Topic III: Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems

謝仁偉 教授
jenwei@mail.ntust.edu.tw
國立台灣科技大學 資訊工程系
2017 Fall

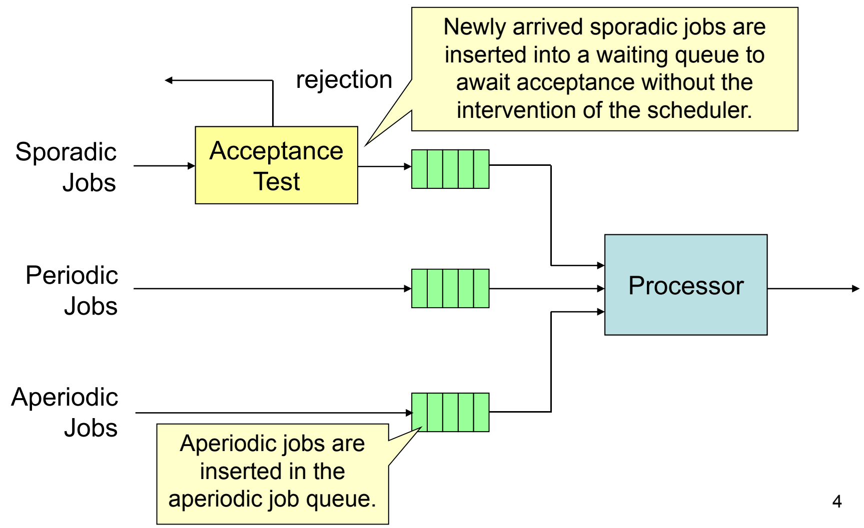
Outline

- Assumptions and Approaches
- Deferrable Servers
- Sporadic Servers
- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
- Scheduling of Sporadic Jobs

Assumptions

- Periodic tasks in the system are independent.
- Aperiodic and sporadic jobs are independent of each other and of the periodic tasks.
- Every job can be preempted at any time.
- The parameters of each sporadic job become known after it is released.
- When the periodic tasks are scheduled according to the given algorithm and there are no aperiodic and sporadic jobs, the periodic tasks meet all their deadlines.

Priority Queues Maintained by the Operating System



Aperiodic Job and Sporadic Job Scheduling Algorithms

- For **sporadic job**: If the scheduler accepts the job, it schedules the job so that the job completes in time without causing periodic tasks and previously accepted sporadic jobs to miss their deadlines.
  - *The problems are how to do the acceptance test and how to schedule the accepted sporadic jobs.*
- For **aperiodic job**: The scheduler tries to complete each aperiodic job as soon as possible.
  - *The problem is how to do so without causing periodic tasks and accepted sporadic jobs to miss deadlines.*

Correctness and Optimality (1/2)

- By a *correct schedule*, we mean one according to which periodic and accepted sporadic jobs never miss their deadlines.
- An *aperiodic job scheduling algorithm* is *optimal* if it minimizes either
  - The response time of the aperiodic job at the head of the aperiodic job queue or
  - The average response time of all the aperiodic jobs for the given queueing discipline.

Correctness and Optimality (2/2)

- An *algorithm for (accepting and) scheduling sporadic jobs* is *optimal* if it accepts each sporadic job newly offered to the system and schedules the job to complete in time if and only if the new job can be correctly scheduled to complete in time by some means.

Alternative Approaches

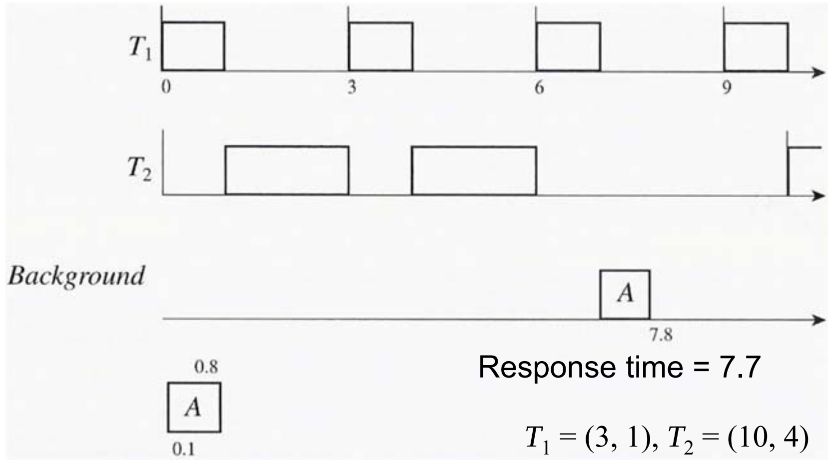
- All the algorithms described later in class attempt to provide improved performance over the three commonly used approaches:
  - Background
  - Polled
  - Interrupt-driven executions
- For the sake of simplicity and clarity, we ignore sporadic jobs for now.

Background Execution

- Aperiodic jobs are scheduled and executed only at times when there is no periodic or sporadic job ready for execution.
- Advantages:
  - This method always produces correct schedules.
  - It is simple to implement.
- Disadvantage:
  - The execution of aperiodic jobs may be delayed and their response times prolonged unnecessarily.

Example of Background Execution

The tasks are scheduled rate monotonically.



Interrupt-Driven Execution

- An obvious way to make the response times of aperiodic jobs as short as possible.
- Whenever an aperiodic job arrives, the execution of periodic tasks are interrupted, and aperiodic job is executed.
- Advantage:
  - Aperiodic job has the shortest possible response time.
- Disadvantage:
  - If aperiodic jobs always execute as soon as possible, periodic tasks may miss some deadlines.

Slack Stealing Algorithm

- Algorithms that make use of the available slack times of periodic and sporadic jobs to complete aperiodic jobs early are called slack-stealing algorithms.
- When slack-stealing can be done with an acceptable overhead, it realizes the advantage of interrupt-driven execution without its disadvantage.

Polled Execution (1/2)

- A **poller** or **polling server** (p_s, e_s) is a periodic task: p_s is its polling period, and e_s is its execution time.
- The poller is ready for execution periodically at integer multiples of p_s and is scheduled together with the periodic tasks in the system according to the given priority-driven algorithm.
- The poller suspends its execution or is suspended by the scheduler either when it has executed for e_s units of time in the period or when the aperiodic job queue becomes empty.

Polled Execution (2/2)

- If at the beginning of a polling period the poller finds the aperiodic job queue empty, it suspends immediately. It will not be ready for execution and able to examine the queue again until the next polling period.

Terms and Jargon (1/2)

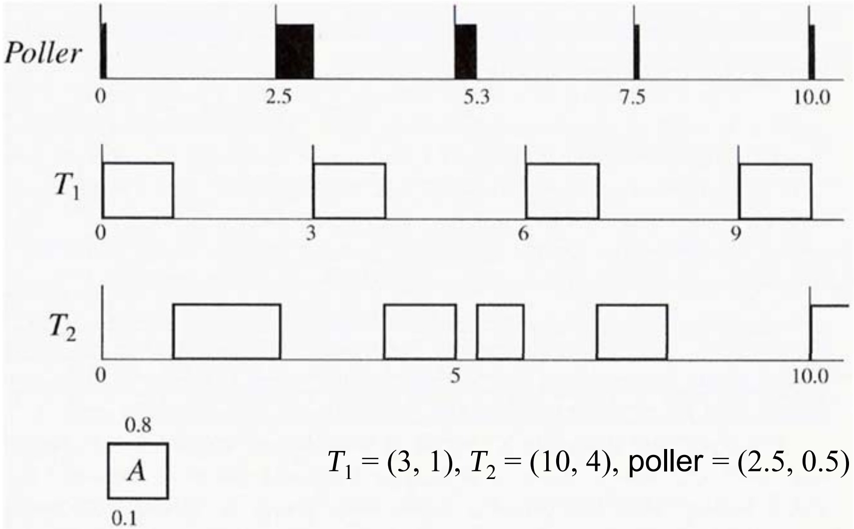
- **Periodic server**: a task that behaves more or less like a periodic task and is created for the purpose of executing aperiodic jobs.
- A periodic server (p_s, e_s) is defined partially by its period p_s and execution time e_s.
  - The server NEVER executes for more than e_s units of time in any time interval of length p_s.
  - The parameter e_s is called the **execution budget** of the server.
  - The ratio $u_s = e_s/p_s$ is the **size** of the server.
  - A poller (p_s, e_s) is a kind of periodic server.

Terms and Jargon (2/2)

  - At the beginning of each period, the budget of the poller is set to e_s. We say that its budget is **replenished** (by e_s units) and call a time instant when the server budget is replenished a **replenishment time**.
- The periodic server is **backlogged** whenever the aperiodic job queue is nonempty.
- The server is **idle** when the queue is empty.
- The server is **eligible** for execution only when it is backlogged and has budget.
- The server budget becomes **exhausted** when the budget becomes zero.

Example of Poller

The tasks are scheduled rate monotonically.



Shortcoming of the Poller

- An aperiodic job that arrives after the aperiodic job queue is examined and found empty must wait for the poller to return a polling period later.
- **Bandwidth-preserving server** can preserve the execution budget when it finds an empty queue and executes later in the period if any aperiodic job arrives. It may be able to shorten the response times of some aperiodic jobs.

Bandwidth-Preserving Servers

- Bandwidth-preserving servers are periodic servers.
- Each type of server is defined by a set of
  - **Consumption Rules**: Give the conditions under which its execution budget is preserved and consumed.
  - **Replenishment Rules**: Specify when and by how much the budget is replenished.

Assumptions

- A backlogged bandwidth-preserving server is ready for execution when it has budget.
  - The scheduler keeps track of the consumption of the server budget and suspends the server when the server budget is exhausted or the server becomes idle.
  - The scheduler moves the server back to the ready queue once it replenishes the server budget if the server still backlogged at the time.

Three Types of Bandwidth-Preserving Servers

- Deferrable Servers
- Sporadic Servers
- Constant Utilization/Total Bandwidth Servers and Weighted Fair-Queueing Servers

Outline

- Assumptions and Approaches
- *Deferrable Servers*
- Sporadic Servers
- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
- Scheduling of Sporadic Jobs

Deferrable Servers

- A deferrable server is the simplest of bandwidth-preserving servers.
  - Like a poller, the execution budget of a deferrable server with period p_s and execution budget e_s is replenished periodically with period p_s.
  - Unlike a poller, when a deferrable server finds no aperiodic job ready for execution, it preserves its budget.

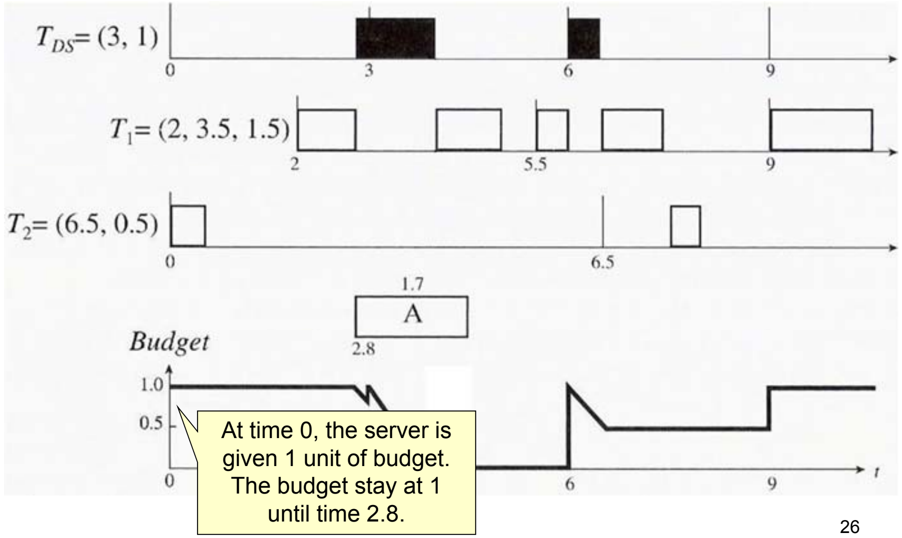
Outline

- Deferrable Servers
  1. ***Operations of Deferrable Servers***
  2. Schedulability of Fixed-Priority Systems Containing Deferrable Server(s)
  3. Schedulability of Deadline-Driven Systems in the Presence of Deferrable Server

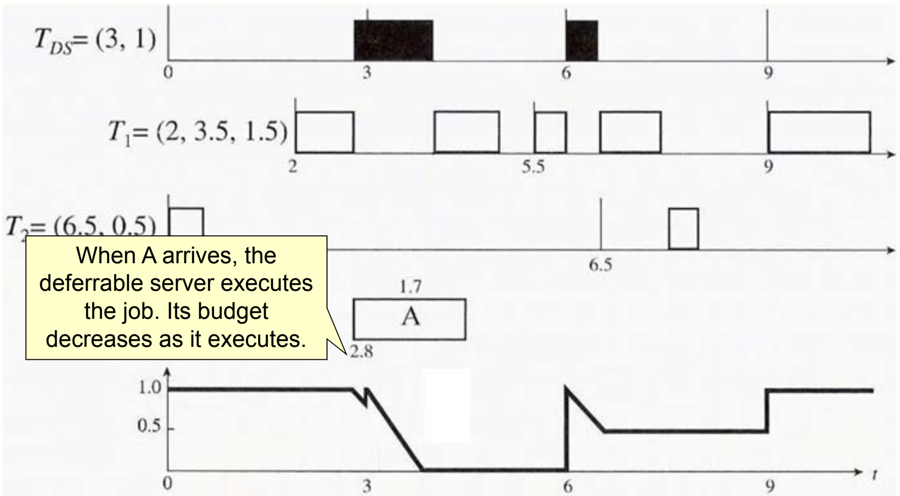
Consumption and Replenishment Rules

- Consumption Rule:
  – The execution budget of the server is consumed at the rate of one per unit time whenever the server executes.
- Replenishment Rule:
  – The execution budget of the server is $\boxed{\text{set to } e_s}$ at time instant kp_s, for $k = 0, 1, 2, \ldots$.
- We note that the server is not allowed to cumulate its budget form period to period.
  – Any budget held by the server immediately before each replenishment time is lost.

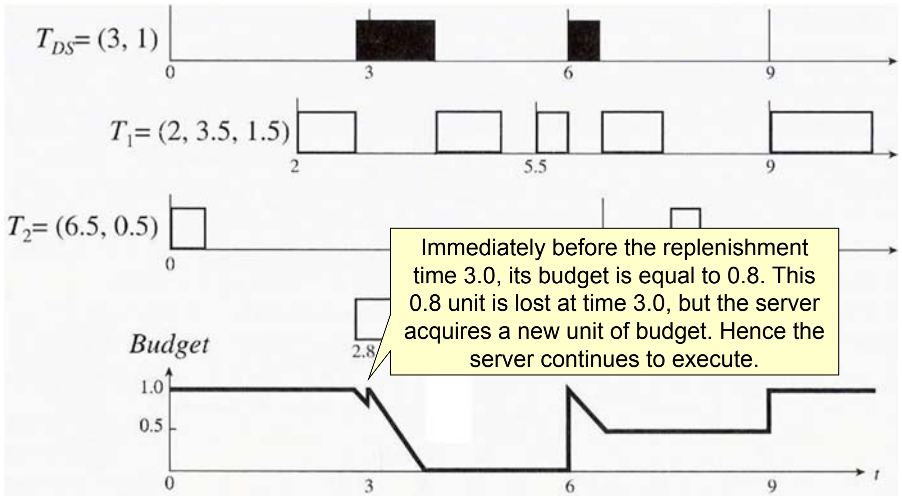
Example of Deferrable Server (RM)



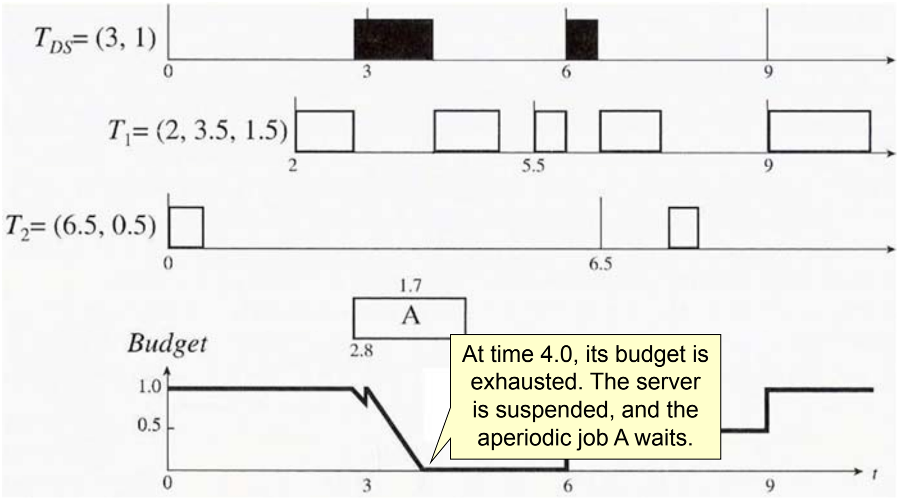
Example of Deferrable Server (RM)



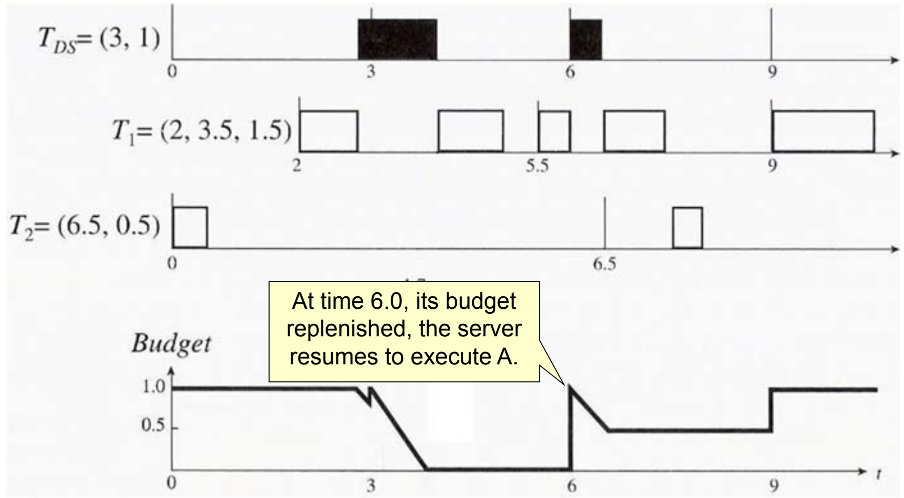
Example of Deferrable Server (RM)



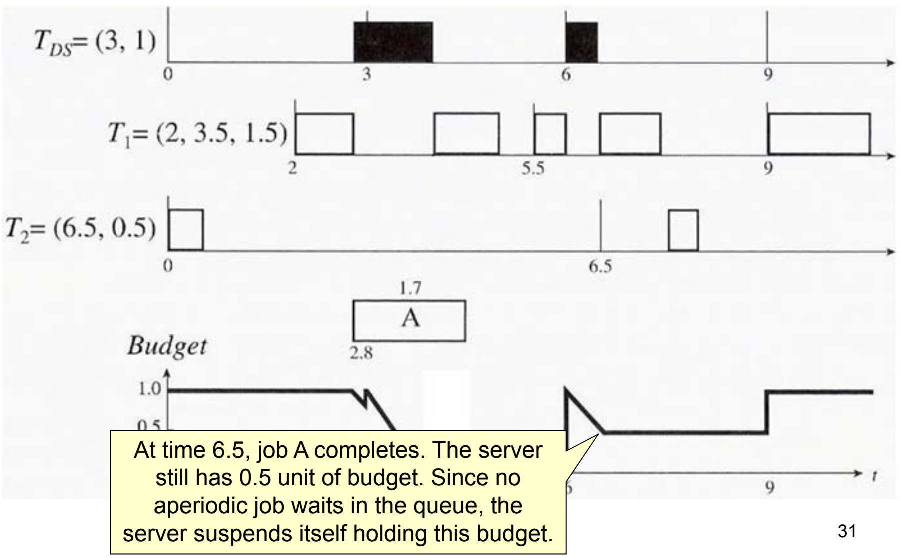
Example of Deferrable Server (RM)



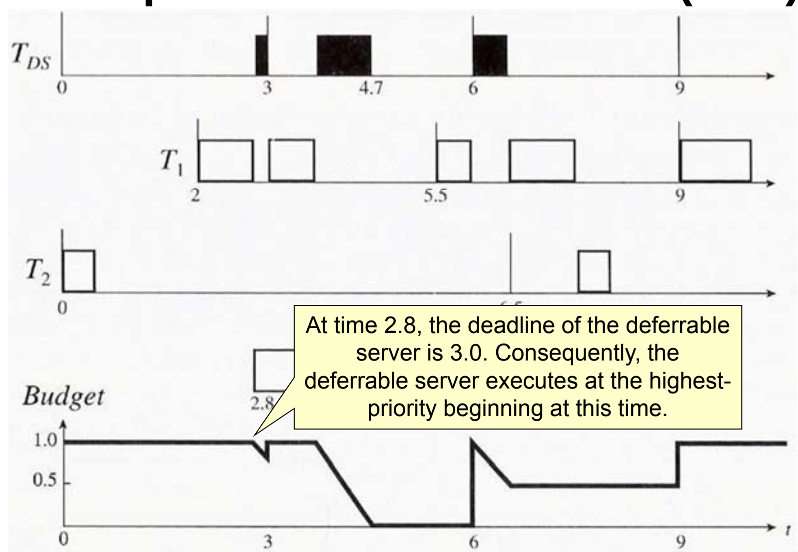
Example of Deferrable Server (RM)



Example of Deferrable Server (RM)

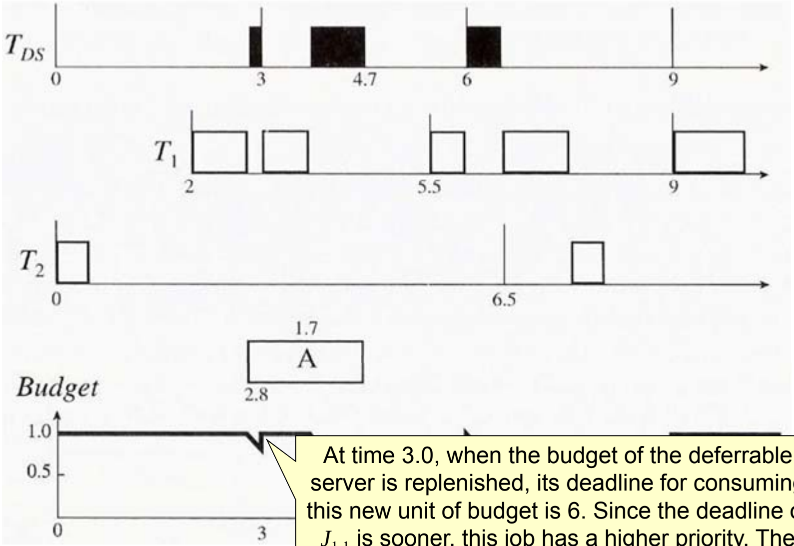


Example of Deferrable Server (EDF)



At any time, the deadline of the server is equal to the next replenishment time

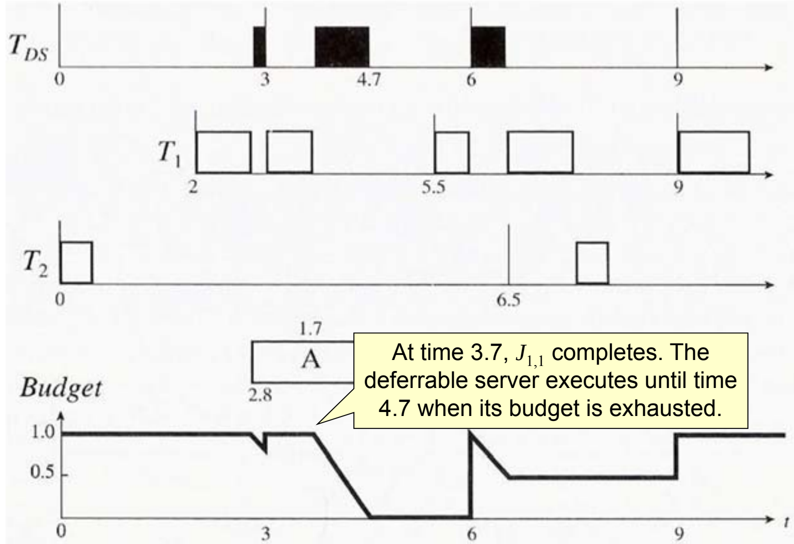
Example of Deferrable Server (EDF)



At time 3.0, when the budget of the deferrable server is replenished, its deadline for consuming this new unit of budget is 6. Since the deadline of $J_{1,1}$ is sooner, this job has a higher priority. The deferrable server is preempted

At any time, the deadline of tl … ne

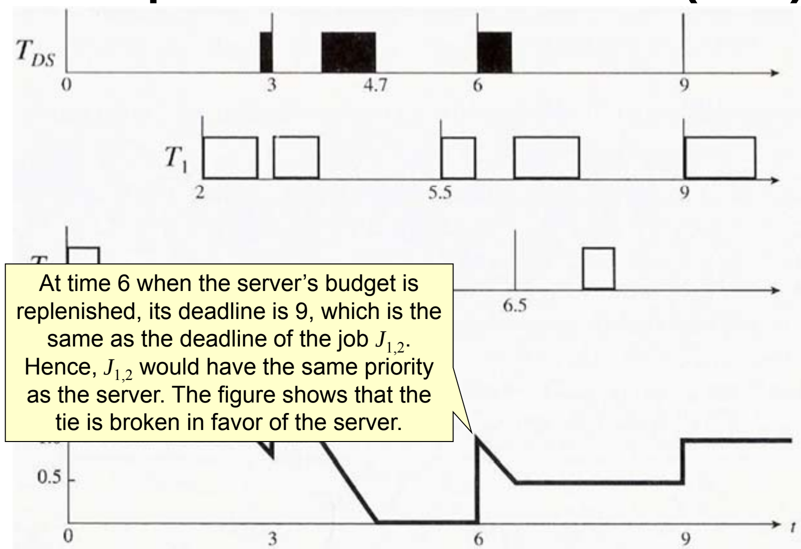
Example of Deferrable Server (EDF)



At time 3.7, $J_{1,1}$ completes. The deferrable server executes until time 4.7 when its budget is exhausted.

At any time, the deadline of the server is equal to the next replenishment time

Example of Deferrable Server (EDF)



At time 6 when the server's budget is replenished, its deadline is 9, which is the same as the deadline of the job $J_{1,2}$. Hence, $J_{1,2}$ would have the same priority as the server. The figure shows that the tie is broken in favor of the server.

