ECE 350 Real-time Operating Systems

Lecture 6: Real-time Systems

Prof. Seyed Majid Zahedi

https://ece.uwaterloo.ca/~smzahedi

Outline

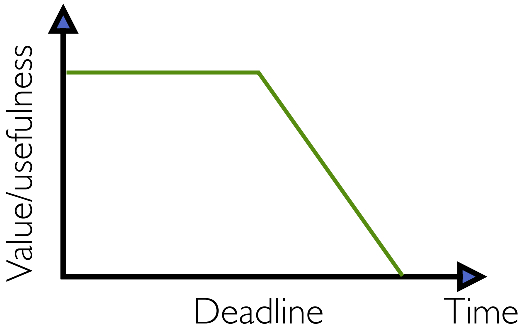
- Real-time systems
    - Definitions and features
- Real-time operating systems
    - Desirable properties, interrupt handling, memory management
- Uniprocessor real-time scheduling
    - RM, EDF, LLF, …
    - Priority inversion
- Multiprocessor scheduling
    - Different scheduling classes, remote blocking

Real-time Systems (RTSes)

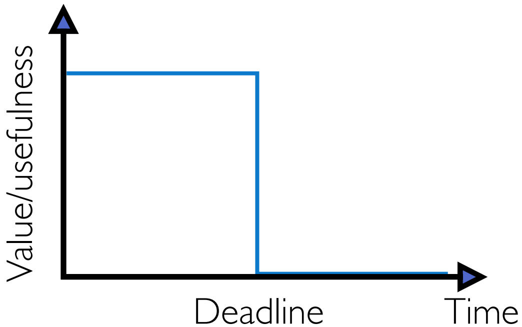
- Definition
  - Systems whose correctness depends on their temporal aspects as well as their functional aspects
- Performance measure
  - Timeliness on timing constraints (deadlines)
  - Speed/average case performance are less significant
- Key property
  - Predictability on timing constraints

Types of Real-time Systems

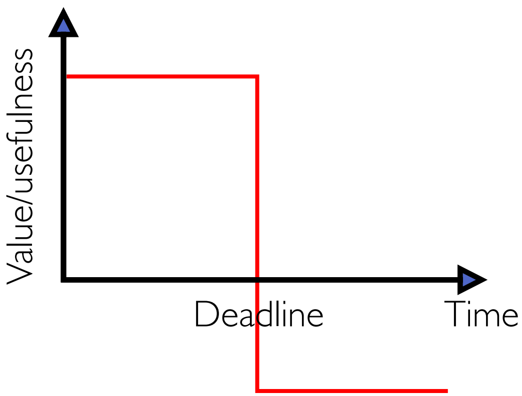
- Soft: must try to meet all deadlines
    - System does not fail if a few deadlines are missed



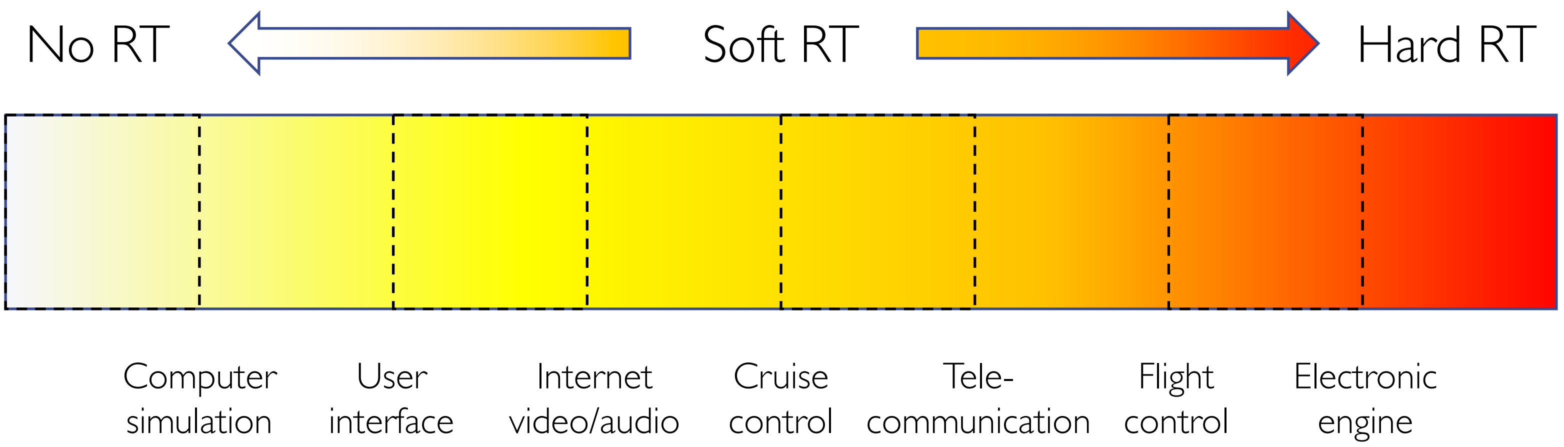
- Firm: result has no use outside deadline window
    - Tasks that fail are discarded



- Hard: must always meet all deadlines
    - System fails if deadline window is missed



Real-time Spectrum



General-purpose OS are Inadequate for RTSes

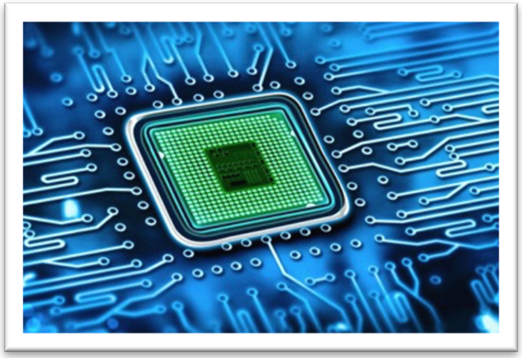
- Multitasking/scheduling
  - Provided through system calls
  - Does not take time into account and could introduce unbounded delays
- Interrupt management
  - Achieved by setting interrupt priority more than process priority
  - Increases system reactivity but may cause unbounded delays even due to unimportant interrupts
- Basic IPC and synchronization primitives
  - May cause priority inversion
  - Causes unbounded delays

Real-time Operating System (RTOS): Desirable Properties

- Predictability
  - Guaranteeing in advance deadline satisfaction
  - Notifying when deadline cannot be guaranteed
- Timeliness
  - Handling tasks with explicit time constraints
- Fault tolerance
  - Avoiding crashes even with HW/SW failures
- Design for peak load
  - All scenarios must be considered
- Maintainability

Microarchitecture

- I/O devices and CPU typically share the same bus
  - DMA steals CPU memory cycle to transfer data (cycle stealing)
  - CPU waits until the transfer is completed
    - Source of non-determinism!
  - Possible solution: time-slice method
    - Each memory cycle is split in two adjacent time slots one for CPU one for DMA
    - More costly, but more predictable!



- Caches speedup execution by keeping data close to CPU
  - The same load/store instruction could experience different delays depending on hitting or missing in cache ⇒ source of non-determinism!
  - Possible solution: processors without cache
    - Slow execution, but more predictable!

System Calls and Interrupts

- System call handler is usually non-preemptable
  - Real-time task could be delayed while handler runs
  - Delays could lead to missed deadlines
  - Possible solution: make all kernel primitives, including system call handlers, preemptable!

- Interrupt handler should remain non-preemptable
  - Interrupt handler runs immediately when interrupts happen
  - While handler runs, execution of interrupted task is delayed
  - Solution1: disable all interrupts, only keep timer interrupt, tasks directly communicate with any I/O devices they need, data is transferred by polling
    - + Unbounded delays due to unexpected device handling is eliminated
    - + Time for data transfers can be estimated precisely
    - + No change of kernel is needed when adding devices
    - – Performance of processor is degraded (busy waiting)
    - – Tasks must know low level details of drives

System Calls and Interrupts (cont.)

