Priority Driven Scheduling of Aperiodic and Sporadic Tasks (1)

Real-Time and Embedded Systems (M)

Lecture 7

UNIVERSITY
of
GLASGOW



Lecture Outline

- Assumptions, definitions and system model
- Simple approaches
  - Background, interrupt-driven and polled execution
  - Periodic servers
- Bandwidth-preserving servers
  - Deferrable server
  - Sporadic server
  - …

The material in lectures 7 & 8 corresponds to (most of) chapter 7 of Liu's book

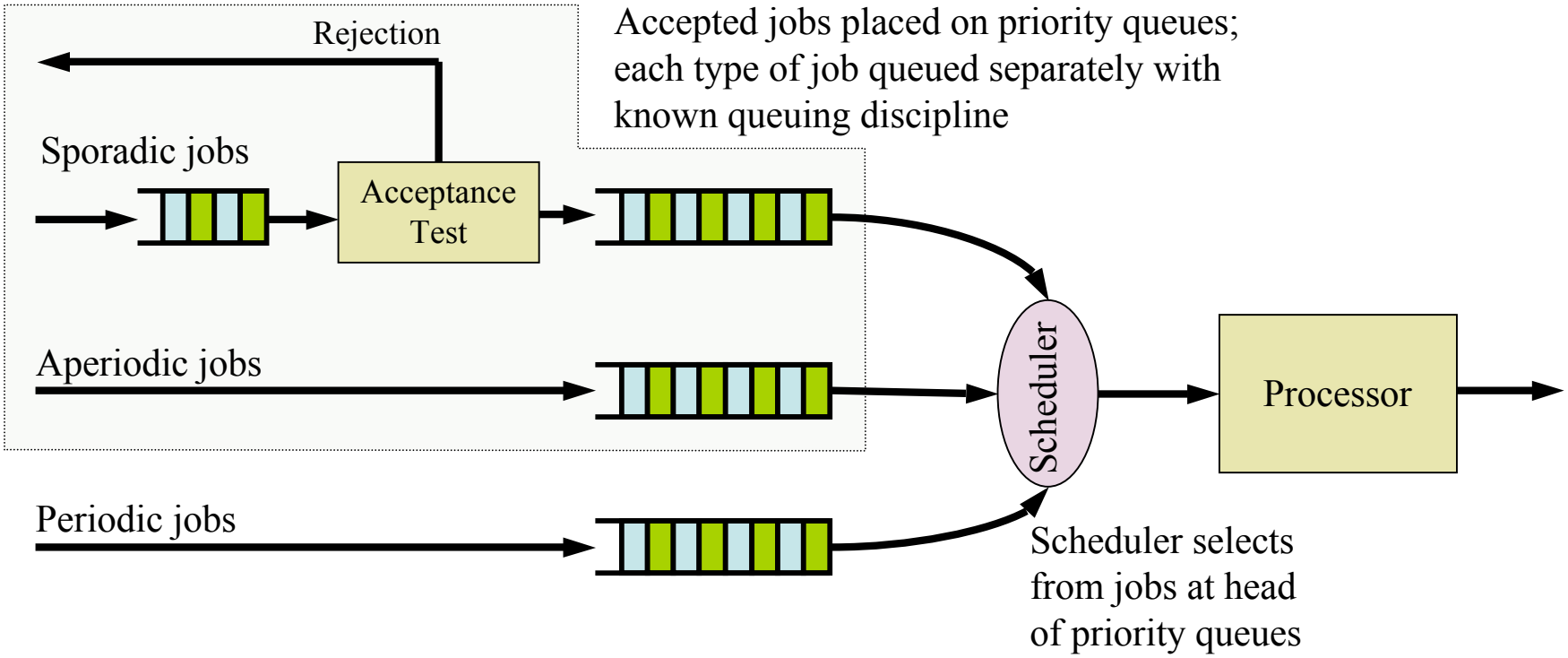
Copyright © 2006 University of Glasgow

Assumptions

- One processor system; independent preemptable periodic tasks scheduling using a priority-driven algorithm
  - Parameters of all periodic tasks are known
  - In the absence of aperiodic and sporadic jobs, periodic tasks meet deadlines
- Aperiodic and/or sporadic jobs exist
  - They are independent of each other, and of the periodic tasks
  - They can be preempted at any time

Copyright © 2006 University of Glasgow

System Model



Copyright © 2006 University of Glasgow

The Scheduling Problem

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

- Aim to complete each aperiodic job as soon as possible, without causing periodic tasks or accepted sporadic jobs to miss deadlines
  - Aperiodic jobs are always accepted

