Week 13– Advanced Real-Time Scheduling

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Outline of This Week

- Schedulability analysis
  - UB Test
  - RT Test
- Including the effects of blocking
- Real-Time Synchronization Protocols
  - Basic Priority Inheritance Protocol
  - Priority Ceiling Protocol
  - Priority Ceiling Emulation Protocol
- Dynamic priority based scheduling

Why Are Deadlines Missed?

- For a given task, consider
  - **Preemption**: time waiting for higher priority tasks
  - **Execution**: time to do its own work
  - **Blocking**: time delayed by lower priority tasks

- The task is schedulable if the sum of its preemption, execution, and blocking is less than its deadline.

Periodic Task: {C, T}

- Periodic task
  - Initiated at fixed intervals
  - Must finish before the relative deadline of the task
- Specifying a task $\{C_i , T_i\}$
  - C_i = worst-case compute time (execution time) for task τ_i
  - T_i = period of task τ_i

We assume that the relative deadline D_i of a task is the same as the period T_i

- Individual task's CPU utilization:
- Total CPU utilization for a set of tasks

$$U_i = \frac{C_i}{T_i}$$

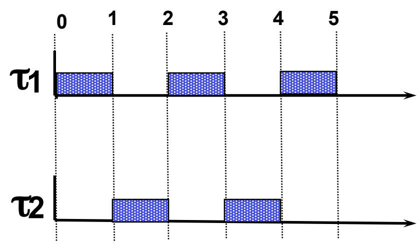
$$U = U_1 + U_2 + ... + U_n$$

- Assumption: Fixed priority preemptive scheduler
- Problem: Given $\{C_i , T_i\}$ for a set of tasks determine
  - Whether the tasks are schedulable?
  - How to assign priorities to the tasks so that the tasks are schedulable?

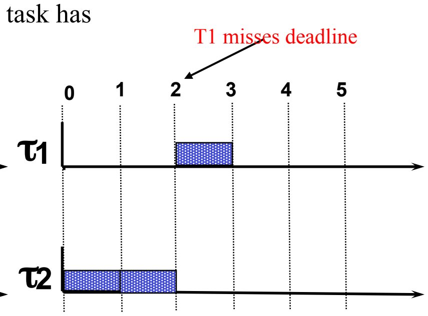
Task Schedulability – Priority Assignment Matters

- A set of tasks is schedulable if all tasks are guaranteed to meet their deadlines
- Why will a set of tasks be not schedulable?
  - Priority assignment
  - The values of C and Ts for the tasks (their utilization factor)

Example – T1 (1,2) & T2 (2,5) are two tasks.
Two cases arise depending on which task has
a higher priority



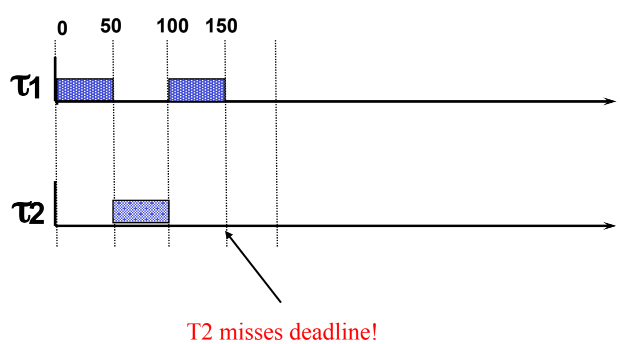
T1 has higher priority



T2 has higher priority

Task Schedulability – CPU Utilization Matters

Consider 2 periodic tasks
{C , T} for the first task (50,100)
{C , T} for the second task (60,150)
Assume both tasks arrive simultaneously at `t=0`
Overall utilization factor for these tasks = 0.9



Task Scheduling

- Liu and Leyland ("*Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*", JACM, 1973) – if there is any ***static*** priority assignment that makes a set of tasks schedulable, then the tasks are schedulable when higher frequency tasks are assigned higher priorities (rate monotonic policy)
  – Rate monotonic priority assignment is optimal under ***static*** priority assignment
  – Can assign priorities to the task according rate monotonic policy without affecting the schedulability of the tasks

