / Slack-Stealing: examples

$T_1 = (3, 1)$; $T_2 = (10, 4)$: RM scheduled; **+** aperiodic job ***A***: ***$r = 0.1$, $e = 2.1$***

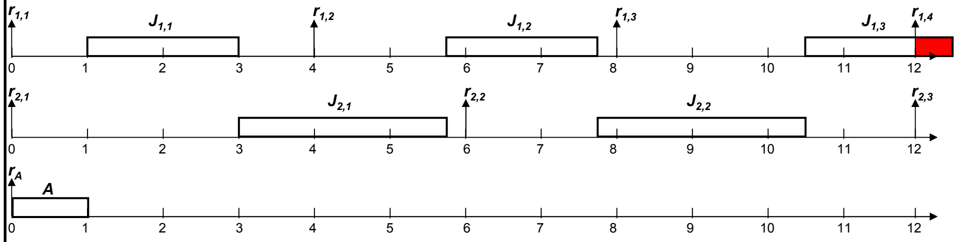


Slack stealing in EDF systems

- Slack stealer (executes aperiodic jobs):
  - ready for execution when the aperiodic job queue is non empty;
  - suspended when the aperiodic job queue is empty.
- The scheduler monitors the available slack $\sigma(t)$ in the system and:
  - when $\sigma(t) > 0$, gives the slack stealer the highest priority;
  - when $\sigma(t) = 0$, gives the slack stealer the lowest priority.
- When executing, the slack stealer executes always the job at the head of the aperiodic job queue.
- Problem: how to compute the slack $\sigma(t)$ available in the system at time t ?
  - static computation (initial slack computed off-line and updated by the scheduler at run time):
    - reduced run-time overhead,
    - reliable only when jitter in inter-release times is low;
  - dynamic computation:
    - only possibility when jitter in inter-release times is high,
    - high run-time overhead.
- We now present a method for static computation of slack in EDF systems.

Slack computation

- Let's examine an example to get some insight into the problem:
  T_1 = (4,2), T_2 = (6,2.75) + aperiodic job *A* released at *t* = 0 with e_A = 1.
  - at *t* = 0 the slacks of the first jobs $J_{1,1}$ and $J_{2,1}$ are:
    $\sigma_{1,1}(0) = D_1 - e_1 = 4 - 2 = 2$
    $\sigma_{2,1}(0) = D_2 - e_2 - e_1 = 6 - 2.75 - 2 = 1.25$
  - if we were to conclude that the slack of the system at *t* = 0 is $\sigma(0)$ = 1.25 and let *A* execute for 1 time unit, …



  - then $J_{1,3}$ would miss its deadline: the execution of A would not affect $J_{1,1}$ and $J_{2,1}$ (they have enough slack), but a later job;
- the slack of the system at *t* = 0 is *not* 1.25, but 0.5: how to compute it?

Slack computation: initial slack

- The slack of the system at the beginning of an hyperperiod is the minimum slack of all jobs released in the hyperperiod;
- if the number of tasks is *n* and the total number of jobs of all tasks in an hyperperiod is *N*, the brute force computation of the system slack is of complexity *O(N)*;
- let's consider a system of *n* tasks $T_1, T_2, \ldots, T_n$, assuming that all the tasks are in phase, and let's index the *N* jobs $J_1, J_2, \ldots, J_N$ released by all tasks in the first hyperperiod in a non-increasing priority order ($d_i \le d_k$ if $i < k$);
- the slack of a job J_i at the beginning of an hyperperiod (*t* = 0) is:

  $$\sigma_i(0) = d_i - \Sigma_{k=1 \text{ to } i}\, e_k$$
- these *N* values $\sigma_i(0)$ of the initial slack of all jobs may be precomputed and inserted in an "initial slack table", together with the initial slack of the system:

  $$\sigma(0) = min\{\sigma_i(0)\}.$$

Slack computation at time t_c

- When the system is in execution within an hyperperiod, the slack of each job decreases from its initial value, when any of these events occur:
  - the processor is idling,
  - the slack stealer is executing,
  - lower priority jobs are executing;
- The slack of J_i at time t_c may be computed as:

  $\sigma_i(t_c) = \sigma_i(0) - I(t_c) - ST(t_c) - \Sigma_{d_k > d_i}\, \xi_k(t_c)$, where:
  - $I(t_c)$ is the processor idle time since $t = 0$ (beginning of the hyperperiod),
  - $ST(t_c)$ is the time the slack stealer has been executing since $t = 0$,
  - $\xi_k(t_c)$ is the time the lower priority job J_k has been executing since $t = 0$.
- The slack of the system at time t_c is the minimum slack at time t_c of all jobs released in the hyperperiod:

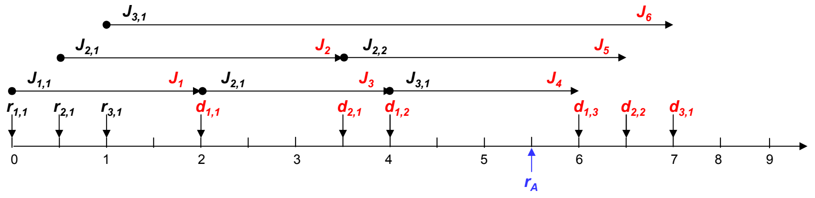
  $\sigma(t_c) = min\{\sigma_i(t_c)\}$.
- The N values $\sigma_i(t_c)$ of the slacks of all jobs J_i must be computed at run time in order to determine $\sigma(t_c)$, the available slack in the system at time t_c.

Slack computation: example

- Example: $T_1 = (2,0.5)$, $T_2 = (0.5,3,1)$, $T_3 = (1,6,1.2)$
  + aperiodic job A released at $t = 5.5$ with $e_A = 1$
- the jobs released in the first hyperperiod ($H = 6$) are: 3 of T_1, 2 of T_2, 1 of T_3; their deadlines are:

  $d_{1,1} = 2$, $d_{1,2} = 4$, $d_{1,3} = 6$,
  $d_{2,1} = 3.5$, $d_{2,2} = 6.5$,
  $d_{3,1} = 7$
- Let's index the jobs according to their deadlines:
- The following figure indicates the feasible intervals of the six jobs and (in red) their new indexes according to the order of their deadlines:

$J_1 = J_{1,1}$ $(d_1 = 2)$
$J_2 = J_{2,1}$ $(d_2 = 3.5)$
$J_3 = J_{1,2}$ $(d_3 = 4)$
$J_4 = J_{1,3}$ $(d_4 = 6)$
$J_5 = J_{2,2}$ $(d_5 = 6.5)$
$J_6 = J_{3,1}$ $(d_6 = 7)$



Slack computation: example (2)

- The initial slacks (**$\sigma_i\ (0) = d_i - \Sigma_{k=1\ to\ i}\ e_k$**) of the 6 jobs are:

  $\sigma_1(0) = d_1 - e_1 = 2 - 0.5 = 1.5$

  $\sigma_2(0) = d_2 - e_1 - e_2 = 3.5 - 0.5 - 1 = 2$

  $\sigma_3(0) = d_3 - e_1 - e_2 - e_3 = 4 - 0.5 - 1 - 0.5 = 2$

  $\sigma_4(0) = d_4 - e_1 - e_2 - e_3 - e_4 = 6 - 0.5 - 1 - 0.5 - 0.5 = 3.5$

  $\sigma_5(0) = d_5 - e_1 - e_2 - e_3 - e_4 - e_5 = 6.5 - 0.5 - 1 - 0.5 - 0.5 - 1 = 3$

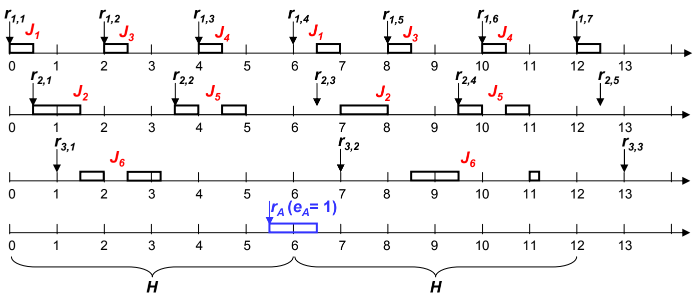
  $\sigma_6(0) = d_6 - e_1 - e_2 - e_3 - e_4 - e_5 - e_6 = 7 - 0.5 - 1 - 0.5 - 0.5 - 1 - 1.2 = 2.3$

