Priority Driven Scheduling of Aperiodic and Sporadic Tasks (2)

Real-Time and Embedded Systems (M)

Lecture 8

UNIVERSITY
of
GLASGOW



Lecture Outline

- Scheduling aperiodic jobs (cont'd)
  - Review: deferrable servers
  - Sporadic servers
  - Constant utilization servers
  - Total bandwidth servers
  - Weighted fair queuing servers
- Scheduling sporadic jobs

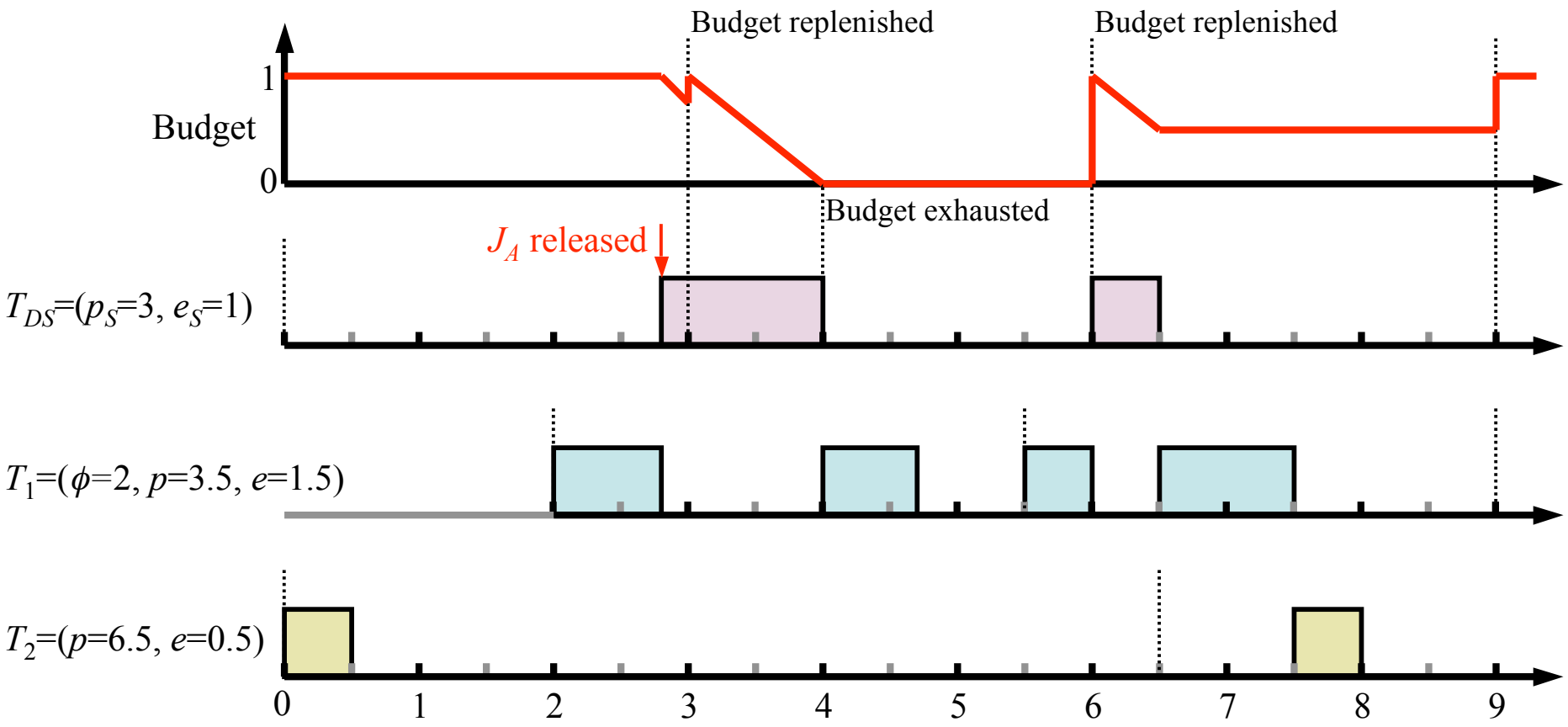
Copyright © 2006 University of Glasgow

Review: Scheduling Aperiodic Tasks

- Lecture 7 introduced the scheduling problem for aperiodic jobs:
  - Aim to complete each aperiodic jobs as soon as possible, without causing periodic tasks or accepted sporadic jobs to miss deadlines
- Simple approaches to scheduling aperiodic jobs not sufficient:
  - Background server is correct, but unduly delays aperiodic jobs
  - Interrupt driven server is (typically) not correct
- Two more complex approaches offer better performance:
  - Slack stealing
  - Periodic servers:
    - Polling server
      - Simple, provably correct, provides a guaranteed fraction of the processor for scheduling aperiodic jobs, but sometimes gives poor response time
    - Deferrable server
      - Improves on the response time of the polling server, maintains its advantages

