CSC 112: Computer Operating Systems
Lecture 6

Real-Time Scheduling

Department of Computer Science,
Hofstra University

Acknowledgement: Lecture slides based on Buttazzo, Hard Real-Time Computing Systems

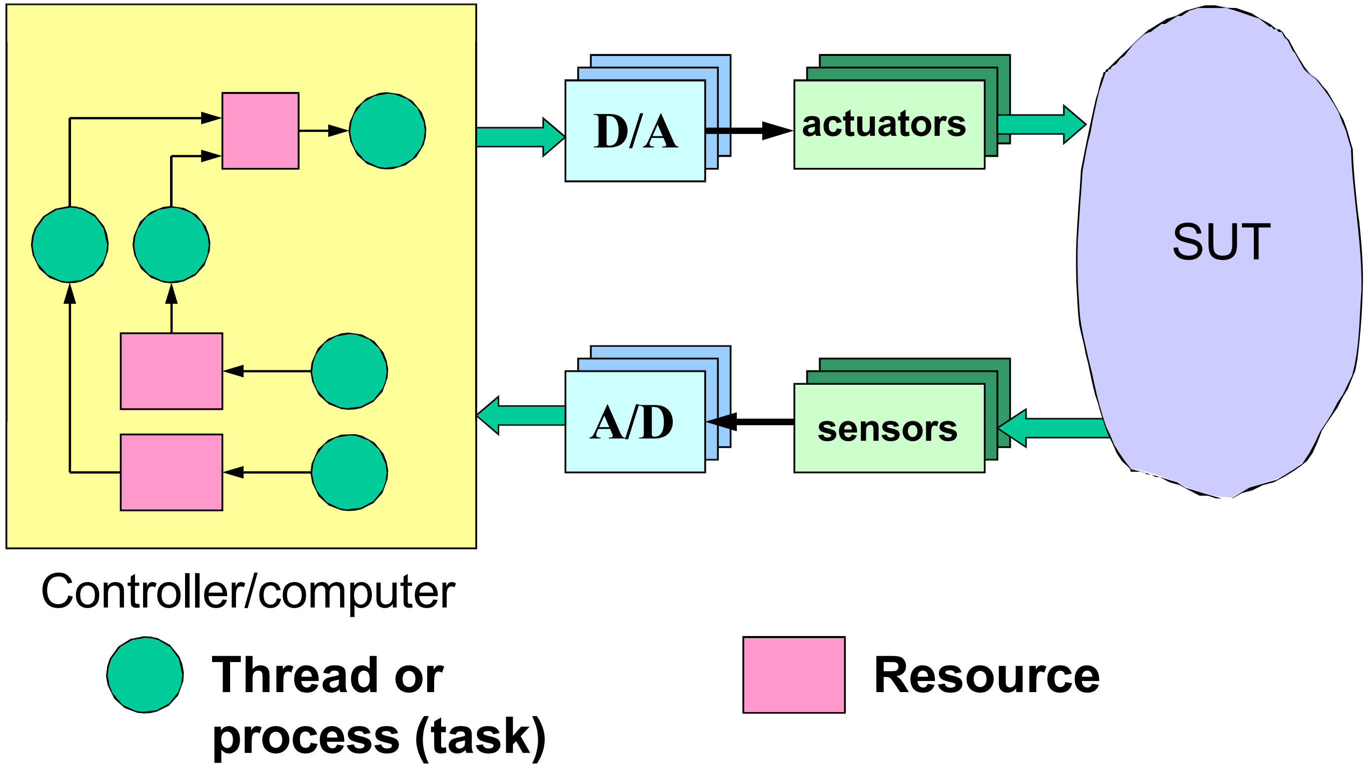
Outline

- Introduction to RTOS and Real-Time Scheduling
- Fixed-Priority Scheduling
- Earliest Deadline First Scheduling
- Least Laxity First (LLF) Scheduling
- Preemptive vs. Non-Preemptive Scheduling
- Multiprocessor Scheduling
- Resource Synchronization Protocols (for Fixed-Priority Scheduling)

Introduction to RTOS and Real-Time Scheduling

Embedded Control Systems

- An embedded control system co´nsists of:
  - The system-under-control (SUT)
    - » may include sensors and actuators
  - The controller/computer
    - » sends signals to the system according to a predetermined control objective
- In the old days, each control task runs on a dedicated CPU
  - No RTOS, bare metal
  - No need for scheduling
  - Just make sure that task execution time < deadline
- Now, multiple control tasks share one CPU
  - Multitasking RTOS
  - Need scheduling to make sure all tasks meet deadlines



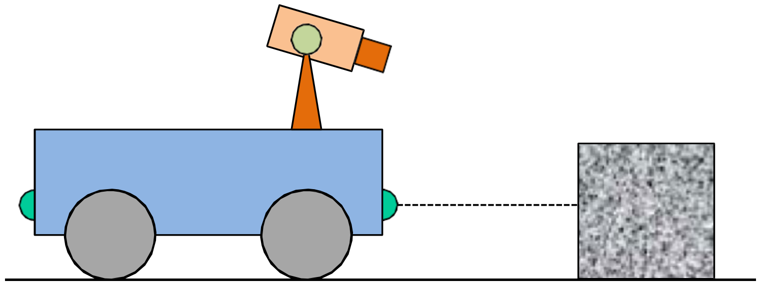
Requirements

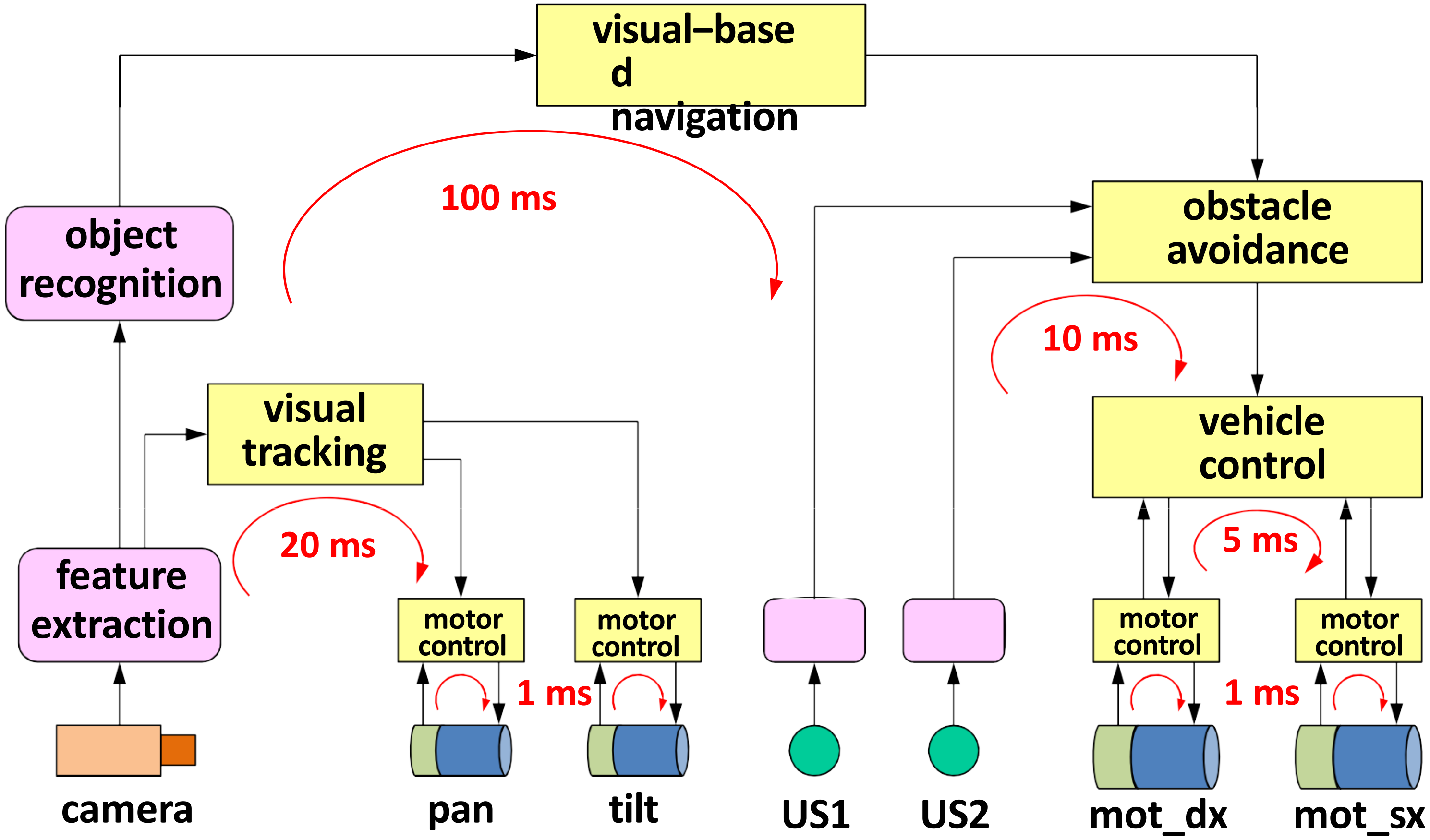
- The tight interaction with the environment requires the system to react to events within precise timing constraints
- Timing constraints are imposed by the dynamics of the environment

- The real-time operating system (RTOS) must be able to execute tasks within timing constraints

A Robot Control Example

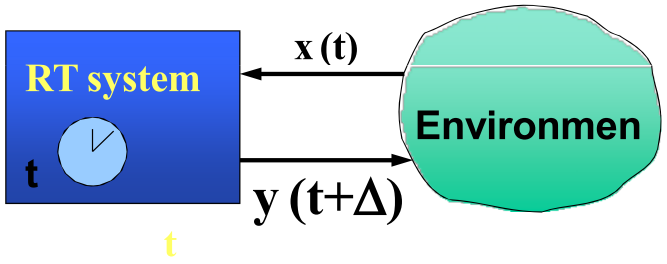
- Consider a robot equipped with:
  - two actuated wheels
  - two proximity (US) sensors
  - a mobile (pan/tilt) camera
  - a wireless transceiver
- Goal:
  - follow a path based on visual feedback
  - avoid obstacles

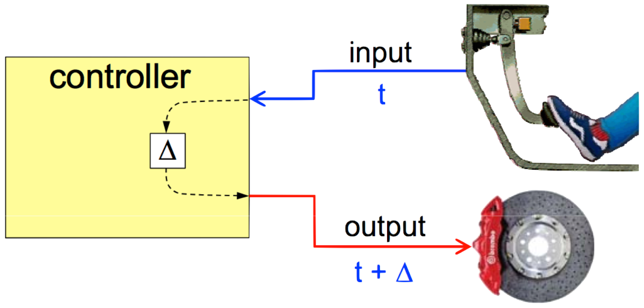


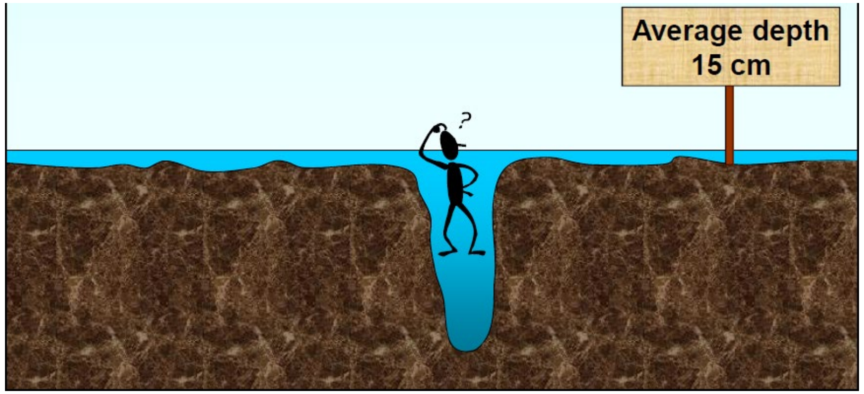


Real-Time Systems

- A computer system that is able to respond to events within precise timing constraints
- A system where the correctness depends not only on the output values, but also on the time at which results are produced
- A real-time system is not a necessarily a real fast system
  - Speed is always relative to a specific environment
  - Running faster is good, but does not guarantee hard real-time constraints
- The objective of a real-time system is to guarantee the worst-case timing behaviour of each individual task
- The objective of a fast system is to optimize the average-case performance
  - A system with fast average-case performance may not meet worst-case timing requirements
  - Analogy: there was a person who drowned in a river with average depth of 15 cm







RTOS Requirements

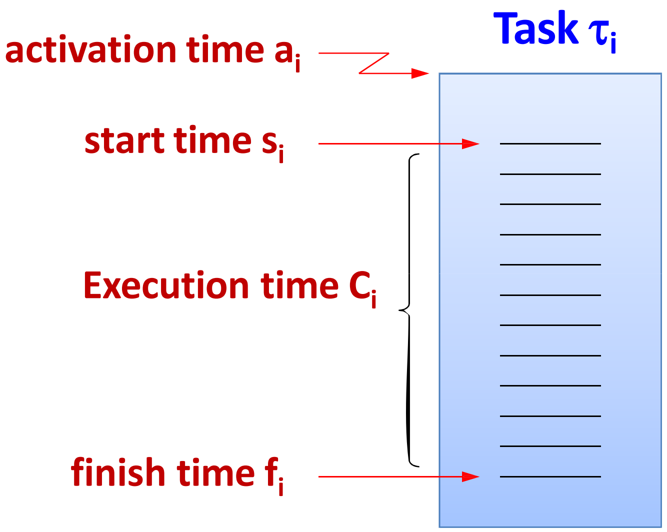
- Timeliness: results must be correct not only in their value but also in the time domain
  - provide kernel mechanism for time management and for handling tasks with explicit timing constraints and different criticality
- Predictability: system must be analyzable to predict the consequences of any scheduling decision
  - if some task cannot be guaranteed within time constraints, system must notify this in advance, to handle the exception (plan alternative actions)
- Efficiency: operating system should optimize the use of available resources (computation time, memory, energy)
- Robustness: must be resilient to peak-load conditions
- Fault tolerance: single software/hardware failures should not cause the system to crash
- Maintainability: modular architecture to ensure that modifications are easy to perform

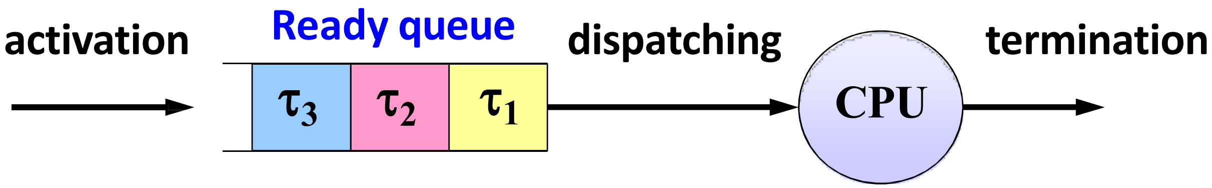
Sources of Nondeterminism

- Architecture
  - cache, pipelining, interrupts, DMA
- Operating System (our focus in this lecture)
  - scheduling, synchronization, communication
- Language
  - lack of explicit support for time
- Design Methodologies
  - lack of analysis and verification techniques

Task

- The concept of concurrent tasks reflects the intuition about the functionality of embedded systems.
  - Task here can refer to either process or thread, depending on the underlying RTOS support
- Tasks help us manage timing complexity:
  - multiple execution rates
    - » multimedia
    - » automotive
  - asynchronous input
    - » user interfaces
    - » communication systems

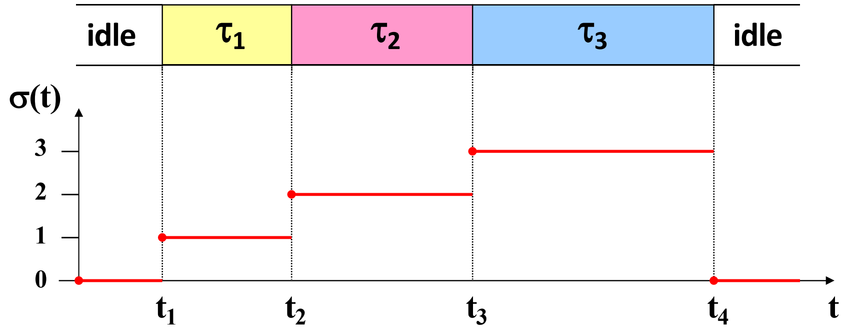




"activation" = "arrival" = "release" time

Schedule

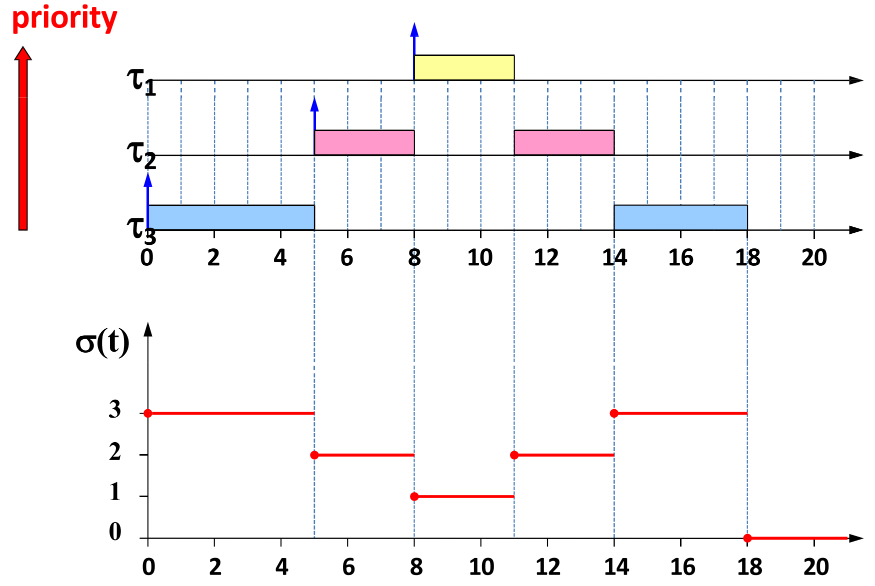
- A specific assignment of tasks to the processor that determines the task execution sequence. Formally:
- Given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a function $\sigma: R^+ \rightarrow N$ that associates an integer k to each time slice $[t_i, t_{i+1})$ with the meaning:
  - $k = 0$: in $[t_i, t_{i+1})$ the processor is idle
  - $k > 0$: in $[t_i, t_{i+1})$ the processor executes τ_k



At times t_1, t_2,…: context switch to a different task

Preemptive vs. Nonpreemptive Scheduling

- A scheduling algorithm is:
  - preemptive: if the active job can be temporarily suspended to execute a more important job
  - non-preemptive: if the active job cannot be suspended, i.e., always runs to completion



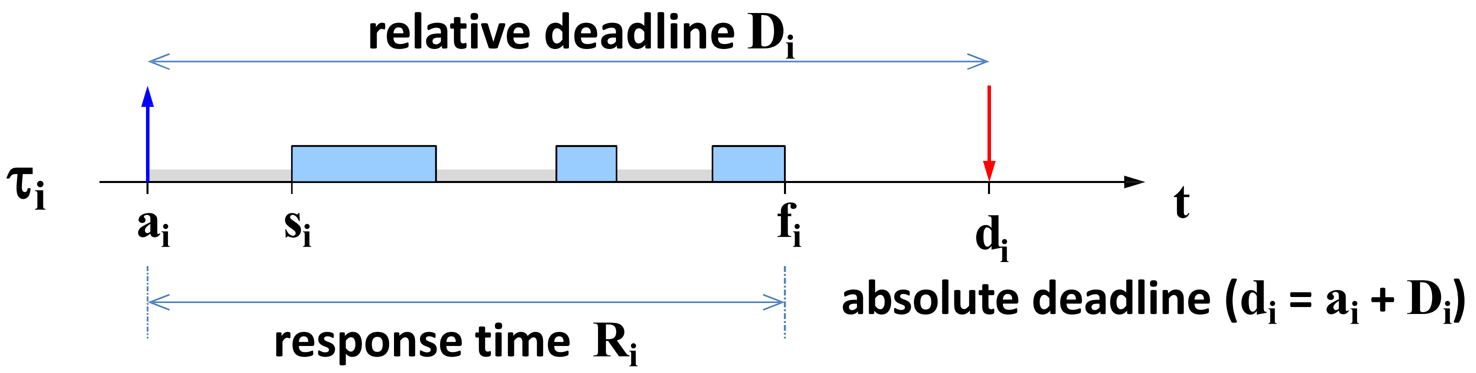
Preemptive scheduling example

Definitions

- Feasible schedule
  - A schedule σ is said to be feasible if all the tasks can complete according to a set of specified constraints.
- Schedulable set of tasks
  - A set of tasks Γ is said to be schedulable if there exists at least one algorithm that can produce a feasible schedule for it.
- Hard real-time task: missing deadline may have catastrophic consequences, so deadline violations are not permitted. A system able to handle hard real-time tasks is a hard real-time system
  - sensory acquisition
  - low-level control
  - sensory-motor planning
- Soft real-time task: missing deadlines causes Quality-of-Service(QoS)/performance degradation, so deadline violations are expected and permitted
  - reading data from the keyboard—user command interpretation
  - message displaying
  - graphical activities

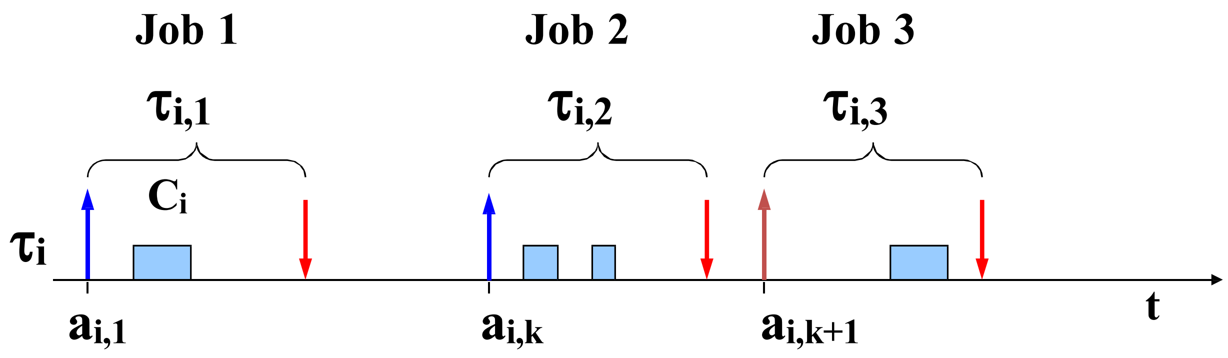
Real-Time Task

- A task characterized by a timing constraint on its response time, called deadline:
  - relative deadline D_i: part of task attribute definition, measured from task arrival time ai
  - Absolute deadline $d_i = a_i + D_i$: measured from some absolute reference time point 0
  - Gantt chart convention: upwards arrows denote job arrival/release times; downwards arrows denote deadlines
- Definition: feasible task
  - A real-time task τ_i is said to be feasible if it completes within its absolute deadline, that is, if $f_i \leq d_i$, or, equivalently, if $R_i \leq D_i$



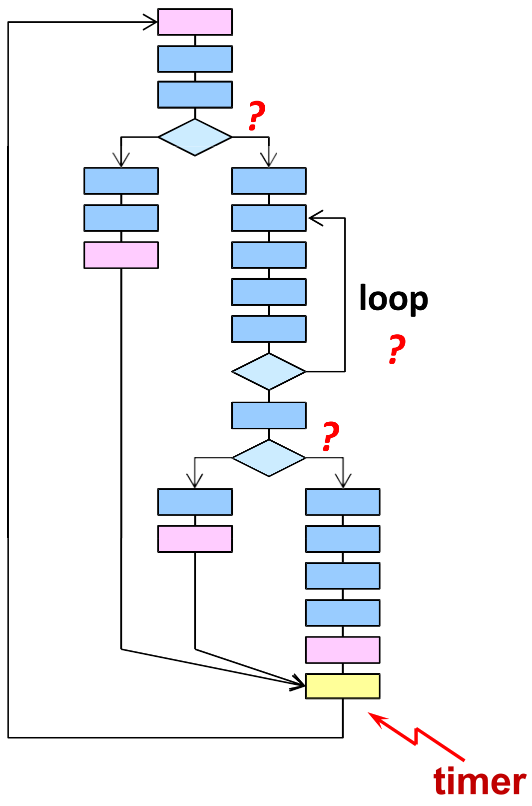
Tasks and Jobs

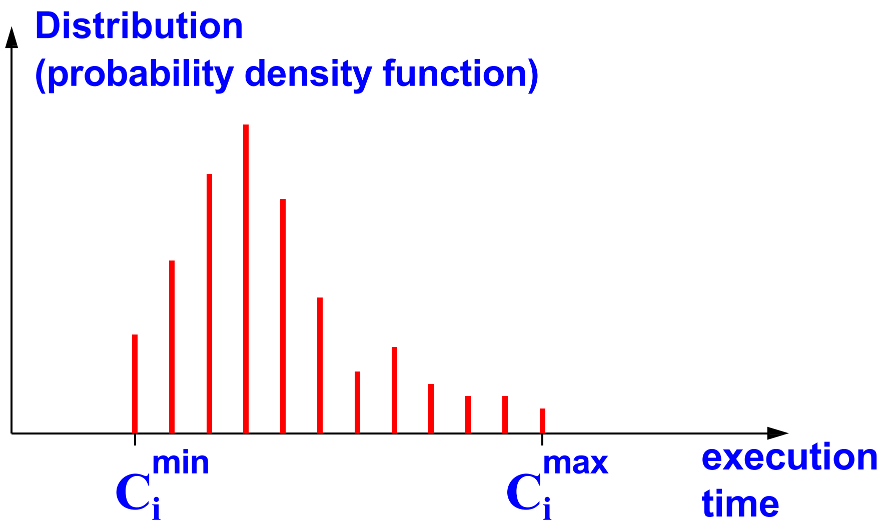
- A task running several times on different input data generates a sequence of instances (jobs)
  - Upwards arrow: task arrival or release times; downwards arrow: task deadlines
- Activation mode:
  - Periodic tasks: the task is activated by the operating system at predefined time intervals
  - Aperiodic tasks: the task is activated at an event arrival



Estimating WCET is Not Easy

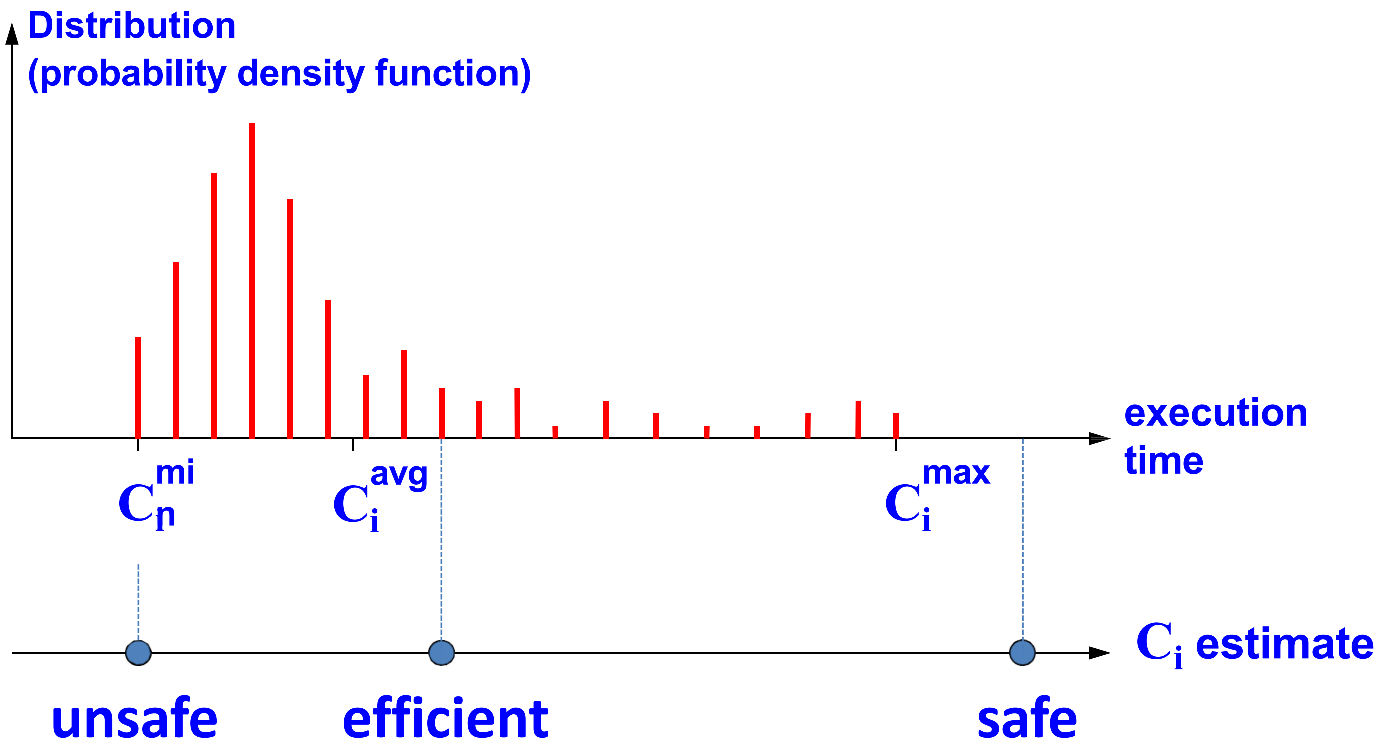
- Each job operates on different data and can take different paths.
- Even for the same data, computation time depends on processor state (cache state, number of preemptions).
- We use C_i to denote C_i^{max} Worst-Case Execution Time (WCET) in this lecture, and assume it is given as part of task parameters.





