Priority-driven Scheduling of Periodic Tasks

- Schedulability test for fixed-priority tasks
  - Since we cannot count on any particular relationships among the phases of tasks in a fixed-priority system, we must identify the worst-case combination of release times of any job $J_{i,c}$ in T_i and all the jobs that have higher priorities than $J_{i,c}$
  - This combination is the worst-case because the response time of $J_{i,c}$ released in such a situation is the largest possible of all combinations of release times
  - We, therefore, define a ***critical instant*** of a task T_i as a time instant such that:
    - The job in T_i released at that instant has the maximum response time of all jobs in T_i (if the response time of every job in T_i is equal to or less than the relative deadline D_i), and
    - The response time of the job released at that instant is greater than D_i if the response time of some jobs in T_i exceed D_i
  - The response time of a job in T_i released at a critical instant is called the ***maximum (possible) response time***

Priority-driven Scheduling of Periodic Tasks

- Theorem: In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant occurs when one of its jobs $J_{i,c}$ is released at the same time with a job from every higher-priority task.
- Why is this important? It turns out that our schedulability test for fixed-priority tasks will be based upon showing that a job $J_{i,c}$ released at a critical instant completes by its relative deadline, D_i – i.e. we don't have to simulate the entire system, we simply have to show that the system has the correct characteristics following a critical instant; in particular, if there are N tasks in the system, we have to show that $J_{N,c}$ completes by its relative deadline, D_N

Priority-driven Scheduling of Periodic Tasks

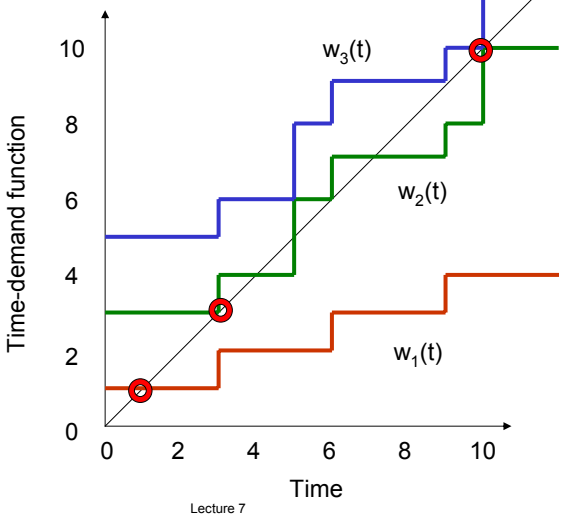
- Time-Demand Analysis
  - We first compute the total demand for processor time by a job released at a critical instant of the task and by all the higher-priority tasks as a function of time from the critical instant
  - We then check whether this demand can be met before the deadline of the job
  - Consider one task at a time, starting with T_1 (highest priority) working down to T_N (lowest priority)
  - To determine whether T_i is schedulable, after finding that all tasks with higher priorities are schedulable, we focus on a job in T_i that is released at time t_0, which is a critical instant of T_i.

Priority-driven Scheduling of Periodic Tasks

- Time-Demand Analysis (continued)
  - At time $t_0 + t$ for $t \geq 0$, the total (processor) time demand of this job and all the higher-priority jobs released in $[t_0, t]$ is given by $w_i(t) = e_i + [t/p_1]e_1 + [t/p_2]e_2 + \ldots + [t/p_{i-1}]e_{i-1}$ for $0 < t \leq p_i$
  - This job can meet its deadline of t_0+D_i if at some time t_0+t at or before the deadline, the supply of processor time, which is equal to t, becomes greater than or equal to $w_i(t)$
  - Since this job of T_i has the maximum possible response time of all jobs in T_i, we conclude that all jobs in T_i can meet their deadlines if this job can meet its deadline
  - $w_i(t)$ is called the time-demand function of the task T_i
  - This condition is sufficient, but not necessary; i.e. if $w_i(t) > t$ for all $0 < t \leq D_i$, we would have to try to establish that critical instants do NOT occur

Priority-driven Scheduling of Periodic Tasks

Rate Monotonic
T_1 = (, 3, 1)
T_2 = (, 5, 2)
T_3 = (, 10, 2)