Copyright © 2006 University of Glasgow

Definitions: Correctness

- A *correct schedule* is one where all periodic tasks, and any sporadic tasks that have been *accepted*, meet their deadlines
- A scheduling algorithm supporting aperiodic and/or sporadic jobs is a *correct* algorithm if it only produces correct schedules for the system

Copyright © 2006 University of Glasgow

Definitions: Optimality

- An aperiodic job scheduling algorithm is *optimal* if it minimizes either:
  - The response time of the job at the head of the aperiodic job queue
  - The average response time of all aperiodic jobs for a given queuing discipline

- A sporadic job scheduling algorithm is *optimal* if it accepts a new sporadic job, and schedules that job to complete by its deadline, if and only if the new job can be correctly scheduled to complete in time
  - An optimal algorithm always produces a feasible schedule if it accepts a job
  - Note: that this is different from the definition of optimal on-line algorithms discussed previously, as that definition required that *all* offered sporadic jobs had to be accepted and completed in time

Copyright © 2006 University of Glasgow

Scheduling Aperiodic Jobs

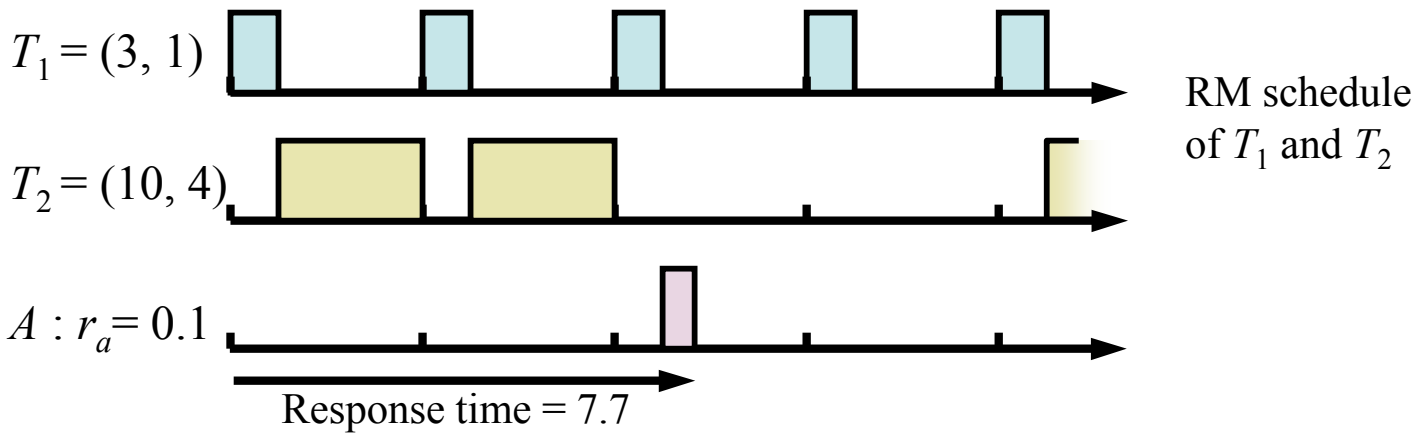
- Consider the simple case: scheduling aperiodic jobs along with a system of periodic jobs
  - Ignore sporadic jobs for now

- Two simple approaches:
  - Execute the aperiodic jobs in the background
  - Execute the aperiodic jobs by interrupting the periodic jobs

Copyright © 2006 University of Glasgow

Background Scheduling of Aperiodic Jobs

- Aperiodic jobs are scheduled and executed only at times when there are no periodic or sporadic jobs ready for execution
- Advantages
  - Clearly produces *correct* schedules
  - Extremely simple to implement
- Disadvantages
  - Not *optimal* since it is almost guaranteed to delay execution of aperiodic jobs in favour of periodic and sporadic jobs, giving unduly long response times for the aperiodic jobs



Copyright © 2006 University of Glasgow

Interrupt Scheduling of Aperiodic Jobs

- How can we improve the response time for aperiodic jobs?
- Whenever an aperiodic job arrives, the execution of periodic tasks is interrupted, and the aperiodic job is executed.
- Advantages
  - Reduces response times of aperiodic jobs
- Disadvantages
  - The algorithm is likely *not correct*, and will often cause periodic/sporadic tasks in the system to miss some deadlines

