Review of Major Concepts

RTES4

Administrivia

- Books donated by IBM available from Myra Smith in F162 (first come – first served)
  - "Linux in a Nutshell - A Desktop Quick Reference" by Ellen Siever (O'Reilly)
  - "Java and XML" by Brett McLaughlin (O'Reilly)
  - "Introduction to Programming in Java" by Patterson Hume & Christine Stephenson (Holt Software Associates Inc.)
- Please fill out the questionnaire before you leave today – we really will look at them
  - Only lists Joe's name - write in Colin on the next line to rate us both
- Answers to problem sets 1 & 2 will be available from
  http://www.dcs.gla.ac.uk/~joe/03-04RTES.html
  by 17:00 today; your papers, together with the assessed marks, will be available from Tracy Skelton during the week of 12 April

Scheduling definitions

- A schedule is an assignment of all jobs in the system on the available processors.
- A ***valid schedule*** satisfies the following conditions:
  - Every processor is assigned to at most one job at any time
  - Every job is assigned at most one processor at any time
  - No job is scheduled before its release time
  - The total amount of processor time assigned to every job is equal to its maximum or actual execution time
  - All the precedence and resource usage constraints are satisfied
- A valid schedule is a ***feasible schedule*** if every job meets its timing constraints.
- A hard real time scheduling algorithm is ***optimal*** if the algorithm always produces a feasible schedule if the given set of jobs has feasible schedules.

Static, Timer-driven Scheduler

- Since the parameters of jobs with hard deadlines are known before the system begins to execute, construct a static schedule of the jobs off-line; periodic static schedule == ***cyclic schedule***
- The amount of processor time allocated to every job is equal to its maximum execution time
- The static schedule guarantees that each job completes by its deadline
- The scheduler dispatches jobs according to the static schedule; as long as no job ever overruns, all deadlines are met
- Since the schedule is calculated off-line, we can employ complex, sophisticated algorithms; in particular, we can choose a feasible schedule from all possible feasible schedules that optimizes some characteristic of the system (e.g. the idle periods for the processor are nearly periodic to accommodate aperiodic jobs)

Static, Timer-driven Scheduler

```
Input: stored schedule ($t_k$, $T(t_k)$) for k = 0, 1, N – 1.
Task SCHEDULER:
    set the next decision point I and table entry k to 0;
    set the timer to expire at $t_k$;
    do forever:
        accept timer interrupt;
        if an aperiodic job is executing, preempt the job;
        current task T = $T(t_k)$;
        increment i by 1;
        compute the next table entry k = I mod (N);
        set the timer to expire at [I / N] * H + tk;
        if the current task T is I,
            let the job at the head of the aperiodic queue execute;
        else,
            let the task T execute;
        sleep;
    end do.
End SCHEDULER.
```

Cyclic Executives

- A table-driven cyclic scheduler for all types of jobs in a multi-threaded system
- Table that drives the scheduler has F entries, where F = H / f; each corresponding entry L(k) lists the names of the job slices that are scheduled to execute in frame k; called a scheduling block
- Cyclic executive takes over the processor and executes at the clock interrupt that signals the start of a frame
  - It determines the appropriate scheduling block for this frame
  - It executes the jobs in the scheduling block
  - It wakes up jobs in the aperiodic job queue to permit them to use the remaining time in the frame
- Major assumptions:
  - Existence of a timer
  - Each timer interrupt is handled by the executive in a bounded time

Pros of Clock-driven Scheduling

- Conceptual simplicity
  - Ability to consider complex dependencies, communication delays, and resource contention among jobs when constructing the static schedule, guaranteeing absence of deadlocks and unpredictable delays
  - Entire schedule is captured in a static table
  - Different operating modes can be represented by different tables
  - No concurrency control or synchronization required
  - If completion time jitter requirements exist, these can be captured in the static schedule[s]
- When the workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary
- Choice of frame size can minimize context switching and communication overheads
- Such systems are relatively easy to validate, test and certify