Copyright © 2006 University of Glasgow

Review: Deferrable Server



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

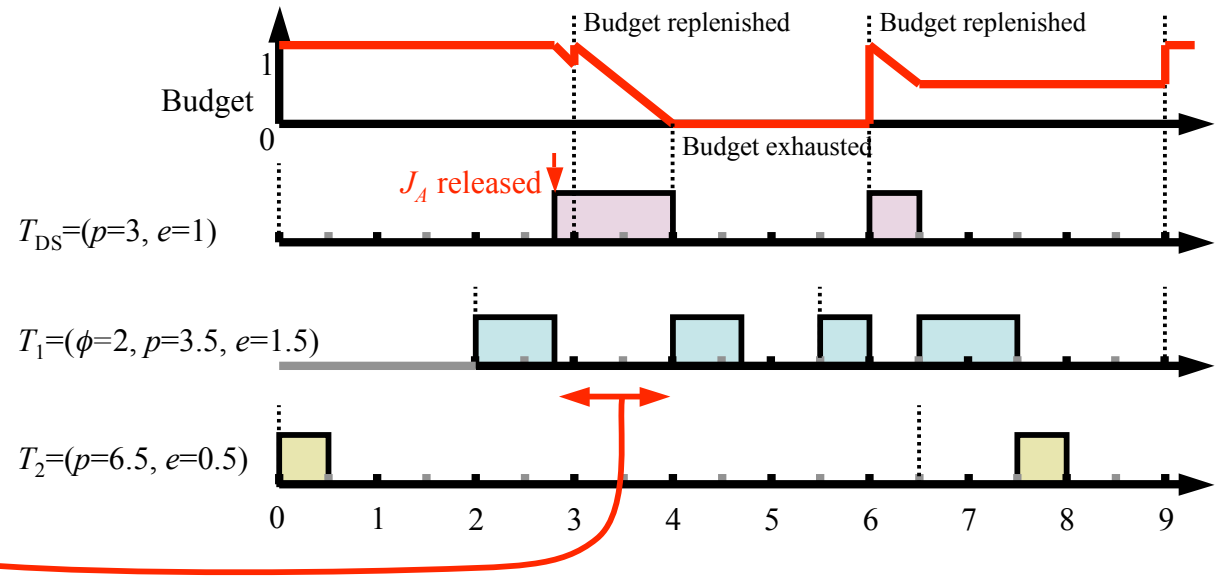
T_{DS} is a deferrable server executing aperiodic job J_A with e_A=1.7, r_A=2.8, which is scheduled as the highest priority task (i.e. *not* rate monotonically)

⇒ This may affect the schedulability of the system (hence results on the maximum schedulable utilization last lecture)

Copyright © 2006 University of Glasgow

Limitations of Deferrable 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:



T_1 blocked for 1.2 units although execution time of the deferrable server is only 1.0 units (with period 3 units)

- The *sporadic server* is designed to eliminate this limitation.
  - A different type of bandwidth preserving server
  - More complex 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

Copyright © 2006 University of Glasgow

A Simple Fixed-Priority Sporadic Server

- Consider a system T of N independent preemptable periodic tasks, plus a single sporadic server task with parameters (p_s, e_s)
  - Tasks are scheduled using a fixed-priority algorithm; system is known to be schedulable if it's assumed that (p_s, e_s) behaves as a standard periodic task

- Definitions:
  - T_H is the subset of periodic tasks with higher priorities than the server
    - That subset may be *idle* when no job in T_H is ready for execution, or *busy*
  - Define t_r as the last time the server budget replenished
  - Define t_f as the first instant after t_r at which the server begins to execute
  - At any time t define:
    - *BEGIN* as the start of the earliest busy interval in the most recent contiguous sequence of busy intervals of T_H starting before t
      - Busy intervals are contiguous if the later one starts immediately the earlier one ends
    - *END* as the end of the latest busy interval in this sequence, if this interval ends before t; define $END = \infty$ if the interval ends after t

