CSC 112: Computer Operating Systems
Lecture 6

Real-Time Scheduling

Department of Computer Science,
Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming

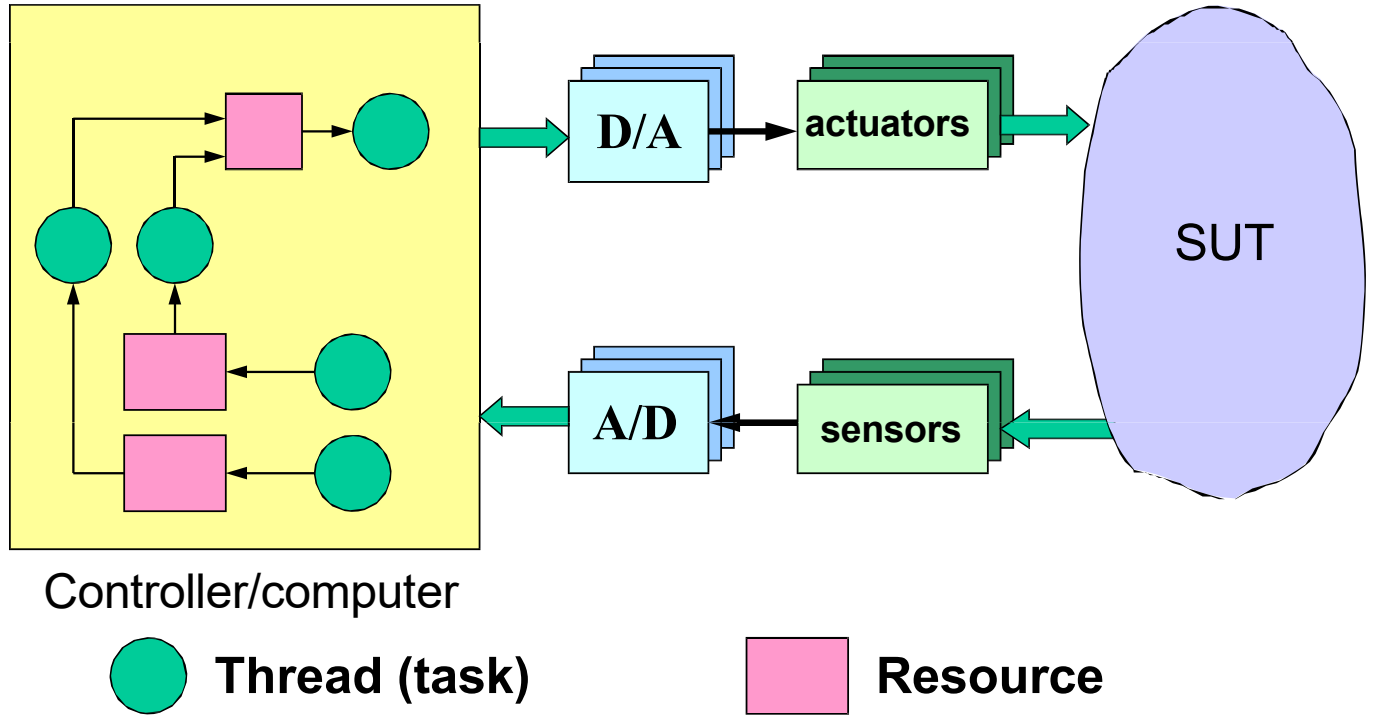
Outline

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

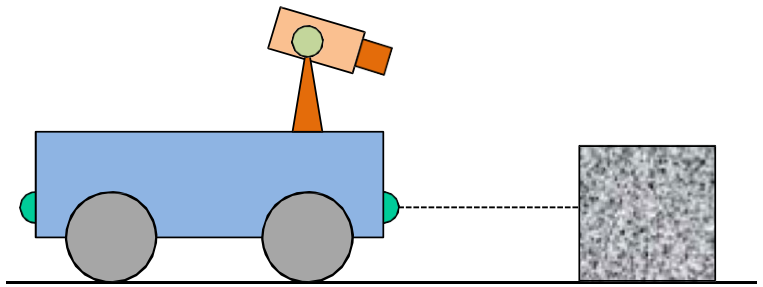
- The tight interaction with the environment requires the system to react to events within precise timing constraints
- Timing constraints are imposed by the dynamics of the environment

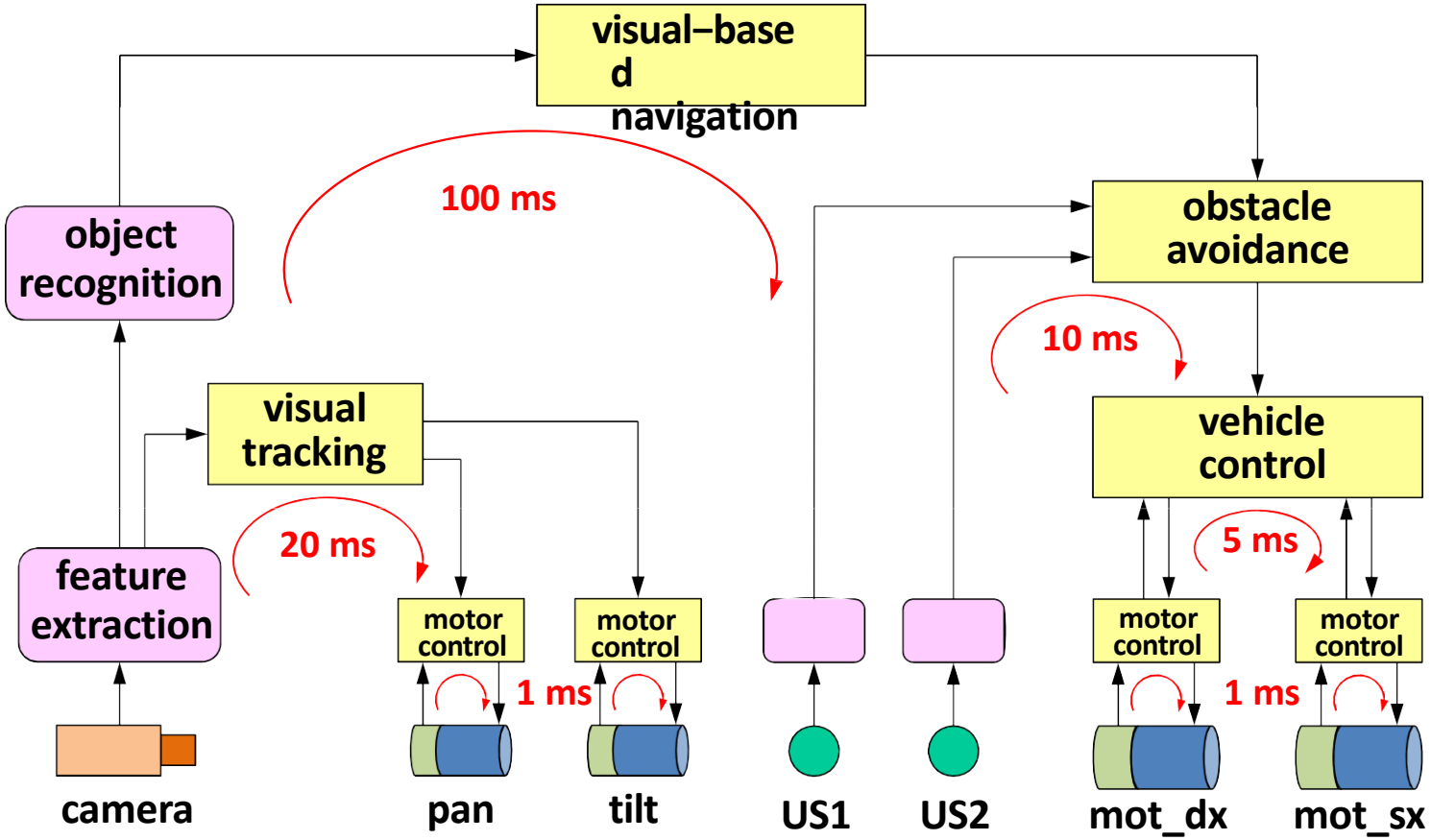
- The real-time operating system (RTOS) must be able to execute tasks within timing constraints

A Robot Control Example

- Consider a robot equipped with:
  - two actuated wheels
  - two proximity (US) sensors
  - a mobile (pan/tilt) camera
  - a wireless transceiver

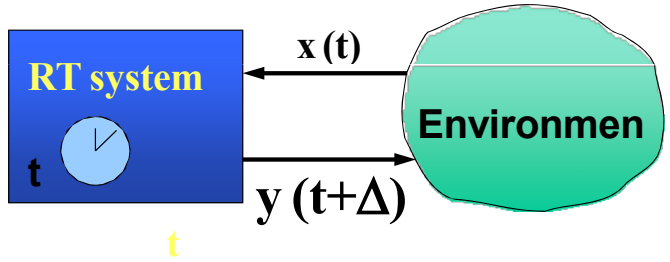
- Goal:
  - follow a path based on visual feedback
  - avoid obstacles

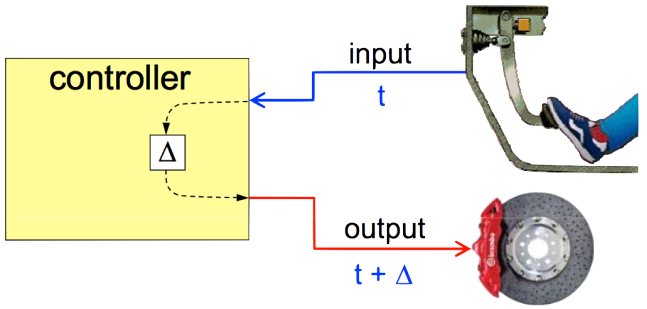




Real-Time Systems

- 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 requirements, e..g.,
  - There was a person who drowned in a river with average depth of 6 inches





RTOS Requirements

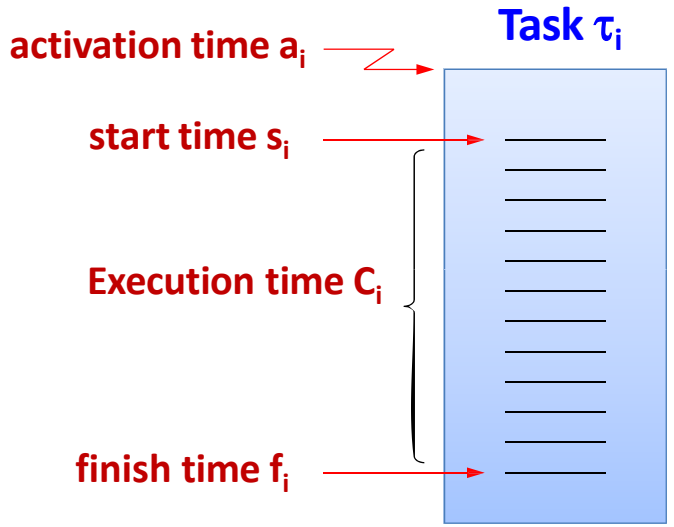
- Timeliness: results must be correct not only in their value but also in the time domain
  - provide kernel mechanism for time management and for handling tasks with explicit timing constraints and different criticality
- Predictability: system must be analyzable to predict the consequences of any scheduling 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

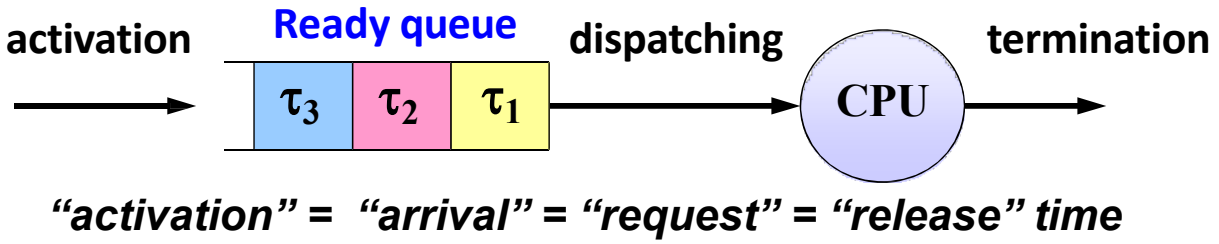
Sources of Nondeterminism

- 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

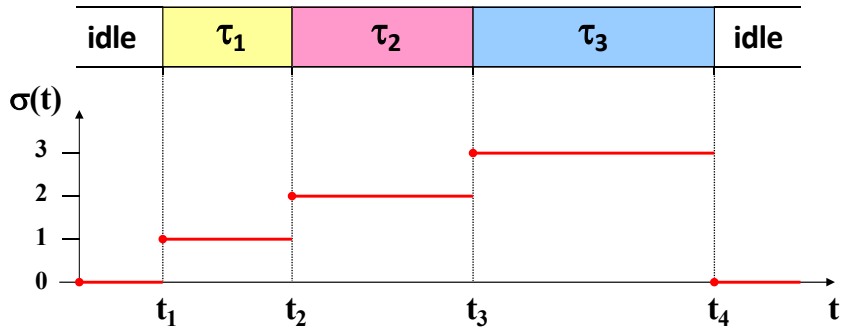
- The concept of concurrent tasks reflects the intuition about the functionality of embedded systems.
  - Task here can refer to either process or thread, depending on the underlying RTOS support
- Tasks help us manage timing complexity:
  - multiple execution rates
    - » multimedia
    - » automotive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