Cons of Clock-driven Scheduling

- Such systems are inflexible – precompilation of knowledge into the scheduling tables means that if anything changes materially, have to redo the table generation – as a result, best suited for systems which are rarely modified once built
- Other disadvantages:
  1. Release times of all jobs must be fixed
  2. All combinations of periodic tasks that might execute at the same time must be known *a priori* so that the combined schedule can be precomputed
  3. The treatment of aperiodic jobs is very primitive; if there is a significant amount of soft-real-time computation in the system, it is unlikely that this structure will yield acceptable response times

Priority-driven Scheduling of Periodic Tasks

- Fixed-priority vs. Dynamic-priority Algorithms
  - A priority-driven scheduler is an on-line scheduler
    - It does NOT precompute a schedule of tasks/jobs
    - It assigns priorities to jobs when they are released and places them on a ready job queue in priority order
    - When preemption is allowed, a scheduling decision is made whenever a job is released or completed
    - At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue
  - A ***fixed-priority*** algorithm assigns the same priority to all the jobs in a task.
  - A ***dynamic-priority*** algorithm assigns different priorities to the individual jobs in a task.
  - The priority of a job is usually assigned upon its release and does not change
  - Three categories of algorithms:
    - Task-level fixed-priority
    - Task-level dynamic-priority and job-level fixed-priority
    - Job-level dynamic-priority

Priority-driven Scheduling of Periodic Tasks

Fixed-priority Algorithms

- Rate-monotonic algorithm (RM)
  - Assigns priorities to tasks based on their periods – the shorter the period, the higher the priority
  - Since the rate is $(\text{period})^{-1}$, the higher the rate, the higher the priority
- Deadline-monotonic algorithm (DM)
  - Assigns priorities to tasks according to their relative deadlines – the shorter the relative deadline, the higher the priority
- When the relative deadline of every task is proportional to its period, the RM and DM algorithms give identical results
- When the relative deadlines are arbitrary, the DM algorithm performs better in the sense that it can sometimes produce a feasible schedule when RM fails, while RM always fails when DM fails

Priority-driven Scheduling of Periodic Tasks

Dynamic-priority algorithms

- Earliest-deadline-first (EDF)
  - The job queue is ordered by earliest deadline
- Least-slack-time-first (LST)
  - The job queue is ordered by least slack time
  - Nonstrict – scheduling decisions are made only when jobs are released or completed
  - Strict – scheduling decisions are made also whenever a queued job's slack time becomes smaller than the executing job's slack time – huge overheads, not used
- First-in-first-out (FIFO)
  - Job queue is first-in-first-out by release time
- Last-in-first-out (LIFO)
  - Job queue is last-in-first-out by release time

Priority-driven Scheduling of Periodic Tasks

- Relative merits
  - Algorithms that do not take into account the urgencies of jobs in priority assignment usually perform poorly (FIFO, LIFO)
  - Algorithms are ranked by their ability to maximize the utilization of the system in terms of meeting job deadlines – maximum value of 1 – EDF is optimal in this sense, while RM and DM are not
  - EDF continues to give high priority to jobs that have already missed their deadlines relative to a job whose deadline is in the future; therefore, EDF is not particularly suitable to systems where overload conditions are unavoidable

Priority-driven Scheduling of Periodic Tasks

- The schedulable utilization of the EDF algorithm
  - Theorem: A system $\mathcal{T} = \{T_{i}, {}_{l} = 1..n\}$ of independent, preemptable periodic tasks with $D_i = p_i$ can be feasibly scheduled on one processor if and only if $U(\mathcal{T}) \leq 1$. (proof covered next Tuesday)
  - Corollaries:
    - A system $\mathcal{T}$ of independent, preemptable periodic tasks with $D_i > p_i$ can be feasibly scheduled on a processor as long as $U(\mathcal{T}) \leq 1$
    - $\mathcal{U}_{EDF}(n)$ for n independent, preemptable periodic tasks with $D_i \geq p_i$ is 1.
    - $\mathcal{U}_{LST}(n)$ for n independent, preemptable periodic tasks with $D_i \geq p_i$ is 1.
  - Note that all of these results are independent of φ_i

Priority-driven Scheduling of Periodic Tasks

- What happens if $D_i < p_i$ for some i?
  - Define $\tau_i = \min(D_i, p_i)$
  - The density for T_i, $\delta_i = e_i / \tau_i$
  - The density of the system, $\Delta(\mathcal{T}) = \delta_1 + \delta_2 + \ldots + \delta_n$
  - Theorem: A system $\mathcal{T}$ of independent, preemptable periodic tasks can be feasibly scheduled on one processor if $\Delta(\mathcal{T}) \leq 1$.
    - Note that this is not a necessary condition, it is simply sufficient – i.e. a system may be feasible when $\Delta(\mathcal{T}) > 1$.