- Utilization bound (UB) test says that a task set is schedulable if its total CPU utilization is less than a bound called the **Liu & Leyland bound**
  – Also shown that the worst case phasing of a set of tasks is when all tasks arrive at the same instant
  – UB test shows that the tasks are schedulable under the worst case phasing

Basic Schedulability: UB Test

- Utilization bound (UB) test: a set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n} - 1)$$

U(1) = 1.0	U(4) = 0.756	U(7) = 0.728
U(2) = 0.828	U(5) = 0.743	U(8) = 0.724
U(3) = 0.779	U(6) = 0.734	U(9) = 0.720

- As the number of tasks goes to infinity, the bound approaches ln(2) = 0.693
  – In other words, any number of independent periodic tasks will meet their deadlines if the total system utilization is under 69%

Assumptions

- The utilization bound test assumes that
  - All tasks are periodic
  - Each task's deadline is at the end of its period
  - Tasks are independent (non-interacting and do not synchronize with each other)
  - Tasks do not suspend themselves in the middle of computations
  - The processor always executes the highest priority task
  - Context switches between tasks take zero time
  - There are no interrupts

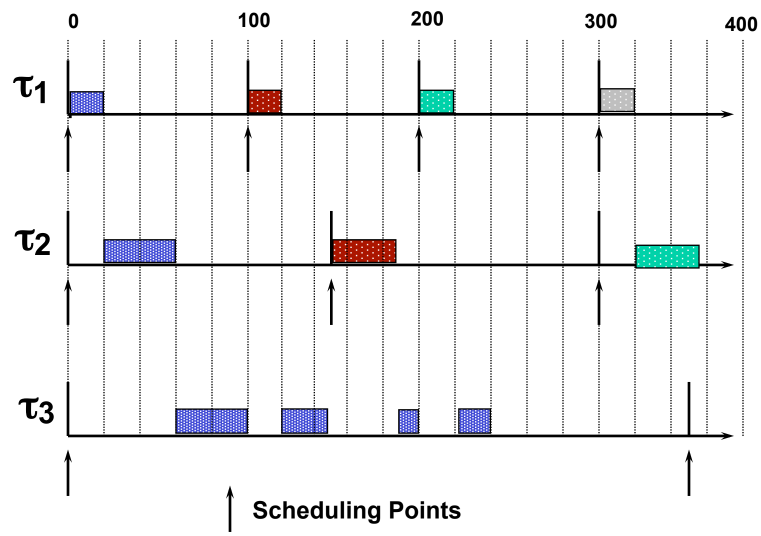
Example Problem: Applying UB Test

	C	T	U
Task t_1:	20	100	0.200
Task t_2:	40	150	0.267
Task t_3:	100	350	0.286

- *Left-hand side*
  - $U_1 + U_2 + U_3$ = total utilization for 3 tasks = .200 + .267 + .286 = .753
- *Right-hand side*
  - U(3) = .779

➢ Apply the UB test: $U_1 + U_2 + U_3 < U(3)$
➢ The periodic tasks in the example are schedulable according to the UB test
➢ Also, 24.7% of the CPU capacity is available

Timeline for Sample Problem



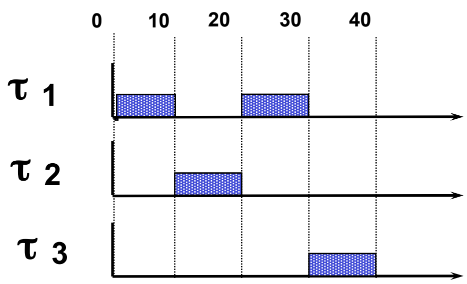
Drawing a Timeline

- Timelines show one possible execution schedule and provide a graphical view of schedule
- Use the following conventions
  - Arrange tasks in rate monotonic order, highest frequency at the top
  - Assume Liu and Leyland "worst-case" phasing, where all tasks start at time t=0 (*unless otherwise mentioned*)
  - Execution time for `T1` is plotted on its line
  - Execution time for `T2` is then plotted on its line, accommodating preemption from `T1's` execution; then this process is repeated for remaining tasks
  - If any task is preempted, its execution time block is divided with a hole in the middle representing the preemption (e.g. `T3` in the previous slide)

