Real-Time Scheduling: EDF and RM

Daniel Mosse
University of Pittsburgh

Daniel Mosse (mosse@cs.pitt.edu)

Acronyms

- RT = real-time (timeliness is as important as functionality)
- HRT = hard real-time (catastrophic consequences)
- SRT = soft real-time (typically monetary consequences)
- NRT = non real-time (i.e., general purpose systems)
- WCET = Worst-case execution time
- EDF = Earliest Deadline First
- RM = Rate Monotonic
- OS = Operating System

RT Scheduling

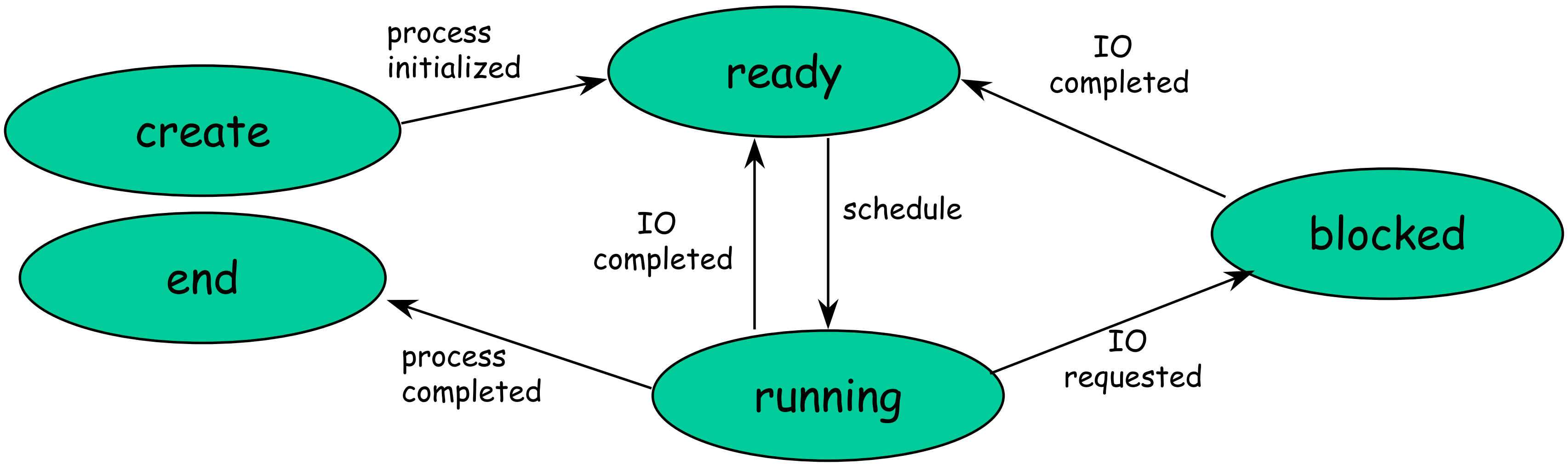
- What should the system know, in addition to NRT tasks?
- Two approaches:
  - OS supports RT. How can this info be transmitted to the OS?
  - Support for RT is outside, extra tool; OS supports fixed priority scheduling. What are the advantages and disadvantages?
- Hardware support? Timer card, with high resolution.
- CPU scheduling support: predictability.
- Same for disk, sensors, actuators, and other peripherals
- Low overhead a plus, but also in NRTSs

RT: Admission Control

- The basis of all HRT systems is that, for processes or threads to be created, need to pass *admission control*
- In NRT systems admission control typically is concerned with starvation of processes due to lack of resources
- In RT systems, the idea is the same, but more constraints are present:
  - All deadlines must be met (i.e., the response time is before deadline)
  - Enough instances of all resources must be present
  - Periodic invocations must be activated within certain latencies