Priority-driven Scheduling of Periodic Tasks

- Schedulability testing
  - A test for the purpose of validating that the given application system meets all its hard deadlines when scheduled according to a particular scheduling algorithm is a ***schedulability test***.
  - If a schedulability test is efficient, then it can be used as an on-line acceptance test.
- Schedulability test for EDF
  - $\Delta(\mathcal{T}) \leq 1$
  - What do we conclude if this test is not satisfied?
    - If $D_i \geq p_i$ for all i, then the system is not schedulable
    - If $D_i < p_i$ for some i, the system may not be schedulable
  - This test is robust – i.e. the test holds true if some jobs execute for less than their maximum execution times; it also holds true if the interrelease times of jobs in a task are longer than the period (minimum interrelease time)

Priority-driven Scheduling of Periodic Tasks

- Optimality of the RM and DM algorithms
  - We've already seen examples wherein these fixed-priority algorithms are not optimal
  - In fact, if the periods of the tasks in the system are related appropriately, then the RM and DM algorithms are optimal
  - A system of periodic tasks is ***simply periodic*** if for every pair of tasks T_i and T_k in the system and $p_i < p_k$, p_k is an integer multiple of p_i. (Recall our avionics example from lecture 2.)
  - Theorem: A system of simply periodic, independent, preemptable tasks whose relative deadlines are ≥ their periods is schedulable on one processor according to the RM algorithm iff its total utilization is ≤ 1.
  - Corollary: The same is true for the DM algorithm.
  - Since fixed-priority algorithms are more constrained, why would one choose to use them?
    - They often lead to more predictable and stable systems

Priority-driven Scheduling of Periodic Tasks

- Schedulable utilization of the RM algorithm
  - Assume $D_i = p_i$ for all I
  - Arbitrary relationships between relative deadlines of tasks
  - $\mathcal{U}_{RM}(n) = n\,(2^{1/n} - 1)$
  - For large n, approaches ln 2 (0.693)
  - $U(\mathcal{T}) \leq \mathcal{U}_{RM}(n)$ is a necessary condition – i.e. if $U(\mathcal{T}) > \mathcal{U}_{RM}(n)$, the RM algorithm (or better yet, the DM algorithm) may be able to find a feasible schedule

n	2	4	6	8	10	12	14	16	18	20	100
$\mathcal{U}_{RM}(n)$	0.828	0.757	0.735	0.724	0.718	0.714	0.710	0.708	0.707	0.705	0.696

Priority-driven Scheduling of Periodic Tasks

- Practical factors
  - A ready job is ***blocked*** when it is prevented from executing by a lower-priority job
  - A ***priority inversion*** occurs whenever a lower-priority job executes while some ready higher-priority job waits
  - Nonpreemptability
    - Many reasons why a job may have nonpreemptable sections
      - Critical section over a resource
      - Some system calls are nonpreemptable
      - Disk scheduling
    - If a job becomes nonpreemptable, priority inversions may occur
    - When attempting to understand whether a task meets all of its deadlines, we must consider not only all the tasks that have higher priorities than that task, also the nonpreemptable portions of lower-priority tasks

Priority-driven Scheduling of Periodic Tasks

- Practical factors (continued)
  - Nonpreemptability (continued)
    - Define $b_i(np)$ as the longest amount of time for which any job in the task T_i can be blocked each time it is blocked due to nonpreemptive lower-priority tasks
  - Self-suspension
    - A job may invoke an external operation (e.g. request and I/O operation), during which time the job is suspended
  - Context Switches
    - Assume we know the maximum number of context switches K_i for a job in T_i
    - Can add $2(K_i + 1)\ t_{CS}$ to the execution time of T_i