At any time, the deadline of the server is equal to the next replenishment time

Deferrable Server + Background Server

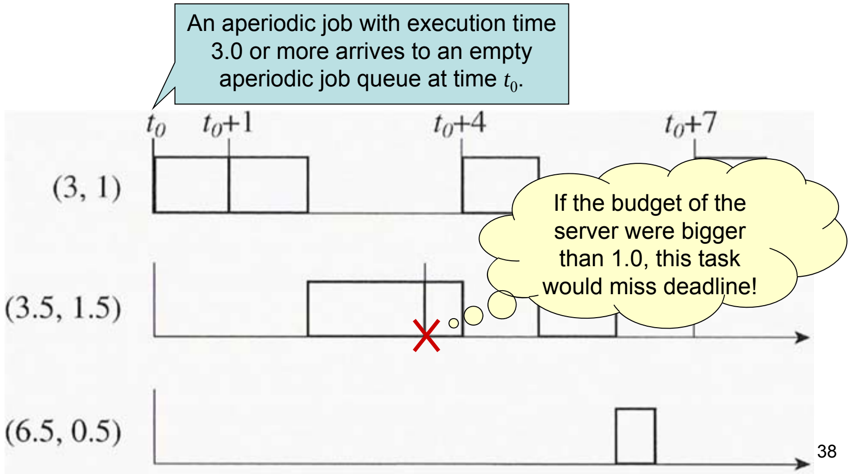
- The responsiveness of the system can be further improved if we combine the use of a deferrable server with background execution.
  – The background server is scheduled whenever the budget of the deferrable server has been exhausted and none of the periodic tasks is ready for execution.
  – When the background server is scheduled, it also execute the aperiodic job at the head of the aperiodic job queue.
  – Job A would be executed from time 4.7 and completed by time 5.2, rather than 6.5 in the previous example.

Outline

- Deferrable Servers
  1. Operations of Deferrable Servers
  2. ***Schedulability of Fixed-Priority Systems Containing Deferrable Server(s)***
  3. Schedulability of Deadline-Driven Systems in the Presence of Deferrable Server

The Factor Limiting the Budget of a Deferrable Server

- Background server vs. Increasing budget



Critical Instant (with Deferrable Servers)

- **Lemma 1.** In a fixed-priority system in which the relative deadline of every independent, preemptable periodic task is no greater than its period and there is a deferrable server (p_s, e_s) with the highest priority among all tasks, a critical instant of every periodic task T_i occurs at time t_0 when all the following are true.

Critical Instant (with Deferrable Servers)

1. One of its job $J_{i,c}$ is release at t_0.
2. A job in every higher-priority task is released at the same time.
3. 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.
4. The next replenishment time of the server is $t_0 + e_s$.

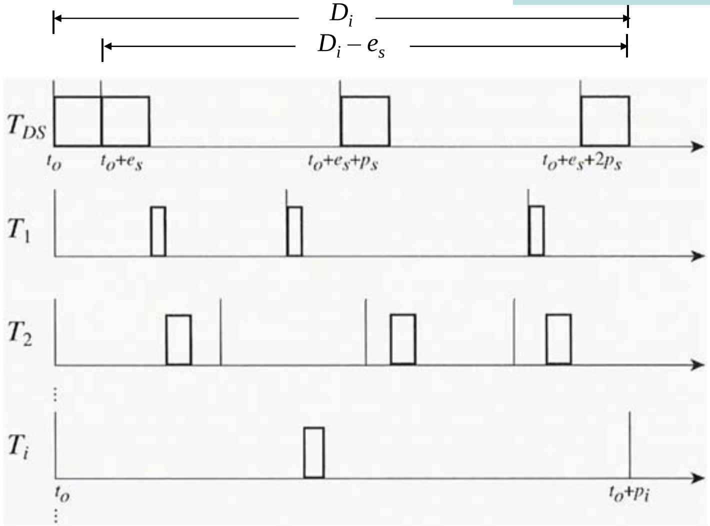
➡ The demand for processor time by each of the higher-priority tasks in its feasible interval $(r_{i,c}, r_{i,c} + D_i]$ is the largest when (1) and (2) are true.

➡ When (3) a… the amount of processor time consumed by …able server in the feasible interval is the largest.

What is the amount?

A Fixed-Priority Schedule After A Critical Instant

$e_s + \lceil (D_i - e_s)/p_s \rceil \times e_s$



Time-Demand Analysis Method (1/2)

- To determine whether the response time of T_i ever exceeds its relative deadline D_i in the case of $D_k \le p_k$ for all $k = 1, 2, \ldots, i$, we check whether $w_i(t) \le t$ is satisfied at any of the values of t that are less than or equal to D_i.

$$w_i(t) = e_i + b_i + e_s + \left\lceil \frac{t - e_s}{p_s} \right\rceil \times e_s + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil \times e_k \quad \text{for } 0 < t \le p_i$$

when the deferrable server (p_s, e_s) has the highest priority.

Time-Demand Analysis Method (2/2)

- We only need to check at values of t that are
  - Integer multiple of p_k for $k = 1, 2, \ldots, i-1$ and at D_i
  - The replenishment times of the server at or before the deadline of $J_{i,c}$ (i.e., at e_s, $e_s + p_s$, $e_s + 2p_s$, …)
- It is also easy to modify the time-demand analysis method for periodic tasks whose response times are longer than the respective periods to account for the effect of a deferrable server.

$$w_{i,j}(t) = je_i + b_i + e_s + \left\lceil \frac{t - e_s}{p_s} \right\rceil \times e_s + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil \times e_k \quad \text{for } (j-1)p_i < t \le w_{i,j}(t)$$

when the deferrable server (p_s, e_s) has the highest priority.

Schedulable Utilization (1/4)

- There is no known schedulable utilization that assures the schedulability of a fixed-priority system in which a deferrable server is scheduled at an arbitrary priority.
- The only exception is the special case when the period p_s of the deferrable server is shorter than the periods of all periodic tasks and the system is scheduled rate-monotonically.

Schedulable Utilization (2/4)

- **Theorem 2.** Consider a system of n independent, preemptable periodic tasks whose periods satisfy the inequalities $p_s < p_1 < p_2 < \ldots < p_n < 2p_s$ and $p_n > p_s + e_s$ and whose relative deadline are equal to their respective periods. This system is schedulable rate-monotonically with a deferrable server (p_s, e_s) if their total utilization is less than or equal to

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

where u_s is the utilization e_s/p_s of the server.

Schedulable Utilization (3/4)

- When the server's period is arbitrary, we can use the schedulable utilization $U_{RM}(n) = n(2^{1/n} - 1)$ to determine whether each periodic task T_i is schedulable if the periodic tasks and the server are scheduled rate-monotonically and the relative deadlines of the periodic tasks are equal to their respective periods.
- The server behaves just like a periodic task (p_s, e_s), except that the server may execute for an additional e_s units of time in the feasible interval of the job.

Schedulable Utilization (4/4)

- We can treat these e_s units of time as additional "blocking time" of the task T_i: T_i is surely schedulable if

$$U_i + u_s + \frac{e_s + b_i}{p_i} \le U_{RM}(i+1)$$

where U_i is the total utilization of the i highest-priority tasks in the system.

Deferrable Server with Arbitrary Fixed Priority (1/4)

- Since any budget of a deferrable server that remains unconsumed at the end of each server period is lost, when the server is not scheduled at the highest priority, the maximum amount of processor time it can consume (and hence is demanded) depends not only on the release times of all periodic jobs relative to replenishment times of the server, but also on the execution times of all the tasks.

Deferrable Server with Arbitrary Fixed Priority (2/4)

- In a system where the deferrable server is scheduled at an arbitrary priority, the time-demand function of a task T_i with a lower-priority than the server is bounded from above by the expression of $w_i(t)$.
- Using this upper bound, we make the time-demand analysis method a **sufficient schedulability test** for any fixed-priority system containing one deferrable server.

Deferrable Server with Arbitrary Fixed Priority (3/4)

- We may want to use different servers to execute different aperiodic tasks.
- By adjusting the parameters and priorities of the servers, the response times of jobs in an aperiodic task may be improved at the expense of the response times of jobs in other aperiodic tasks.

Deferrable Server with Arbitrary Fixed Priority (4/4)

- The time-demand function $w_i(t)$ of a periodic task T_i with a lower priority than m deferrable servers, each of which has period $p_{s,k}$ and execution budget $e_{s,k}$, is given by

$$w_i(t) \le e_i + b_i + \sum_{k=1}^{m} \left(1 + \left\lceil \frac{t - e_{s,k}}{p_{s,k}} \right\rceil \right) \times e_{s,k} + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil \times e_k \quad \text{for } 0 < t \le p_i$$

Outline

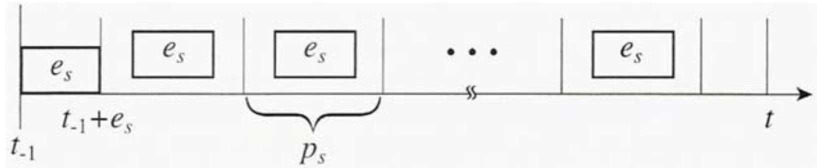
- Deerrable Servers
  1. Operations of Deferrable Servers
  2. Schedulability of Fixed-Priority Systems Containing Deferrable Server(s)
  3. ***Schedulability of Deadline-Driven Systems in the Presence of Deferrable Server***

Schedulable Utilization (1/5)

- Let t be the deadline of a job $J_{i,c}$ in some periodic task T_i and t_{-1} ($< t$) be the latest time instant at which the processor is either idle or is executing a job whose deadline is after t.
- We observe that if $J_{i,c}$ does not complete by t, the total amount $w_{DS}(t - t_{-1})$ of processor time consumed by the deferrable server in the interval $(t_{-1}, t]$ is at most equal to

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

Schedulable Utilization (2/5)



1. At time t_{-1}, its budget is equal to e_s and the deadline for consuming the budget is (and hence the budget is to be replenished at) $t_{-1} + e_s$.
2. One or more aperiodic jobs arrive at t_{-1}, and the aperiodic job queue is never empty hereafter, at least until t.
3. The server's deadline $t_{-1} + e_s$ is earlier than the deadlines of all the periodic jobs that are ready for execution in the interval $(t_{-1}, t_{-1} + e_s]$.

Schedulable Utilization (3/5)

- Since $\lfloor x \rfloor \leq x$ for all $x \geq 0$ and $u_s = e_s/p_s$, $w_{DS}(t - t_{-1})$ must satisfy the following inequality if the job $J_{i,c}$ does not complete by its deadline at t:

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

Schedulable Utilization (4/5)

- **Theorem 3.** A periodic task T_i in a system of n independent, preemptive periodic tasks is scheduled with a deferrable server with period p_s, execution budget 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 suppose that a job misses its deadline at t and consider the interval $(t_{-1}, t]$.

Schedulable Utilization (5/5)

- The total amount of processor time used by each periodic task T_k in the time interval is bounded from above by $\boxed{e_k(t - t_{-1})/p_k.}$
- The fact that a job misses its deadline at t tell us

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

- Dividing both sides of this inequality by $t - t_{-1}$, we get the equation for the case of $D_k \geq p_k$.

Example

- Consider a system of three periodic tasks $T_1 = (3, 0.6)$, $T_2 = (5.0, 0.5)$, and $T_3 = (7, 1.4)$. Suppose they are scheduled with a deferrable server whose period is 4 and execution time is 0.8. Are all three tasks in the system schedulable?
- According to Theorem 3, the left-hand side of the equation for the three tasks are 0.913, 0.828, and 0.792, respectively.
- Hence, all three tasks are schedulable.

Remark

- A deferrable server in a deadline-driven system 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. (We treat it as the blocking time suffered by the job.)
- In a fixed-priority system, the blocking time can be as large as e_s.
- In a deadline-driven system, the blocking time is only $(p_s - e_s)\, u_s$. (Theorem 3)

The latter is always less than the former, why?

Corollary

- A periodic task T_i in a system of n independent, preemptive periodic tasks is scheduled according to the EDF algorithm, together with m deferrable servers, each of which has period $p_{s,k}$, execution budget $e_{s,k}$, and utilization $u_{s,k}$, if

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + \sum_{k=1}^{m} u_{s,k}\left(1 + \frac{p_{s,k} - e_{s,k}}{D_i}\right) \leq 1$$

Outline

- Assumptions and Approaches
- Deferrable Servers
- *Sporadic Servers*
- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
- Scheduling of Sporadic Jobs

Sporadic Servers (1/3)

- A deferrable server may delay lower-priority tasks for more time than a period task with the same period and execution time.
- Sporadic servers are designed to improve over a deferrable server in this respect.
  - The consumption and replenishment rules of sporadic server algorithms ensure that each sporadic server with period p_s and budget e_s never demands more processor time than the periodic task (p_s, e_s) in any time interval.

Sporadic Servers (2/3)

- We can treat the sporadic server exactly like the periodic task (p_s, e_s) when we check for the schedulability of the system.
- 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.

Sporadic Servers (3/3)

- Different kinds of sporadic servers differ in their consumption and replenishment rules.
- More complicated rules allow a server to
  - Preserve its budget for a longer time
  - Replenish the budget more aggressively
  - Execute at a higher priority in a deadline-driven system
- By using different rules, you can trade off the responsiveness of the server for the overhead in its implementation.

Outline

- Sporadic Servers
  1. ***Sporadic Server in Fixed-Priority Systems***
  2. Enhancements of Fixed-Priority Sporadic Server
  3. Simple Sporadic Servers in Deadline-Driven Systems

Assumptions and Notations (1/4)

- There is only one sporadic server in a fixed-priority system $\mathbf{T}$ of n independent, preemptable periodic tasks.
- The server has an arbitrary priority π_S. (If the server has the same priority as some periodic task, the tie is always broken in favor of the server.)
- Assume we have chosen the p_s and e_s and have validated that the periodic task (p_s, e_s) and the system $\mathbf{T}$ are schedulable according to the fixed-priority algorithm used by the system.

Assumptions and Notations (2/4)

- During an interval when the aperiodic job queue is never empty, the server behaves like the periodic task (p_s, e_s) in which some jobs may take longer than one period to complete.
- We use $\mathbf{T}_H$ to denote the subset of periodic tasks that have higher priorities than the server.
- A server busy interval is a time interval which begins when an aperiodic job arrives at an empty aperiodic job queue and ends when the queue becomes empty again.

Assumptions and Notations (3/4)

- t_r denotes the latest (actual) replenishment time.
  - The scheduler sets t_r to the current time each time it replenishes the server's execution budget.
- t_f denotes the first instant after t_r at which the server begins to execute.
- t_e denotes the latest effective replenishment time.
  - When the server first begins to execute after a replenishment, the scheduler determines the latest effective replenishment time t_e based on the history of the system and sets the next replenishment time to $t_e + p_s$. (The next replenishment time is p_s units away from t_e, as if the budget was last replenished at t_e and hence the name effective replenishment time.)

Assumptions and Notations (4/4)

- At any time t, *BEGIN* is the beginning instant of the earliest busy interval among the latest contiguous sequence of busy intervals of the higher-priority subsystem $\mathbf{T}_H$ that started before t. (Two busy intervals are contiguous if the later one begins immediately after the earlier one ends.)
- *END* is the end of the latest busy interval in the above defined sequence if this interval ends before t and equal to infinity if the interval ends after t.

Simple Sporadic Server (1/3)

- *Consumption Rules of Simple Fixed-Priority Sporadic Server*: At any time t after t_r, the server's execution budget is consumed at the rate of 1 per unit time until the budget is exhausted when either one of the following two conditions is 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 and *END* < t.

The server consumes its budget at any time t if it has executed since t_r but at t, it is suspended and the higher-priority subsystem $\mathbf{T}_H$ is idle.

Simple Sporadic Server (1/3)

- *Consumption Rules of Simple Fi... Priori...* *Sporadic S...* execu... per ... eithe... ...ue. When thes... ..., the server holds its ...get.
  - **C1**: The server is executing.
  - **C2**: The server has executed since t_r and *END* < t.

Once the server has executed since a replenishment, it consumes its budget except during the time interval(s) when it is preempted.

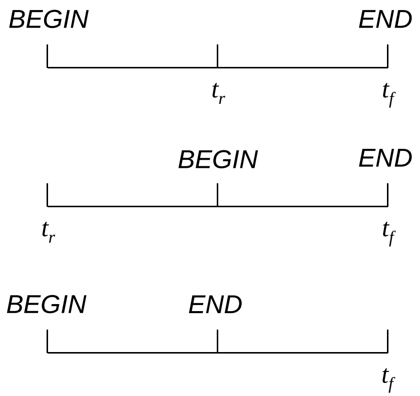
Simple Sporadic Server (2/3)

- *Replenishment Rules of Simple Fixed-Priority Sporadic Server*:
  - **R1**: Initially when the system begins execution and each time when the budget is replenished, the execution budget = e_s, and t_r = the current time.
  - **R2**: At time t_f, if *END* $= t_f$, $t_e = \max(t_r,$ BEGIN$)$. If END $< t_f$, $t_e = t_f$. The next replenishment time is set at $t_e + p_s$.
  - **R3**: The next replenishment occurs at the next replenishment time, except under the following conditions. Under these conditions, replenishment is done at times stated below.

Illustration of Rule 2

– **R2**: At time t_f, if *END* $= t_f$, $t_e = \max(t_r, \text{BEGIN})$. If END < t_f, $t_e = t_f$. The next replenishment time is set at $t_e + p_s$.

How about END > t_f?



$t_e = t_r$

$t_e = BEGIN$

$t_e = t_f$

Simple Sporadic Server (3/3)

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.

b) If the system $\mathbf{T}$ becomes idle before the next replenishment time $t_e + p_s$ and becomes busy again at t_b, the budget is replenished at $\min(t_e + p_s, t_b)$.

Simple Sporadic Server (3/3)

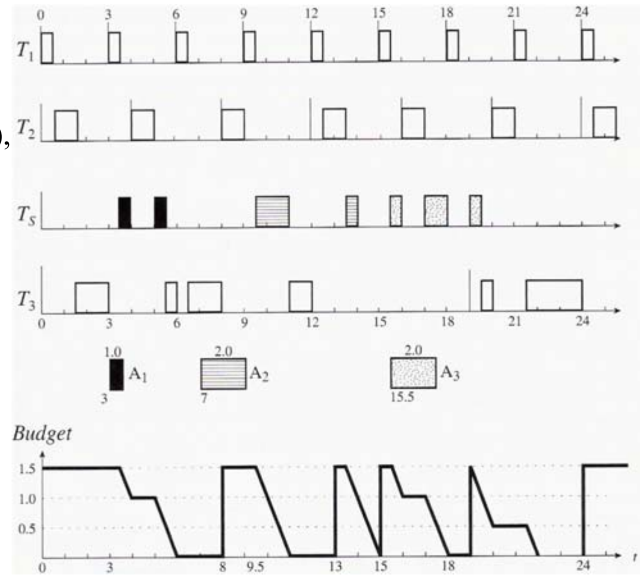
a) If the next replenishment time $t_e + p_s$ is earlier than t_f, the budget is replenished as soon as it is exhausted.

b) I… re… a… p_s, t_b).

When this rule applies, the server emulates the situation when a job in T_s takes more than one server period to complete.

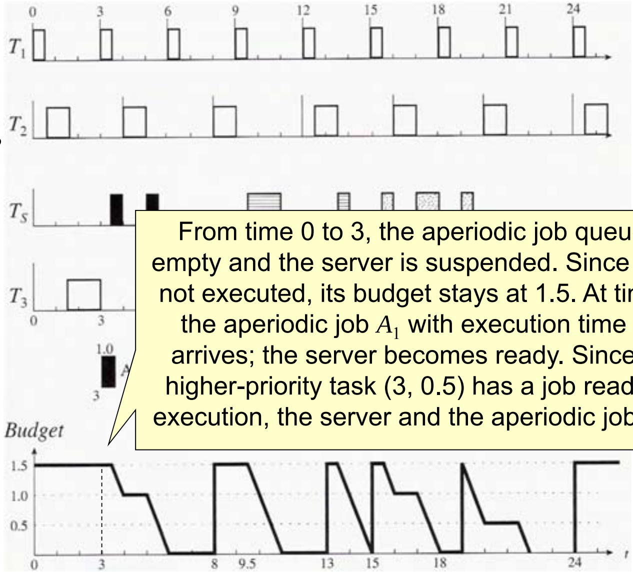
Illustrative Example with RM (1/10)

$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



Illustrative Example with RM (1/10)

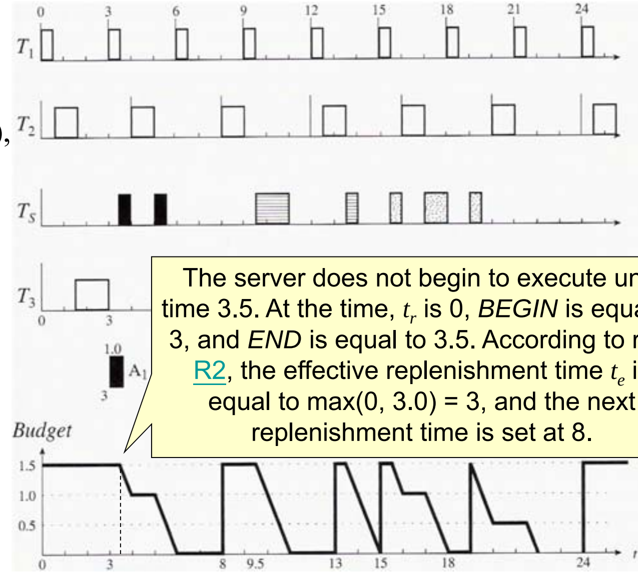
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



From time 0 to 3, the aperiodic job queue is empty and the server is suspended. Since it has not executed, its budget stays at 1.5. At time 3, the aperiodic job A_1 with execution time 1.0 arrives; the server becomes ready. Since the higher-priority task (3, 0.5) has a job ready for execution, the server and the aperiodic job wait.

Illustrative Example with RM (2/10)

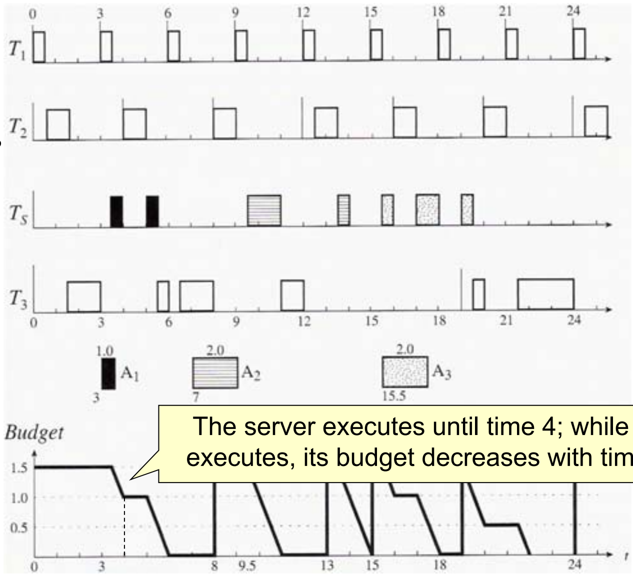
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



The server does not begin to execute until time 3.5. At the time, t_r is 0, *BEGIN* is equal to 3, and *END* is equal to 3.5. According to rule R2, the effective replenishment time t_e is equal to max(0, 3.0) = 3, and the next replenishment time is set at 8.

Illustrative Example with RM (3/10)

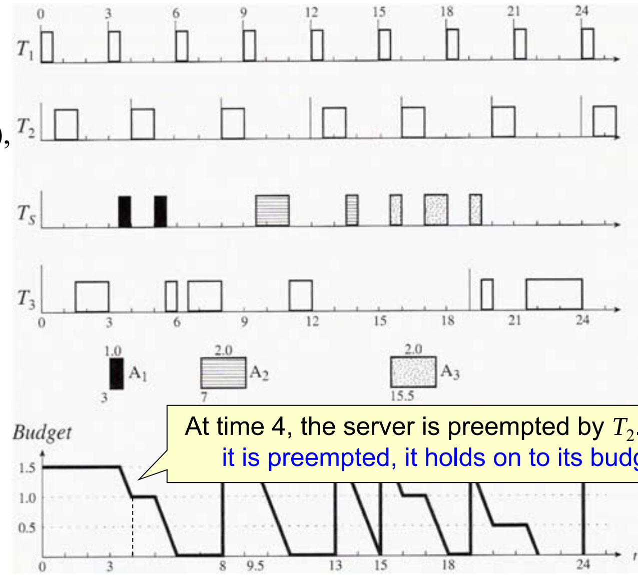
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



The server executes until time 4; while it executes, its budget decreases with time.

Illustrative Example with RM (4/10)

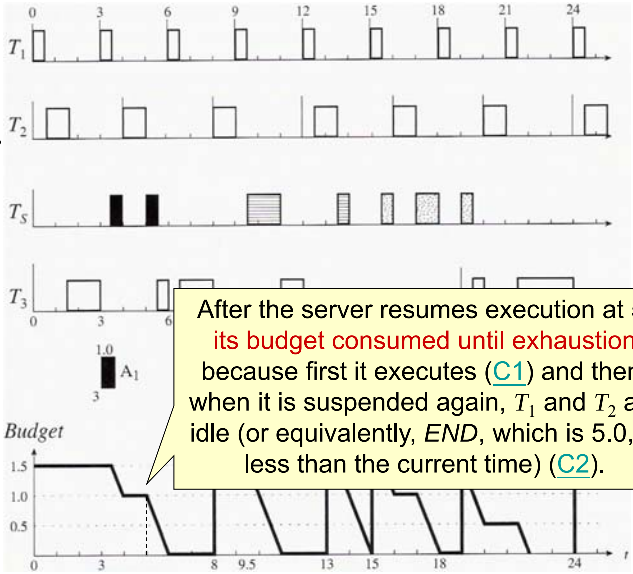
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



At time 4, the server is preempted by T_2. While it is preempted, it holds on to its budget.

Illustrative Example with RM (5/10)

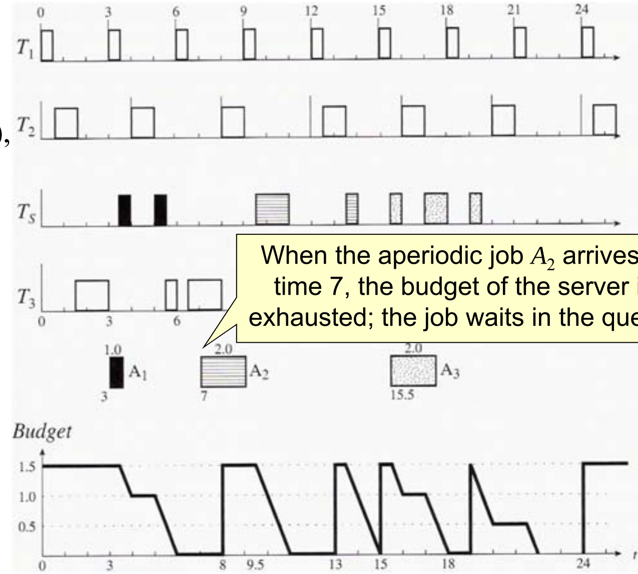
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



After the server resumes execution at 5, its budget consumed until exhaustion because first it executes (C1) and then, when it is suspended again, T_1 and T_2 are idle (or equivalently, *END*, which is 5.0, is less than the current time) (C2).