Priority-driven Scheduling of Periodic Tasks

Time	Queue	Execute	Time	Queue	Execute
0	$J_{1,1}[1]; J_{2,1}[2]; J_{3,1}[2]$	$J_{1,1}$	16	$J_{2,4}[2]$	$J_{2,4}$
1	$J_{2,1}[2]; J_{3,1}[2]$	$J_{2,1}$	18	$J_{1,7}[1]$	$J_{1,7}$
3	$J_{1,2}[1]; J_{3,1}[2]$	$J_{1,2}$	19		
4	$J_{3,1}[2]$	$J_{3,1}$	20	$J_{2,5}[2]; J_{3,3}[2]$	$J_{2,5}$
5	$J_{2,2}[2]; J_{3,1}[1]$	$J_{2,2}$	21	$J_{1,8}[1]; J_{2,5}[1]; J_{3,3}[2]$	$J_{1,8}$
6	$J_{1,3}[1]; J_{2,2}[1]; J_{3,1}[1]$	$J_{1,3}$	22	$J_{2,5}[1]; J_{3,3}[2]$	$J_{2,5}$
7	$J_{2,2}[1]; J_{3,1}[1]$	$J_{2,2}$	23	$J_{3,3}[2]$	$J_{3,3}$
8	$J_{3,1}[1]$	$J_{3,1}$	24	$J_{1,9}[1]; J_{3,3}[1]$	$J_{1,9}$
9	$J_{1,4}[1]$	$J_{1,4}$	25	$J_{2,6}[2]; J_{3,3}[1]$	$J_{2,6}$
10	$J_{2,3}[2]; J_{3,2}[2]$	$J_{2,3}$	27	$J_{1,10}[1]; J_{3,3}[1]$	$J_{1,10}$
12	$J_{1,5}[1]; J_{3,2}[2]$	$J_{1,5}$	28	$J_{3,3}[1]$	$J_{3,3}$
13	$J_{3,2}[2]$	$J_{3,2}$	29		
15	$J_{1,6}[1]; J_{2,4}[2]$	$J_{1,6}$			

Priority-driven Scheduling of Periodic Tasks

- We can see that the time-demand function $w_i(t)$ is a staircase function
- The "steps" occur at integer multiples of the period for higher-priority tasks
- The value of $w_i(t) - t$ linearly decreases from a step until the next step
- If our primary interest is the schedulability of a task, it suffices to check whether $w_i(t) \leq t$ at the time instants when a higher-priority job is released
- Our schedulability test becomes:
  - Compute $w_i(t)$
  - Check whether $w_i(t) \leq t$ is satisfied for the following values of t
    $t = j*p_k$; $k = 1, 2, \ldots, i$; $j = 1, 2, \ldots, [\min(p_i, D_i)/p_k]$

Priority-driven Scheduling of Periodic Tasks

- Tick Scheduling
  - All of our previous discussion of priority-driven scheduling was driven by job release and job completion events
  - Alternatively, can perform priority-driven scheduling at periodic events (timer interrupts) generated by a hardware clock – i.e. tick (or time-based) scheduling
  - Additional factors to account for in schedulability analysis
    - The fact that a job is ready to execute will not be noticed and acted upon until the next clock interrupt (tick); this will delay the completion of the job
    - A ready job that is yet to be noticed by the scheduler must be held somewhere other than the ready job queue, the pending job queue
    - When the scheduler executes, it moves jobs in the pending queue to the ready queue according to their priorities; once in the ready queue, the jobs execute in priority order

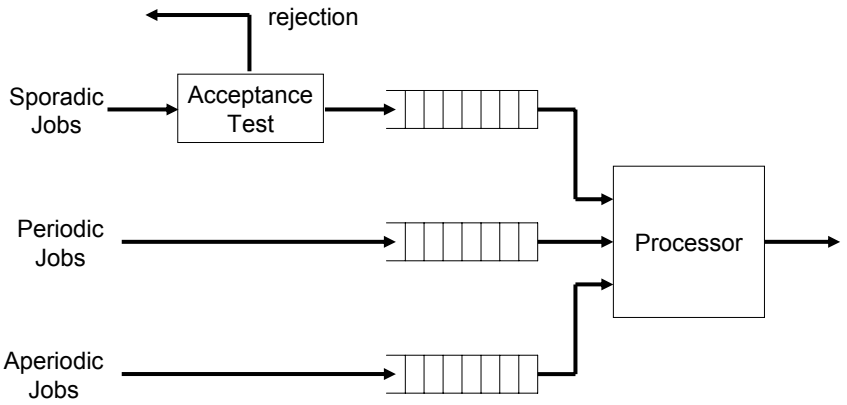
Priority-driven Scheduling of Periodic Tasks

- Tick Scheduling (continued)
  - Denote the tick size by p_0, which is the length of time between consecutive clock interrupts
  - We model the scheduler as a periodic task T_0 whose period is p_0; it is the highest priority task; its execution time e_0 is the time required to service the clock interrupt
  - We also have to account for the time required to move a job from the pending queue to the ready queue; call this time S_0 (for Staging time)
  - It is fairly obvious that the available utilization for tasks other than the tick scheduler is reduced by two factors:
    - $u_0 = e_0/p_0$
    - Within a hyperperiod, H, each task T_i will have released $[H/p_i]$ jobs; each job will have been staged from pending to ready queues; therefore, the total additional utilization consumed by staging is $S_0 * (H/p_1 + H/p_2 + \ldots + H/p_i)$

