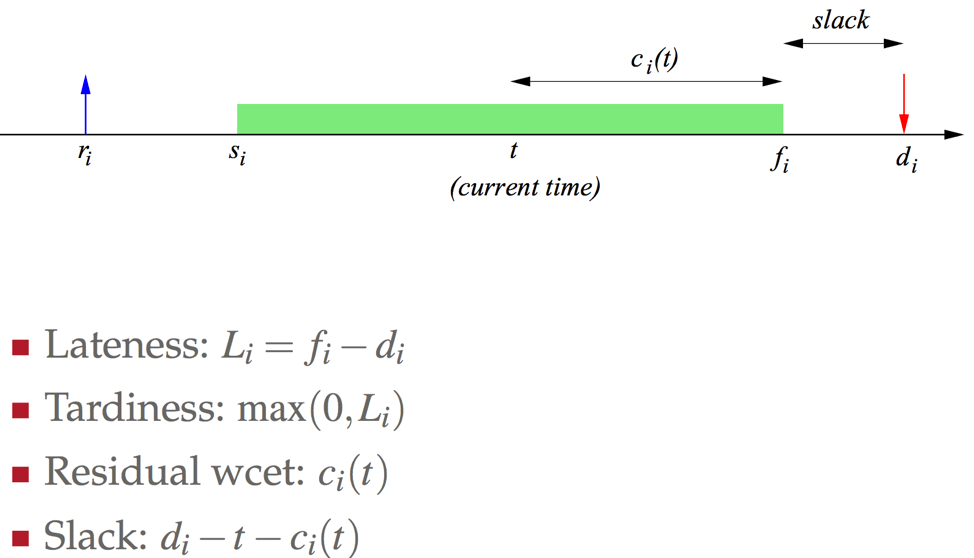
Lecture 14: Definitions

*Worst case execution time*: ci(t) = fi - si



Slack is defined by t ,ci(t) rather than fi because we are not far enough in the execution to know the finish time. Only know finish time once a process terminates.

Lecture 15: Scheduling Algorithms

**Real Time Systems**

*Job:* sequence of operations that is executed by processor

* Request time/ arrival time ri
* Start time si
* Finishing time fi
* Absolute deadline di

*Task:* sequence of jobs

**Scheduling Algorithms**

*Preemptive*: The running task can be temporarily suspended to execute another task; inspect new processes priority

*Non-preemptive:* The running task cannot be suspended until completion or until it is blocked

* Relinquish CPU voluntarily, quantum expiration, or system call that makes you blocked

*Static*: All processes’ priority previously assigned and don’t change

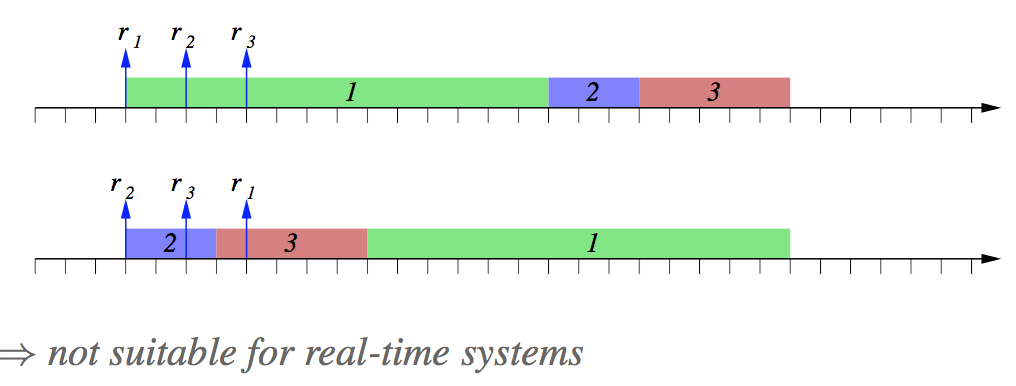
*Dynamic*: Process in middle of it’s schedule, changes its priority; forces you to re-evaluate the schedule