Illustrative Example with RM (6/10)

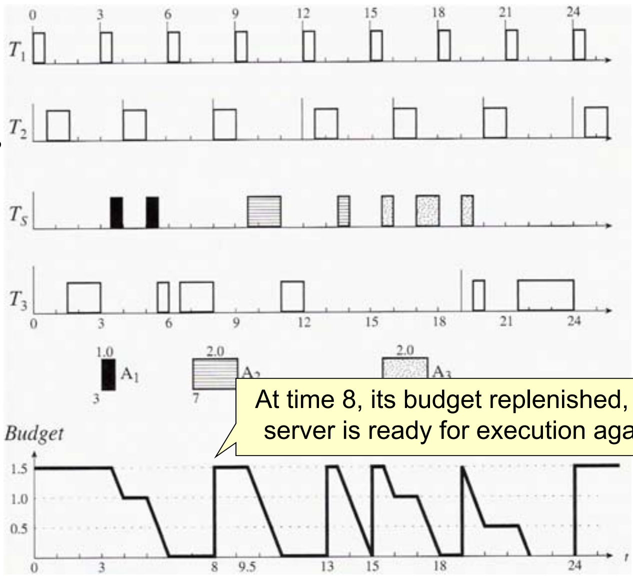
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



When the aperiodic job A_2 arrives at time 7, the budget of the server is exhausted; the job waits in the queue.

Illustrative Example with RM (7/10)

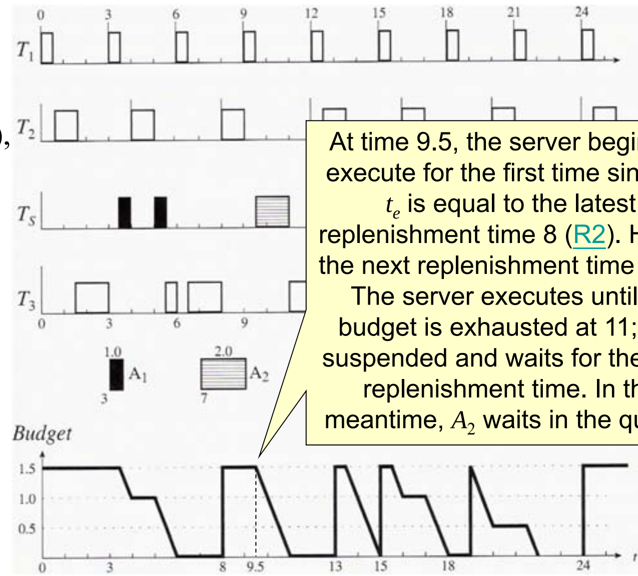
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



At time 8, its budget replenished, the server is ready for execution again.

Illustrative Example with RM (8/10)

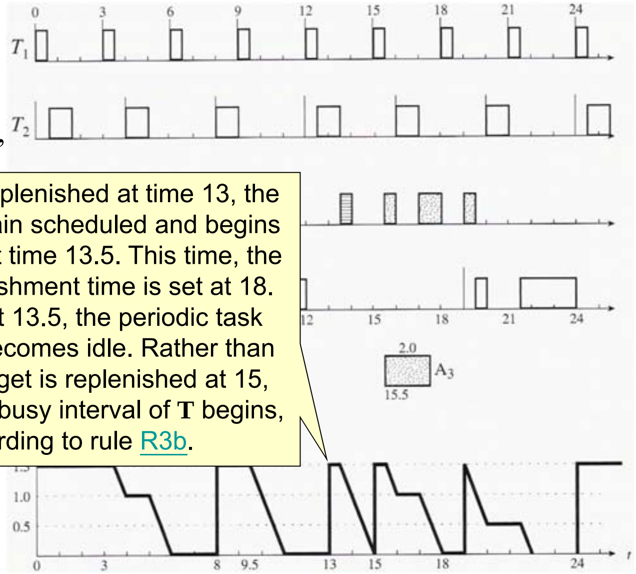
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



At time 9.5, the server begins to execute for the first time since 8. t_e is equal to the latest replenishment time 8 (R2). Hence the next replenishment time is 13. The server executes until its budget is exhausted at 11; it is suspended and waits for the next replenishment time. In the meantime, A_2 waits in the queue.

Illustrative Example with RM (9/10)

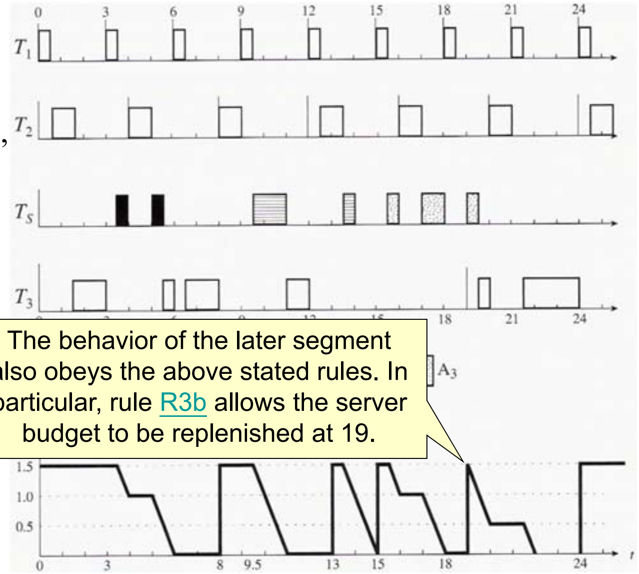
$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$



Its budget replenished at time 13, the server is again scheduled and begins to execute at time 13.5. This time, the next replenishment time is set at 18. However at 13.5, the periodic task system **T** becomes idle. Rather than 18, the budget is replenished at 15, when a new busy interval of **T** begins, according to rule R3b.

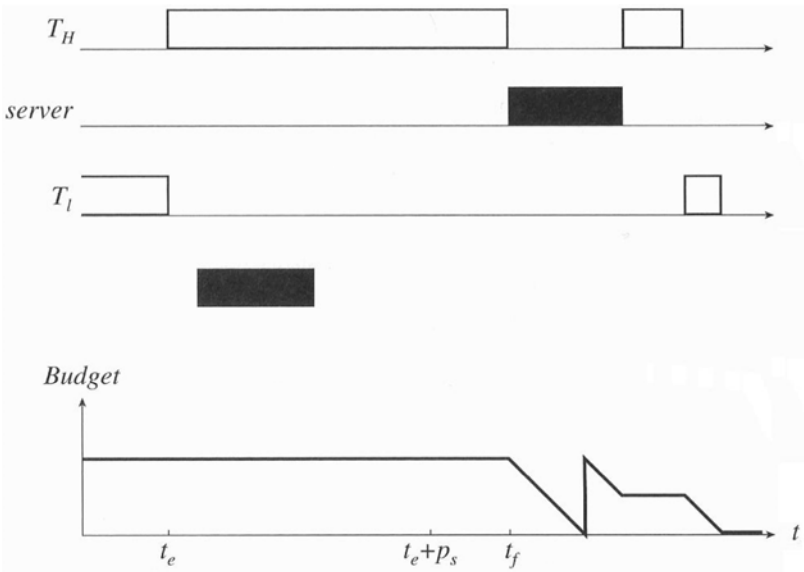
Illustrative Example with RM (10/10)

$T_1 = (3, 0.5)$,
$T_2 = (4, 1.0)$,
$T_3 = (19, 4.5)$,
$T_S = (5, 1.5)$

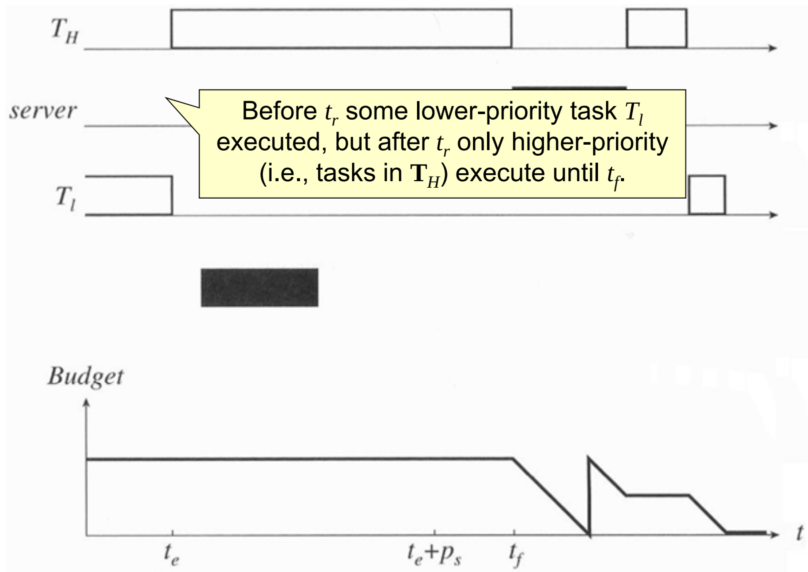


The behavior of the later segment also obeys the above stated rules. In particular, rule R3b allows the server budget to be replenished at 19.

A Situation where Rule R3a Applies (1/4)

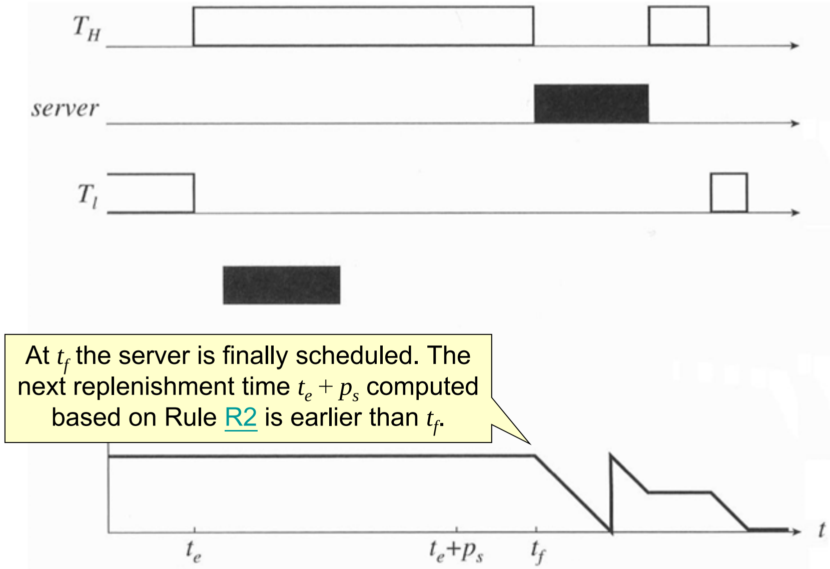


A Situation where Rule R3a Applies (2/4)



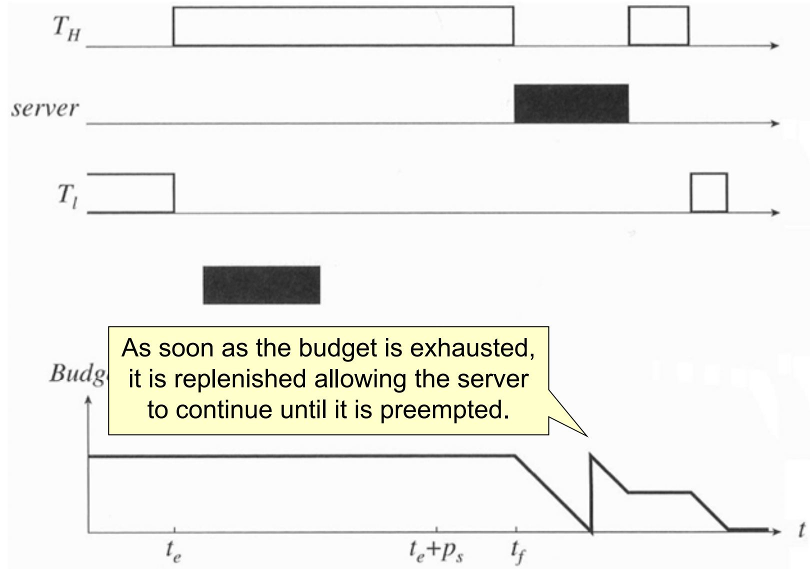
Before t_r some lower-priority task T_l executed, but after t_r only higher-priority (i.e., tasks in $\mathbf{T}_H$) execute until t_f.

A Situation where Rule R3a Applies (3/4)



At t_f the server is finally scheduled. The next replenishment time $t_e + p_s$ computed based on Rule R2 is earlier than t_f.

A Situation where Rule R3a Applies (4/4)



As soon as the budget is exhausted, it is replenished allowing the server to continue until it is preempted.

Informal Proof of Correctness (1/8)

- We shall explain why a server following the above stated rules emulates the periodic task $T_s = (p_s, e_s)$.
- The actual interrelease-times of jobs in T_s are sometimes larger than p_s, and their execution times are sometimes smaller than e_s.
- ➡ In this case, the rules and the sporadic server are correct.
- ➡ The only exception is when rule R3b is applied.

Informal Proof of Correctness (2/8)

- The consumption rule C1 ensures that each server job never executes for more time than its execution budget e_s.
- C2 applies when the server becomes idle while it still has budget.
  - The budget of an idle simple sporadic server continues to decrease with time as if server were executing.
- ➡ Because of these two rules, each sever job never executes at times when the corresponding job of the periodic task T_s does not.

Informal Proof of Correctness (3/8)

- C2 also means that the server holds on to its budget at any time t after its budget is replenished at t_r when
  - (a) Some higher-priority job is executing
    - The server waits for higher-priority tasks
    - Its budget does not decrease because the server is not scheduled independent of whether the server is ready or suspend.
  - (b) The server has not executed since t_r
    - This emulates the situation when the current job of the periodic task T_s is released late.

Informal Proof of Correctness (4/8)

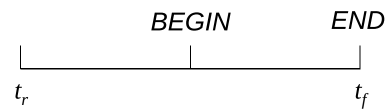
- To show the correctness of the replenishment rules R2 and R3a, we note that the next replenishment time is always set at the p_s time units after the effective release-time t_e of the current server job and the next release-time is never earlier than the next replenishment time.
- ➡ Consecutive replenishments occurs at least p_s units apart.
- ➡ Rule R2 is designed to make the effective replenishment time as soon as possible without making the server behave differently from a periodic task.

Informal Proof of Correctness (5/8)

- At time t_f when the server executes for the first time after t_r, the effective replenishment time t_e is set at t_r if higher-priority tasks have executed throughout the interval (t_r, t_f).
- ➡ This emulates a job in T_s released at t_r but has to wait for higher-priority tasks to become idle before it can execute.

Informal Proof of Correctness (6/8)

- If lower-priority tasks executed in this interval, t_e is set to the latest time instant when a lower-priority task executes.
- ➡ This emulates a job in T_s that is released at t_e and waits for higher-priority tasks to become idle and then begins execution.



t_e = *BEGIN*

Informal Proof of Correctness (7/8)

- Rule R3a applies when the current server job has to wait for more than p_s units of time before its execution can begin.
- This rule allows the budget to be replenished as soon as it is exhausted.
- When this rule applies, the server emulates the situation when a job in T_s takes more than one server period to complete.

Informal Proof of Correctness (8/8)

- When rule R3b applies, the server may behave differently from the periodic task T_s.
- This rule is applicable only when a busy interval of the periodic task system $\mathbf{T}$ ends.

  A server job is "released" at the same time as the job in $\mathbf{T}$ which begins the new busy interval.
- This condition was taken into account in the schedulability test that found $\mathbf{T}$ to be schedulable together with the periodic task T_s, and the behavior of the system in the new busy interval is independent of that in the previous busy interval.

Outline

- Sporadic Servers
  1. Sporadic Server in Fixed-Priority Systems
  2. ***Enhancements of Fixed-Priority Sporadic Server***
  3. Simple Sporadic Servers in Deadline-Driven Systems

Sporadic/Background Server (1/2)

- Rule R3b of the simple fixed-priority sporadic server is overly conservative.
- In the previous example, the schedulability of the periodic system will not be adversely affected if we replenish the server budget at 18.5 and let the server execute until 19 with its budget undiminished.
- We call a sporadic server that claims the background time a sporadic/background server; in essence, it is a combination of a sporadic server and a background server.

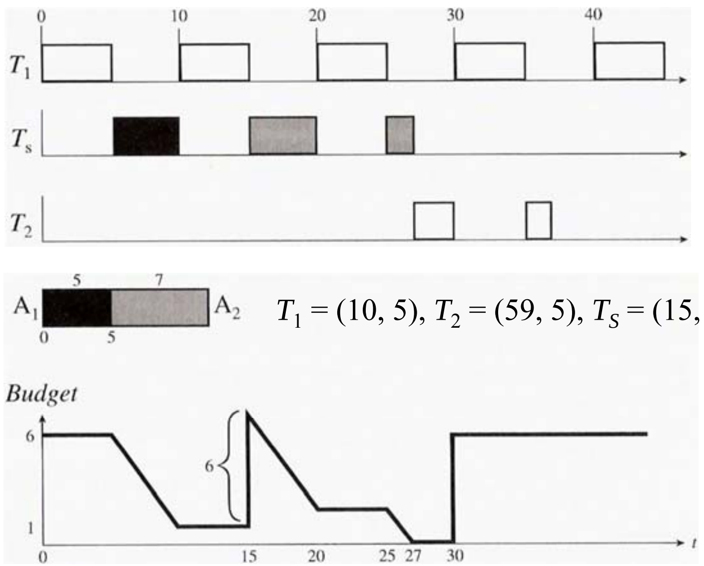
Sporadic/Background Server (2/2)

- *Consumption rules of simple sporadic/background servers* are the same as the rules of simple sporadic servers except when the periodic task system is idle. As long as the periodic task system is idle, the execution budget of the server stays at e_s.
- *Replenishment rules of simple sporadic/background servers* are the same as those of the simple sporadic servers except R3b.
  - The budget of a sporadic/background server is replenished at the beginning of each idle interval of the periodic task system.
  - t_r is set at the end of the idle interval.

Cumulative Replenishment (1/4)

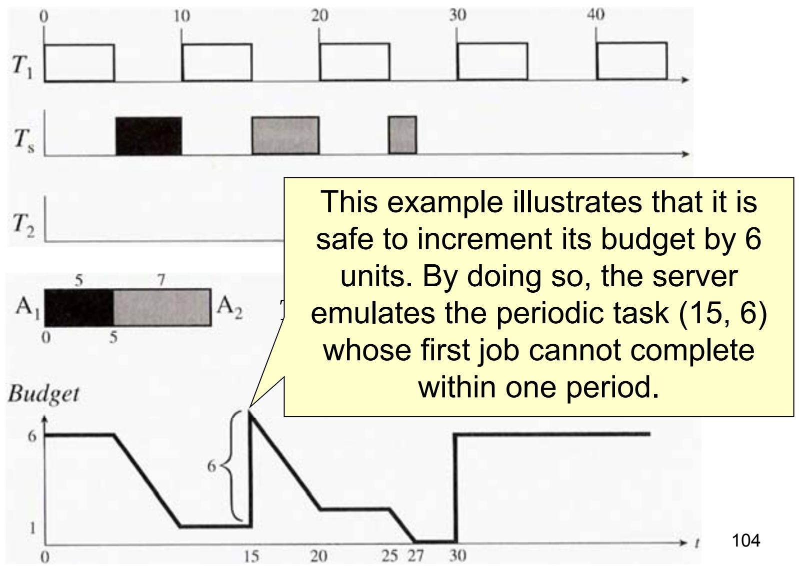
- A way to give the server more budget is to let the server keep any budget that remains unconsumed – Rather than *setting* the budget to e_s, we *increment* its budget by e_s units at each replenishment time.
- The server emulates a periodic task in which some jobs do not complete before the subsequent jobs in the task are released.

Example (1/2)

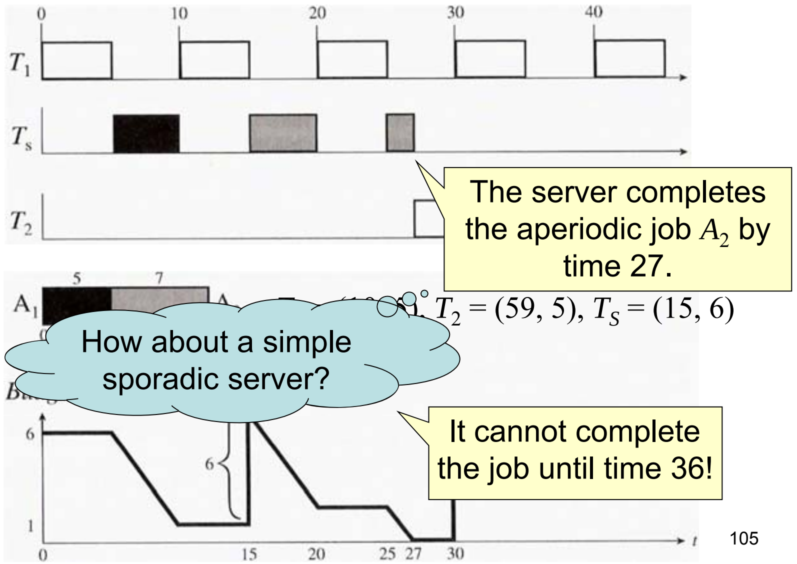


$T_1 = (10, 5)$, $T_2 = (59, 5)$, $T_S = (15, 6)$

Example (1/2)



Example (2/2)



Cumulative Replenishment (2/4)

- *Modifications of consumption rules*:
  - Rule C1 of simple sporadic server always applies.
  - It is necessary to treat the e_s-unit chunk of new budget replenished at the latest replenishment time differently from the chunk of old budget left unconsumed at the time.
  - Whenever the server has both new budget and old budget, it always consumes the old budget first; this emulates the behavior that jobs in a periodic task are executed in the FIFO order whenever more than one job of the task is ready for execution at the same time.

Cumulative Replenishment (3/4)

- It is still safe to consume the new budget according to C1 and C2.
- We can not use rule C2 to govern the consumption of the old budget.
- After t_r, the old budget should be consumed at the rate of 1 per unit time when the server is suspended and the higher-priority subsystem $\mathbf{T}_H$ is idle independent of whether the server has executed since t_r. (illustrate)

Illustration of the Rule

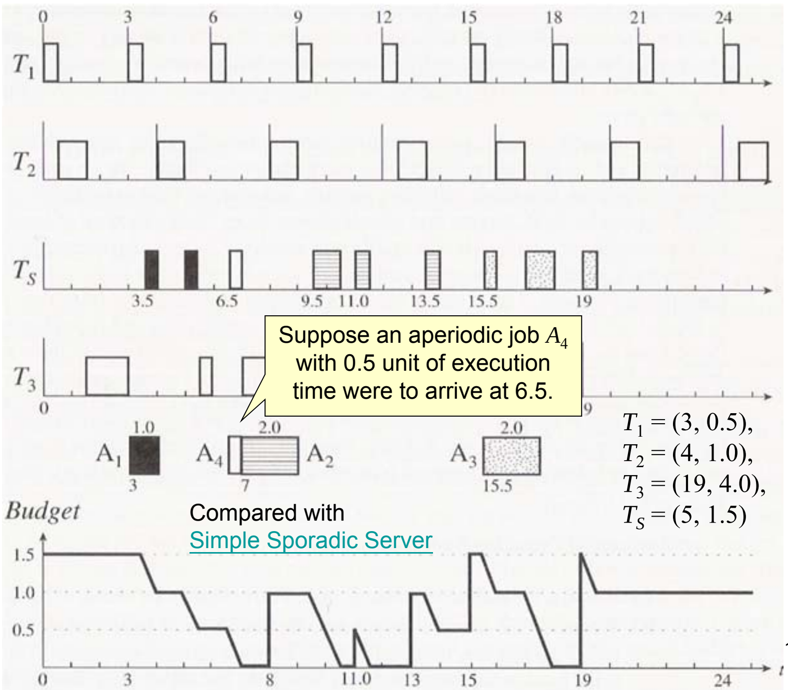
- Suppose job A_2 is released at time 17 instead.
- From time 15 to 17 while the lower-priority task (59, 5) is executing, the server holds on to its new budget replenished at time 15 according to rule C2.
- The 1-unit chunk of old budget should be consumed by time 16, leaving the server with only its new budget at 16.

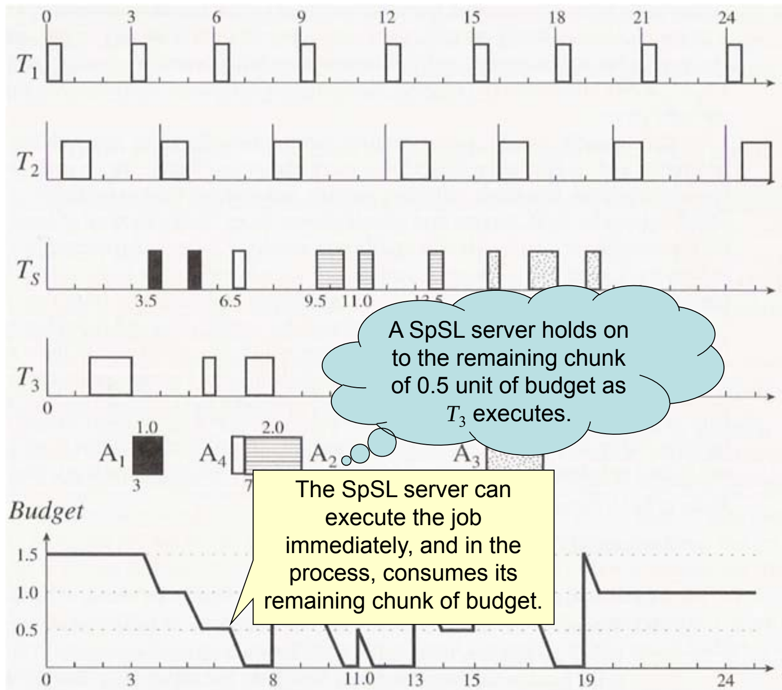
Cumulative Replenishment (4/4)

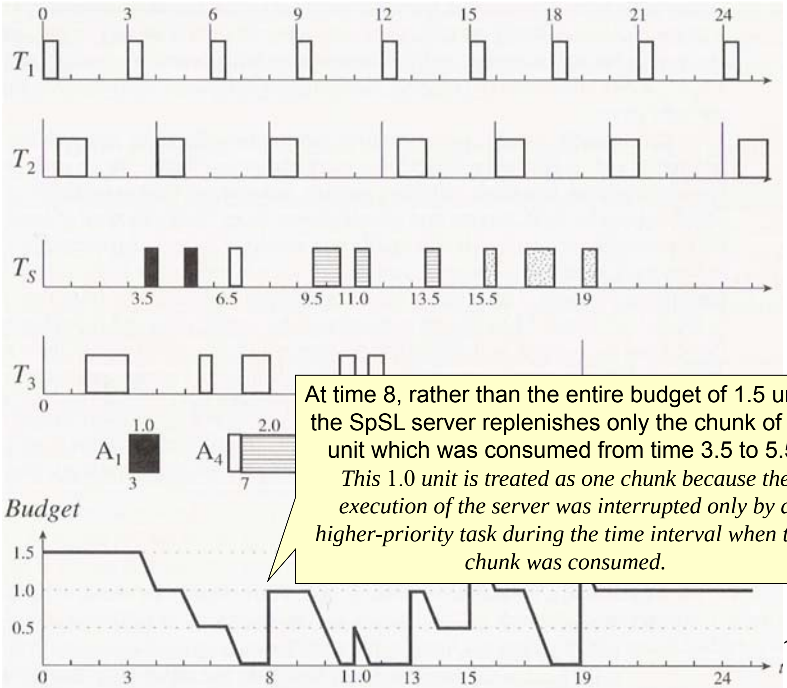
- *Modifications of replenishment rules*:
  - We can further improve the replenishment rule R3a as follows: Replenish the budget at time $t_r + p_s$ whenever the higher-priority subsystem $\mathbf{T}_H$ has been busy throughout the interval $(t_r, t_r + p_s]$.