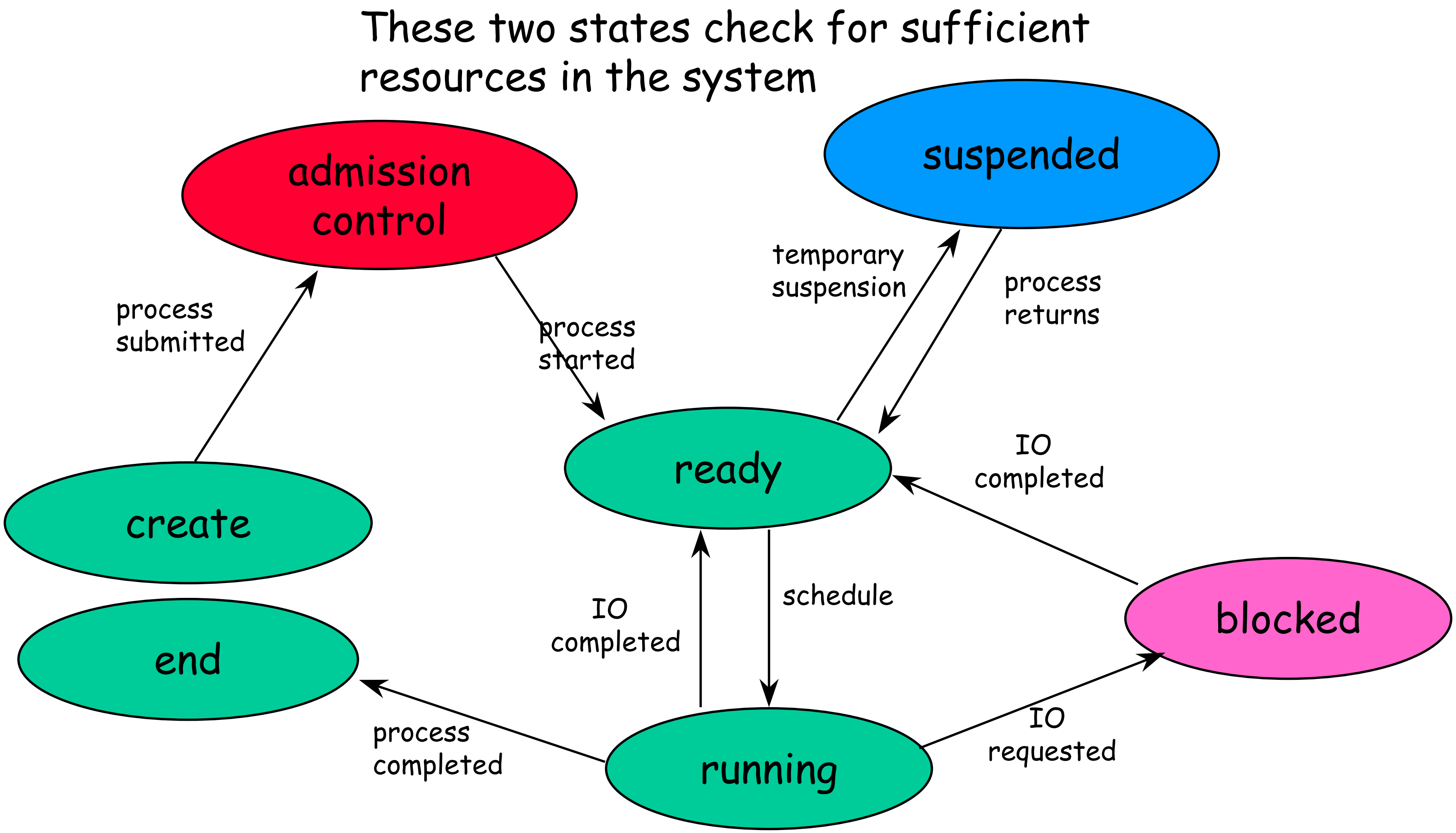
State Diagram

- Create: PCB and other resources are setup
- End: resources held are returned to the OS (freed)
- Context switching: saves HW context; updates PCB
- States are typically implemented as queues, lists, sets

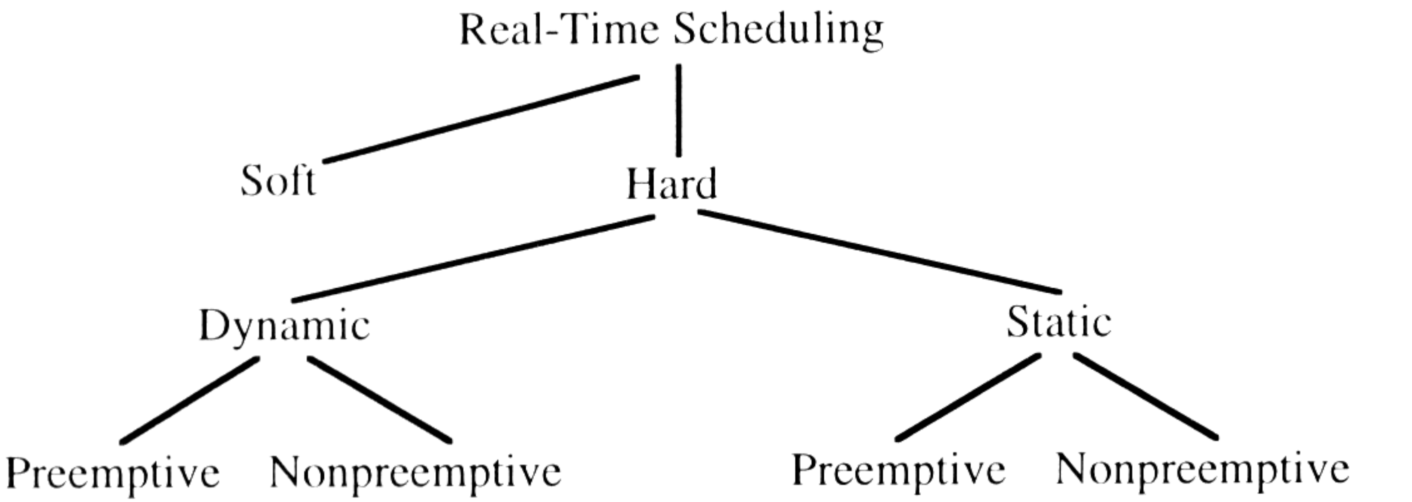


Daniel Mosse (mosse@cs.pitt.edu)

Complete State Diagram



Types of RT Scheduling



Source: Kopetz

- Dynamic vs. Static
  - Dynamic schedule computed at run-time based on tasks really executing
  - Static schedule done at compile time for all *possible* tasks
- Preemptive permits one task to preempt another one of lower priority, but non-preemptive does not require state to be saved… This lecture considers ONLY preemptive systems

Source: Koopman

Daniel Mosse (mosse@cs.pitt.edu)

Static vs Dynamic Schedules

- Static schedules are great in some systems
- Time-triggered schedules (build a priori) are also great in some systems
- BUT, sometimes, when things are dynamic, dynamic schedules offer more flexibility, easier maintenance, and better resource utilization
  - WCET is not the worst case
  - Tasks that are supposed to start are not ready to start
  - An urgent task is scheduled when it is scheduled, no earlier
  - Need to build a schedule to the Least Common Multiple of the periods of all tasks in the system (perhaps exponential)
  - When a task changes, need to rebuild the schedule

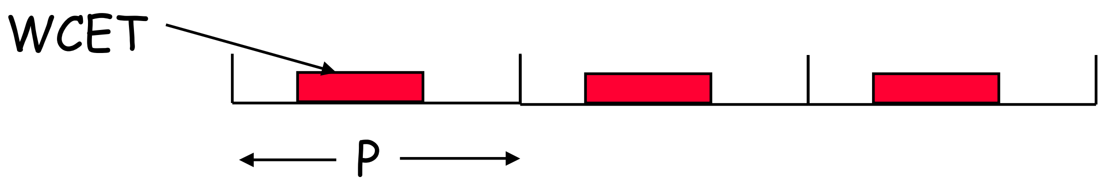
Dynamic (priority-based) Scheduling

- One would like to send the tasks to the system, and let the system execute them and guarantee deadlines are met
- For that, we need *admission control*: clearly, one cannot use more than 100% of the CPU cycles that exist
- Admission control is a way to analyze the tasks so that one can guarantee BEFORE RUNNING that deadlines are met
- Then submit the tasks to the system, and the scheduler knows how to schedule the tasks accordingly
  - Rate Monotonic schedulers give higher priority to tasks with smaller period (think of a smaller deadline!)
  - Earliest Deadline First schedulers give higher priority to tasks with (guess!) earliest deadline (again, think of a smaller deadline!)