Predictability/Safety vs. Efficiency

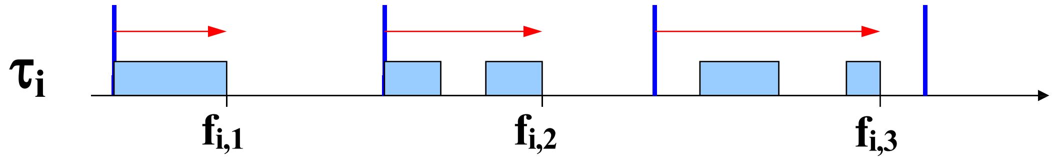
- Tradeoff between safety and efficiency in estimating the WCET C_i
  - Setting a large C_i achieves high predictability and safety, since it is unlikely to be exceeded at runtime; but it hurts efficiency, since the system needs to reserve more CPU time for the task. Suitable for hard real-time tasks.
  - Setting a small C_i achieves high efficiency, but hurts safety, since the task may execute for more than its C_i estimate. Suitable for soft real-time tasks.



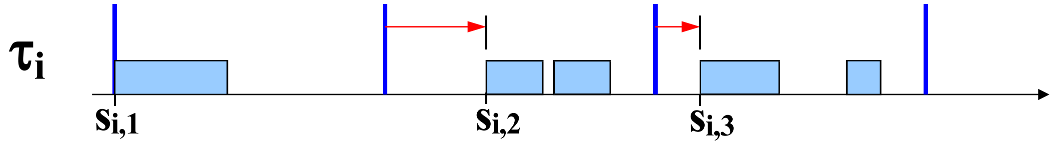
Jitter

- It is a measure of the time variation of a periodic event:

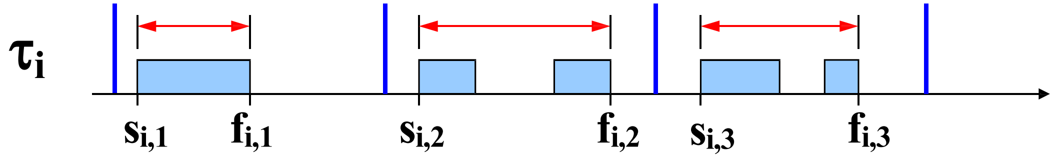
Finish-time Jitter



Start-time Jitter

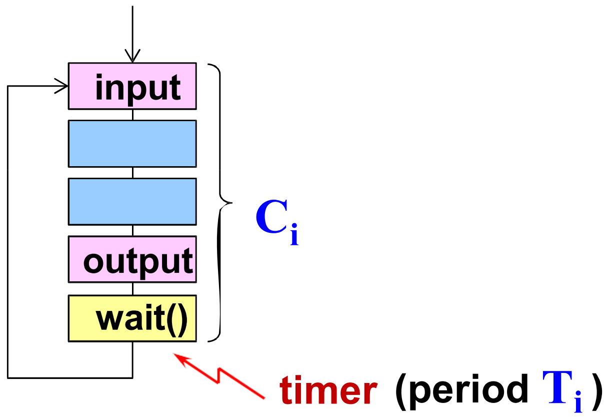


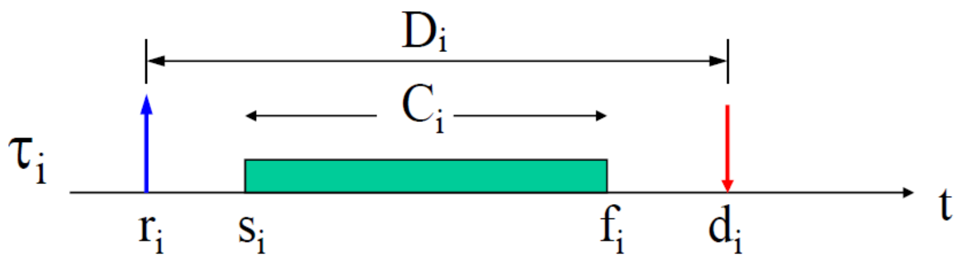
Completion-time Jitter (I/O Jitter)



Periodic Task

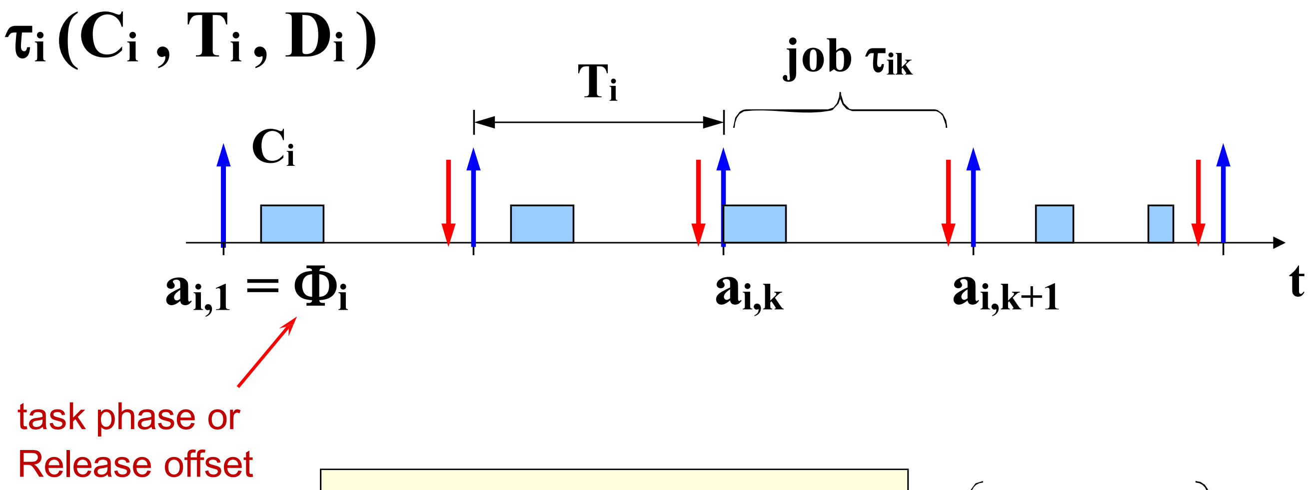
- A periodic task τ_i has a tuple of 3 attributes (C_i, T_i, D_i):
  - Worst-Case Execution Time (WCET) C_i; Period T_i; Relative Deadline D_i
  - Implicit deadline if $D_i = T_i$; Constrained deadline if $D_i \leq T_i$
- It generates an infinite sequence of jobs in every period: $\tau_{i,1}, \tau_{i,1}, \ldots, \tau_{i,k}, \ldots$





$\mathbf{r_i}$	release time (arrival time $\mathbf{a_i}$)
$\mathbf{s_i}$	start time
$\mathbf{C_i}$	worst-case execution time (wcet)
$\mathbf{d_i}$	absolute deadline
$\mathbf{D_i}$	relative deadline
$\mathbf{f_i}$	finishing time

A job of task τ_i



$$a_{i,k} = \Phi_i + (k-1)\, T_i$$
$$d_{i,k} = a_{i,k} + D_i$$

often $D_i = T_i$

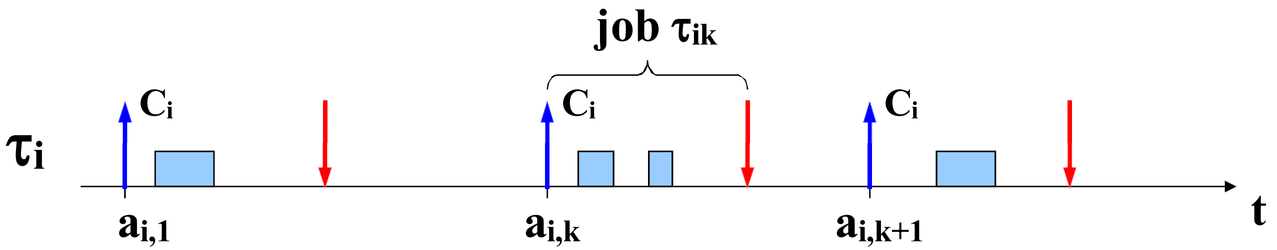
Multiple jobs released by task τ_i

Aperiodic & Sporadic Task

- Aperiodic task: jobs may arrive at arbitrary time instants
- Sporadic task: arrival times with a minimum interarrival time constraint

- **Aperiodic:** $a_{i,k+1} > a_{i,k}$
- **Sporadic:** $a_{i,k+1} \geq a_{i,k} + T_i$

minimum interarrival time

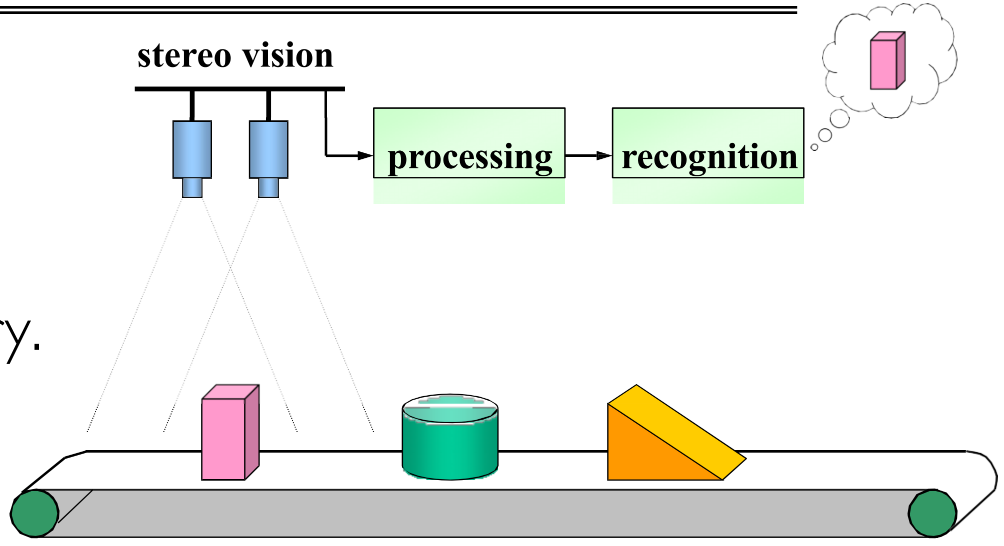


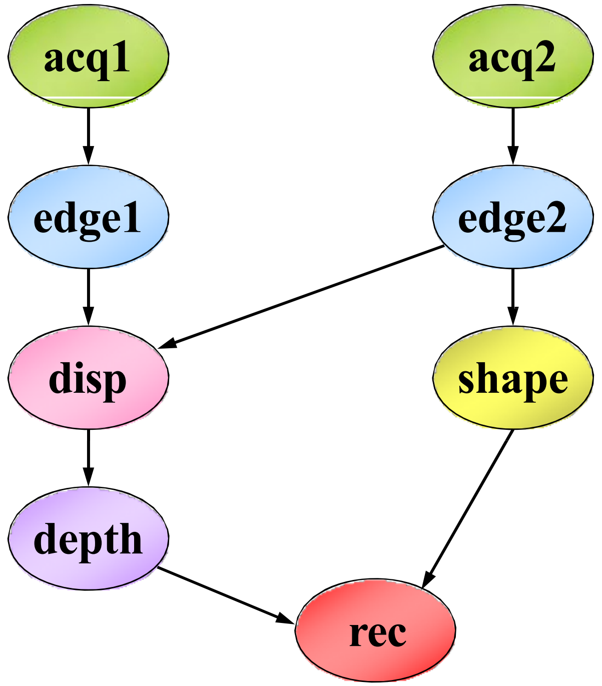
Types of Constraints

- Timing constraints
  - Deadline, jitter
- Precedence constraints
  - Relative ordering among task executions
- Resource constraints
  - Synchronization when accessing mutually-exclusive resources (shared data)

Precedence Constraints

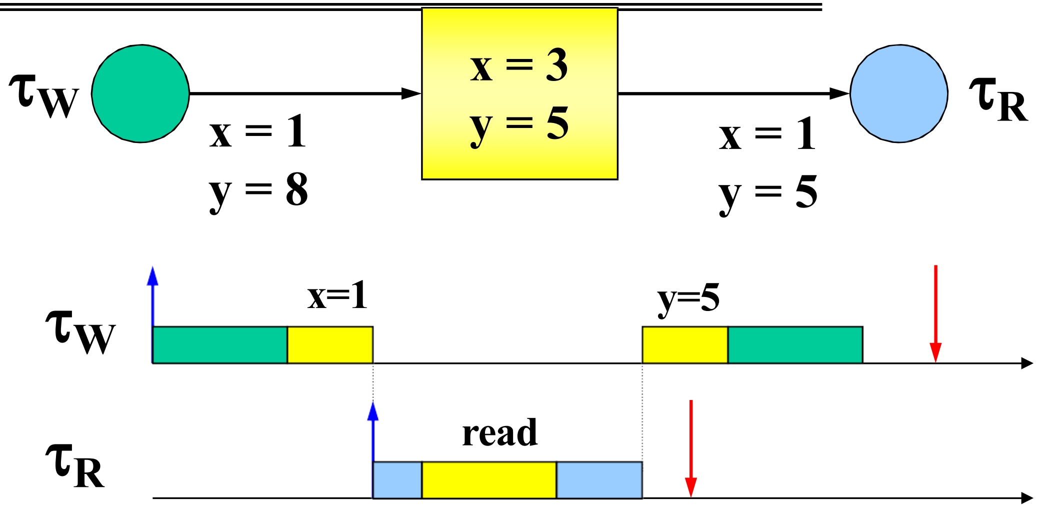
- Tasks must be executed with specific precedence relations, specified by a Directed Acyclic Graph (Precedence Graph)
- Example application of parts inspection in a factory. Tasks:
  - Image acquisition (acq1, acq2)
  - Edge detection (edge1, edge2)
  - Shape detection (shape), pixel disparities (disp)
  - Height determination (height), recognition (rec)



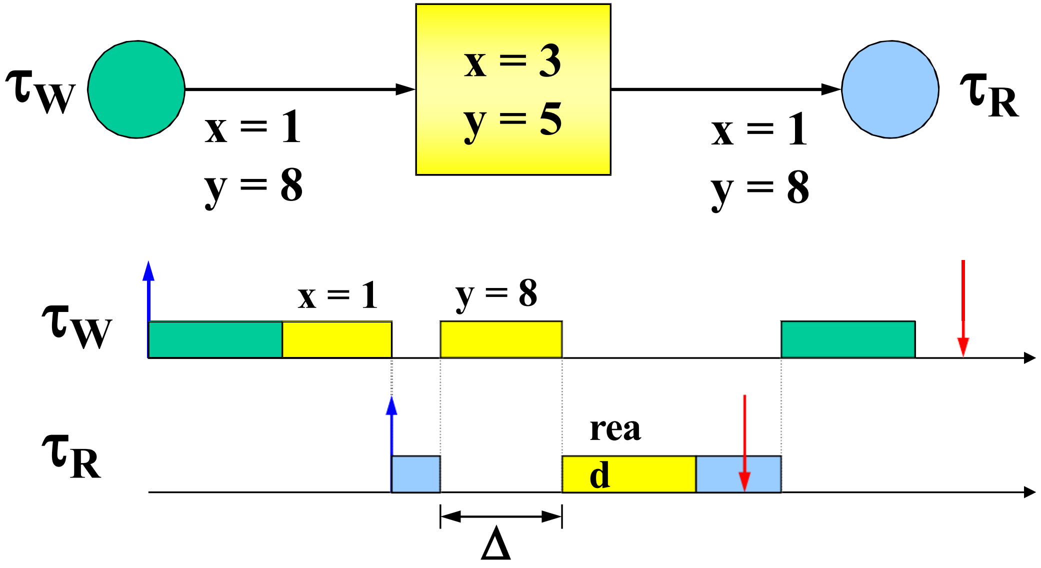


Resource Constraints

- To ensure data consistency, shared data must be accessed in mutual exclusion
- Example: the writer task τ_W writes to variables x and y; the reader task τ_R reads x and y. The pair of variables (x, y) should be updated atomically, i.e., τ_R should read either $(x, y) = (1,8)$ or $(x, y) = (3,5)$.
- Left upper: an erroneous scenario when τ_R reads a set of inconsistent values $(x, y) = (3,5)$.
- Left lower: protecting the critical section (yellow parts) with a mutex lock ensures atomicity.



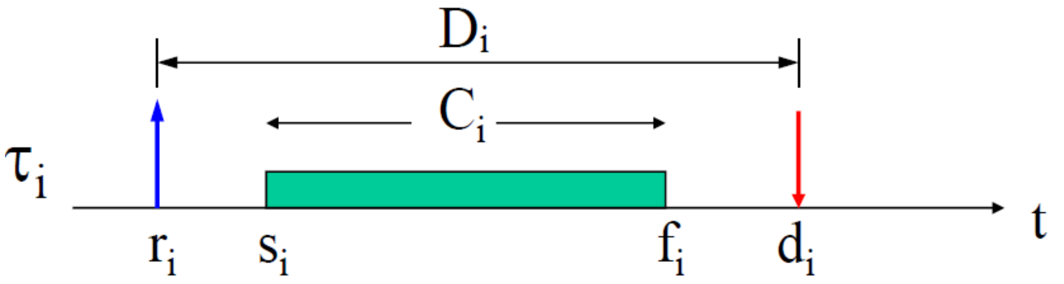
Erroneous scenario with no lock protection



Correct scenario with lock protection

Scheduling Metrics

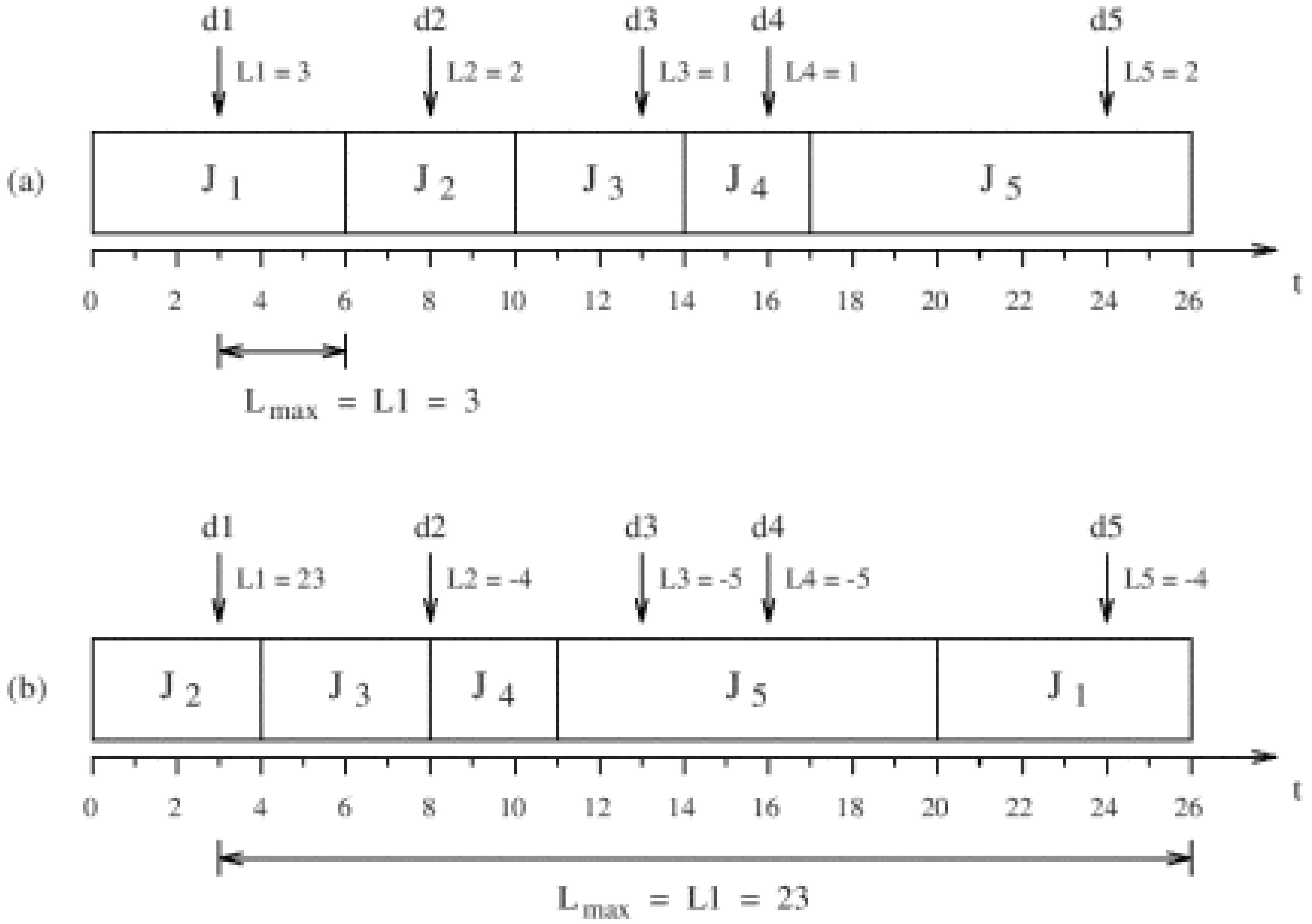
- Lateness $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; if a task completes before the deadline, its lateness is negative.
- Tardiness or exceeding time $E_i = \max(0, L_i)$ is the time a task stays active after its deadline; if a task completes before the deadline, its tardiness is 0.



$\mathbf{r_i}$	release time (arrival time $\mathbf{a_i}$)
$\mathbf{s_i}$	start time
$\mathbf{C_i}$	worst-case execution time (wcet)
$\mathbf{d_i}$	absolute deadline
$\mathbf{D_i}$	relative deadline
$\mathbf{f_i}$	finishing time

Example: Lateness

- Which schedule is better depends on application requirements:
- In (a), the maximum lateness is minimized with $L_{max} = 3$, but all jobs J_1 to J_5 miss their deadlines.
- In (b), the maximal lateness is larger with $L_{max} = 23$, but only one job J_1 misses its deadline.

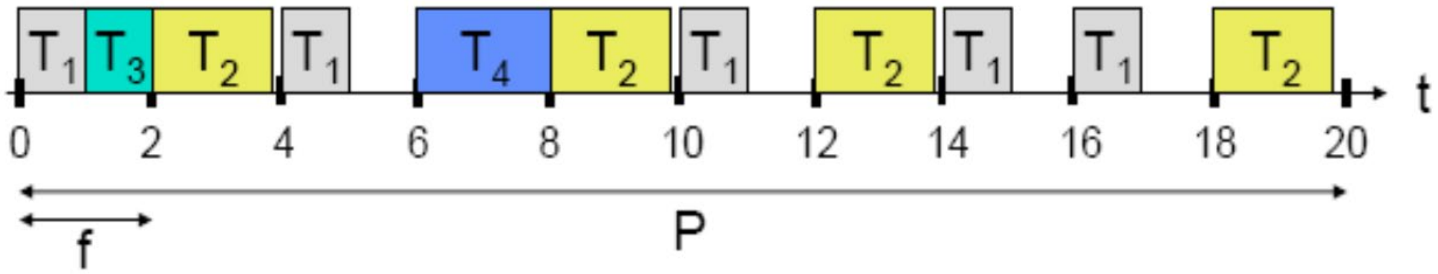


Scheduling Algorithms

- Static cyclic scheduling (offline)
  - All task invocation times are computed offline and stored in a table; Runtime dispatch is a simple table lookup
- Online scheduling;
  - Fixed priority scheduling (also called static-priority scheduling)
    - » Each task is assigned a fixed priority; Runtime dispatch is priority-based
  - Dynamic priority scheduling
    - » Task priorities are assigned dynamically at runtime, e.g., Earliest Deadline First (EDF), Least-Laxity First (LLF)
  - Non-real-time scheduling, e.g., round-robin, multi-level queue…

Static Cyclic Scheduling

- The same schedule is executed once during each hyper-period (least common multiple of all task periods in a taskset).
  - The hyper-period is partitioned into frames of length f.
    - » If a task's WCET exceeds f, then programmer needs to cut it to fit within a frame, and save/restore program state manually
  - The schedule is computed offline and stored in a table. Runtime task dispatch is a simple table lookup.
- Pros:
  - Deals with precedence, exclusion, and distance constraints
  - Efficient, low-overhead for runtime task dispatch
  - Lock-free at runtime
- Cons:
  - Task table can get very large if task periods are relatively prime
  - Maintenance nightmare: complete redesign when new tasks are added, or old tasks are deleted
- Not widely used
  - Except in certain safety-critical systems such as avionic systems