- Solution II: disable interrupts, only keep timer interrupts, handle I/O by timer-activated device routines within kernel tasks, data is transferred by polling
  - \+ Unbounded delays due to unexpected device handling is eliminated
  - \+ Execution time of periodic device routines can be estimated in advance
  - \+ Hardware details are encapsulated in dedicated device routines
  - − Performance of processor is degraded (still busy waiting for I/O)
  - − Kernel has to be modified every time new device is added
  - − Communicating with devices involves more inter-task communications than first solution

- Solution III: enable interrupts, reduce device drivers to least possible size, driver activates proper task to handle device, kernel runs driver-activated tasks like any other task, user tasks may have higher priority than driver-activated tasks
  - \+ Busy waiting is eliminated
  - \+ Kernel doesn't need to be modified when adding new devices!
  - o Communicating with devices may still involve some inter-task communications
  - o Unbounded delay due to unexpected device handling is not eliminated but is dramatically reduced

Memory Management

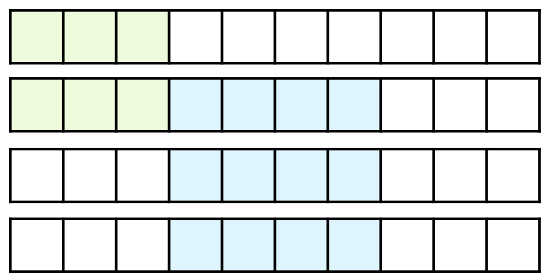
- General-purpose OSes allow each process to access only its own virtual address
  - Virtual and physical address spaces are divided into virtual and physical *pages*
  - Virtual to physical page translations are stored in memory as *page tables (PT)*
  - To speedup memory accesses, PT entries are cached in *translation lookaside buffer (TLB)*
  - TLBs introduce non-deterministic delay (hit or miss)
  - Possible solution: avoid using virtual memory, allow user tasks to use physical memory

- When memory is full, most OSes swap out some pages to make room for others (this is called *paging*, more about it later!)
  - Accessing evicted pages causes *page fault*
  - Page fault handling & page replacement policy cause non-deterministic delays
  - Possible solution: use selective page locking to increase determinism

Programming Languages

- Dynamic memory operations (e.g., malloc and free) are unpredictable
  - Memory allocator runs non-deterministic space optimization algorithms
  - Requests could fail due to fragmentation even if there is free memory

```
p1 = malloc(3)
p2 = malloc(4)
free(p1)
p1 = malloc(4)
```

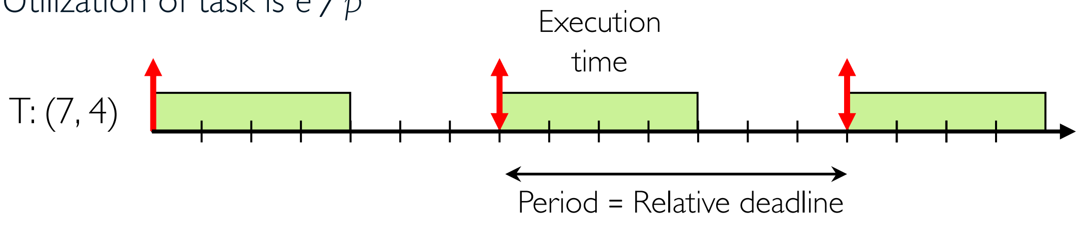


Defragmentation is not feasible, all copies of `p2` become broken if OS moves allocated

  - Possible solution: partition memory into fixed-size blocks (*partition pools*)
  - Another solution: prevent dynamic data structures all together
    - Flexibility is reduced in dynamic environment
- Recursion could lead to unbounded execution time
  - Possible solution: only allow time-bound loops
- Example of RT programming languages
  - Real-Time Java, Concurrent C, Euclid

Scheduling

- General-purpose schedulers are system-oriented
  - Minimize avg. system throughput
- RTSes need task-centric scheduling
  - Minimize worst-case response time for each task
  - Predictability ≠ fast computing
- Real-time tasks
  - Periodic: set of jobs arriving at fixed period (p)
    - Each job has worst-case execution time (e) and hard relative deadline ($d \leq p$, usually $d = p$)
    - Utilization of task is e / p



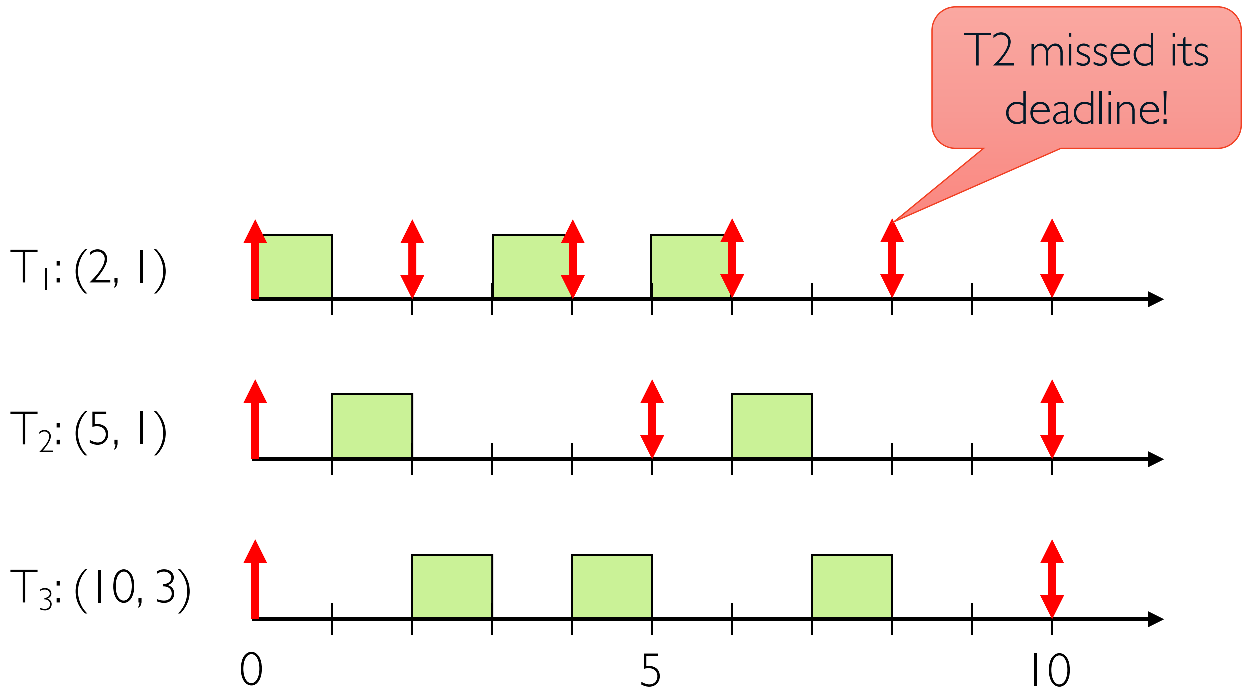
  - Aperiodic: arrive randomly without any hard deadline
  - Sporadic task arrives randomly with hard deadline
  - Independent vs. interdependent and preemptable vs. non-preemptable

Terminology of Real-time Scheduling

- Static scheduling: priority of each task does not change over time
  - E.g., rate monotonic (RM)
- Dynamic scheduling: priority of each job does not change over time
  - E.g., earliest deadline first (EDF)
- Fully-dynamic scheduling: priority of each job could change over time
  - E.g., least laxity first (LLF)
- Schedule S is *feasible* if all deadline are met
- Task set T is *schedulable* under scheduling class C if there exists scheduling algorithm A in C that produces feasible schedule for T
- Scheduling algorithm A is *optimal* w.r.t. scheduling class C if it produces feasible schedule for any schedulable task set T under C