Priorities as Scheduling

- In both dynamic scheduling algorithms that we consider here (EDF and RM), the priorities of the tasks are a guide for the scheduler to dispatch the task
- In EDF, it is the *explicit deadline* that functions as the priority, and therefore the programmer or system integrator has to know about deadlines
- In RM, the *period of the task* determines the priority, and therefore the system integrator has to have global knowledge of all tasks' periods (so that s/he can determine whether a task is higher or lower priority)
- In any case, the priorities can be manipulated in an explicit or implicit manner by the programmer or system integrator

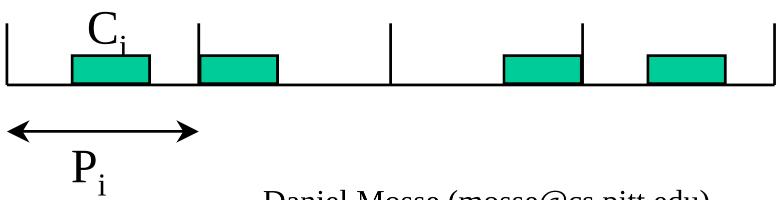
Dynamic Scheduling Notation



- Example is a periodic RT task, with 3 instances
- Assume non-preemptive system with <u>5 Restrictions</u>:
  1. Tasks $\{T_i\}$ are periodic, with hard deadlines and no jitter
  2. Tasks and instances are completely independent
  3. Deadline = period ($p_i = d_i$)
  4. WCET c_i is known and constant
  5. Context switching is free (zero cost)

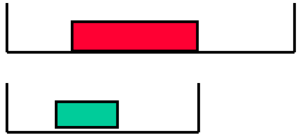
Earliest Deadline First (EDF)

- Compare with your own tasks, such as work tasks
- Preemptive *or* non-preemptive, EDF is *optimal* (in the sense that it will find a feasible schedule if one exists)
- *A feasible* schedule is one in which all deadlines are met
- EDF works with preemptive *periodic* tasks: there is a *minimum* interarrival between instances. Could instances be separated by more than one period? How about less?
- Only requirement is to meet all the deadlines
- With a single task, the requirement is: $U = c/p \leq 1$, that is, a task must be executable in a single CPU

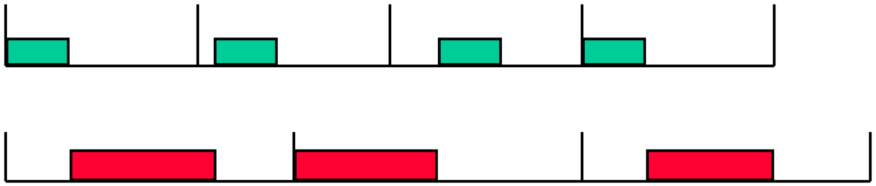


EDF (cont)

- Did you notice the characteristic of EDF: the priority of the tasks is not fixed, relative to each other
- Again, compare with daily tasks: which has priority?
- For example, let there be 2 tasks ready in the system



- Which executes first, when? What’s the order?

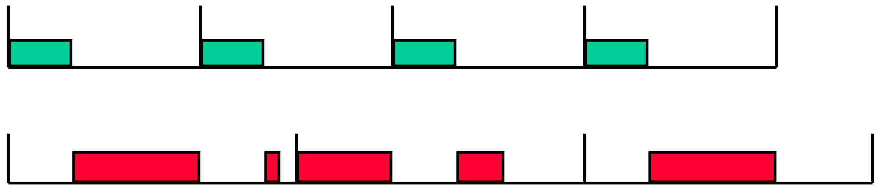


EDF (cont)

- The same math for a single task also works for multiple tasks:
  - A schedule is feasible iff U < 1, that is, $\sum c_i / p_i \leq 100\%$
  - The share (utilization) of each task is obviously also restricted, but the combined utilization cannot exceed 100%
- For EDF, every time a new instance is ready, there is a need for checking whether this task is the highest priority one
- The relative priority of tasks (see previous slide) can change, depending on the instances, time, etc.
- Can we do better than having to perform all these checks?

Rate Monotonic (RM)

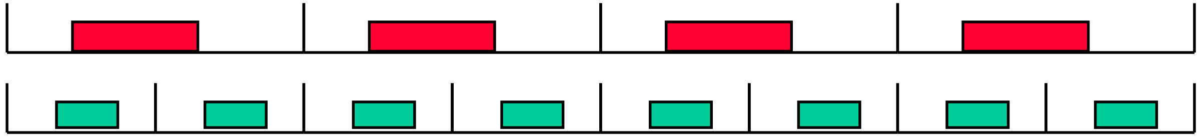
- In RM, the priority of the tasks is fixed with respect to each other.
- The priority is computed as the inverse of the period.
- Dissect the name: *rate* (which means it depends on the period) and *monotonic* (increases or decreases only)
- Reasoning behind it: the more frequent a developer wants to do a task, the higher priority it should have.
- How efficient is it? as efficient as EDF?