Schedule

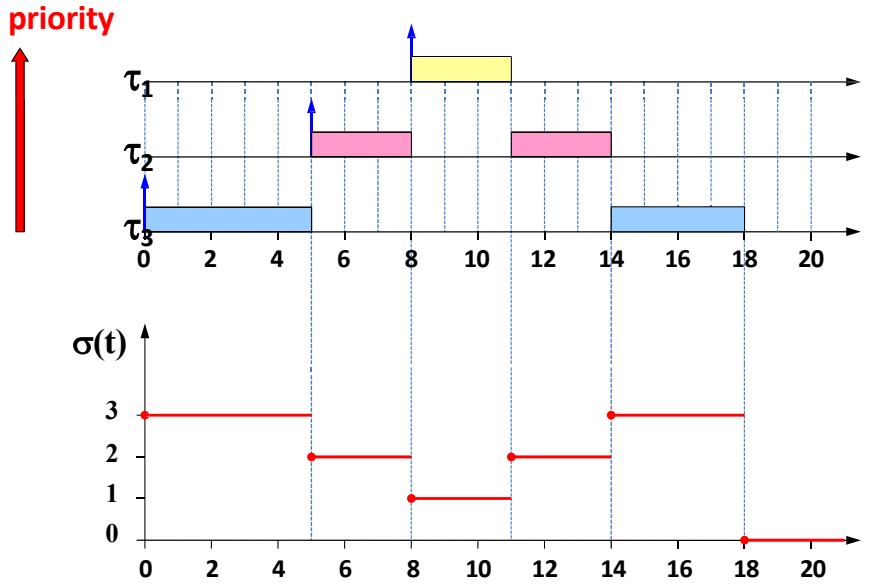
- A specific assignment of tasks to the processor that determines the task execution sequence. Formally:
- Given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a schedule is a function $\sigma: R^+ \rightarrow N$ that associates an integer k to each time slice $[t_i, t_{i+1})$ with the meaning:
 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



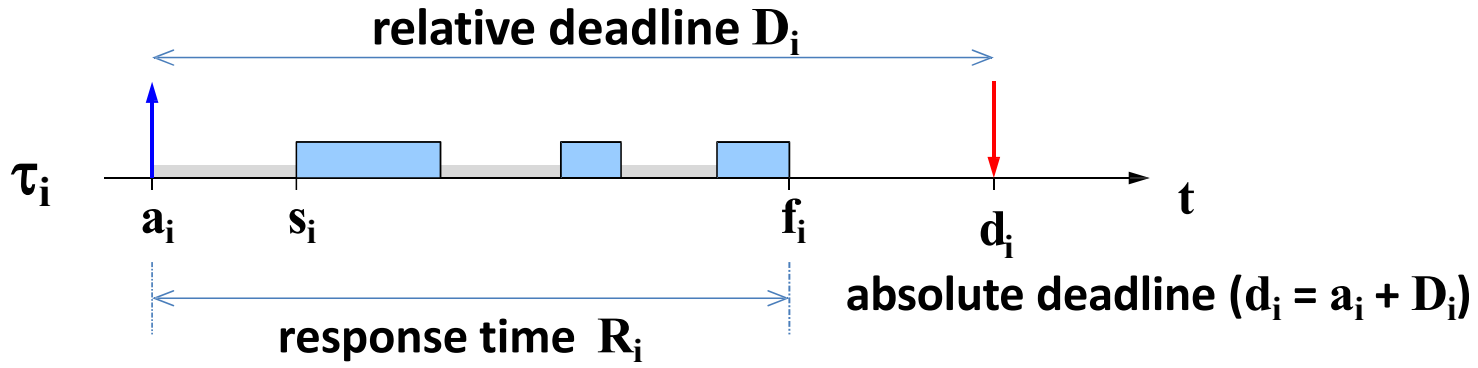
Preemptive scheduling example

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

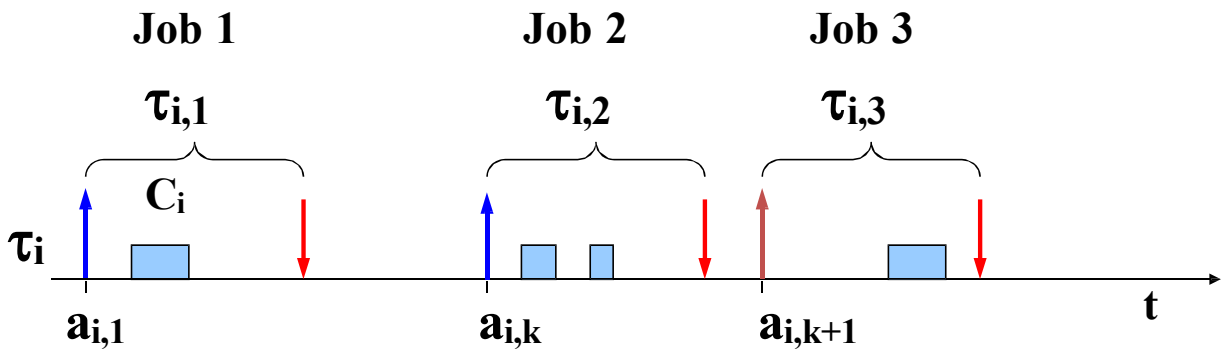
Real-Time Task

- 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
- 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 $R_i \leq D_i$



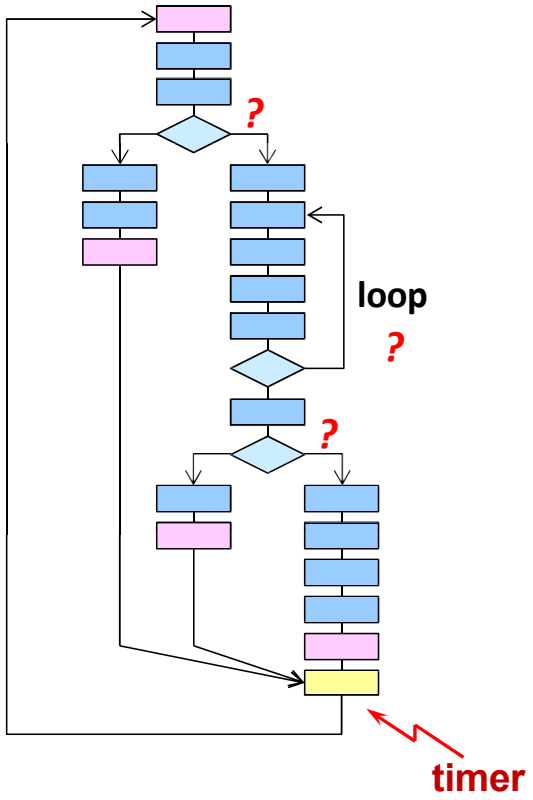
Tasks and Jobs

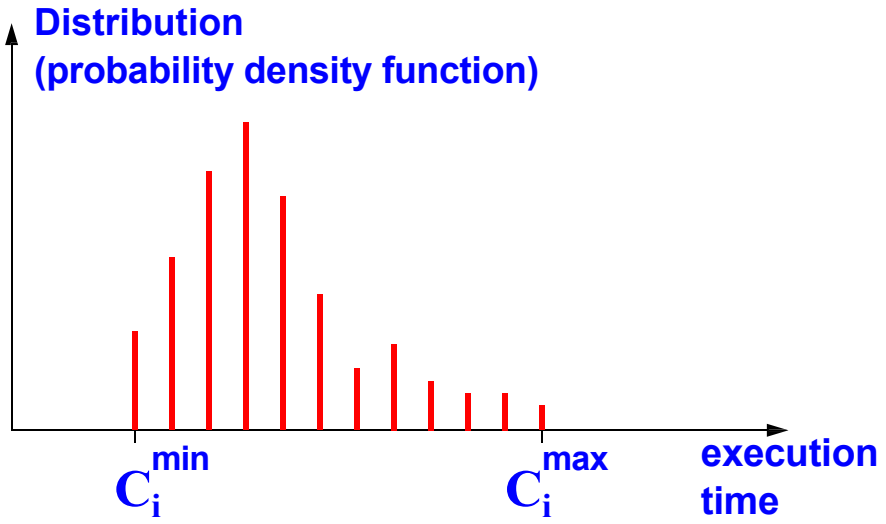
- A task running several times on different input data generates a sequence of instances (jobs)
  - Upwards arrow: task arrival or release times; downwards arrow: task deadlines
- Activation mode:
  - Periodic tasks: the task is activated by the operating system at predefined time intervals
  - Aperiodic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



Estimating WCET is Not Easy

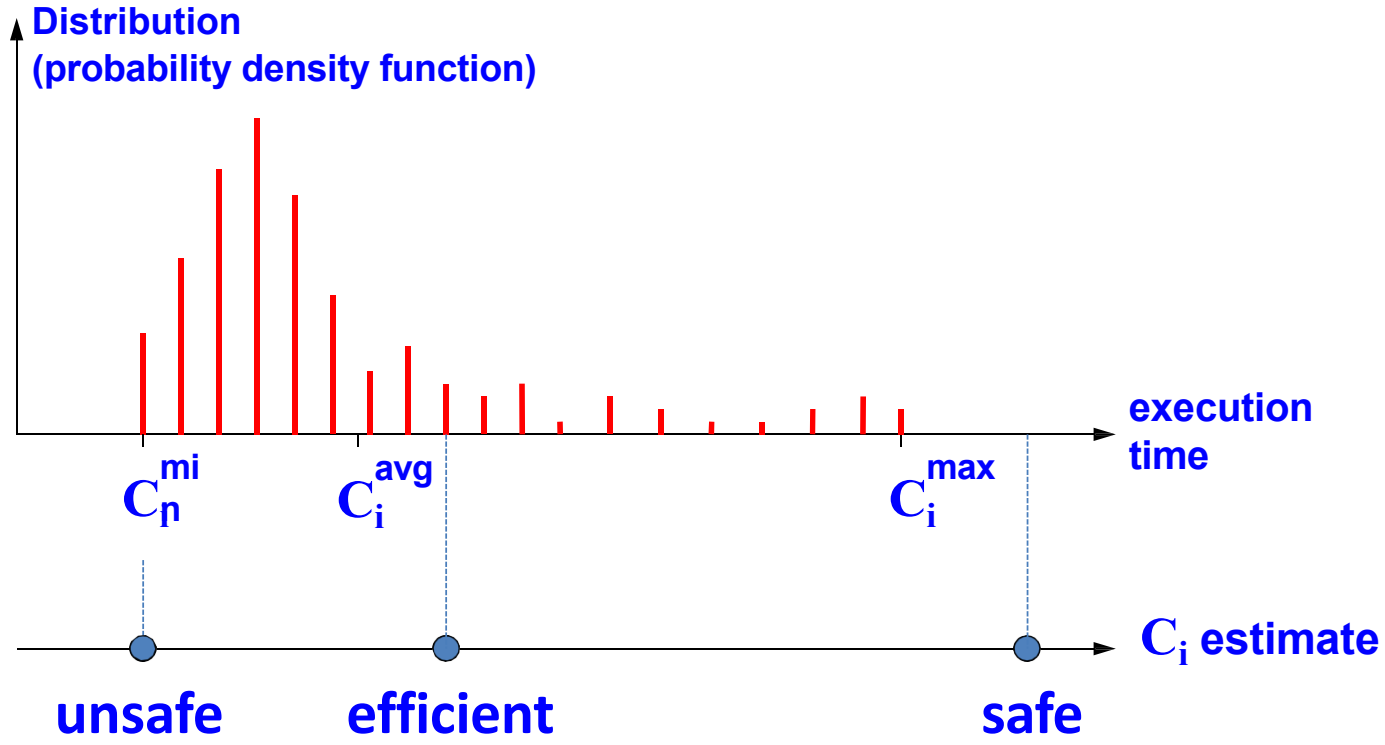
- Each job operates on different data and can take different paths.
- Even for the same data, computation time depends on the processor state (cache, prefetch queue, number of preemptions).
- We use C_i to denote C_i^{max} Worst-Case Execution Time (WCET) in this lecture, and assume it is given as part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





Predictability/Safety vs. Efficiency

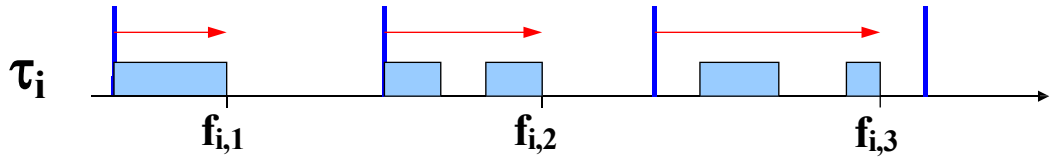
- Tradeoff between safety and efficiency in estimating the WCET C_i
  - Setting a large C_i achieves high predictability and safety, since it is unlikely to be exceeded at runtime; but it hurts efficiency, since the system needs to reserve more CPU time for the task. Suitable for hard real-time tasks.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