Copyright © 2006 University of Glasgow

Slack Stealing for Aperiodic Jobs

- Neither background or interrupt driven execution ideal
- A better alternative is *slack stealing*
  - Periodic jobs are be scheduled to complete before their deadline; there may be some *slack time* between completion of the periodic job and its deadline
  - Since we know the execution time of periodic jobs, can move the slack time earlier in the schedule, running periodic jobs 'just in time' to meet their deadlines
  - Execute aperiodic jobs in the slack time, ahead of periodic jobs
- Reduces response time for aperiodic jobs and is correct, but more complex and difficult to reason about

- Outlined operation for clock-driven systems in lecture 4; the book covers slack stealing in priority-driven systems
  - Conceptually simple, but computing available slack is difficult in practice for priority-driven systems

Copyright © 2006 University of Glasgow

Polled Execution for Aperiodic Jobs

- Another common way to schedule aperiodic jobs is using a *polling server*
  - A periodic job (p_s, e_s) scheduled according to the usual periodic algorithm *as the highest priority job*
  - When executed, it examines the aperiodic job queue
    - If an aperiodic job is in the queue, it is executed for up to e_s time units
    - If the aperiodic queue is empty, the polling server self-suspends, giving up its execution slot
    - The server does not wake-up once it has self-suspended, aperiodic jobs which become active during a period are not considered for execution until the next period begins

- Simple to prove correctness, performance less than ideal – since execute aperiodic jobs in particular timeslots – can we improve?
  - Yes, this is but the simplest *periodic-server* for aperiodic jobs

Copyright © 2006 University of Glasgow

Periodic Servers

- A task that behaves much like a periodic task, but is created for the purpose of executing aperiodic jobs, is a *periodic server*
  - A periodic server, $T_{PS} = (p_{PS}, e_{PS})$ never executes for more than e_{PS} units of time in any time interval of length p_{PS}
    - The parameter e_{PS} is called the *execution budget* (or simply *budget*) of the periodic server
    - When a server is scheduled and executes aperiodic jobs, it *consumes* its budget at the rate of 1 per unit time; the budget has been *exhausted* when it reaches 0
    - A time instant when the budget is replenished is called a *replenishment time*
  - A periodic server is *backlogged* whenever the aperiodic job queue is nonempty; it is *idle* if the queue is empty
  - The periodic server is scheduled as any other periodic task based upon the priority scheme used by the scheduling algorithm
    - Except: the server is eligible for execution *only* when scheduled *and when it is backlogged and has non-zero budget*

Copyright © 2006 University of Glasgow

Periodic Servers

- Different kinds of periodic server differ in how the budget is consumed when idle, and when the budget is replenished

- A polling server is a simple kind of periodic server
  - The budget is replenished to e_s at the beginning of each period
  - The budget is immediately consumed if there is no work when the server is scheduled

Copyright © 2006 University of Glasgow

Bandwidth-Preserving Servers

- A deficiency of the polling server algorithm is that if the server is scheduled when it is not backlogged, it loses its execution budget until it is replenished when it is next released
  - An aperiodic job arriving just after the polling server has been scheduled and found the aperiodic job queue empty will have to wait until the next replenishment time

- Want to *preserve* execution budget of the server when it finds an empty queue, to execute an aperiodic job that arrives later in the period, if doing so will not affect the correctness of the schedule

- Algorithms that improve the polling approach in this manner are called *bandwidth-preserving server* algorithms

[Often used for scheduling network packets; hence the name]

Copyright © 2006 University of Glasgow

Bandwidth-Preserving Servers

- Bandwidth-preserving servers are periodic servers with additional rules for *consumption* and *replenishment* of their budget
- How do such servers work?
  - A backlogged bandwidth-preserving server is ready for execution when it has budget
  - The scheduler keeps track of the consumption of the budget and suspends the server when its is exhausted, or the server becomes idle
  - The scheduler moves the server back to the ready queue once it replenishes its budget, if the server is backlogged at that time
  - If arrival of an aperiodic job causes the server to become backlogged, and it has budget, the server is put back on the ready queue
    - This last rule overcomes limitation of polling server

Copyright © 2006 University of Glasgow

Bandwidth-Preserving Servers

- Many types of bandwidth-preserving server:
  - Deferrable servers
  - Sporadic servers
  - Constant utilization servers
  - Total bandwidth servers
  - Weighted fair queuing servers
  - …