- the initial slack of the system is:

  $\sigma(0) = min\{\sigma_i(0)\} = 1.5$

Slack computation: example (3)

- at $t_c = 5.5$, when A is released:,
  - $ST(5.5) = 0$ (the slack stealer has never executed), $I(5.5)=0.8$,
  - $\Sigma_{d_k > d_i}\ \xi_k(t_c) = 0$ (since all 6 jobs have completed their execution),
  - the slack of the system is $\sigma(5.5) = \sigma(0) - ST(5.5) - I(5.5) = 1.5 - 0.8 = 0.7$
  - since $\sigma(5.5) > 0$, the slack stealer is scheduled to execute A;
- at $t_c = 6$ a new hyperperiod starts:
  - the values of I, ST and ξ_i (for $i = 1$ to N) are all 0;
  - the initial system slack is again $\sigma(0) = 1.5 > 0$;
  - since the aperiodic job queue is not empty, the slack stealer goes on:



Quicker slack computation

- We have seen that in a system of n tasks the brute force computation of the system slack is of complexity $O(N)$, where N is the total number of jobs of all tasks in an hyperperiod: this may require an unacceptable high run-time overhead;
- we now describe an algorithm that performs the computation using a pre-computed slack table of size $O(N^2)$ with complexity $O(n)$ (usually $n << N$: the number of tasks is much smaller than the number of jobs in an hyperperiod).
- again we index the N jobs $J_1, J_2, \ldots, J_N$ released by all tasks in the first hyperperiod in a non-increasing priority order ($d_i \leq d_k$ if $i < k$);
- the slack of a job J_i at the beginning of an hyperperiod ($t = 0$) is:

  $\boldsymbol{\sigma_i (0) = d_i - \Sigma_{k=1 \text{ to } i}\, e_k}$
- let $\boldsymbol{\omega(j;k) = \min_{j \leq i \leq k} \{\sigma_i (0)\}}$ (with $j \leq k$) be the minimum initial slack of all jobs whose deadlines are in the range $[d_i,\ d_k]$;
- then, $\sigma_i (0) = \omega(i;i)$ and the initial slack of the system is $\sigma(0) = \omega(1;N)$.
- The pre-computed table stores the N^2 values of $\omega(j;k)$ for$1 \leq j \leq k \leq$ N.

Quicker slack computation (2)

- The initial slacks ($\boldsymbol{\sigma_i (0) = d_i - \Sigma_{k=1 \text{ to } i}\, e_k}$) of the 6 jobs are:

  $\sigma_1(0) = d_1 - e_1 = 2 - 0.5 = 1.5$

  $\sigma_2(0) = d_2 - e_1 - e_2 = 3.5 - 0.5 - 1 = 2$

  $\sigma_3(0) = d_3 - e_1 - e_2 - e_3 = 4 - 0.5 - 1 - 0.5 = 2$

  $\sigma_4(0) = d_4 - e_1 - e_2 - e_3 - e_4 = 6 - 0.5 - 1 - 0.5 - 0.5 = 3.5$

  $\sigma_5(0) = d_5 - e_1 - e_2 - e_3 - e_4 - e_5 = 6.5 - 0.5 - 1 - 0.5 - 0.5 - 1 = 3$

  $\sigma_6(0) = d_6 - e_1 - e_2 - e_3 - e_4 - e_5 - e_6 = 7 - 0.5 - 1 - 0.5 - 0.5 - 1 - 1.2 = 2.3$
- these values are the diagonal elements $\sigma_i(0)=\omega(i;i)$ in the pre-computed table:
- all other elements in the table are immediately derived from the diagonal elements, : ($\boldsymbol{\omega(j;k) = \min_{j \leq i \leq k} \{\sigma_i (0)\} = \min_{j \leq i \leq k} \{ \omega(i;i) \}}$)

i	*j* 1	2	3	4	5	6
1	ω(1;1)	ω(1;2)	ω(1;3)	ω(1;4)	ω(1;5)	ω(1;6)
2		ω(2;2)	ω(2;3)	ω(2;4)	ω(2;5)	ω(2;6)
3			ω(3;3)	ω(3;4)	ω(3;5)	ω(3;6)
4				ω(4;4)	ω(4;5)	ω(4;6)
5					ω(5;5)	ω(4;6)
6						ω(6;6)

i	*j* 1	2	3	4	5	6
1	1.5	1.5	1.5	1.5	1.5	1.5
2		2.0	2.0	2.0	2.0	2.0
3			2.0	2.0	2.0	2.0
4				3.5	3.0	2.3
5					3.0	2.3
6						2.3

Quicker slack computation (3)

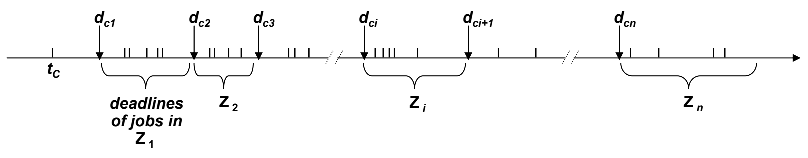
- When the system is in execution within an hyperperiod, the slack of each job decreases from its initial value when any of these events occur:
  - the processor is idling,
  - the slack stealer is executing,
  - lower priority jobs are executing;
- The slack of J_i at time t_c may be computed as:

  $\sigma_i (t_c) = \sigma_i (0) - I(t_c) - ST(t_c) - \Sigma_{d_k > d_i} \xi_k(t_c)$, where:
  - $\sigma_i(0) = \omega(i;i)$,
  - $I(t_c)$ is the processor idle time since $t = 0$ (beginning of the hyperperiod),
  - $ST(t_c)$ is the time the slack stealer has been executing since $t = 0$,
  - $\xi_k(t_c)$ is the time the lower priority job J_k has been executing since $t = 0$.
- The slack of the system at time t_c is the minimum slack at time t_c of all jobs released in the hyperperiod.

Quicker slack computation (4)

- In order to speed up the computation of $\sigma_i (t_c)$, we observe that at time t_c :
  - there are n current jobs (t_c is in their feasible interval), one per each task,
  - all jobs that precede the current ones have completed, and their execution times have been accounted for in the pre-computed values of the initial slacks $\sigma_{ci} (0)$ of all current jobs J_{ci} ,
  - all jobs that follow the current ones have non started (for them $\xi_k(t_c) = 0$),
  - only current jobs may have started execution and their (partial) execution times $\xi_k(t_c)$ must be subtracted from the slack of higher priority ones.
- Let's consider the deadlines of the n jobs J_{ci} (i = 1 to n) that are current at t_c :

  d_{c1} , d_{c2} , …, d_{cn} arranged in such a way that $d_{c1} \leq d_{c2} \leq \ldots \leq d_{cn}$.
- we may partition all the periodic jobs in the hyperperiod that have deadlines after t_c into n subsets Z_i (i = 1 to n) in such a way that a job is in Z_i if its deadline belongs to the time interval [d_{ci} , d_{ci+1}):



Quicker slack computation (5)

- since the values of $\xi_k(t_c)$ are non zero only for current jobs, we observe that, in computing the slack of J_i at t_c : $\sigma_i(t_c) = \sigma_i(0) - I(t_c) - ST(t_c) - \Sigma_{d_k > d_i} \xi_k(t_c)$, for every job in each subset Z_i, the amount $\Sigma_{d_k > d_i} \xi_k(t_c)$ is the same as for J_{ci} and is equal to $\Sigma_{dc_k > dc_i} \xi_{ck}(t_c)$, which may be rewritten as $\Sigma_{k=i+1 \text{ to } n} \xi_{ck}(t_c)$ (the sum of the execution portions of current jobs with deadlines larger than d_{ci});
- also the amounts $I(t_c)$ and $ST(t_c)$ are the same for every job in each subset Z_i;
- then the slack of every job in Z_i is equal to its initial slack minus the same quantity: $I(t_c) + ST(t_c) + \Sigma_{k=i+1 \text{ to } n} \xi_{ck}(t_c)$;
- then the job that at t_c has the minimum slack among the jobs in Z_i is the one that has the minimum initial slack:
- the minimum slack of all the jobs in Z_i can be derived from the pre-computed initial slack table:

  $\omega_i(t_c) = \omega(c_i ; c_{i+1} - 1) - I(t_c) - ST(t_c) - \Sigma_{k=i+1 \text{ to } n} \xi_{ck}(t_c)$ for $i = 1, 2, \ldots, n - 1$,

  and

  $\omega_n(t_c) = \omega(c_n; N) - I(t_c) - ST(t_c)$
- the slack of the system at t_c is:

  $\sigma(t_c) = \min_{1 \le i \le n} \omega_i(t_c)$
- it is easy to see that the above computation has complexity $O(n)$.