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Assumptions
  - One processor
  - Periodic tasks are independent
  - Aperiodic and sporadic jobs are independent of each other and of the periodic tasks
  - Every job can be preempted at any time
  - We are focussing on admission control of sporadic jobs – i.e. we will NOT accept a sporadic job if we cannot guarantee that it will meet its deadline
  - p_i and e_i are known for all periodic tasks
  - A priority-driven algorithm is used to schedule the system
  - In the absence of aperiodic and sporadic jobs, the periodic tasks meet all deadlines

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- More assumptions
  - The operating system maintains the priority queues shown in the previous slide
  - Ready periodic jobs are placed in the periodic task queue, ordered by the priorities assigned according to the selected scheduling algorithm
  - Each accepted sporadic job is assigned a priority and placed in a priority queue; it may not be the same as the periodic task queue
  - Newly arrived aperiodic jobs are placed in the aperiodic task queue
  - Newly arrived sporadic jobs are placed in a waiting queue to await acceptance
  - The scheduler is ONLY concerned with scheduling the jobs at the tops of the priority queues onto the processor according to the algorithm
  - The queueing discipline used to order aperoidic jobs among themselves is given

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Aperiodic and Sporadic Job Scheduling Algorithms
  - Need to solve the following problems:
    - 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 tasks or previously accepted sporadic jobs to miss their deadlines
    - Try to complete each aperiodic job as soon as possible, without causing periodic tasks and accepted sporadic jobs to miss their deadlines

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- More definitions
  - A ***correct schedule*** is one for which periodic and accepted sporadic tasks never miss their deadlines
  - An aperiodic or sporadic scheduling algorithm is ***correct*** if it produces only correct schedules of the system.
  - 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 OR the average response time of all aperiodic jobs for the given queueing discipline
  - A sporadic job scheduling algorithm (acceptance + scheduling) is optimal if it accepts each newly arrived sporadic job and schedules the job to complete by its deadline if and only if the new job can be correctly scheduled to complete in time by some means – 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

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

1. Background Approach
   - Aperiodic jobs are scheduled and executed only at times when there are no periodic or sporadic jobs ready for execution
   - Pros
     - Clearly produces correct schedules
     - Extremely simple to implement
   - Cons
     - Almost guaranteed to delay the execution of aperiodic jobs and to unnecessarily prolong their response times

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

2. Interrupt-Driven Execution Approach
   - Whenever an aperiodic job arrives, the execution of periodic tasks is interrupted, and the aperiodic job is executed.
   - Pros
     - Minimizes response times of aperiodic jobs
   - Cons
     - Periodic/sporadic tasks will probably miss some deadlines – i.e. the algorithm is not correct

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

3. Slack Stealing Approach
   - Whenever an aperiodic job arrives, the execution of periodic tasks is interrupted, and the aperiodic job is executed ONLY IF IT IS SAFE TO DO SO.
   - Need to keep track of the slack associated with each periodic/sporadic job; upon receipt of the interrupt, can schedule an aperiodic job if the periodic/sporadic jobs have slack > 0.
   - Pros
     - Minimizes response times of aperiodic jobs
     - Generates correct schedules
   - Cons
     - Much more complex algorithm, since we need to keep track of the slack in all jobs

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Polled Executions
  - A ***poller*** or ***polling server*** is a periodic task T_S with p_S as its polling period and e_S as its execution time.
  - When the poller executes, it examines the aperiodic job queue; if the queue is nonempty, it executes the job at the head of the queue.
  - 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.

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- More Definitions
  - A task that behaves more or less like a periodic task and is created for the purpose of executing aperiodic jobs is called a ***periodic server***.
  - A periodic server, $T_{PS} = (\varphi_{PS}, p_{PS}, e_{PS})$ 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.
  - The ratio $u_{PS} = e_{PS} / p_{PS}$ is the ***size*** of the periodic server.
  - A poller is a kind of periodic server; at the beginning of each period, the budget of the poller is set to e_S – i.e. its budget is ***replenished*** by e_S units
  - A time instant when the server budget is replenished is called a ***replenishment time***.

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Yet More Definitions
  - A periodic server is ***backlogged*** whenever the aperiodic job queue is nonempty
  - The server is ***idle*** if the queue is empty
  - The server is ***eligible*** (ready) ***for execution*** ONLY WHEN IT IS BACKLOGGED AND HAS BUDGET
  - The server is scheduled just like any other periodic task based upon the priority scheme used by the scheduling algorithm
  - When the server is scheduled and executes aperiodic jobs, it ***consumes*** its budget at the rate of 1 per unit time
  - The server budget has been ***exhausted*** when the budget becomes 0.
  - Different kinds of periodic servers differ in how the server budget changes when the server still has budget but the server is idle.