SpSL Sporadic Servers

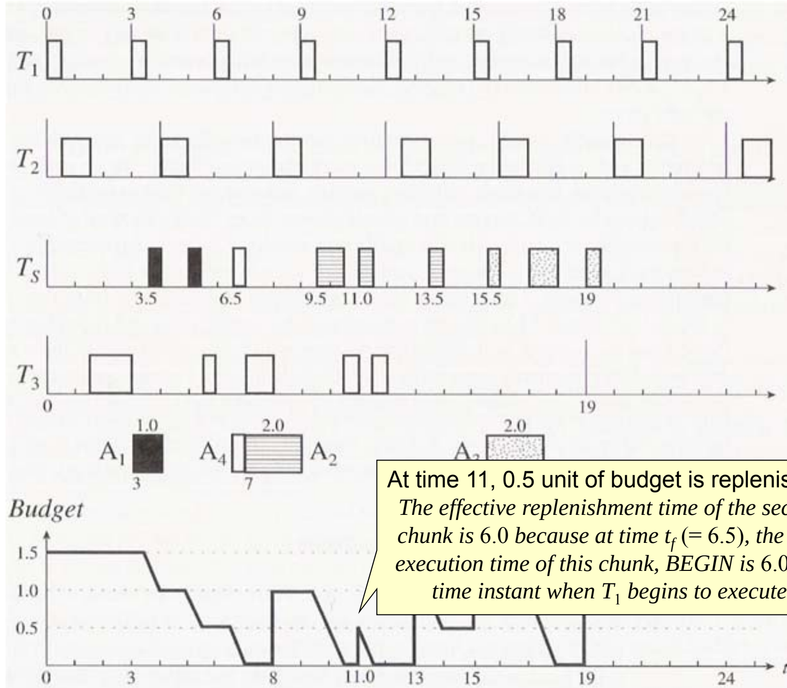
- A Sprunt, Sha, and Lehoczky (SpSL) sporadic server preserves unconsumed chunks of budget whenever possible and replenishes the consumed chunks as soon as possible.
- A SpSL server with period p_s and execution budget e_s emulates several periodic tasks with the same period and total execution time equal to e_s.



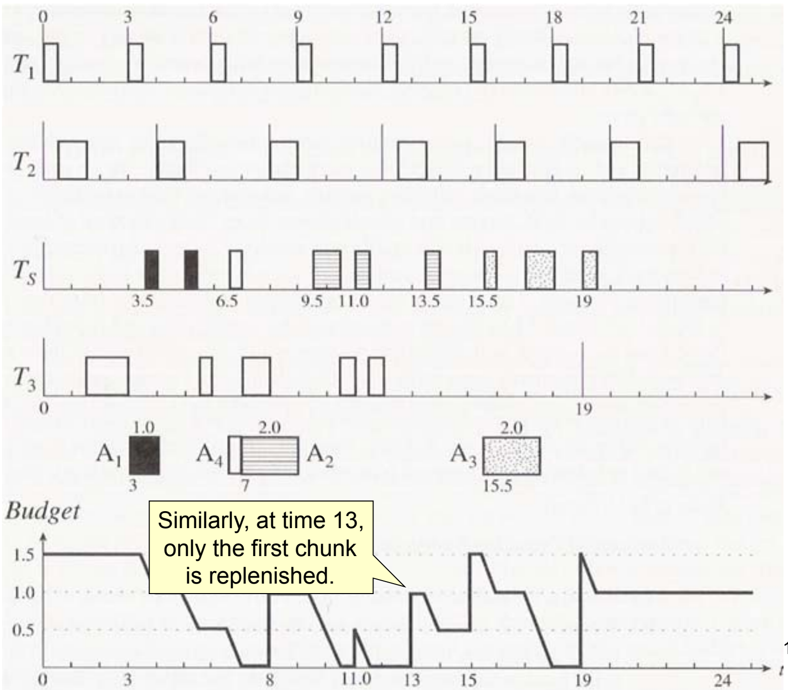




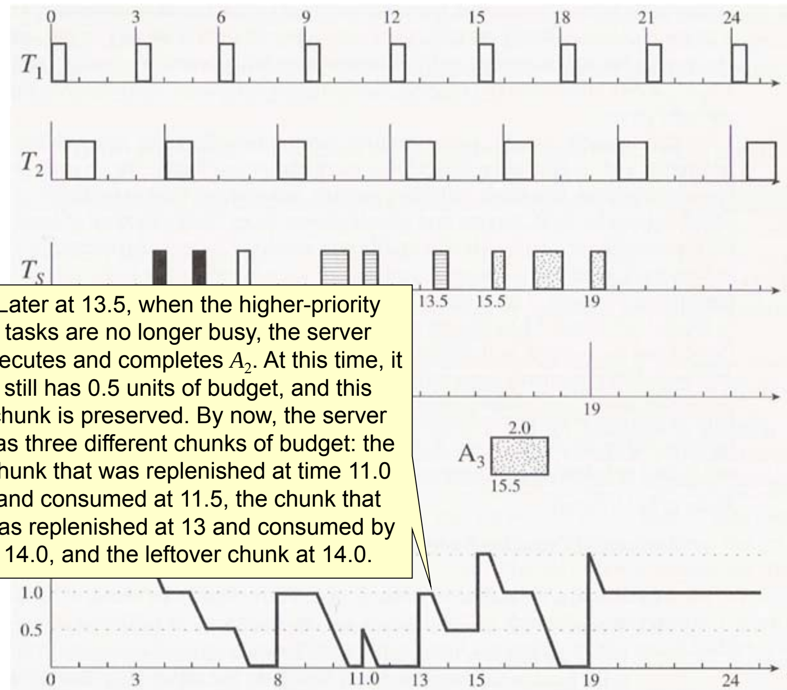
At time 8, rather than the entire budget of 1.5 units, the SpSL server replenishes only the chunk of 1.0 unit which was consumed from time 3.5 to 5.5. *This* 1.0 *unit is treated as one chunk because the execution of the server was interrupted only by a higher-priority task during the time interval when this chunk was consumed.*



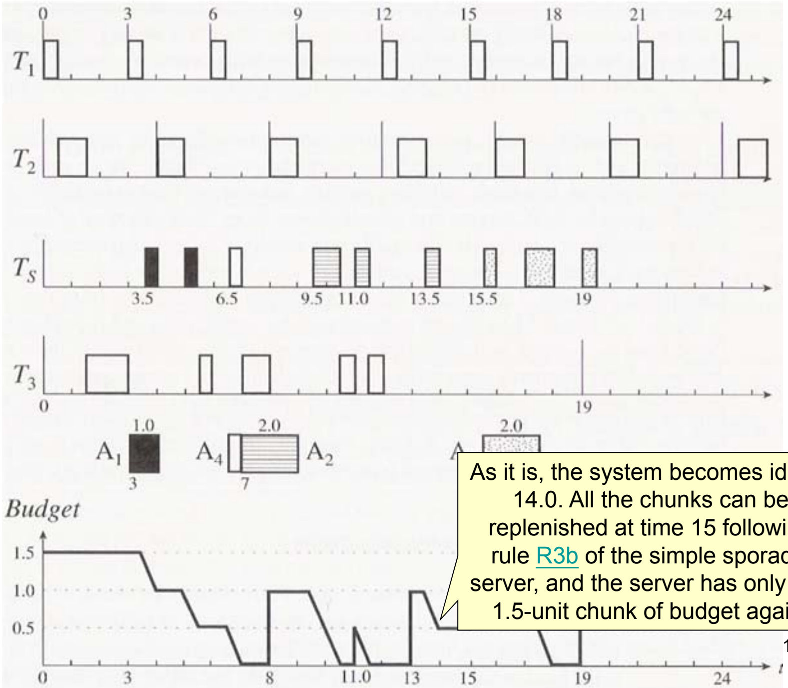
At time 11, 0.5 unit of budget is replenished. *The effective replenishment time of the second chunk is* 6.0 *because at time* t_f (= 6.5), *the first execution time of this chunk, BEGIN is* 6.0, *the time instant when* T_1 *begins to execute.*



Similarly, at time 13, only the first chunk is replenished.



Later at 13.5, when the higher-priority tasks are no longer busy, the server executes and completes A_2. At this time, it still has 0.5 units of budget, and this chunk is preserved. By now, the server has three different chunks of budget: the chunk that was replenished at time 11.0 and consumed at 11.5, the chunk that was replenished at 13 and consumed by 14.0, and the leftover chunk at 14.0.



As it is, the system becomes idle at 14.0. All the chunks can be replenished at time 15 following rule R3b of the simple sporadic server, and the server has only one 1.5-unit chunk of budget again.

Advantage of SpSL Servers

- When the aperiodic job queue frequently alternates from being nonempty to empty and vice versa, a SpSL server oftentimes can respond sooner to new arrivals of aperiodic jobs than a simple sporadic server.
- A SpSL server emulates several periodic tasks, one per chunk.
- As time progresses and its budget breaks off into more and more chunks, the original periodic task (p_s, e_s) breaks up into more and more periodic tasks whose total execution time is e_s.

Cost of SpSL Servers

- The additional cost of SpSL servers over simple servers arises due to the need of keeping track of the consumption and replenishment of different chunks of budget.
- The overhead of the SpSL server over the simple version can be as high as $O(N)$, where N is the number of jobs in a hyperperiod of the periodic tasks.

Rules of SpSL Servers (1/2)

- *Breaking of Execution Budget into Chunks*:
  - **B1**: Initially, the budget = e_s and $t_r = 0$. There is only one chunk of budget.
  - **B2**: Whenever the server is suspended, the last chunk of budget being consumed just before suspension, if not exhausted, is break up into two chunks: The first chunk is the portion that was consumed during the last server busy interval, and the second chunk is the remaining portion. The first chunk inherits the next replenishment time of the original chunk. The second chunk inherits the last replenishment time of the original chunk.

Rules of SpSL Servers (2/2)

- *Consumption Rules*:
  - **C1**: The server consumes the chunks of budget in order of their last replenishment times.
  - **C2**: The server consumes its budget only when it executes.
- *Replenishment Rules*:
  - The next replenishment time of each chunk of budget is set according to rule R2 and R3 of the simple sporadic server. The chunks are consolidated into one whenever they are replenished at the same time.

The Overhead of the SpSL Server (1/2)

- The overhead of the SpSL server over the simple version consists of the time and space required to maintain the last and the next replenishment times of individual chunks of the server budget.
- In the worst case, this can be as high as $O(N)$, where N is the number of jobs in a hyperperiod of the periodic tasks.

The Overhead of the SpSL Server (2/2)

- However, it is relative simple for the operating system to monitor the budget consumption of a SpSL server, because its budget is consumed only when it executes. (In contrast, the budget of a simple sporadic server may also be consumed during the execution of lower-priority tasks.)

Priority Exchange

- According to rule C2, once the server has executed since a replenishment, it consumes its budget when the server is idle or some lower-priority task executes.
- We can stop the consumption by allowing the server to trade time with the executing lower-priority task.

Illustration (1/6)

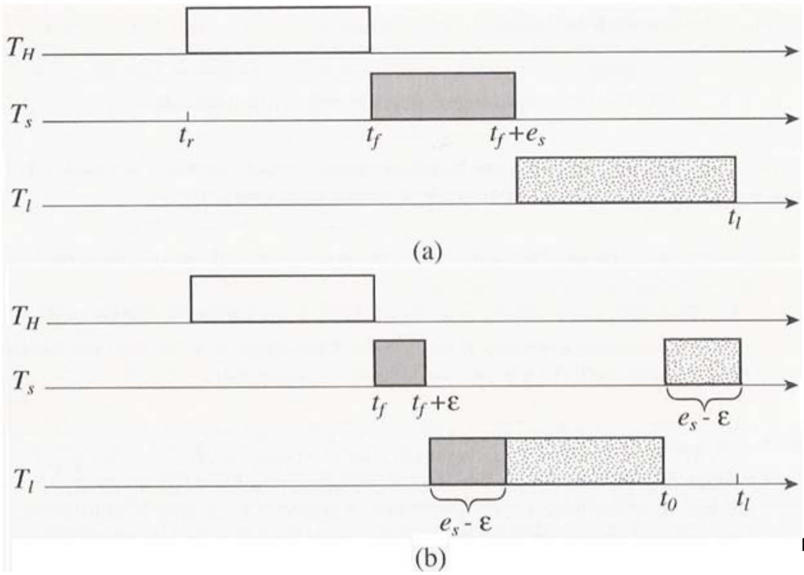
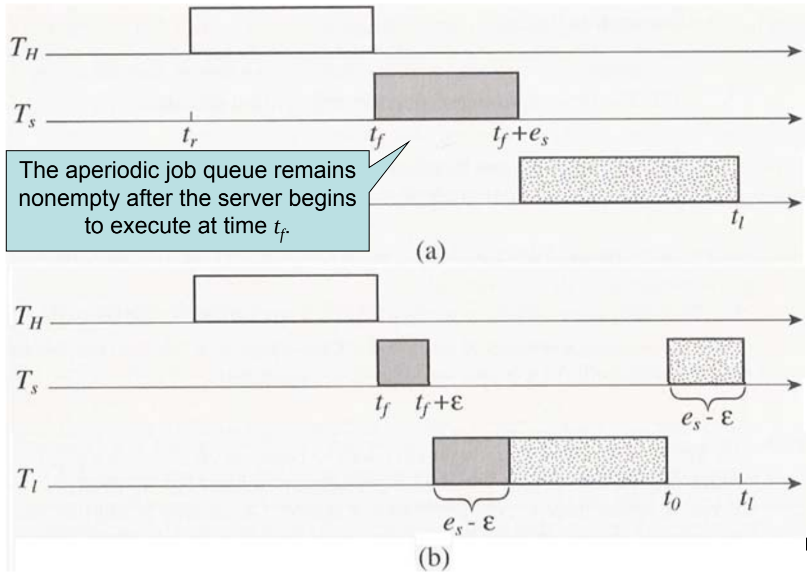
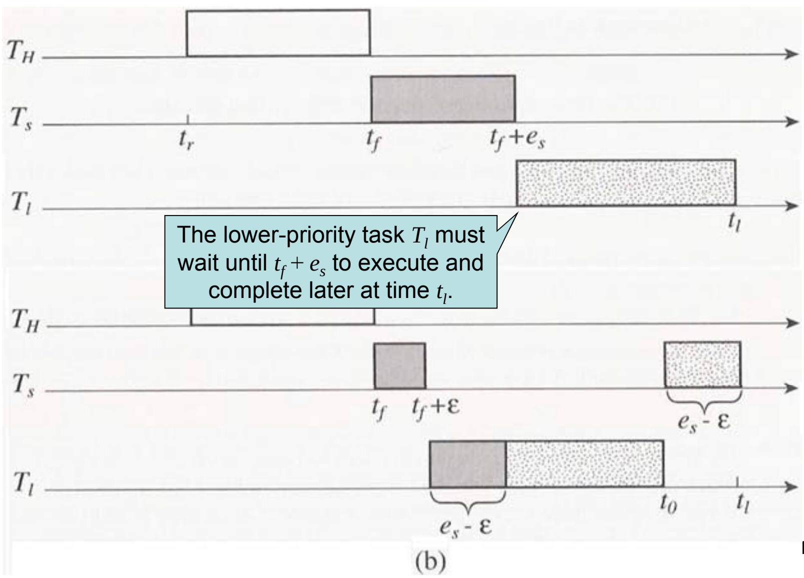


Illustration (2/6)



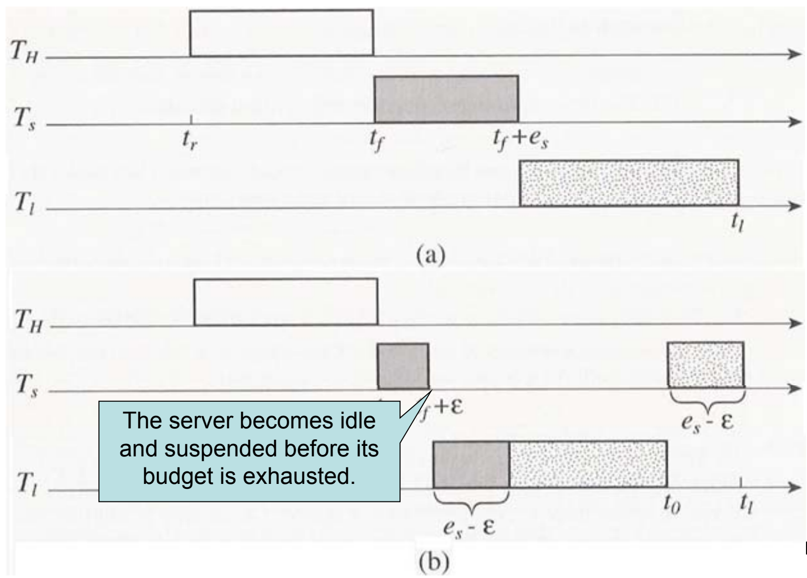
The aperiodic job queue remains nonempty after the server begins to execute at time t_f.

Illustration (3/6)



The lower-priority task T_l must wait until $t_f + e_s$ to execute and complete later at time t_l.

Illustration (4/6)



The server becomes idle and suspended before its budget is exhausted.

Illustration (5/6)

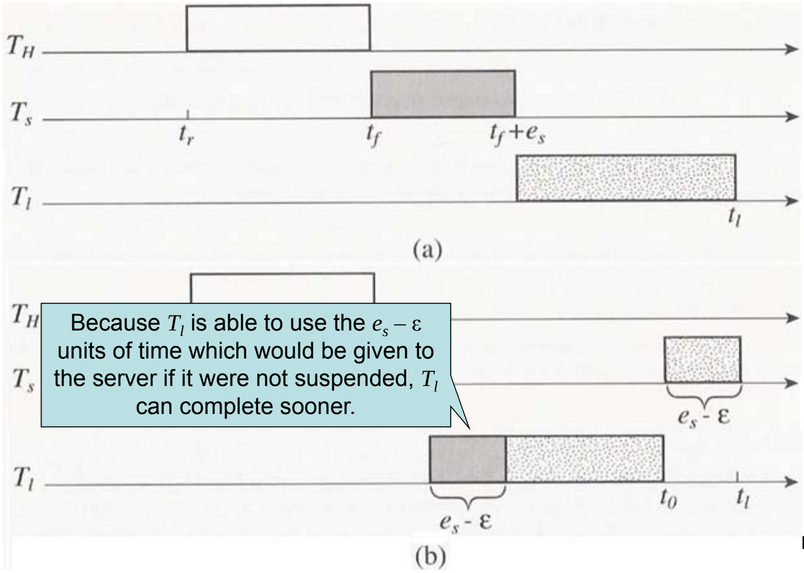
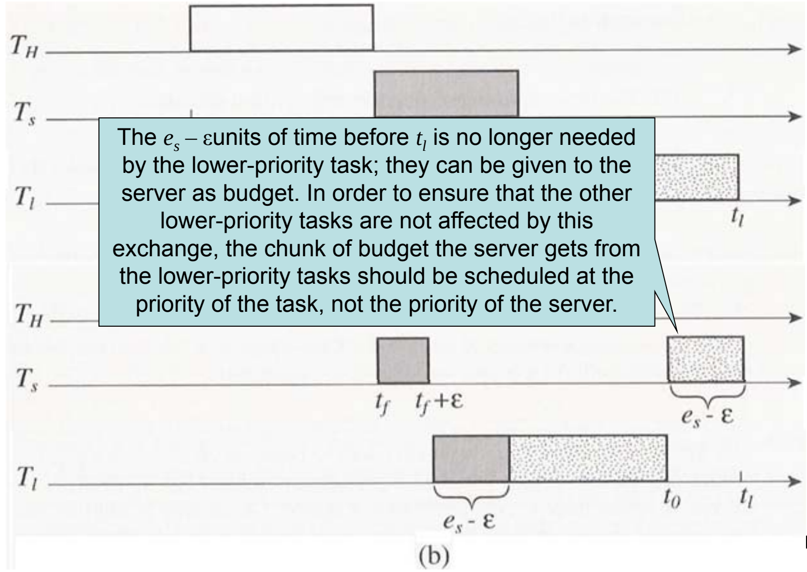


Illustration (6/6)



Priority Exchange Server

- Trading time and priorities among the server and the lower-priority tasks in this manner is exactly what a priority-exchange server does.
- ➡ Priority exchange is particularly advantageous when the aperiodic job queue becomes empty frequently.
- ➡ The high overhead of priority exchange makes this type of server impractical.

Outline

- Sporadic Servers
  1. Sporadic Server in Fixed-Priority Systems
  2. Enhancements of Fixed-Priority Sporadic Server
  3. ***Simple Sporadic Servers in Deadline-Driven Systems***

Simple Sporadic Servers in Deadline-Driven Systems (1/7)

- When the periodic tasks are scheduled on the EDF basis, the priorities of the tasks vary as their jobs are released and completed.
- The membership of the subset of tasks with higher priorities than the server varies with time.
- Some of the rules of simple sporadic servers stated earlier for fixed-priority systems must be modified such that a simple sporadic server of period p_s and budget e_s following the rules behave like a periodic task (p_s, e_s).

Simple Sporadic Servers in Deadline-Driven Systems (2/7)

- The server is ready for execution only when it is backlogged (i.e., the aperiodic job queue is nonempty) and its deadline d (and hence its priority) is set.
- The server is suspended whenever its deadline is undefined or when the server is idle (i.e., the aperiodic job queue becomes empty).

Simple Sporadic Servers in Deadline-Driven Systems (3/7)

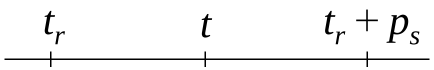
- *Consumption Rules of Simple Deadline-Driven Sporadic Server*: The server's execution budget is consumed at the rate of one per unit time until the budget is exhausted when either one of the following two conditions is true. When these conditions are not true, the server holds its budget.
  - **C1**: The server is executing.
  - **C2**: The server deadline d is defined, the server is idle, and there is no job with a deadline before d ready for execution.

Simple Sporadic Servers in Deadline-Driven Systems (4/7)

- *Replenishment Rules of Simple Deadline-Driven Sporadic Server*:
  - **R1**: Initially and at each replenishment time, t_r is the current time, and the budget = e_s. Initially, t_e and the server deadline d are undefined.
  - **R2**: Whenever t_e is defined, $d = t_e + p_s$, and the next replenishment time is $d = t_e + p_s$. Otherwise, when t_e is undefined, d remains undefined. t_e is determined (defined) as follows:

Simple Sporadic Servers in Deadline-Driven Systems (5/7)

a) At time t when an aperiodic job arrives at an empty aperiodic job queue, the value of t_e is determined based on the history of the system before t as follows:
   1. If only jobs with deadlines earlier than $t_r + p_s$ have executed throughout the interval (t_r, t), $t_e = t_r$.
   2. If some job with deadline after $t_r + p_s$ has executed in the interval (t_r, t), $t_e = t$.



Simple Sporadic Servers in Deadline-Driven Systems (6/7)

b) At replenishment time t_r,
   1. If the server is backlogged, $t_e = t_r$, and
   2. If the server is idle, t_e and d become undefined.

– **R3**: The next replenishment occurs at the next replenishment time, except under the following conditions. Under these conditions, the next replenishment is done at times stated below.
   a) If the next replenishment time $t_e + p_s$ is earlier than the time t when the server first becomes backlogged since t_r, the budget is replenished as soon as it is exhausted.
   b) The budget is replenished at the end of each idle interval of the periodic task system $\mathbf{T}$.

Simple Sporadic Servers in Deadline-Driven Systems (7/7)

- The differences in rules of simple sporadic servers in deadline-driven and fixed-priority systems are small.
- It just so happens that if we schedule the periodic tasks in the previous example according to the EDF algorithm and make sporadic server follow the rules of deadline-driven simple sporadic servers, the schedule segment we get is essential the same.

Outline

- Assumptions and Approaches
- Deferrable Servers
- Sporadic Servers
- *Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers*
- Scheduling of Sporadic Jobs

Bandwidth Preserving Servers

- Three bandwidth preserving server algorithms that offer a simple way to schedule aperiodic jobs in deadline-driven systems:
  – Constant Utilization [1]
  – Total Bandwidth [2]
  – Weighted Fair-Queueing [3]

[1] Deng, Z., J. W. S. Liu, and J. Sun, "A Scheme for Scheduling Hard Real-Time Applications in Open System Environment," *Proceedings of 9th Euromicro Workshop on Real-Time Systems*, pp. 191-199, June 1997.

[2] Spuri, M., and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real Time Systems Journal*, Vol. 10, pp. 179-210, 1996.

[3] Demers, A., S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Proceedings of ACM Sigcomm*, pp. 1-12, 1989, and *Journal of Internetworking Research and Experience*, October 1990.

Generalized Processor Sharing (GPS)

- Above three algorithms more or less emulate the GPS algorithm.
- GPS is an idealized weighted round-robin algorithm; it gives each backlogged server in each round an infinitesmally small time slice of length proportional to the server size.

Outline

- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
  1. ***Schedulability of Sporadic Jobs in Deadline-Driven Systems***
  2. Constant Utilization Server Algorithm
  3. Total Bandwidth Server Algorithm
  4. Fairness and Starvation
  5. Preemptive Weighted Fair-Queueing Algorithm

Definitions

- The density of a sporadic job J_i that has release time r_i, maximum execution time e_i and deadline d_i is the ratio $e_i/(d_i - r_i)$.
- A sporadic job is said to be active in its feasible interval $(r_i, d_i]$; it is not active outside of this interval.