Copyright © 2006 University of Glasgow

A Simple Fixed-Priority Sporadic Server

- Consumption rule:
  - At any time t after t_r, if the server has budget and if either of the following two conditions is true, the server's budget is consumed at the rate of 1 per unit time:

    C1: The server is executing

    C2: The server has executed since t_r and $END < t$
  - When they are not true, the server holds its budget

- That is:
  - The server executes for no more time than it has execution budget
  - The server retains its budget if:
    - A higher-priority job is executing, or
    - It has not executed since t_r
  - Otherwise, the budget decreases when the server executes, or if it idles while it has budget

Copyright © 2006 University of Glasgow

A Simple Fixed-Priority Sporadic Server

- Replenishment rules

  R1: When system begins executing, and each time budget is replenished, set the budget to e_S and t_r = the current time.

  R2: When server begins to execute (defined as time t_f)
  if $END = t_f$ then
  $t_e = \max(t_r, BEGIN)$
  else if $END < t_f$ then
  $t_e = t_f$
  The next replenishment time is set to $t_e + p_S$.

  t_e = the *effective replenishment time*

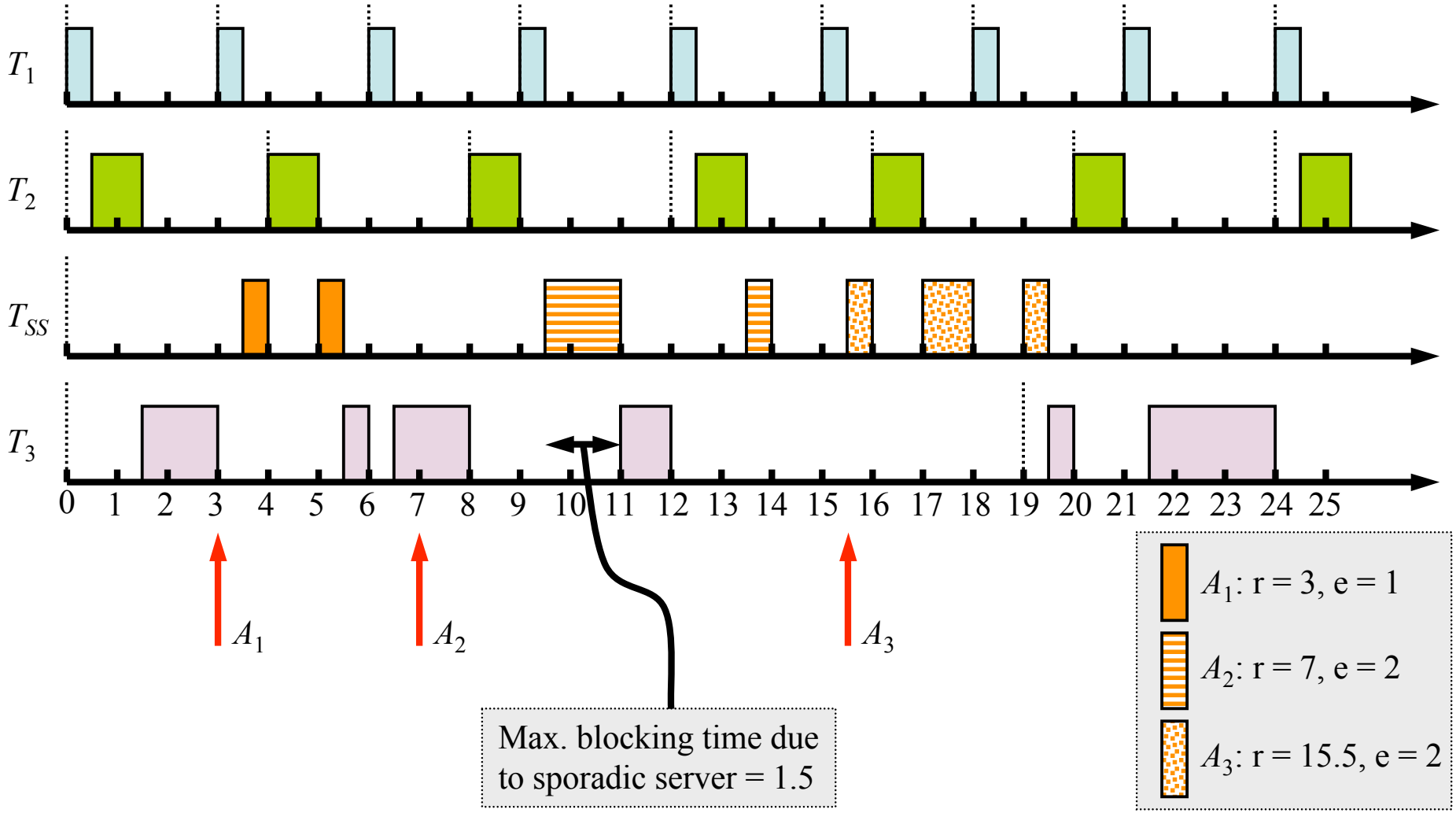
  R3: The next replenishment occurs at the next replenishment time ($= t_e + p_S$), except under the following conditions:

  (a) If $t_e + p_S$ is earlier than t_f the budget is replenished as soon as it is exhausted

  (b) If T becomes idle before $t_e + p_S$, and becomes busy again at t_b, the budget is replenished at $\min(t_b, t_e + p_S)$

Copyright © 2006 University of Glasgow

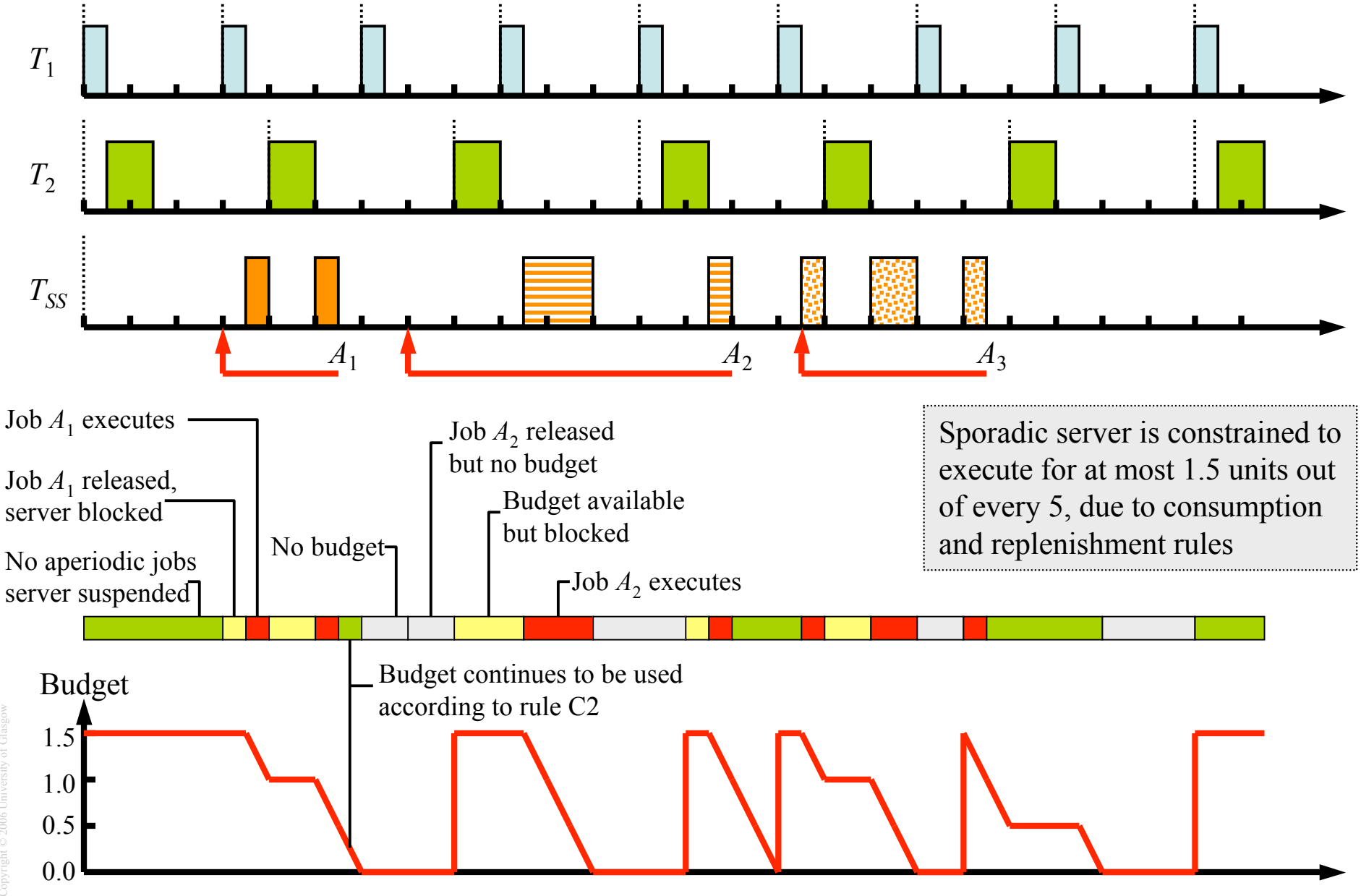
Example: Fixed-Priority Sporadic Server



T_1=(3, 0.5), T_2=(4, 1.0), T_3=(19, 4.5), T_{ss}=(5, 1.5)
Rate monotonic schedule; simple sporadic server

Copyright © 2006 University of Glasgow

Example: Fixed-Priority Sporadic Server



Sporadic server is constrained to execute for at most 1.5 units out of every 5, due to consumption and replenishment rules

Copyright © 2006 University of Glasgow

A Simple Fixed-Priority Sporadic Server

- A sporadic server is more complex than a polling server or a deferrable server
  - Consumption and replenishment rules require keeping track of a lot of data, several cases to consider when making scheduling decisions
- This complexity is acceptable, because schedulability of a sporadic server is *much* easier to demonstrate

- Theorem: for the purpose of validating schedulability, you can treat a simple sporadic server (p_s, e_s) in a fixed-priority system exactly the same as any other task T_i with $p_i=p_s$ and $e_i=e_s$
  - The actual inter-release times of the sporadic server will sometimes be greater than p_s, and their execution times less than e_s, but this does not affect correctness

Copyright © 2006 University of Glasgow

Other Fixed-Priority Sporadic Servers

- It is possible to replenish the budget of a sporadic server more aggressively, and to preserve it for longer, than does a simple sporadic server
- Further improves response time of aperiodic jobs, at the price of more complex consumption and replenishment rules, and higher scheduling overhead

- Examples:
  - A *sporadic/background server* which claims all background time, in addition to the time claimed by the periodic component of the server
  - A *cumulative replenishment server* which keeps any remaining budget at the end of each period for use in following periods
  - …