UB Test

- UB test has three possible outcomes:
  - $0 < U \leq U(n)$ → Success (tasks will meet deadlines)
  - $U(n) < U \leq 1.00$ → Inconclusive
  - $1.00 < U$ → Overload (one or more deadlines missed)
- UB test is conservative
  - More precise test need to be applied when the UB test is inconclusive
  - Example of a case where UB test is inconclusive but tasks are schedulable:

```
T1= {10,20}, T2 = {10, 40}, T3 = {10, 40}
U1+U2+U3=1
```



Response-Time Test (RT Test)

- Theorem
  - For a set of independent, periodic tasks, if each task meets its first deadline, with worst-case task phasing, the deadline will always be met (again, rate monotonic scheduling is assumed)

- **Let a_n = response time of task i** where

$$a_{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{a_n}{T_j} \right\rceil C_j \qquad \text{where } a_0 = \sum_{j=1}^{i} C_j$$

  - Test terminates/converges when $a_{n+1} = a_n$

- This test must be **repeated for every task t_i if required**
  - The value of *i* will change depending upon the task you are looking at
- Stop the test once current iteration yields a value of a_{n+1} beyond the deadline for that task (else, you may never terminate).
- The square parentheses represent a 'ceiling function'

Example: Applying RT Test – I

• Taking the sample problem, we increase the compute time of τ_1 from 20 to 40; is the task set still schedulable? Assume T=D as before.

	C	T	U
✓Task τ_1:	~~20~~ 40	100	~~0.200~~ 0.4
✓Task τ_2:	40	150	0.267
? Task τ_3:	100	350	0.286

• Utilization of first two tasks: 0.667 < U(2) = 0.828

– First two tasks are schedulable by UB test

• Utilization of all three tasks: 0.953 > U(3) = 0.779

– UB test is inconclusive

– Need to apply RT test to the third task

Example: Applying RT Test – II

•Use RT test to determine if τ_3 meets its first deadline: i = 3

•Compute the response time iterations, i.e., a_0, $a_{1, \ldots}$

•Wait for the test to converge and then compare with the deadline T

$$a_0 = \sum_{j=1}^{3} C_j = C_1 + C_2 + C_3 = 40 + 40 + 100 = 180$$

$$a_1 = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{a_0}{T_j} \right\rceil C_j = C_3 + \sum_{j=1}^{2} \left\lceil \frac{a_0}{T_j} \right\rceil C_j$$

$$= 100 + \left\lceil \frac{180}{100} \right\rceil (40) + \left\lceil \frac{180}{150} \right\rceil (40) = 100 + 80 + 80 = 260$$

Example: Applying the RT Test – III

$$a_2 = C_3 + \sum_{j=1}^{2} \left\lceil \frac{a_1}{T_j} \right\rceil C_j = 100 + \left\lceil \frac{260}{100} \right\rceil (40) + \left\lceil \frac{260}{150} \right\rceil (40) = 300$$

$$a_3 = C_3 + \sum_{j=1}^{2} \left\lceil \frac{a_2}{T_j} \right\rceil C_j = 100 + \left\lceil \frac{300}{100} \right\rceil (40) + \left\lceil \frac{300}{150} \right\rceil (40) = 300$$

$a_3 = a_2 = 300$ Done! Test has converged

•Now, compare with deadline
•Task τ_3 is schedulable using RT test

$$a_2 = 300 < T = 350$$

Underlying Assumptions

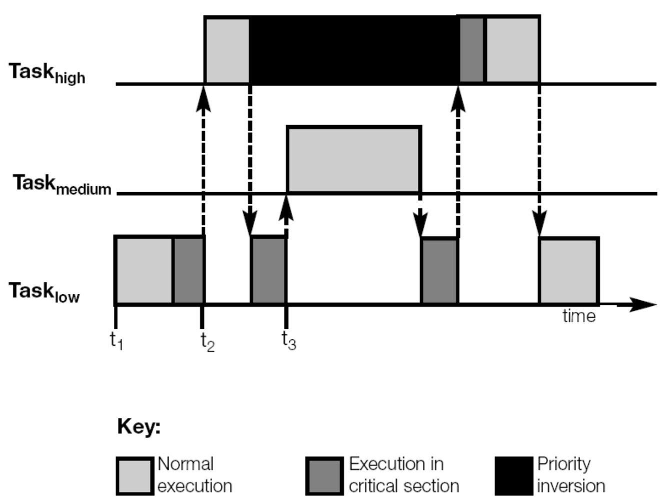
- UB and RT tests share the same limitations