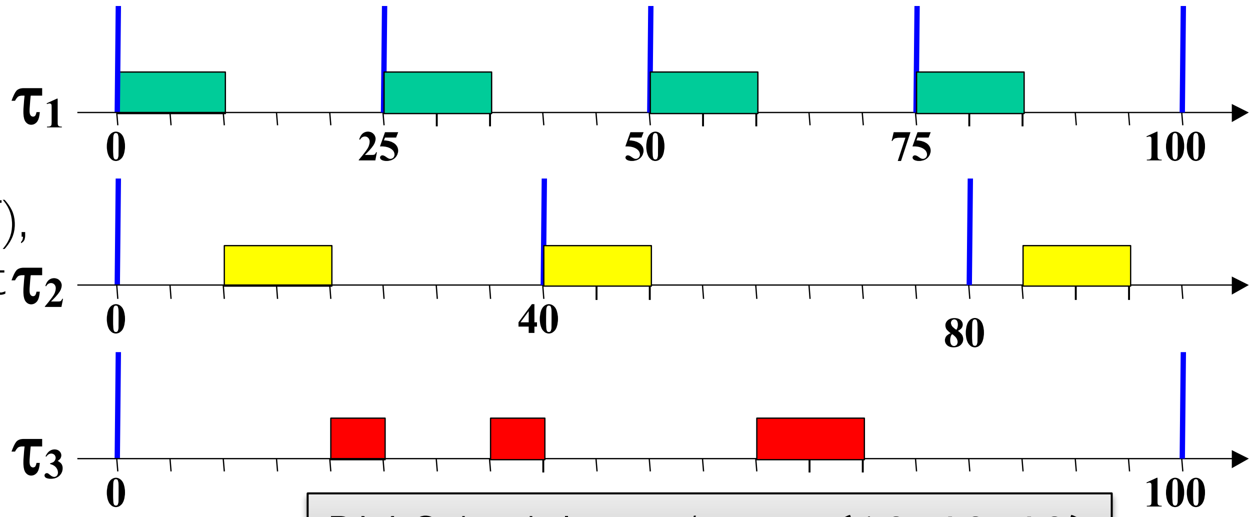
Fixed-Priority Scheduling

Fixed Priority Scheduling

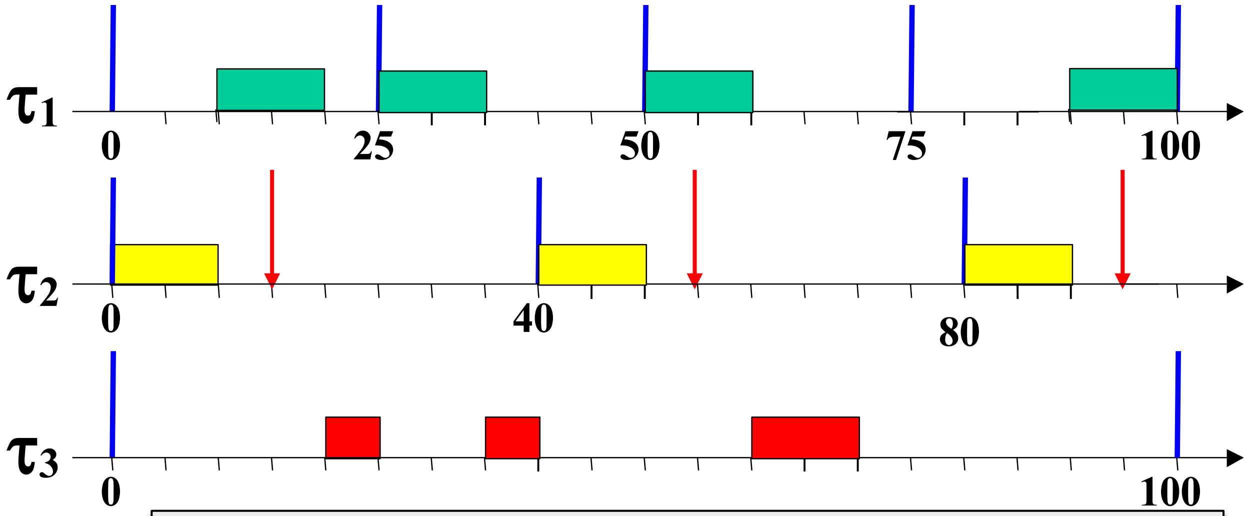
- Each task is assigned a fixed priority for all its invocations
- Pros:
  - Predictability
  - Low runtime overhead
  - Temporal isolation during overload
- Cons:
  - Cannot achieve 100% utilization in general, except when task periods are harmonic
- Widely used in most commercial RTOSes and CAN bus

Rate Monotonic & Deadline Monotonic Scheduling

- Rate Monotonic (RM)
  - Assign higher priority to task with smaller period
  - For implicit deadline tasksets (deadline D = period T), RM is the optimal priority assignment, i.e., if a taskset is not schedulable with RMS priority assignment, then it is not schedulable with any other fixed priority assignment
- Deadline Monotonic (DM)
  - Assign higher priority to task with smaller relative deadline
  - For constrained deadline tasksets (D ≤ T), DM is the optimal priority assignment
- Why do we want D < T?
  - Some events happen infrequently, but need to be handled urgently
- Example taskset: $\tau_1 = (10, 25, 25), \tau_2 = (10, 40, 40)$ or $(10, 40, 15), \tau_3 = (20, 100, 100)$



RM Scheduling w/ $\tau_2 = (10, 40, 40)$



DM Scheduling w/ $\tau_2 = (10, 40, 15)$
(D_2=15 indicated by red downward arrow for τ_2)

Two Schedulability Analysis Approaches

IMPORTANT

- Utilization bound test
  - Calculate total CPU utilization and compare it to a known bound
  - Polynomial time complexity
  - Pessimistic: sufficient but not necessary condition for schedulability

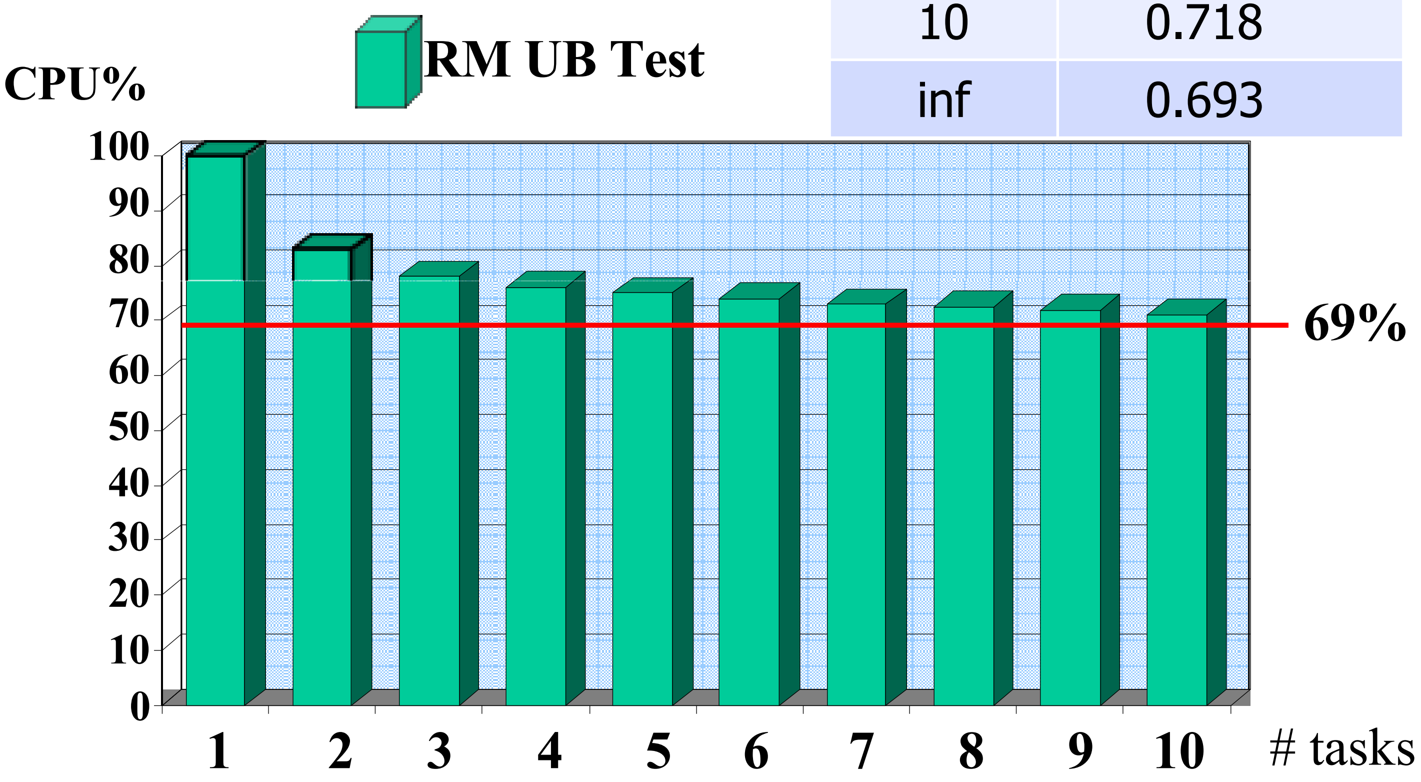
- Response Time Analysis (RTA)
  - Calculate Worst-Case Response Time R_i for each task Tau_i and compare it to its deadline D_i
  - Pseudo-polynomial time complexity
    - » An algorithm runs in pseudo-polynomial time if its running time is polynomial in the numeric value of the input (which is exponential in the length of the input – its number of digits).
  - Accurate: necessary and sufficient condition for schedulability

Utilization Bound Test

IMPORTANT

- A taskset is schedulable under RM scheduling if system utilization $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$
  - $U \rightarrow 0.69$ as $N \rightarrow \infty$
  - Assumptions: task period equal to deadline ($P_i = D_i$); task with smaller period P_i is assigned higher priority (RM priority assignment); tasks are independent (no resource sharing)
- Sufficient but not necessary condition
  - Guaranteed to be schedulable if test succeeds
  - May still be schedulable even if test fails
- Special case: if periods are harmonic (larger periods divisible by smaller periods), then utilization bound is 1 (necessary and sufficient condition)

# Tasks	RM Util Bound
1	1.00
2	0.828
3	0.780
4	0.757
5	0.743
10	0.718
inf	0.693

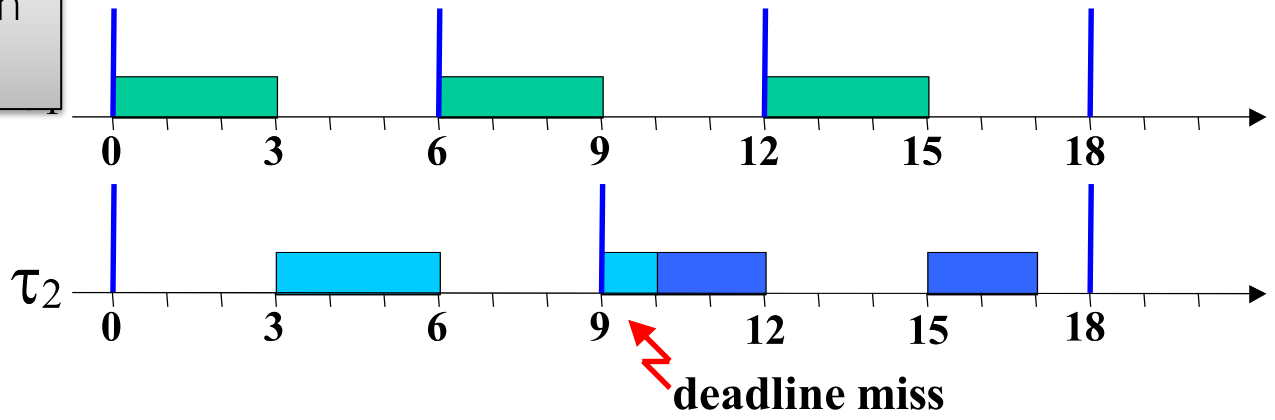


Utilization Bound Test Examples

We use the notation $\tau_i (C_i, T_i, D_i)$ to denote task τ_i with WCET C_i Period T_i, Deadline D_i

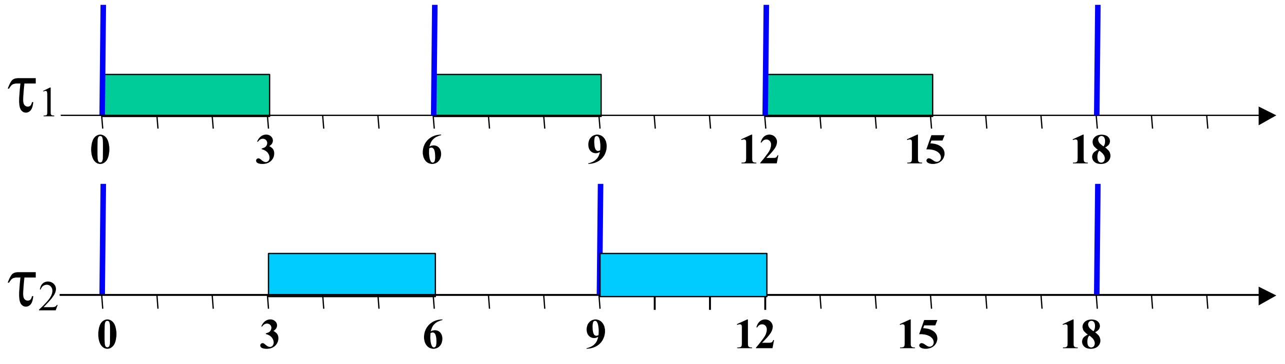
Taskset $\tau_1 (3, 6, 6)$, $\tau_2 (4, 9, 9)$
unschedulable

$$U = \frac{3}{6} + \frac{4}{9} = 0.944 > 0.828$$



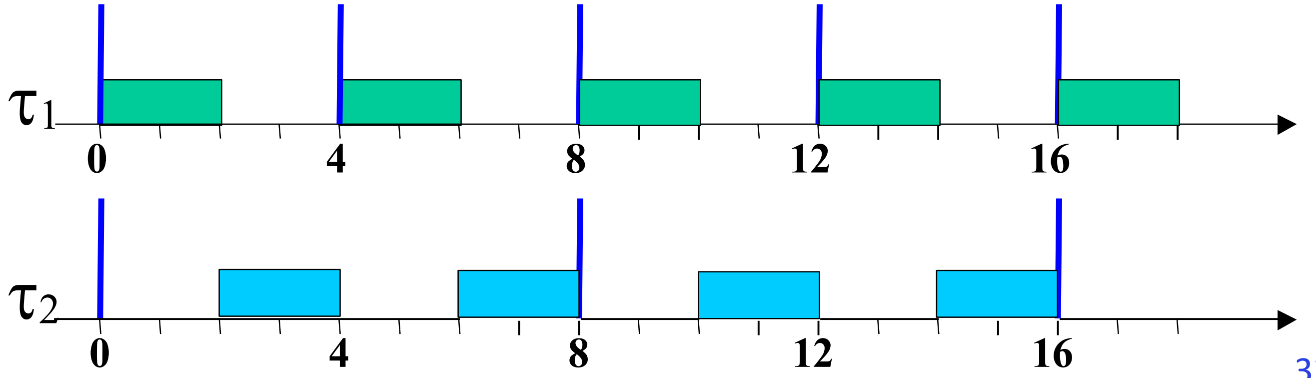
Taskset $\tau_1 (3, 6, 6)$, $\tau_2 (3, 9, 9)$
schedulable (UB test is sufficient but not necessary condition)

$$U = \frac{3}{6} + \frac{3}{9} = 0.833 > 0.828$$



Taskset $\tau_1 (2, 4, 4)$, $\tau_2 (4, 8, 8)$
schedulable (periods are harmonic)

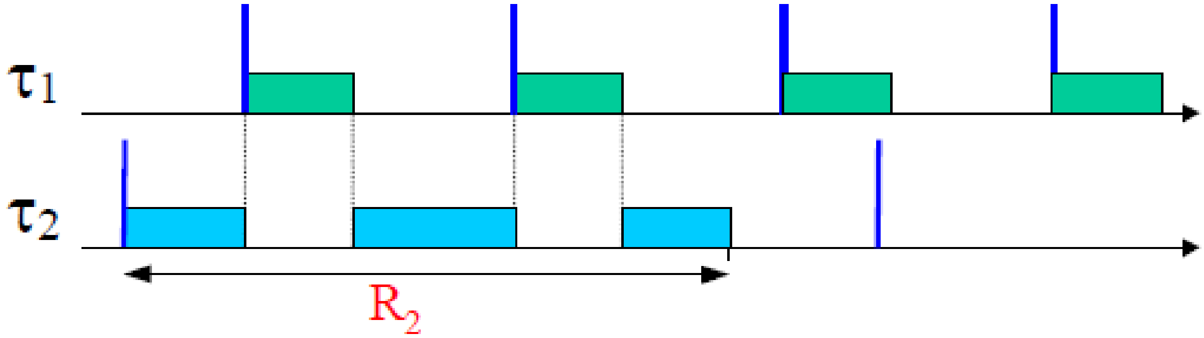
$$U = \frac{2}{4} + \frac{4}{8} = 1.0 > 0.828$$



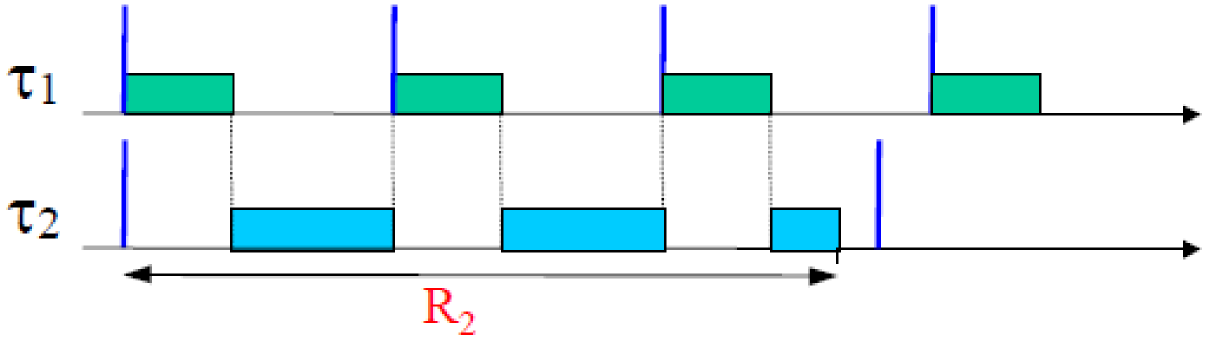
Response Time Analysis (RTA)

- Assumptions:
  - Consider the synchronous taskset: *all tasks are initially released at time 0 simultaneously*, called the *critical instant*. This is the worst-case when each task experiences maximum amount of interference from higher priority tasks: if the taskset is schedulable with this assumption, then it will be schedulable for any other release offset.
  - No resource sharing (no critical sections)
  - Figure shows task τ_2 has the worst-case response time R_2 if it is initially released at time 0, simultaneously with higher priority task τ_1 (lower figure)

τ_1, τ_2 initially released with a non-zero offset, not all at time 0. τ_2 experiences 2 preemptions by τ_1 and has shorter response time



τ_1, τ_2 initially released at time 0 simultaneously, the critical instant. τ_2 experiences 3 preemptions by τ_1 and has longer response time



Response Time Analysis (RTA)

IMPORTANT

- For each task τ_i, compute its Worst-Case Response Time (WCRT) R_i and compare to its deadline D_i. τ_i is schedulable iff $R_i \le D_i$. The taskset is schedulable if all tasks are schedulable (necessary and sufficient condition. "iff" stands for "if and only if").
- Task τ_i's WCRT R_i is computed by solving the following recursive equation to find the *minimum fixed-point solution*:
- $R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$
  - where $hp(i)$ is the set of tasks with higher priority than task τ_i.
  - $\lceil\rceil$ is the ceiling operator, e.g., $\lceil 1.1 \rceil = 2$, $\lceil 1.0 \rceil = 1$
  - $\left\lceil \frac{R_i}{T_j} \right\rceil$ denotes the number of times HP task τ_j pre-empts τ_i during its one job execution; $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$ denotes the total preemption delay caused by HP task τ_j to τ_i during its one job execution

An Example Taskset

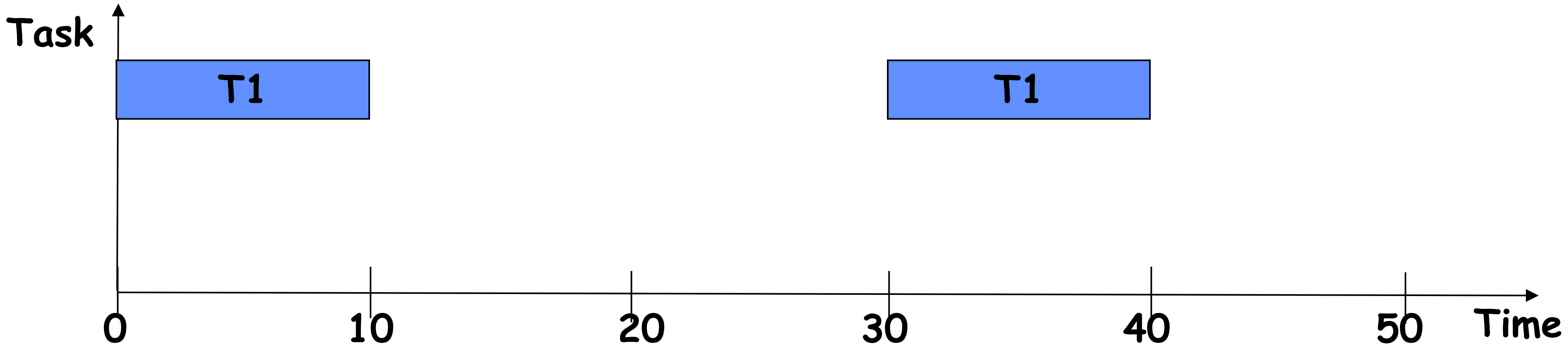
- Consider a taskset of 3 task with (C_i, T_i, D_i) of $(10, 30, 30), (10, 40, 40), (12, 52, 52)$. Under RM, task priorities are assigned to be High for T1, Medium for T2, and Low for T3
- System Utilization $U = \sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{10}{30} + \frac{10}{40} + \frac{12}{52} = 0.81 > 0.78$
  - Utilization Bound $(N = 3) = 3 * \left(2^{1/3} - 1\right) = 0.78$
- Utilization bound test fails, but taskset is actually schedulable

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

Task T1

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

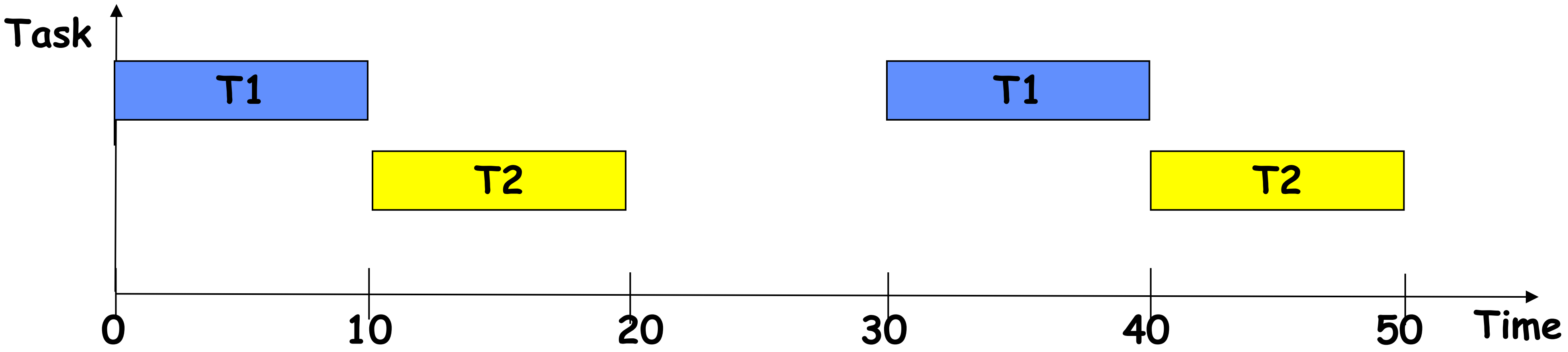
- T1 is the highest priority task, with no interference from other tasks $hp(1) = \emptyset$
- $R_1 = C_1 + 0 = 10$
- $R_1 < D_1$, so T1 is schedulable



Task T2

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

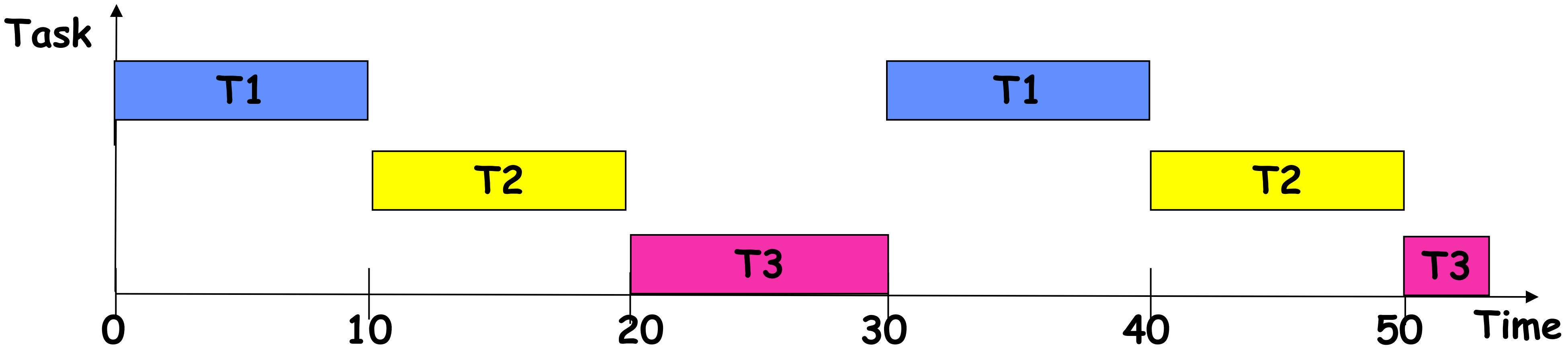
- T2 is the medium priority task, with interference from higher priority Task 1 $hp(2) = 1$
- $R_2 = C_2 + \lceil \frac{R_2}{T_1} \rceil * C_1 = 10 + \lceil \frac{R_2}{30} \rceil * 10$
- Solve for $R2$ iteratively, starting with initial value $R_2 = C_2 = 10$:
  - Iteration 1: $R_2 = 10 + \left\lceil \frac{10}{30} \right\rceil * 10 = 10 + 1 * 10 = 20$
  - Iteration 2: $R_2 = 10 + \left\lceil \frac{20}{30} \right\rceil * 10 = 10 + 1 * 10 = 20$
- Hence $R_2 = 20 < D_2 = 40$, so T2 is schedulable



Task T3

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

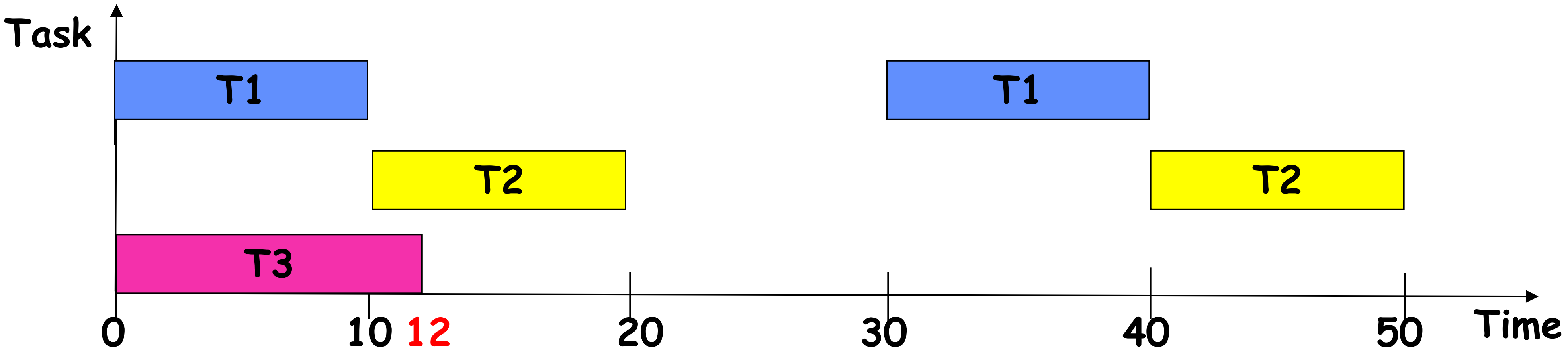
- T3 is the lowest priority task, with interference from higher priority tasks $hp(3) = \{1,2\}$
- $R_3 = C_3 + \lceil \frac{R_3}{T_1} \rceil * C_1 + \lceil \frac{R_3}{T_2} \rceil * C_2 = 12 + \lceil \frac{R_3}{30} \rceil * 10 + \lceil \frac{R_3}{40} \rceil * 10$
- Solve for R_3 iteratively, starting with initial value $R_3 = C_3 = 12$:
  - Iteration 1: $R_3 = 12 + \lceil 12/30 \rceil * 10 + \lceil 12/40 \rceil * 10 = 32$
  - Iteration 2: $R_3 = 12 + \lceil 32/30 \rceil * 10 + \lceil 32/40 \rceil * 10 = 42$
  - Iteration 3: $R_3 = 12 + \lceil 42/30 \rceil * 10 + \lceil 42/40 \rceil * 10 = 52$
  - Iteration 4: $R_3 = 12 + \lceil 52/30 \rceil * 10 + \lceil 52/40 \rceil * 10 = 52$
- Hence $R_3 = 52 \leq D_3 = 52$, so T3 is schedulable