*Offline*: make the schedule on paper and implement

* All tasks arrive at the same time (decide who runs first)
* Handling periodic tasks
* Very reliable if you know your environment

**First Come First Serve**

* There is no quantum, a task can only execute when the tasks before have completed
* Non-preemptive
* Dynamic
* Online
* Heuristic
* Very unpredictable: response time depends strongly on task arrivals (response time = f – r; from arrival to completion)

****

**Arrivals are a permutation of each other**

**Average Response time: blue arrow to end of green block**

*Average Response time*: a measure of merit; how productive a scheduler can be

**Shortest Job First**

Pick the task with the shortest computation time (worst case execution time)

* Non-preemptive or preemptive
* Static wcet ci is known and fixed
* Online or offline
* **Minimizes the average response time** (See notecard for proof)
* **SJF is NOT suitable for real-time in the sense of feasibility**

**Priority scheduling**

Tasks with highest priority is selected first; if same priority FCFS is used

* Preemptive
* Static/Dynamic
* Online

**Issues**

* Starvation: low priority tasks may experience very long delays because of preemption by higher priorities
* SOLUTION: *aging*: priority increases with waiting time
  + Quantum guarantees you will not starve as well

Faking the previous algorithms we have seen

* IF pi ∝ 1/ci :shortest job fist
* IF pi = constant : FCFS
* IF mimic FCFS and have a priority scheduler; pi = 1/ri

**Round Robin**

The queue is FCFS, however each task cannot execute more than Q time unites (quantum)

When Q time units have elapsed, the task is put into the ready queue (LAB 4)

* For small Q, executing on a virtual processor
* For large Q, RR = FCFS

Lecture 16: Real-time Algorithms

**Multi-level Scheduling**

Can mix and match different scheduling algorithms

**Real-Time Algorithms**

***GOAL: meet the deadlines***

Tasks can be scheduled by

* Relative deadlines Di (static)
* Absolute deadlines di (dynamic)

**Earliest Due Date**

Select the task with the earliest *relative* deadline

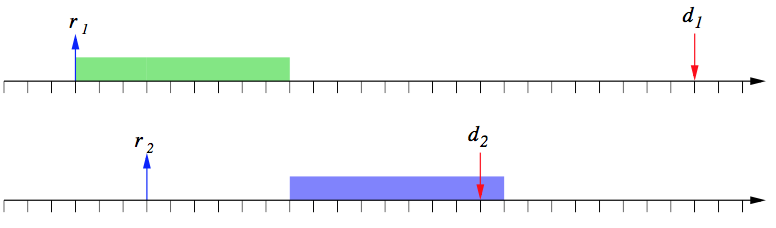
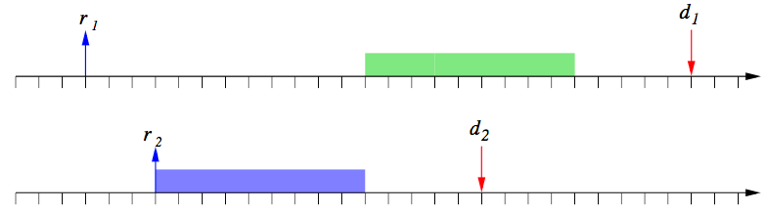
* All tasks arrive simultaneously
* Fixed priority (Di is known); offline
* Preemption is not an issue. ONLY look at when deadline is to schedule
* **Minimizes the max lateness (L = f - d)**
* **If the schedule is feasible, EDD will give a feasible schedule**

*Jackson’s Rule:* Given a set of n independent tasks, any alg that executes the tasks in increasing deadlines is optimal for max lateness