  Unclear if the complexity of these variants is worthwhile…

Copyright © 2006 University of Glasgow

Simple Dynamic-Priority Sporadic Server

- It is possible to define a simple sporadic server to operate in a dynamic-priority environment
  - E.g. when using EDF or LST scheduling

- Consumption and replenishment rules are conceptually similar to those for a fixed-priority scheduler, with minor modifications that account for the difference in scheduling algorithm
  - [See book for details]

- Provides same schedulability guarantees as the simple sporadic server for fixed-priority schedulers
  - A simple sporadic server (p_s, e_s) in an EDF or LST system can be treated exactly the same as any other task T_i with $p_i=p_s$ and $e_i=e_s$ when performing schedulability analysis

Copyright © 2006 University of Glasgow

Other Bandwidth Preserving Servers

- Now consider three other bandwidth preserving server algorithms:
  - Constant utilization server
  - Total bandwidth server
  - Weighted fair queuing server

- All are approximations to an ideal *generalised processor sharing* algorithm
  - Aim is to assign a portion of the available processor time to a task, making it believe it was executing on a slower processor, independent of any other tasks
  - Aiming to provide fair sharing, timing isolation, or guaranteed throughput
  - Widely used in network scheduling, but can also be used to schedule servers for aperiodic jobs

Copyright © 2006 University of Glasgow

Constant Utilization Server

- A constant utilization server reserves a known fraction, $\tilde{u}_s$, of the processor time for execution of the server
- Like other bandwidth preserving servers, it has a budget and is defined in terms of consumption and replenishment rules
- When the budget is non-zero, the server is scheduled with other tasks on an EDF basis
  - The budget and deadline of the server are chosen such that the utilization of the server is constant when it executes, and that it is always given enough budget to complete the job at the head of its queue each time its budget is replenished
  - The server never has any budget if it has no work to do