RTA for T3: Initial Condition

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

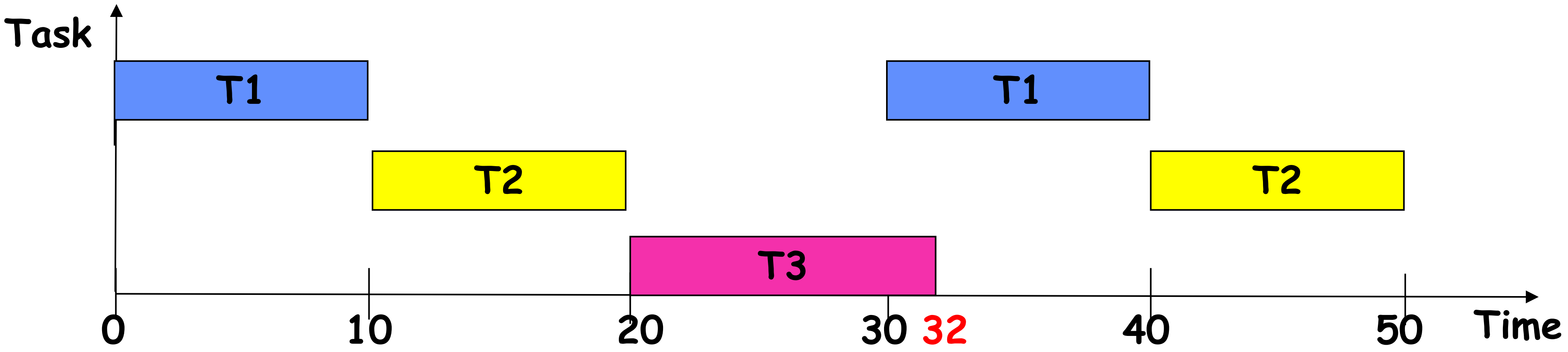
- Initially $R_3 = 12$
- We have not taken into account any preemption delays from higher priority tasks T1 and T2 yet



RTA for Task 3: Iteration 1

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

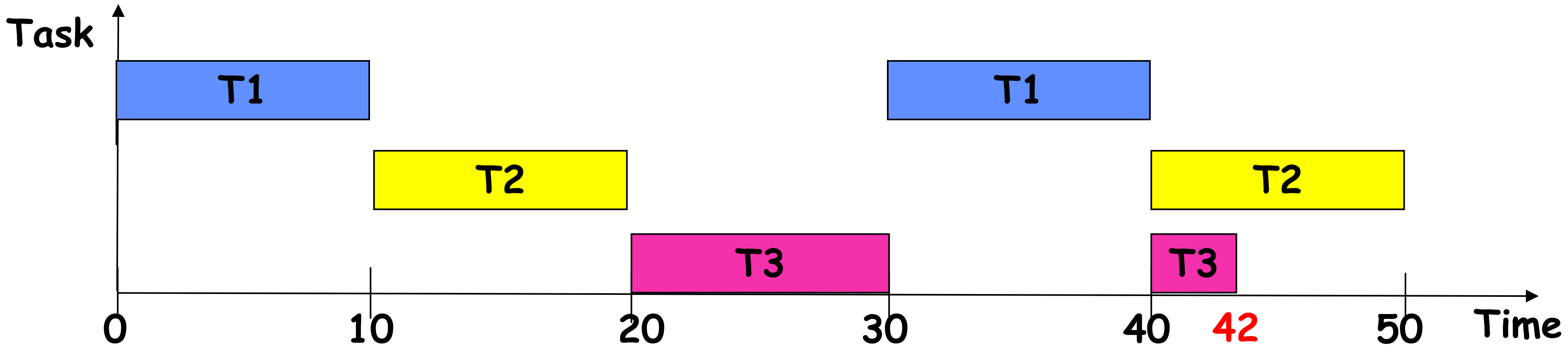
- $R_3 = 12 + \left\lceil \frac{12}{30} \right\rceil * 10 + \left\lceil \frac{12}{40} \right\rceil * 10$
- $= 12 + 1 * 10 + 1 * 10 = 32$
- T1 preempts T3 once, and T2 preempts T3 once
  - since all 3 tasks are released at time 0 (synchronous release time assumption), and T1 and T2 have higher priority than T3



RTA for Task 3: Iteration 2

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

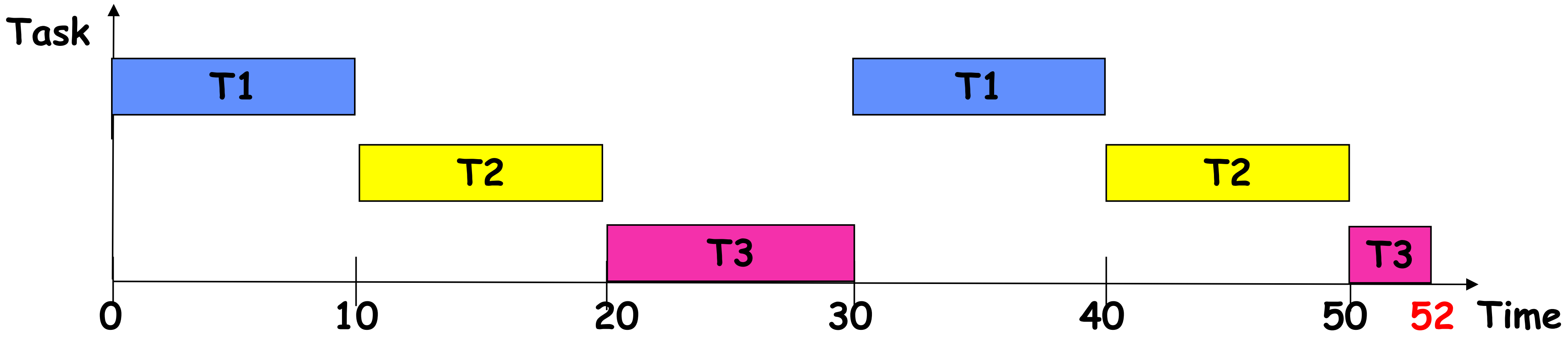
- $R_3 = 12 + \left\lceil \frac{32}{30} \right\rceil * 10 + \left\lceil \frac{32}{40} \right\rceil * 10$
- $= 12 + 2 * 10 + 1 * 10 = 42$
- T1 preempts T3 twice, and T2 preempts T3 once
  - Since T3 has not finished execution at time 30, and another job of higher priority task T1 is released at time 30 and preempts T3



RTA for Task 3: Iteration 3

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

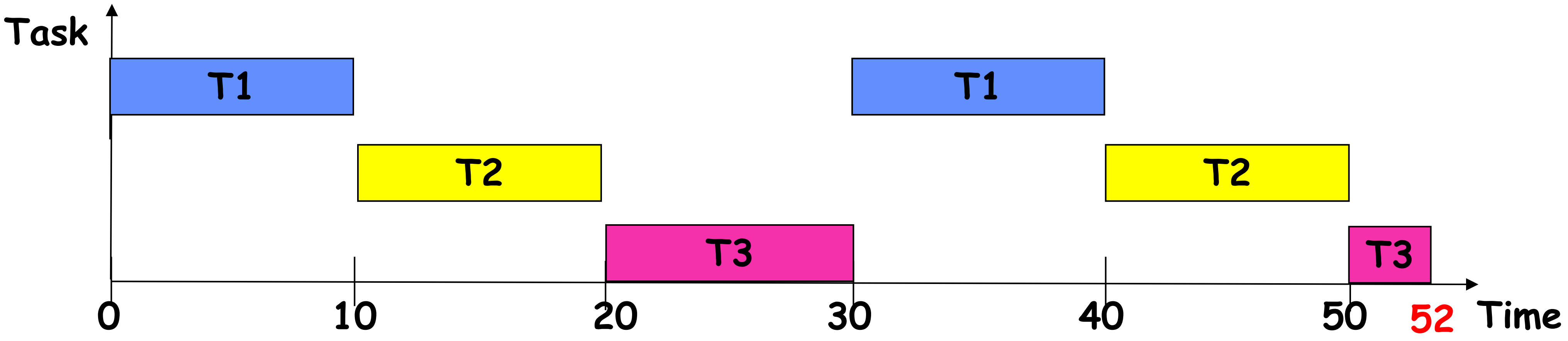
- $R_3 = 12 + \left\lceil \frac{42}{30} \right\rceil * 10 + \left\lceil \frac{42}{40} \right\rceil * 10$
- $= 12 + 2 * 10 + 2 * 10 = 52$
- T1 preempts T3 twice, and T2 preempts T3 twice
  - Since T3 has not finished execution at time 40, and another job of higher priority task T2 is released at time 40 and preempts T3



RTA for Task 3: Iteration 4

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	52	12	L

- $R_3 = 12 + \left\lceil \frac{52}{30} \right\rceil * 10 + \left\lceil \frac{52}{40} \right\rceil * 10 = 12 + 2 * 10 + 2 * 10 = 52$
- T1 preempts T3 twice, and T2 preempts T3 twice
  - Since T3 has finished execution at time 52, and the next arrivals of T1 and T2 are at time 60 and 80, respectively, so T3 will not experience additional preemptions from T1 and T2
- Now the recursive equation has converged, and we have obtained the WCRT of T3 $R_3 = 52 \le D_3 = 52$



When T3 is Unschedulable

- The recursive equation may not converge, i.e., a task's WCRT may be infinity, e.g., suppose we change T2's WCET to be 20, then:

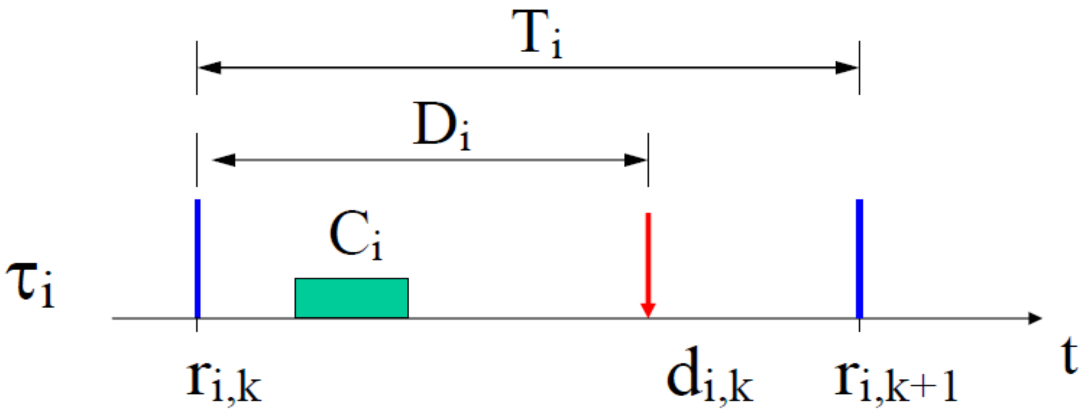
Task	T=D	C	Prio
T1	30	10	H
T2	40	20	M
T3	52	12	L

- $R_3 = C_3 + \lceil \frac{R_3}{T_1} \rceil * C_1 + \lceil \frac{R_3}{T_2} \rceil * C_2 = 12 + \lceil \frac{R_3}{30} \rceil * 10 + \lceil \frac{R_3}{40} \rceil * 20$
- Solve for R_3 iteratively, starting with initial value $R_3 = C_3 = 12$:
  - Iteration 1: $R_3 = 12 + \lceil 12/30 \rceil * 10 + \lceil 12/40 \rceil * 20 = 42$
  - Iteration 1: $R_3 = 12 + \lceil 42/30 \rceil * 10 + \lceil 42/40 \rceil * 20 = 72$
  - Iteration 3: $R_3 = 12 + \lceil 72/30 \rceil * 10 + \lceil 72/40 \rceil * 20 = 82$
  - Iteration 4: $R_3 = 12 + \lceil 82/30 \rceil * 10 + \lceil 82/40 \rceil * 20 = 102$
  - …
- Hence $R_3 \to \infty$. This means that T3's first job never finishes execution due to interferences by higher priority tasks, hence T3 is unschedulable
- It is also possible for T3 to be unschedulable if R_3 converges but it exceeds its deadline D_3, e.g., if we set $D_3 = 50$, then $R_3 = 52 > D_3 = 50$ (another job of T3 is released at time 50, but RTA for the current job is not affected by the newly-released job.)

Task	T=D	C	Prio
T1	30	10	H
T2	40	10	M
T3	50	12	L

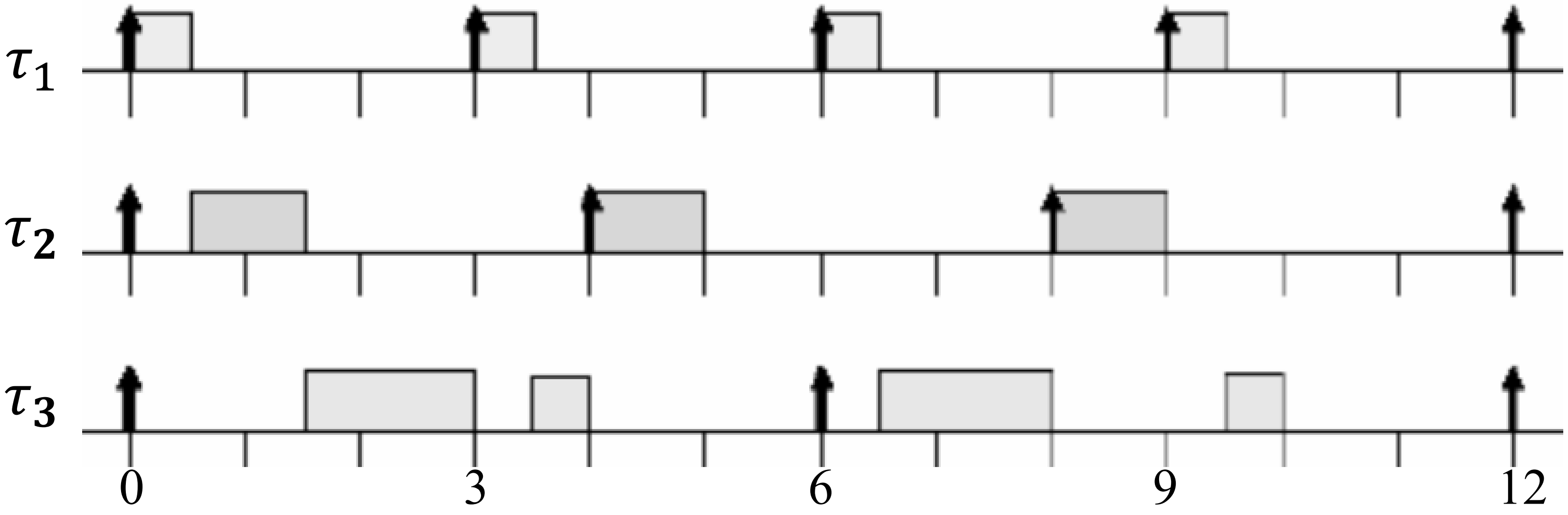
DM for Constrained Deadline Tasksets ($D \leq T$)

- Deadline monotonic (Fixed Priority):
  - A task with smaller relative deadline gets higher priority $P_i \propto 1/D_i$
  - For constrained deadline tasksets ($D \leq T$), DM is the optimal priority assignment
  - No Utilization Bound test for RM or DM, for tasksets with $D \leq T$; must use Response Time Analysis (RTA)
  - Consider a taskset with two tasks both with $(C_i, T_i, D_i) = (1, 2, 1)$. Using RTA, assuming τ_1 has higher priority (since task periods are equal, we can assign either task higher priority), we can determine $R_1 = 1 \leq D_2 = 1, R_2 = 2 > D_2 = 1$, hence it is unschedulable

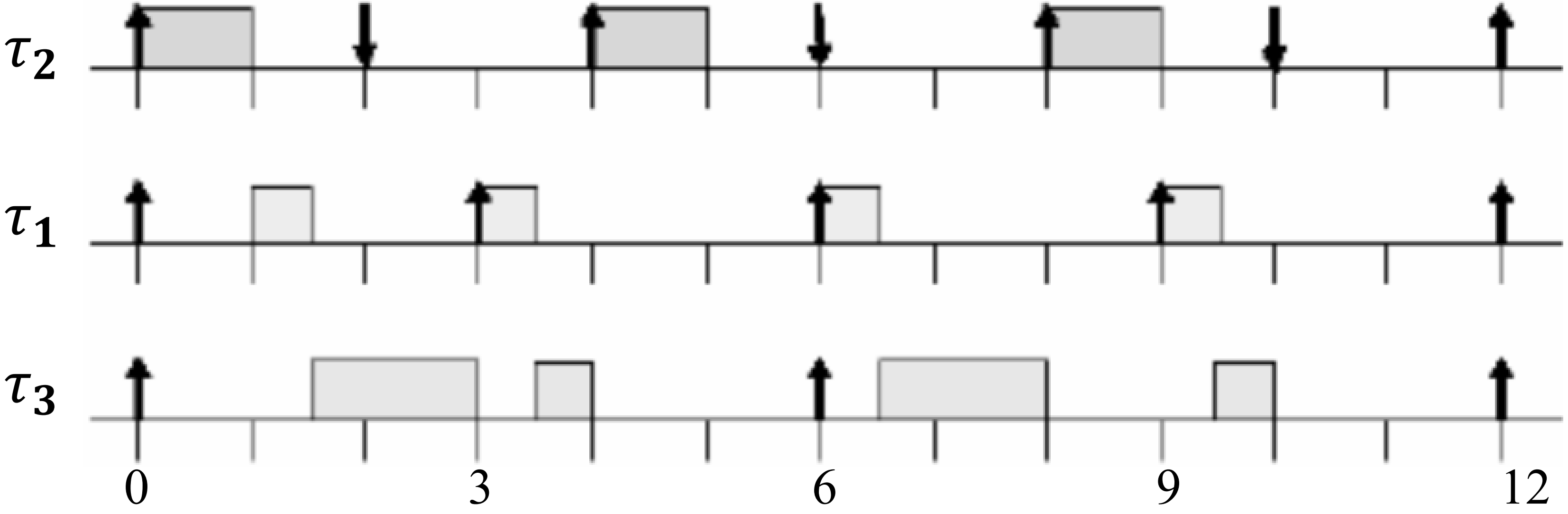


RM vs. DM Example

- Three tasks: $\tau_1 = (0.5, 3, 3), \tau_2 = (1, 4, 4), \tau_3 = (2, 6, 6)$
- Under RM (or DM), priority ordering $\tau_1 > \tau_2 > \tau_3$



- Three tasks with τ_2 assigned a smaller deadline of $D_2 = 2$: $\tau_1 = (0.5, 3, 3), \tau_2 = (1, 4, 2), \tau_3 = (2, 6, 6)$
- Under DM, priority ordering $\tau_2 > \tau_1 > \tau_3$



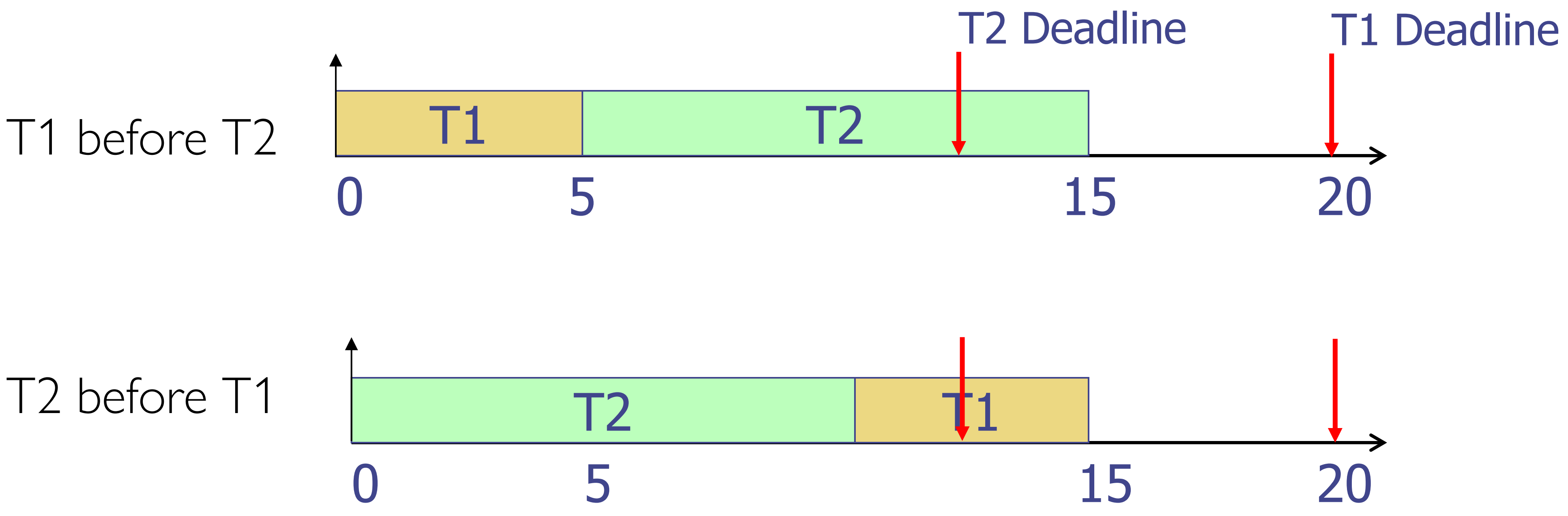
Earliest Deadline First (EDF) Scheduling

Earliest Deadline First (EDF)

- As each jon enters the system, it is assigned a deadline, and its priority is determined by its absolute deadline d_i
  - The job with the earlier deadline is assigned the higher priority
  - This priority assignment is dynamic because a periodic task's priority changes for each job released by the task (vs. fixed-priority scheduling, where a periodic task is assigned a fixed priority for all its jobs)
- Pros:
  - Optimal: can achieve 100% CPU utilization
- Cons:
  - Poor temporal isolation during overload
  - c.f. RM vs. EDF: Robustness under Overload

EDF Scheduling Example

- Say you have two tasks, both released at time 0
  - T1 has WCET 5 ms, with deadline of 20 ms
  - T2 has WCET 10 ms, with deadline of 12 ms
- Non-EDF scheduling: T1 before T2, T2 misses its deadline at 12
- EDF scheduling: T2 before T1, both tasks meet their deadlines

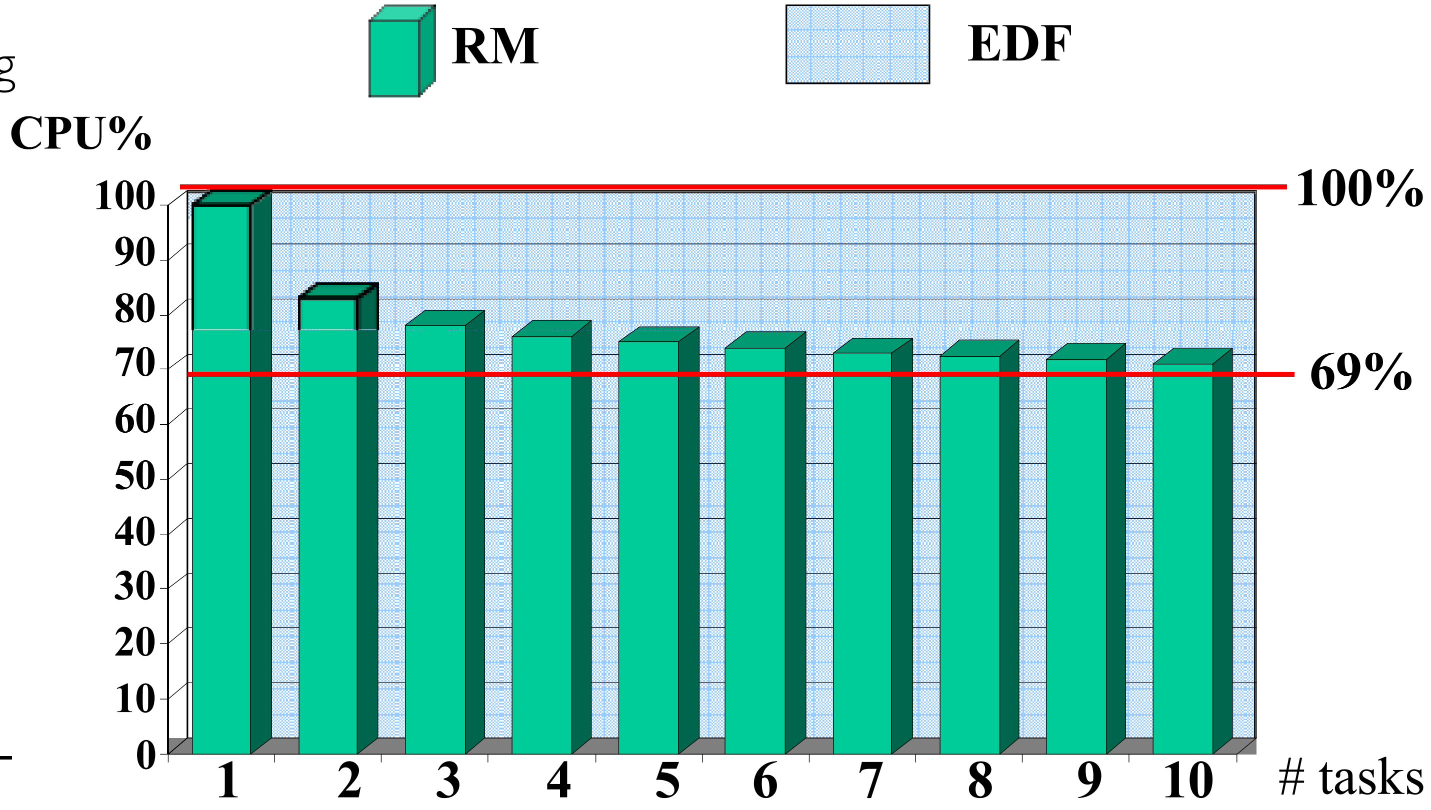


Convention: Upwards arrows indicate arrival time; Downwards arrows indicate deadline

Schedulable Utilization Bound: EDF vs. RM

IMPORTANT

- The schedulable utilization bound for EDF Scheduling is 1 (necessary and sufficient condition):
  - A taskset is schedulable under EDF scheduling iff system utilization does not exceed 1 $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$
    - » "iff" stands for "if and only if"
  - Assumptions: task period equal to deadline ($P_i = D_i$); tasks are independent (no resource sharing)
- Recall: schedulable utilization bound for Fixed-Priority scheduling (sufficient but not necessary condition):
  - A taskset is schedulable under RM scheduling if system utilization $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$
  - $U \to 0.69$ as $N \to \infty$