- All tasks run on a single processor
- All tasks are periodic and noninteracting
- Deadlines are always at the end of the period
- There are no interrupts
- Rate-monotonic priorities are assigned
- There is zero context-switch overhead
- Tasks do not suspend themselves

Interacting Tasks

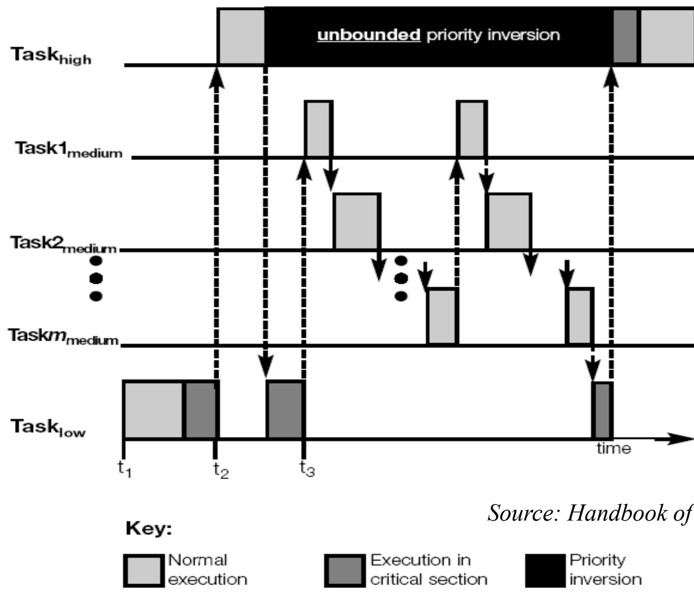
- So far we have assumed tasks do not share resources (semaphores, files, …)
- Tasks in practical real-time systems share resources
- Sharing resources can cause two problems
  - Deadlocks (as in the case of non real-time systems)
  - Blocking also called priority inversion
    - Happens in non real-time systems, but can cause tasks to miss deadlines in real time systems
- Priority Inversion
  - Delay to a task's execution caused by interference from lower priority tasks is known as priority inversion
  - UB test can be modified to account for priority inversion by adding *blocking time* to the worst-case execution time of the task (to be covered soon)
  - Identifying and evaluating the effect of sources of priority inversion is important in schedulability analysis

Priority Inversion



Source: Handbook of Real-time Systems

Unbounded Priority Inversion



Source: Handbook of Real-time Systems

UB Test with Blocking

- A set of `n` tasks (assuming they are ordered in decreasing priority) are schedulable if

$$\boxed{\frac{C_1}{T_1}+\frac{C_2}{T_2}+\ldots+\frac{C_{i-1}}{T_{i-1}}}+\boxed{\frac{C_i}{T_i}}+\boxed{\frac{B_i}{T_i}}<U(i)$$

- Preemption due to higher priority tasks

Task's run time

Blocking due to lower priority tasks

for all `i=1,2,3,… n`

- B_i = maximum priority inversion (blocking) encountered by any instance of task T_i
- $B_n = 0$
- Blocking is also included in the RT test – perform the test as before but add the blocking term

$$a_{n+1} = B_i + C_i + \sum_{j=1}^{i-1} \left\lceil \frac{a_n}{T_j} \right\rceil C_j$$