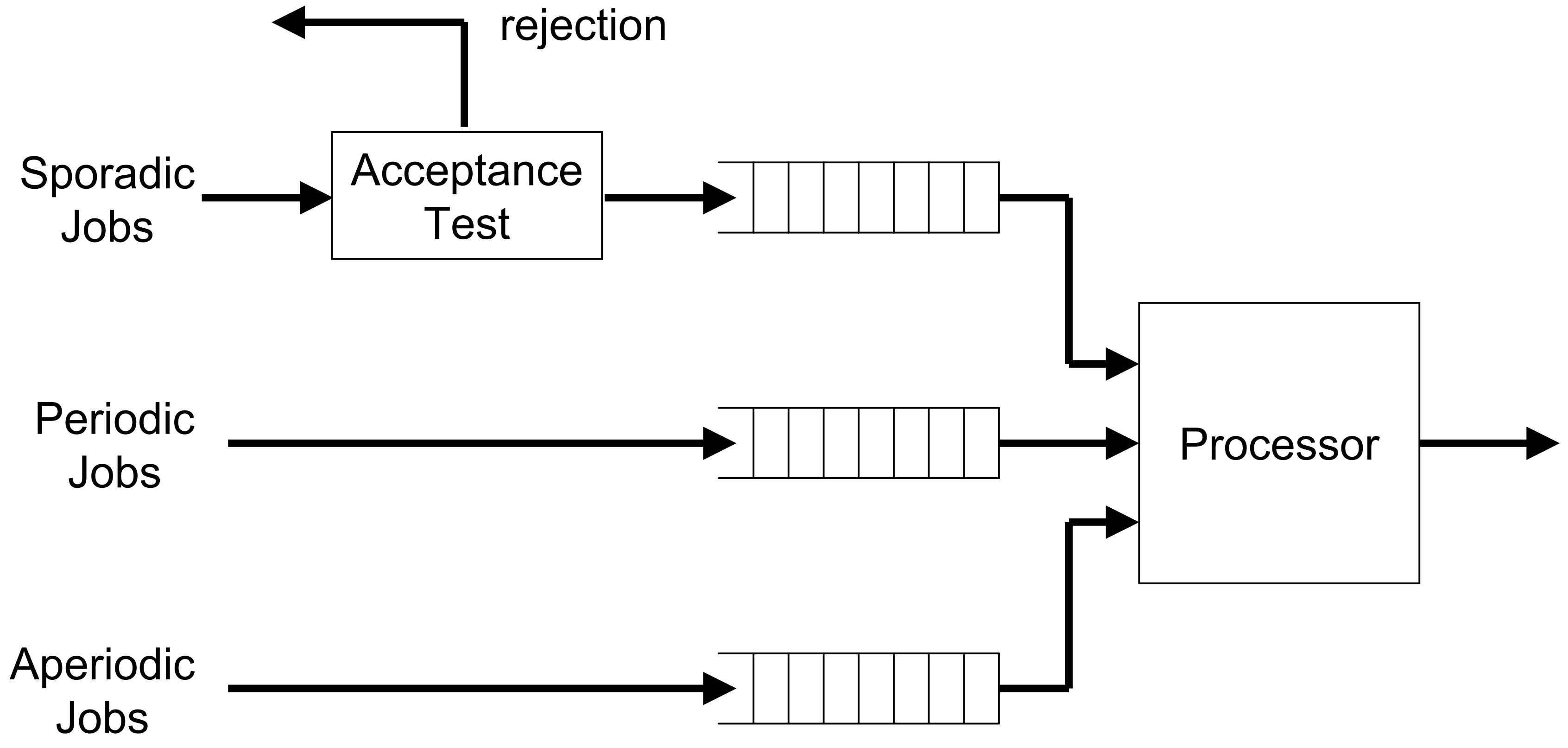
Priority-driven Scheduling of Periodic Tasks

- 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

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems



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

- 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

- Bandwidth-preserving server algorithms
  - 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
  - We would like to be able to ***preserve*** the execution budget of the server when it finds an empty queue, such that it can 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***

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Deferrable server
  - The simplest of BP servers
  - 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 set to e_S at time instants kp_S, for $k = 0, 1, 2, \ldots$
  - Note that the server is not allowed to carry over budget from period to period

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Schedulability of deadline-driven systems with a deferrable server
  - The deadline of a deferrable server is its next replenishment time
  - A period task T_i in a system of N independent, preemptive, 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

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

Note that if the deferrable server was being treated just like any other periodic task, the second term on the left hand side would just be u_S.