Sufficient Schedulability Condition

- **Theorem 4.** A system of independent, preemptable sporadic jobs is schedulable according to the EDF algorithm if the total density of all active jobs in the system is no greater than 1 at all times.

  Proof. Please see the handout.
- **Example.** Consider three sporadic jobs each of which has a relative deadline of 2 and execution time of 1. They are released at time instants 0, 0.5, 1.0. The total density of jobs is 1.5 in (1, 2], yet they are schedulable on the EDF basis.

Schedulability of Sporadic Tasks (1/3)

- A sporadic task S_i is a stream of sporadic jobs.
- Let $S_{i,j}$ denote the jth job in the task S_i (i.e., the release time of $S_{i,j}$ is later than the release times of $S_{i,1}, S_{i,2}, \ldots, S_{i,j-1}$).
- Let $e_{i,j}$ denote the execution time of $S_{i,j}$, and $p_{i,j}$ denote the length of time between the release times of $S_{i,j}$ and $S_{i,j+1}$.
- At the risk of abusing the term, we call $p_{i,j}$ the period of the sporadic job $S_{i,j}$ and the ratio $e_{i,j}/p_{i,j}$ the instantaneous utilization of the job.

Schedulability of Sporadic Tasks (2/3)

- The instantaneous utilization $\tilde{u}_i$ of a sporadic task is the maximum of the instantaneous utilizations of all the jobs in this task (i.e., $\tilde{u}_i = \max_j (e_{i,j} / p_{i,j})$).
- We assume that the instantaneous utilization of a sporadic task is a known parameter of the task.
- **Corollary 5.** A system of n independent, preemptable sporadic tasks, which is such that the relative deadline of every job is equal to its period, is schedulable on a processor according to the EDF algorithm if the total instantaneous utilization (i.e., $\sum_{i=1}^{n} \tilde{u}_i$) is equal to or less than 1.

Schedulability of Sporadic Tasks (3/3)

- **Corollary 6.** A system of independent, preemptable periodic and sporadic tasks, which is such that the relative deadline of every job is equal to its period, is schedulable on a processor according to the EDF algorithm if the sum of the total utilization of the periodic tasks and the total instantaneous utilization of the sporadic tasks is equal to or less than 1.

Outline

- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
  1. Schedulability of Sporadic Jobs in Deadline-Driven Systems
  2. ***Constant Utilization Server Algorithm***
  3. Total Bandwidth Server Algorithm
  4. Fairness and Starvation
  5. Preemptive Weighted Fair-Queueing Algorithm

Constant Utilization Server

- Scheduling aperiodic jobs amid periodic tasks in a deadline-driven system.
- The server is defined by its size, which is its instantaneous utilization $\tilde{u}_s$; this fraction of processor time is reserved for the execution of aperiodic jobs.
- While a sporadic server emulates a periodic task, a constant utilization server emulates a sporadic task with a constant instantaneous utilization, and hence its name.

Consumption and Replenishment Rules (1/2)

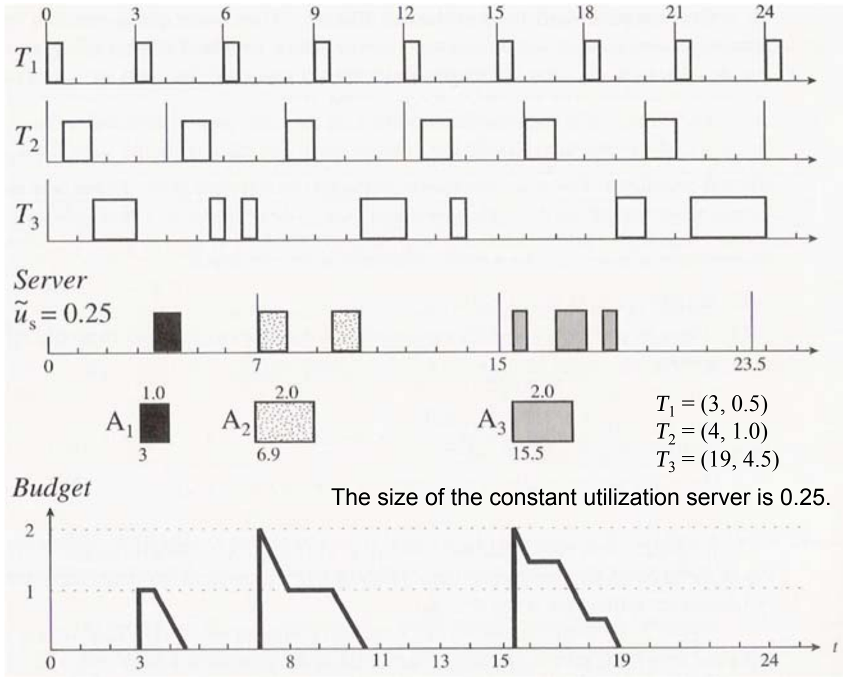
- *Consumption Rules of a Constant Utilization Server*: A server consumes its budget only when it executes.
- Notations for Replenishment Rules:
  - $\tilde{u}_s$: the size of the server.
  - e_s: its budget
  - d: its deadline
  - t: the current time
  - e: the execution time of the job at the head of the aperiodic job queue.

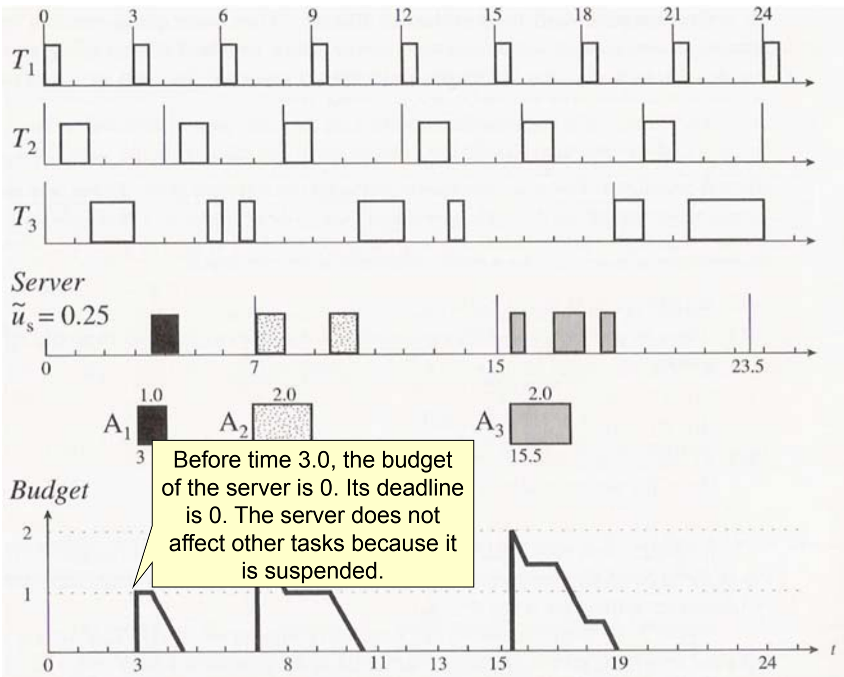
Consumption and Replenishment Rules (2/2)

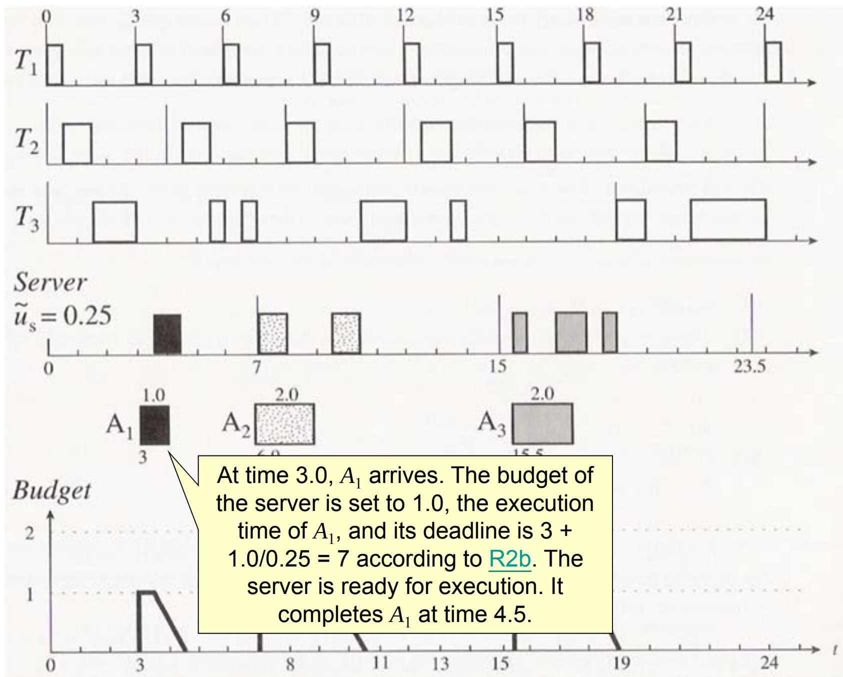
- *Replenishment Rules of a Constant Utilization Server of Size* $\tilde{u}_s$:
  - **R1**: Initially, $e_s = 0$, and $d = 0$.
  - **R2**: When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue,
    a) If $t < d$, do nothing;
    b) If $t \geq d$, $d = t + e/\tilde{u}_s$, and $e_s = e$.
  - **R3**: At the deadline d of the server,
    a) If the server is backlogged, set the server deadline to $d + e/\tilde{u}_s$ and $e_s = e$;
    b) If the server is idle, do nothing.

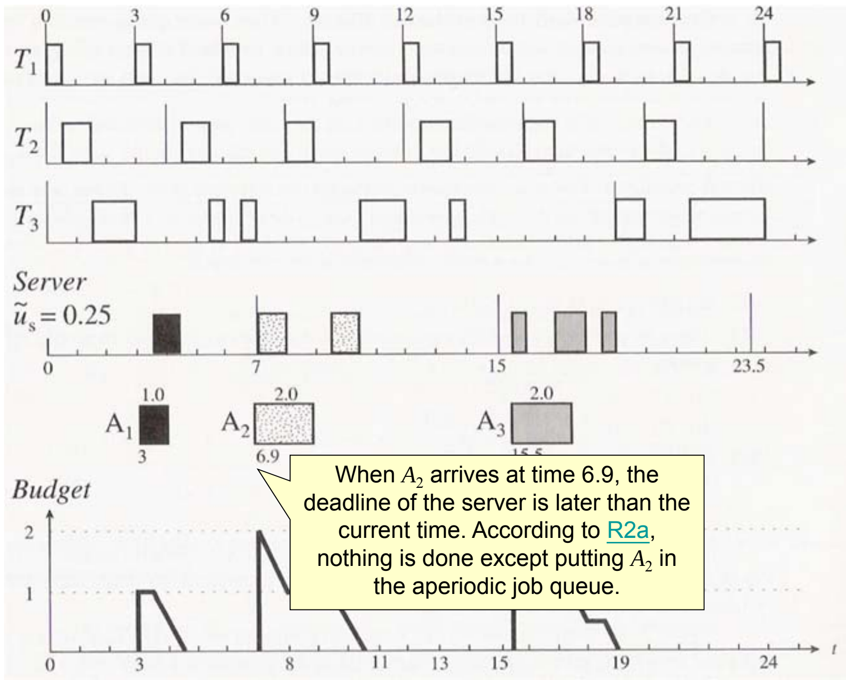
Remark 1

- A constant utilization server is always given enough budget to complete the job at the head of its queue each time its budget is replenished.
- Its deadline is set so that its instantaneous utilization is equal to $\tilde{u}_s$.

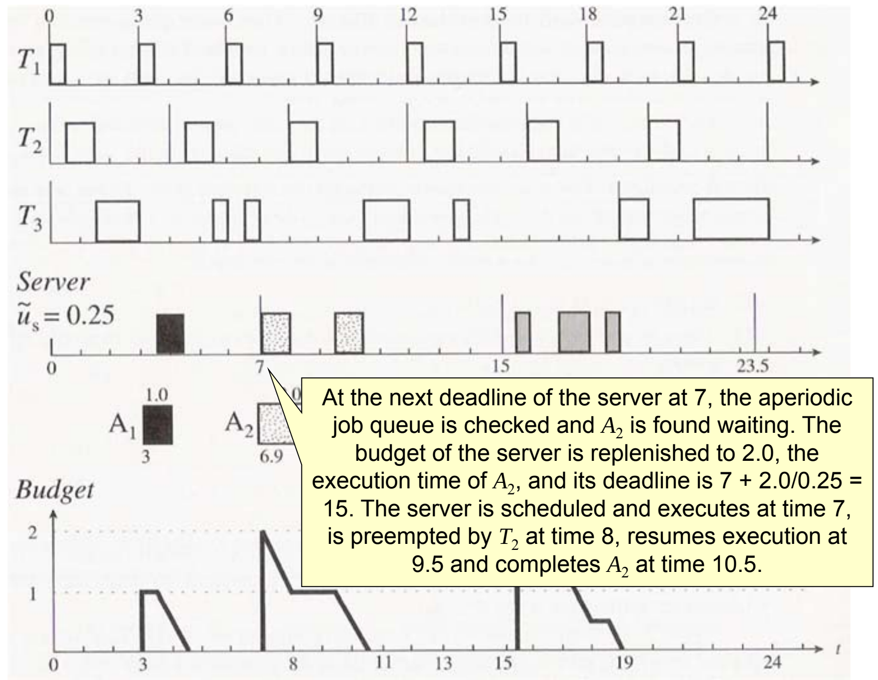




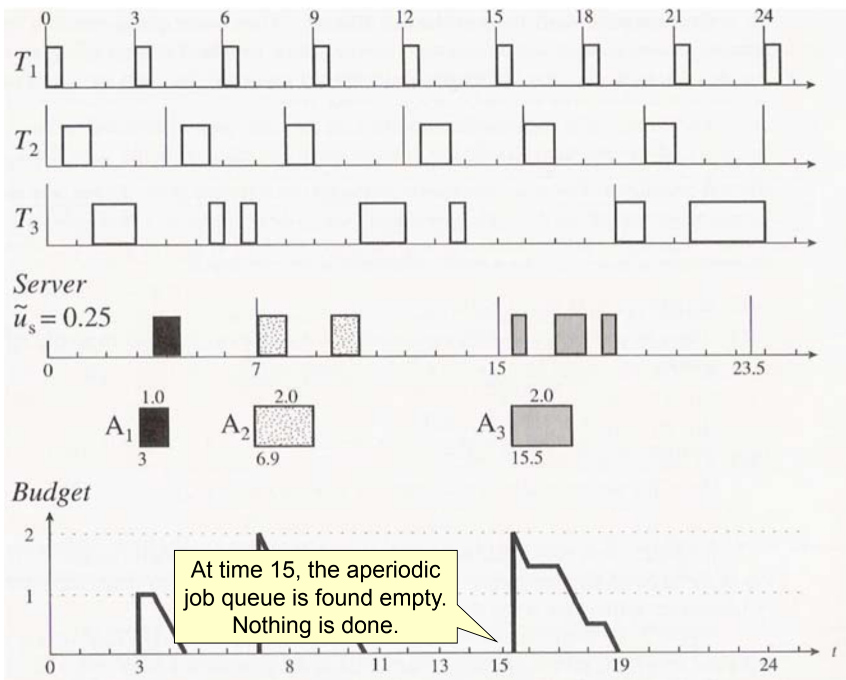




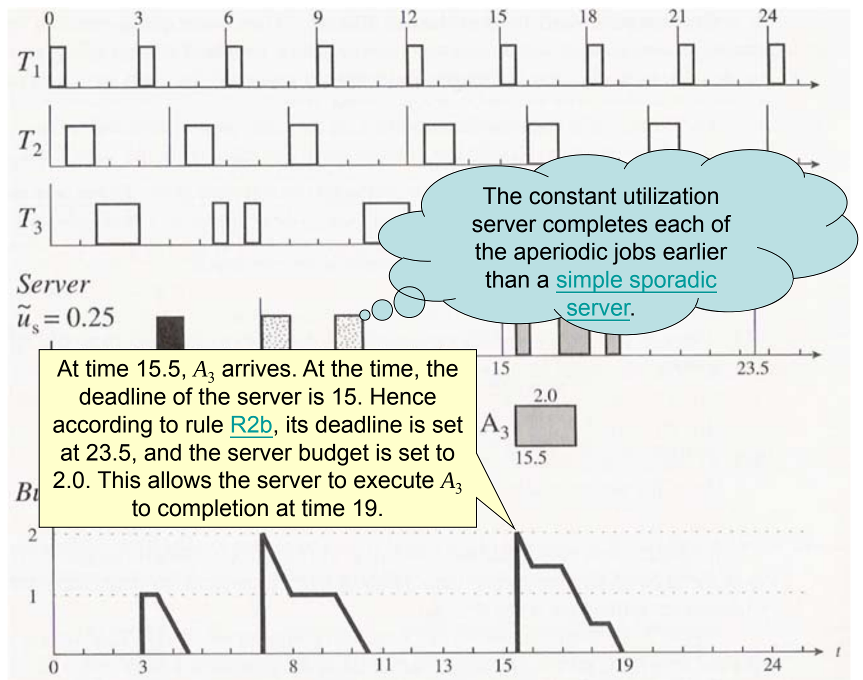
When A_2 arrives at time 6.9, the deadline of the server is later than the current time. According to R2a, nothing is done except putting A_2 in the aperiodic job queue.



At the next deadline of the server at 7, the aperiodic job queue is checked and A_2 is found waiting. The budget of the server is replenished to 2.0, the execution time of A_2, and its deadline is 7 + 2.0/0.25 = 15. The server is scheduled and executes at time 7, is preempted by T_2 at time 8, resumes execution at 9.5 and completes A_2 at time 10.5.



At time 15, the aperiodic job queue is found empty. Nothing is done.



The constant utilization server completes each of the aperiodic jobs earlier than a simple sporadic server.

At time 15.5, A_3 arrives. At the time, the deadline of the server is 15. Hence according to rule R2b, its deadline is set at 23.5, and the server budget is set to 2.0. This allows the server to execute A_3 to completion at time 19.

Scheduling Aperiodic Jobs with Unknown Execution Times (1/2)

- One way is to give the server a fixed size budget e_s and fixed period $e_s / \tilde{u}_s$ just like sporadic and deferrable servers.
- When an aperiodic job with execution time e shorter than e_s completes, we reduce the current deadline of the server by $(e_s - e)/\tilde{u}_s$ units before replenishing the next e_s units of budget and setting the deadline accordingly.

Scheduling Aperiodic Jobs with Unknown Execution Times (2/2)

- An aperiodic job with execution time larger than e_s is executed in more than one server period.
- We can treat the last chunk of such a job in the manner described above if the execution time of this chunk is less than e_s.

Outline

- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
  1. Schedulability of Sporadic Jobs in Deadline-Driven Systems
  2. Constant Utilization Server Algorithm
  3. ***Total Bandwidth Server Algorithm***
  4. Fairness and Starvation
  5. Preemptive Weighted Fair-Queueing Algorithm

Total Bandwidth Server (1/2)

- In the previous example, suppose that A_3 were to arrive at time 14 instead.
  - Since 14 is before current server deadline 15, the scheduler must wait until time 15 to replenish the budget of the constant utilization server.
  - A_3 waits in the interval from 14 to 15, while the processor idles!
- ➡ The total bandwidth server algorithm improves the responsiveness of a constant utilization server by allowing the server to claim the background time not used by periodic tasks.

Total Bandwidth Server (2/2)

- The scheduler replenish the server budget as soon as the budget is exhausted if the server is backlogged at the time or as soon as the server becomes backlogged.
- The rules of a total bandwidth server are even simpler than the rules of a constant utilization server.

Consumption and Replenishment Rules (1/2)

- *Consumption Rule of a Total Bandwidth Server*: A server consumes its budget only when it executes.
- *Replenishment Rules of a Total Bandwidth Server of size* $\tilde{u}_s$:
  - **R1**: Initially, $e_s = 0$ and $d = 0$.
  - **R2**: When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue, set d to $\max(d, t) + e / \tilde{u}_s$, and $e_s = e$.

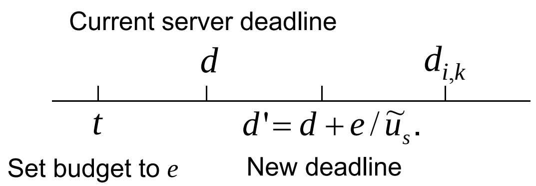
Consumption and Replenishment Rules (2/2)

- **R3**: When the server completes the current aperiodic job, the job is removed from its queue.
  a) If the server is backlogged, the server deadline is set to $d + e / \tilde{u}_s$, and $e_s = e$.
  b) If the sever is idle, do nothing.
- For a given set of aperiodic jobs and server size, both kinds of servers have the same sequence of deadlines, but the budget of a total bandwidths server may be replenished earlier than that of a constant utilization server.
- As long as a total bandwidth server is backlogged, it is always ready for execution.

Correctness of a TBS (1/4)

- To see why it works correctly, let us examine how the server affects periodic jobs and other servers when its budget is set to e at a time t before the current server deadline d and its deadline is postponed to the new deadline $d' = d + e / \tilde{u}_s$.
- We compare the amount of processor time demanded by the server with the amount of time demanded by a constant utilization server of the same size $\tilde{u}_s$ before the deadline $d_{i,k}$ of a periodic job $J_{i,k}$ whose deadline is later than the server's new deadline d'.

Correctness of a TBS (2/4)



- If the job $J_{i,k}$ is ready at t, then the amounts of time consumed by both servers are the same in the interval from t to $d_{i,k}$. Why?
- If $J_{i,k}$ is not yet released at t, then the time demanded by the TBS in the interval $(r_{i,k}, d']$ is less than the time demanded by the CUS. Why?

Correctness of a TBS (3/4)

- In any case, the total bandwidth server will not cause a job such as $J_{i,k}$ to miss its deadline if the constant utilization server will not.
- **Corollary 7.** When a system of independent, preemptable periodic tasks is scheduled with one or more total bandwidth and constant utilization servers on the EDF basis, every periodic task and every server meets its deadlines if the sum of the total density of periodic tasks and the total size of all servers is no greater than 1.

Correctness of a TBS (4/4)

- The expression “a server meets its deadline” or “a server is schedulable” means that the budget of the server is always consumed by the deadline set at the time when the budget was replenished.

Nonpreemptability (1/4)

- When some periodic tasks and sporadic jobs have nonpreemptable portions, the effect of nonpreemptability is a reduction in the schedulable utilization.
- Notations:
  - $b_{\max}(np)$: the maximum execution time of nonpreemptable portions of all periodic tasks and jobs executed by servers.
  - $D_{\min}$: the minimum of the relative deadlines of all periodic tasks and the effective execution times of jobs executed by all servers in the system.

Nonpreemptability (2/4)

- By the effective execution time of a job executed by a server, we mean the ratio of the job execution time and the server size.

- The nonpreemptive version of the total bandwidth server algorithm is called the virtual clock algorithm*.

* Zhang, L., "VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks," *Proceedings of ACM SIGCOMM*, 1990.

Nonpreemptability (3/4)

- **Corollary 8.** When a system of periodic tasks is scheduled with one or more total bandwidth and constant utilization servers on the EDF basis, every periodic task and every server meets its deadlines if the sum of the total density of the periodic tasks and the total size of all servers is no greater than $1 - b_{max}(np)/D_{min}$.

Nonpreemptability (4/4)

- **Corollary 9.** If the sum of the total density of all the periodic tasks and the total size of total bandwidth and constant utilization servers that are scheduled on the EDF basis is no greater than 1, the tardiness of every periodic task or server is no greater than $b_{max}(np)$.

Outline

- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
  1. Schedulability of Sporadic Jobs in Deadline-Driven Systems
  2. Constant Utilization Server Algorithm
  3. Total Bandwidth Server Algorithm
  4. ***Fairness and Starvation***
  5. Preemptive Weighted Fair-Queueing Algorithm

Example of Starvation (1/2)

- Consider a system consisting solely of two total bandwidth servers, TB_1 and TB_2, each of size 0.5.
- Suppose that in the interval $(0, t)$, for some $t > 0$, server TB_1 remains backlogged, but server TB_2 remains idle.
- By time t, TB_1 have executed for t units of time and its deadline is at least equal to $2t$.
- If at time t, a stream of jobs, each with execution time small compared with t, arrives and keeps TB_2 backlogged after t.

Example of Starvation (2/2)

- In the interval $(t, 2t)$, the deadline of TB_2 is earlier than the deadline of TB_1.
- Hence, TB_2 continues to execute, and TB_1 is starved during this interval.
- By a scheduling algorithm being fair within a time interval, we mean that the fraction time of processor time in the interval attained by each server that is backlogged throughout the interval is proportional to the server size.

Definition of Fairness (1/2)

- Consider a system consisting solely of n (> 1) servers:
  - Each sever executes an aperiodic or sporadic task.
  - For $i = 1, 2, \ldots, n$, the size of the i th server is $\tilde{u}_i$.
  - $\sum_{i=1}^{n} \tilde{u}_i$ is no greater than 1.
  - $\tilde{w}_i(t_1, t_2)$, for $0 < t_1 < t_2$, denote the total attained processor time of the i th server in the time interval (t_1, t_2), that is, the server executes for $\tilde{w}_i(t_1, t_2)$ unit of time during this interval.
  - The ratio of $\tilde{w}_i(t_1, t_2) / \tilde{u}_i$ is called the normalized service attained by the i th server.

Definition of Fairness (2/2)

- A scheduler (or the scheduling algorithm used by the scheduler) is fair in the interval (t_1, t_2) if the normalized services attained by all servers that are backlogged during the interval differ by no more than the fairness threshold $FR \geq 0$.
- In the ideal case, FR is equal to zero, and

$$\frac{\tilde{w}_i(t_1, t_2)}{\tilde{w}_j(t_1, t_2)} = \frac{\tilde{u}_i}{\tilde{u}_j}$$

  for any $t_2 > t_1$ and i th server that are backlogged throughout the time interval (t_1, t_2).
  Equivalently, $\tilde{w}_i(t_1, t_2) = \tilde{u}_i(t_2 - t_1)$.