Copyright © 2006 University of Glasgow

Constant Utilization Server

- Consumption rule:
  - A constant utilization server only consumes budget when it executes
- Replenishment rules:
  - Initially, budget $e_s = 0$ and deadline $d = 0$
  - When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue
    - If $t < d$, do nothing ($\Rightarrow$ server is busy; wait for it to become idle)
    - If $t \geq d$ then set $d = t + e/\tilde{u}_s$ and $e_s = e$
  - At the deadline d of the server
    - If the server is backlogged, set $d = d + e/\tilde{u}_s$ and $e_s = e$ ($\Rightarrow$ was busy when job arrived)
    - If the server is idle, do nothing
  - The server is always given enough budget to complete the job at the head of its queue, with known utilization, when the budget is replenished

Copyright © 2006 University of Glasgow

Total Bandwidth Server

- A constant utilization server gives a known fraction of processor capacity to a task; but cannot claim unused capacity to complete the task earlier
- A *total bandwidth* server improves responsiveness by allowing a server to claim background time not used by the periodic tasks
  - Change the replenishment rules slightly, leave all else the same:
    - Initially, $e_s = 0$ and $d = 0$
    - When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue
      - Set $d = \max(d, t) + e/\tilde{u}_s$ and $e_s = e$
    - When the server completes the current aperiodic job, the job is removed from the queue and
      - If the server is backlogged, set $d = d + e/\tilde{u}_s$ and $e_s = e$
      - If the server is idle, do nothing
  - Always ready for execution when backlogged
- Will assign *at least* fraction $\tilde{u}_s$ of the processor to a task

Copyright © 2006 University of Glasgow

Weighted Fair Queuing Server

- Aim of the constant utilization and total bandwidth servers is to assign some fraction of processor capacity to a task
- When assigning capacity there is the issue of *fairness*:
  - A scheduling algorithm is *fair* within any particular time interval if the fraction of processor time in the interval attained by each backlogged server is proportional to the server size
    - Not only do all tasks meet their deadline, but they all make continual progress according to their share of the processor, no *starvation*
  - Constant utilization and total bandwidth servers are fair on the long term, but can diverge significantly from fair shares in the short term
    - Total bandwidth server partly by design, since it uses background time, but also has fairness issues when there is no spare background time

- As we discuss in lecture 17, the *weighted fair queuing* algorithm can also be used to share processor time between servers, and is designed to ensure fairness in allocations

Copyright © 2006 University of Glasgow

Scheduling Sporadic Jobs

- Have focussed considerable effort on improving response time of aperiodic jobs
- Now turn to the problem of scheduling sporadic jobs alongside a system of periodic tasks and aperiodic jobs

- Recall the sporadic job 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

Copyright © 2006 University of Glasgow

Model for Scheduling Sporadic Jobs

- When sporadic jobs arrive, they are both accepted and scheduled in EDF order
  - In a dynamic-priority system, this is the natural order of execution
  - In a fixed-priority system, the sporadic jobs are executed by a bandwidth preserving server, which performs an acceptance test and runs the sporadic jobs in EDF order
  - In both cases, no new scheduling algorithm is required

- Definitions:
  - Sporadic jobs are denoted by $S_i(r_i, d_i, e_i)$ where r_i is the release time, d_i is the (absolute) deadline, and e_i is the maximum execution time
  - The density of a sporadic job $\Delta_i = e_i/(d_i - r_i)$
    - The total density of a system of n jobs is $\Delta = \Delta_1 + \Delta_2 + \ldots + \Delta_n$
  - The job is active during its feasible interval $(r_i, d_i]$

Copyright © 2006 University of Glasgow

Sporadic Jobs in Dynamic-Priority Systems

- Theorem: 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 ≤ 1 at all times
  - This is the standard schedulability test for EDF systems, but including both periodic and sporadic jobs
  - This test uses the *density* since deadlines may not equal periods; hence it is a sufficient test, but not a necessary test

- What does this mean?
  - If we can bound the frequency with which sporadic jobs appear to the running system, we can guarantee that none are missed
  - Alternatively, when a sporadic job arrives, if we deduce that the total density would exceed 1 in its feasible interval, we reject the sporadic job (admission control)