Copyright © 2006 University of Glasgow

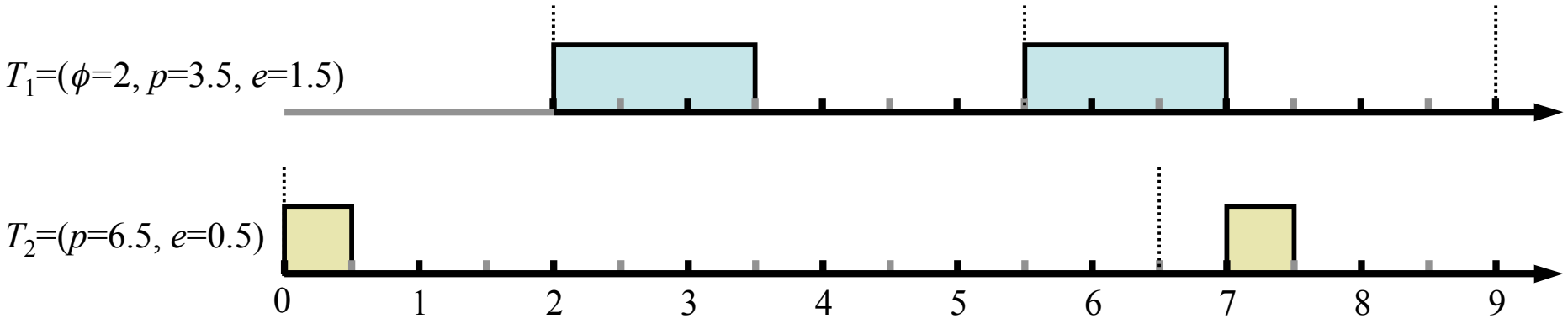
Deferrable Server

- The simplest bandwidth-preserving server
  - Improves response time of aperiodic jobs, compared to polling server

- Consumption rule:
  - The budget is consumed at the rate of one per unit time whenever the server executes
  - Unused budget is retained throughout the period, to be used whenever there are aperiodic jobs to execute
    - Instead of discarding the budget if no aperiodic job to execute at start of period, keep in the hope a job arrives
- Replenishment rule:
  - The budget is set to e_S at multiples of the period
    - i.e. time instants $k \cdot p_S$, for $k = 0, 1, 2, \ldots$
  - Note: the server is not allowed to carry over budget from period to period