$$\text{where } a_0 = B_i + \sum_{j=1}^{i} C_j$$

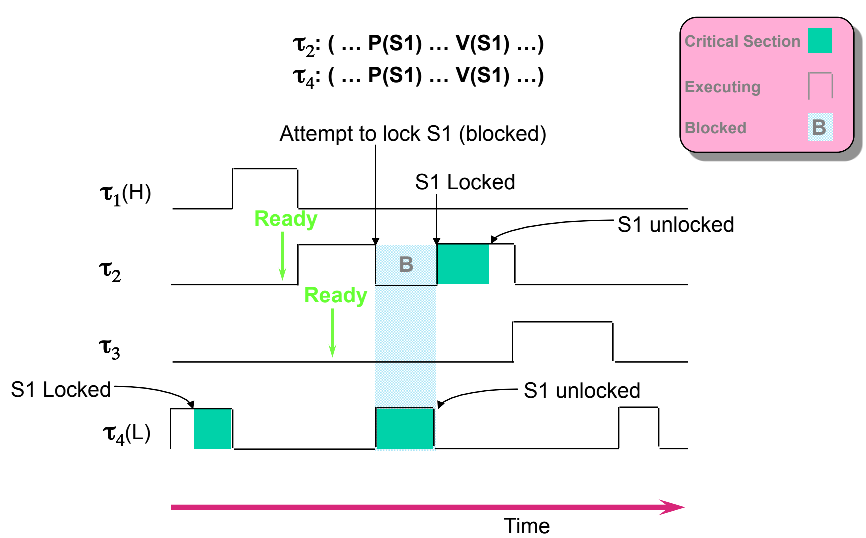
Real-time Synchronization Protocols

- Basic Priority Inheritance Protocol
- Priority Ceiling Protocol
- Priority Ceiling Emulation Protocol
  - Also called highest locker priority protocol (you need to implement this in lab 4)

- Each protocol prevents **unbounded** priority inversion
  - You cannot avoid priority inversion, but you can put a time bound on how long it will take

- Basic Priority Inheritance Protocol
  - A task runs at its own priority until it is blocking a higher priority task, in which case the priority of the task is raised to that of the higher priority task it is blocking

Basic Priority Inheritance Protocol



Can Deadlocks Occur in Priority Inheritance?

- Task `T1` needs mutex `L1` and then `L2`
- Task `T2` needs mutex `L2` and then `L1`
- Task `T2` has a higher priority than `T1`

- Suppose `T1` runs, locks `L1` and is then preempted by `T2`
- Now, `T2` runs, locks `L2` and needs `L1`

- According to priority inheritance protocol, `T1` will be elevated to `T2's` priority, and will start to run, but will soon need `L2`
  - L2 has been previously locked by task T2
  - T2 cannot release L2 because it is blocked, waiting for L1

- Both tasks are deadlocked!
- How do we work around this?

Priority Ceiling Protocol

- Each resource is assigned a priority ceiling which is the priority of the highest task that uses the resource

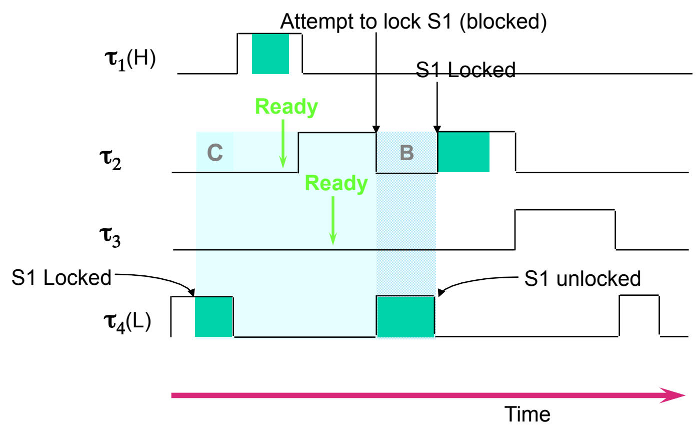
- Current priority ceiling – highest priority ceiling of the resources currently acquired by the tasks

- Rules of priority ceiling protocol
  - **Scheduling Rule**: Each job runs at its assigned priority outside a critical section
  - **Allocation Rule**: Whenever a job J requests a critical section R (resource), one of the following two conditions happen
    1. R is held by another job; request fails and J blocks
    2. R if free
       i. If J's priority is higher than the current priority ceiling, R is allocated to J
       iii. If J's priority is not higher than the current priority ceiling, R is allocated to J only if J is the job holding the resources whose priority ceiling is equal to the current priority ceiling
  - **Priority-Inheritance Rule**: When J blocks in allocation rule 1 or 2(ii) the job blocking J inherits J's priority