Slack computation: the scheduler

- The scheduler must compute the slack times of every job (with the purpose of finding their minimum value, which is the slack $\sigma(t_c)$ of the system), at any time t_c when any of these events occur:
  - a job is released,
  - a job completes,
  - the slack is exhausted;
- at these times the scheduler performs the following operations:
  - update the values of I, ST, and ξ_i , for i = 1 to N: these values are set to zero at the beginning of each hyperperiod; then a counter is used to keep track of the times when the processor idles, or when the slack stealer is executing, or when a periodic job J_i is executing;
  - if the aperiodic queue is non empty,
    - perform slack computation to find $\sigma(t_c)$ (using the method described in the previous slide);
    - if $\sigma(t_c) > 0$, then schedule the slack stealer to execute the aperiodic job at the head of the queue;
    - else schedule the highest priority periodic job;
  - else schedule the highest priority periodic job.

Slack stealing vs. servers

- We have seen that slack computation in EDF systems, though conceptually simple, is rather time consuming;
- in fixed priority systems the problem of monitoring the slack available in the system is even more complex;
- for these reasons the execution of aperiodic job is rarely performed by slack stealers;
- the use of aperiodic job servers (such as polling servers, or other types of servers) is preferred;
- an aperiodic job server is some sort of periodic task (scheduled together with the other "real" periodic tasks of some application) whose purpose is to provide a fraction of processor utilization for the execution of aperiodic jobs when the aperiodic job queue is not empty.

Polling Server

- Polling Server (Poller) $PS = (p_s, e_s)$: scheduled as a periodic task, the polling server takes care of executing aperiodic jobs when there are any waiting;
  - p_s: poller becomes ready for execution every p_s time units,
  - e_s: is the upper bound on execution time.
- Terminology:
  - execution budget: e_s
  - size of the server: $u_s = e_s / p_s$ (it's the maximum utilization factor of *PS*),
  - replenishment rule: the budget is set to e_s at the beginning of each period.
  - consumption rules:
    - the poller consumes its budget at the rate of 1 per time unit, while it is executing aperiodic jobs;
    - the poller exhausts its budget whenever it finds the aperiodic job queue empty;
  - whenever the budget is exhausted, the scheduler removes the poller from the periodic queue until it is replenished;
  - the server is idle when the aperiodic job queue is empy;
  - the server is backlogged when the aperiodic job queue is not empy.

Polling Server: example

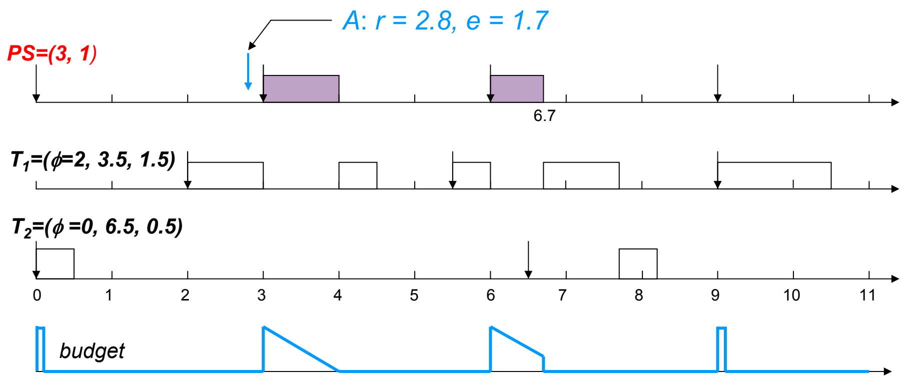
Rate-Monotonic Scheduling: *PS=(3, 1)*, *T_1=(ϕ =2, 3.5, 1.5), T_2=(ϕ =0, 6.5, 0.5)*

$U = 1/3 + 1.5/3.5 + 0.5/6.5 \cong 0.8388 > \mathbf{U_{RM}}(3) = 0.780$

⇨the task set may not be RM schedulable; perform time-demand analysis:

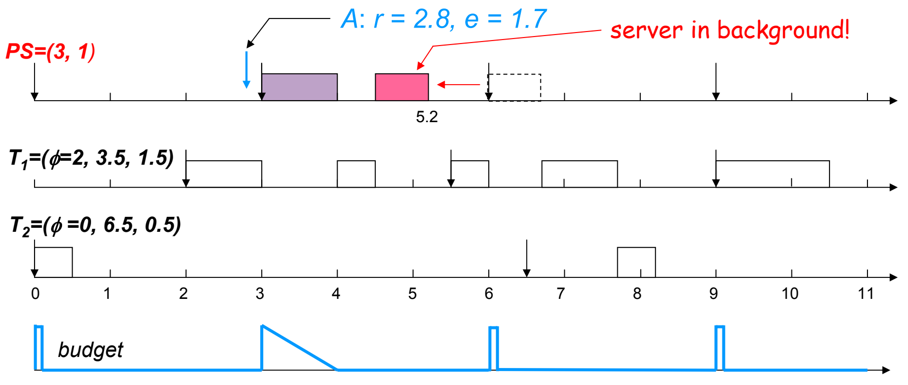
$w_1(3.5) = e_1 + \lceil 3.5/p_s \rceil e_s = 1.5 + 2 = 3.5 \leq 3.5$ (OK)

$w_2(6.5) = e_2 + \lceil 6.5/p_s \rceil e_s + \lceil 6.5/p_1 \rceil e_1 = 0.5 + 3 + 2\times 1.5 = 6.5 \leq 6.5$ (OK)



Polling Server + Background Server

Rate-Monotonic Scheduling: from $t = 4.5$ to $t = 5.5$ the processor idles: this time interval could be used to proceed execution of A (in background)



- using the background time, the response time of A is reduced from:
  - 3.9 (= 6.7 - 2.8 polling server alone) to
  - 2.4 (= 5.2 - 2.8 polling server + background server).

Bandwidth Preserving Servers

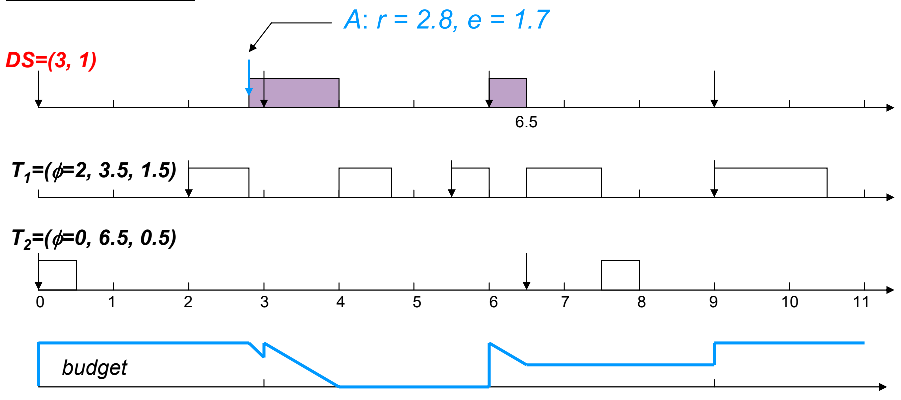
- Problem with polling servers:
  - aperiodic jobs released after the poller has found the queue empty, must wait for the poller to examine the queue again, one polling period later: their response time is unduly longer;
  - if the poller could preserve its budget, when it finds the aperiodic job queue empty, and use it later in the period, the response time of some aperiodic jobs would be shortened.
- Bandwidth-preserving server algorithms:
  - improve in this manner upon polling approach,
  - use periodic servers,
  - are defined by consumption and replenishment rules.
- 3 types of bandwidth-preserving server algorithms:
  - deferrable servers,
  - sporadic servers,
  - constant utilization / total bandwidth / weighted fair queuing - servers.