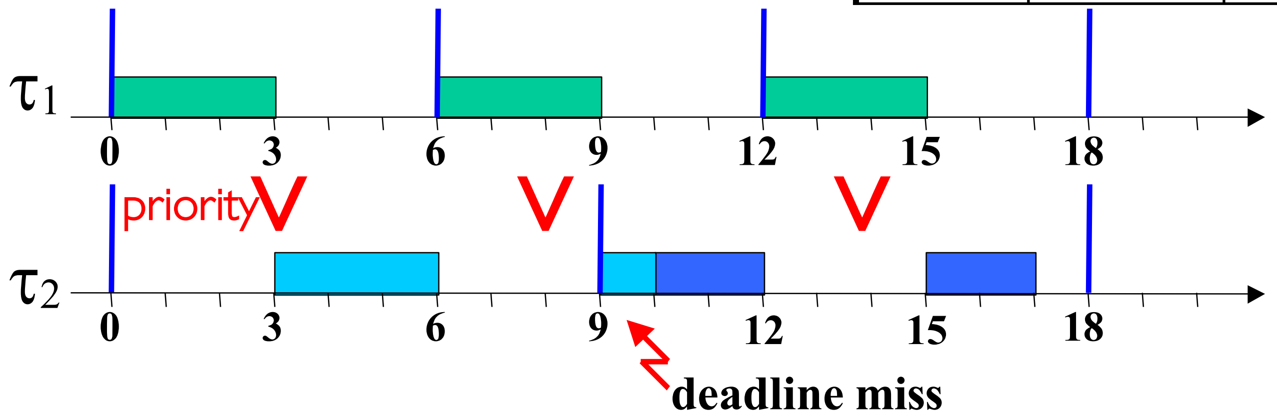


RM vs. EDF Example

Task	T=D	C
τ_1	6	3
τ_2	9	4

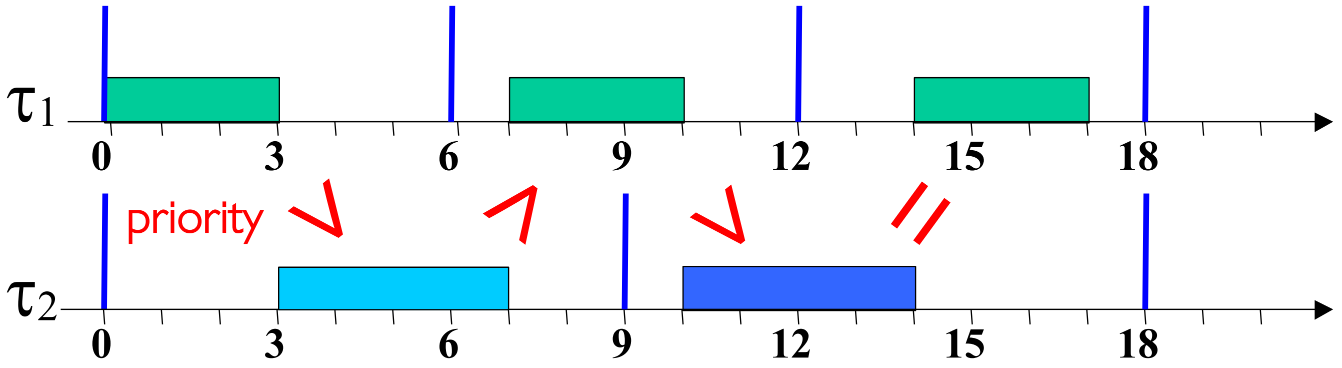
Under RM (Fixed-Priority scheduling), all jobs of τ_1 (with smaller period T=6) have higher priority than all jobs of τ_2 (with larger period T=9). Taskset unschedulable with RM

$$U = \frac{3}{6} + \frac{4}{9} = 0.944 > 0.828$$



Under EDF (Dynamic Priority scheduling), different jobs of τ_1 and τ_2 may have different priorities, depending on their absolute deadlines d_i, which is different for each newly-released job every period. Taskset schedulable with EDF

$$U = \frac{3}{6} + \frac{4}{9} = 0.944 < 1.0$$



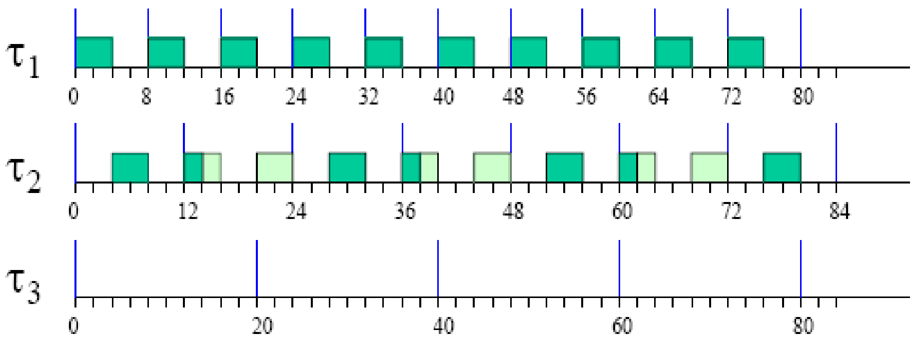
When two jobs have equal priority, the newly arrived job does not preempt the running job

RM vs. EDF: Robustness under Overload

- Under permanent overload, with CPU utilization U > 1
  - Under EDF, all tasks execute at a slower rate with “period rescaling”, i. e., all tasks are delayed evenly
  - Under RM, higher priority tasks are protected while lower priority tasks are delayed or complete blocked
  - Recall Slide 25 Example Lateless
- Under transient overload, when some job overruns (executes longer than expected temporarily)
  - Under EDF, task overruns can cause deadline miss of arbitrary task
  - Under RM: task overruns only affect lower priority tasks
- Conclusion: RM offers better temporal isolation for higher priority tasks, at the expense of lower priority tasks

RM under permanent overload

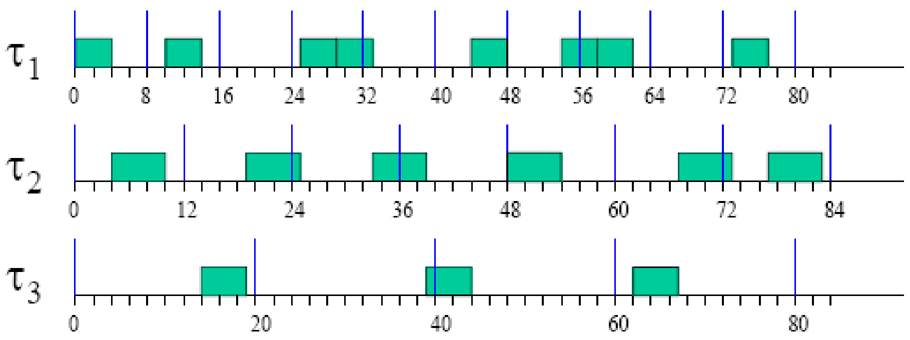
$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$



- High priority tasks execute at the proper rate
- Low priority tasks are completely blocked

EDF under permanent overload

$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$

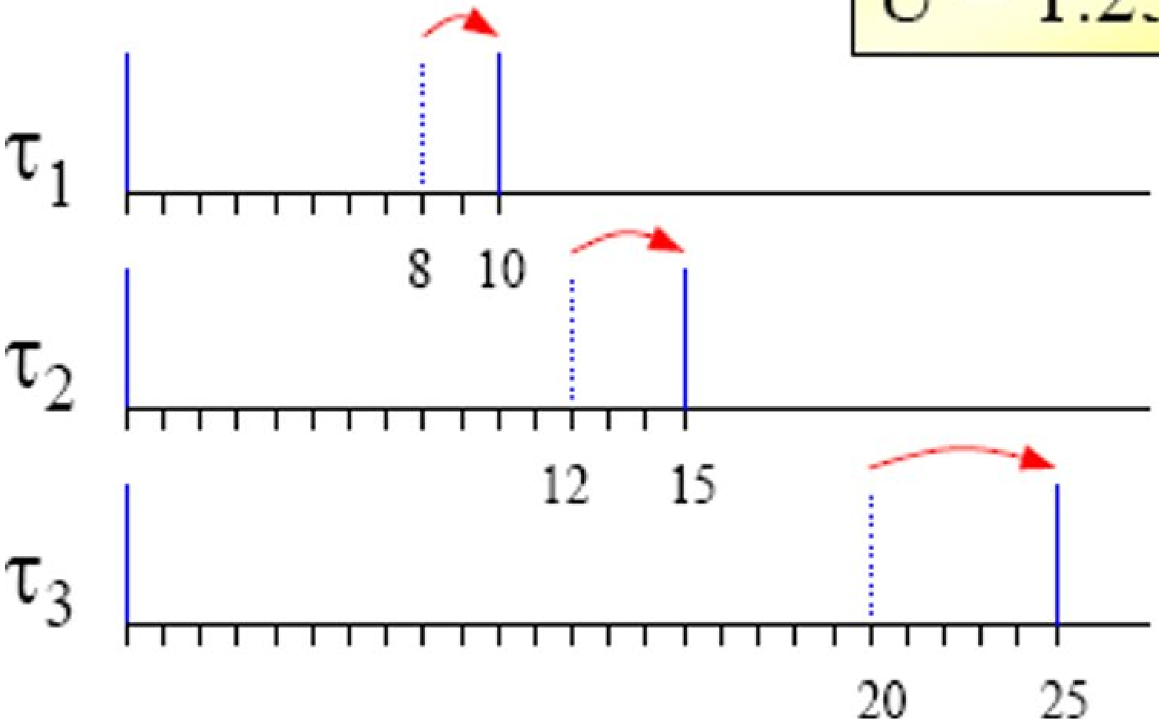


- All tasks execute at a slower rate
- No task is blocked

EDF Period Rescaling

- Theorem on Period Rescaling [Cervin et al. 2003]:
  - If system utilization $U > 1$, tasks are executed with an average period $T_i' = T_i U$ under EDF scheduling

U = 1.25

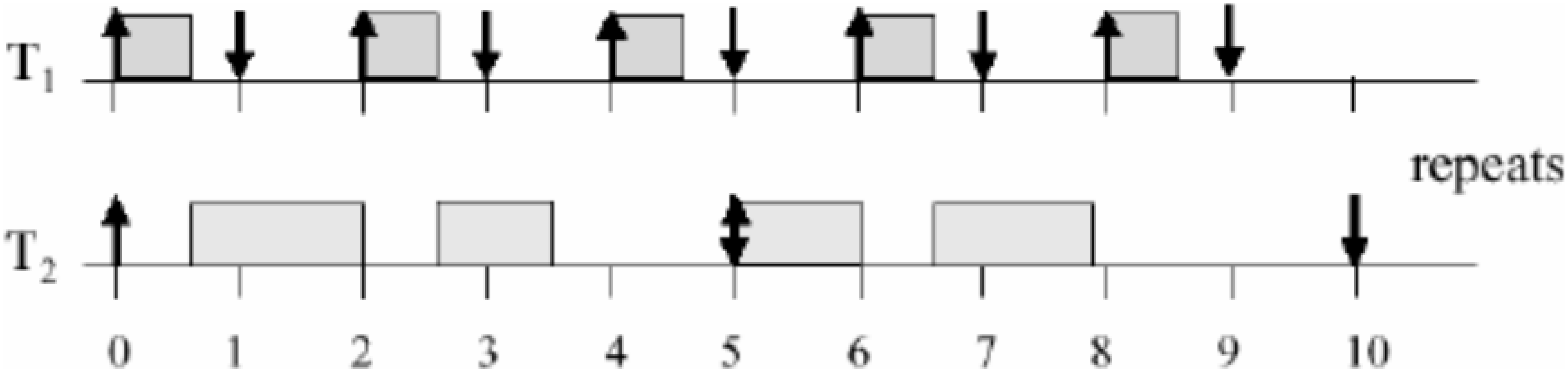


	T_i	T'_i
τ_1	8	10
τ_2	12	15
τ_3	20	25

EDF for Constrained Deadline Tasksets (D ≤ T)

IMPORTANT

- Earliest Deadline First (Dynamic-Priority):
  - A task with smaller absolute deadline gets higher priority $P_i \propto 1/d_i$
  - EDF is still optimal, but instead of Utilization Bound, we use Density Bound to determine schedulability
  - Density of task τ_i is defined as $\delta_i = \frac{C_i}{\min(D_i, T_i)}$. Taskset is schedulable if system density does not exceed 1: $\Delta = \sum_i \delta_i \leq 1$ (sufficient but not necessary condition)
    - » (Demand Bound Function can be used as necessary and sufficient condition (not covered))
  - Consider a taskset with two tasks both with $(C_i, T_i, D_i) = (1, 2, 1)$. It is obviously unschedulable under any scheduling algo. System utilization is $U = \frac{1}{2} + \frac{1}{2} = 1$; System density $\Delta = \frac{1}{1} + \frac{1}{1} = 2$. But we cannot determine schedulabiity based on $\Delta > 1$.
  - Consider a taskset with two tasks $\tau_1 = (0.6, 2, 1), \tau_2 = (2.3, 5, 5)$. $\Delta = \frac{0.6}{1} + \frac{2.3}{5} = 1.06$. Yet the taskset is schedulable under EDF:



Summary of Schedulability Analysis Algorithms

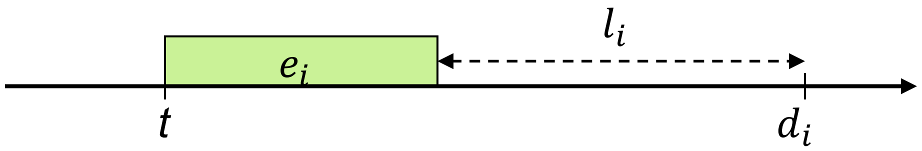
IMPORTANT

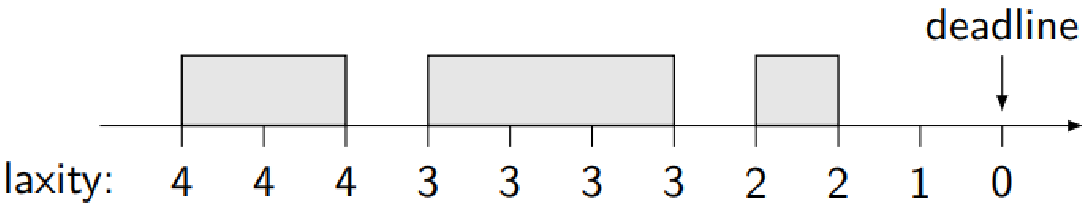
	Fixed-Priority Scheduling		Dynamic Priority Scheduling	
Optimal Scheduling Algorithm	**Rate Monotonic (RM)** Scheduling for implicit deadline taskset (D=T)	**Deadline Monotonic (DM)** Scheduling for constrained deadline taskset (D≤T)	**Earliest Deadline First (EDF)** Scheduling for implicit deadline taskset (D=T)	**Earliest Deadline First (EDF)** Scheduling for constrained deadline taskset (D≤T)
Schedulability Analysis Algorithm	Utilization Bound (UB) test $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$ (sufficient condition) or Response Time Analysis (RTA) (necessary and sufficient) $R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$	RTA Response Time Analysis (RTA) (necessary and sufficient) $R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$	Utilization Bound (UB) test $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$ (necessary and sufficient)	Density Bound test $\Delta = \sum_i \frac{C_i}{\min(D_i, T_i)} \leq 1$ (sufficient condition) or Demand Bound Function (not covered)

Least Laxity First (LLF) Scheduling

Least Laxity First (LLF)

- LLF assigns priority to jobs dynamically based on their laxity (slack)
  - With absolute deadline d_i and remaining execution time e_i, laxity of τ_i's job at time t is $l_i = d_i - t - e_i$. Job with the smallest laxity has the highest priority
  - If an active job runs in the previous time slot, then its laxity remains the same, as t is incremented by 1, and e_i is decremented by 1
  - If an active job does not run in the previous time slot, then its laxity is decremented by 1, as t is incremented by 1, and e_i remains the same
  - While an active job waits and does not run, its laxity decreases and its priority increases

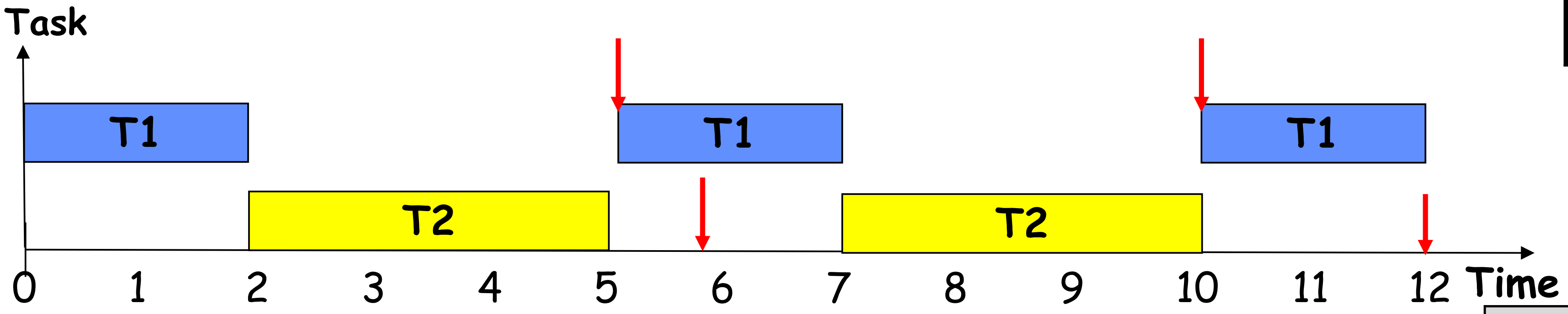




- EDF and LLF are both optimal scheduling algorithms, i.e., they both have schedulable utilization bound of 1
  - LLF incurs frequent context switches, hence is less practical than EDF

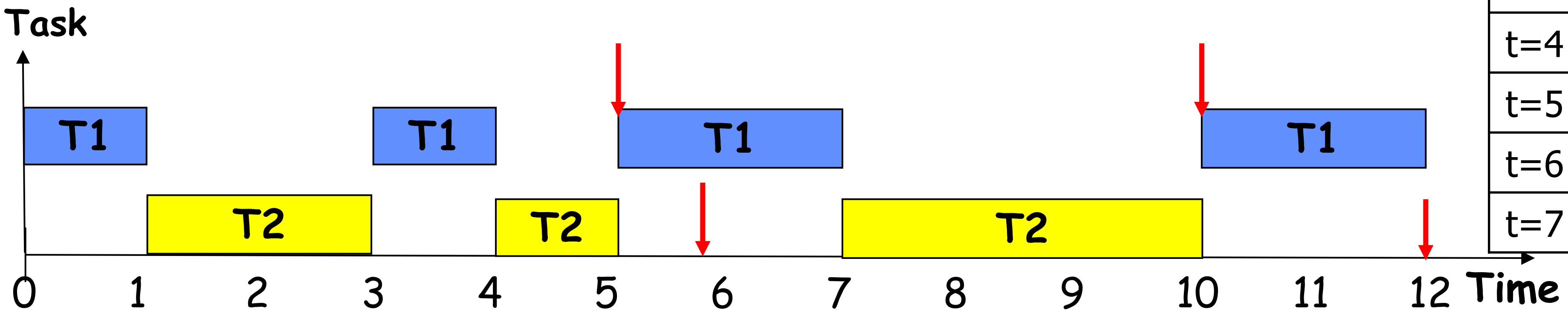
RM, EDF, LLF Example

Task	T=D	C
T1	5	2
T2	6	3



EDF and RM have the same schedule

Time	τ_1 Laxity	τ_2 Laxity	Running Task
t=0	5-0-2=3	6-0-3=3	τ_1 (tie)
t=1	5-1-1=3	6-1-3=2	τ_2
t=2	5-2-1=2	6-2-2=2	τ_2 (tie)
t=3	5-3-1=1	6-3-1=2	τ_1
t=4	τ_1 done	6-4-1=1	τ_2
t=5	10-5-2=3	τ_2 done	τ_1
t=6	10-6-1=3	12-6-3=3	τ_1 (tie)
t=7	τ_1 done	12-7-3=2	τ_2



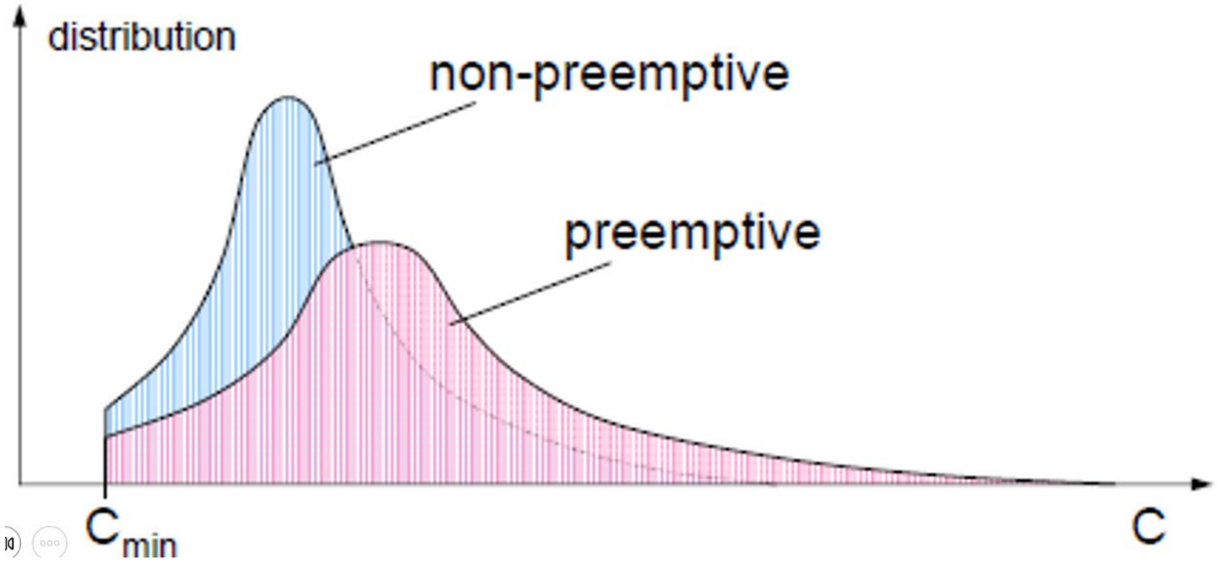
LLF has more frequent context switches

Preemptive vs. Non-Preemptive Scheduling

Preemptive vs. Non-Preemptive Scheduling

- Non-preemptive scheduling pros:
  - It reduces runtime overhead
    - Less context switches
    - No mutex locks needed for critical sections
  - It preserves program locality, improving the effectiveness of CPU cache
    - As a result, task WCET becomes smaller and execution time distribution becomes more predictable (shown on right)
  - Sometimes NP scheduling can improve schedulability
- Cons:
  - Reduced schedulability
  - Scheduling anomalies

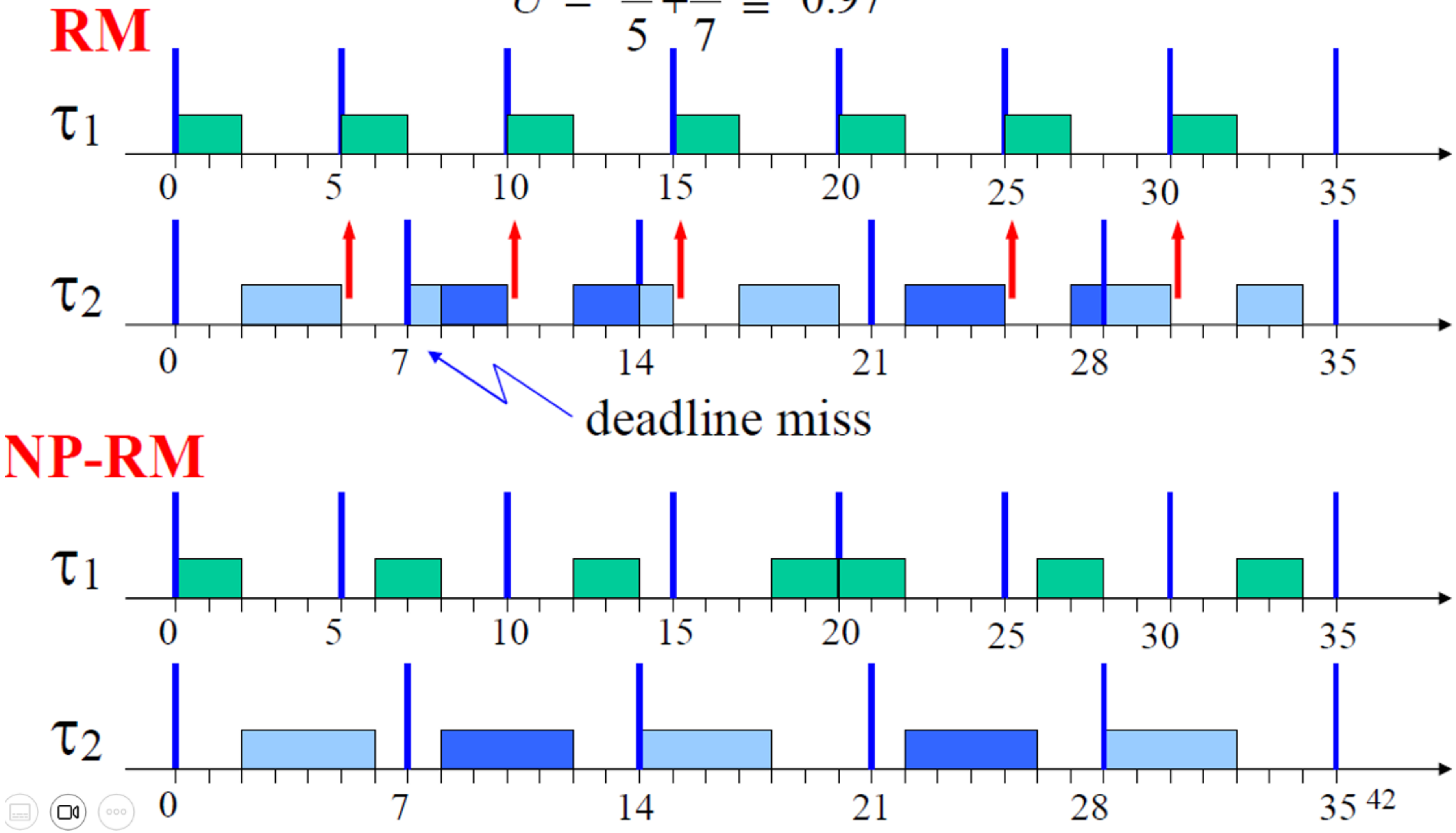
- Preemptive scheduling pros:
  - Better schedulability (higher CPU utilization)
- Cons:
  - Runtime overhead due to frequent context-switches
  - Destroys program locality so task WCET becomes larger



Sometimes NP Scheduling Improves Schedulability

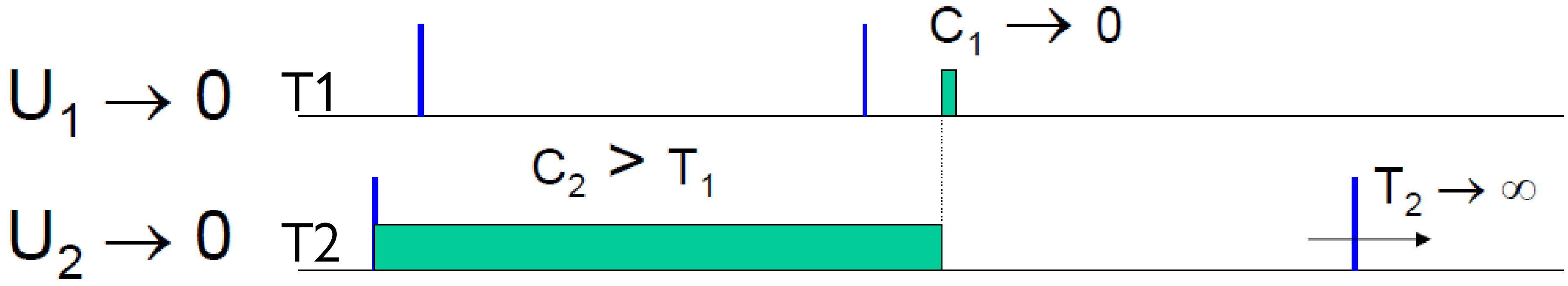
- An example where NP scheduling improves schedulability (for fixed-priority sched' '' `

$$U = \frac{2}{5} + \frac{4}{7} \cong 0.97$$



Disadvantage of NP Scheduling: Reduced Schedulability

- In general, NP scheduling reduces schedulability. The utilization bound under NP scheduling drops to zero due to blocking time
- An example with with two tasks T1 and T2, CPU utilization of nearly 0, yet unschedulable.
  - If C_2 (WCET of T2) $\geq$ T_1 (period of T1), then the taskset is unschedulable with arbitrarily small system CPU utilization $\frac{C_1}{T_1} + \frac{C_2}{T_2} \rightarrow \frac{0}{T_1} + \frac{C_2}{\infty}$ (when C_1 goes to 0 and T_2 goes to infinity)
  - This example is valid whether τ_1 or τ_2 has higher priority: even if τ_1 has higher priority, it may be released very shortly after τ_2 is released at time 0, and it has to wait for τ_2 to finish due to NP scheduling