Copyright © 2006 University of Glasgow

Admission Control for Sporadic Jobs/EDF

- At time t there are n active sporadic jobs in the system
- The EDF scheduler maintains a list of the jobs, in non-decreasing order of deadline
  - The deadlines partition the time from t to ∞ into $n + 1$ discrete intervals: $I_1, I_2, \ldots, I_{n+1}$
    - I_1 begins at t and ends at the earliest sporadic job deadline
    - For each $1 \leq k \leq n$, each interval I_{k+1} begins when the interval I_k ends, and ends at the next deadline in the list (or ∞ for I_{n+1})
  - The scheduler maintains the total density $\Delta_{s,k}$ of each interval I_k
- Let I_l be the interval containing the deadline d of the new sporadic job $S(t, d, e)$
  - The scheduler accepts the job if $\frac{e}{d-t} + \Delta_{s,k} \leq 1 - \Delta$ for all k=1, 2, …, l
  - i.e. accept if the new sporadic job can be added, without increasing the density of any intervals past 1

Copyright © 2006 University of Glasgow

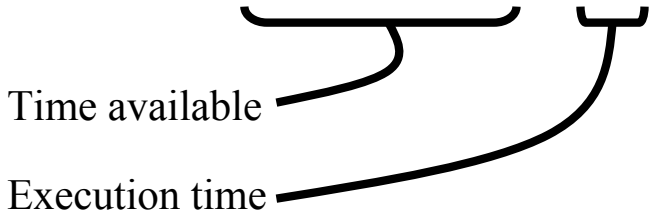
Admission Control for Sporadic Jobs/EDF

- Notes:
  - This acceptance test is *not* optimal: a sporadic job may be rejected even though it could be scheduled
    - The result for the schedulable utilization is based on the *density* and hence is sufficient but not necessary
    - It is possible to derive a – much more complex – expression for schedulability which takes into account slack time, and is optimal. Unclear if the complexity is worthwhile.
  - This acceptance test assumes every sporadic jobs is ready for execution when released
    - If this is not the case, must modify the acceptance test to take into account the time when the jobs become ready, rather than their release time, when testing the intervals to see if their density exceeds 1

Copyright © 2006 University of Glasgow

Sporadic Jobs in Fixed-Priority Systems

- One way to schedule sporadic jobs in a fixed-priority system is to use a sporadic server to execute them
- Because the server (p_s, e_s) has e_s units of processor time every p_s units of time, the scheduler can compute the least amount of time available to every sporadic job in the system
  - Assume that sporadic jobs ordered among themselves in EDF
  - When first sporadic job $S_1(t, d_{s,1}, e_{s,1})$ arrives, there is at least $\lfloor (d_{s,1} - t)/p_s \rfloor \cdot e_s$ units of processor time *available to the server* before the deadline of the job
  - Therefore it accepts S_1 if the slack of the job $\sigma_{s,1}(t) = \lfloor (d_{s,1} - t) / p_s \rfloor e_s - e_{s,1} \geq 0$



[cont'd]

Copyright © 2006 University of Glasgow

Sporadic Jobs in Fixed-Priority Systems

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

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

  where $\xi_{s,k}$ is the execution time of the completed part of the existing job S_k The job cannot be accepted if $\sigma_{s,i}(t) < 0$

- If $\sigma_{s,i}(t) \geq 0$, the scheduler then checks if any existing sporadic job S_k with deadline after $d_{s,i}$ may be adversely affected by the acceptance of S_i
  - This is done by checking if the slack $\sigma_{s,k}(t)$ for each S_k at the time is at least equal to the execution time $e_{s,i}$ of S_i
  - i.e. the job S_i is accepted if $\sigma_{s,k}(t) - e_{s,i} \geq 0$ for every existing sporadic job S_k with deadline not less than $d_{s,i}$

- The acceptance test for fixed-priority systems is more complex than that for dynamic-priority systems, but is still of reasonable time complexity to be implemented "on-line"

Copyright © 2006 University of Glasgow

Summary

- Have discussed further:
  - Scheduling aperiodic jobs (cont'd)
    - Sporadic servers
    - Constant utilization servers
    - Total bandwidth servers
    - Weighted fair queuing servers
  - Scheduling sporadic jobs

- Tutorial on Tuesday will recap the material from lectures 7 and 8

- Problem set 3 now available: due at 5pm on 6th February

Copyright © 2006 University of Glasgow