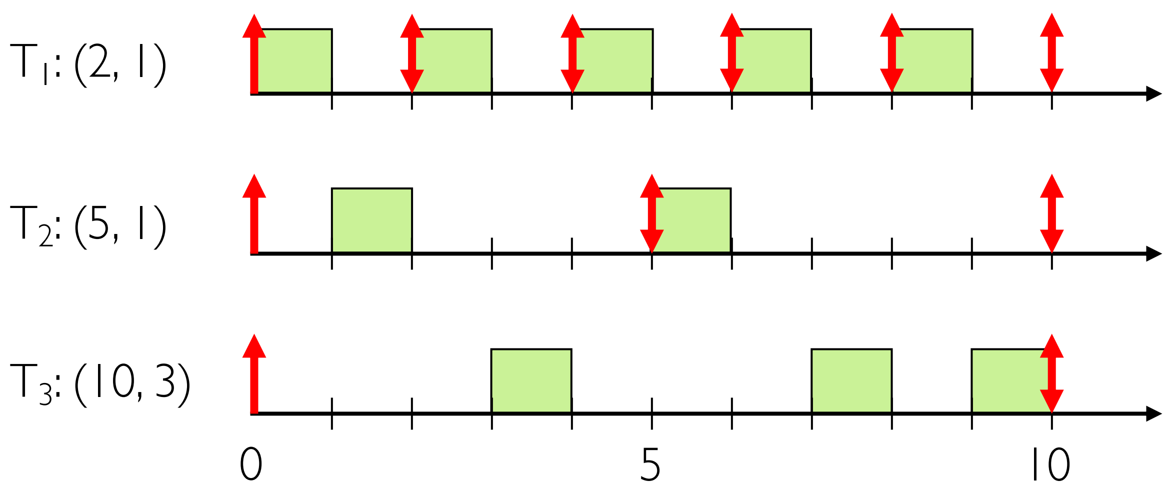
General-purpose Schedulers for RTSes

- RR example:

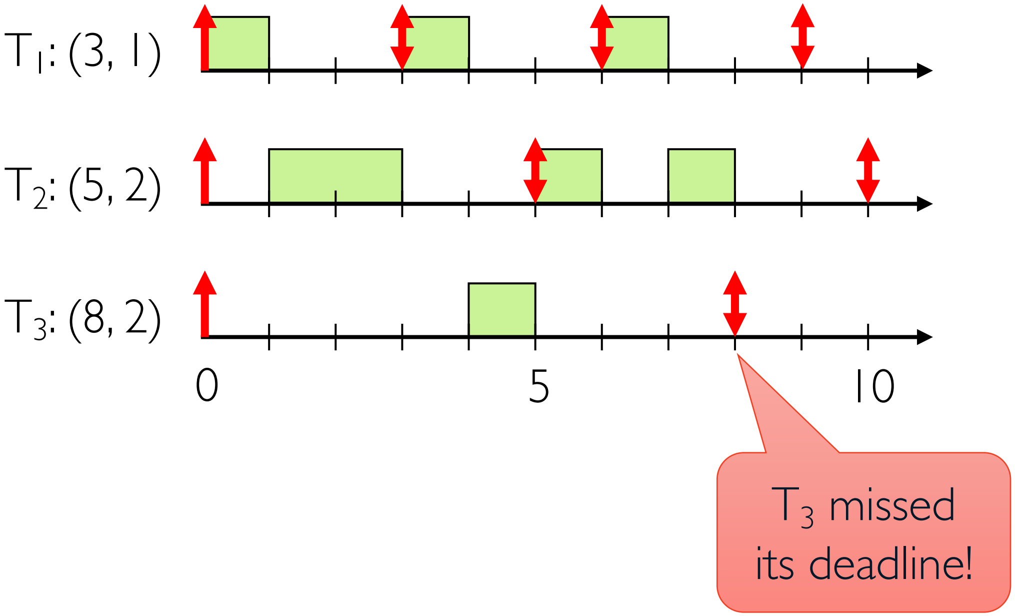


Rate Monotonic (RM)

- RM makes following assumptions
  - Tasks are periodic and independent with known and fixed execution times
- RM is static online scheduling policy
  - Higher priorities are assigned to tasks with shorter periods
  - Priority of each task is fixed and doesn't change at run-time
  - RM is optimal w.r.t. static schedulers



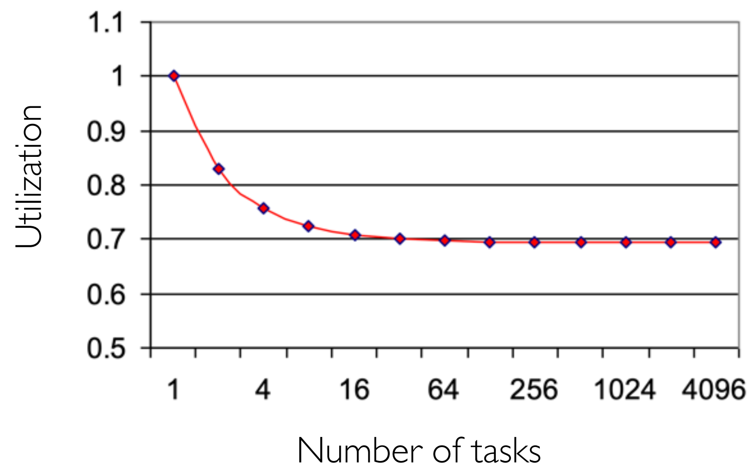
RM: Schedulablity



RM: Schedulablity Test [Liu & Layland 1973]

- For n periodic tasks with execution time e_i and deadline and period p_i, RM is guaranteed to produce feasible schedule if

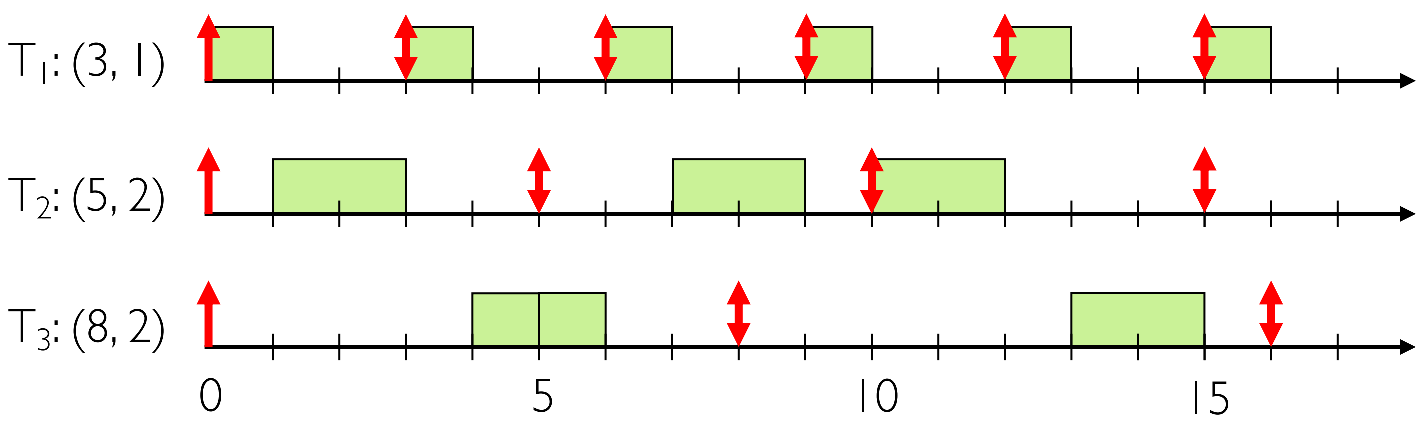
$$\sum_{i=1}^{n}\left(\frac{e_i}{p_i}\right) \leq n(2^{1/n} - 1)$$



- If condition does not hold, then deadlines may or may not be met!
- Example: $T_1(3,1)$, $T_2(5,2)$, $T_3(8,2)$
  - $1/3 + 2/5 + 2/8$ (= 98.33) ≥ $3(2^{1/3}-1)$ (≈ 0.78) ⇒ No guarantee!