Disadvantage of NP Scheduling: Scheduling Anomalies

- Scheduling anomaly: three tasks under NP fixed-priority scheduling witjh priority ordering $\tau_1 > \tau_2 > \tau_3$ and NP
- Doubling the processor speed (reducing task execution times by half) makes task τ_1 miss its deadline, since τ_3 starts earlier before τ_1 is released, causing a long delay to it due to NP scheduling (this anomaly does not occur for preemptive scheduling)

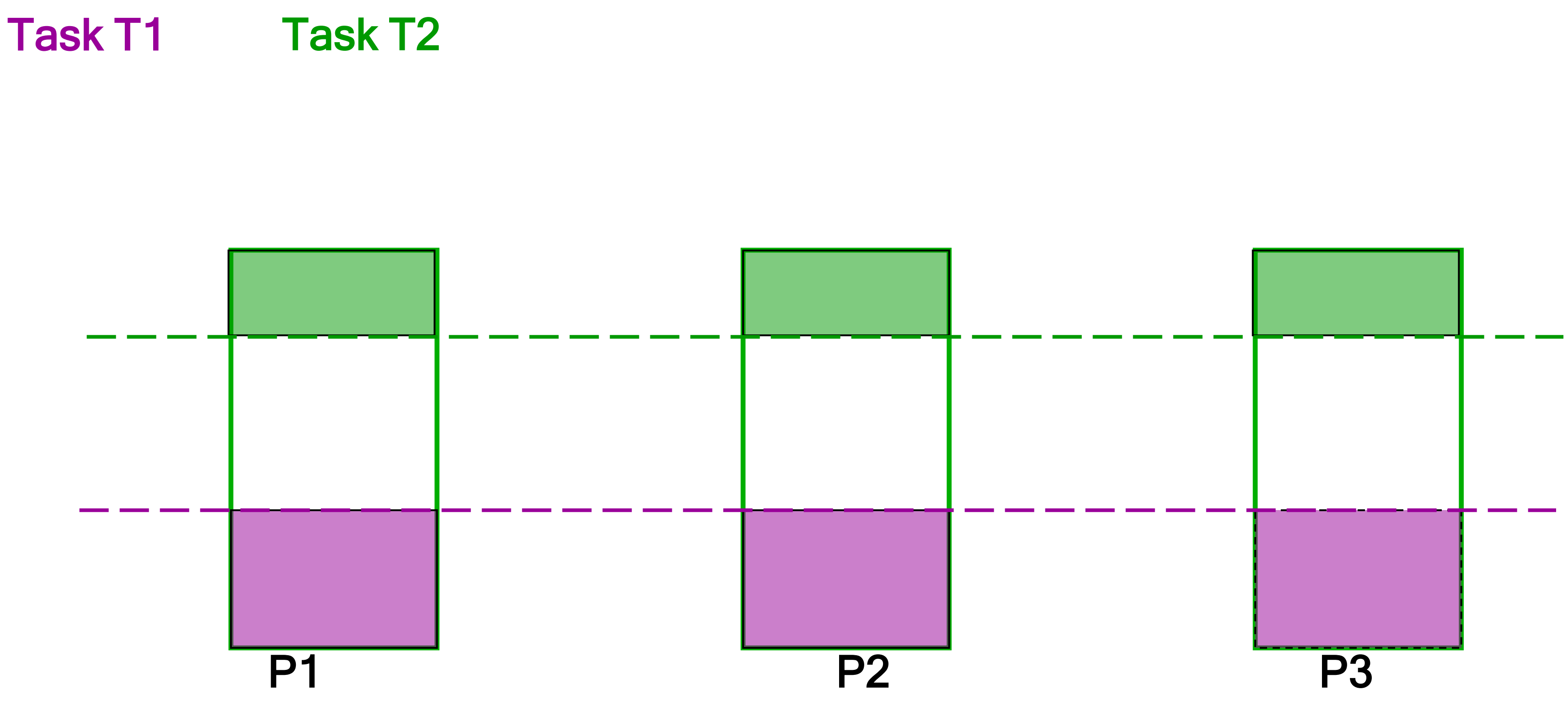
Multiprocessor Scheduling

Multiprocessor models

- Identical multiprocessors:
  – each processor has the same computing capacity
- Uniform multiprocessors:
  – different processors have different computing capacities
- Heterogeneous multiprocessors:
  – each (task, processor) pair may have a different computing capacity
- MP scheduling
  – Many NP-hard problems, with few optimal results, mainly heuristic approaches
  – Only sufficient schedulability tests

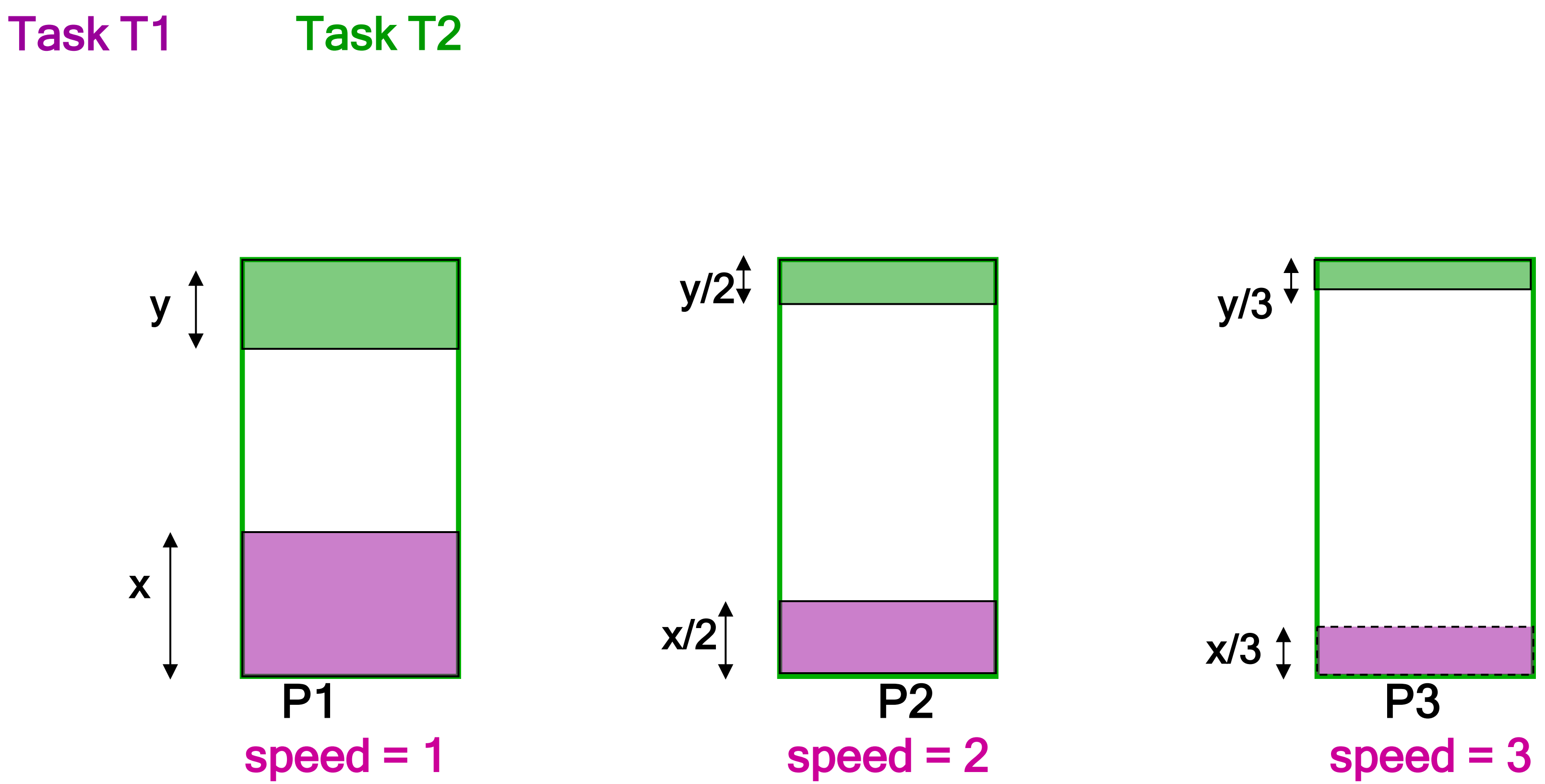
Multiprocessor Models

Identical multiprocessors: each processor has the same speed



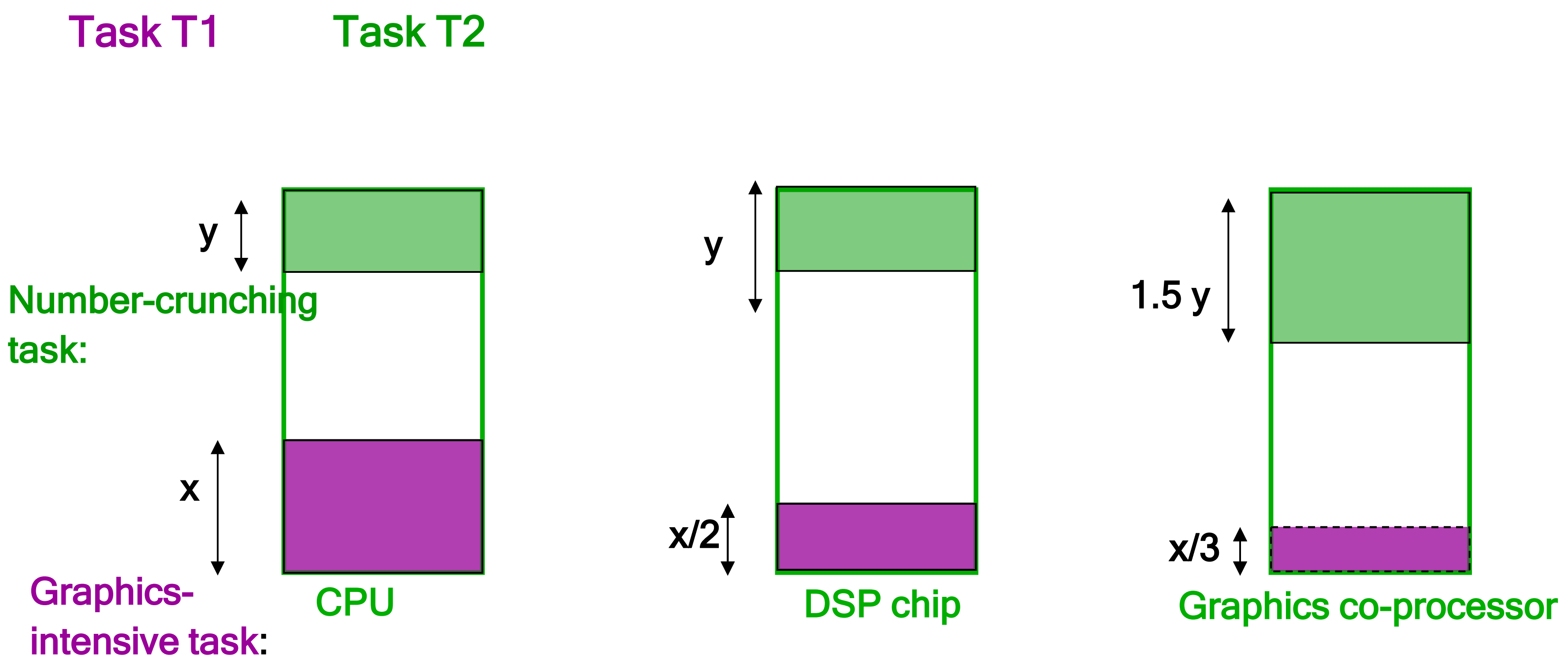
Multiprocessor Models

Uniform multiprocessors: different processors have different speeds



Multiprocessor Models

Heterogeneous multiprocessors: each **(task, processor)** pair may have a different relative speed, due to specialized processor architectures

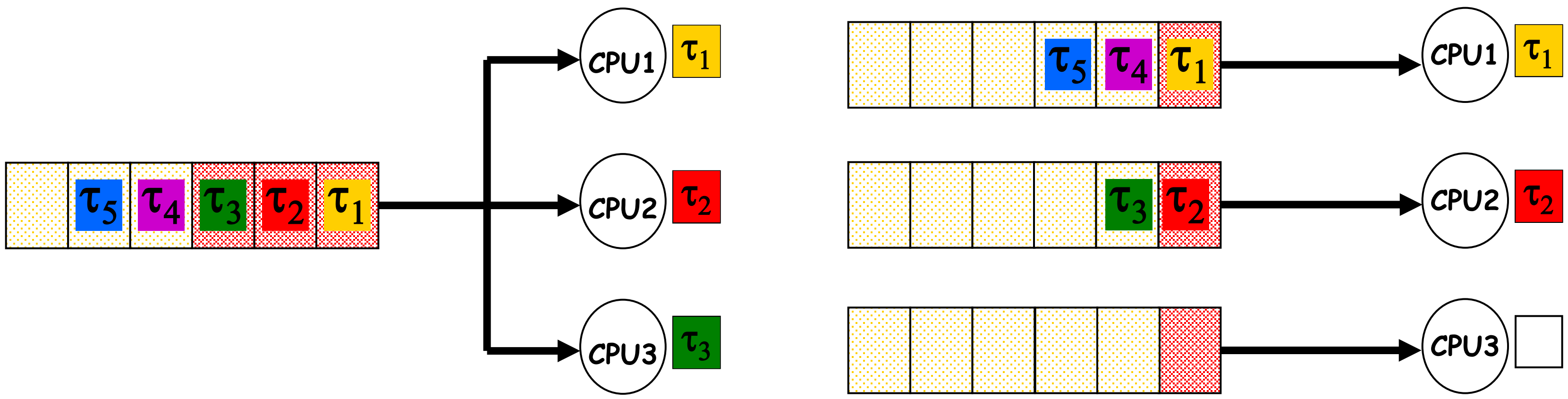


Global vs partitioned scheduling

- Global scheduling
  – All ready jobs are kept in a common (global) queue; when selected for execution, a job can be dispatched to an arbitrary processor, even after being preempted
- Partitioned scheduling
  – Each task may only execute on a specific processor

Global scheduling:
Single system-wide queue

Partitioned scheduling:
per-processor queues



Global Scheduling vs. Partitioned Scheduling

- Global Scheduling
- Advantages:
  - Runtime load-balancing across cores
    - More effective utilization of processors and overload management
  - Supported by most multiprocessor operating systems
    - Windows, Linux, MacOS...
- Disadvantages:
  - Low schedulable utilization
  - Weak theoretical framework

- Partitioned Scheduling
- Advantages:
  - Mature scheduling framework
  - Uniprocessor scheduling theory scheduling are applicable on each core; uniprocessor resource access protocols (PIP, PCP…) can be used
  - Partitioning of tasks can be done by efficient bin-packing algorithms
- Disadvantages:
  - No runtime load-balancing; surplus CPU time cannot be shared among processors

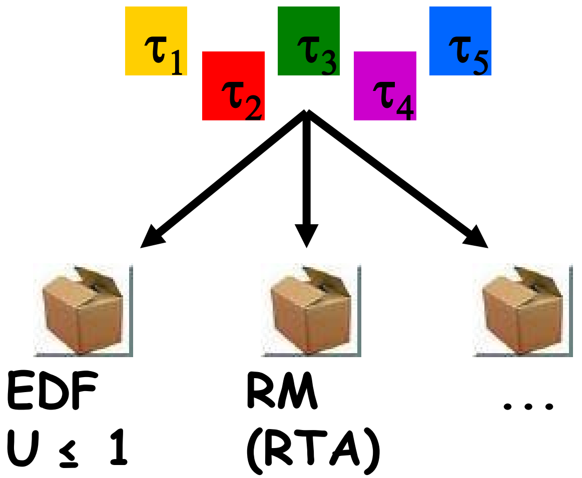
Partitioned Scheduling

- Scheduling problem reduces to:

Bin-packing problem + **Uniprocessor scheduling problem**

NP-hard | **Well-known**

Various heuristics algorithms
First Fit (FF)
Best Fit (BF)
Worst Fit (WF)
Next Fit (NF)
...



Partitioned Scheduling

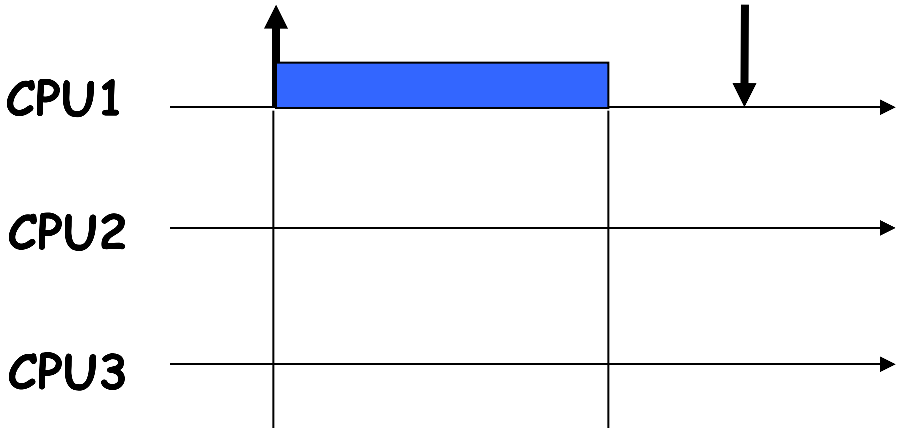
- Bin-packing algorithms:
  - The problem concerns packing objects of varying sizes in boxes (”bins”) with some optimization objective, e.g., minimizing number of used boxes (best-fit), or minimizing the maximum workload for each box (worst-fit)
- Application to multiprocessor scheduling:
  - Bins are represented by processors and objects by tasks
  - The decision whether a processor is ”full” or not is derived from a utilization-based feasibility test.
- Since optimal bin-packing is a NP-complete problem, partitioned scheduling is also NP-complete
- Example: Rate-Monotonic-First-Fit (RMFF): (Dhall and Liu, 1978)
  - Let the processors be indexed as 1, 2, …
  - Assign the tasks to processor in the order of increasing periods (that is, RM order)
  - For each task τ_i, choose the lowest previously-used processor *j* such that τ_i, together with all tasks that have already been assigned to processor j, can be feasibly scheduled according to the utilization-based schedulability test
  - Additional processors are added if needed

Assumptions for Global Scheduling

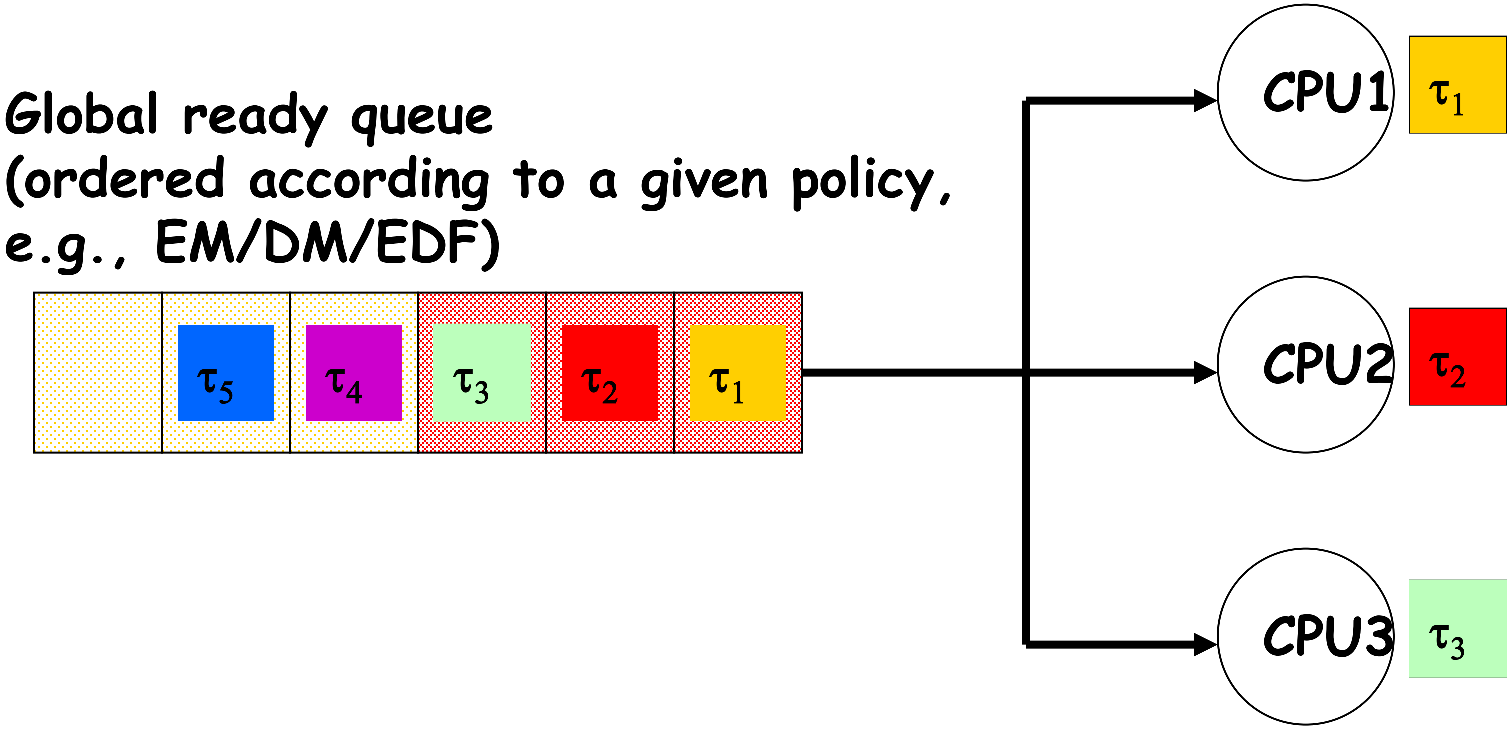
- Identical multiprocessors
- Work-conserving:
  - At each instant, the highest-priority jobs that are eligible to execute are selected for execution upon the available processors
  - No processor is ever idle when the ready queue is non-empty
- Preemption and Migration support
  - A preempted task can resume execution on a different processor with 0 overhead, as cost of preemption/migration is integrated into task WCET
- No job-level parallelism
  - the same job cannot be *simultaneously* executed on more than one processor, i.e., we do not consider parallel programs that can run on multiple processors in parallel

Source of Difficulty

- The "no job-level parallelism" assumption leads to difficult scheduling problems
- "The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors" [Liu'69]



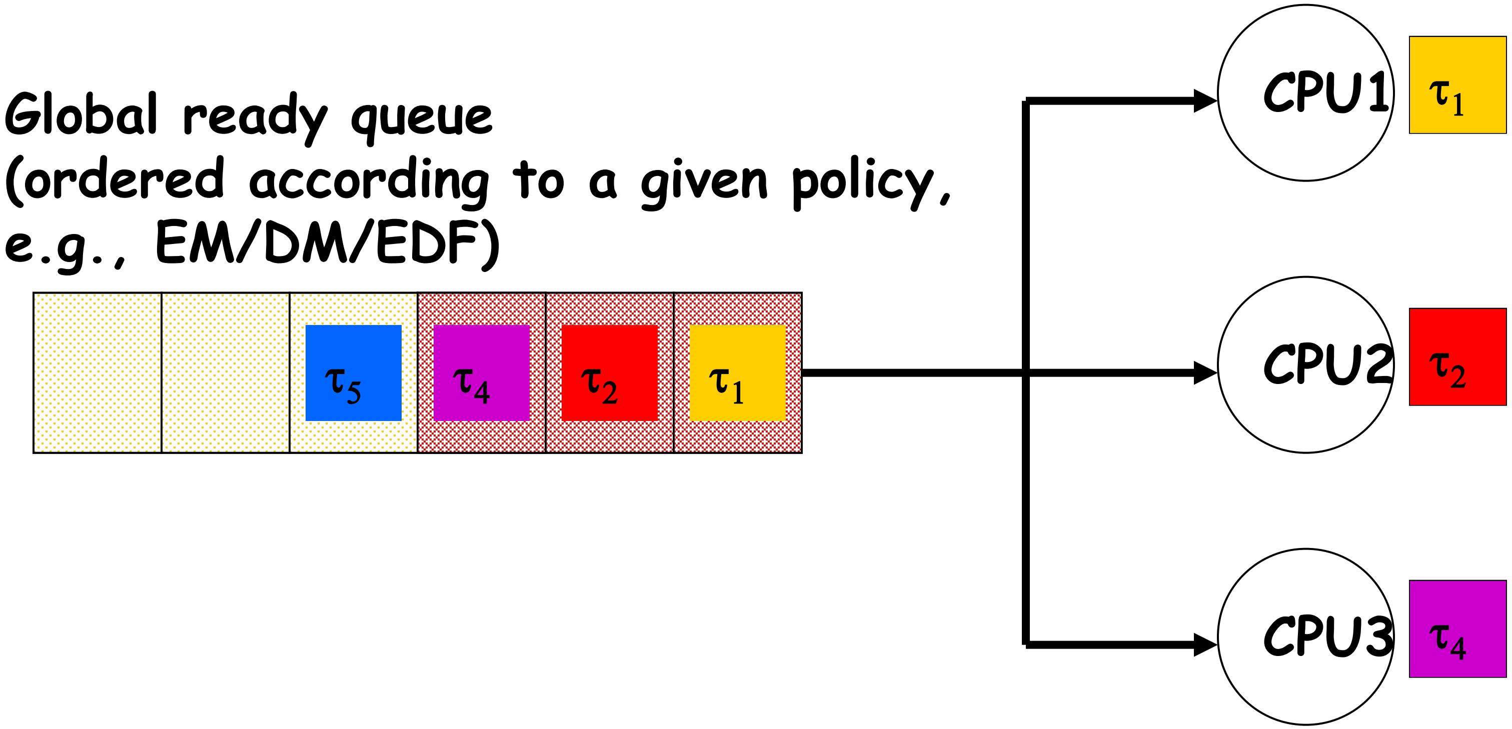
Global scheduling example



The first m jobs in the queue are scheduled upon the m CPUs

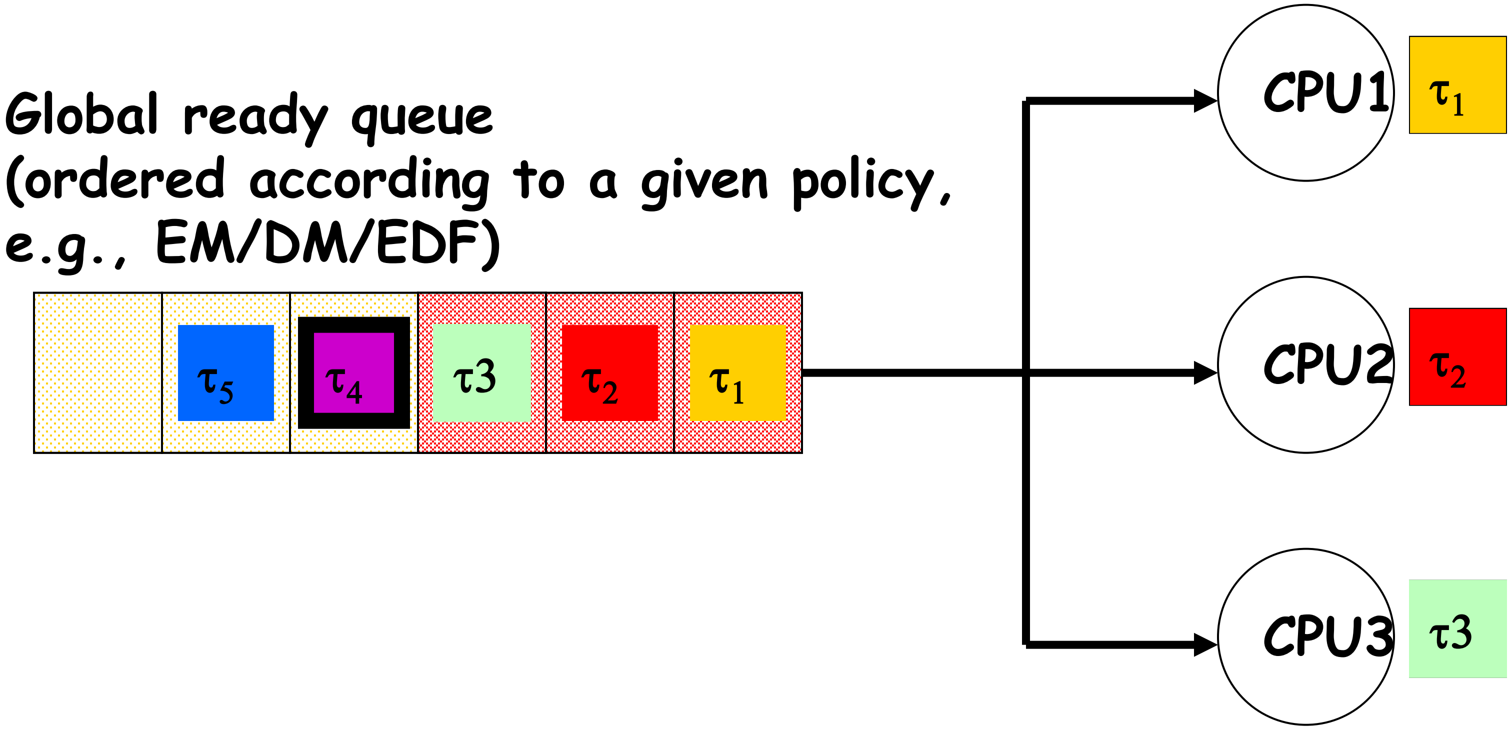
Global scheduling example

Global ready queue
(ordered according to a given policy,
e.g., EM/DM/EDF)