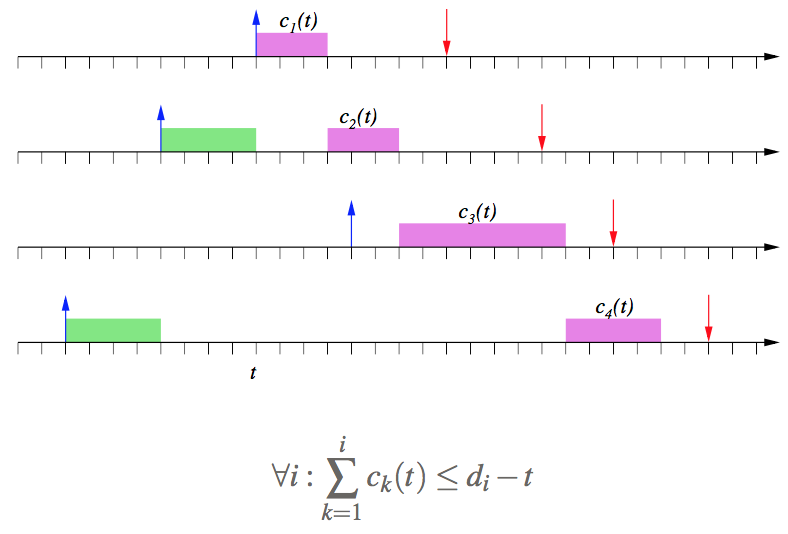
* If Lmax(σ) is max lateness of a schedule, the for all σ, **Lmax(σEDD) ≤ Lmax(σ)**
* See notecard for proof

**Earliest Deadline First**

Online version of EDD. Select the task with the earliest absolute deadline

* Tasks may arrive at any time
* When new task arrives (have preemption), dynamic priority (di depends on when the tasks arrive); always preempt/reassign priorities by deadline
* **\*\*\*\*\*\*\*\*\*\*Minimizes the max lateness (L = f - d) IF preemptive**
  + **Under non-preemptive, EDF is NOT optimal, UNLESS alg has knowledge of future**
  + ****
  + **Example of EDF without preemption**
  + ****
  + **Example of EDF w/o preemption but knowledge of future; now meets deadline by delaying first task**
* **If the schedule is feasible, EDF will give a feasible schedule**

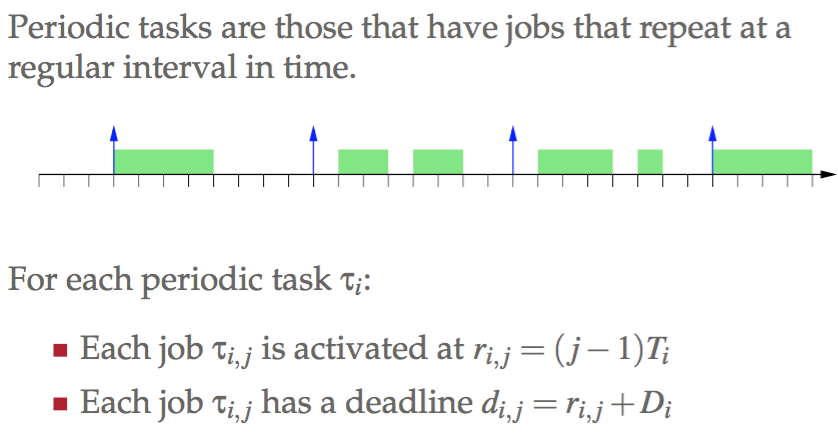
**EDF Feasibility**

****

Glitch here to handle the arrival of P3; but P3 doesn’t go into the CPU

Lecture 17: Periodic Scheduling

*Periodic tasks*: throw a new job at regular time intervals



Periodic job between that period must be done before the next arrival

**Timeline Scheduling**

*Process*

* Find GCD: that will be your tick marks; how often a new task will arrive
* Find LCM: that will be the period that you will repeat; how long the program has to be

Can implement as an ISR, have variable that remembers state. “If in first sub-period do…”