Deferrable Servers

- Rules:
  - Consumption: execution budget consumed only when server executes.
  - Replenishment: execution budget is set to e_s at each multiple of p_s.
- Preserves budget when no aperiodic job is ready.
- Any budget held prior to replenishment is lost (no carry-over).

Example: deferrable server with RM

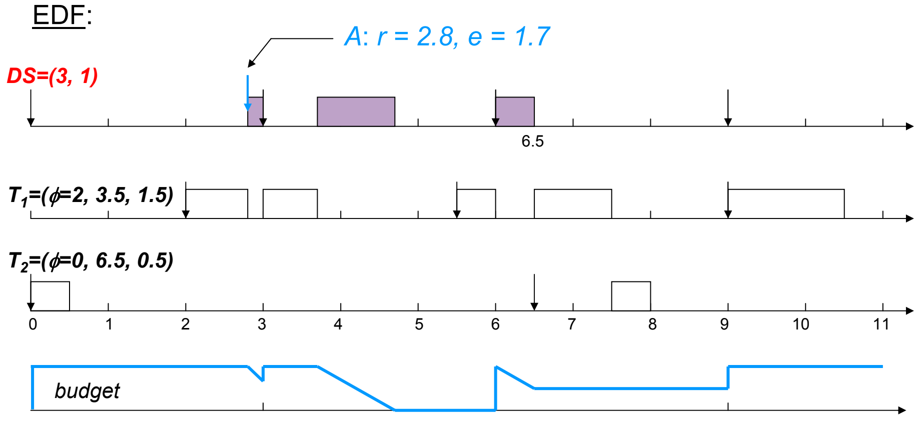
Rate-Monotonic:



- is schedulability guaranteed by the time-demand analysis of slide 7?
- it would if *DS* behaved as a periodic task (we will face this problem later)

Example: deferrable server with EDF

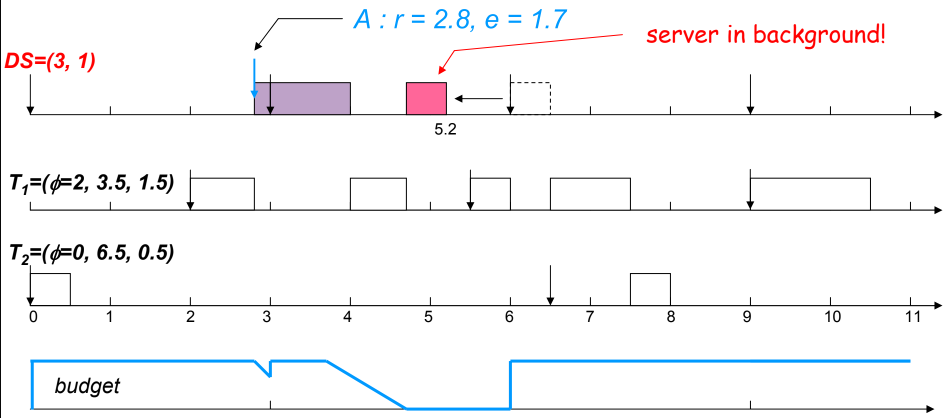
EDF:



- is EDF schedulability guaranteed? ($U \leq 1/3 + 1.5/3.5 + 0.5/6.5 \cong 0.8388 < 1$)
- it would if *DS* behaved as a periodic task (we will face this problem later)

Deferrable Server + Background Server

Rate-Monotonic: from t = 4.7 to t = 5.5 the processor idles:
this time interval could be used to proceed execution of A (in background)

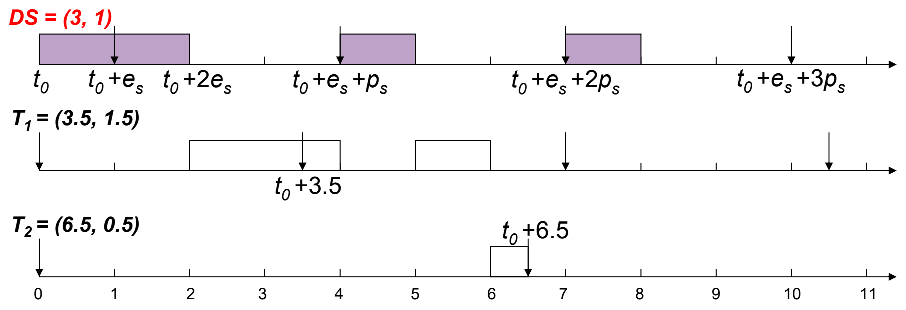


- what about schedulability?

  the use of background time does not affect schedulability.

DS: Why not increase the budget?

Rate-Monotonic - consider a critical instant t_0 for T_1 and T_2; *DS* backlogged:



- in the figure the *DS* (highest priority) at $t = t_0$ starts using up its budget twice in a row ($e_s + e_s$ = 2 time units): once at the end of one of its periods, and once again at the beginning of the next one (back to back);
- the job $J_{1,1}$ of T_1, released at t = 0 has just (3.5 - 1 - 1) = 1.5 time units for its execution available in its period (from t_0 to 3.5);
- if the budget of *DS* were any greater than 1, the job $J_{1,1}$ of T_1 would miss its deadline at t = 3.5

DS: Fixed-Priority systems

(case 1: *DS* has highest priority)

- Lemma: in a fixed-priority periodic system in which $D_i \leq p_i$, with a deferrable server T_{DS} (p_s, e_s) with highest priority, a critical instant for a periodic task T_i happens when:
  (1) $r_{i,c} = t_0$ for some job $J_{i,c}$ in T_i ,
  (2) a job of every higher-priority tasks is released at time t_0,
  (3) the budget of (backlogged) server is e_s at time t_0,
  (4) next replenishment time is $t_0 + e_s$
- Intuitively:
  - low-priority tasks suffer from a "back-to-back" hit by the deferrable server;
  - this is due to the fact that, unlike a "true" periodic task (and unlike a polling server), the deferrable server may be "released" later than the beginning of its period.
- a schedulable task set including a "true" periodic task T (p_s, e_s) may be no longer schedulable, if T is a deferrable server T_{DS} (p_s, e_s).

Time-Demand analysis with *DS*

- when there is a deferrable server, the time-demand analysis for tasks (with priority lower than the *DS*), must consider this back-to-back effect:
- starting from a critical instant (at $t = 0$) the maximum amount of time demanded by the (higher priority) deferrable server T_{DS} (p_s, e_s) is:

$$e_s + \left\lceil \frac{t - e_s}{p_s} \right\rceil e_s$$

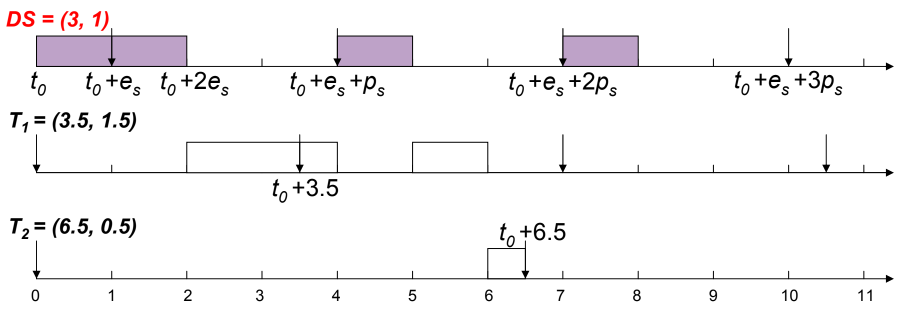
- this amount must be added to the time-demand function of any (lower priority) task T_i:

$$w_i(t) = e_i + e_s + \left\lceil \frac{t - e_s}{p_s} \right\rceil e_s + \sum_{k-1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k$$

- to determine whether the response time of T_i exceeds its relative deadline D_i , in the case $D_k \leq p_k$ (for all k = 1, .. i), we check whether $w_i(t) \leq t$ is satisfied at any of the following values of $t \leq D_i$:
  - at t = integer multiples of p_k (for k = 1, .. i -1) (when the time demand function increases due to the release of higher priority tasks T_k) and at D_i
  - at $t = e_s$, $e_s + p_s$, $e_s + 2p_s$, $e_s + 3p_s$, $e_s + \lceil (D_i - e_s)/p_s \rceil p_s$ (when the time demand function increases due to the replenishment of T_{DS}).