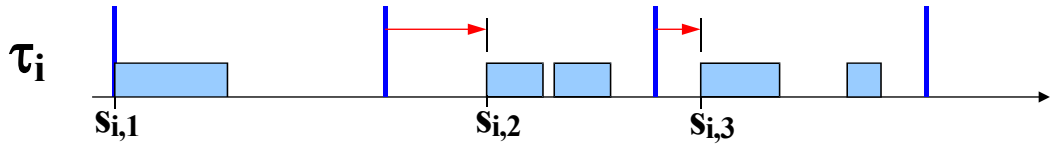
Jitter

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

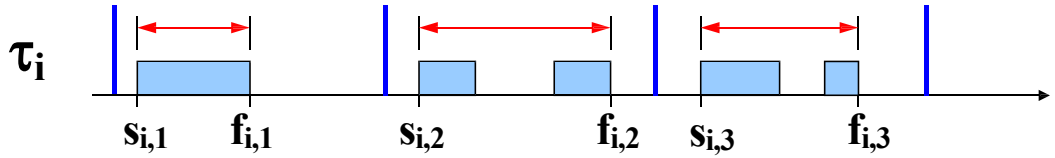
Finish-time Jitter



Start-time Jitter

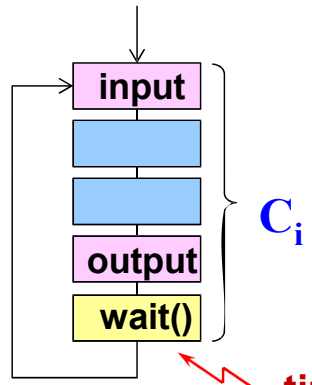


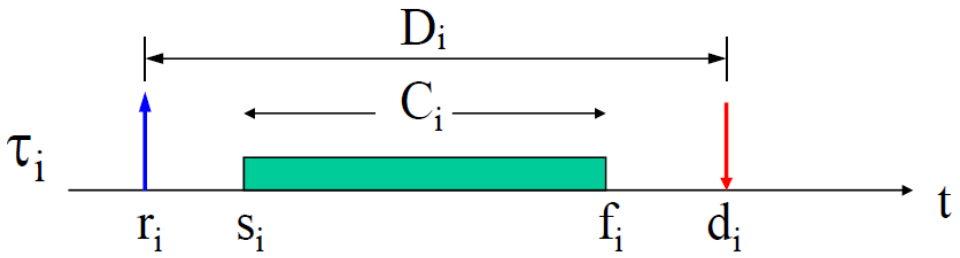
Completion-time Jitter (I/O Jitter)



Periodic Task

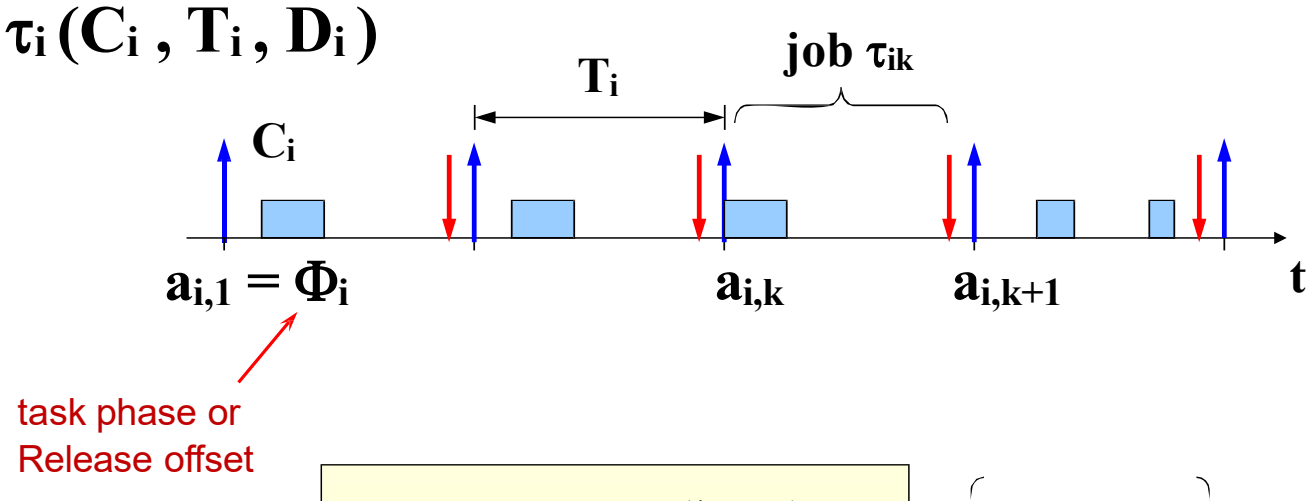
- A periodic task τ_i has a tuple of 3 attributes (C_i, T_i, D_i):
  - Worst-Case Execution Time (WCET): C_i; Period T_i; Relative Deadline D_i
- It generates an infinite sequence of jobs in every period: $\tau_{i,1}, \tau_{i,1}, \ldots, \tau_{i,k}, \ldots$





$\mathbf{r_i}$	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 $D_i = T_i$

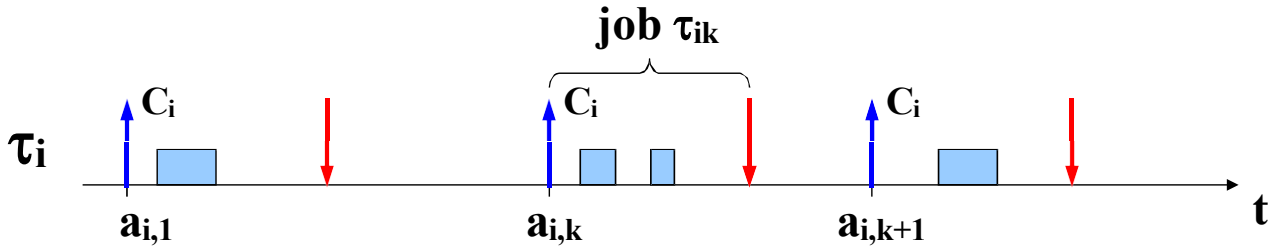
Multiple jobs released by task τ_i

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



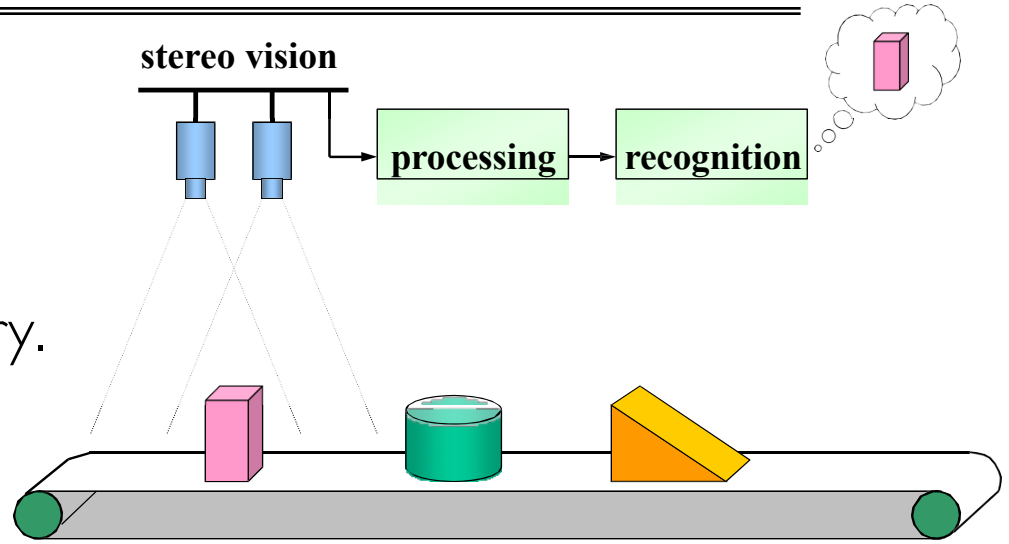
Types of Constraints

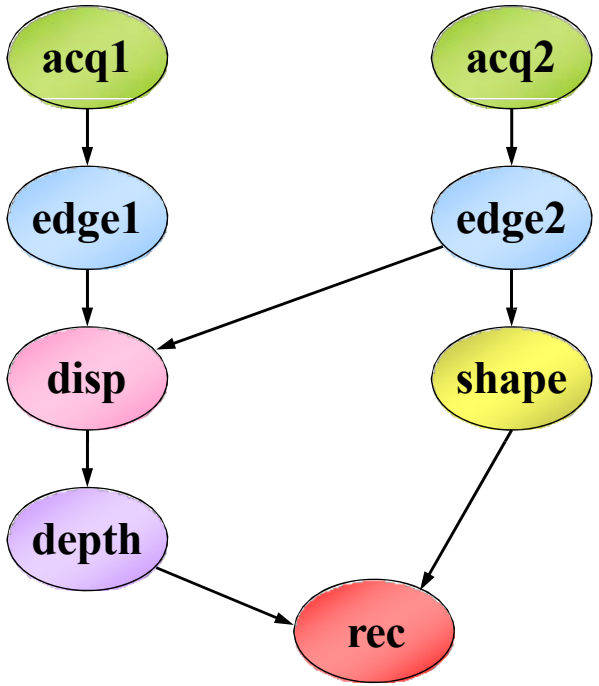
- Timing constraints
  - Deadline, jitter.
- Precedence constraints
  - Relative ordering among task executions.
- Resource constraints
  - Synchronization when accessing mutually-exclusive resources (shared data).

21

Precedence Constraints

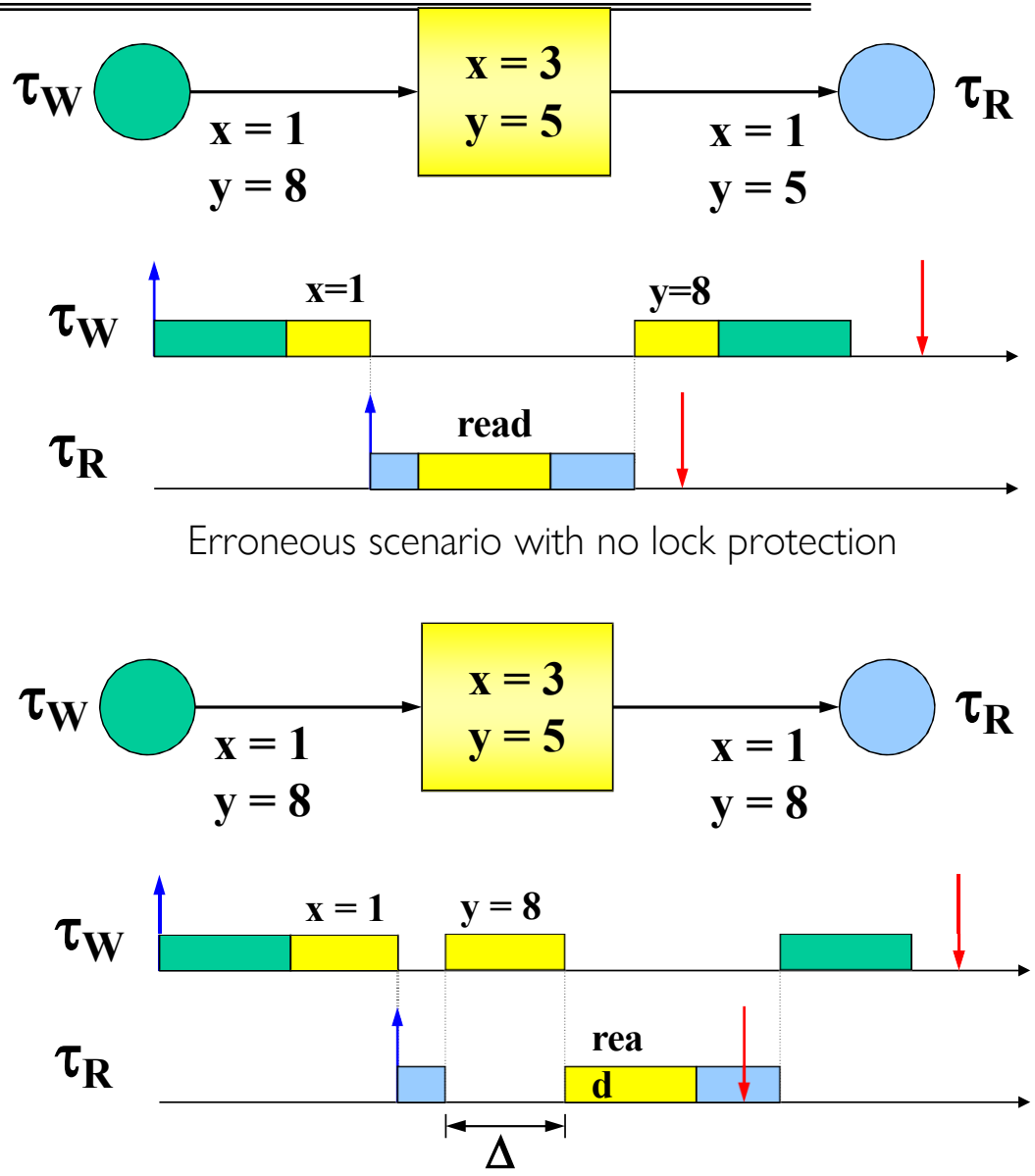
- Tasks must be executed with specific precedence relations, specified by a Directed Acyclic Graph (Precedence Graph)
- Example application of parts inspection in a factory. Tasks:
  - Image acquisition (acq1, acq2)
  - Edge detection (edge1, edge2)
  - Shape detection (shape), pixel disparities (disp)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





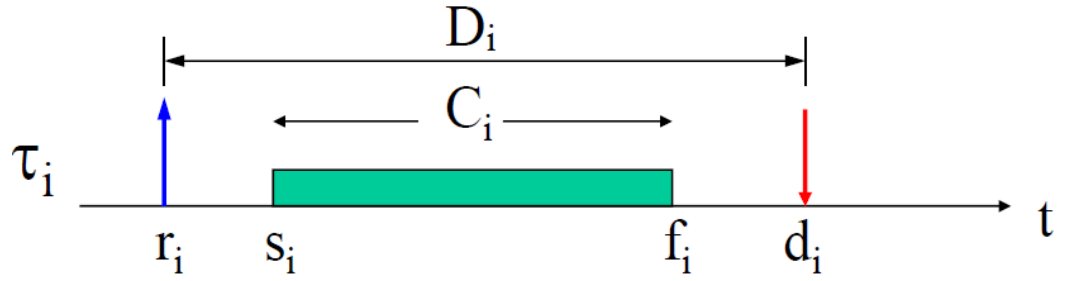
Resource Constraints

- 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.