Earliest Deadline First (EDF)

- EDF is dynamic online scheduling policy
  - Scheduler always schedules active task with earliest deadline
  - Current priority of tasks depends on how close their deadline is
  - Tasks' priorities change during execution
  - EDF is optimal w.r.t. all online schedulers



EDF: Schedulablity Test [Liu & Layland 1973]

- Even EDF won't work if you have too many tasks
- For *n* periodic tasks with execution time e_i and deadline and period p_i, EDF is guaranteed to produce feasible schedule if

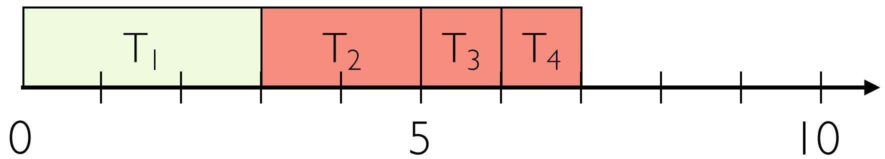
$$\sum_{i=1}^{n} \left(\frac{e_i}{p_i} \right) \leq 1$$

- System is overloaded if

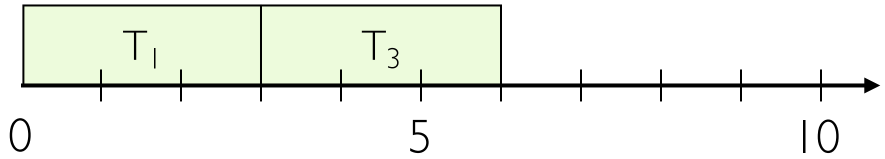
$$\sum_{i=1}^{n} \left(\frac{e_i}{p_i} \right) > 1$$

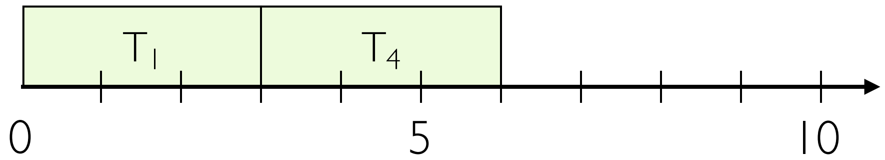
Overloaded System under EDF

- EDF schedule could be suboptimal for overloaded system
- Domino effect example: $T_1(4,3), T_2(5,3), T_3(6,3), T_4(7,3)$



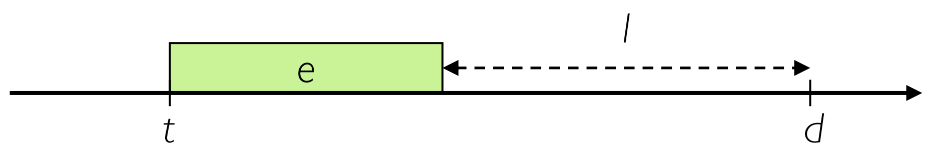
- Better schedules



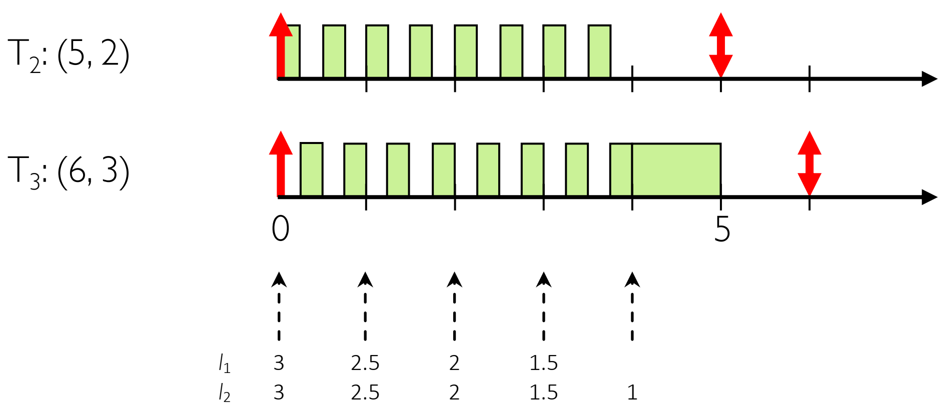


Least Laxity First (LLF)

- LLF dynamically assigns priority to jobs based on their laxity (slack)
  - With absolute deadline d and remaining execution time e, laxity at time t is $l = d - t - e$



  - Job with the smallest laxity has the highest priority
  - LLF is also optimal w.r.t. all online schedulers
  - LLF is impractical to implement because laxity tie results in frequent context switches

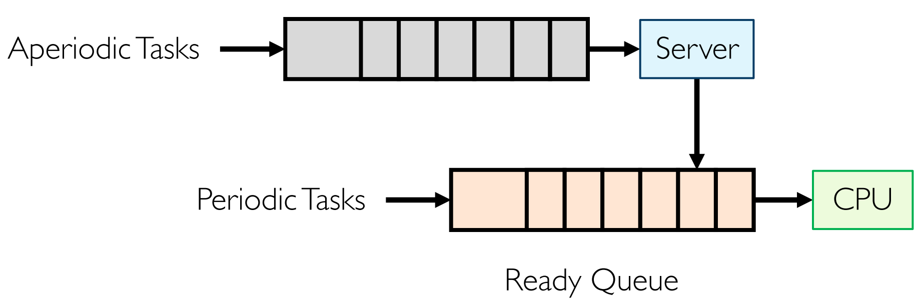


RM vs. EDF vs. LLF

- Rate monotonic (RM)
  - Simpler implementation, even in systems without explicit support for timing constraints (periods, deadlines)
  - Predictability for highest priority tasks
- Earliest deadline first (EDF)
  - Full processor utilization
  - Misbehavior during overload conditions
- Least laxity first (LLF)
  - Full processor utilization
  - Misbehavior when there are jobs with equal laxity

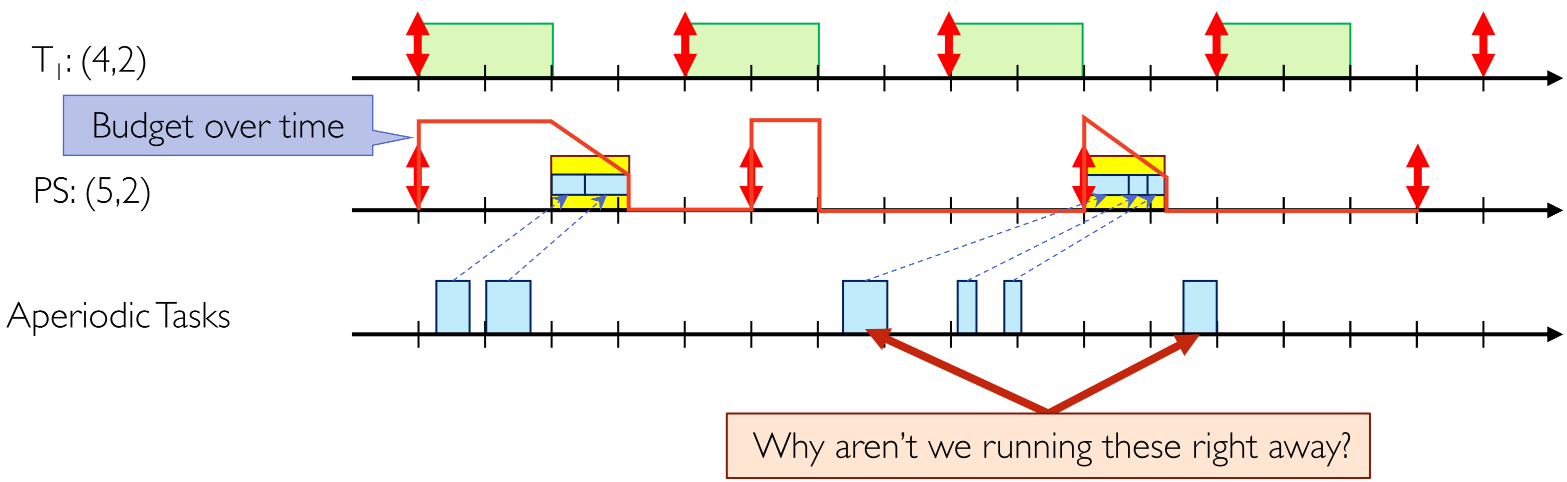
Scheduling Mixed Periodic and Aperiodic Tasks