T-D analysis with *DS*: example

Rate-Monotonic:



time-demand analysis of schedulability:

$$w_i(t) = e_i + e_s + \lceil \frac{t - e_s}{p_s} \rceil e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil e_k$$

$w_1(3.5) = e_1 + e_s + \lceil (3.5 - e_s)/p_s \rceil e_s = 1.5 + 1 + 1 = 3.5 \leq 3.5$ (OK)

$w_2(6.5) = e_2 + e_s + \lceil (6.5 - e_s)/p_s \rceil e_s + \lceil 6.5/p_1 \rceil e_1$

$= 0.5 + 1 + 2 + 2\times 1.5 = 6.5 \leq 6.5$ (OK)

DS: arbitrary Fixed-Priority

(case 2: *DS* may not have the highest priority)

- any budget that is not consumed at end of the server period is lost;
- if *DS* is not the highest priority task, then the maximum amount of time DS can consume may be limited, since it depends on:
  - release time of higher priority periodic jobs (with respect to replenishment times),
  - execution times of all tasks;
- in this situation, the back-to-back hit effect may never take place completely and the time used by the server in a time interval of length *t* may never be as large as $e_s + \lceil (t - e_s)/p_s \rceil e_s$
- however that is an upper bound on time demanded by the *DS* for tasks with priority lower than *DS* and we may use the time-demand analysis method as a sufficient schedulability test for any fixed-priority system containing one deferrable server (correctly, if not accurately):

$$w_i(t) \leq e_i + e_s + \lceil \frac{t - e_s}{p_s} \rceil e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil e_k$$

Schedulable utilization

- there is no known schedulable utilization that assures the schedulability of a fixed priority system with a deferrable server scheduled at an arbitrary priority.
  - Only exception, in the special case when:
    - $p_s < p_1 < p_2 < \ldots < p_n$
    - $p_n < 2p_s$
    - $p_n > p_s + e_s$
    - $p_i = D_i$
    - rate-monotonic scheduling
  - for this case, the schedulable utilization is:

$$U_{RM/DS}(n) = (n-1)\left(\frac{u_s + 2^{\frac{1}{n-1}}}{u_s + 1} - 1\right)$$

Using the schedulable utilization

- Assume that T_i has lower priority than the server T_{DS}.
- $T_{DS}\,(p_s, e_s)$ behaves like a periodic task (p_s, e_s), except that it may execute for at most e_s additional time units during the interval $(r_{i,c}, r_{i,c} + D_i)$:
  - we may consider it as an additional blocking time for T_i

$$\sum_{k=1}^{i} u_k + u_s + \frac{e_s + b_i}{p_i} \leq U_X(i+1)$$

X = scheduling algorithm

- Example (RM):

  T_1 = (3, 0.6)

  T_{DS} = (4, 0.8)

  T_2 = (5, 0.5)

  T_3 = (7, 1.4)

- T_1: not affected by T_{DS}
- T_2: $\sum_{k=1}^{2} u_k + u_s + \frac{e_s}{p_2} = 0.66 \leq 0.7797 = U_{RM}(3)$
- T_3: $\sum_{k=1}^{3} u_k + u_s + \frac{e_s}{p_3} = 0.8143 \not\leq 0.757 = U_{RM}(4)$ no!

- T_3 may not be schedulable (sufficient condition, but not necessary); to find out: perform time-demand analysis (next slide).

Example: time-demand analysis

- Example (RM):

  $T_1 = (3, 0.6)$

  $T_{DS} = (4, 0.8)$

  $T_2 = (5, 0.5)$

  $T_3 = (7, 1.4)$

- the utilization factor method did not succeed to prove that T_3 is schedulable
- to find out, we perform time-demand analysis:

$$w_i(t) = \boxed{e_i + e_s + \lceil \frac{t - e_s}{p_s} \rceil e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil e_k \ \leq t}$$

- find whether the above inequality $w_3(t) \leq t$ has solutions for $t \leq 7$:
  - evaluate it only for $t = 7$ and for values of t at which the step function rises: multiples of p_k and at DS replenishment times;
  - starting with $t = 7$:

    $w_3(7) = 1.4 + 0.8 + \lceil (7 - 0.8)/4 \rceil 0.8 + \lceil 7/3 \rceil 0.6 + \lceil 7/5 \rceil 0.5$

    $w_3(7) = 2.2 + 2 \times 0.8 + 3 \times 0.6 + 2 \times 0.5$

    $w_3(7) = 6.6 \leq 7$
- for $t = 7$ the inequality is satisfied ⇨ T_3 is schedulable

Multiple deferrable servers

- Introducing more than one deferrable server, it is possible to adjust their parameters (priorities, budgets, periods) to improve the response time of some aperiodic tasks at the expense of other aperiodic tasks;
  - the effect of each deferrable server must be taken into account in performing time-demand analysis;
  - time demand for a task T_i with priority lower than m deferrable servers $T_{DSq}(p_{s,q}, e_{s,q})$ $(q = 1,\dots m)$:

$$w_i(t) \leq e_i + \sum_{q=1}^{m} \left(1 + \lceil \frac{t - e_{s,q}}{p_{s,q}} \rceil \right) e_{s,q} + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil e_k$$

Schedulability for EDF Systems

- Theorem: A periodic task T_i in a system of n independent, preemptive periodic tasks is schedulable together with a *DS* with period p_s, execution time e_s, and utilization u_s, according to the EDF algorithm if

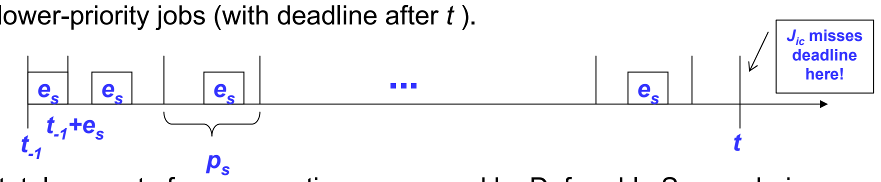
$$\sum_{k-1}^{n} \frac{e_k}{min(D_k, p_k)} + u_s \left(1 + \frac{p_s - e_s}{D_i}\right) \leq 1$$

Proof:

- we will show that if some job $J_{i,c}$ misses its deadline, than the above inequality is not satisfied.

Proof

- let t be the missed deadline of some job $J_{i,c}$ and t_{-1} be the latest instant at which the processor is idle or is executing a job with deadline after t:
- during interval (t_{-1}, t] the processor never idles and no time is given to lower-priority jobs (with deadline after t).



- total amount of processor time consumed by Deferrable Server during interval (t_{-1}, t] is at most equal to:

$$e_s + \left\lfloor \frac{t - t_{-1} - e_s}{p_s} \right\rfloor e_s$$

- this happens when (see figure):
  - at t_{-1} *DS* has full budget e_s , highest priority, and is replenished at $t_{-1} + e_s$;
  - the aperiodic job queue is never empty in the interval (t_{-1}, t);
- we used "*floor*" instead of "*ceiling*" because the last release of *DS* has deadline at or after t: in the latter case it is not given any processor time before t (EDF); in the first case the fraction is an integer (equal to its *floor*).