Daniel Mosse (mosse@cs.pitt.edu)

RM admission control

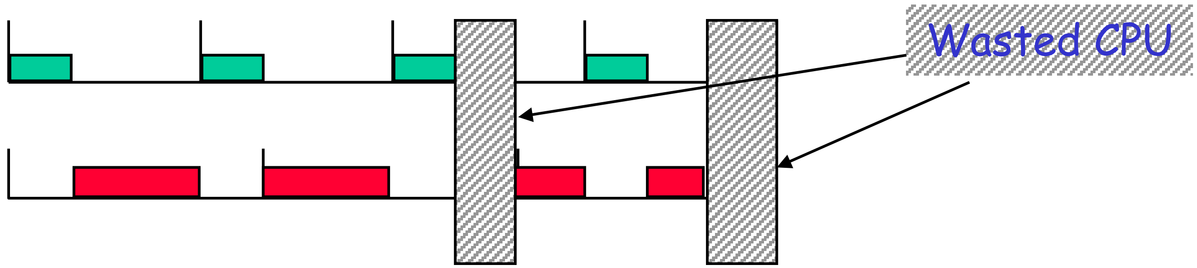
- Let us consider the easy and good case for RM: harmonic periods (that is, all periods are multiples of each other)
- In this case, the admission control for RM is the same as it is for EDF
  - A schedule is feasible iff U < 1, that is, $\sum c_i / p_i \leq 100\%$
  - The share (utilization) of each task is obviously also restricted, but the combined utilization cannot exceed 100%
- Note that the task with the shorter period will also be the task with the earliest deadline at any given time



Daniel Mosse (mosse@cs.pitt.edu)

RM (cont)

- Which scheduling policy is more efficient? Can RM be any more efficient than EDF? Can RM be any more efficient than EDF?
- Depends on how one looks at *efficiency*, which can be defined as less dispatching (context switching) overhead, can be defined as higher resource utilization without considering overhead, or a combination thereof
- In general, RM may allow for less CPU to be used. Example:



RM (cont)

- So, in general, the CPU cannot be fully utilized when tasks are scheduled following RM, and the admission control has to reflect this issue.
- This is because RM is for fixed-priority tasks (tasks' priorities do not change in time, they're always the same, and therefore their relative priority is also the same)
- Liu and Layland devised a test to check whether task sets could be scheduled:
  - If $\sum c_i / p_i \leq n (2^{1/n} - 1)$, then all n tasks will meet their deadlines
- However, RM can be implemented in hardware
  - How? (see next slide)
  - Is it worth it?
  - It reduces the scheduling overhead, memory overhead, stack overhead

Implementing RM in hardware

- Good for control systems, in which sensors are separate devices: temperature, pressure, RPM, acceleration, smell, etc
- Devices also must be able to send signals to the CPU to activate the tasks on a periodic basis
- Associate each device to an interrupt priority, according to the inverse of the period
- Tasks are handled by a PIC (programmable interrupt controller) which activates interrupt service routines (ISRs)
- The tasks must be cooperative, since they will execute on the same stack (like threads, but not really threads)
- Advantages: Low context switch overhead, no scheduling overhead, low memory allocation overhead, highly collaborative

Implementing RM in "regular" OSs

- Most unix-like OSs nowadays provide the means for what they call "real-time priorities"
- These are tasks that run above the NRT tasks, at fixed priorities
- The OS does not need to know what the period and/or deadline for the tasks are, but the system integrator has to determine the priority of each task
- Since RM has a well-defined, fixed-priority relation between priority and period, it's easy to do.
- Note that this only defines the order to run the processes, but does NOT make it a RT OS!!! All the other issues (interrupts, disks, deamons, etc) are still NRT!!!

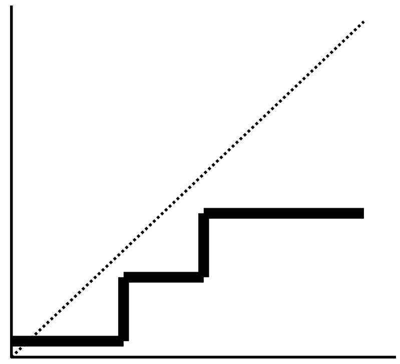
Exact Characterization and Response Time Analysis

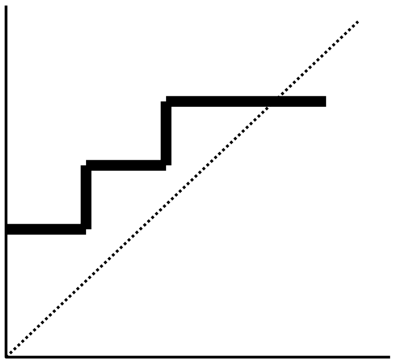
- How can the admission control tests (also called feasibility test) be simplified or complicated?
- The LL
- Take the task with the highest priority. It's response time is simply C_1
- T_2 has a different response time. How can we compute it?
- $R_2 = C_1 + C_2$. Is this correct? Why or why not?
- $R_2 = (P_2 / P_1)\ C_1 + C_2$.
- Is this correct? Why or why not?