- One idea: run aperiodic tasks as soon as they arrive
  - Response time for aperiodic tasks is minimized, but it's unbounded for periodic tasks
- Another idea: assign aperiodic tasks lowest priority (run if no periodic task runs)
  - Simple, bad response time for aperiodic tasks (applicable if they have no strict timing requirement)
- Better idea: aperiodic tasks can be served by periodically invoked *server*
  - Server can be accounted for in periodic task schedulability analysis
  - Server has period p_s and budget B_s
  - Server can serve aperiodic tasks until budget expires
  - Servers have different flavors depending on details of when they are invoked, what priority they have, and how budgets are replenished



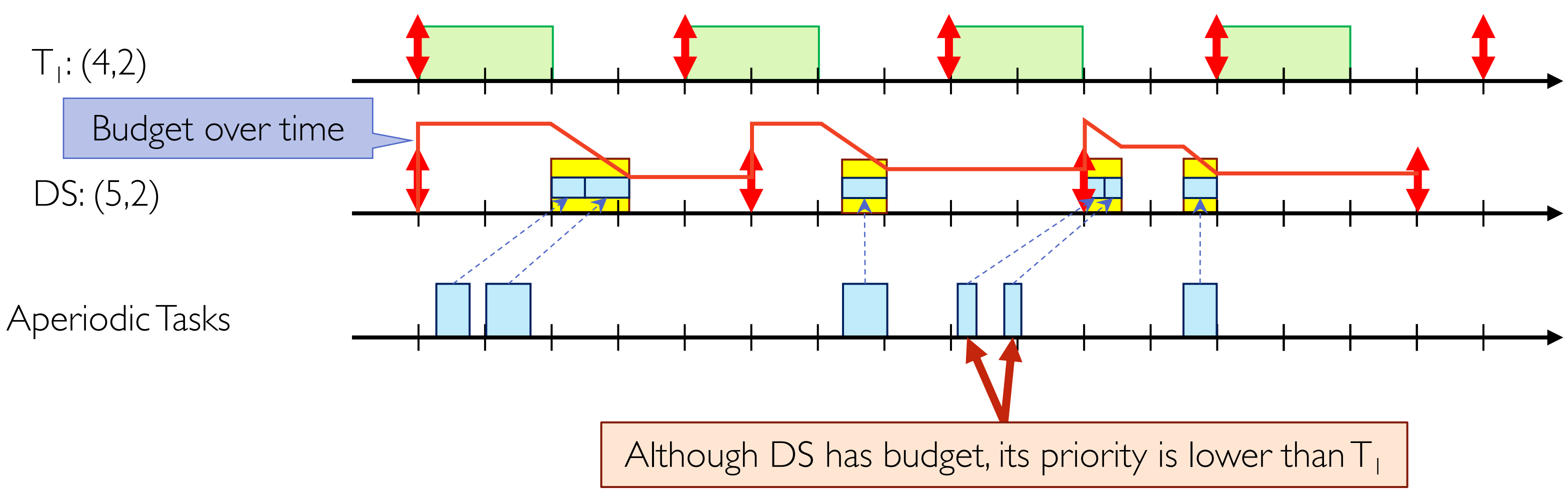
Polling Server (PS)

- Periodic tasks and PS are scheduled based on RM
- Aperiodic arrivals are queued until PS is invoked
- At the beginning of its period, PS serves queued aperiodic tasks
- PS suspends itself when queue becomes empty or budget expires
- PS is treated as regular periodic task in schedulability analysis



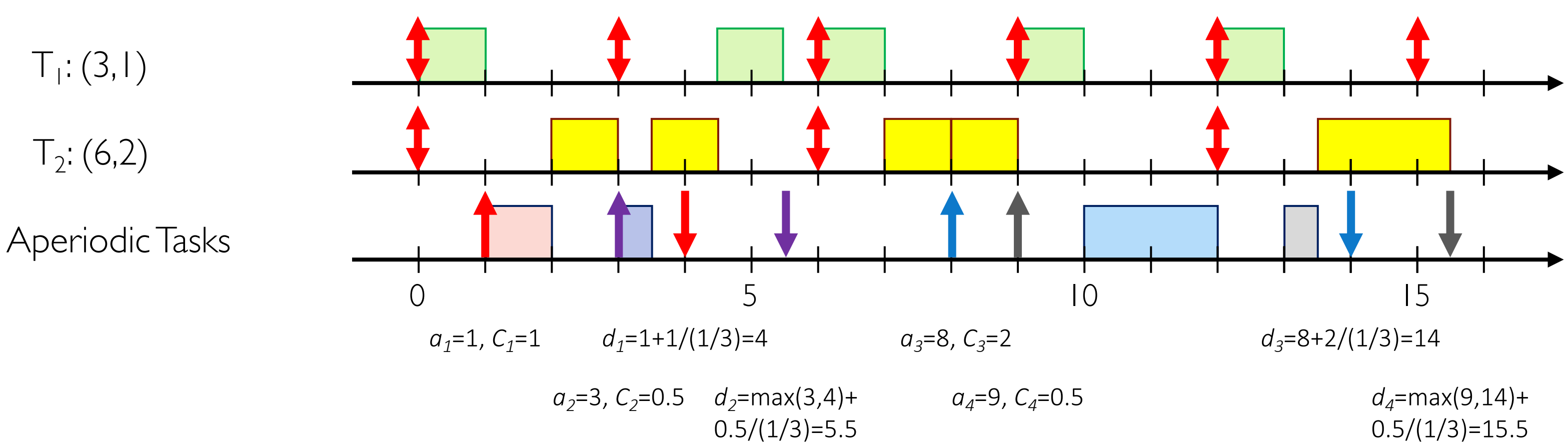
Deferrable Server (DC)

- Basic approach is like polling server
- DS preserves its budget when queue becomes empty
  - But no cumulation: at the beginning of period, budget is reset to its full value
- DS is demand driven
  - Periodic tasks are ready to run at the beginning of their periods
  - DS can run during its period only in response to aperiodic-task arrivals



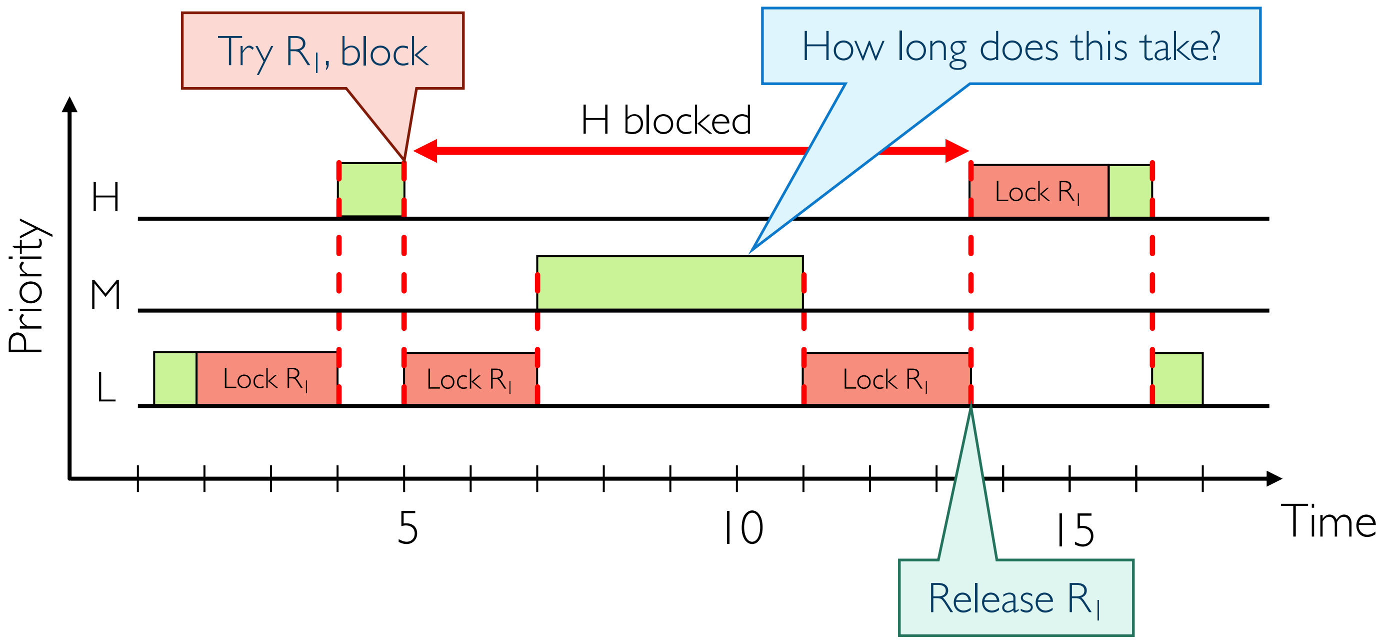
Total Bandwidth Server (TBS)