Proof (2)

- the time demanded by *DS* in the interval (t_{-1}, t) is:

$$w_{DS}(t-t_{-1}) \le e_s + \lfloor \frac{t-t_{-1}-e_s}{p_s} \rfloor e_s \le e_s + \frac{t-t_{-1}-e_s}{p_s} e_s = u_s(t-t_{-1}+p_s-e_s)$$

we observe that (as in the case of fixed-priority systems) in the interval (t_{-1}, t) the *DS* may demand an extra amount of time, in addition to the amount $u_s(t - t_{-1})$ required at most by a "real" periodic task T_s=(p_s , e_s):

- **in fixed-priority systems, this extra amount of time is at most e_s ;**
- **in deadline-driven systems, this extra amount of time is at most $(p_s - e_s)u_s$; since $(p_s - e_s)u_s = e_s - e_su_s = e_s (1 - u_s) < e_s$, this amount is always less than e_s**

- the time demanded by any other task T_k in the interval (t_{-1}, t) is:

$$w_k(t-t_{-1}) = e_k \lfloor \frac{t-t_{-1}}{p_k} \rfloor \le e_k \frac{t-t_{-1}}{p_k}$$

- J_{ic} misses deadline at time *t* if there is not enough time to finish by time *t*:

$$t-t_{-1} < \sum_{k=1}^{n} \frac{e_k}{p_k}(t-t_{-1}) + u_s(t-t_{-1}+p_s-e_s)$$

Proof (3)

- J_{ic} misses its deadline at time *t*, if there is not enough time to finish by time *t*:

$$t-t_{-1} < \sum_{k=1}^{n} \frac{e_k}{p_k}(t-t_{-1}) + u_s(t-t_{-1}+p_s-e_s)$$

- dividing both sides of the inequality by (t - t_{-1}) we get:

$$\sum_{k=1}^{n} \frac{e_k}{p_k} + u_s\left(1+\frac{p_s-e_s}{t-t_{-1}}\right) > 1$$

- since
  - $\min(D_k , p_k) \le p_k$ and
  - $D_i \le (t - t_{-1})$ ($D_i = (d_{ic} - r_{ic})$, with $d_{ic} = t$ and $r_{ic} \ge t_{-1}$ by definition of t_{-1}),

  we have:

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + u_s\left(1+\frac{p_s-e_s}{D_i}\right) > 1$$

Proof (4)

- we have shown that if some job misses its deadline at time *t*, than the following inequality holds:

$$\sum_{k=1}^{n} \frac{e_k}{min(D_k, p_k)} + u_s\left(1 + \frac{p_s - e_s}{D_i}\right) > 1$$

- in other words, the system of *n* tasks is schedulable if:

$$\sum_{k=1}^{n} \frac{e_k}{min(D_k, p_k)} + u_s\left(1 + \frac{p_s - e_s}{D_i}\right) \leq 1$$

- this is a sufficient condition for schedulability; but it is not necessary.
- The term $(p_s - e_s)u_s$, which gives the maximum amount of time required by a $DS(p_s, e_s)$ in the feasible interval of any job, in addition to the time required by a normal periodic task $T(p_s, e_s)$, may be considered as an additional blocking time suffered by the job.

Example: *DS* in an EDF system

- Example (EDF):
  - T_1 = (3, 0.6)
  - T_2 = (5, 0.5)
  - T_3 = (7, 1.4)
  - T_{DS} = (4, 0.8)

- to find out whether the system is schedulable, we consider one task at the time and evaluate the inequality:

$$\sum_{k=1}^{n} \frac{e_k}{min(D_k, p_k)} + u_s\left(1 + \frac{p_s - e_s}{D_i}\right) \leq 1$$

the value of the first term is: $e_1/p_1 + e_2/p_2 + e_3/p_3 = 0.6/3 + 0.5/5 + 1.4/7 = 0.5$

T_1: 0.5 + (0.8/4)(1 + (4 - 0.8)/3) = 0.5 + 0.413 = 0.913

T_2: 0.5 + (0.8/4)(1 + (4 - 0.8)/5) = 0.5 + 0.328 = 0.828

T_3: 0.5 + (0.8/4)(1 + (4 - 0.8)/7) = 0.5 + 0.291 = 0.791

⇨ the 3 tasks are schedulable.

- we note that a *DS* task behaves like a periodic task (p_s, e_s), except that it may execute an extra amount of time in the feasible interval of any job:
  - in a fixed-priority system, this extra amount of time is at most e_s;
  - in a deadline-driven system, the amount of time is at most $(p_s - e_s)u_s$ (which is always less than e_s).

Sporadic servers

- Problem with Deferrable Server: $T_{DS}(p_s, e_s)$ may delay lower-priority jobs longer than the periodic task $T(p_s, e_s)$.
- Sporadic Server (SS): in any time interval a SS never uses more time than the periodic task $T(p_s, e_s)$ with the same parameters. Then we can treat $T_{SS}(p_s, e_s)$ just as a periodic task.
- A system of periodic tasks containing a sporadic server may be schedulable, while the same system containing a deferrable server with the same parameters is not.
- Different types of sporadic servers differ in their consumption and replenishment rules: more complicated rules allow the server:
  - to maintain its budget for a longer time,
  - to replenish the budget more aggressively
- The name "sporadic" server indicates that replenishment times may occur earlier or later than p_s time units after the previous one: inter-release times may be shorter or arbitrarily longer than p_s (similarly to sporadic tasks).

SS in fixed-priority systems

- Notations:

– $\boldsymbol{T}$:	task system with n tasks,
– T_{SS} :	Sporadic Server, with arbitrary priority,
– $\boldsymbol{T}_H$:	subset of $\boldsymbol{T}$ with priority higher than T_{SS},
– t_r :	latest replenishment time,
– t_f :	first instant after t_r at which the server begins to execute,
– t_e :	*effective* replenishment time: the scheduler determines t_e based on history (in some circumstances t_e may be later than t_r), and sets the next replenishment time to $t_e + p_s$,
– *BEGIN* :	at any time t, it is the latest instant at which a lower-priority task was executing (or the system was idle), before the latest busy interval of $\boldsymbol{T}_H$ that started before t,
– *END* :	end of the latest time interval before t during which $\boldsymbol{T}_H$ is constantly busy.

Simple SS: consumption rules

- Consumption Rules:

  The server's execution budget is consumed (at the rate of 1 per time unit) at any time t after t_r until the budget is exhausted, whenever either one of the following two conditions are true:
  (when these conditions are not true, the server holds its budget)

  - C1: the server is executing.
  - C2: the server has executed since t_r, has become idle before time t, and T_H is idle ($END < t$).

Rule C2 means that:

- unlike a deferrable server, the budget of an idle simple sporadic server continues to decrease with time after it becomes idle and T_H is also idle;
- the sporadic server holds on to its budget when:
  - some higher priority job is executing, or
  - the server has not executed since t_r.

Simple SS: replenishment rules

- Replenishment Rules:

  R1: the execution budget is set to e_s and the current time t_r is recorded initially and each time the budget is replenished.