Daniel Mosse (mosse@cs.pitt.edu)

RM response time analysis

- $R_2 = \lceil P_2 / P_1 \rceil \, C_1 + C_2$.
- Is this correct? Why or why not?
- What is/are the condition/s we have to check?
- How many periods do we have to check this condition?
- May also be called *fixed point computation*, since all this is done when response time does not increase anymore

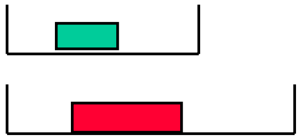




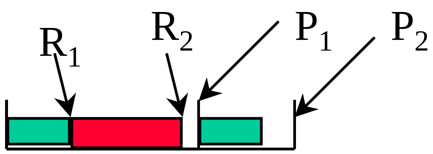
RM: response time analysis

- Clearly, $R_2 \geq C_1 + C_2$, but it may be that $R_2 > C_1 + C_2$
- This will happen if the second instance of T_1 preempts T_2
- In this case, $R_2 \geq 2C_1 + C_2$. In fact, if the third instance of T_1 preempts T_2 also, $R_2 \geq 3C_1 + C_2$.
- We can derive a recurrence relation, and keep increasing R_2
  $R^2{}_2 = (R^1{}_2 / P_1) C_1 + C_2$
- The recurrence stops when
  - $R^i{}_j = R^{i+1}{}_j$ (response time does not increase further: accept task)
  - $R^i{}_j > D_j$ (task misses the deadline: reject task)
- This test has to be done in increasing order of priority; note that this is only for FIXED priority scheduling of RT tasks

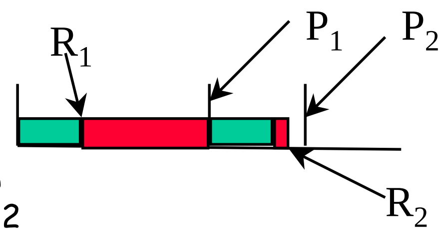
RM: response time analysis algorithm



- Again, consider only 2 tasks for simplicity
- $R_1 = C_1$ no problem. R_2?
  - case 1: $C_2 \le P_1 - C_1$ which causes T_2 to finish before the end of P_1, which means that $R_2 = C_1 + C_2$



  - case 2: $C_2 > P_1 - C_1$ which causes the new instance of T_1 to preempt T_2, which means that $R_2 \ge 2C_1 + C_2$



  - If the preemption is done once, $R_2 = 2C_1 + C_2$
  - However, by preempting T_2 the response time of T_2 is postponed, which may cause preemption to occur twice, and thus $R_2 = 3C_1 + C_2$
  - And so on

Daniel Mosse (mosse@cs.pitt.edu)

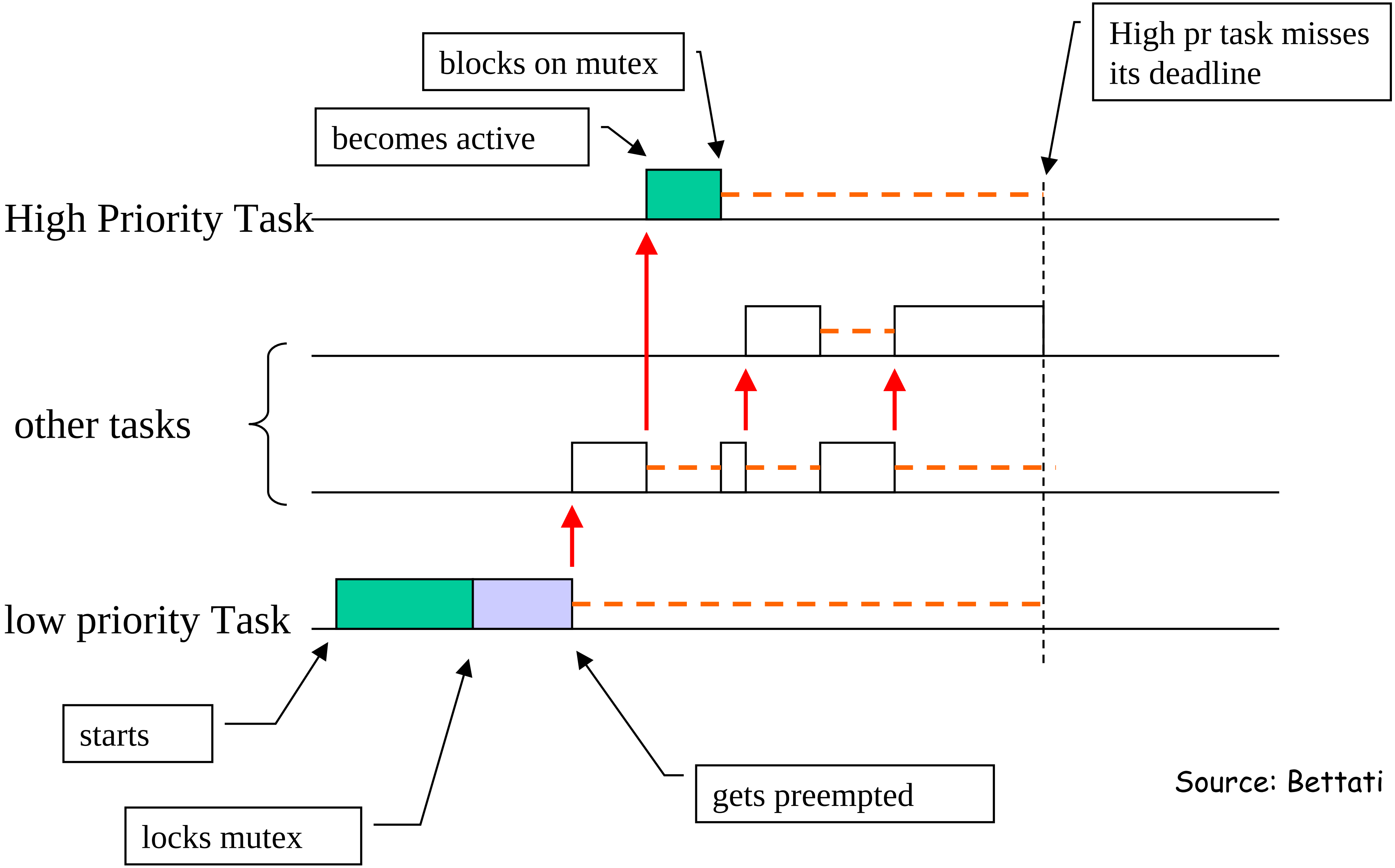
RM: response time analysis example

- Consider the following tasks <C, P>
- T_1=<1,2> T_2=<2,5>
  - $R_1 = C_1 = 1$ but R_2?
  - $R^0_2 = 2+1 = 3$, at least; that is larger than P_1
  - $R^1_2 = \text{ceil}(3/2)\ 2 = 2 * 2 = 4$
  - $R^2_2 = \text{ceil}(4/2)\ 2 = 2 * 2 = 4$
  - Since $R^1_2 = R^2_2$ the task is accepted. on to the next task
- We do this type of computation for every task *in order of priority*