Elimination of Starvation (1/2)

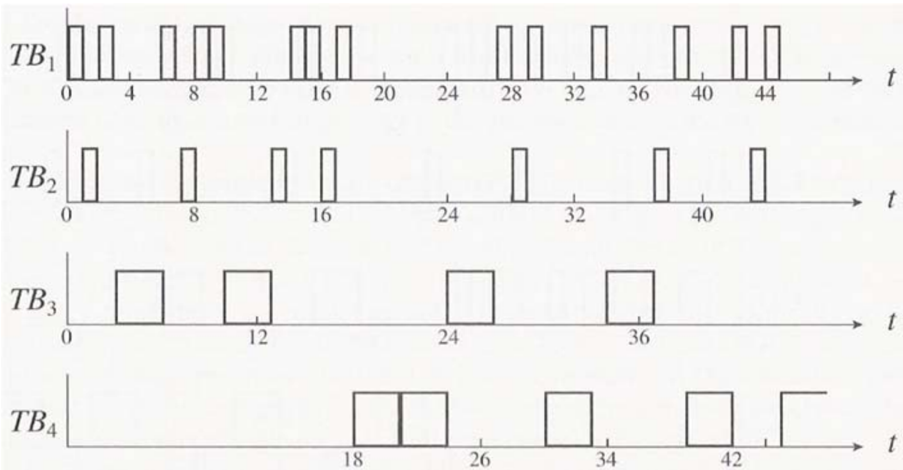
- In the previous example, the starvation problem is due to the way in which the total bandwidth server algorithm makes background time available to TB_1.
- In general, the deadline of a backlogged total bandwidth server is allowed to be arbitrarily far in the future when there is spare processor time.
- The simple scheme eliminates starvation and improves fairness by keeping the deadlines of backlogged servers sufficiently close to the current time.

Elimination of Starvation (2/2)

- *Replenishment Rules of a Starvation-Free Constant Utilization/Background Server*:
  - **R1**–**R3**: Within any busy interval of the system, replenish the budget of each backlogged server following rules of a constant utilization server.
  - **R4**: Whenever a busy interval of the system ends, replenish the budgets of all backlogged servers.

CUS is starvation-free. Why?

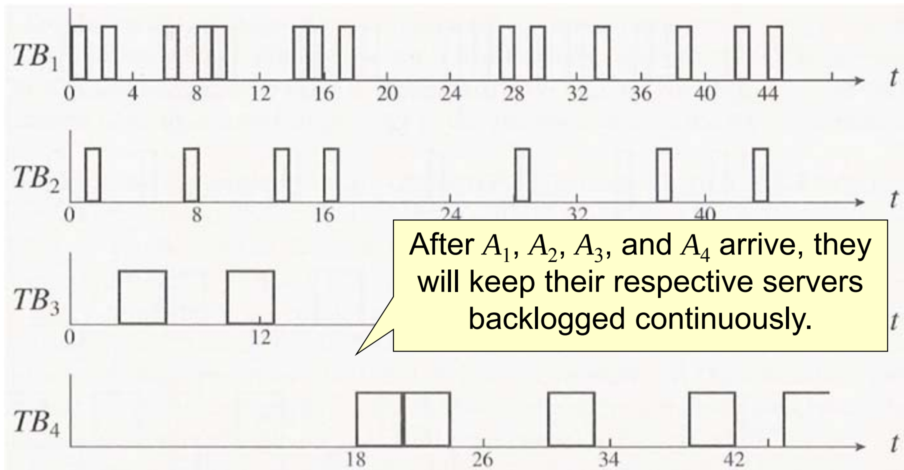
Behavior of Total Bandwidth Servers



$\tilde{u}_1 = \frac{1}{4}, \tilde{u}_2 = \frac{1}{8}, \tilde{u}_3 = \frac{1}{4}, \tilde{u}_4 = \frac{3}{8}$

A_1, A_2 with execution times = 1 arriving from $t = 0$.
A_3 with execution times = 3 arriving from $t = 0$.
A_4 with execution times = 3 arriving from $t = 18$.

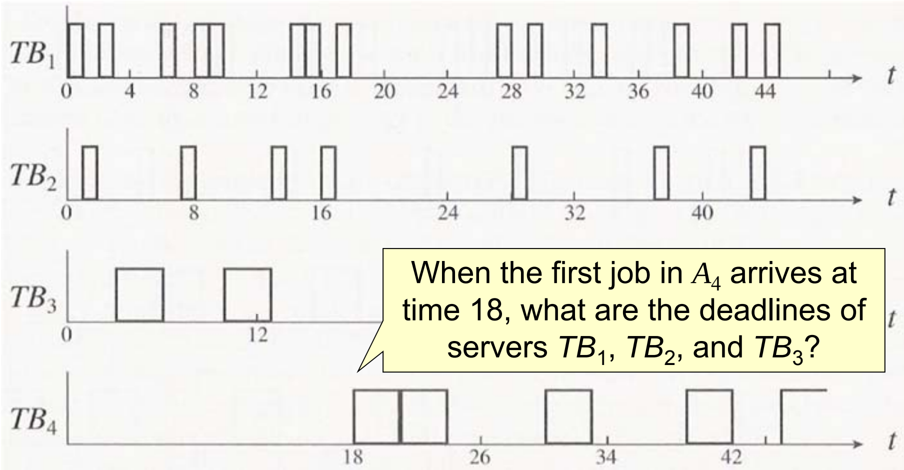
Behavior of Total Bandwidth Servers



$\tilde{u}_1 = \frac{1}{4}, \tilde{u}_2 = \frac{1}{8}, \tilde{u}_3 = \frac{1}{4}, \tilde{u}_4 = \frac{3}{8}$

A_1, A_2 with execution times = 1 arriving from $t = 0$.
A_3 with execution times = 3 arriving from $t = 0$.
A_4 with execution times = 3 arriving from $t = 18$.

Behavior of Total Bandwidth Servers

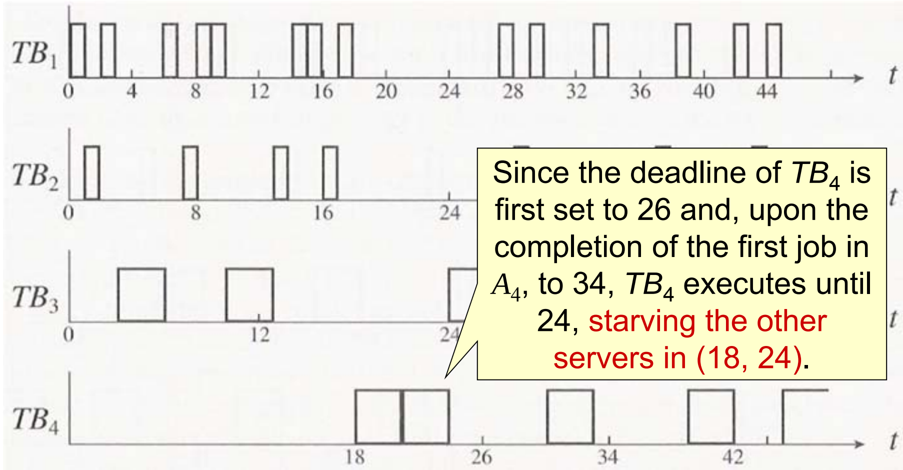


When the first job in A_4 arrives at time 18, what are the deadlines of servers TB_1, TB_2, and TB_3?

$\tilde{u}_1 = \frac{1}{4}, \tilde{u}_2 = \frac{1}{8}, \tilde{u}_3 = \frac{1}{4}, \tilde{u}_4 = \frac{3}{8}$

A_1, A_2 with execution times = 1 arriving from $t = 0$.
A_3 with execution times = 3 arriving from $t = 0$.
A_4 with execution times = 3 arriving from $t = 18$.

Behavior of Total Bandwidth Servers

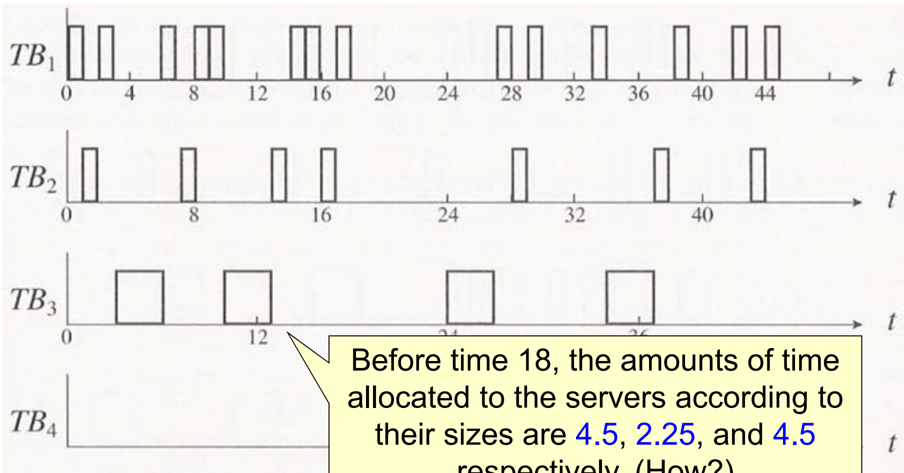


Since the deadline of TB_4 is first set to 26 and, upon the completion of the first job in A_4, to 34, TB_4 executes until 24, starving the other servers in (18, 24).

$\tilde{u}_1 = \frac{1}{4}, \tilde{u}_2 = \frac{1}{8}, \tilde{u}_3 = \frac{1}{4}, \tilde{u}_4 = \frac{3}{8}$

A_1, A_2 with execution times = 1 arriving from $t = 0$.
A_3 with execution times = 3 arriving from $t = 0$.
A_4 with execution times = 3 arriving from $t = 18$.

Behavior of Total Bandwidth Servers

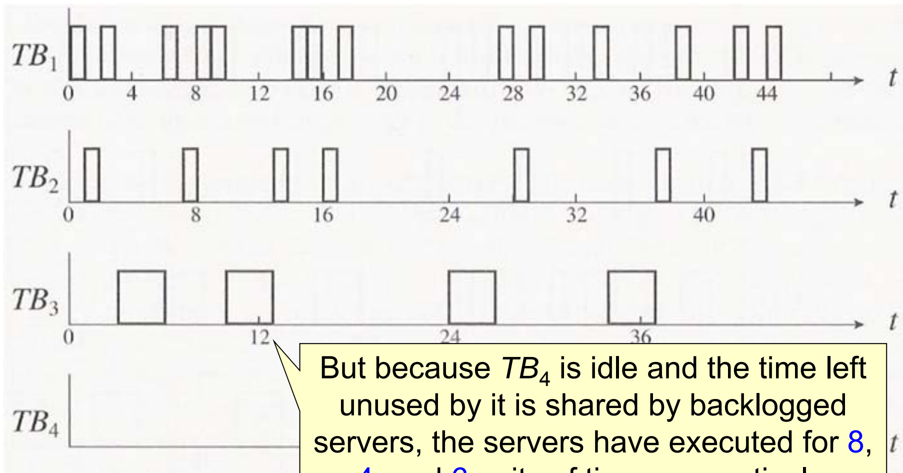


Before time 18, the amounts of time allocated to the servers according to their sizes are 4.5, 2.25, and 4.5 respectively. (How?)

$\tilde{u}_1 = \frac{1}{4}, \tilde{u}_2 = \frac{1}{8}, \tilde{u}_3 = \frac{1}{4}, \tilde{u}_4 = \frac{3}{8}$

A_1, A_2 with execution times = 1 arriving from $t = 0$.
A_3 with execution times = 3 arriving from $t = 0$.
A_4 with execution times = 3 arriving from $t = 18$.

Behavior of Total Bandwidth Servers

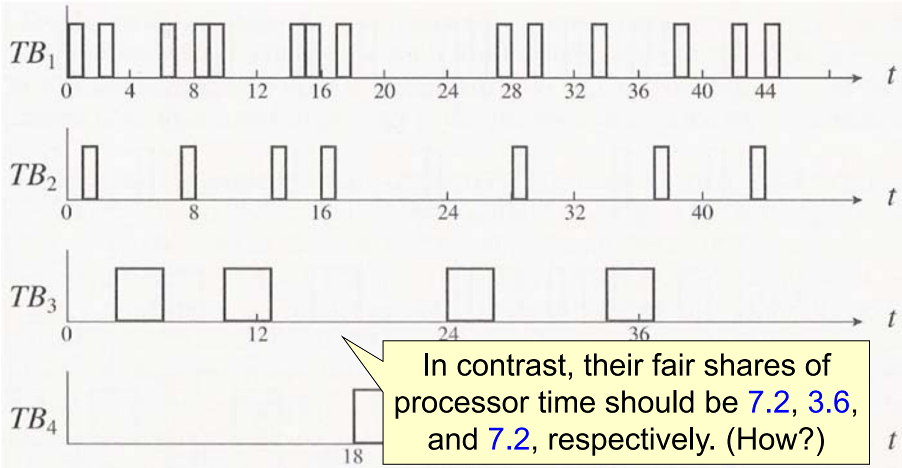


But because TB_4 is idle and the time left unused by it is shared by backlogged servers, the servers have executed for 8, 4, and 6 units of time, respectively.

$\tilde{u}_1 = \frac{1}{4}, \tilde{u}_2 = \frac{1}{8}, \tilde{u}_3 = \frac{1}{4}, \tilde{u}_4 = \frac{3}{8}$

A_1, A_2 with execution times = 1 arriving from $t = 0$.
A_3 with execution times = 3 arriving from $t = 0$.
A_4 with execution times = 3 arriving from $t = 18$.

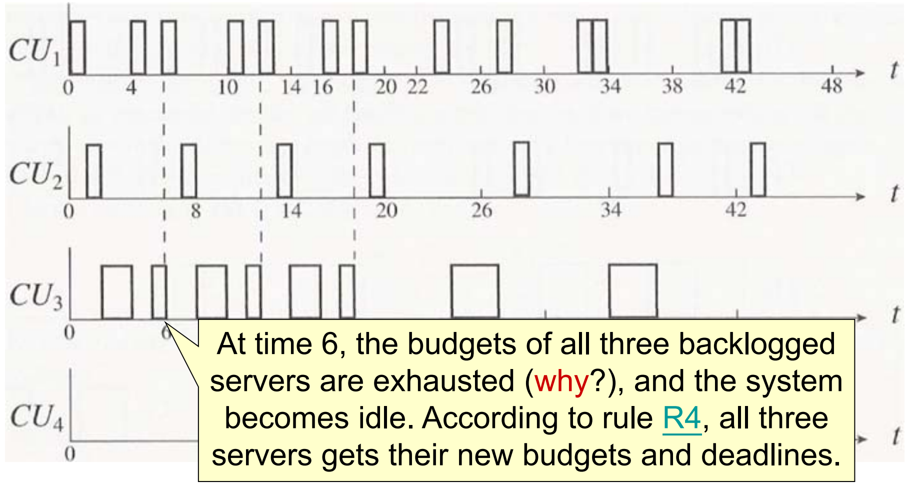
Behavior of Total Bandwidth Servers



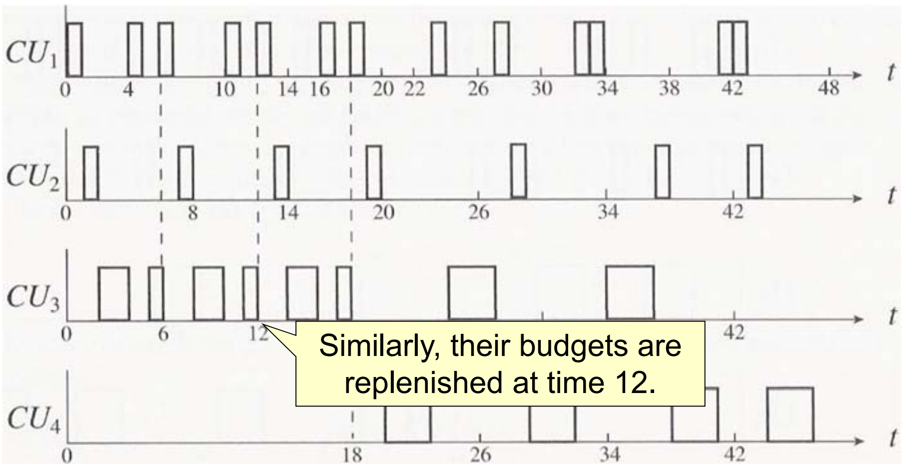
$\tilde{u}_1 = \frac{1}{4}, \tilde{u}_2 = \frac{1}{8}, \tilde{u}_3 = \frac{1}{4}, \tilde{u}_4 = \frac{3}{8}$

A_1, A_2 with execution times = 1 arriving from $t = 0$.
A_3 with execution times = 3 arriving from $t = 0$.
A_4 with execution times = 3 arriving from $t = 18$.

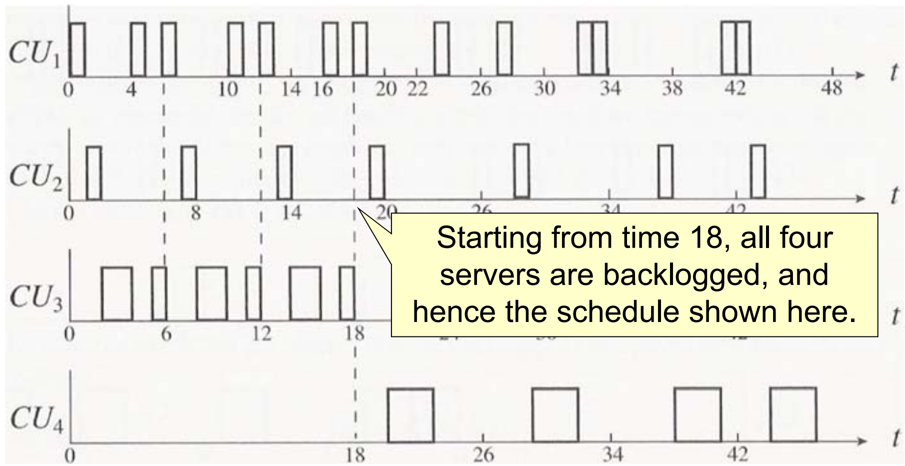
Behavior of Starvation-Free Constant Utilization/Background



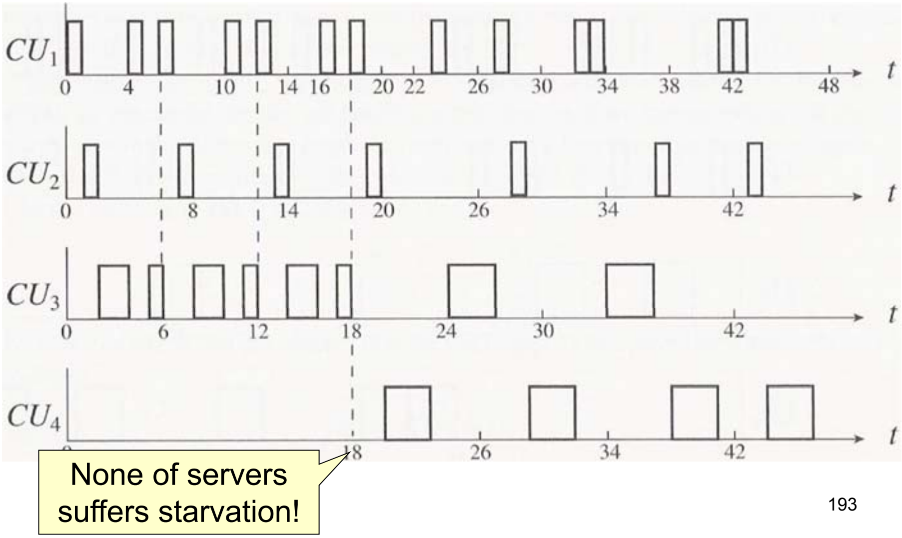
Behavior of Starvation-Free Constant Utilization/Background



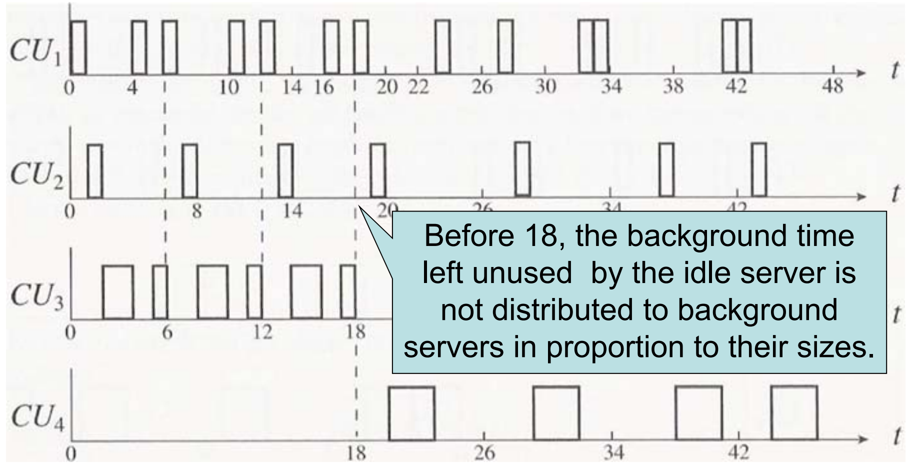
Behavior of Starvation-Free Constant Utilization/Background



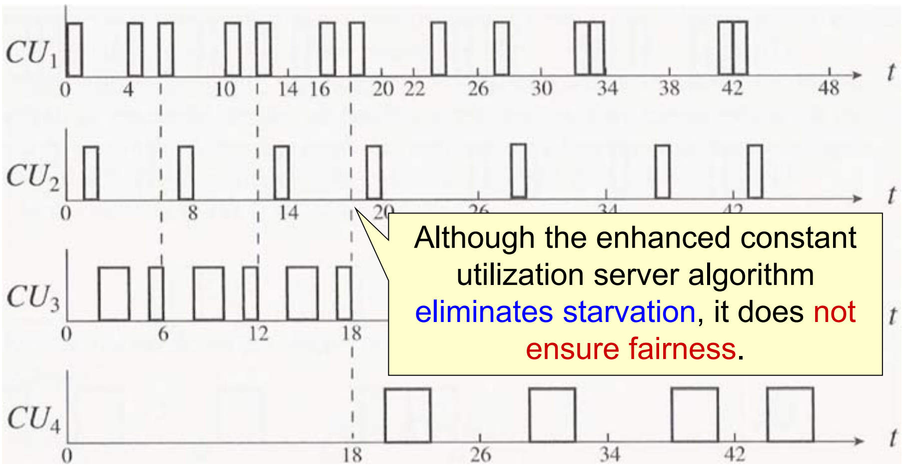
Behavior of Starvation-Free Constant Utilization/Background



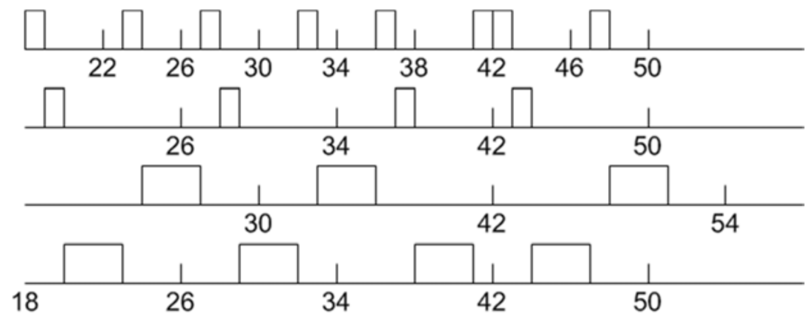
Behavior of Starvation-Free Constant Utilization/Background



Behavior of Starvation-Free Constant Utilization/Background



Correction



Outline

- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
  1. Schedulability of Sporadic Jobs in Deadline-Driven Systems
  2. Constant Utilization Server Algorithm
  3. Total Bandwidth Server Algorithm
  4. Fairness and Starvation
  5. ***Preemptive Weighted Fair-Queueing Algorithm***

Weighted Fair-Queueing

- The Weighted Fair-Queueing (WFQ) algorithm (or the PGPS (packet-by-packet GPS algorithm)) is a nonpreemptive algorithm for scheduling packet transmissions in switched networks.
- The WFQ algorithm is designed to *ensure fairness* among multiple servers.
- The replenishment rules of a WFQ server appear to be the same as those of a total bandwidth server, except for how the deadline is computed at each replenishment time.
- The total bandwidth server algorithm is unfair, but the WFQ algorithm gives bounded fairness.

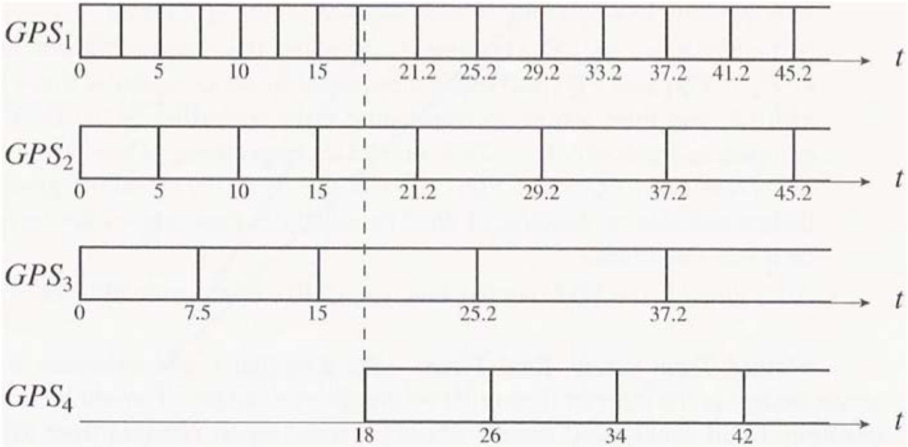
Emulation of GPS Algorithm (1/2)

- A WFQ server consumes its budget only when it executes.
- Its budget is replenished when it first becomes backlogged after being idle.
- As long as it is backlogged, its budget is replenished each time when it completes a job.
  - At each replenishment time, the server budget is set to the execution time of the job at the head of its queue.

Emulation of GPS Algorithm (2/2)

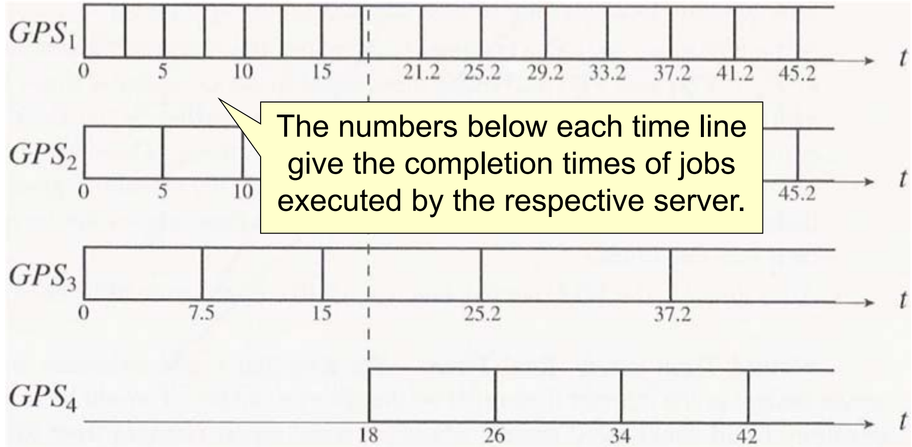
- The replenishment rules of the WFQ algorithm are such that a WFQ server emulates a GPS server of the same size.
- ➡ The deadline of the WFQ server is the time at which a GPS server would complete the job at the head of the server queue.
- ➡ The GPS scheduler schedules backlogged servers on a weighted round-robin basis, with an infinitesmally small round length and the time per round given to each server is proportional to the server size.