  R2: t_e and the next replenishment time are determined at time t_f, when the server first begins to execute since t_r; at time t_f, t_e is set as follows:

  - $t_e = t_r$ if T_H has been busy throughout the interval (t_r, t_f)
    (i.e. $t_e = t_r$ if $END = t_f$ and $t_r \geq BEGIN$); else
  - t_e = the latest instant at which a lower-priority task executes in (t_r, t_f)
    (i.e. $t_e = t_f$ if $END < t_f$; $t_e = BEGIN$ if $END = t_f$ and $t_r < BEGIN$)

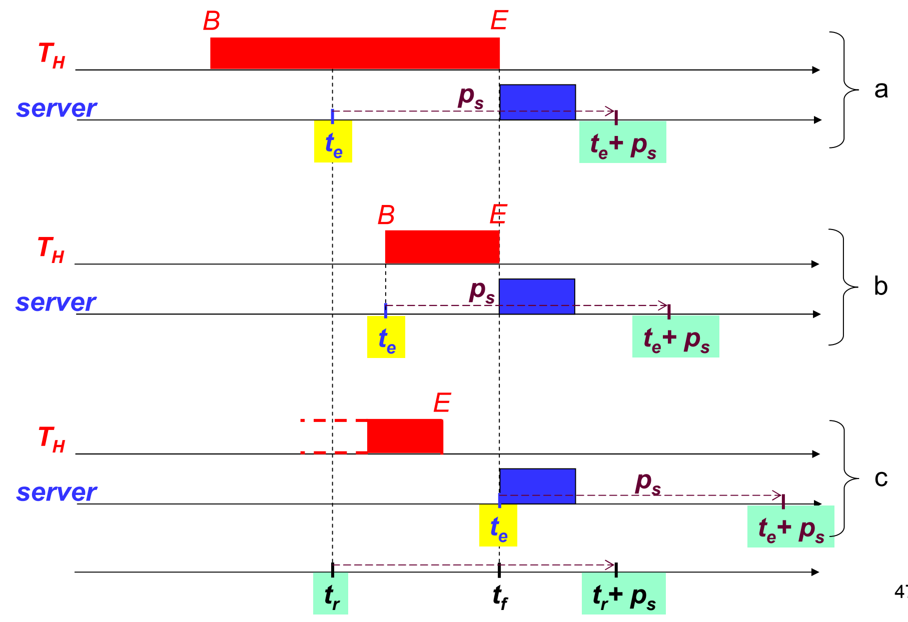
  the next replenishment time is then set at $t_e + p_s$.
  (note that the next replenishment time in never earlier than $t_r + p_s$)

  R3: The next replenishment occurs at the next replenishment time, except under the following conditions, when it may be done sooner or later:

  (a) if the next replenishment time $t_e + p_s$ is earlier than t_f, the budget is replenished as soon as it is exhausted (**later**);

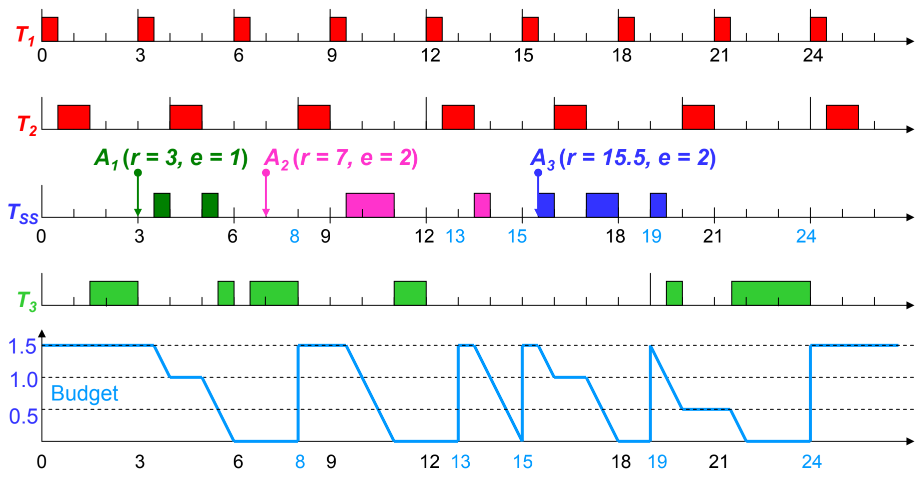
  (b) the budget is replenished at time t whenever the system T has been idle before t and a periodic job is released at t (**sooner**).

SSS: replenishment rule R2



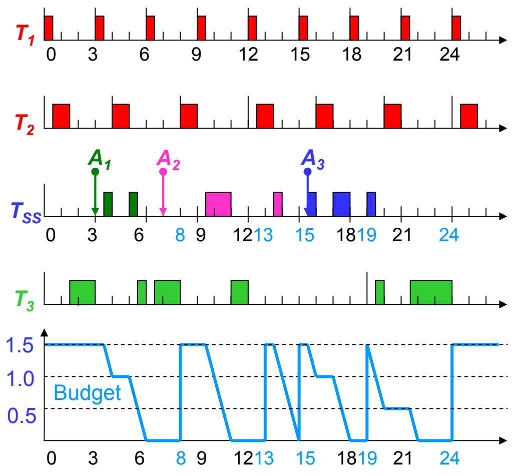
Simple Sporadic Server: Example

$T_1 = (3, 0.5)$ $T_2 = (4, 1.0)$ $T_3 = (19, 4.5)$ $T_{SS} = (5, 1.5)$ **(RM scheduling)**



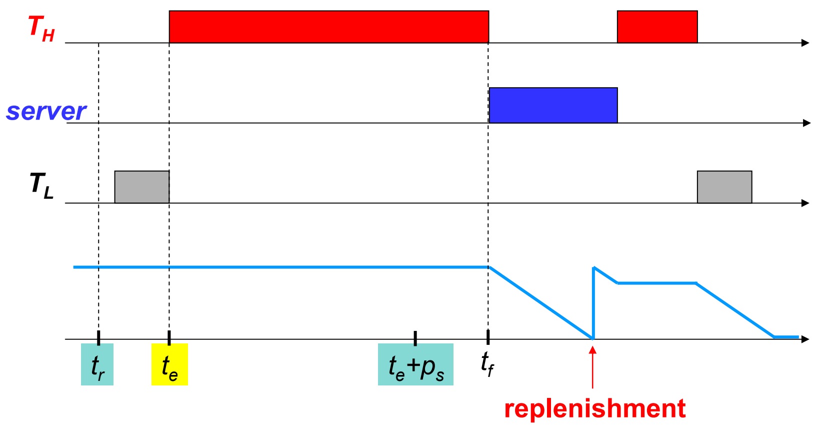
Simple Sporadic Server: Example

T_1 = (3, 0.5) T_2 = (4, 1.0) T_3 = (19, 4.5) T_{SS} = (5, 1.5) (RM scheduling)



0-3.0: Aperiodic queue is empty: budget stays at 1.5.
3-3.5: AP job arrives, but has to wait for H.P. job to finish.
3.5: Server starts.
R2 -> t_e=3: latest time at which LP task T_3 executed
R1 -> next replenishment time = 8
3.5-4: Server executes.
C1 -> budget decreases
4: Server preempted by T_2. Holds its budget (C1+C2).
5: Server resumes execution: consumes budget to exhaustion:.
5-5.5: executing (C1)
5.5-6: T_1 and T_2 idle (C2)
7.0: AP job arrives, but budget exhausted. Job waits.
8.0: Budget replenished
9.5: Server begins: t_e = 8 (R2);
next replenishment time = 13.
11: Server exhausts budget. A_2 not completed.
13.0: Budget replenished.
13.5: Server begins: t_e = 13 (R2);
next replenishment time = 18.
Server consumes budget to exhaustion:.
13.5-14: executing (C1)
14-15: T_1 and T_2 idle (C2)
14-15: task system idle -> replenishment time = 15,
when system starts again (R3b)
…

A situation when rule R3a applies



Simple SS: proof of correctness

“Correctness”: the server never demands more time in any interval than the corresponding periodic task $T_s = (p_s, e_s)$: so in schedulability analysis we can treat the server as the periodic task (if T_s sometimes has inter-release times larger than p_s and execution times shorter than e_s, the schedulability of lower priority tasks is not adversely affected);

- rule R2 means that, while ***T*** is not idle, the time between two consecutive replenishments is never lower than p_s (it is equal to p_s when T_H has been always busy in the interval t_r— t_f; it is larger otherwise);
- the time between two consecutive replenishments may be lower than p_s only when rule R3b applies (when the whole system ***T*** has become idle).

- we consider first the case when ***T*** has become idle and rule R3b applies:
  - we recognize that in this case the behavior of the system, when it starts to be busy again, is independent of what happened in the previous busy interval (we may think that the system is starting from scratch).
- we consider then the case when ***T*** is never idle, and rule R3b never applies:
  - we show that in this case the server correctly “emulates” task $T_s= (p_s,e_s)$ (with some inter-release times > p_s and some $e_i < e_s$).