Scheduling Metrics

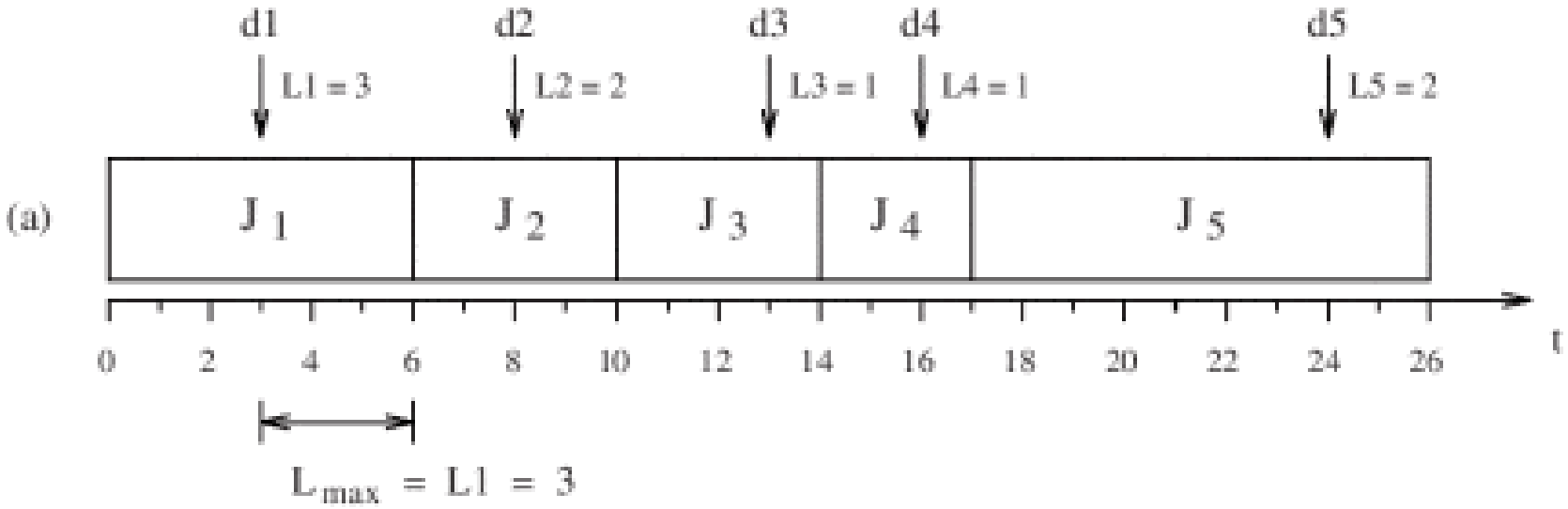
- Lateness $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; if a task completes before the deadline, its lateness is negative.
- Tardiness or exceeding time $E_i = \max(0, L_i)$ is the time a task stays active after its deadline; if a task completes before the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

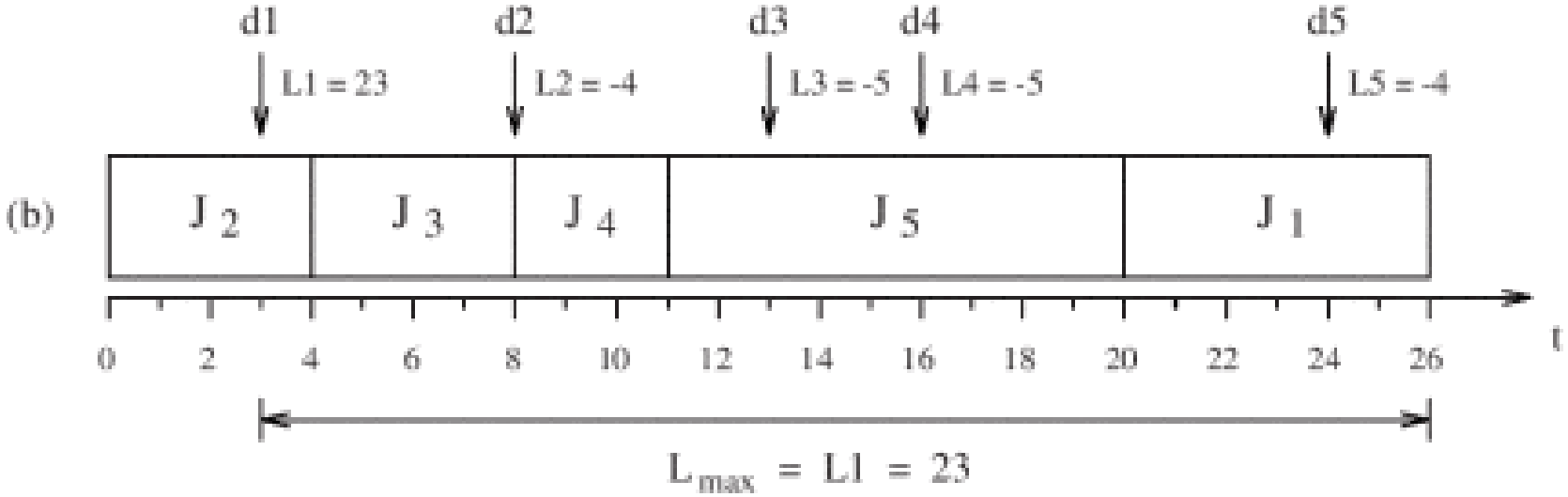


$\mathbf{r_i}$	release time (arrival time $\mathbf{a_i}$)
$\mathbf{s_i}$	start time
$\mathbf{C_i}$	worst-case execution time (wcet)
$\mathbf{d_i}$	absolute deadline
$\mathbf{D_i}$	relative deadline
$\mathbf{f_i}$	finishing time

Example: Lateness

- Which schedule is better depends on application requirements:
- In (a), the maximum latenes is minimized, but all jobs J_1 to J_5 miss their deadlines.
- In (b), the maximal lateness is larger, but only one job J_1 misses its deadline.



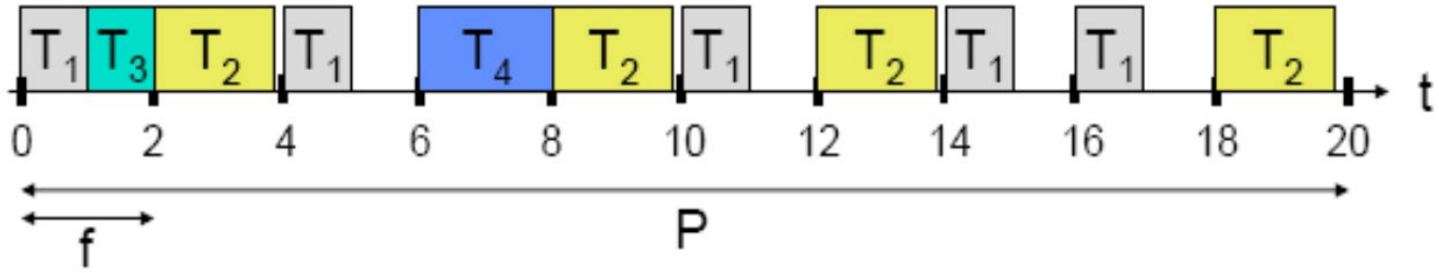


Scheduling Approaches

- Static cyclic scheduling
  - All task invocation times are computed offline and stored in a table
  - Runtime dispatch is a simple table lookup
- Fixed 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)

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



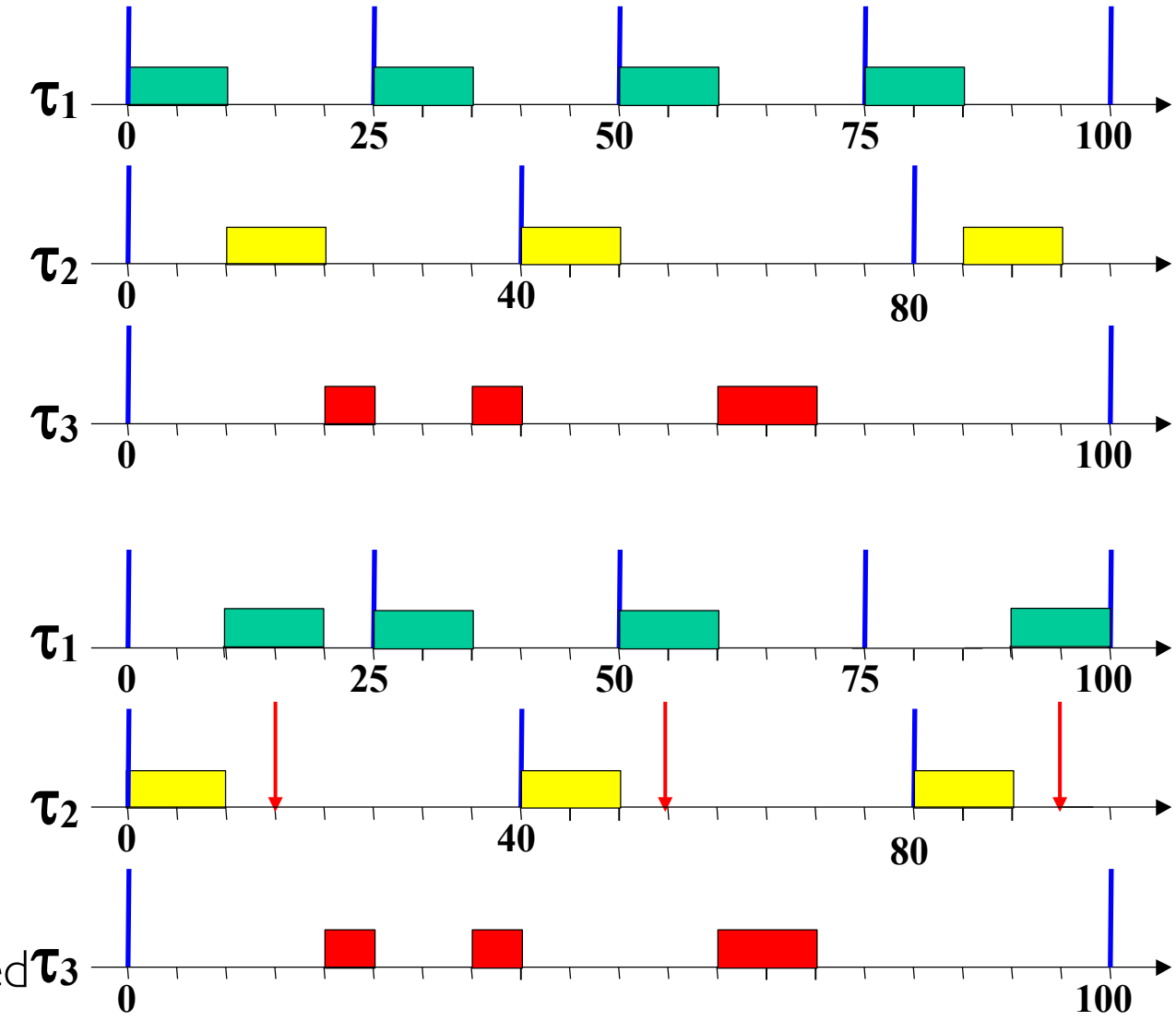
Fixed-Priority Scheduling

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 Scheduling (RMS)
  - Assign higher priority to task with smaller period
  - When D = T, RMS 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 Scheduling (DMS)
  - Assign higher priority to task with smaller relative deadline (indicated by red downward arrow for τ_2)
  - When D < T, DMS is the optimal priority assignment
- Why do we want deadline < period?
  - Some events happen infrequently, but need to be handled urgently