PCP Example 1

τ_1: (… P(S0) … V(S0) …)
τ_2: (… P(S1) … V(S1) …)
τ_4: (… P(S1) … V(S1) …)

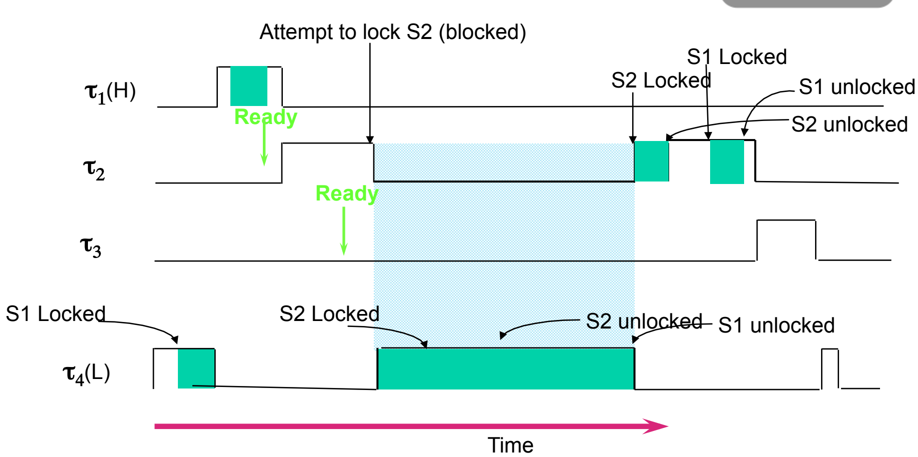
Critical Section
Executing
Blocked B



PCP Example 2

τ_1: (… P(S0) …………………… V(S0) …)
τ_2: (… P(S2) … V(S2) …P(S1)…V(S1)…)
τ_4: (… P(S1) … P(S2) …V(S2)…V(S1) …)

Critical Section
Executing
Blocked B

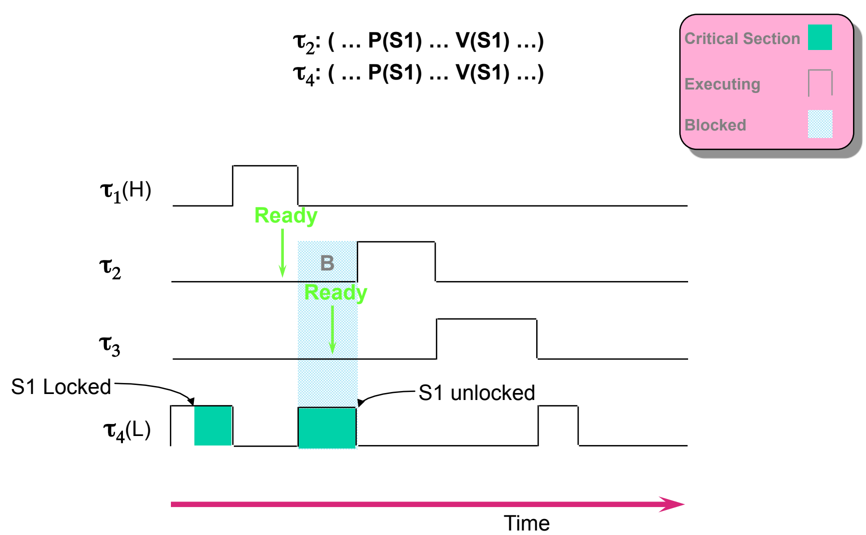


Priority Ceiling Protocol

- Advantage: Prevents deadlock
  - If processes share resources, the process that grabs the first resource is allowed to grab all other resources while all other processes get blocked due to allocation rule 1 or 2(ii)
  - Example
    - Task `T1` needs mutex `L1` and then `L2`
    - Task `T2` needs mutex `L2` and then `L1`
    - Task `T2` has a higher priority than `T1`
    - Priority ceiling of `L1` is and `L2` is
    - Suppose `T1` runs, locks `L1` and is then preempted by `T2`
    - Now, `T2` runs, tries to lock `L2`…
- Disadvantage:
  - Need to do task analysis to determine priority ceiling of resources
  - Can cause unnecessary blocking of tasks (in example 2, task 2 need not block for S2)

Priority Ceiling Emulation Protocol

Task executes at a priority higher than or equal to the priority ceiling of a resource as soon as it gets access to the resource



How Real is Priority Inversion?