Resource Sharing

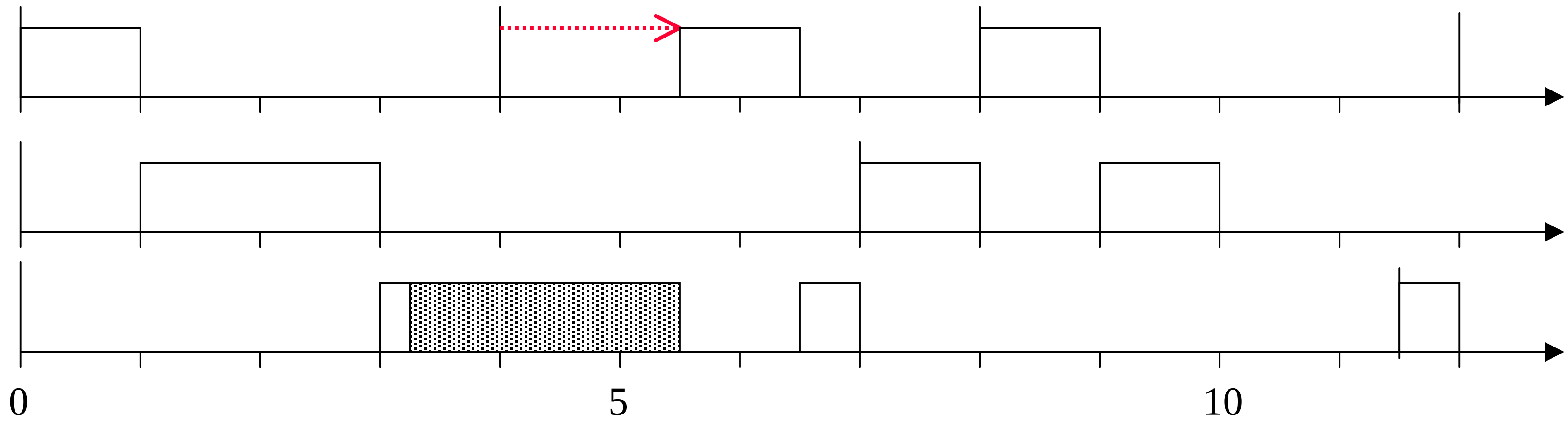
- When resources are shared (resources can be anything threads use, such as memory locations, variables, devices, etc), there has to be a synchronization mechanism
- Usually, semaphores and/or lock variables are used (rarely monitors are used: do not pause this program for a special monitors insert in the last slide)
- Semaphores may cause priority inversion: a high priority task is blocked by a low priority task (same with non-preemptive scheduling)

Priority Inversion



Source: Bettati

Disallow Preemption of Tasks in Critical Section



- Analysis identical to analysis with non-preemptable portions
- Define: β = maximum duration of all critical sections
- Task T_i is schedulable if

$$\sum_{k=1}^{i} \frac{e_k}{p_k} + \frac{\beta}{p_i} = F_X(i)$$

Source: Bettati

- Problem: critical sections can be rather long.

Priority Inheritance

- Jobs that are not blocked are scheduled according to the scheduling algorithm
- Priority Inheritance:
  - Basic strategy for controlling priority inversion:

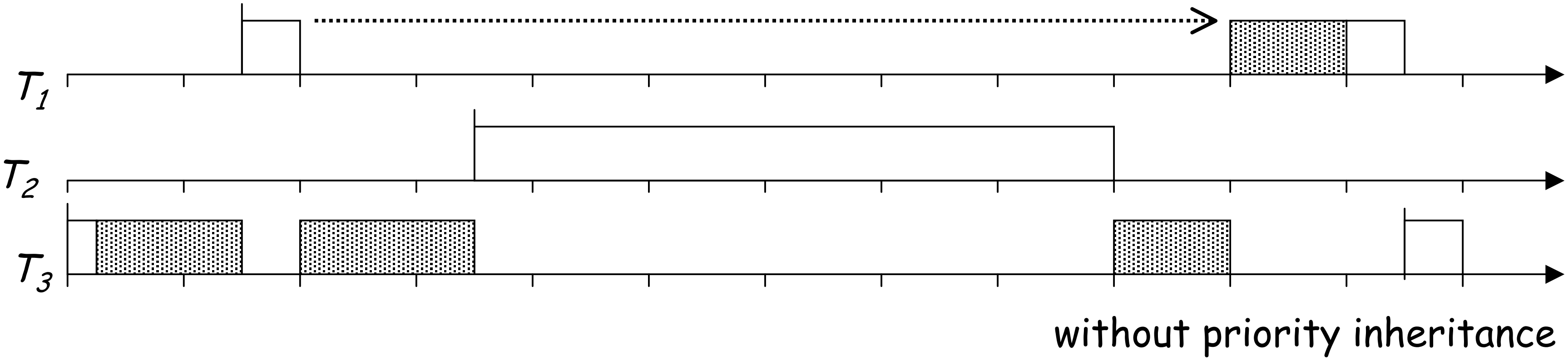
    Let π be the priority of J

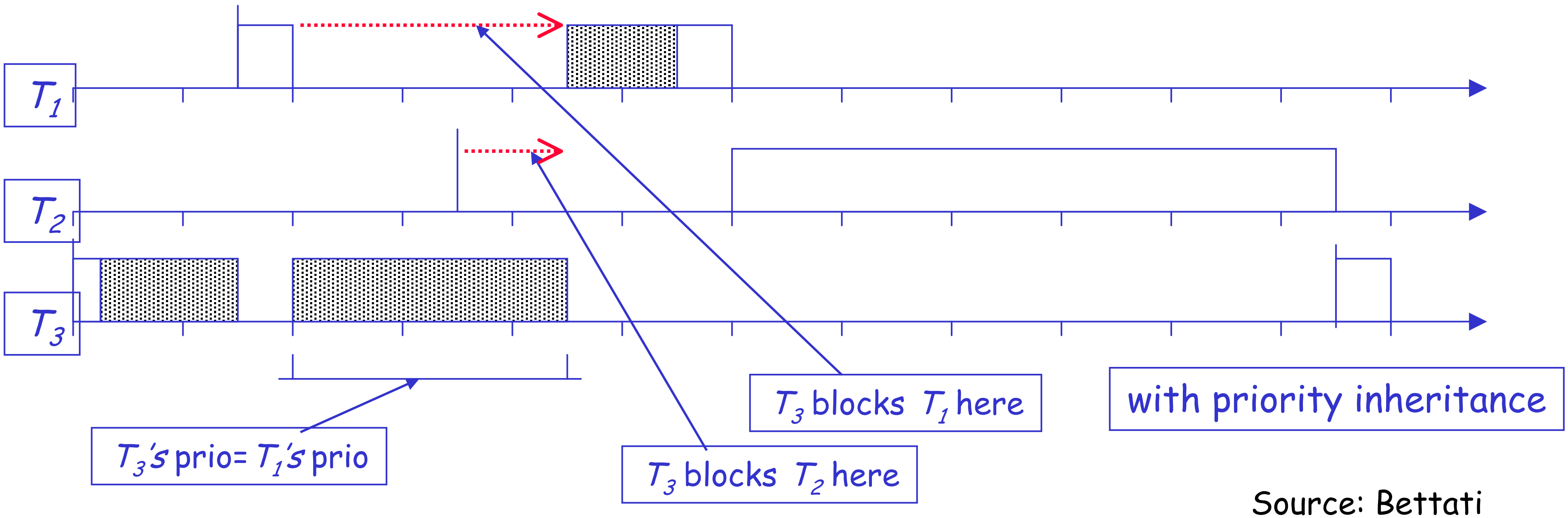
    and π' be the priority of J'

    and $\pi' < \pi$

    then the priority of J' is set to π whenever J is blocked by J'
- Priority Inheritance is transitive

Priority Inheritance controls PrioInv

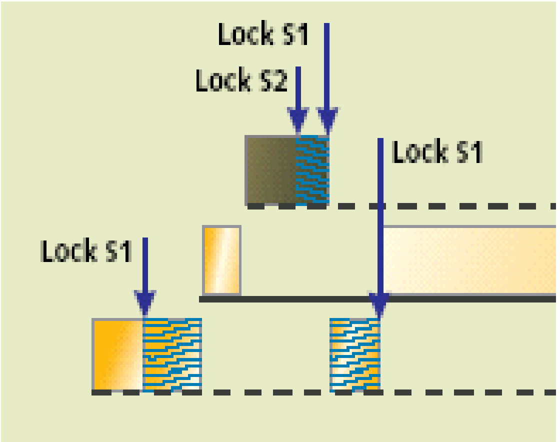




Source: Bettati

Problem with priority inheritance

- The main problem with priority inheritance is that it may cause deadlocks
- If there are more than one resource, and the tasks are going to use them, a deadlock is possible. Example:
  - Low priority task, L, locks a resource (acquires S1)
  - L gets preempted by a higher priority task, H, locks a second resource (acquires S2). Since L is not using S2, no priority is inherited
  - H tries to get S1, and blocks
  - L is promoted, resumes, tries to get S2, and blocks
  - DEADLOCK!



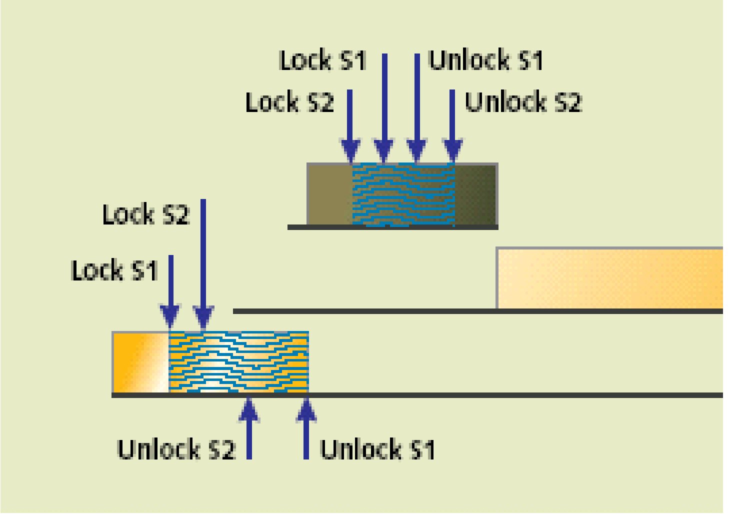
Source: Tindell'00

Priority Ceiling?