- Periodic tasks and TBS are scheduled based on EDF
  - Aperiodic and periodic tasks are both inserted in the same ready queue
  - Aperiodic tasks are artificially assigned deadline such that TBS's utilization does not exceed its given bandwidth U_{TBS}
  - Aperiodic task T_i with computation time C_i arriving at time a_i is assigned deadline $d_i = \max(d_{i-1}, a_i) + C_i / U_{TBS}$
  - Example: $T_1(3,1)$ and $T_2(6,2)$,
    - T_1 and T_2 are schedulable if $U_{TBS} \leq 1 - 2/3 = 1/3$



Scheduling Interdependent Tasks: Synchronization

- Problem of deciding whether it is possible to schedule set of periodic tasks, that use semaphores to enforce mutual exclusion is NP-hard [1]
- General-purpose synchronization primitives allow *priority inversion*



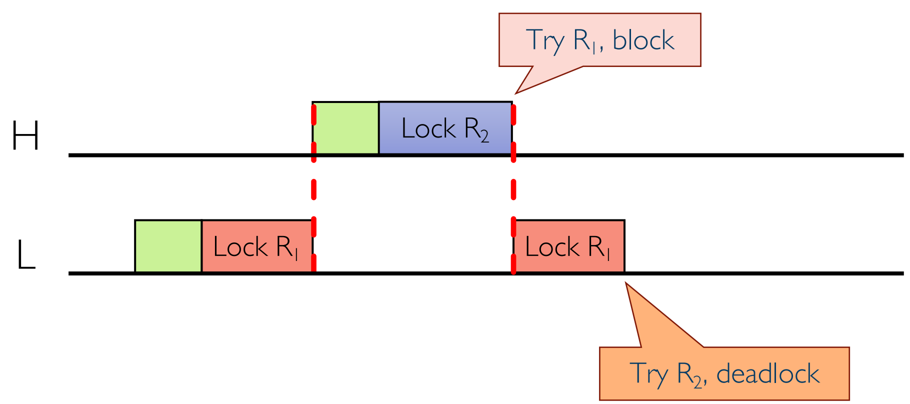
[1] A.K. Mok, "Fundamental Design Problems of Distributed Systems for Hard Real Time Environments", PhD Thesis, Laboratory for Computer Science (MIT), MIT/LCS/TR-297. (1983).

Priority Inversion and MARS Pathfinder

- Landed on Martian surface on July 4th, 1997
- After it started gathering data, it began experiencing total system resets, each resulting in losses of data
  - Pathfinder had single shared information bus used by low and high-priority tasks
  - Low-priority task ran infrequently and used bus to publish its data, while holding mutex on bus
  - Every system reset started by low-priority task getting interrupted while holding mutex
  - Interrupt handler scheduled medium-priority task
  - High-priority task was blocked waiting for low-priority task which was waiting for medium-priority task to finish
  - After some time, watchdog timer went off, noticing that bus has not been executed for some time, it concluded that something had gone bad, and initiated total system reset

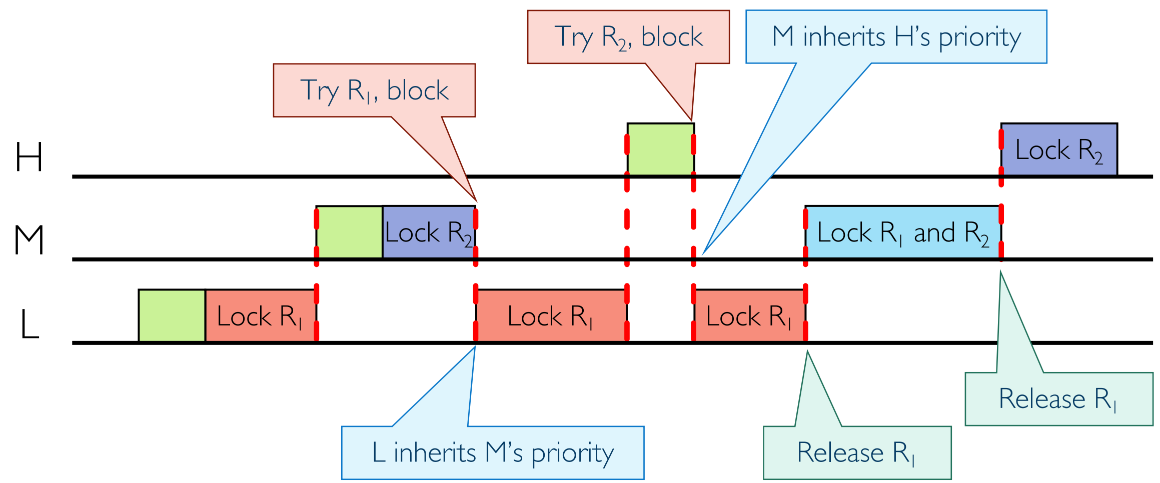
Priority Inheritance Protocol (PIP)

- PIP increases priority of task to maximum priority of any task waiting for any resource locked by the task
  - If lower-priority task L has locked any resources required by higher-priority task H, then priority of L is increased to priority of H
  - Once task unlocks resources, it runs with its original priority
- RIP does not prevent *deadlock*



PIP and Chained Blocking

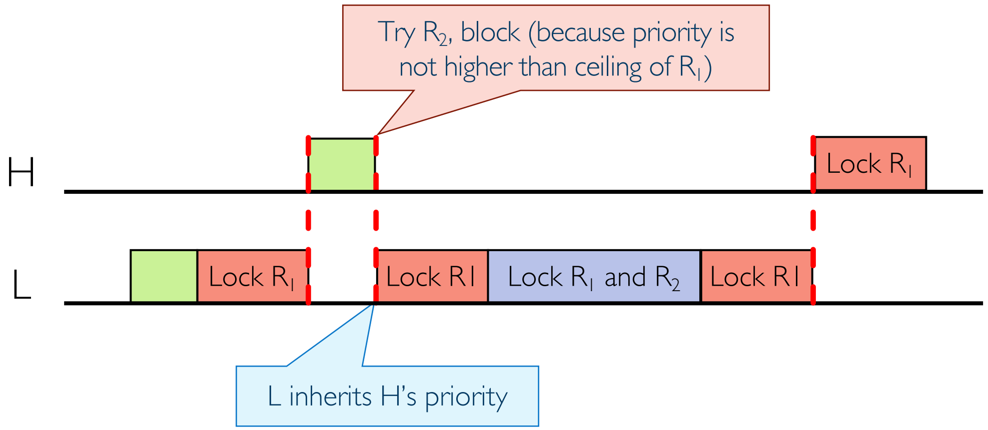
- PIP does not prevent *chained blocking*