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- 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
  - Sporadic server are 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

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Consumption and replenishment rules
  - Definitions
    - t_r denotes the latest (actual) replenishment time
    - t_f denotes the first instant after tr at which the server begins to execute
    - t_e denotes the latest effective replenishment time
    - At any time t, BEGIN is the beginning instant of the earliest busy interval among the latest contiguous sequence of busy intervals of T_H that started before t.
    - END is the end of the latest busy interval in this sequence if this interval ends before t and equal to ∞ if the interval ends after t
  - The scheduler sets t_r to the current time each time it replenishes the server's execution budget.
  - 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

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Recall Simple Sporadic Server for fixed-priority
  - Release rules: the server is enabled whenever it is backlogged and has budget
  - Consumption rules: budget is consumed when
    1. the server is executing
    2. the server has executed since t_r and (lower-priority jobs are executing OR the system is idle)
  - Replenishment rules
    1. initially, and at each replenishment time, budget := e_s and set t_r := current time
    2. t_f is the time that the server is 1st scheduled after t_r; calculate t_e as:
       a. If t_f coincides with the end of a higher-priority job, t_e := max(t_r, BEGIN)
       b. If lower-priority job or idle system preceded t_f, $t_e := t_f$

       the next replenishment time is set for $t_e + p_s$
    3. Replenishment occurs at $t_e + p_s$ except …
       a. If $t_e + p_s < t_f$, budget is replenished as soon as exhausted
       b. If system idles before $t_e + p_s$ and becomes busy again at $t_b < t_e + p_s$, the budget is replenished at t_b

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- What do these rules really mean?
  - View each replenishment time as the nominal "release time" of a server job; actual release time is t_e
  - C1 implies each server job executes for no more time than its execution budget
  - C2 implies that the server retains its budget if
    - A higher-priority job is executing, or
    - It has not executed since t_r
    - This implies that if the server idles while it has budget, budget continues to decrease over time
  - R2 makes the effective replenishment time as soon as possible commensurate with the server acting like a periodic task (, p_s, e_s)
  - R3a assumes that $D_s > p_s$, and that this fact was taken into account in determining the schedulability of the system
  - The system is correct without rule R3b; the text discusses why it is still correct with R3b; in essence, it causes replenishment to happen sooner if the system becomes busy after an idle interval ➔ that the system will be able to react more quickly to the arrival of an aperiodic task

Scheduling Aperiodic and Sporadic Jobs in Priority-driven Systems

- Simple Sporadic Server for dynamic-priority (EDF)
  - Release rules: the server is enabled whenever it is backlogged, it has budget, and its deadline d is defined
  - Consumption rules: budget is consumed when
    1. the server is executing
    2. the deadline d is defined, the server is idle, and there are no jobs with a deadline before d ready for execution
  - Replenishment rules
    1. initially, and at each replenishment time, budget := e_s and set t_r := current time; initially, t_e and d are undefined
    2. whenever t_e is defined, $d = t_e + p_s$, and the next replenishment time is $t_e + p_s$; t_e defined as follows:
       a. At time t when an aperiod job arrives at an empty queue
          i. If only jobs with deadlines earlier than $t_r + p_s$ have executed in (t_r, t), $t_e := t_r$
          ii. If any jobs with deadlines after $t_r + p_s$ have executed in (t_r, t), $t_e := t$
       b. At replenishment time t_r
          i. If server is backlogged, $t_e := t_r$
          ii. If server is idle, t_e and d become undefined
    3. Replenishment occurs at $t_e + p_s$ except …
       a. If $t_e + p_s < t$ when server first becomes backlogged after t_r, budget is replenished as soon as exhausted
       b. budget is replenished at end of each idle interval of **T**

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

Resources and Resource Access Control

- Basic Priority-inheritance Protocol
  - Works with any preemptive, priority-driven scheduling algorithm
  - Does not require any prior knowledge of the jobs' resource requirements
  - Does NOT prevent deadlock
  - If one uses some other mechanism to prevent deadlock, it ensures that no job is ever blocked indefinitely due to uncontrolled priority inversion