Advantages

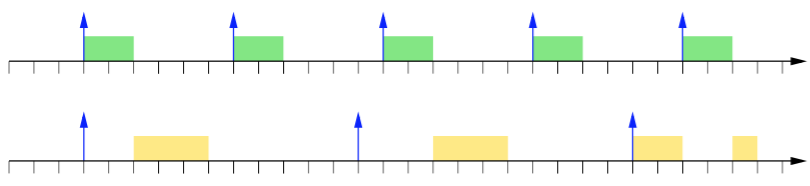
* Very simple, predictable
* Low run-time overhead
* Jitter can be controlled

Disadvantages

* Not robust to overloads/overruns (delays difficult to recover from)
* Difficult to expand the schedule
* Not easy to handle aperiodic activities

**Rate Monotonic Scheduling**

Each task is assigned a fixed priority proportional to its rate (Hence, higher priority for shorter cycle time)



Yellow does not run because the scheduler knows green is coming, it may choose to defer yellow

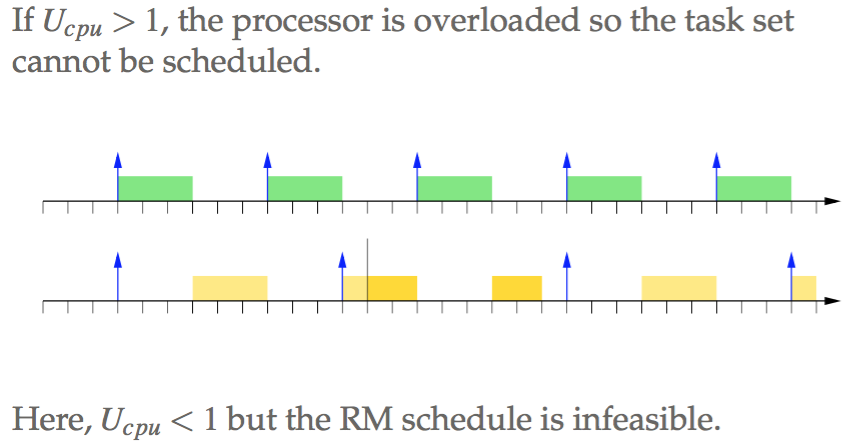
**Feasibility**

Ci : computation time; time units using CPU

Ti : idle time

Pressure a particular task puts on CPU/ how much the CPU is utilized: Ui = Ci / Ti

* Hence CPU busy Ui of the time
* CPU idle 1 – Ui of the time
* Note ΣUi ≤ 1 is a ***necessary*** *but not sufficient condition for feasibility*

**

Can by hand make this match possibly. But RM does not do that.

However, RM can succeed at capacity; increasing C1 or C2 causes missed deadlines

* **KEY**: The main disadvantage of RM is that you can hardly ever use CPU to max capacity because its too difficult to produce a feasible schedule
* Can’t be guaranteed feasibility EVEN if ΣUi < 1
* If there exists a fixed priority assignment which leads to a feasible schedule, then RM produces a feasible schedule (this is def on RM!)

**Resource Constraints**

* Blocking on a CS can increase the delay
* Lower priority process could run “forever” in NCS and conflict over CS -> this is priority inversion

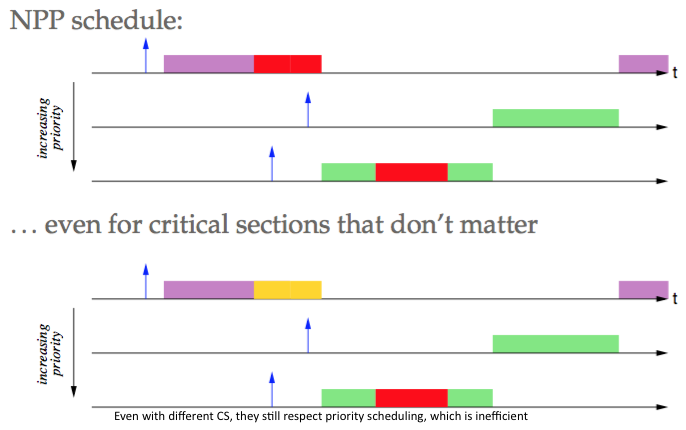
*Priority Inversion*: a high-priority task was blocked by a lower priority task for an unbounded interval of time