Behavior of GPS Servers (1/6)



The sizes of GPS1, GPS2, GPS3, and GPS4 are 1/4, 1/8, 1/4, and 3/8.

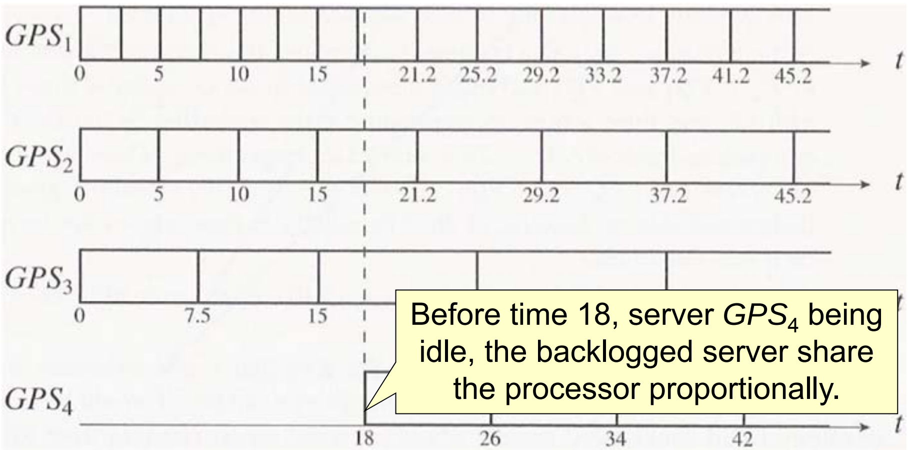
Behavior of GPS Servers (2/6)



The numbers below each time line give the completion times of jobs executed by the respective server.

The sizes of GPS1, GPS2, GPS3, and GPS4 are 1/4, 1/8, 1/4, and 3/8.

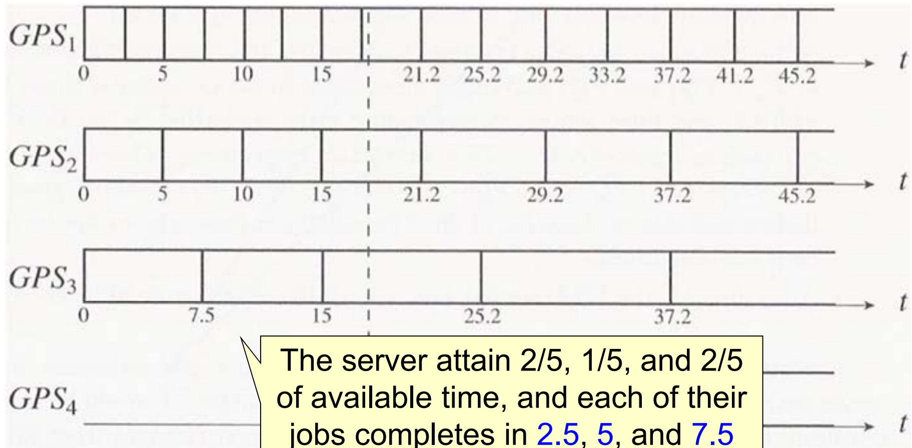
Behavior of GPS Servers (3/6)



Before time 18, server GPS_4 being idle, the backlogged server share the processor proportionally.

The sizes of GPS1, GPS2, GPS3, and GPS4 are 1/4, 1/8, 1/4, and 3/8.

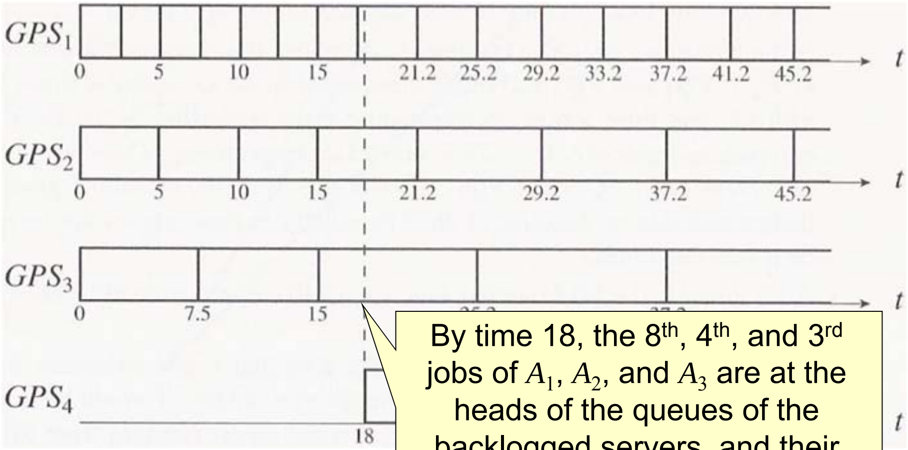
Behavior of GPS Servers (4/6)



The server attain 2/5, 1/5, and 2/5 of available time, and each of their jobs completes in 2.5, 5, and 7.5 units of time, respectively. (How?)

The sizes of GP... 1/8, 1/4, and 3/8.

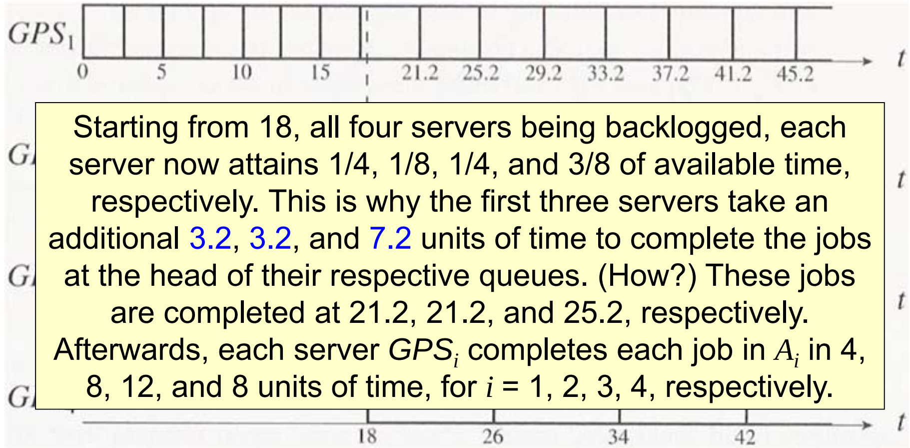
Behavior of GPS Servers (5/6)



By time 18, the 8th, 4th, and 3rd jobs of A_1, A_2, and A_3 are at the heads of the queues of the backlogged servers, and their remaining execution times are 0.8, 0.4, and 1.8, respectively. (How?)

The sizes of GPS1, GPS2, GPS3, and GPS4 are 1/4, 1/8, 1/4, and 3/8.

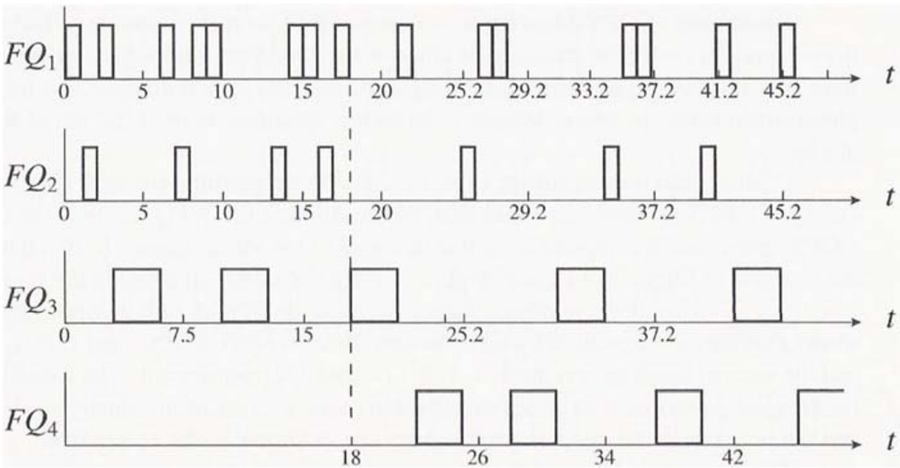
Behavior of GPS Servers (6/6)



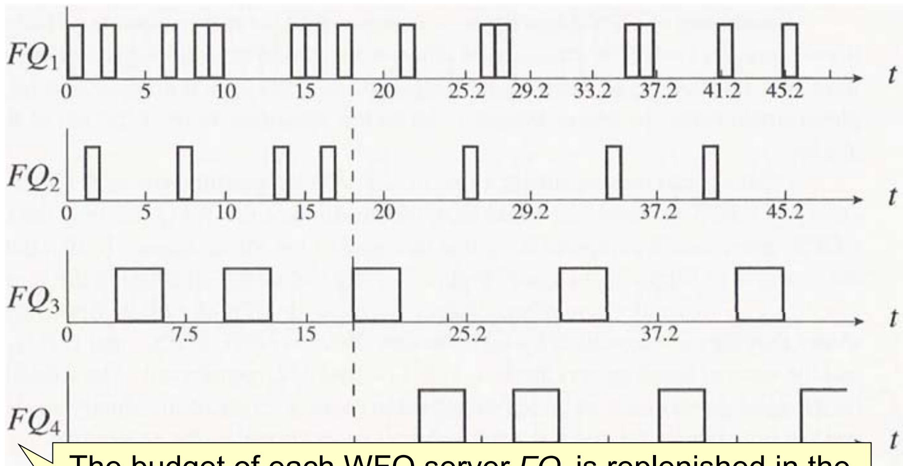
Starting from 18, all four servers being backlogged, each server now attains 1/4, 1/8, 1/4, and 3/8 of available time, respectively. This is why the first three servers take an additional 3.2, 3.2, and 7.2 units of time to complete the jobs at the head of their respective queues. (How?) These jobs are completed at 21.2, 21.2, and 25.2, respectively. Afterwards, each server GPS_i completes each job in A_i in 4, 8, 12, and 8 units of time, for i = 1, 2, 3, 4, respectively.

The sizes of GPS1, GPS2, GPS3, and GPS4 are 1/4, 1/8, 1/4, and 3/8.

Behavior of WFQ Servers (1/8)

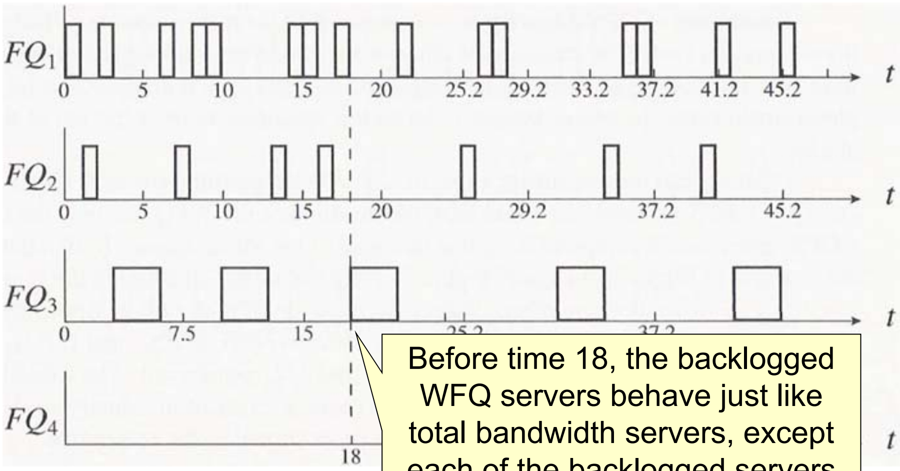


Behavior of WFQ Servers (2/8)



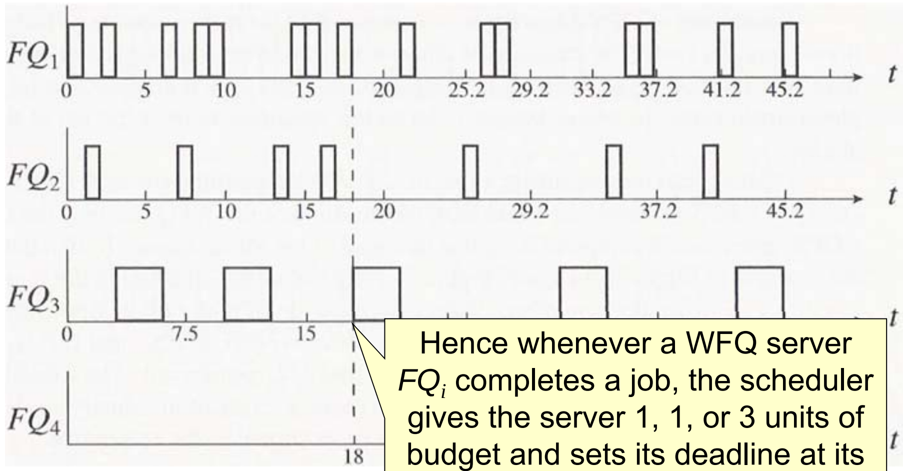
The budget of each WFQ server FQ_i is replenished in the same manner as the corresponding total bandwidth server, except for the way the server deadline is computed.

Behavior of WFQ Servers (3/8)



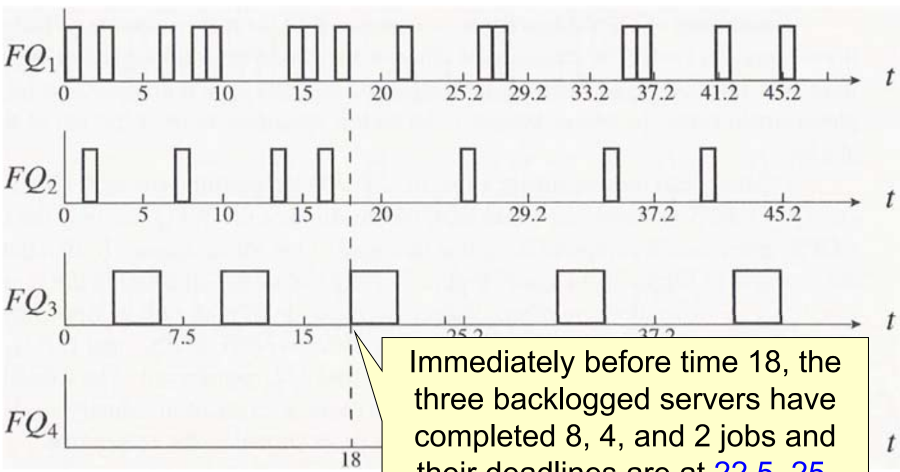
Before time 18, the backlogged WFQ servers behave just like total bandwidth servers, except each of the backlogged servers gets 2/5, 1/5, and 2/5 of processor time, respectively.

Behavior of WFQ Servers (4/8)



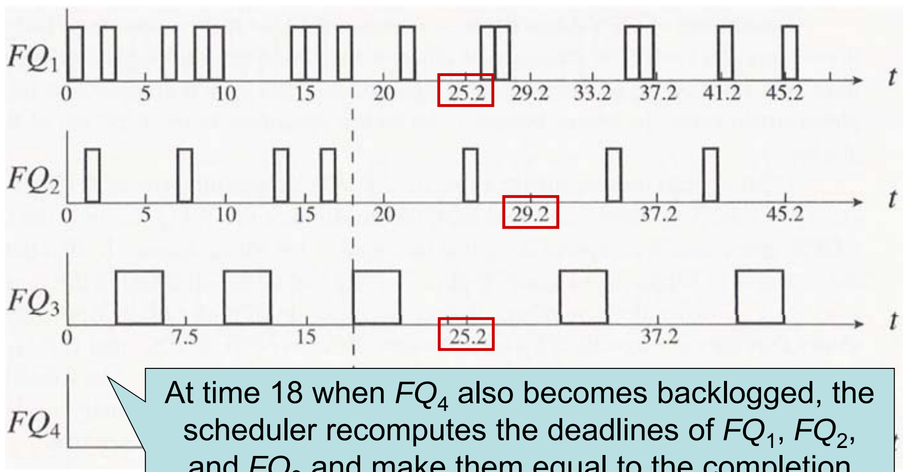
Hence whenever a WFQ server FQ_i completes a job, the scheduler gives the server 1, 1, or 3 units of budget and sets its deadline at its current deadline plus 2.5, 5, or 7.5 for i = 1, 2, 3, respectively. (Why?)

Behavior of WFQ Servers (5/8)



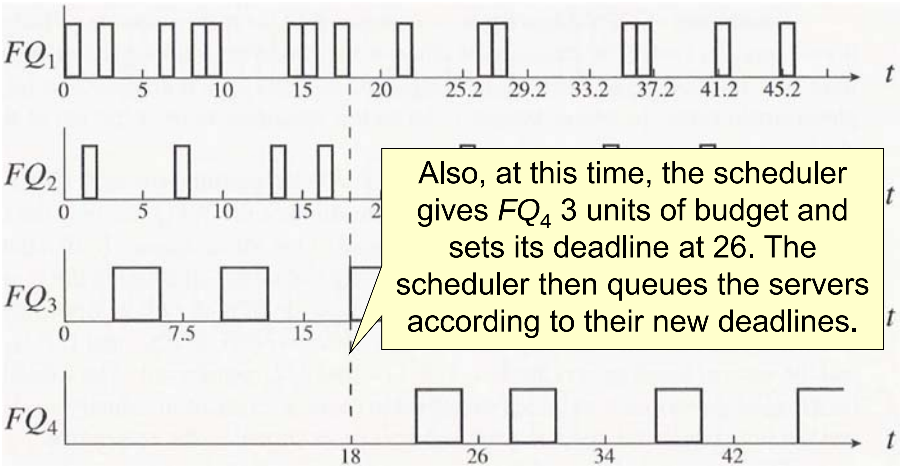
Immediately before time 18, the three backlogged servers have completed 8, 4, and 2 jobs and their deadlines are at 22.5, 25, and 22.5, respectively. (Why?)

Behavior of WFQ Servers (6/8)



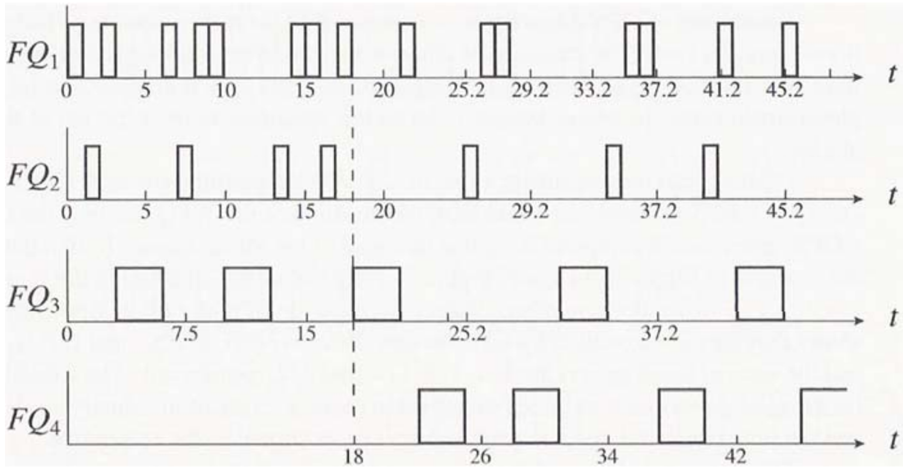
At time 18 when FQ_4 also becomes backlogged, the scheduler recomputes the deadlines of FQ_1, FQ_2, and FQ_3 and make them equal to the completion times of the 9th job, 5th job, and 3rd job of corresponding tasks according to the GPS schedule.

Behavior of WFQ Servers (7/8)



Also, at this time, the scheduler gives FQ_4 3 units of budget and sets its deadline at 26. The scheduler then queues the servers according to their new deadlines.

Behavior of WFQ Servers (8/8)



After time 18, the WFQ servers behave just like total bandwidth servers again.

Virtual Time vs. Real-Time (1/4)

- If the scheduler were to replenish server budget in the manner illustrated by the previous example, it would have to recompute the deadlines of all backlogged servers whenever some server changes from idle to backlogged and vice versa.
- A "budget replenishment" in a packet switch corresponds to the scheduler giving a ready packet a time stamp (i.e., a deadline) and inserting the packet in the outgoing queue stored in order of packet time stamps.

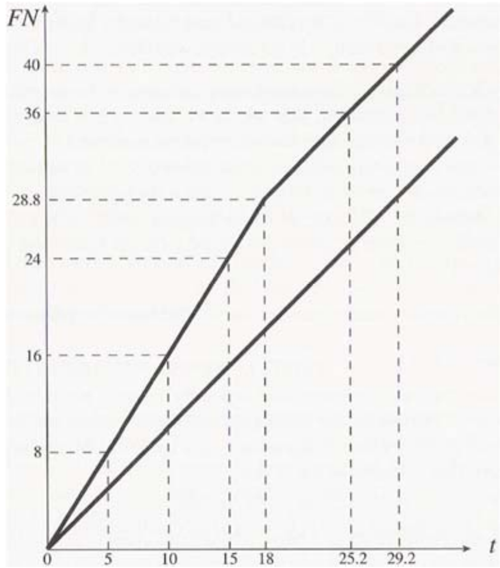
Virtual Time vs. Real-Time (2/4)

- To compute a new deadline and time stamp again of an already queued packet would be unacceptable, from the standpoint of both scheduling overhead and switch complexity.
- Fortunately, this recomputation of server deadlines is not necessary if instead of giving servers deadlines measured in real time, the scheduler gives servers virtual-time deadlines, called finish numbers.

Virtual Time vs. Real-Time (3/4)

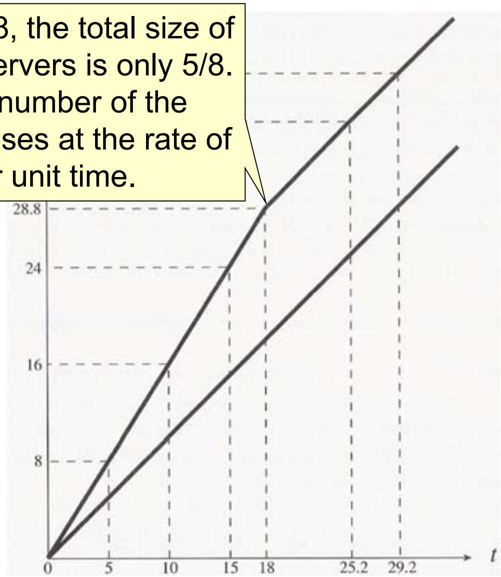
- The finish number of a server gives the number of the round in which the server budget would be exhausted if the backlogged servers were scheduled according to the GPS algorithm.

Mapping from Real-Time to Virtual Time



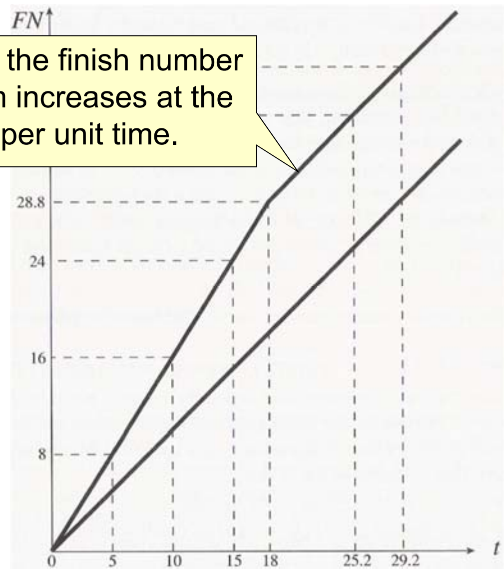
Mapping from Real-Time to Virtual Time

Before time 18, the total size of backlogged servers is only 5/8. The finish number of the system increases at the rate of 8/5 per unit time.



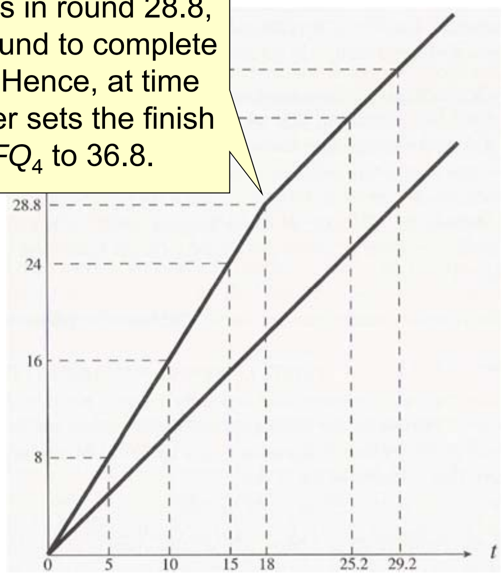
Mapping from Real-Time to Virtual Time

After time 18, the finish number of the system increases at the rate of 1 per unit time.



ime to Virtual Time

Take A_4 as an example: At time 18, the system is in round 28.8, and it takes 8 round to complete each job in A_4. Hence, at time 18, the scheduler sets the finish number of FQ_4 to 36.8.



Virtual Time vs. Real-Time (4/4)

- An alternative is for the scheduler to give each server a finish number each time it replenishes the server budget.
- It then schedule eligible servers according to their finish numbers; the smaller the finish number, the higher the priority.

Rules of Preemptive WFQ Algorithm (1/4)

- *Scheduling Rule*:
  - A WFQ server is ready for execution when it has budget and a finish number.
  - The scheduler assigns priorities to ready WFQ servers based on their finish numbers: the smaller the finish number, the higher the priority.
- *Consumption Rule*:
  - A WFQ server consumes its budget only when it executes.

Notations

- t is the length of time interval from the beginning of the current system busy interval to the current time.
- f_n_i denotes the finish number of the server FQ_i, e_i its budget and $\tilde{u}_i$ its size. e denotes the execution time of the job at the head of the server queue.
- U_b denotes the total size of all backlogged servers at t, and F_N denotes the finish number of system at time t. t_{-1} denotes the previous time when F_N and U_b were updated.

Rules of Preemptive WFQ Algorithm (2/4)

- *Initialization Rules*:
  - **I1**: For as long as all servers (and hence the system) are idle, $F\_N = 0$, $U_b = 0$, and $t_{-1} = 0$. The budget and finish numbers of every server are 0.
  - **I2**: When the first job with execution time e arrives at queue of some server FQ_k and starts a busy interval of the system,
    - a) $t_{-1} = t$, and increment U_b by $\tilde{u}_k$, and
    - b) set the budget e_k of FQ_k to e and its finish number f_n_k to $e/\tilde{u}_k$.