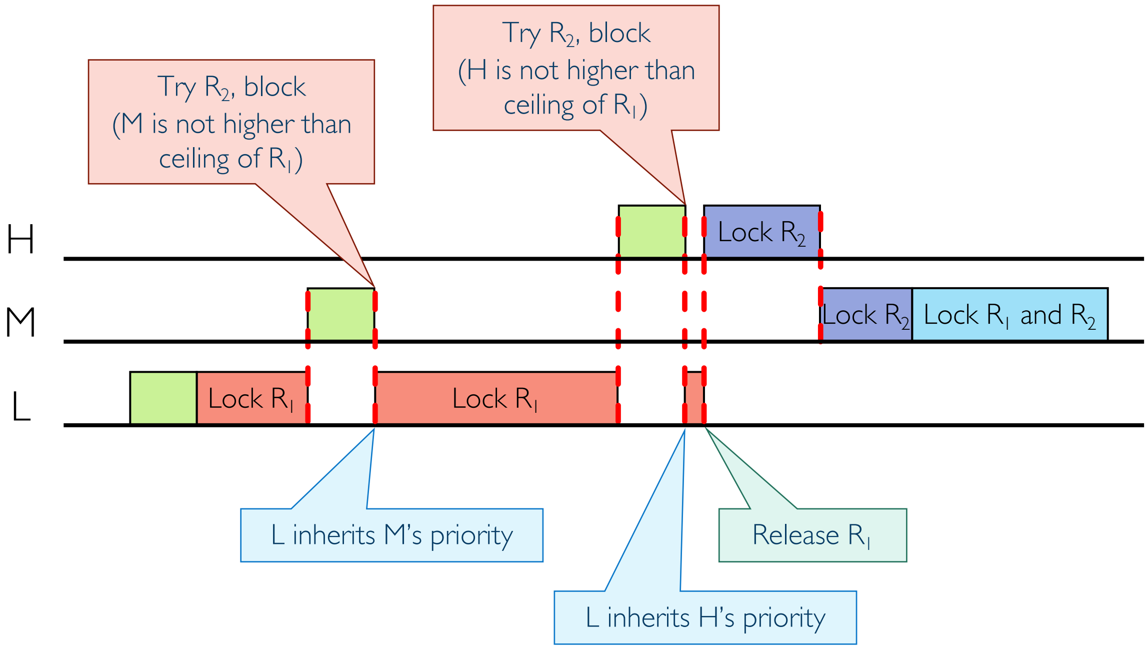
- H must wait for L and M

Priority Ceiling Protocol (PCP)

- Each resource is assigned priority ceiling
  - Equal to the highest priority of any task that can lock it
- Each task can lock resources only if its priority is higher than priority ceilings of all resources currently locked by other tasks
- Each task runs at its assigned priority unless it has locked any resource needed by higher priority task
- After task unlocks resources, it runs with its original priority
- PCP prevents deadlocks



PIP Prevents Chained Blocking



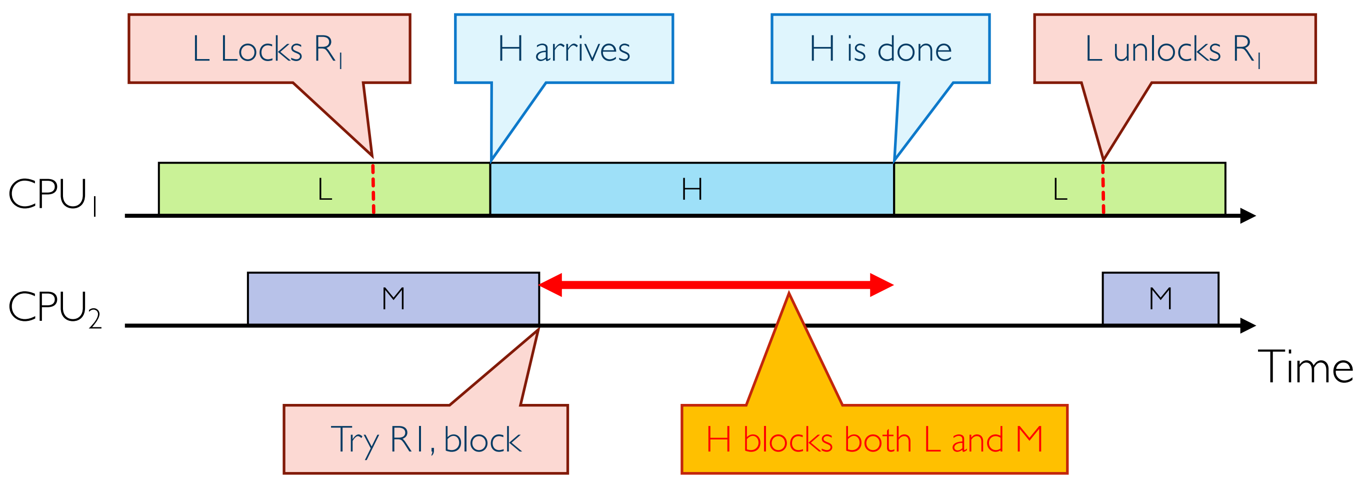
- H only waits for L

EDF and Deadline Interchange

- Deadline interchange is analogous to priority inversion
- Task which has locked resources could be preempted by another task with earlier deadline that needs those resources
- To avoid this, scheduler should assign to running task earliest deadline from among other tasks waiting for it

Multiprocessors and Remote Blocking

- In uniprocessors, it is acceptable if high-priority task pre-empts lower-priority ones
- In multiprocessors, this is not necessarily desirable
  - Example: high-priority task H and low-priority task L are assigned to CPU_1
  - Medium-priority task M runs on CPU_2



  - H is more important than either M or L, but is it more important than M and L?

Multiprocessor Scheduling

- No migration (partitioned): each task and its jobs must run on single CPU
- Restricted migration: each job must run on single CPU
  - Different jobs of the same task may run on different CPUs
- Full migration: each job can migrate between CPUs

	No Migration	Restricted Migration	Full Migration
Static	(S,N)	(S,R)	(S,F)
Dynamic	(D,N)	(D,R)	(D,F)
Fully Dynamic	(F,N)	(F,R)	(F,F)

(.,N)-based Schedulers

- Finding optimal assignment of N periodic tasks to M CPUs is equivalent to *bin-packing*
    - It's NP-hard problem

- Several polynomial-time heuristics have been proposed
    - First fit: assign each task to CPU that can accept it
      (based on feasibility test according to that CPU's uniprocessor scheduler)
    - Best fit: assign each task to CPU that can accept it and will have minimal remaining spare capacity

- Worst-case utilization is (M+1)/2
    - E.g., M+1 tasks with $e = 1 + \varepsilon$ and $p = 2$ cannot be scheduled by any (.,N) scheduler
    - Almost half of resources could be left underutilized

Some Other Scheduling Classes

- (D,R)-based: jobs have fixed priority and must run on single CPU
  - Suitable for task sets in which each job has considerable amount of state (it is not desirable to migrate jobs between processors)
- (D,F)-based: jobs have fixed priority but can migrate
  - Preemption, and hence migration, can only happen because of new job arrival
  - E.g., *global EDF*: use single ready queue for all CPUs, set priorities according to EDF
    - No longer optimal: $T_1(10,5), T_2(10,5), T_3(11,7)$ on 2 CPUs
    - EDF runs T_1 and T_2 first in parallel $\Rightarrow T_3$ misses its deadline
- (S,F)-based: tasks have fixed priority, jobs can migrate
  - E.g., global RM: us single ready queue for all CPUs, set priorities according to RM
- Worst-case utilization of any (x,y)-based scheduler is (M+1)/2, unless x = y = F [1]

[1] Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J. H., & Baruah, S. K. (2004). A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms.

(F,F)-based Schedulers

- Global LLF: use single ready queue for all CPUs, set priorities based on LLF
  - It schedules any instance that global EDF can schedule
  - Like global EDF, it is not optimal

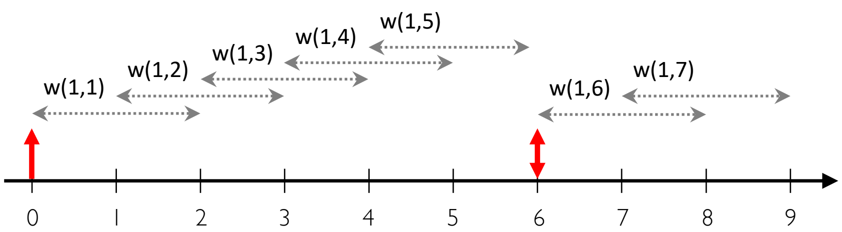
- P-fair scheduling: allocate CPU time to enforce *proportionate progress*
  - Is optimal (both for uniprocessors and multiprocessors)
  - Produces feasible schedule for M CPUs and any task set T with $U_T \leq M$

P-fairness

- Main idea: allocate CPU time to each task i in proportion to its weight $w_i = e_i / p_i$
- Divide time into small time quanta
  - All parameters are integer multiples of time quantum (e.g., $e_i, p_i \in \mathbb{Z}^+$)
- Define lag for each task to captures discrepancy between what it should have received and what it actually received
  - $lag(i,t) = t \times w_i - allocated(i,t)$
- Schedule S is periodic if and only if for all task i and any integer k
  - $allocated(i,k \times p_i) = k \times e_i$
- Schedule S is P-fair if and only if for all task i and time t
  - $-1 < lag(i,t) < 1$
- Any P-fair schedule is periodic
  - At $t = k \times p_i$, $allocated(i,t)$ and $t \times w_i$ are both integers $\Rightarrow$ $allocated(i,t) = k \times e_i$
  - Periodic schedules aren't necessarily P-fair (why?)