Two Schedulability Analysis Approaches

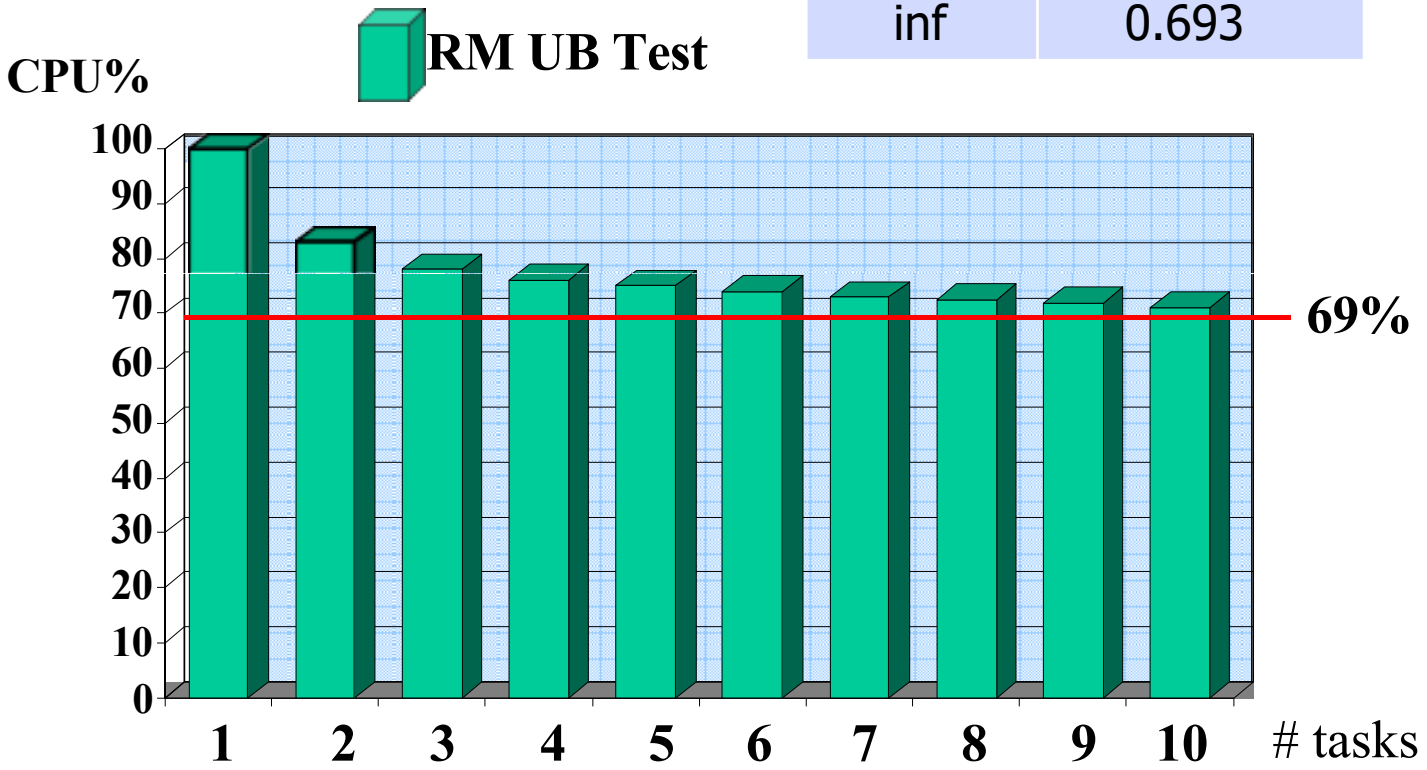
- Utilization bound test
  - Calculate total CPU utilization and compare it to a known bound
  - Polynomial time complexity
  - Pessimistic: sufficient but not necessary condition for schedulability

- 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

- A taskset is schedulable under RM scheduling if its total 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 fai
  - If periods are harmonic (larger period divisible by smaller periods), then utilization bound is 1

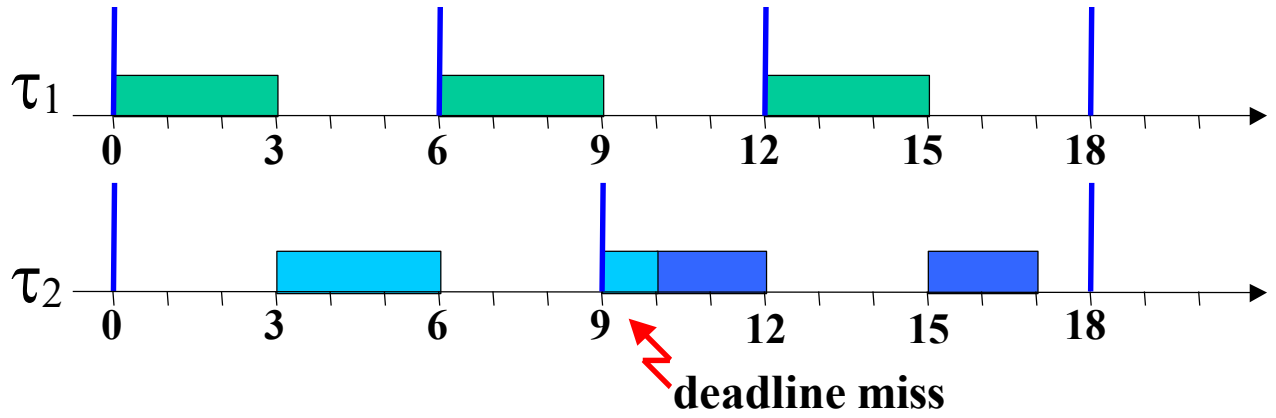
# Tasks	Bound
1	1.00
2	0.828
3	0.780
4	0.757
5	0.743
10	0.718
inf	0.693



Utilization Bound Test Examples

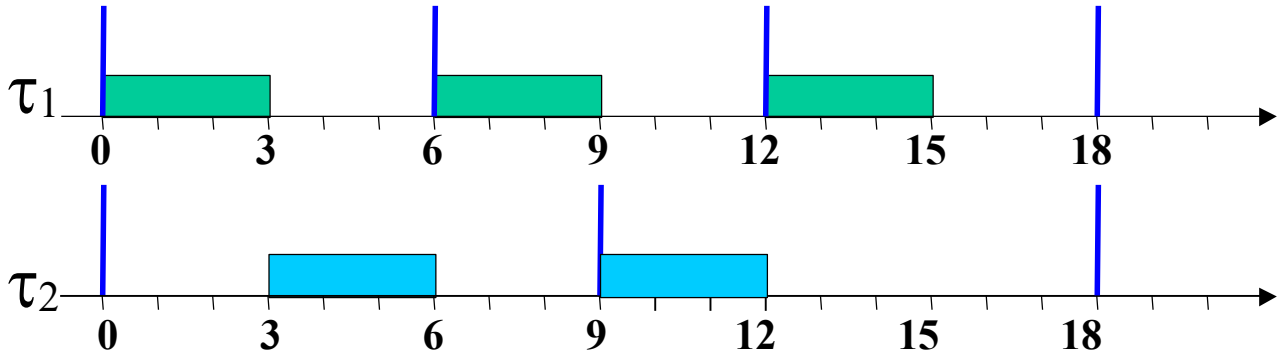
Taskset non-schedulable

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



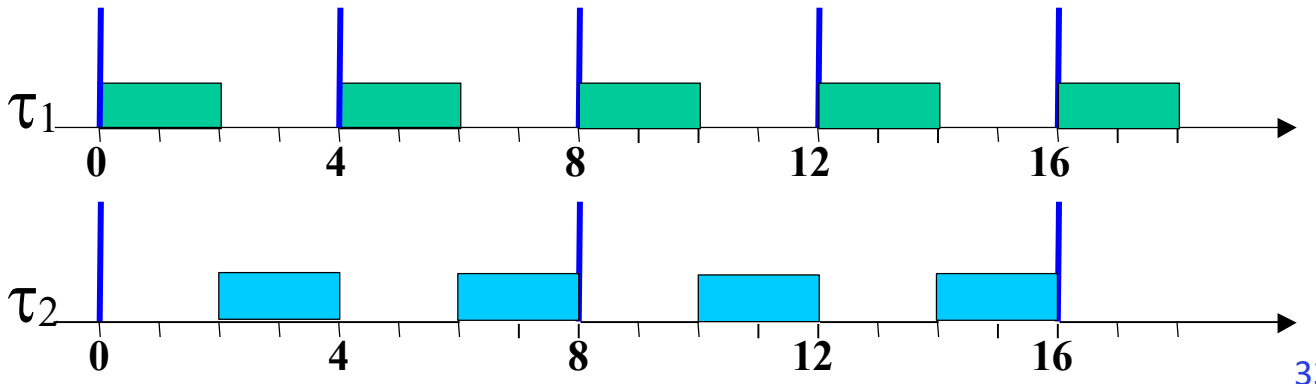
Taskset schedulable (UB test is sufficient but not necessary condition)

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



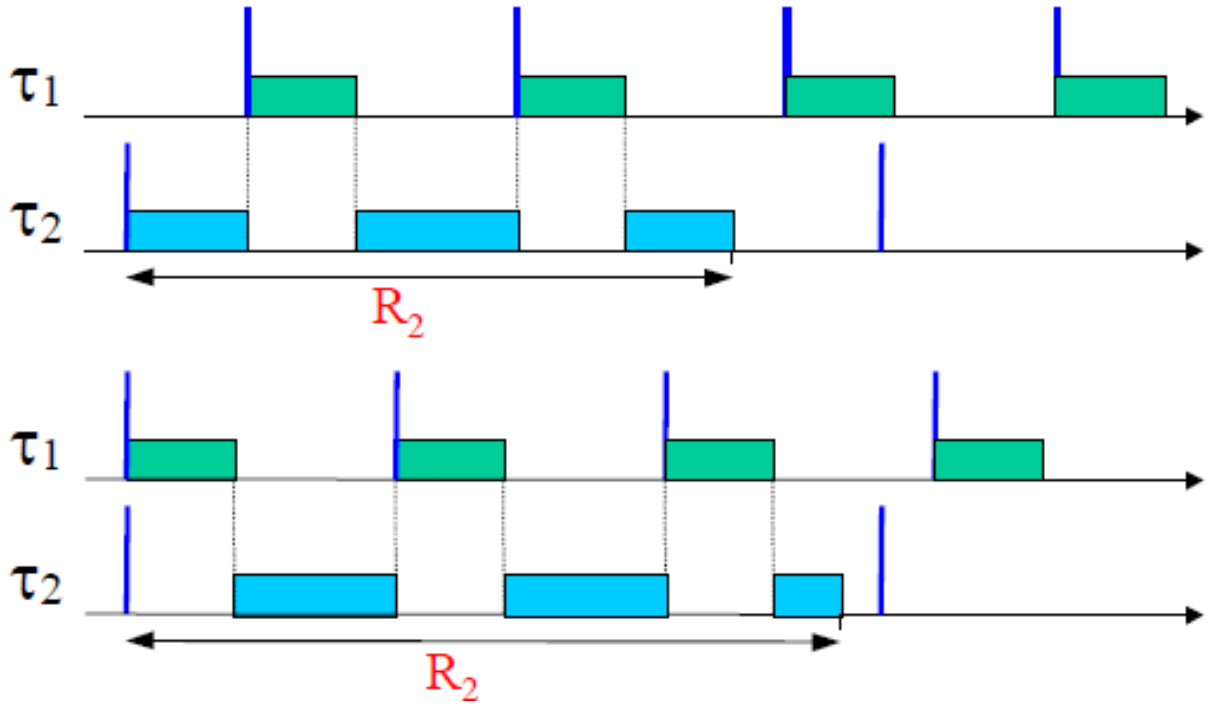
Taskset schedulable (periods are harmonic)

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



Response Time Analysis (RTA)

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



Response Time Analysis (RTA)

- 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 if $R_i \leq D_i$
- The taskset is schedulable of all tasks are schedulable.
- WCRT R_i is computed by solving the following recursive equation:
- $R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$
  - 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

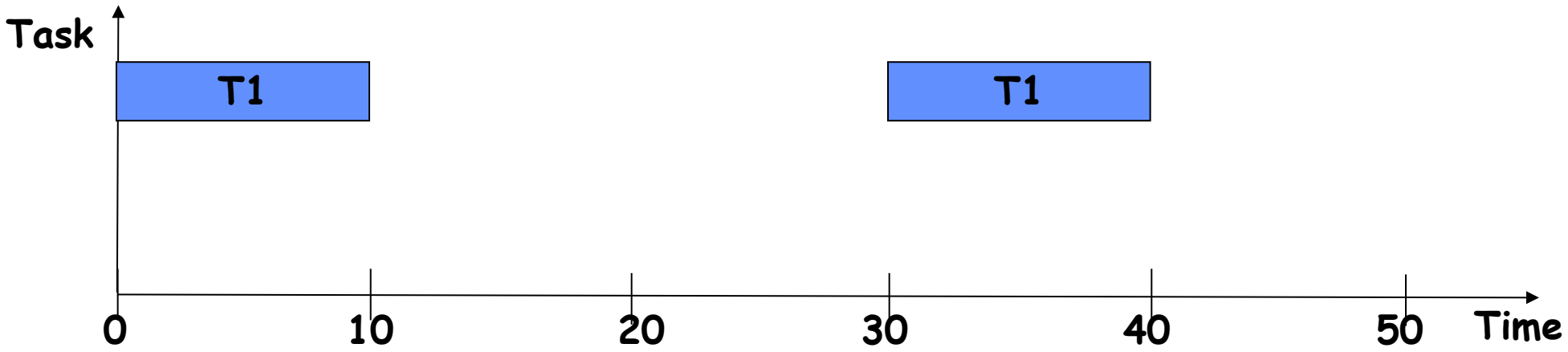
- Utilization = $\sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{10}{30} + \frac{10}{40} + \frac{12}{52} = 0.81$
- Utilization bound $(N = 3) = 3 * \left(2^{1/3} - 1\right) = 0.78$
- Utilization bound test fails since 0.81 > 0.78
- But taskset is actually schedulable