Copyright © 2006 University of Glasgow

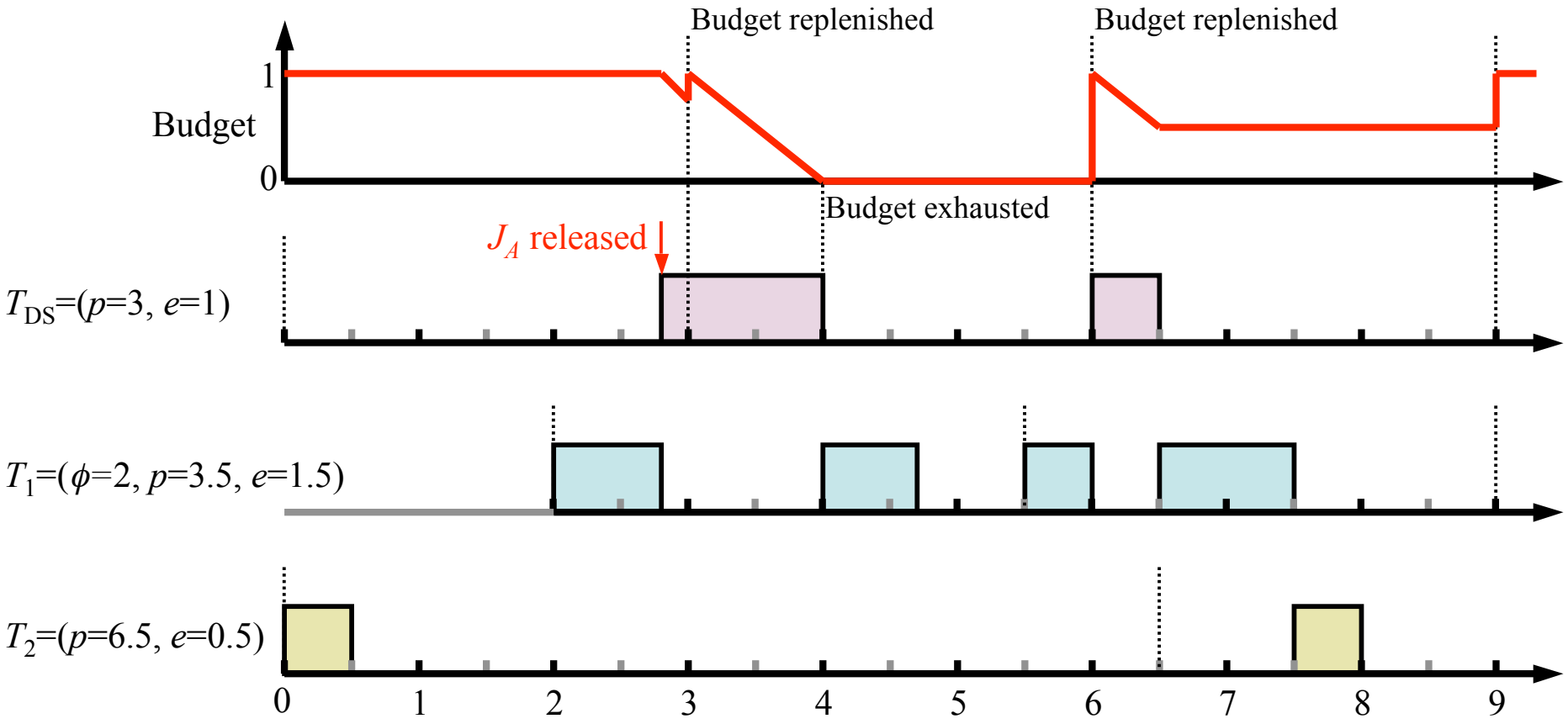
Deferrable Server: Example



Periodic tasks T_1 and T_2 are scheduled according to the rate-monotonic algorithm

Copyright © 2006 University of Glasgow

Deferrable Server: Example



Periodic tasks T_1 and T_2 are scheduled according to the rate-monotonic algorithm
T_{DS} is a deferrable server executing aperiodic job J_A with e_A=1.7, r_A=2.8

Copyright © 2006 University of Glasgow

Deferrable Server: Schedulability (1)

- In general, to determine schedulability of a *fixed-priority* system with a deferrable server, find a critical instant and conduct time-demand analysis as usual

- Theorem: In a fixed-priority system T in which $D_i \leq p_i$ for all i, with a deferrable server (p_S, e_S) with the highest priority among all tasks, a critical instant of every periodic tasks T_i occurs at a time t_0 when all of the following are true:
  - One of its jobs $J_{i,c}$ is released at t_0
  - A job in every higher-priority periodic task is released at t_0
  - The budget of the server is e_S at t_0, one or more aperiodic jobs are released at t_0, and they keep the server backlogged hereafter
  - The next replenishment time of the server is $t_0 + e_S$

[Follows from the theorem about critical instants; lecture 6]

Copyright © 2006 University of Glasgow

Deferrable Server: Schedulability (1)

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

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

Execution time of job J_i

Execution time of higher priority jobs started during this interval

Execution time of the deferrable server

  - To determine whether the task T_i is schedulable, we simply have to check whether $w_i(t) \le t$ for some $t \le D_i$.
  - Remember, this is a sufficient condition, not necessary – i.e. if this condition is not true, the system may not be schedulable

Copyright © 2006 University of Glasgow

Deferrable Server: Schedulability (1)

- In general, no maximum schedulable utilization test to determine schedulability for a fixed-priority system with a deferrable server
- One special case:
  - 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 $p_n > p_s + e_s$, where the relative deadlines are equal to their respective periods, can be scheduled rate-monotonically with a deferrable server provided that the utilization $U < U_{RM/DS}(n)$ where:

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

Copyright © 2006 University of Glasgow

Deferrable Server: Schedulability (2)

- It is easier to reason about the schedulability of a *deadline-driven* system with a deferrable server
  - The deadline of a deferrable server is its next replenishment time
  - A periodic task T_i in a system of N independent, preemptable, periodic tasks is schedulable with a deferrable server with period p_S, execution budget e_S and utilization u_S, according to the EDF algorithm if:

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

  - *Must* be calculated for each task in the system, since D_i included

- Example:
  - Tasks T_1=(3, 0.6), T_2=(5.0, 0.5), T_3=(7, 14) scheduled with a deferrable server p_s=4, e_s=0.8
  - The left-hand side of the above inequality is 0.913, 0.828 and 0.792 respectively; hence the three tasks are schedulable

Copyright © 2006 University of Glasgow

Sporadic Servers

- Limitation of deferrable servers – they may delay lower-priority tasks for more time than a periodic task with the same period and execution time
- The *sporadic server* is designed to eliminate this limitation. Its consumption and replenishment rules ensure that a sporadic server with period p_S and budget e_S never demands more processor time than a periodic task with the same parameters

[Details in lecture 8]

Copyright © 2006 University of Glasgow

Summary

- Assumptions, definitions and system model
- Simple approaches
  - Background, interrupt-driven and polled execution
  - Periodic servers
- Bandwidth-preserving servers
  - Deferrable server
  - Sporadic server
  - …

Should have an initial understanding of how to schedule aperiodic jobs in a more optimal manner

Copyright © 2006 University of Glasgow