SSS: proof of correctness (2)

- we are considering the case when ***T*** is never idle, and rule R3b never applies: we will show that all consumption and replenishment rules are correct, i.e. the server emulates the periodic task $T_s= (p_s,e_s)$.

⇨ rule R1 is correct: view replenishment times t_r as nominal “release times” of the server job; actual release times t_e may be later than nominal ones;

- rule C1: each server job never executes for more than its budget e_s.
- rule C2: after sporadic server gets idle, its budget decreases as if it was executing.

$\Rightarrow$ each server job only executes at times when a job of T_s would.

- rule C2 also means that server holds on to its budget when:
  - jobs in T_H are executing:
    - obviously correct
  - the server has not executed since t_r:
    - in this case, when the server starts executing at t_f, t_e is set equal to t_f (R2), and the server emulates a job of $T_s= (p_s,e_s)$ with actual release time later than its nominal release time.

⇨ rules C1 and C2 are correct.

SSS: proof of correctness (3)

- rules R2 and R3a make sure that the next replenishment time is always set p_s time units later than effective release time t_e and the next effective release time is never earlier than the next replenishment time.
  - R2: makes next effective replenishment times as early as possible without making the server behave differently from a periodic task :
    - t_e is set equal to t_r if T_H has been always busy in the interval $t_r — t_f$: this emulates a job of T_s released at t_r and delayed by H.P. jobs;
    - t_e is set equal to a value later than t_r if L.P. jobs have executed in the interval $t_r — t_f$; there are two possible cases:
      1. L.P. jobs were executing immediately before t_f: in this case the server emulates a job of T_s released later than usual;
      2. L.P. jobs have executed only before a busy interval of T_H (from t_r to *BEGIN*; in this case t_f = *END*): in this case the server emulates a job of T_s released (later) at t_e and delayed by H.P. jobs;
  - R3a: emulates a situation where jobs in T_s takes more time to complete than one period (this works if jobs are allowed to have $D_i > p_i$).

⇨ rules R2 and R3a are correct.

Sporadic/Background Server

- In the previous example, at time 18.5, A_3 remains incomplete (because the server budget is exhausted) while the periodic system is idle:
  – rule R3b is overly conservative,
  – the server could use the background time (when the periodic system is idle) to execute aperiodic jobs, regardless of its budget, without affecting the schedulability of the periodic system.
- A SPORADIC/BACKGROUND SERVER is an enhancement of the SSS that takes advantage of this possibility:
  – consumption rules: the same as for SSS, except when the periodic system is idle: (C3) under these circumstances the budget stays at e_s ;
  – replenishment rules: the same as for SSS, except R3b: the budget is replenished as soon as the periodic system becomes idle and t_r is set at the end of the idle interval.
- A sporadic/background server is a combination of a SSS and a background server.

Other enhancements of SSS

- Cumulative Replenishment

  at replenishment time:
  - the amount of the previous budget left unconsumed is kept (old budget),
  - the budget is incremented by e_s : after replenishment there may be 2 chunks of budget: the old chunk (leftover) and the new chunk (e_s),
  - rule C2 must be modified:
    - the old chunk of budget must be consumed first (this emulates the behavior of jobs executed in FIFO order when more than one is ready at the same time);
    - after t_r the old chunk of budget decreases at the rate of 1 per time unit also when the server has not executed since t_r (it had before t_r).

Other enhancements of SSS (2)

- SpSL Server:

  the budget is consumed and replenished in chunks:
  - chunk breaking rule: whenever the server is suspended, the chunk of budget being consumed just before suspension is broken in two chunks:
    - the first chunk is the portion that was consumed: it inherits the next replenishment time of the original chunk;
    - the second chunk is the remaining portion: it inherits the last replenishment time of the original chunk (its effective and next replenishment times will be determined later);
  - consumption rules:
    - the server consumes its budget only when it is executing;
    - chunks of budget are consumed in the order of their last replenishment time
  - replenishment rules: the same rules R2 and R3 of the SSS apply; the chunks are consolidated into one whenever they are replenished at the same time.
- SpSL Server emulates several periodic tasks (one for each chunk; their number varies with time) with the same period and total execution time = e_s.

SSS in EDF systems

in EDF systems some definitions have no meaning; t_e has a different meaning

- Notations:
  - $\boldsymbol{T}$: task system with *n* tasks.
  - T_{SS} : Sporadic Server, with arbitrary priority.
  - $\boldsymbol{T}_H$: subset of $\boldsymbol{T}$ with higher priority than T_{SS}.
  - t_r : latest replenishment time.
  - t_f : first instant after t_r at which server begins to execute.
  - t_e : *effective* replenishment time: the scheduler determines t_e based on history (in some circumstances t_e may be later than t_r), and sets the next replenishment time to $t_e + p_s$.
  - *BEGIN* : at any time *t*, it is the latest instant at which a lower-priority task was executing (or the system was idle), before the latest busy interval of $\boldsymbol{T}_H$ that started before *t*.
  - *END* : end of the latest time interval during which $\boldsymbol{T}_H$ is constantly busy.

SSS in EDF systems

- Notations:
  - $\boldsymbol{T}$: task system with *n* tasks.
  - T_{SS} : Sporadic Server.
  - t_r : latest replenishment time.
  - t_e : *effective* replenishment time:
    - is undefined when the aperiodic job queue is empy
    - is determined when an aperiodic job arrives at an empty aperiodic job queue
  - *d* : *deadline* of the current sporadic server job:
    - is defined ($d = t_e + p_s$) when t_e is defined

The server is ready for execution (t_e and d are defined):
- when it is backlogged (the aperiodic queue is not empty);

The server is suspended (t_e and d are undefined):
- when it is idle (the aperiodic queue is empty).

SSS in EDF: consumption rules

- Consumption Rules:

  The server's execution budget is consumed (at the rate of 1 per time unit) until the budget is exhausted, whenever either one of the following two conditions are true:
  (when these conditions are not true, the server holds its budget)

  - C1: the server is executing.
  - C2: the server deadline d is defined, the server has become idle (i.e. the last aperiodic job was terminated before d), and there is no job with deadline before d ready for execution.

Rule C2 means that:

- the budget of an idle simple sporadic server continues to decrease with time after it becomes idle and no higher priority job (with deadline < d) is ready;
- the sporadic server holds on to its budget when:
  - the server is not executing because higher priority jobs (with deadline before d) are executing.

SSS in EDF: replenishment rules

- Replenishment Rules:

  R1: initially, and at each replenishment time, the execution budget is set to e_s and the current time t_r is recorded; initially t_e and d are undefined;

  R2: when t_e is undefined, d remains also undefined; t_e is defined as follows:

  (a) at time t, when an aperiodic job arrives at an empty aperiodic queue:
  - $t_e = t_r$ if only jobs with deadlines $\leq t_r + p_s$ have executed in (t_r, t);
  - $t_e = t$ if some job with deadline $> t_r + p_s$ has executed in (t_r, t);

  deadline and next replenishment time are then set at $d = t_e + p_s$.

  (b) at the next replenishment time $t_r = d$:
  - $t_e = t_r$ if the server is backlogged;
  - t_e and d become undefined if the server is idle.

  R3: the next replenishment occurs at the next replenishment time, except under the following conditions, when it may be done sooner or later:

  (a) if the next replenishment time $t_e + p_s$ is earlier than t when the server first becomes backlogged since t_r, the budget is replenished as soon as it is exhausted (later);

  (b) the budget is replenished at the end of each idle interval of the periodic task set **T** (sooner).

SSS in EDF: replenishment rules

- Consumption and replenishment rules for SSS in EDF and in fixed-priority systems are very similar.
- e.g. if we consider the system:

  $T_1 = (3, 0.5)$ $T_2 = (4, 1.0)$ $T_3 = (19, 4.5)$ $T_{SS} = (5, 1.5)$

  $A_1\,(r = 3, e = 1)$ $A_2\,(r = 7, e = 2)$ $A_3\,(r = 15.5, e = 2)$

  the time diagram we would get when tasks T_1, T_2, T_3 and the server T_{SS} are scheduled with EDF algorithm, is very similar to the diagram in slide 48, obtained with a RM scheduler.