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

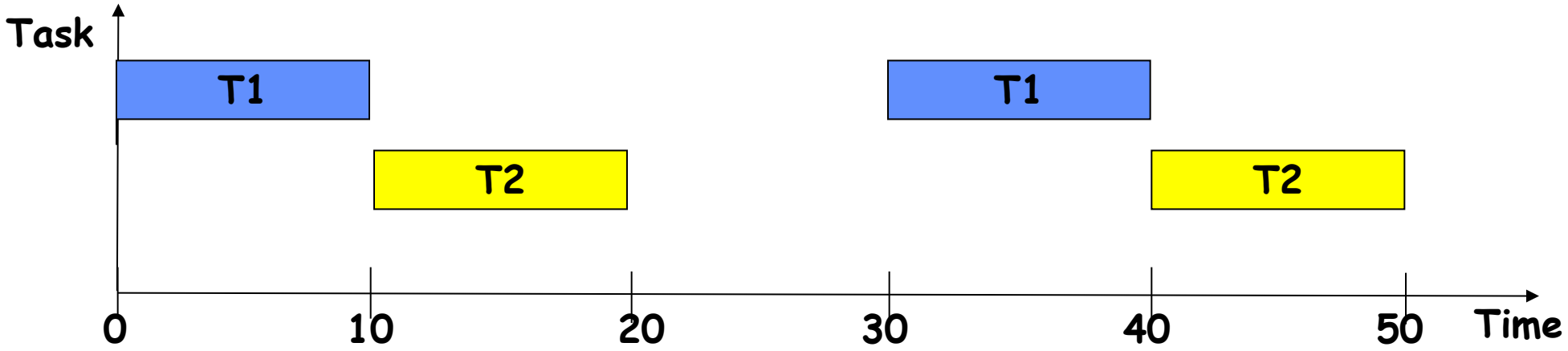
- T1 is the highest priority task, with no interference from other tasks $hp(1) = \emptyset$
- $R_1 = C_1 + 0 = 10$
- $R_1 < D_1$, so T1 is schedulable



Task T2

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

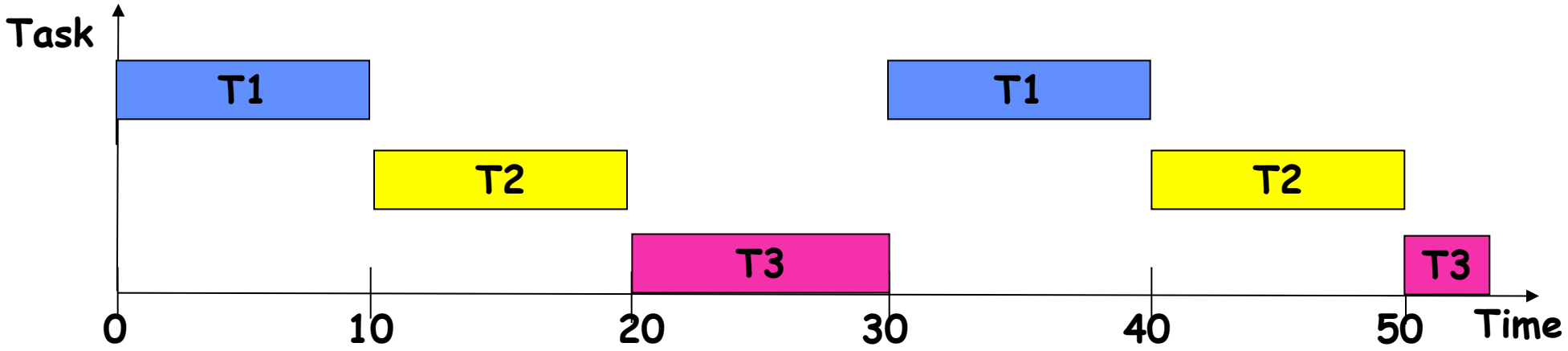
- T2 is the medium priority task, with interference from higher priority Task 1 $hp(2) = 1$
- $R_2 = C_2 + \lceil \frac{R_2}{T_1} \rceil * C_1 = 10 + \lceil \frac{R_2}{30} \rceil * 10$
- Solve for $R2$ recursively, starting with initial value $R_2 = C_2 = 10$:
  - Iteration 1: $R_2 = 10 + \lceil \frac{10}{30}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

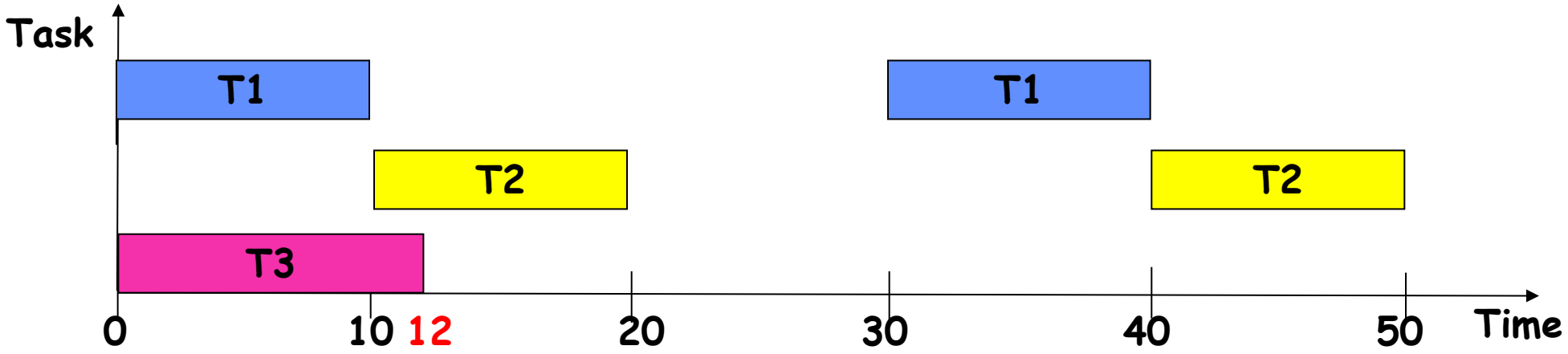
- T3 is the lowest priority task, with interference from higher priority tasks $hp(3) = \{1,2\}$
- $R_3 = C_3 + \lceil \frac{R_3}{T_1} \rceil * C_1 + \lceil \frac{R_3}{T_2} \rceil * C_2 = 12 + \lceil \frac{R_3}{30} \rceil * 10 + \lceil \frac{R_3}{40} \rceil * 10$
- Solve for R_3 recursively, starting 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

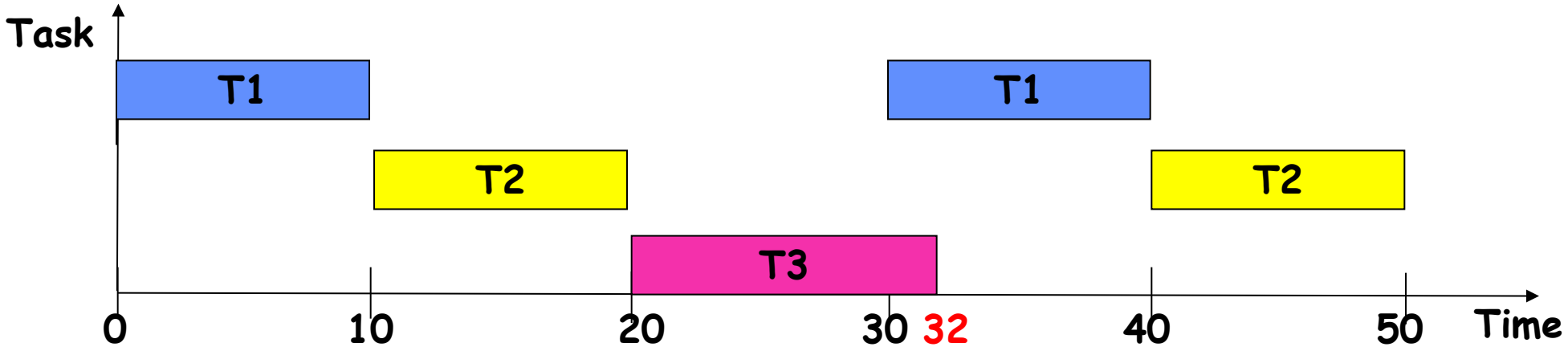
- Initially $R_3 = 12$
- We have not taken into account any preemption delays from higher priority tasks 1 and 2 yet



RTA for Task 3: Iteration 1

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

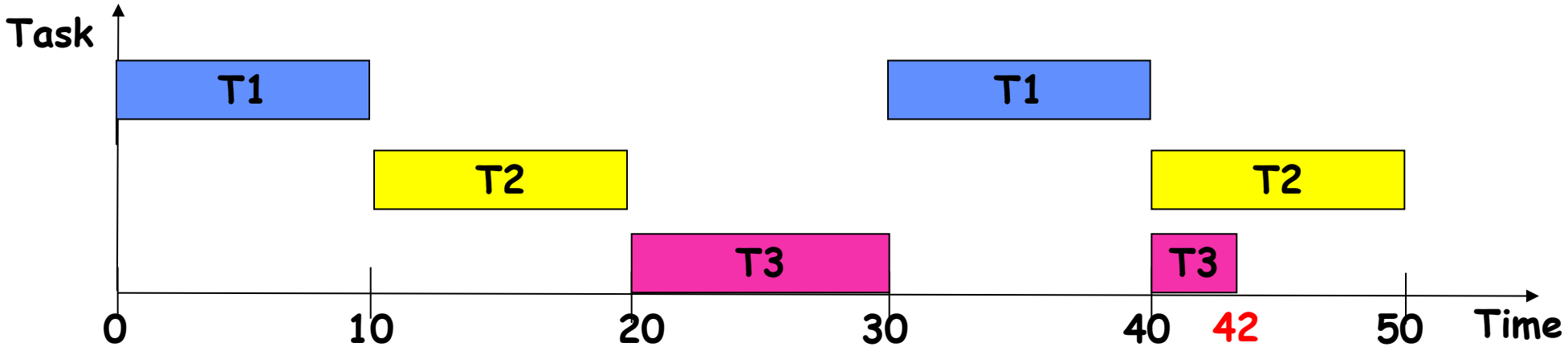
- $R_3 = 12 + \left\lceil \frac{12}{30} \right\rceil * 10 + \left\lceil \frac{12}{40} \right\rceil * 10$
- $= 12 + 1 * 10 + 1 * 10 = 32$
- T1 preempts T3 once, and T2 preempts T3 once
  - since all 3 tasks are released at time 0 (synchronous release time assumption), and T1 and T2 have higher priority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

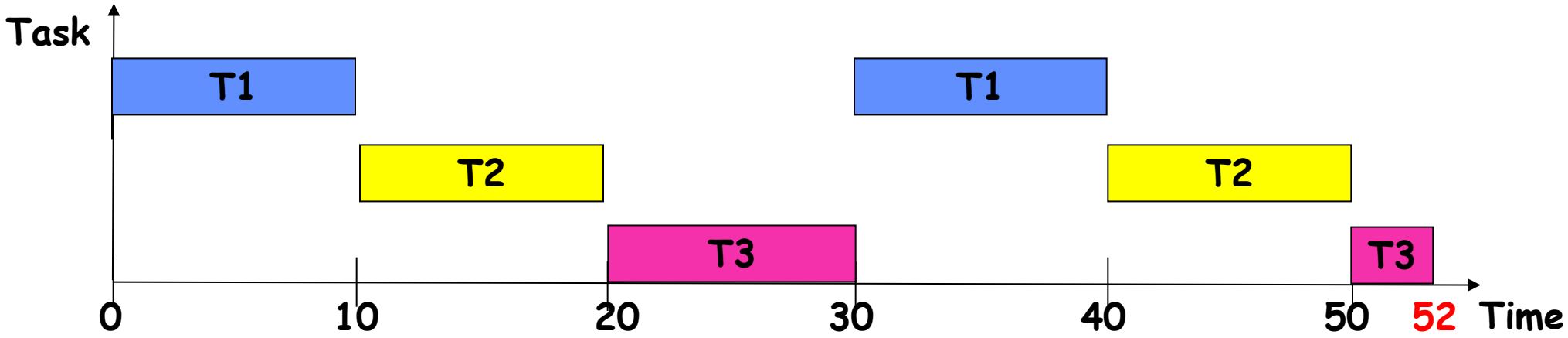
- $R_3 = 12 + \left\lceil \frac{32}{30} \right\rceil * 10 + \left\lceil \frac{32}{40} \right\rceil * 10$
- $= 12 + 2 * 10 + 1 * 10 = 42$
- T1 preempts T3 twice, and T2 preempts T3 once
  - Since T3 has not finished execution at time 30, and another job of higher priority task T1 is released at time 30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