Resources and Resource Access Control

- Basic Priority-inheritance Protocol
  - Scheduling Rule: ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities. At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority. It remains at this priority except when the priority-inheritance rule is invoked.
  - Allocation Rule: when a job J requests a resource R at time t:
    - If R is free, R is allocated to J until J releases it
    - If R is not free, the request is denied and J is blocked
  - Priority-inheritance rule: when the requesting job J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J; J_l executes at its inherited priority until it releases R; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquired the resource R

Resources and Resource Access Control

- What does this mean?
  - A job J is only denied a resource when the resource it requests is held by another job
  - At time t when J requests the resource, it has the highest priority among all ready jobs → the current priority $\pi_l(t)$ of the job J_l directly blocking J is never higher than the priority $\pi(t)$ of J

Resources and Resource Access Control

- Properties of the Priority-inheritance Protocol
  - Different types of blocking
    - Direct blocking (time 6)
    - Priority-inheritance blocking (time 6)
    - Transitive blocking (time 9)
  - Does NOT prevent deadlock – simple piecemeal acquisition in different orders problem
  - Does not minimize the blocking times suffered by jobs since it is so aggressive

Resources and Resource Access Control

- Priority-ceiling Protocol
  - Extends priority-inheritance to prevent deadlock and to further reduce blocking times
  - Makes two key assumptions:
    - The assigned priorities of all jobs are fixed
    - The resources required by all jobs are known *a priori* before the execution of any job begins
  - Need two additional terms to define the protocol:
    - The ***priority ceiling*** of any resource R_k is the highest priority of all the jobs that require R_k and is denoted by $\Pi(R_k)$
    - At any time t, the current priority ceiling $\mathit{\Pi}(t)$ of the system is equal to the highest priority ceiling of the resources that are in use at the time; if all resources are free at the time, $\mathit{\Pi}(t)$ is equal to Ω, a nonexistent priority level that is lower than the lowest priority level of all jobs

Resources and Resource Access Control

- The Basic Priority-Ceiling Protocol
  - Scheduling rules
    a. At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority; it remains at that priority except as defined in the priority-inheritance rule
    b. Every ready job J is scheduled preemptively and in a priority-driven manner at its current priority $\pi(t)$
  - Allocation rules – whenever a job J requests a resource R at time t, one of the following occurs:
    a. R is held by another job; J's request fails and J becomes blocked
    b. R is free
       i. If J's priority $\pi(t)$ is higher than the current priority ceiling $\Pi(t)$, R is allocated to J
       ii. If J's priority $\pi(t)$ is not higher than the ceiling $\Pi(t)$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\Pi(t)$; otherwise, J's request is denied, and J becomes blocked
  - Priority-inheritance rule: when J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J; J_l executes at its inherited priority until the time when it releases every resource whose priority ceiling is equal to or higher than $\pi(t)$; at that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it was granted the resource(s)

Resources and Resource Access Control

- Important results for priority-ceiling
  - When resource accesses of a system of preemptive, priority-driven jobs on one processor are controlled by the priority-ceiling protocol, deadlock can never occur (remember, this was for **fixed-priority** algorithms).
  - When resource accesses of preemptive, priority-driven jobs on one processor are controlled by the priority-ceiling protocol, a job can be blocked for at most the duration of one critical section – i.e. there is no transitive blocking under the priority-ceiling protocol.

Real-Time Operating Systems Concepts

- Differences from general purpose systems
- Flexible microkernel architecture
- Emphasis on predictability, not performance
- Embedded in, interacting with, a larger system
  - Hardware customised to the application
  - Sensors and actuators; interaction with the environment
  - Closed system, trusted applications
- Often resource constrained and safety critical
- Strong reliability requirements

Features of Real-Time Operating Systems

- Clocks and Timers
  - Nanokernels, implementing a cyclic executive
  - Clock interrupts and timer services
    - Resolution, accuracy and stability
    - Timer and scheduling jitter, latency, effects on scheduling
- Interrupt handling
  - Blocking due to device access; non-preemptable regions
  - Scheduled tasks to service interrupts
- Memory access and protection
  - Overheads; desirability (or otherwise) of memory protection
  - Effects on system call and interrupt latency