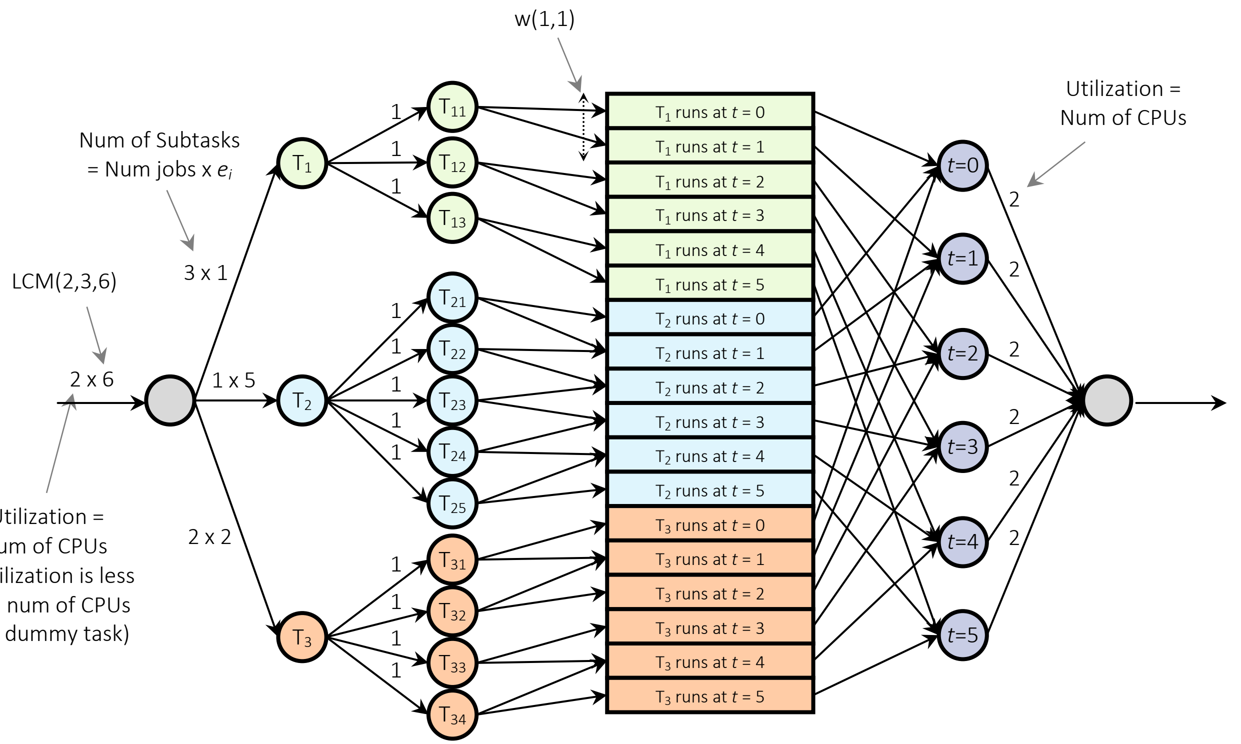
Subtasks and Pseudo Parameters

- Divide task *i* into quantum-sized subtasks
  - T_{ij} denotes the j^{th} subtask of task *I*
- Pseudo-release: Let r(*i*,*j*) denote the earliest time T_{ij} could be scheduled
  - $r(i,j) = \min t\ (\geq 0): (t+1) \times w_i - j > -1 = \left\lfloor \frac{j-1}{w_i} \right\rfloor$
- Pseudo-deadline: Let d(*i*,*j*) denote the latest time by which T_{ij} must have been scheduled
  - $d(i,j) = \max t\ (\geq 0): (t-1) \times w_i - (j-1) < 1 = \left\lceil \frac{j}{w_i} \right\rceil$
- Window: Let w(*i*,*j*) = [r(*i*,*j*),d(*i*,*j*)] denote the interval during which T_{ij} must be scheduled
  - Window overlaps, denoted by b(*i*,*j*) = d(*i*,*j*) – r(*i*,*j*+1), are either 0 or 1
  - Example: T_1(6,5)
    - r(1,1) = 0, d(1,1) = 2
    - r(1,2) = 1, d(1,2) = 3
    - r(1,3) = 2, d(1,3) = 4
    - …



Existence of P-fair Schedule

- Example: scheduling $T_1(2,1)$, $T_2(6,5)$, and $T_3(3,2)$ on two CPUs



- *Integral flow theorem*: If all edges have integral capacity, then integral maximal flow exists
- Network flow problem has integer solution ⇒ P-fair schedule exists

P-fair (PF) Scheduling Algorithm

- PF priorities subtasks on earliest-pseudo-deadline-first (EPDF) basis

- At time t, T_{ij} has higher priority than T_{mn} ($T_{ij} \succ T_{mn}$), if any of following holds
  I. $d(i,j) < d(m,n)$
  II. $d(i,j) = d(m,n)$ and $b(i,j) > b(m,n)$
  III. $d(i,j) = d(m,n)$, $b(i,j) = b(m,n) = 1$, and $T_{i(j+1)} \succ T_{m(n+1)}$

- If neither subtask has priority over other, then tie can be broken arbitrarily

- Intuition behind (II): scheduling T_{ij} earlier prevents it from shortening $w(i,j+1)$
  - Makes it easier to schedule $T_{i(j+1)}$ by its pseudo-deadline
  - Similar intuition behind (III)

PF Discussion

- PF incurs very high overheads by making scheduling decisions at every time quantum

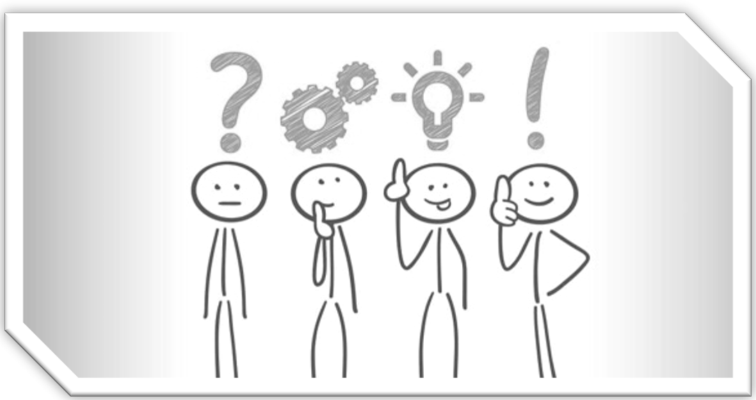
- Also, all processors need to synchronize on boundary between quanta when scheduling decisions are made

- Extensions to PF try to mitigate some of these problems
  - E.g., PD, PD^2, ERfair

Summary

- Real-time systems have strict timing constraints
- General-purpose operating systems are inadequate for real-time systems
- Real-time operating systems should provide predictability
  - Memory management, interrupt handling, scheduling, etc.
- Scheduling in real-time systems is task-centric
  - All tasks should meet their deadlines
  - Worst-case execution time is important not average throughput
- Optimal scheduler exist for uniprocessor systems
  - RM, EDF, and LLF
- Scheduling real-time tasks on multiprocessors is challenging
  - Optimal uniprocessor scheduler are no longer optimal for multiprocessors
  - Partitioning tasks between processors is a "hard" problem
  - Optimal schedulers exist, but they typically incur high overhead

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, Canny, Lee (Insup), Drews, and Andersson (Björn)