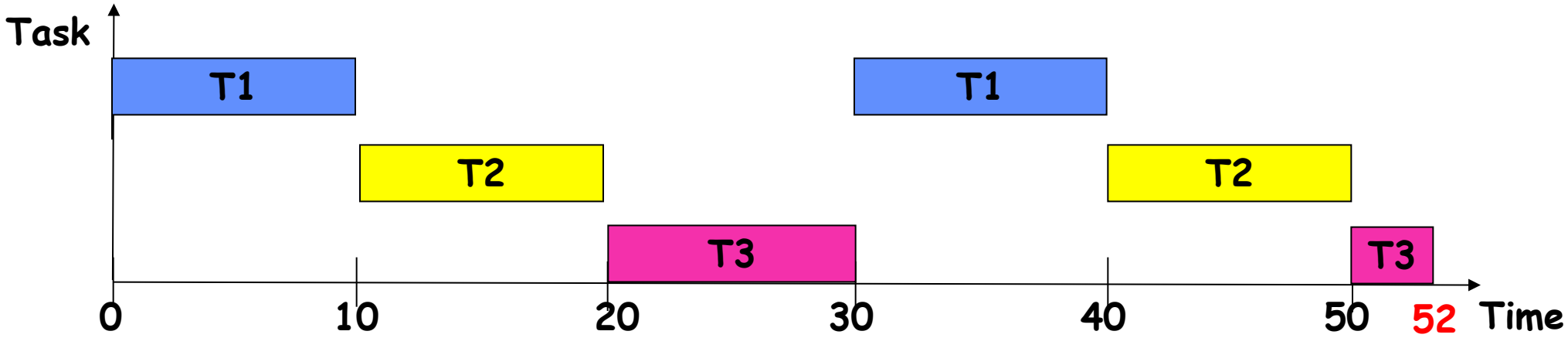
- $R_3 = 12 + \left\lceil \frac{42}{30} \right\rceil * 10 + \left\lceil \frac{42}{40} \right\rceil * 10$
- $= 12 + 2 * 10 + 2 * 10 = 52$
- T1 preempts T3 twice, and T2 preempts T3 twice
  - Since T3 has not finished execution at time 40, and another job of higher priority task T2 is released at time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

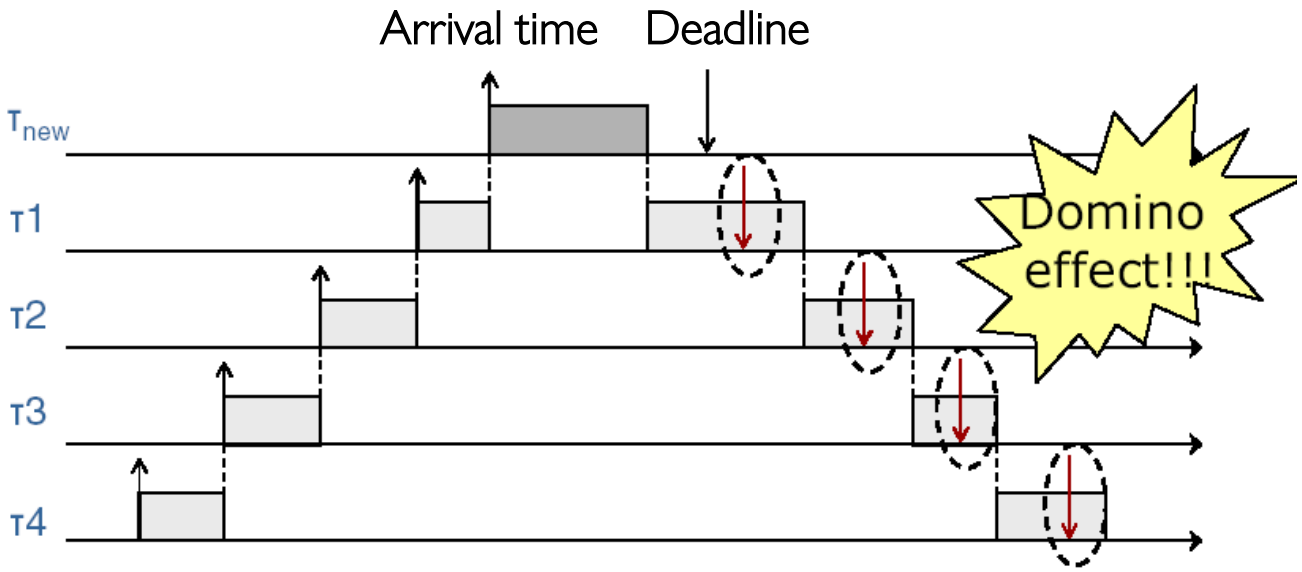
- $R_3 = 12 + \left\lceil \frac{52}{30} \right\rceil * 10 + \left\lceil \frac{52}{40} \right\rceil * 10 = 12 + 2 * 10 + 2 * 10 = 52$
- T1 preempts T3 twice, and T2 preempts T3 twice
  - Since T3 has finished execution at time 52, and the next arrivals of T1 and T2 are at time 60 and 80, respectively, 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 to be $R_3 = 52$



Earliest Deadline First (EDF) Scheduling

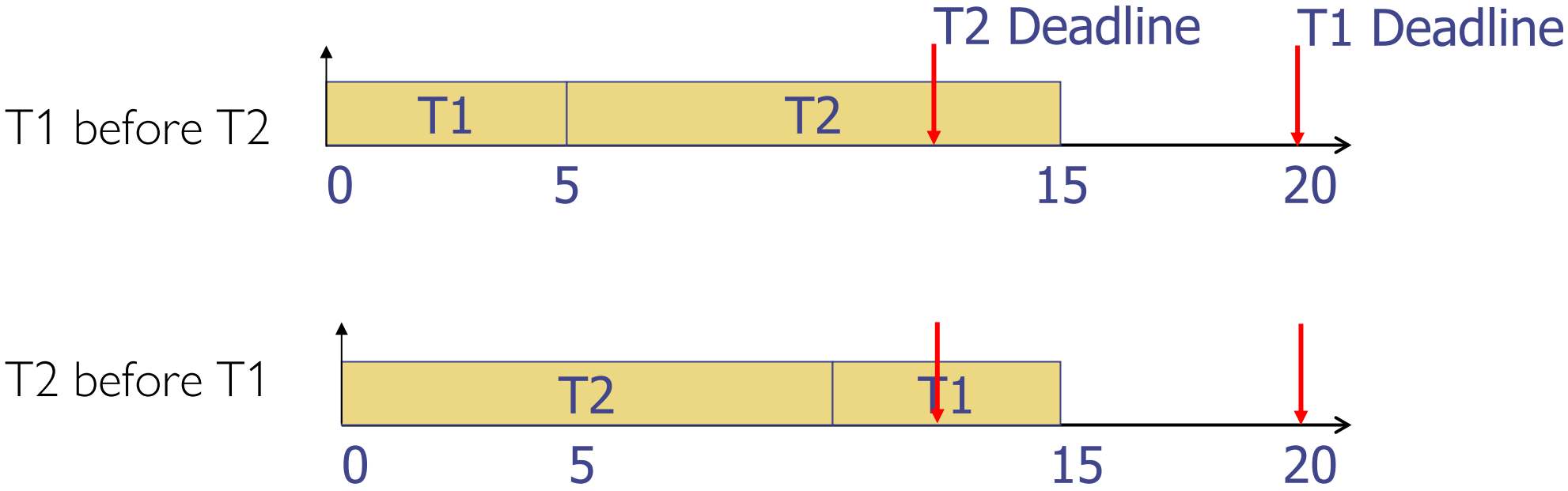
Earliest Deadline First (EDF)

- As each task enters the system, it is assigned a deadline, and its priority is determined by its absolute deadline d_i
  - The task with the earlier deadline is assigned the higher priority
  - Upwards arrows indicate arrival time; Downwards arrows indicate deadline
- Pros:
  - Optimal: can achieve 100% CPU utilization
- Cons:
  - High runtime overhead
  - Poor temporal isolation during overload



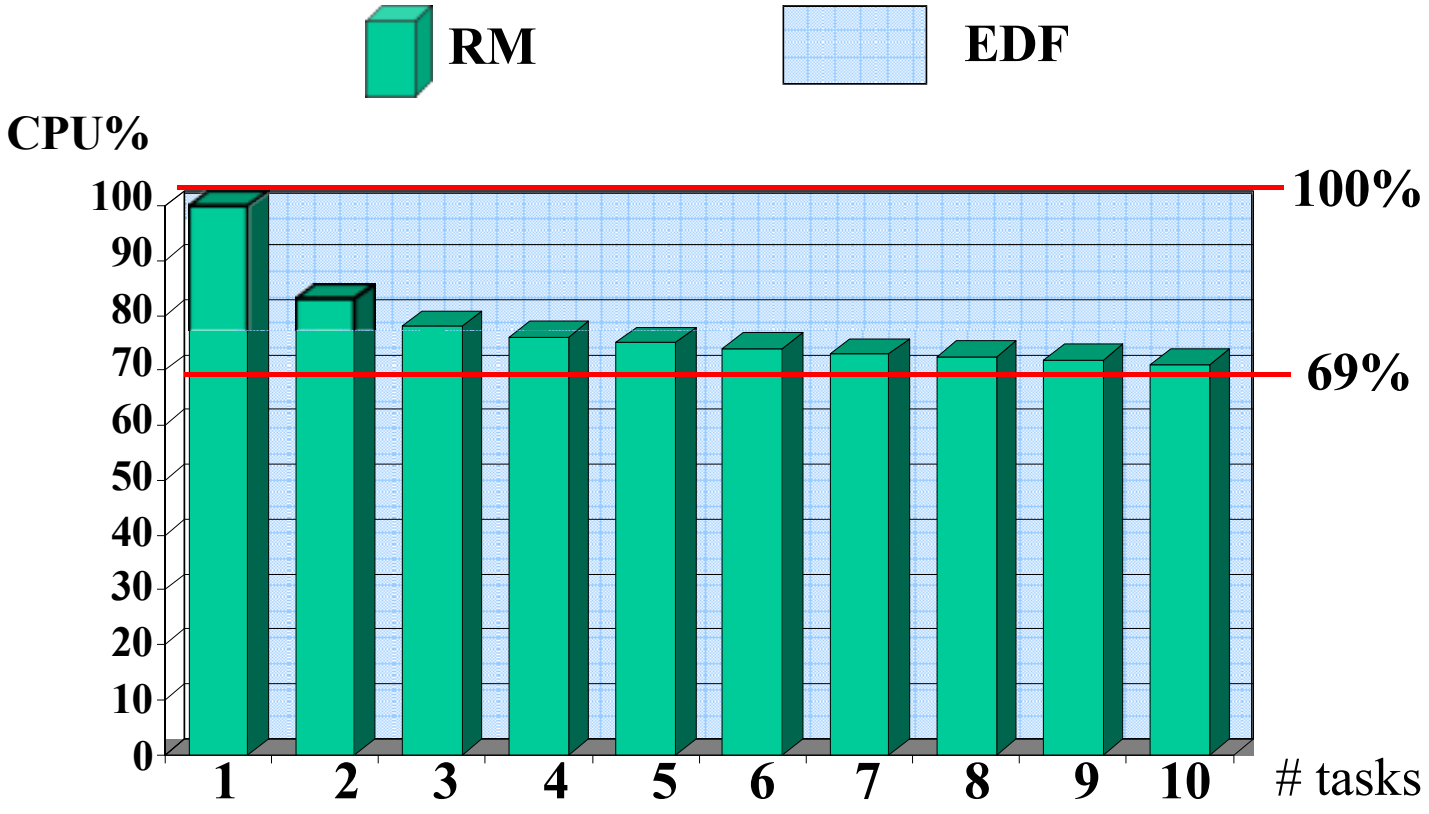
EDF Scheduling Example

- 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



Schedulable Utilization Bound: RM vs. EDF

- Recall the schedulable utilization bound for Fixed-Priority scheduling:
- A taskset is schedulable under RM scheduling if its total utilization $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$
  - $U \to 0.69$ as $N \to \infty$
- The schedulable utilization bound for EDF Scheduling is 1:
  - A taskset is schedulable under EDF scheduling if its total utilization $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$
  - Assumptions: task period equal to deadline ($P_i = D_i$); tasks are indepenc (no resource sharing)