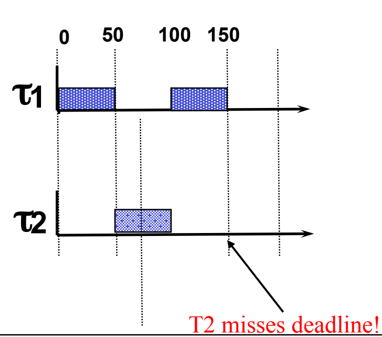
- Mars Rover experienced total system resets
- Operating system used: WindRiver's VxWorks
  - Preemptive priority scheduling of threads
- Rover's priority-based architecture
  - High-priority thread managed the information bus
  - Medium-priority thread ran a communications task
  - Low-priority data-gathering thread used bus to publish data
  - Bus access governed by mutex
- What happened?
  - High-pri task blocked, waiting for low-pri task to release mutex
  - Interrupt would occur, causing med-pri task to be scheduled
  - Watchdog timer would notice that high-pri task did not run for a while, and cause a total system restart

Earliest Deadline First Scheduling

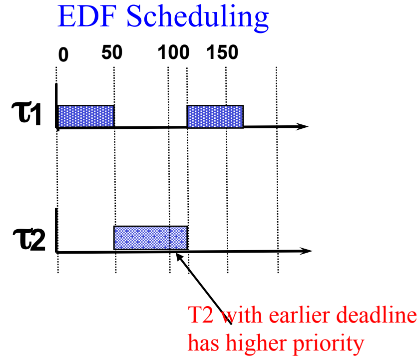
- RMS assumes that the priorities assigned to different tasks are fixed
- Can a scheduling algorithm do better if the priorities assigned to tasks were allowed to change?
- Earliest Deadline First: assign highest priority to the task with earliest deadline

Example: Consider 2 periodic tasks
$\tau 1$ (50, 100) and $\tau 2$ (60, 150)

RMS Scheduling



EDF Scheduling



Key Result

- Earliest deadline first scheduling is optimal (single processor):
  - If a set of tasks are schedulable under dynamic priority assignment, EDF will produce a feasible schedule

- What about CPU utilization? How do we know a set of tasks are schedulable?
  - A dynamic priority schedule exists if and only if CPU utilization is less than 1

- Other dynamic priority assignments can be optimal as well (besides EDF)
  - Least slack time first (also called minimum laxity first) scheduling is optimal too!
  - At any time t, slack of a task is defined to be equal to (deadline – t – remaining execution time)
    - Least slack time assigns highest priority to the task with least slack

Static Vs. Dynamic Priority Assignment

- Since dynamic priority assignment provides better CPU utilization, why not use EDF all the time?
  - EDF (and LST) are complicated enough to have unacceptable overhead
  - Timing behavior of a system that uses fixed-priority assignment is more predictable
    - If a job overruns its execution time (under overload), high priority task will still get scheduled first in fixed-priority scheduling (graceful degradation) ➔ possible to predict which tasks will miss their deadlines

- Because of the above-mentioned reasons, fixed-priority scheduling is more prevalent in real-time systems

Common Misconceptions About Real-Time

- There is no science in real-time-system design
  - What do you think?
- Advances in supercomputing hardware will take care of real-time requirement
  - The old "**buy a faster processor**" argument…
  - What do you think?
- Real-time computing is equivalent to fast computing
  - What do you think?
- It is not meaningful to talk about guaranteeing real-time performance when things can fail
  - What do you think?
- Real-time systems function in a static environment
  - What do you think?

Summary

- Scheduling in real-time systems
- Utilization bound test, RT test
- Handling priority inversion:
  - Basic Priority Inheritance
  - Priority Ceiling Protocol
  - Priority Ceiling Emulation Protocol