Lecture 18: Priority Inversion

**Non- Preemptive Protocol**

* Preemption is NOT allowed in CS
* Implementation: when a task enters a CS, then increase the priority to max priority

Disadvantage

* High priority tasks that do not interfere with the CS will be blocked
* 
* Even with different CS, there is still priority inversion; as yellow CS has nothing to do with other processes; “overkill”

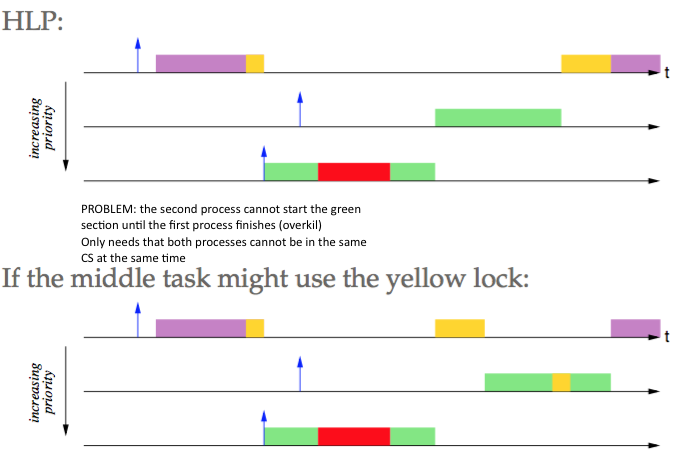
**Highest Locker Priority**

* A task in CS gets the highest priority among the tasks that MIGHT use the SAME CS
* Implementation: when a task enters a CS, inc priority to the max value of the tasks that MAY access the CS

Disadvantage

* A task could be blocked because it MIGHT enter the CS not because it is ACTUALLY in the CS.

Priority scheduling: usually online; It could be offline, if you knew all the data

* 
* Again, “overkill” P2 green NCS could have gone before P1. Instead must yield all the way

**Locks**

* Possible deadlock; avoid nested locking. Otherwise, use a predetermined order to acquire locks

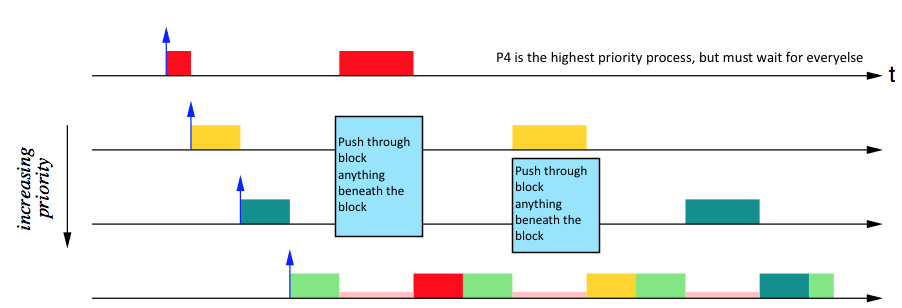
**Priority Inheritance Protocol**

* A task in CS increases its priority *only* if it blocks other tasks
  + If you’re in CS but no one else cares then don’t increase priority
* A task in CS inherits the highest priority among the tasks it blocks

**Blocking**

* *Direct*: task blocked on a lock
* *Push-through*: task blocked because a lower priority task inherits a higher priority

**Chained Blocking**



**Priority Ceiling Protocol**

* Offline: ceiling; max priority of all that tasks that use that lock
* Online (dynamic): a task can enter the CS only if it’s priority > max of all ceiling for all the locks that are currently **active**
* Like in PIP tasks inherit the highest priority of the tasks they block

**Overall**

* Every algorithm solves priority inversion
* Difference: performance and complexity
* Choose the simplest algorithm