Rules of Preemptive WFQ Algorithm (3/4)

- *Rules for Updating F_N and Replenishing Budget of FQ_i during a System Busy Interval*
  - **R1**: When a job arrives at the queue of FQ_i, if FQ_i was idle immediately priori to this interval,
    - a) increment system finish number F_N by $(t - t_{-1})/\mathrm{U_b}$,
    - b) $t_{-1} = t$, and increment U_b by $\tilde{u}_i$, and
    - c) set budget e_i of FQ_i to e and its finish number f_n_i to $F\_N + e/\tilde{u}_k$ and place the server in the ready server queue in order of nonincreasing finish numbers.

Rules of Preemptive WFQ Algorithm (4/4)

- **R2**: Whenever FQ_i completes a job, remove the job from the queue of FQ_i,
  - a) if the server remains backlogged, set server budget e_i to e and increment its finish number by $e/\tilde{u}_i$.
  - b) if the server becomes idle, update U_b and F_N as follows:
    - 1) Increment the system finish number F_N by $(t - t_{-1})/U_b$.
    - 2) $t_{-1} = t$ and decrement U_b by $\tilde{u}_i$.

Preemptive WFQ Algorithm

- In summary, the scheduler updates the finish number F_N of the system and the total size U_b of all backlogged servers each time an idle server becomes backlogged or a backlogged server becomes idle.

Example

- Suppose in the previous example (pp. 206), server FQ_1 were to become idle at time 37 and later at time 55 become backlogged again.
  - At time 37, t_{-1} is 18, the value of F_N computed at 18 is 28.8, and the U_b in the time interval (18, 37] is 1. Following rule R2b, F_N is incremented by 37 – 18 = 19 and hence becomes 47.8. U_b becomes 3/4 starting from 37, and t_{-1} = 37.
  - At time 55 when FQ_1 becomes backlogged again, the new value of F_N, U_b and t_{-1} computed according to rule R1 are 47.8 + (55 – 37)/0.75 = 71.8, 1, and 55, respectively.

Outline

- Assumptions and Approaches
- Deferrable Servers
- Sporadic Servers
- Constant Utilization, Total Bandwidth, and Weighted Fair-Queueing Servers
- *Scheduling of Sporadic Jobs*

Scheduling of Sporadic Jobs

- The scheduler performs an acceptance test on each sporadic job upon its arrival.
- In a deadline-driven system, sporadic jobs are scheduled with periodic jobs on the EDF basis.
- In a fixed-priority system, sporadic jobs are executed by a bandwidth preserving server.

Terminologies

- We refer to each individual sporadic job as S_i.
- $S_i(t, d, e)$: The sporadic job is released at time t and has maximum execution time e and (absolute) deadline d.
- An acceptance test is optimal if it accepts a sporadic job if and only if the sporadic job can be feasibly scheduled without causing periodic jobs or sporadic jobs in the system to miss their deadlines.

Outline

- Scheduling of Sporadic Jobs
  1. ***A Simple Acceptance Test in Deadline-Driven Systems***
  2. A Simple Acceptance Test in Fixed-Priority Systems
  3. Integrated Scheduling of Periodic, Sporadic, and Aperiodic Tasks

Theoretical Basis

- In a deadline-driven system where the total density of all the periodic tasks is Δ, all the accepted sporadic jobs can meet their deadlines as long as the total density of all the active sporadic jobs is no greater than $1 - \Delta$ at all times.

Acceptance Test Procedure (1/4)

First Sporadic Job $S(t, d, e)$:

- The scheduler accepts S if its density $e/(d-t)$ is no greater than $1 - \Delta$.
- If the scheduler accepts the job, the (absolute) deadline d of S divides the time after t into two disjoint time intervals: the interval I_1 at and before d and the interval I_2 after d.
  - The job S is active in the former but not in the later.
  - The total densities $\Delta_{s,1}$ and $\Delta_{s,2}$ of the active jobs in these two intervals are equal to $e/(d-t)$ and 0, respectively.

Acceptance Test Procedure (2/4)

General Case:

- At time t when the scheduler does an acceptance test on $S(t, d, e)$, there are n_s active sporadic jobs in the system.
- The scheduler maintains a non-decreasing list of (absolute) deadlines of these sporadic jobs.
- These deadlines partition the time interval from t to infinity into $n_s + 1$ disjoint intervals: $I_1, I_2, \ldots, I_{n_s+1}$.
  - I_1 begins at t and ends at the first deadline in the list.
  - For $1 \le k \le n_s$, each subsequent interval I_{k+1} begins when the previous interval I_k ends and ends at the next deadline in the list or, in the case of I_{n_s+1}, at infinity.

Acceptance Test Procedure (3/4)

– Some of these interval have zero length when the sporadic jobs have nondistinct deadlines.

- The scheduler also keeps up-to-date the total density $\Delta_{s,k}$ of the sporadic jobs that are active during each of these intervals.
- Let I_l be the time interval containing the deadline d of the new sporadic job $S(t, d, e)$. Based on Theorem 4, the scheduler accepts the job S if

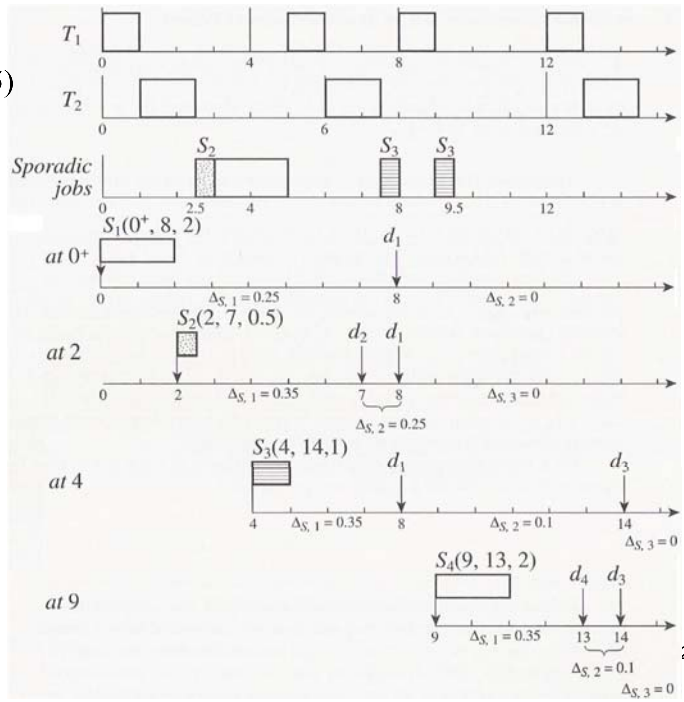
$$\frac{e}{d-t} + \Delta_{s,k} \le 1 - \Delta \qquad (1)$$

for all $k = 1, 2, \ldots, l$.

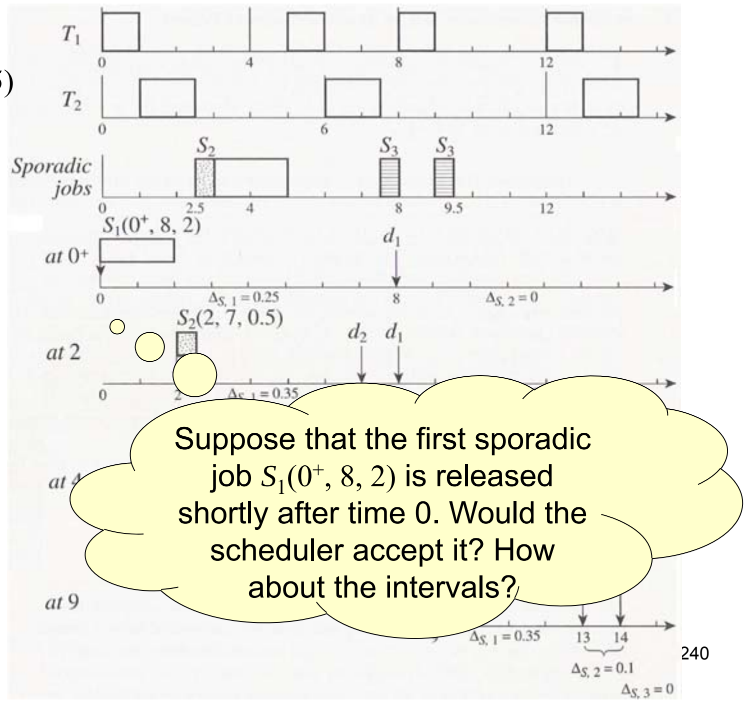
Acceptance Test Procedure (4/4)

- If these conditions are satisfied and S is accepted, the scheduler divides the interval I_l into two intervals: The first half of I_l ends at d, and the second half of I_l begins immediately after d.
- We now call the second half I_{l+1} and rename the subsequent intervals $I_{l+2}, \ldots, I_{n_s+2}$.
- The scheduler increments the total density $\Delta_{s,k}$ of all active sporadic jobs in each of the intervals I_1, I_2, …, I_l by the density $e/(d-t)$ of the new job.

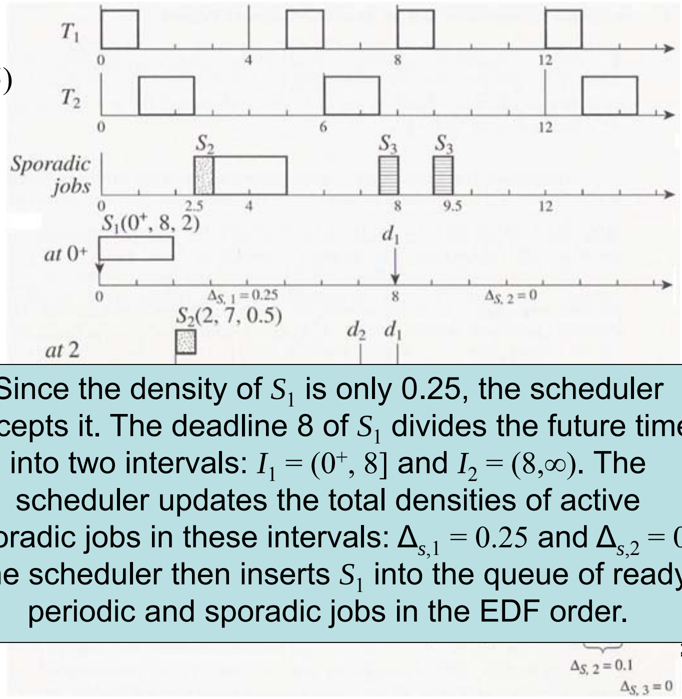
$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



$T_1 = (4, 1)$
$T_2 = (6, 1.5)$

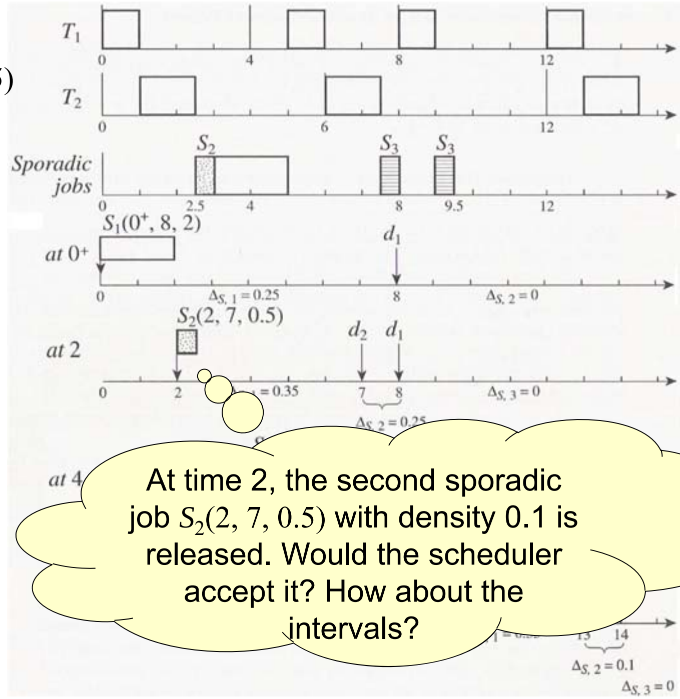


$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



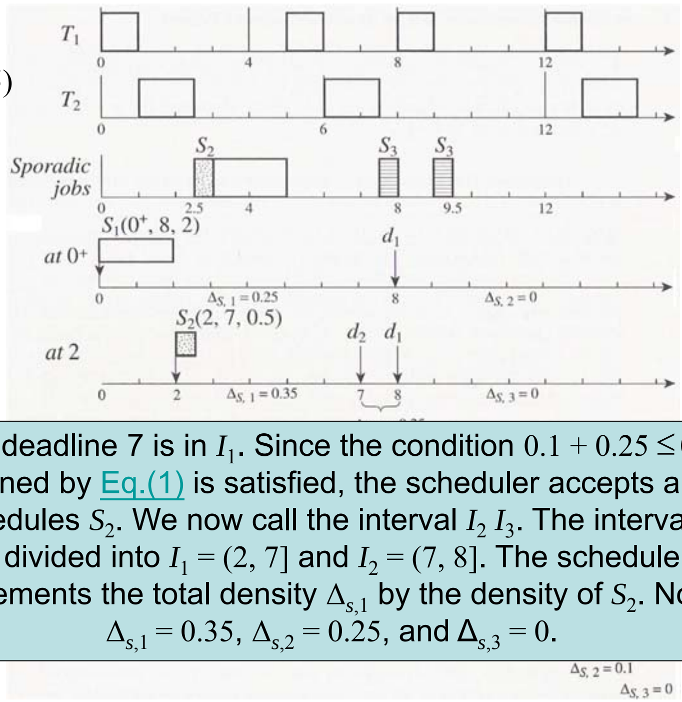
Since the density of S_1 is only 0.25, the scheduler accepts it. The deadline 8 of S_1 divides the future time into two intervals: $I_1 = (0^+, 8]$ and $I_2 = (8,\infty)$. The scheduler updates the total densities of active sporadic jobs in these intervals: $\Delta_{s,1} = 0.25$ and $\Delta_{s,2} = 0$. The scheduler then inserts S_1 into the queue of ready periodic and sporadic jobs in the EDF order.

$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



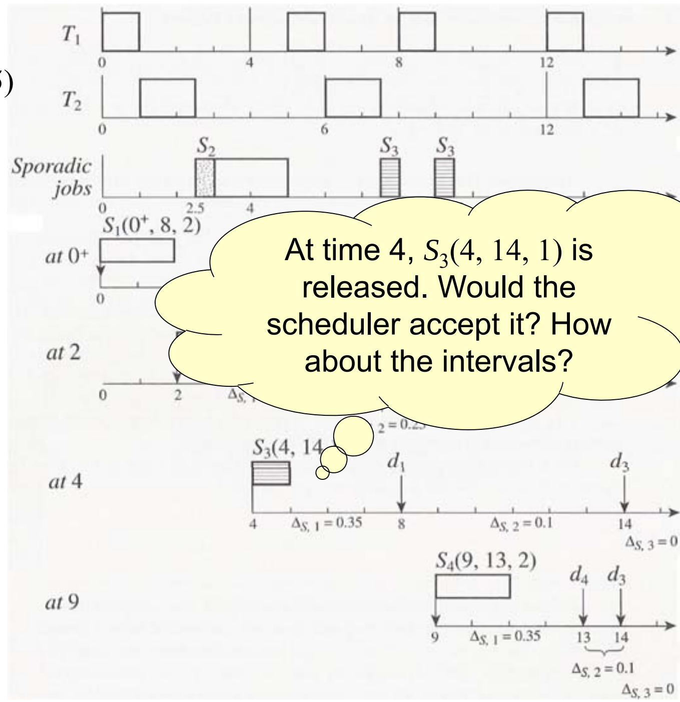
At time 2, the second sporadic job $S_2(2, 7, 0.5)$ with density 0.1 is released. Would the scheduler accept it? How about the intervals?

$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



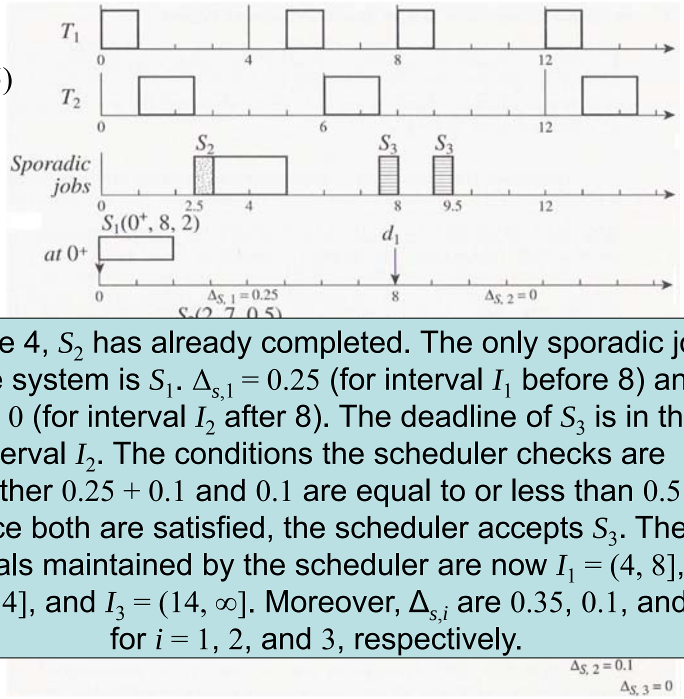
S_2's deadline 7 is in I_1. Since the condition $0.1 + 0.25 \leq 0.5$ defined by Eq.(1) is satisfied, the scheduler accepts and schedules S_2. We now call the interval I_2 I_3. The interval I_1 is divided into $I_1 = (2, 7]$ and $I_2 = (7, 8]$. The scheduler increments the total density $\Delta_{s,1}$ by the density of S_2. Now, $\Delta_{s,1} = 0.35$, $\Delta_{s,2} = 0.25$, and $\Delta_{s,3} = 0$.

$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



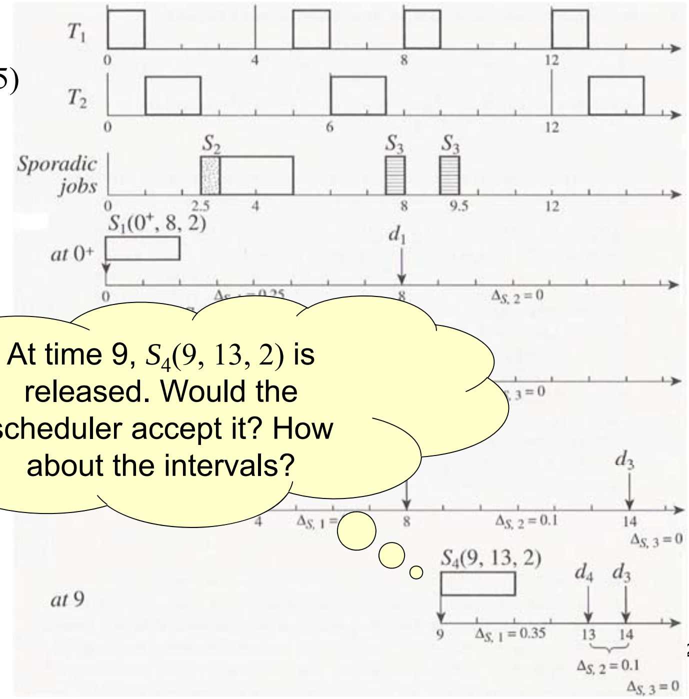
At time 4, $S_3(4, 14, 1)$ is released. Would the scheduler accept it? How about the intervals?

$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



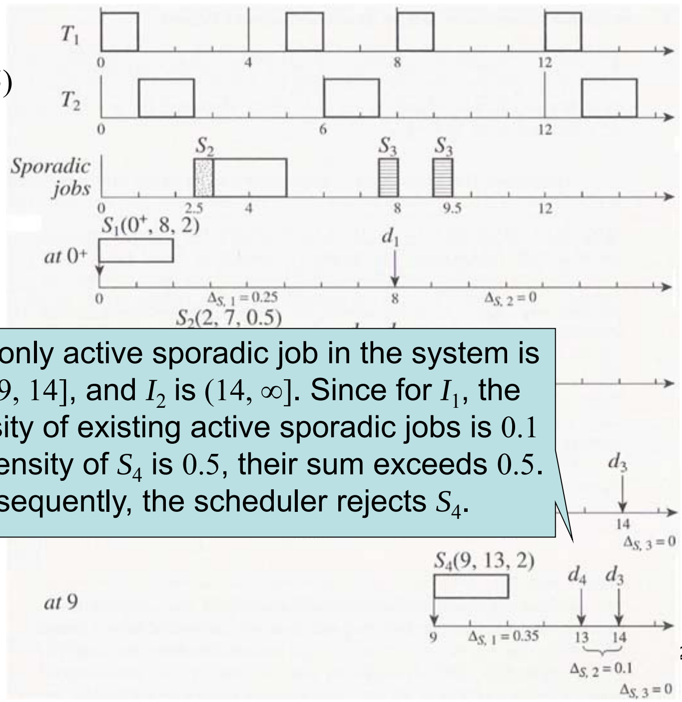
At time 4, S_2 has already completed. The only sporadic job in the system is S_1. $\Delta_{s,1} = 0.25$ (for interval I_1 before 8) and $\Delta_{s,2} = 0$ (for interval I_2 after 8). The deadline of S_3 is in the interval I_2. The conditions the scheduler checks are whether $0.25 + 0.1$ and 0.1 are equal to or less than 0.5. Since both are satisfied, the scheduler accepts S_3. The intervals maintained by the scheduler are now $I_1 = (4, 8]$, $I_2 = (8, 14]$, and $I_3 = (14, \infty]$. Moreover, $\Delta_{s,i}$ are 0.35, 0.1, and 0 for $i = 1, 2,$ and 3, respectively.

$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



At time 9, $S_4(9, 13, 2)$ is released. Would the scheduler accept it? How about the intervals?

$T_1 = (4, 1)$
$T_2 = (6, 1.5)$



Now, the only active sporadic job in the system is S_3. I_1 is $(9, 14]$, and I_2 is $(14, \infty]$. Since for I_1, the total density of existing active sporadic jobs is 0.1 and the density of S_4 is 0.5, their sum exceeds 0.5. Consequently, the scheduler rejects S_4.

Enhancement and Generalization

- Two points are worthy of observing:
  1. The acceptance test is not optimal, meaning that a sporadic job may be rejected while it is schedulable.
     - The schedulability condition given by Theorem 4 is sufficient but not necessary.
     - In the example above, the system becomes idle at time 9.5 and does not become busy again until 12. So, S_4 is acceptable.
  2. The acceptance test described above assumes that every sporadic job is ready for execution upon its arrival.
     - In general, its feasible interval may begin some time after its acceptance test time.

Enhancement for The 1st Point

- An enhancement is to have the scheduler also compute the slack of the system.
  - Suppose that the scheduler were to compute the slack of the system dynamically at time 9: The current busy interval of the periodic tasks and accepted sporadic jobs ends at time 9.5 and the next busy interval does not begin until time 12.
  - ➡ This leaves the system with 2.5 units of slack before time 13, the deadline of S_4. Hence, S_4 is schedulable.
  - ➡ The shortcoming of an acceptance test that makes use of dynamic-slack computation is its high run-time overhead.

Generalization of The 2nd Point

- We can modify the simple acceptance test in a straightforward fashion to accommodate arbitrary ready times of sporadic jobs.
  - The ready times and deadlines of sporadic jobs in the system partition the future time into disjoint time intervals.
  - The scheduler maintains information on these intervals and the total densities of active sporadic jobs in them in a similar manner, but now it may have to maintain as many as $2N_s + 1$ intervals, where N_s is the maximum number of sporadic jobs in the system.

Outline

- Scheduling of Sporadic Jobs
  1. A Simple Acceptance Test in Deadline-Driven Systems
  2. ***A Simple Acceptance Test in Fixed-Priority Systems***
  3. Integrated Scheduling of Periodic, Sporadic, and Aperiodic Tasks

A Simple Acceptance Test in Fixed Priority Systems (1/5)

- 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.
- ➡ This leads to a simple acceptance test.

A Simple Acceptance Test in Fixed Priority Systems (2/5)

- When the first sporadic job $S_1(t, d_{s,1}, e_{s,1})$ arrives, the server has at least $\lfloor (d_{s,1} - t) / p_s \rfloor \times e_s$ units of processor time before the deadline of the job.
- The scheduler accepts S_1 if the slack of the job

$$\sigma_{s,1}(t) = \lfloor (d_{s,1} - t) / p_s \rfloor \times e_s - e_{s,1}$$

is larger than or equal to 0.

A Simple Acceptance Test in Fixed Priority Systems (3/5)

- To decide whether a new job $S_i(t, d_{s,i}, e_{s,i})$ is acceptable when there are n_s sporadic jobs in the system, the scheduler computes the slack $\sigma_{s,i}$ of S_i according to

$$\sigma_{s,i}(t) = \lfloor (d_{s,i} - t) / p_s \rfloor \times e_s - e_{s,i} - \sum_{d_{s,k} < d_{s,i}} \left(e_{s,k} - \xi_{s,k} \right)$$

where $\xi_{s,k}$ is the execution time of the completed portion of the existing sporadic job S_k.
- The new job S_i cannot be accepted if its slack is less than 0.

A Simple Acceptance Test in Fixed Priority Systems (4/5)

- If $\sigma_{s,i}(t)$ is no less than 0, the scheduler then checks whether any existing sporadic job S_k whose deadline is after $d_{s,i}$ may be adversely affected by the acceptance of S_i.
  - This can easily be done by checking whether the slack $\sigma_{s,k}(t)$ of S_k at the time is equal to or larger than $e_{s,i}$.
- The scheduler accepts S_i if $\sigma_{s,k}(t) - e_{s,i} \geq 0$ for every existing sporadic job S_k with deadline equal to or later than $d_{s,i}$.

A Simple Acceptance Test in Fixed Priority Systems (5/5)

- If the scheduler accepts S_i, it stores $\sigma_{s,i}(t)$ for later use and decrements the slack of every sporadic job with a deadline equal to or later than $d_{s,i}$ by the execution time $e_{s,i}$ of the new job.

Outline

- Scheduling of Sporadic Jobs
  1. A Simple Acceptance Test in Deadline-Driven Systems
  2. A Simple Acceptance Test in Fixed-Priority Systems
  3. ***Integrated Scheduling of Periodic, Sporadic, and Aperiodic Tasks***

Integrated Scheduling

- In principle, we can schedule sporadic and aperiodic tasks together with periodic tasks according to either bandwidth-preserving server approach or the slack-stealing approach, or both.
  - If we steal slack from sporadic and periodic jobs in order to speed up the completion of aperiodic jobs, some sporadic jobs may not be acceptable later while they may be acceptable if no slack is used.
  - Both the implementation and validation of the system are straightforward when we use bandwidth-preserving servers to execute aperiodic or sporadic jobs.