When a job τ_3 finishes its execution, the next job in the queue τ_4 is scheduled on the available CPU

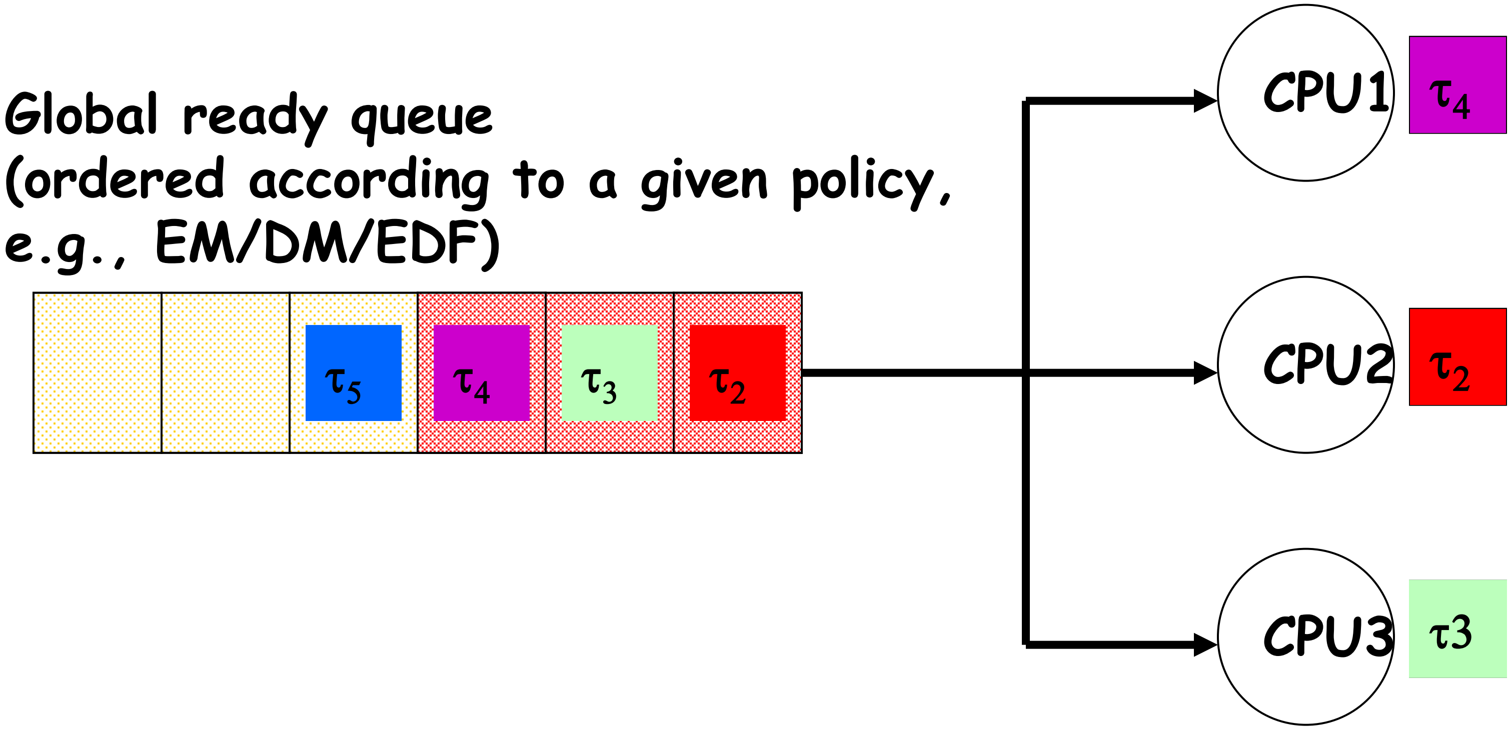
Global scheduling example



When a new higher-priority job τ_3 arrives in its next period T_3, it preempts the job with lowestpriority τ_4 among the executing ones

Global scheduling example

Global ready queue
(ordered according to a given policy,
e.g., EM/DM/EDF)



When another job τ_1 finishes its execution, the preempted job τ_4 can resume its execution. Net effect: τ_4 "migrated" from CPU3 to CPU1

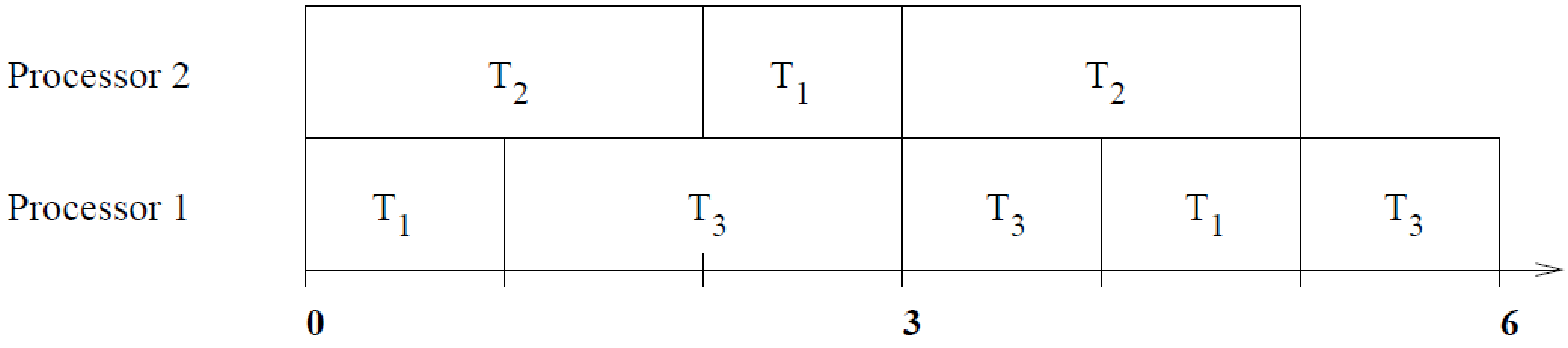
Global vs. Partitioned

- Global (work-conserving) and partitioned scheduling algorithms are incomparable:
  - There are tasksets that are schedulable with a global scheduler, but not with a partitioned scheduler, and vice versa.

Global vs Partitioned (FP) Scheduling

Task	T=D	C	Prio
T1	2	1	H
T2	3	2	M
T3	3	2	L

- A taskset schedulable with global scheduling, but not partitioned scheduling. System utilization $U = \frac{1}{2} + \frac{2}{3} + \frac{2}{3} = 1.83$
- Global FP scheduling is schedulable with priority assignment $p_1 > p_2 > p_3$ (or $p_2 > p_1 > p_3$)
- Partitioned scheduling is unschedulable, since assigning any two tasks to the same processor will cause that processor's utilization to exceed 1, so the bin-packing problem has no feasible solution



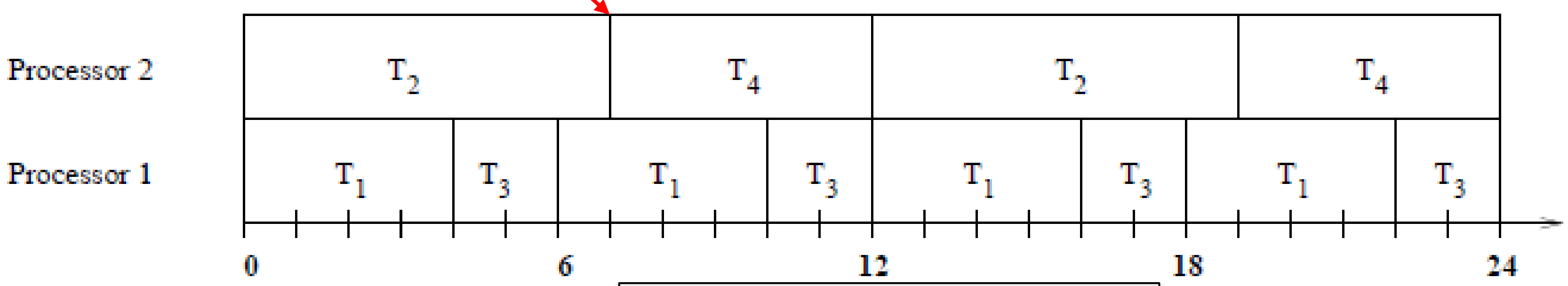
A feasible execution trace under global scheduling

Global vs Partitioned (FP) Scheduling

Task	T=D	C	Prio
T1	6	4	4(H)
T2	12	7	3
T3	12	4	2
T4	24	10	1(L)

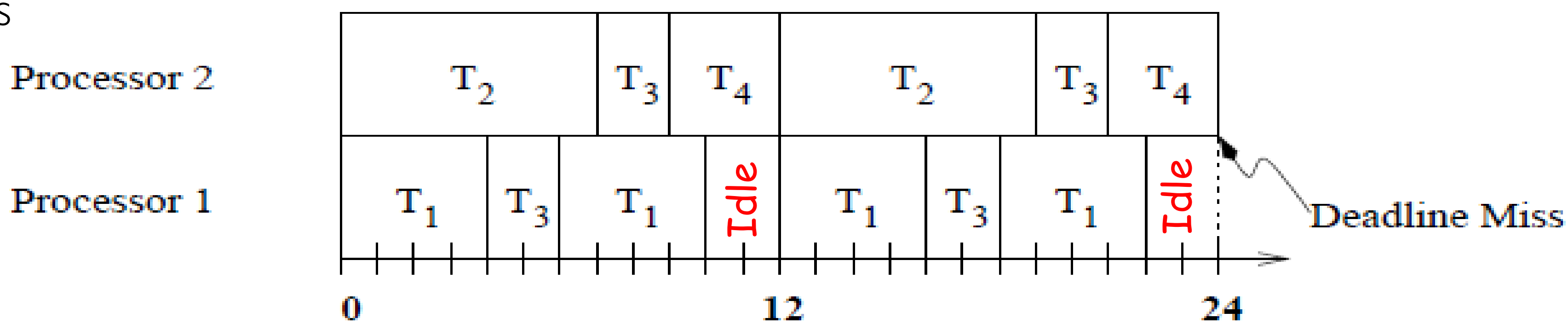
- A taskset schedulable with partitioned scheduling, but not global scheduling. System utilization $U = \frac{4}{6} + \frac{7}{12} + \frac{4}{12} + \frac{10}{24} = 2.0$, hence the two processors must be fully utilized with no possible idle intervals
- Partitioned FP scheduling with RM priority assignment ($p_1 > p_2 > p_3 > p_4$) is schedulable. T1, T3 assigned to Processor 1; T2, T4 assigned to Processor 2. Both processors have utilization 1.0, and harmonic task periods
- Global FP scheduling with RM priority assignment $p_1 > p_2 > p_3 > p_4$ is unschedulable. Compared to partitioned scheduling, the difference is at time 7, when T3 (with higher priority than T4) runs on Processor 2. This causes idle intervals on Processor 1 [10,12] and [22,24], since only one task T4 is ready during these time intervals. Since taskset $U = 2.0$ on 2 processors, any idle interval will cause the taskset to be unschedulable

At time 7, T4 runs on Processor 2



A feasible execution trace under partitioned scheduling

At time 7, T3 runs on Processor 2



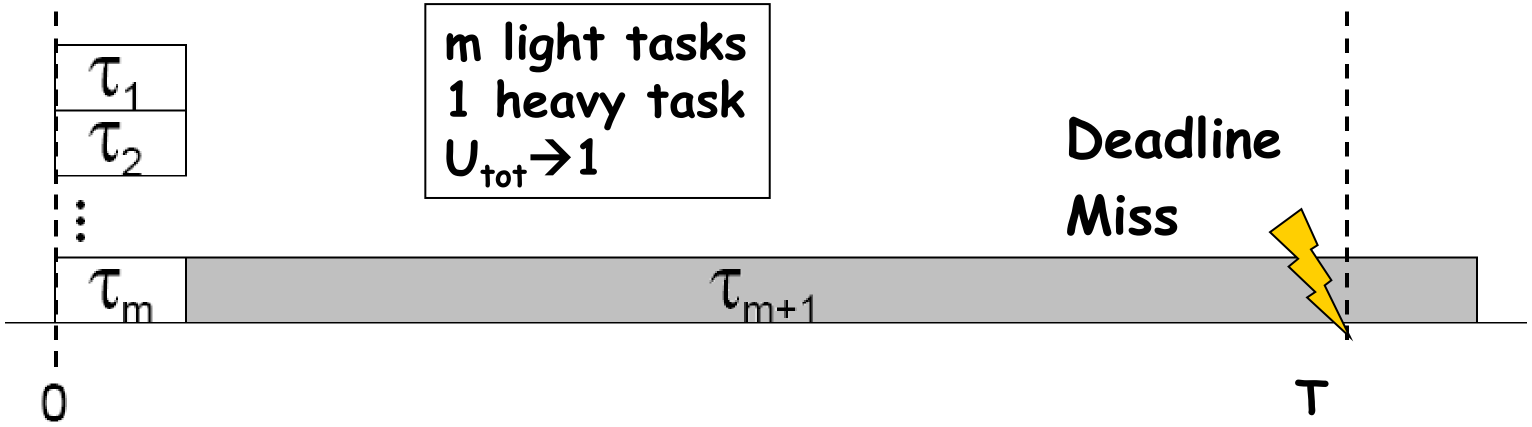
An infeasible execution trace under global scheduling

Difficulties of Global Scheduling

- Dhall's effect
  - With RM, DM and EDF, some low-utilization task sets can be unschedulable regardless of how many processors are used.
- Scheduling anomalies
  - Decreasing task execution time or increasing task period may cause deadline misses
- Hard-to-find worst-case
  - The worst-case does not always occur when a task arrives at the same time as all its higher-priority tasks
- Dependence on relative priority ordering (omitted)
  - Changing the relative priority ordering among higher-priority tasks may affect schedulability for a lower-priority task

Dhall's effect

- Global RM/DM/EDF can fail at very low utilization
- Example: m processors, n=m+1 tasks. Tasks $\tau_1, \ldots, \tau_m$ are light tasks, with small $C_i = 1$, $T_i = D_i = T - 1$; Task τ_{m+1} is a heavy task, with large $C_i = T$, $T_i = D_i = T$. $T > 1$ is some constant value
- For global RM/DM/EDF, Task τ_{m+1} has lowest priority, so $\tau_1, \ldots, \tau_m$ must run on m processors starting at time 0, causing τ_{m+1} to miss its deadline
- One solution: assign higher priority to heavy tasks
  - If heavy task τ_{m+1} is assigned the highest priority, then it runs from time 0 to T and meets its deadline; The light tasks can run on other processors and meet their deadlines as well



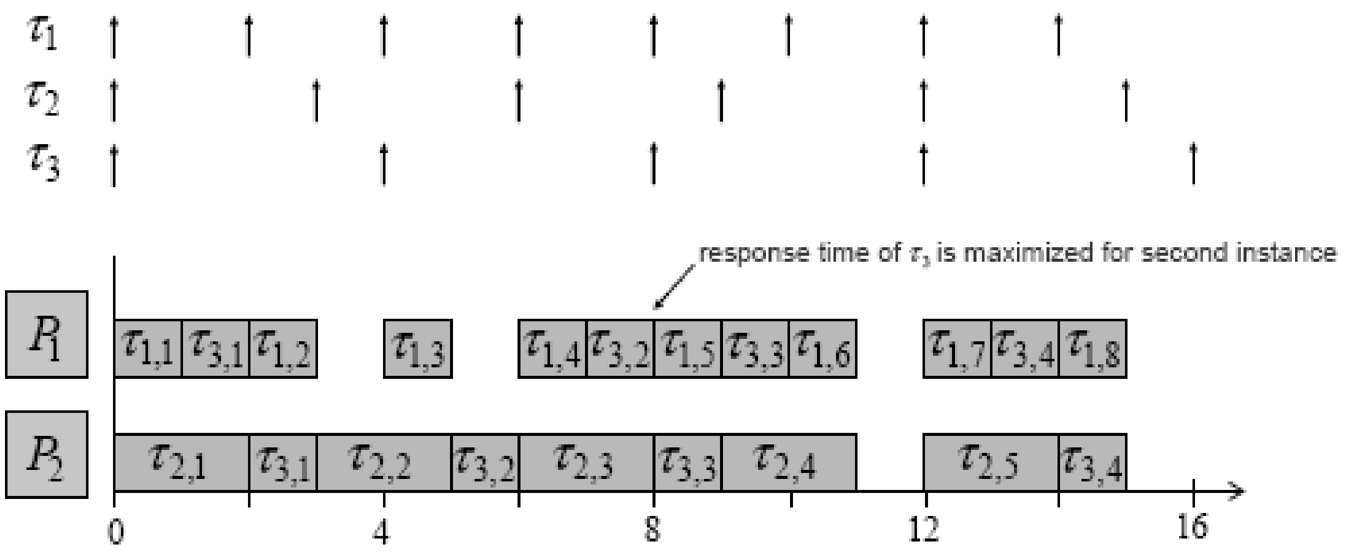
Hard-to-Find Worst-Case

- For uniprocessor scheduling, the worst case occurs when all tasks are initially released at time 0 simultaneously, called the critical instant (recall Slide Response Time Analysis (RTA))
- This is no longer true for multiprocessor scheduling, as the worst-case interference for a task does not always occur at time 0, when all tasks are initially released at time 0 simultaneously
  - Response time for task τ_3 is maximized for its 2nd job $\tau_{3,2}$ (8-4=4), which does not arrive at the same time as its higher priority tasks; not for its 1st job $\tau_{3,1}$(3-0=3), which arrives at the same time as its higher priority tasks

Hard-to-find critical instant:

(RM scheduling)

$$\tau_1 = \{ C_1 = 1, T_1 = 2 \}$$
$$\tau_2 = \{ C_2 = 2, T_2 = 3 \}$$
$$\tau_3 = \{ C_3 = 2, T_3 = 4 \}$$



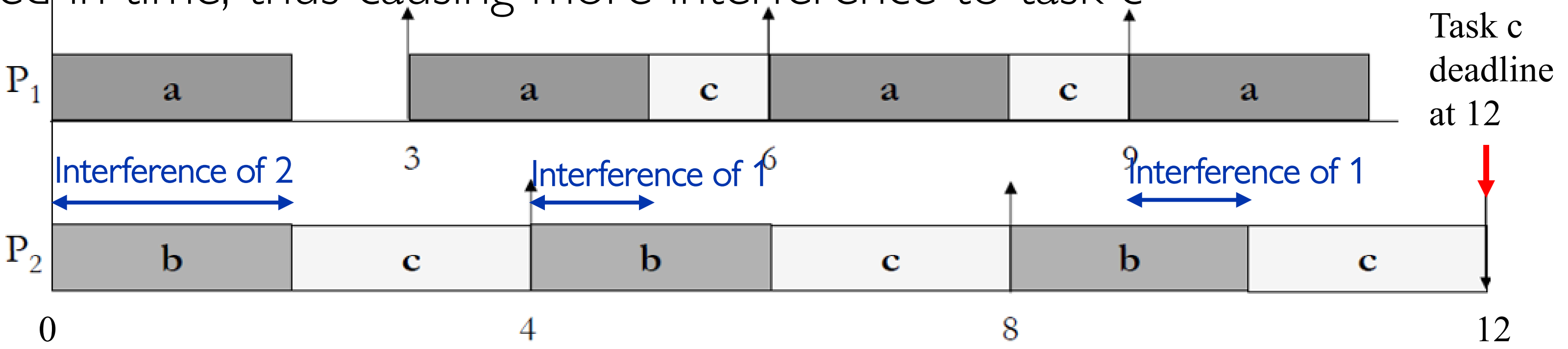
MP Scheduling Anomalies

- Decrease in processor demand (decreasing task execution time or increasing task period) may cause deadline misses!
- Anomaly 1
  - Decrease in processor demand from higher-priority tasks can *increase the interference on a lower-priority task because of* change in the time when the tasks execute
- Anomaly 2
  - Decrease in processor demand of a task *negatively affects the task* itself because change in the task arrival times cause it to suffer more interference

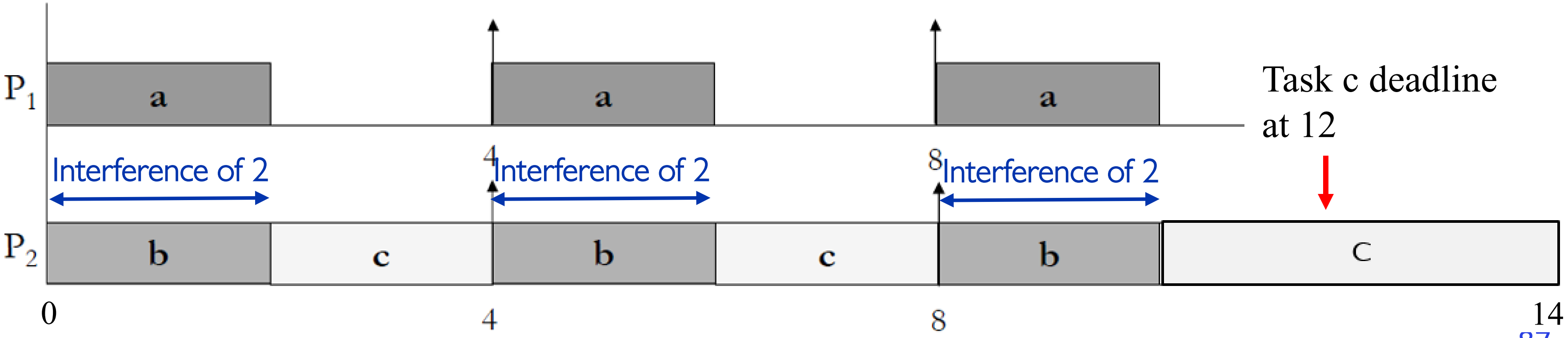
Scheduling Anomaly Example 1

- Three tasks on two processors under global scheduling
- With Task a's period $T_a = 3$, system utilization $\sum U_i = 1.83$. WCRT of task c is $R_c = 12 \le D_c = 12$. $R_c = C_c + I_c = 8 + I_c$, where $I_c = 2 + 1 + 1 = 4$ is interference by higher priority tasks a and b. (Task c experiences inference when both processors are busy executing higher priority tasks a and b.) Task c is schedulable but saturated, as any increase in its WCET or interference would make it unschedulable.
- With Task a's period $T_a = 4$, system utilization $\sum U_i = 1.67$ is reduced. But WCRT of task c increases: $R_c = 14 > D_c = 12$. $R_c = 8 + I_c$ where $I_c = 2 + 2 + 2 = 6$, since execution segments of tasks a and b on two processors are aligned in time, thus causing more interference to task c

Task	T=D	C	Util	Prio
a	3	2	0.67	H
b	4	2	0.5	M
c	12	8	0.67	L



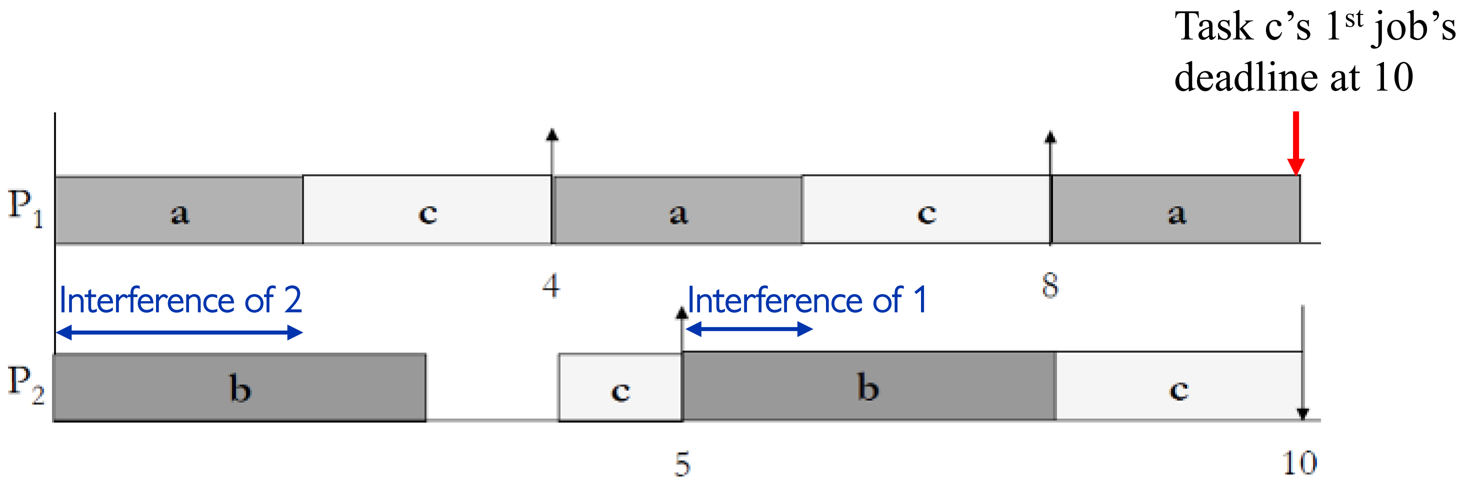
Task	T=D	C	Util	Prio
a	4	2	0.5	H
b	4	2	0.5	M
c	12	8	0.67	L



Scheduling Anomaly Example 2

- Three tasks on two processors under global scheduling
- With Task c's period $T_c = 10$, system utilization $\sum U_i = 1.8$. WCRT of task c is $R_c = 10 \leq D_c = 10$. $R_c = C_c + I_c = 7 + 3 = 10$, where $I_c = 2 + 1 = 3$ is interference by higher priority tasks a and b. Its 1st job meets its deadline at time 10. This schedule repeats in future periods, hence task c is schedulable but saturated, as any increase in its WCET or interference would make it unschedulable.