Implementation of Tasks

- Task implementation using threads and processes
  - Thread control block and context
  - Task parameters
- Periodic thread abstraction
  - Benefits of the abstraction; simplified programming model
  - Implementation issues
- Sporadic and aperiodic threads
  - Implementation using a background server thread
- Need for an acceptance test before starting a real-time task, to ensure it is schedulable

Implementation of Schedulers

- Task state diagram, mapping onto scheduler queues
- Scheduler queue structures to implement fixed priority scheduling
  - Example: Rate Monotonic
- Dynamic priority scheduling
  - Expensive using the queue architecture for fixed priority
  - Queuing structure to directly support EDF scheduling
  - Sporadic and aperiodic tasks
- Background servers
  - Trivial implementation of a slack stealer
  - Why more complex servers need scheduler support for efficiency
- Tasks with temporal distance constraints; DCM scheduling
- Scheduling real time and non-real time tasks together
  - Open system architecture and the two-level scheduler

Scheduling Standards and APIs

- POSIX 1003.1b real time scheduling
  - Outline of the API
  - Scheduling classes: FIFO and RR tasks, difference from other time sharing tasks
  - Scheduling parameters
- POSIX 1003.1c “pthreads”
  - Similarity to the POSIX 1003.1b API
  - Implementation of threads, mapping onto kernel scheduled entities, effects on thread scheduling

Implementing and using POSIX scheduling

- The POSIX real-time scheduling abstraction
  - Implementation on a fixed priority queue scheduler
- Use of POSIX scheduling primitives to implement:
  - Rate Monotonic
    - Effects of limited priority levels
  - Slack stealing background tasks
    - Understanding of why other types of server are difficult to implement under the POSIX scheduling abstraction
    - Need direct scheduler support
- Extending the POSIX APIs
  - Possible implementation of EDF and other algorithms

Resource Access Control

- Locking and critical sections
- Concurrency primitives
  - Semaphores, mutexes and condition variables
  - Priority inheritance support in POSIX mutexes
- Implementation of priority inheritance
  - Complexity of operations; amount of queue walking
- Priority inheritance and stack-based priority ceiling protocol
  - Comparison of scheduling
  - Implementation strategies
- Message queues, signals and events
  - Prioritization and message based priority inheritance
  - Synchronous, asynchronous and blocking messages

Flexible Applications

- Real-time on general purpose systems
- Flexible computation
  - Sieve
  - Milestone
  - Multiple versions
- Workload model for flexible jobs
- Dependent jobs
- Criteria of optimality
  - Characteristics of error functions
- Scheduling flexible applications
  - Off-line
  - On-line and approximate; heuristics

Real-Time Communications

- Modelling both generic and packet networks
  - Queuing delays, transit time
- Performance metrics: loss, throughput, delay, jitter
  - Jitter distribution; modelling
  - Clock skew
- Application requirements
  - Sensitivity to throughput, absolute delay, delay jitter
- Scheduling communications in Controller Area Networks
- Modelling the approximate properties of an IP network
  - Understanding variability inherent in IP networks
  - Examples of performance
- Understanding that the network is a resource that can be scheduled and modelled

Quality of Service in Packet Networks

- Requirements to implement enhanced service networks
  - Packet scheduling algorithm
  - Admission control
  - Signalling protocol (RSVP)
- Service disciplines
  - Provision of a proportional fair share, timing isolation
  - Making packet networks predictable
- Weighted fair queuing
  - Properties; service guarantees
    - Per-hop and end-to-end delay, given assumptions
  - Packet scheduling algorithm, concepts and implementation
  - Approximation using frame-based WFQ
- Weighted round robin
  - Implementation
  - Guarantees on throughput and delay bounds

Real-Time on IP networks

- Behaviour of IP
- Timing properties of UDP and TCP
- Reliability/timeliness tradeoff
- Overview of RTP
    - End-to-end principle
    - Application layer framing
    - Protocol overview
- Media playout and synchronisation
    - Playout buffers, choosing buffering delay

The End…

- Revision lecture on 27th April, 3pm, F171
- Answers to sample exam will be discussed

- Any questions?