Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



- Schedulability of Sporadic Jobs in a Deadline-driven System
  - Recall that a system of periodic tasks is schedulable using EDF if the total density of the system is ≤ 1.
  - We define the density of a sporadic job J_k that has release time r_k, maximum execution time e_k, and absolute deadline d_k as $e_k/(d_k - r_k)$
  - A sporadic job is said to be ***active*** in its feasible interval $(r_k, d_k]$; it is not active outside of this interval.
  - 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 ≤ 1 at all times. (sufficient condition)
  - 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 arrive, if we deduce that the total density would exceed 1 in its feasible interval, we can reject the sporadic job (admission control).

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



- The sporadic task S_k is a stream of sporadic jobs.
- $S_{k,j}$ is the j^{th} job in the task S_k
- $e_{k,j}$ denotes the execution time of $S_{k,j}$
- $p_{k,j}$ denotes the length of time between the release times of $S_{k,j}$ and $S_{k,j+1}$
- The ratio $e_{k,j}/p_{k,j}$ is the instantaneous utilization of the job $S_{k,j}$
- The instantaneous utilization $\tilde{u}_k$ of the sporadic task S_k is the maximum of the instantaneous utilizations of all jobs in the task, $\tilde{u}_k = \max_j (e_{k,j}/p_{k,j})$
- [Note that for a periodic task, the utilization $u_k = \max_j(e_{k,j})/\max_j(p_{k,j})$, and that $u_k \geq \tilde{u}_k$]
- We assume that $\tilde{u}_k$ is known for each sporadic task S_k in the system
- A system of independent, preemptable periodic and sporadic tasks, in which the relative deadline of each job is identical to its period, is schedulable on a processor using the EDF algorithm if the sum of the total utilization of the periodic tasks and the total instantaneous utilization of the sporadic tasks is ≤ 1

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



- Constant Utilization Server
  - Server size is its instantaneous utilization $\tilde{u}_S$ - this is the fraction of the processor time reserved for the execution of aperiodic jobs
  - As with deferrable servers, the deadline d of a constant utilization server is always defined
  - A constant utilization server emulates a sporadic task with a constant instantaneous utilization
  - Assume $\tilde{u}_S$ is the size of the server, e_S is its budget, d is its deadline, t is the current time, and e denotes the execution time required by the job at the head of the aperiodic queue

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



- Constant Utilization Server (continued)
  - Consumption rule: it only consumes budget when it executes
  - Replenishment rules
    1. Initially, $e_S := 0$ and $d := 0$.
    2. 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 $e_S = e$
    3. At the deadline d of the server
       a. If the server is backlogged, $d := d + e/\tilde{u}_S$ and $e_S = e$
       b. If the server is idle, do nothing
  - 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; the deadline is set so that its instantaneous utilization is equal to $\tilde{u}_S$.

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



- Total Bandwidth Server
  - This algorithm improves the responsiveness of a constant utilization server by allowing the server to claim the background time not used by the periodic tasks
  - Replenishment rules:
    1. Initially, $e_S := 0$ and $d := 0$
    2. When an aperiodic job with execution time e arrives at time t at an empty aperiodic job queue, $d := \max(d,t) + e/\tilde{u}_S$ and $e_S := e$
    3. When the server completes the current aperiodic job, the job is removed from the queue and
       a. If the server is backlogged, $d := d + e/\tilde{u}_S$ and $e_S := e$
       b. If the server is idle, do nothing
  - As long as a total bandwidth server is backlogged, it is ALWAYS ready for execution

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



- Fairness and Starvation
  - A scheduling algorithm is ***fair*** within a particular time interval if the fraction of processor time in the interval attained by each backlogged server is proportional to the server size – e.g., fairness is important in data transmission in switched networks
  - Size of k^{th} server is $\tilde{u}_k$; for $0 < t_1 < t_2$, $\omega_k(t_1,t_2)$ is the total attained processor time for the k^{th} server in the time interval (t_1, t_2)
  - $\omega_k(t_1,t_2)/ \tilde{u}_k$ is the ***normalized service*** attained by the k^{th} server
  - A scheduler is fair in the interval (t_1, t_2) if the normalized service attained by all servers that are backlogged during the interval differ by no more than the ***fairness threshold*** FR ≥ 0 – i.e. if there are 2 or more servers, $\max_{j,k} (|\omega_j(t_1,t_2)/ \tilde{u}_j - \omega_k(t_1,t_2)/ \tilde{u}_k|) \leq$ FR.
  - If FR = 0, then $\omega_k(t_1,t_2) = \tilde{u}_k * (t_2 – t_1)$

Resources and Resource Access Control



- Continue to focus on a single processor
- System has ρ types of resources named R_1, R_2, …, R_ρ
- There are ν_k indistinguishable units of resource R_k
- Serially reusable resources are granted to jobs on a nonpreemptive basis and used in a mutually exclusive manner
- Resources that have an infinite number of units have no effect on scheduling; therefore, we assume that every resource R_k has a finite number of units
- If a resource can be used by more than one job at a time, we model that resource as having many units, each used mutually exclusively.
- We assume a lock-based concurrency control mechanism
- When a job wants to use n_k units of resource R_k, it executes a lock to request them – $L(R_k, n_k)$
- When the job is finished with the resources, it unlocks them – $U(R_k, n_k)$
- The segment of a job that begins at a lock and ends at a matching unlock is a critical section
- Resources are released in a LIFO order

Resources and Resource Access Control



- Use the expression [R, n; e] to represent a critical section regarding n units of R, with the critical section requiring e units of execution time
- Nested critical sections (first lock R_1, then lock R_2, then lock R_3, …, unlock R_3, unlock R_2, unlock R_1) is represented as $[R_1; e_1 [R_2; e_2 [R_3; e_3]]]$
- Two jobs ***conflict*** with one another if some of the resources they require are of the same type; they ***contend*** for a resource if one job requests a resource that the other job has already been granted
- If a lock request fails, the requesting job is blocked and loses the processor; when the requested resource is available, it is unblocked
- A resource access-control protocol is a set of rules that govern
  1. When and under what conditions each request for resource is granted
  2. How jobs requiring resources are scheduled.

Resources and Resource Access Control

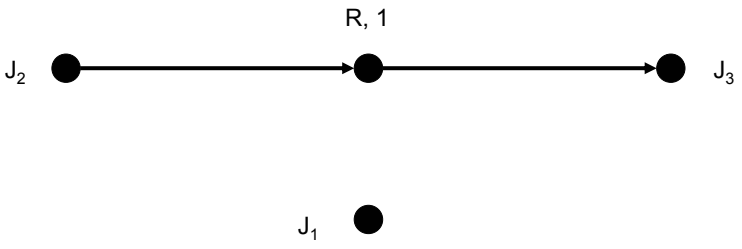


- Priority inversion – a lower-priority job executes while some ready higher-priority job waits
- Contention for resources can cause priority inversions to occur, even if the jobs are preemptable, since a lower-priority job holding a lock on a resource will prevent a higher-priority job requiring that resource from executing
- Deadlock can result from piecemeal acquisition of resources; classic example of two jobs needing resources X and Y; if one job acquires locks in the order X then Y, and the other job acquires them in the opposite order, we end up with a deadlock; classic solution is to impose a fixed acquisition order over the set of lockable resources, and all jobs attempt to acquire the resources in that order
- We will assume that every job is preemptable on the processor and that no job ever suspends itself
- A higher-priority job is directly blocked by a lower-priority job if the latter holds some resource that the former requests and is not allocated

Resources and Resource Access Control



- Wait-for graphs



Resources and Resource Access Control



- Resource requirements graph