- The Priority Ceiling Protocol solves the deadlock problem by raising the priority of the task to the highest priority of all the tasks that may lock the resource in question
- When a task Ti attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceiling of all semaphores currently locked by tasks other than Ti
- When a task blocks other tasks (directly or indirectly), it inherits the highest of their priorities

Priority Ceiling Protocol: Example

- Similar to the example of deadlocks
- L locks S1, but now it immediately gets promoted
- So, it continues to execute, acquires S2 without being preempted by other higher-priority tasks
- When it unlocks (releases) S1, it is returned to its original priority
- THEN (and only then) it can get preempted
- NO DEADLOCK

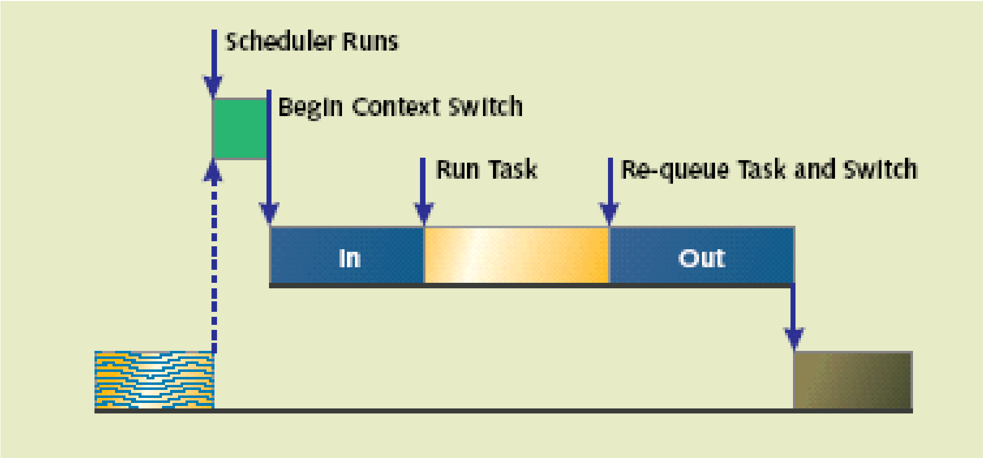


Source: Tindell'00

Response time analysis with shared resources

- Before, we had RM deal with only independent tasks
- Adding resources adds the problem of *priority inversion*, and therefore, a means of dealing with it is needed.
- We know that priority inversion will cause some high priority tasks to be delayed. We account for this extra delay in doing schedulability analysis?
  - Add a term for blocking (or PrioInv) in the utilization equations for RM
  - This extra term accounts for the amount of time that another task may be blocking this task during execution

Overheads

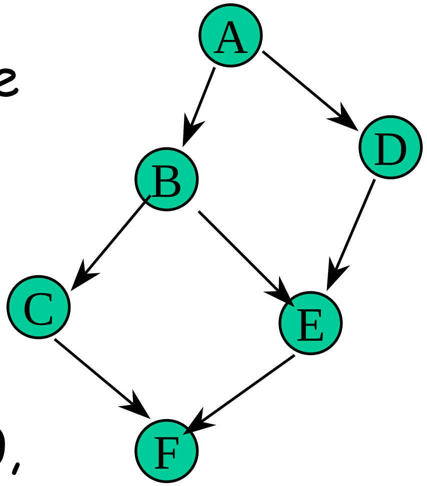


- Up to now, we have an ideal system, with the instantaneous preemption, context switch, scheduling, etc
- How can one incorporate these overheads in the feasibility tests? How much will they influence the issue?
  - It's not free, but as CPUs gets faster it gets cheaper compared to real time
- In a similar way to the priority inversion issues, we can add another term to the utilization and feasibility equations, reflecting the overheads.
- Biggest issue seems to be *repopulating the cache* on a context switch

RM/EDF during overloads

- Overloads can be caused when a task takes a little longer than WCET; possible causes
  - WCET tool not accurate
  - Tests didn't cover whole input spectrum
- In RM, the tasks with higher priority will always run, and the tasks with lower priority will suffer
  - Not fair, since offending task may be high-priority task
  - Predictable: high-priority tasks are more important (are they?)
- Although EDF is more efficient (can get 100% utilization), it suffers from a big problem under overloads:
  - If the schedule is tightly packed (all tasks finish exactly at their deadlines) and the first task takes a little longer than WCET, then all tasks will miss their deadlines

EDF with precedence constraints

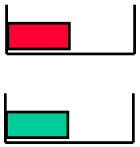


- Consider a task graph: nodes are tasks, edges are precedence constraints
- One only deadline for the entire application
- How can we apply EDF?
- Easy solution considers semaphores
- Complicate problem: all tasks are ready at time 0, and no semaphores are used.
- A single graph will have a single deadline
  - Do a topological sort of the graph
  - Start from the bottom and reduce the deadline of each task by a little bit (it does not matter how much)
  - Consider all tasks to be independent
  - Run EDF

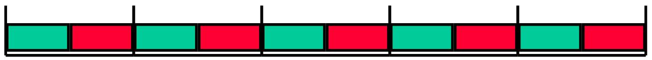
Least Laxity Scheduling

- Least Laxity is also optimal for single processors (like EDF)
- If sum of task utils is less than 100%, task set is feasible
- Algorithm: dispatch the task with the smallest laxity, which is the largest amount of time that a task can be delayed (some type of procrastination index)
- In a sense, it is similar to EDF, in that it runs the *most urgent* tasks in the set (the metric by which *urgency* is measure differs, though)
- A problem occurs with LLF, when tasks have the same laxity: too many preemptions

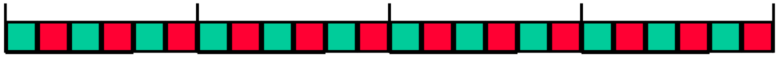
Least Laxity Scheduling (cont)



- Take 2 tasks with requirements <3,6>
- The EDF schedule would be the following:



- However, in the LLF schedule, both tasks have the same laxity when they become ready.
- Then, one task runs, its laxity remains the same, while the other task's laxity decreases. LLF schedule is



- As can be seen from the figures, the schedule for LLF has many preemptions/context switches

Summary

- Dynamic Scheduling is a Good Thing, when your system is somewhat predictable (periodic)
- Allows for flexibility, but designers have to beware of
  - Priority Inversion
  - Deadlocks
  - Overhead
- Designers typically choose between RM and EDF
- EDF is more efficient in general, but RM is as efficient (and more, if considering overhead) for harmonic task sets