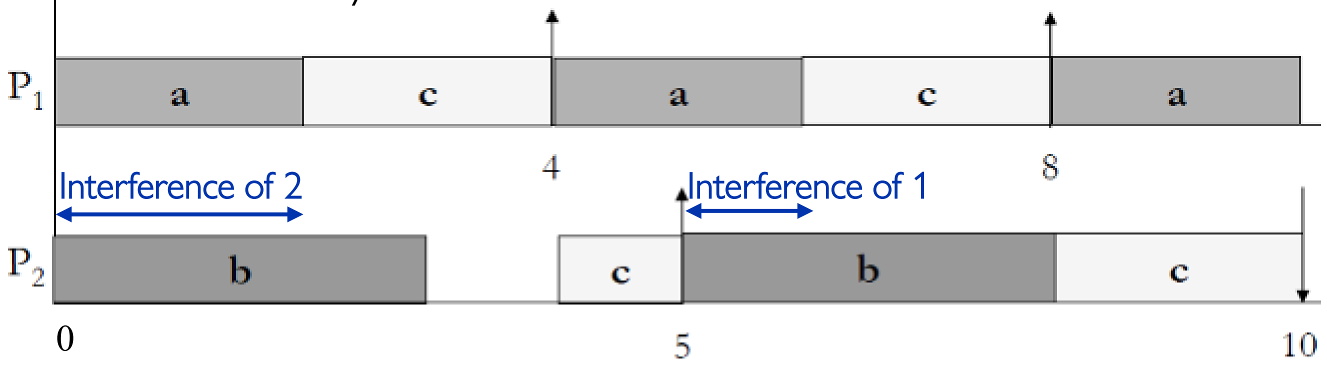
Task	T=D	C	Util	Prio
a	4	2	0.5	H
b	5	3	0.6	M
c	10	7	0.7	L

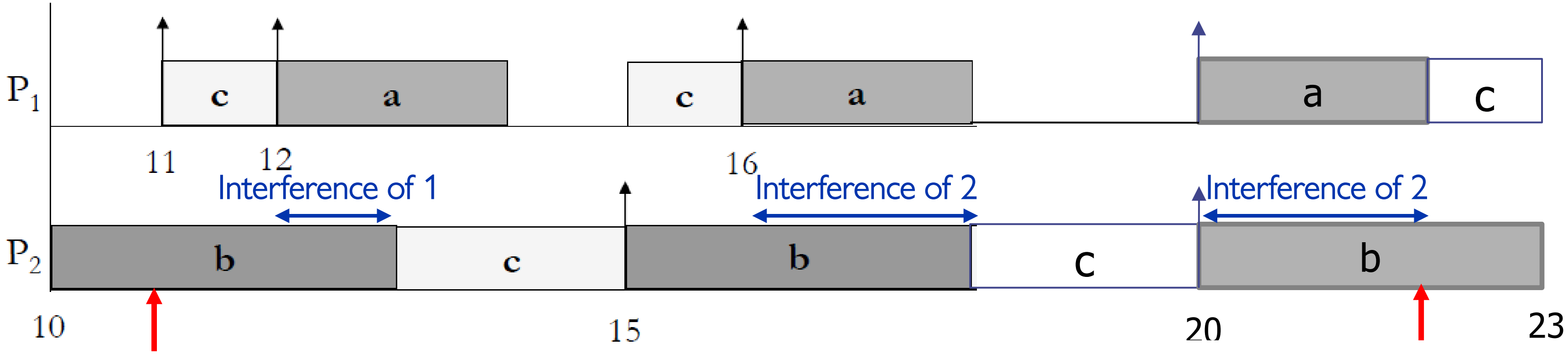


Scheduling Anomaly Example 2

- With Task c's period $T_c = 11$, system utilization $\sum U_i = 1.74$ is reduced. WCRT of task c is $R_c = 12 > D_c = 10$. Its 1st job has response time $C_c + I_c = 7 + 3 = 10 \leq D_c = 11$, where $I_c = 2 + 1 = 3$, but this is not task c's WCRT.
- Its 2nd job has response time $C_c + I_c = 7 + 5 = 12 > D_c = 11$, where $I_c = 1 + 2 + 2 = 5$. The 2nd job finishes at time 11+12=23, and misses its deadline at time 22.
- Another example where the worst-case interference for task c does NOT occur at time 0, when all tasks are initially released at time 0 simultaneously

Task	T=D	C	Util	Prio
a	4	2	0.5	H
b	5	3	0.6	M
c	11	7	0.64	L





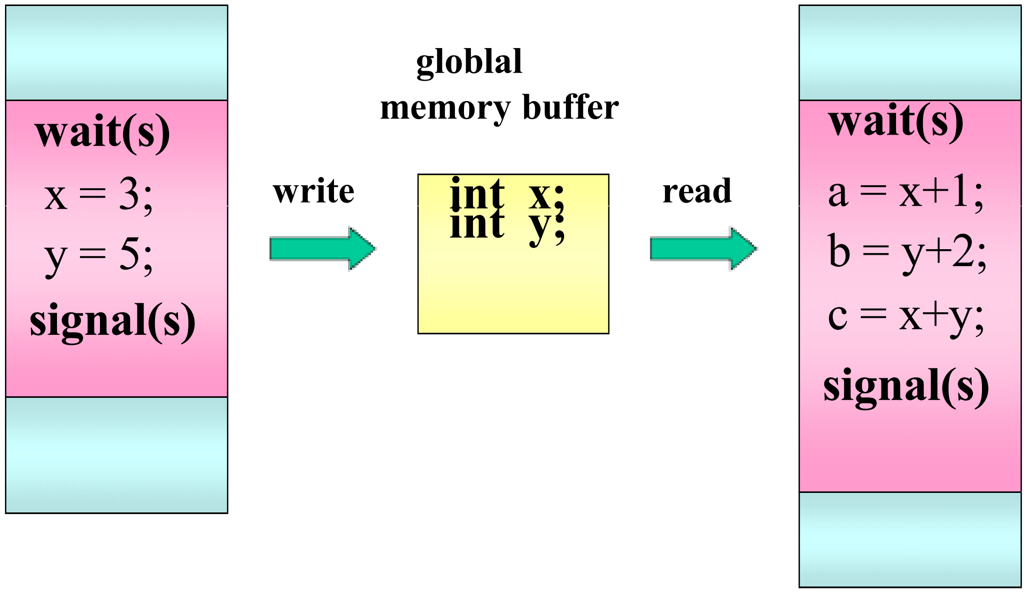
Task c's 1st job's deadline at 11

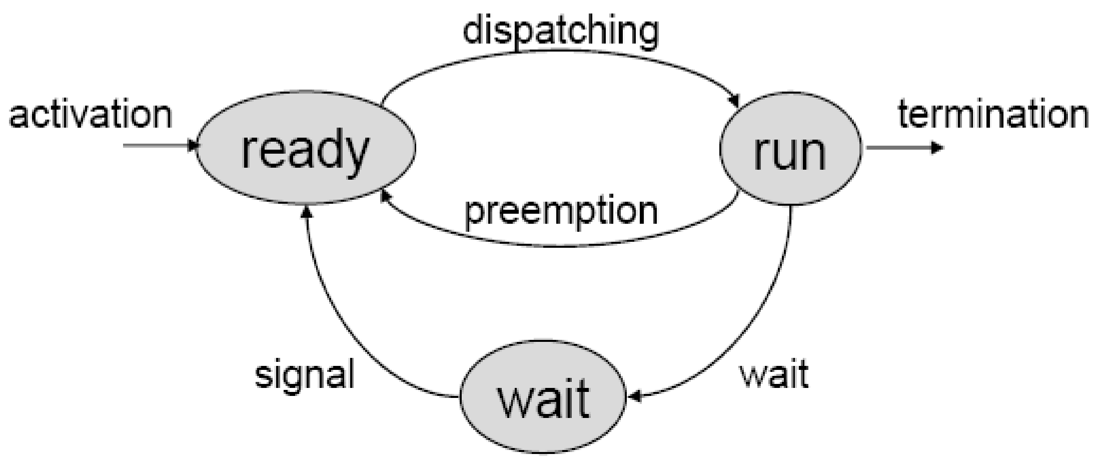
Task c's 2nd job's deadline at 22

Resource Synchronization Protocols (for Fixed-Priority Scheduling)

Resource Sharing

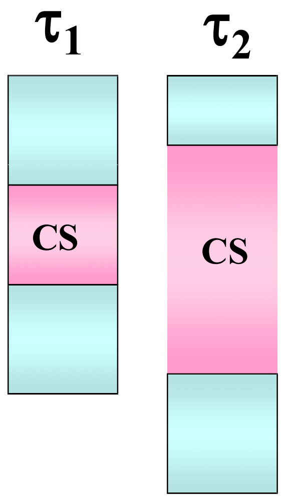
- Some shared resources do not allow simultaneous accesses but require mutual exclusion. A piece of code executed under mutual exclusion constraints is called a critical section.
- When two tasks access shared resource, mutexes or binary semaphores are used to protect critical section
- Each shared resource R_i must be protected by a semaphore S_i, and each critical section (CS) using resource R_i must begin with wait(S_i) and end with signal(S_i)
- A task waiting for an exclusive resource is said to be blocked on that resource. Otherwise, it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the associated resource becomes free.
- Tasks blocked on the same resource are kept in a queue. When a running task invokes wait(S_i) when S_i is locked, it enters a waiting state, until another task invokes signal(S_i) to unlock S_i

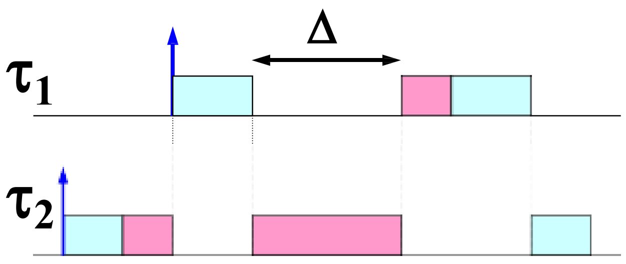




Blocking Time

- Lower-priority tasks can cause delay to higher-priority tasks due to blocking time
- If higher priority task τ_1 tries to lock a semaphore that is locked by lower priority task τ_2, τ_1 blocks until τ_2 unlocks the semaphore, and τ_1 experiences a blocking delay Δ.
  - Since typical Critical Sections are very short, it seems this blocking time delay Δ Is bounded by the longest critical section in lower-priority tasks, hence acceptable?
- No, blocking delay may be unbounded!





Example Taskset

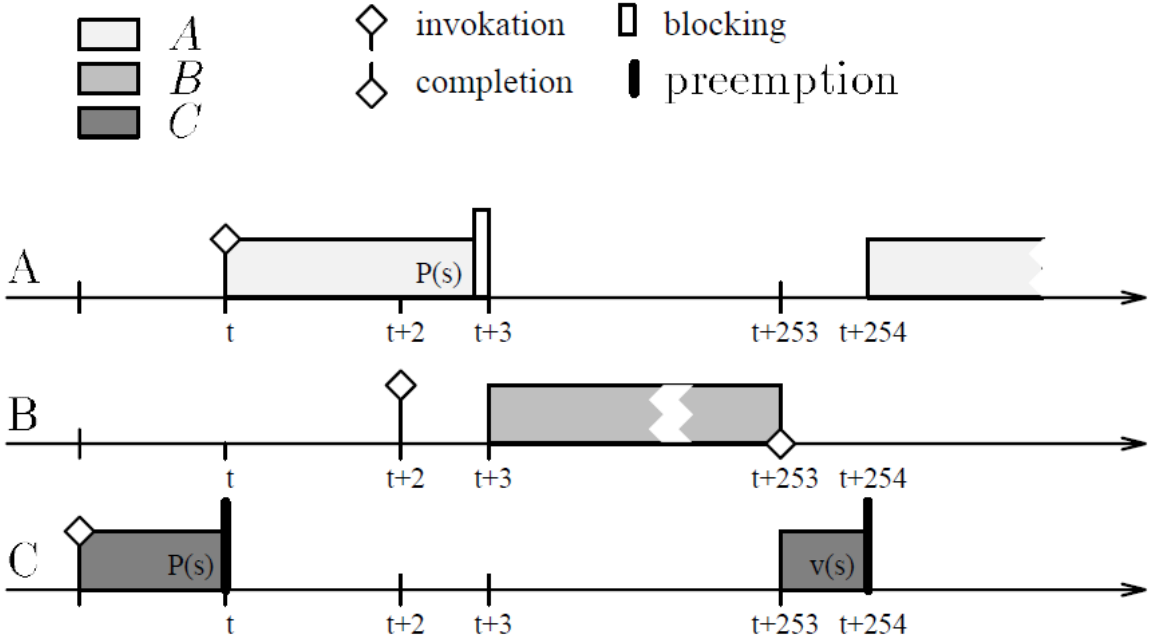
- system utilization $U = \frac{5}{50} + \frac{250}{500} + \frac{1000}{3000} = 0.93 > 0.780$
  – Since utilization exceeds the Utilization Bound of 0.780 of 3 tasks under RM scheduling, we cannot determine schdulability by the Utilization Bound test
- RTA shows that the taskset is schedulable by computing WCRT of each task (without shared resources):
  – $R_A = C_A + 0 = 5 + 0 = 5$
  – $R_B = C_B + \left\lceil \frac{R_B}{T_A} \right\rceil \cdot C_A = 250 + \left\lceil \frac{R_B}{50} \right\rceil \cdot 5 = 280$
  – $R_C = C_C + \left\lceil \frac{R_C}{T_A} \right\rceil \cdot C_A + \left\lceil \frac{R_C}{T_B} \right\rceil \cdot C_B = 1000 + \left\lceil \frac{R_C}{50} \right\rceil \cdot 5 + \left\lceil \frac{R_C}{500} \right\rceil \cdot 250 = 2500$

Task	T	D	C	Prio
A	50	10	5	H
B	500	500	250	M
C	3000	3000	1000	L

Priority Inversion

Task	T	D	C	Prio
A	50	10	5	H
B	500	500	250	M
C	3000	3000	1000	L

- HP: High-Priority; MP: Medium-Priority; LP: Low-Priority
- t: LP C locks s
- t+Δ: HP task A preempts LP task C
- t+2: MP task B is invoked, but cannot start running due to HP task A running
- t+3: HP task A tries to lock s, blocks since LP task C is holding s; MP task B starts running
- t+253: MP task B finishes; LP task C starts running
- t+254: LP task C unlocks s; HP last A starts running, but it already missed its deadline long ago!



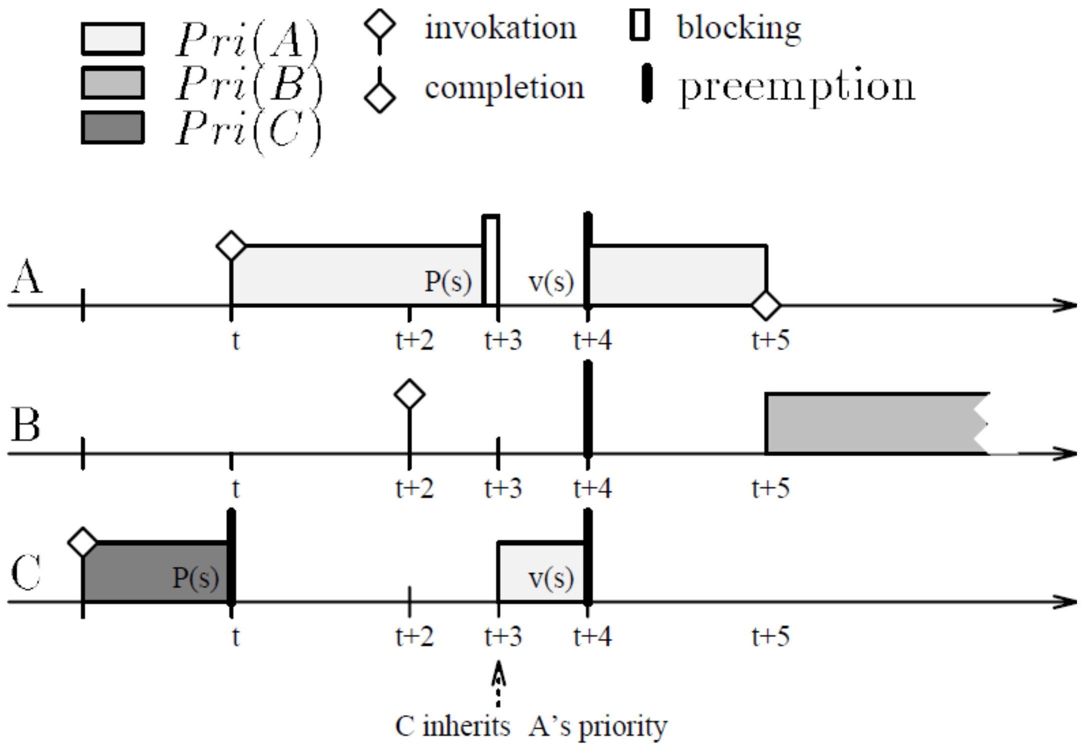
Priority Inversion and Priority Inheritance

- Priority inversion happened
  - High priority task (A) is blocked by low-priority task (B) for an unbounded interval of time.
    - More than the longest critical section of B
- In 1997, this bug caused the Mars pathfinder to freeze up occasionally without explanation, and then starts working again
- Fixed by uploading a software patch enabling Priority-Inheritance Protocol (PIP)
  - When a task locks a semaphore, it inherits the highest priority of all tasks blocked waiting for the semaphore
- A task in a CS increases its priority if it blocks other higher priority tasks, by inheriting the highest priority among those tasks it blocks.
  - $P_{CS} = \max\{P_k | \tau_k \text{ blocked on CS}\}$

With PIP

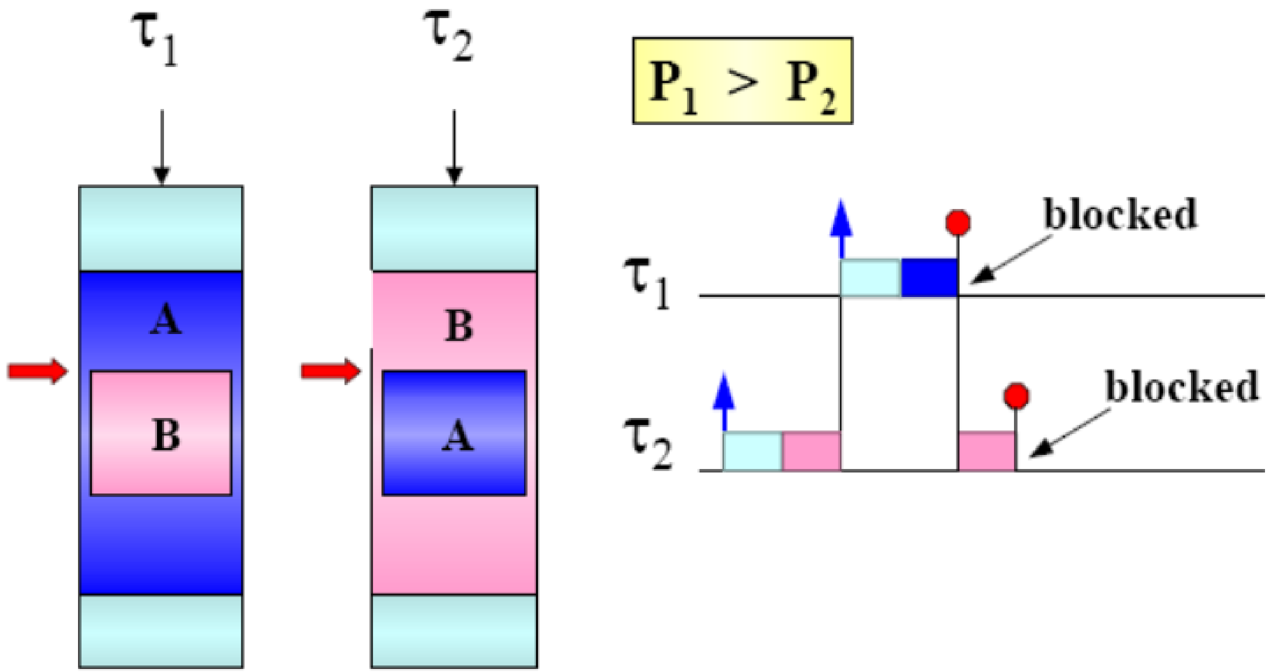
Task	T	D	C	Prio
A	50	10	5	H
B	500	500	250	M
C	3000	3000	1000	L

- t: LP C locks s
- t+Δ: HP task A preempts LP task C
- t+2: MP task B is invoked, but cannot start running due to HP task A running
- t+3: HP task A tries to lock s, blocks since LP task C is holding s; C inherits A's priority and starts running
  - MP task B cannot start running, hence cannot cause unbounded blocking to HP task A
- t+4: LP task C unlocks s, and returns to its regular Low priority; HP task A locks s and starts running
- t+5: HP task A finishes and meets its deadline.



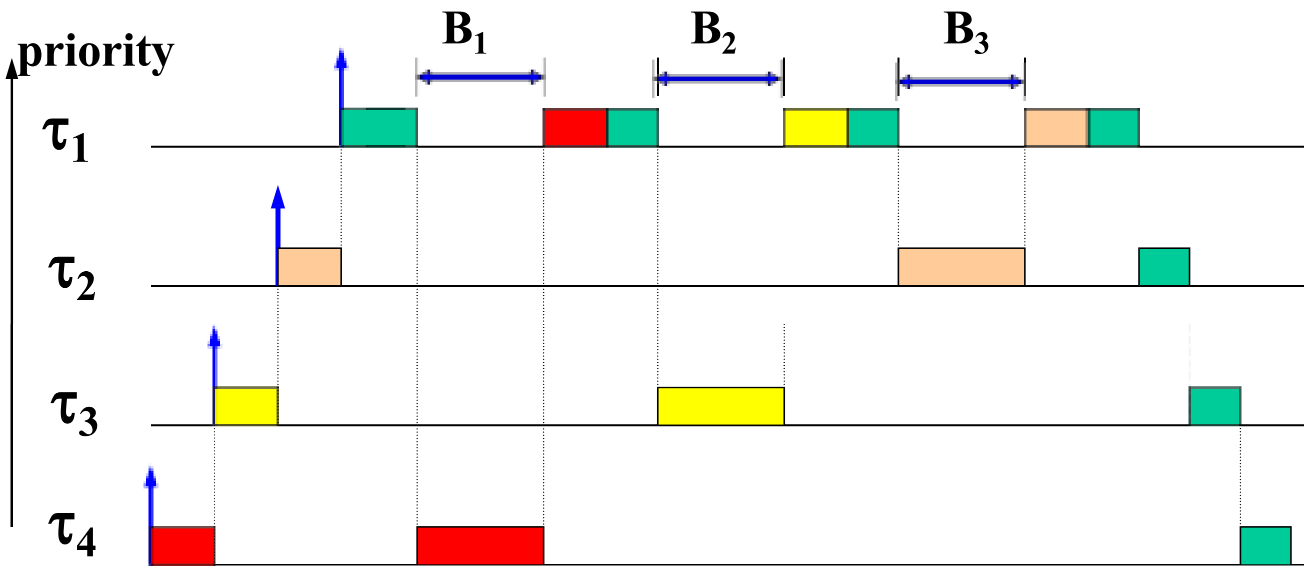
PIP Properties

- PIP does not prevent deadlocks. Classic deadlock scenario:
  - – LP task τ_2 locks s_1 at time t
  - – HP task τ_1 starts running after t and locks s_2, and tries to lock s_1, blocked by τ_2
  - – τ_2 inherits τ_1's priority and starts running
  - – τ_2 tries to lock s_2, but τ_1 holds s_2. Deadlocked!



PIP Properties

- Chained blocking:
  - Theorem: task τ_i can be blocked at most once by each lower priority task
    - If **n** is the number of tasks with priority less than τ_i, and **m** is the number of semaphores on which τ_i can be blocked, then
  - **Theorem:** τ_i can be blocked at most for the duration of **min(n, m)** critical sections
- Priority Ceiling Protocol is a more advanced protocol, which prevents deadlocks and reduces blocking time



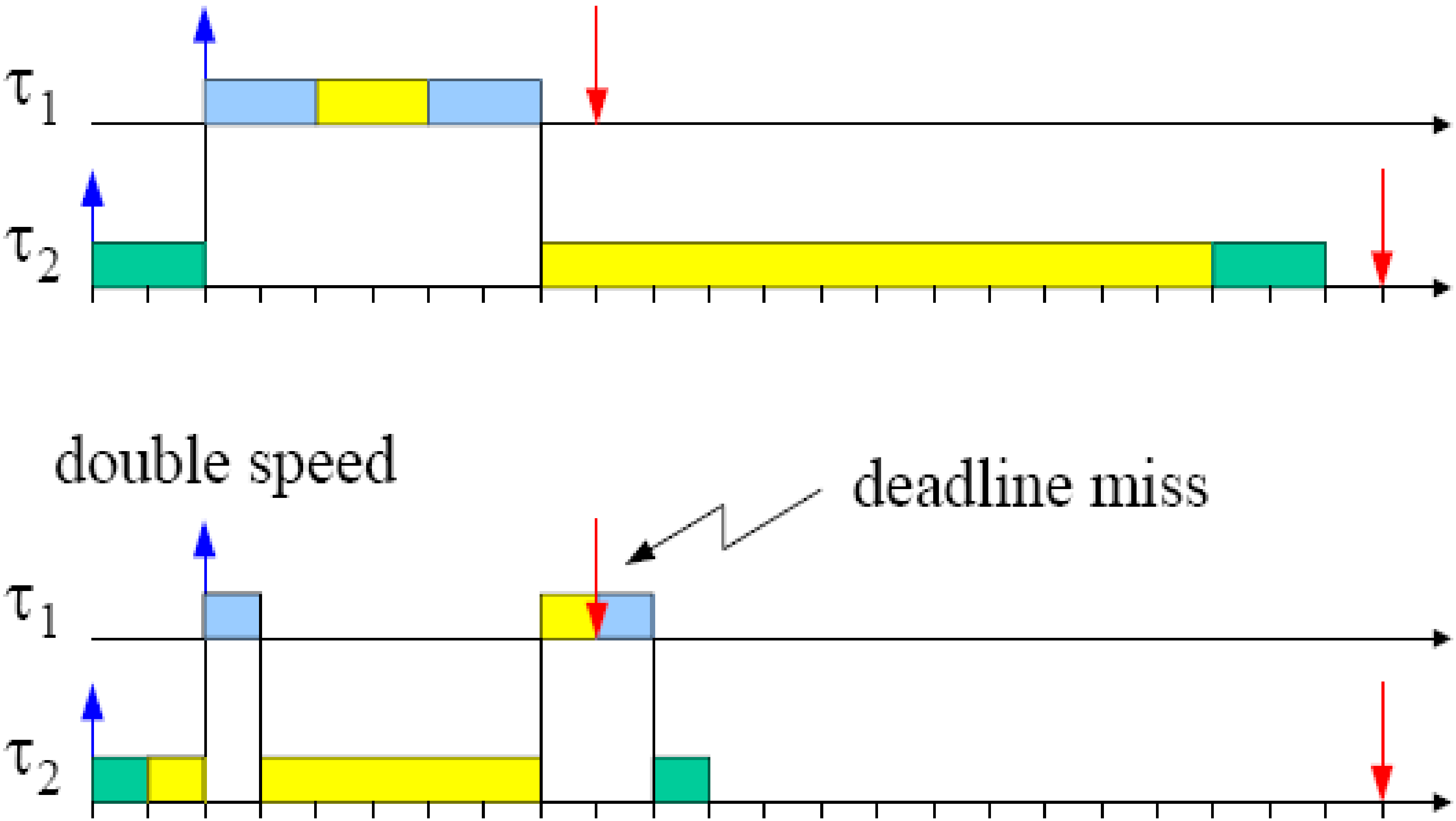
Schedulability Analysis

- Schedulable utilization bound for RM scheduling with blocking time:
  - A taskset is schedulable under RM scheduling with blocking time if
  - $\forall i,$ priority level i utilization $U_i = \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$

- Response Time Analysis (RTA) for RM scheduling with blocking time:
  - WCRT R_i is computed by solving the following recursive equation:
  - $R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$
  - where B_i is the maximum blocking time experienced by task τ_i due to shared resources

Scheduling Anomaly w/ Resource Synchronization

- Doubling processor speed causes T1 to miss its deadline
  - (Yellow part denotes a critical section shared by T1 and T2)



Online Resources

- Priority-Driven Scheduling, Marilyn Wolf
  - https://www.youtube.com/watch?v=zSgr_oFmjqI&list=PLzwefUCNStZsmz5fWPVwVvTo1iPeGmG9M&index=4
- RMS and EDF, Marilyn Wolf
  - https://www.youtube.com/watch?v=oHMC2aO8GII&list=PLzwefUCNStZsmz5fWPVwVvTo1iPeGmG9M&index=5
- Real-Time Scheduling Models, Marilyn Wolf (long)
  - https://www.youtube.com/watch?v=WIoSQ7ZEKXk