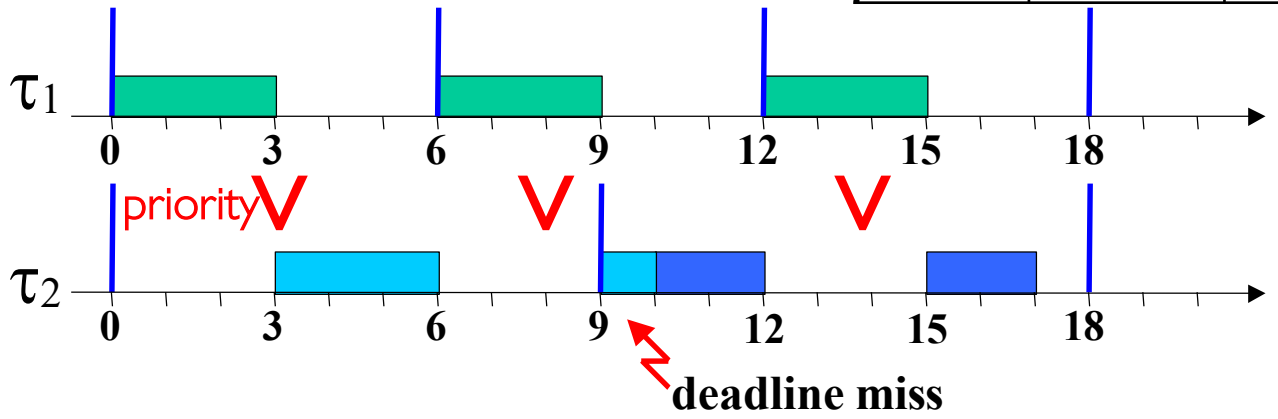


RM vs. EDF Example

Task	T=D	C
τ_1	6	3
τ_2	9	4

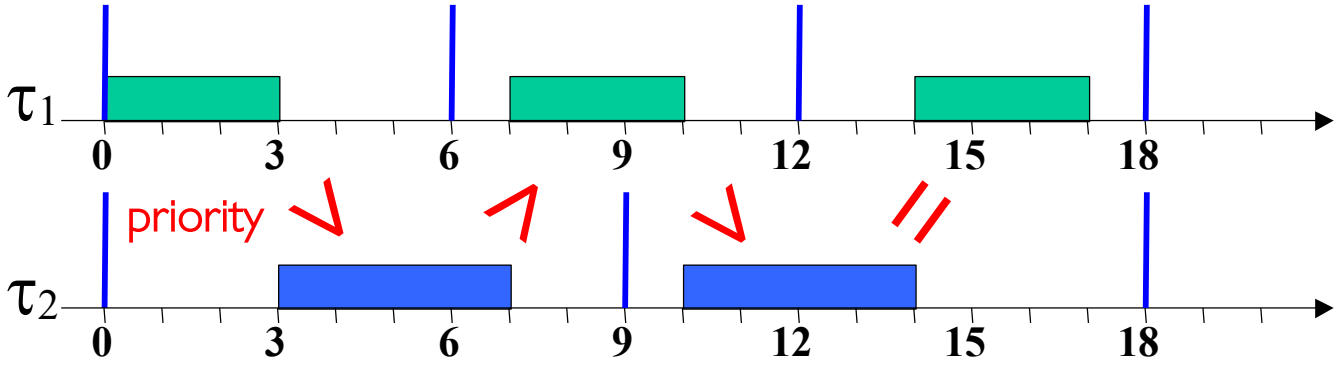
Under RM (Fixed-Priority scheduling), all jobs of τ_1 have higher priority than all jobs of τ_2. Taskset non-schedulable with RM

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



Under EDF (Dynamic Priority scheduling), different jobs of τ_1 and τ_2 may have different priorities, depending on their absolute deadlines d_i. Taskset schedulable with EDF

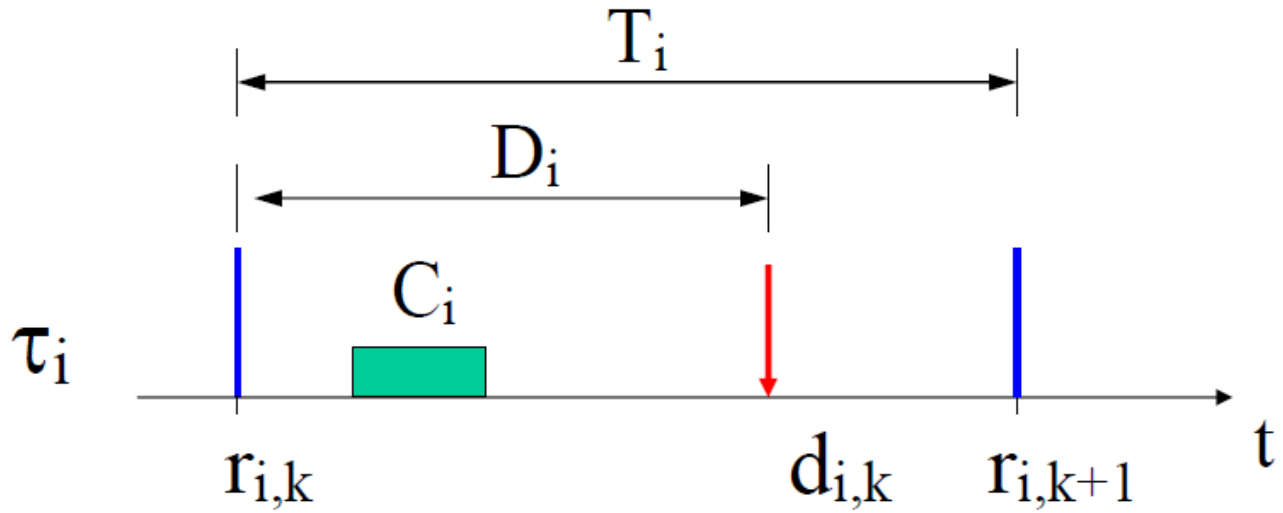
$$U = \frac{3}{6} + \frac{4}{9} = 0.944 < 1.0$$



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

Handling Tasks with D < T

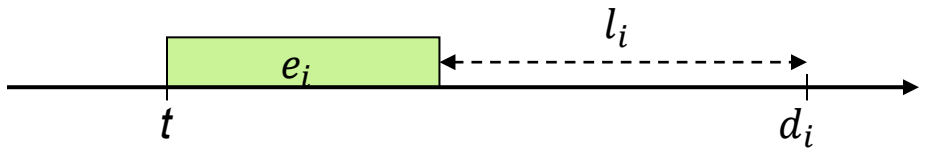
- Deadline monotonic (Fixed Priority):
  - A task with smaller relative deadline gets higher priority $P_i \propto 1/D_i$
  - Schedulability analysis: No utilization bound test; need to use Response Time Analysis (RTA)
- Earliest Deadline First (Dynamic-Priority):
  - A task with smaller absolute deadline gets higher priority $P_i \propto 1/d_i$
  - Schedulability analysis: No utilization bound test; need to use Processor Demand Analysis (details omitted)



Least Laxity First (LLF) Scheduling

Least Laxity First (LLF)

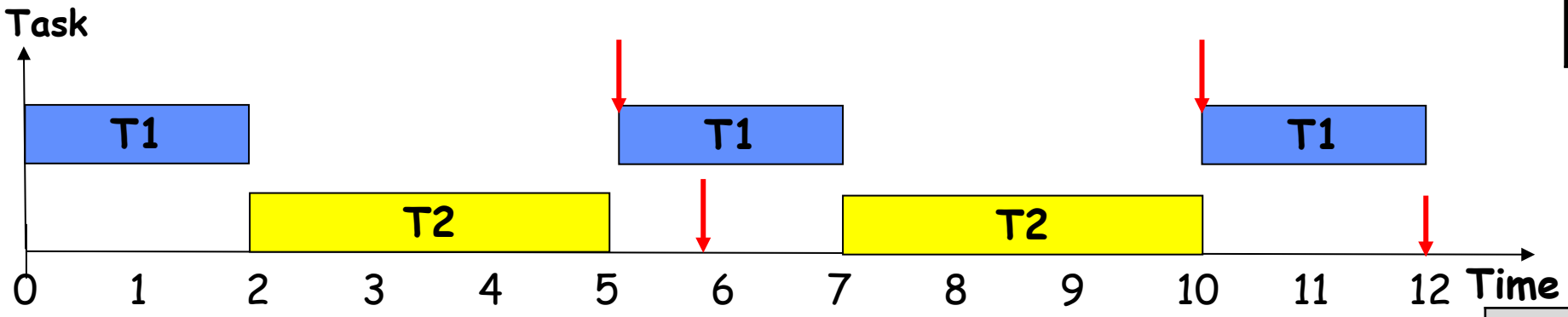
- LLF dynamically assigns priority to jobs based on their laxity (slack)
  - With absolute deadline d_i and remaining execution time e_i, laxity of τ_i's job at time t is $l_i = d_i - t - e_i$. Job with the smallest laxity has the highest priority
  - If an active job runs in the previous time slot, 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



- LLF is also optimal w.r.t. all online schedulers
  - EDF and LLF are both optimal scheduling algorithms
- LLF incurs frequent context switches, hence is less practical than EDF

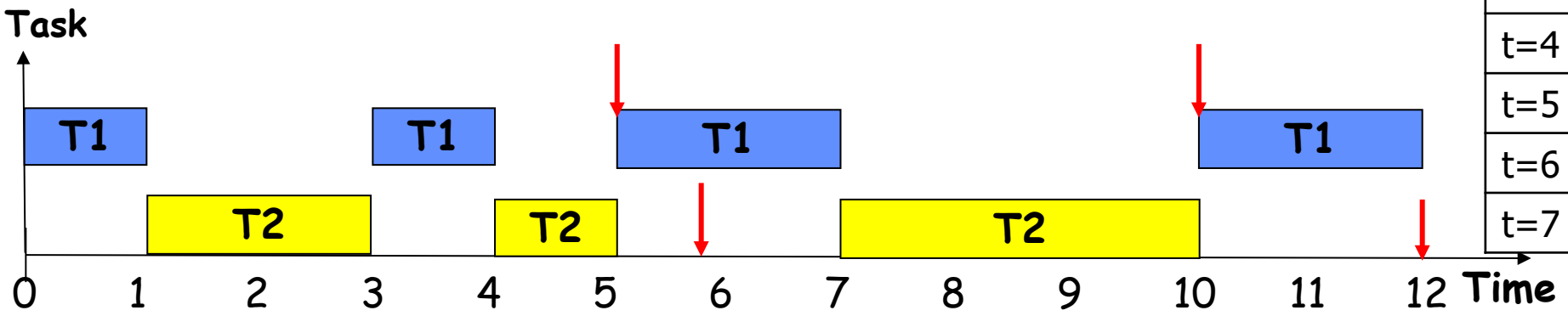
RM, EDF, LLF Example

Task	T=D	C
T1	5	2
T2	6	3



EDF and RM have the same schedule

Time	T_1 Laxity	T_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

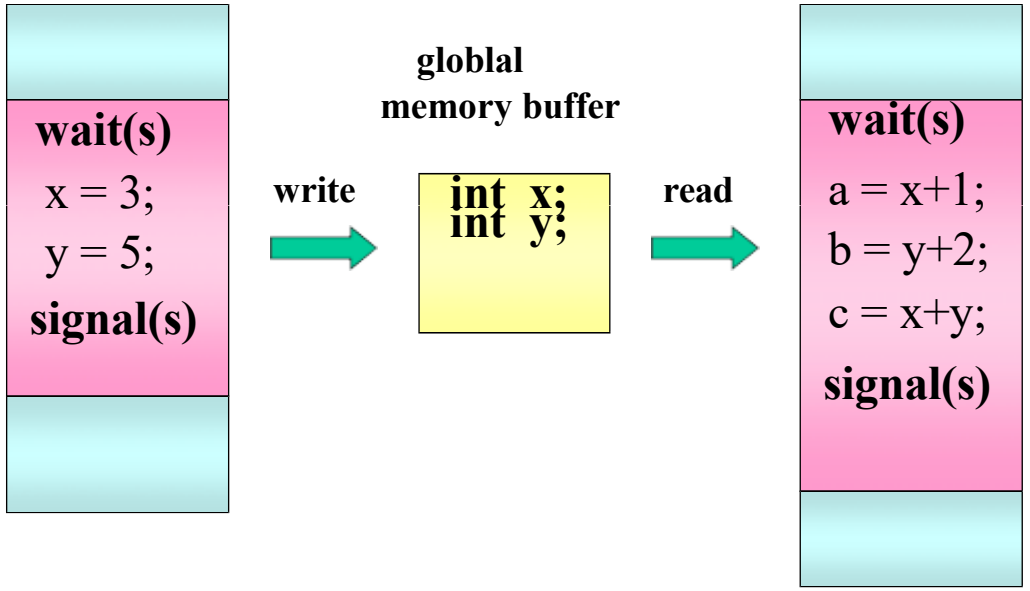


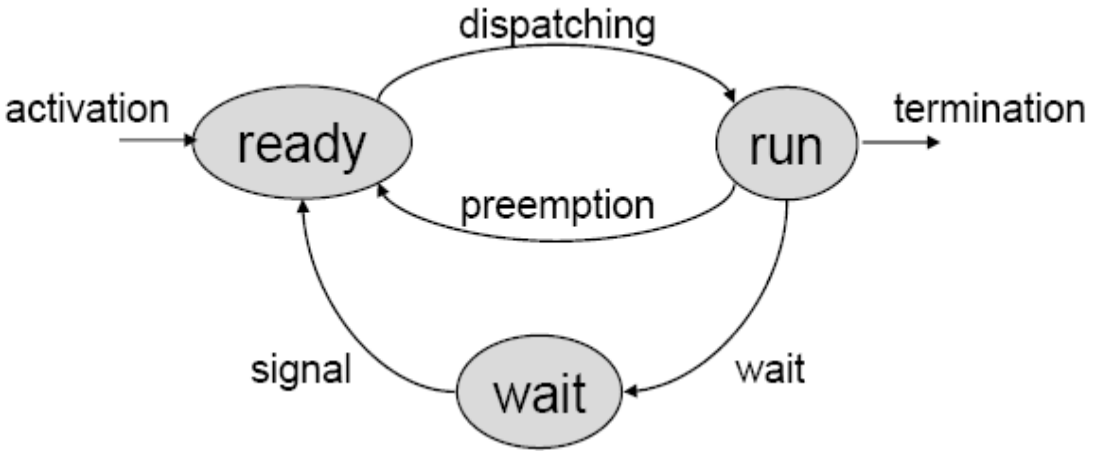
LLF has more frequent context switches

Resource Synchronization Protocols (for Fixed-Priority Scheduling)

Resource Sharing

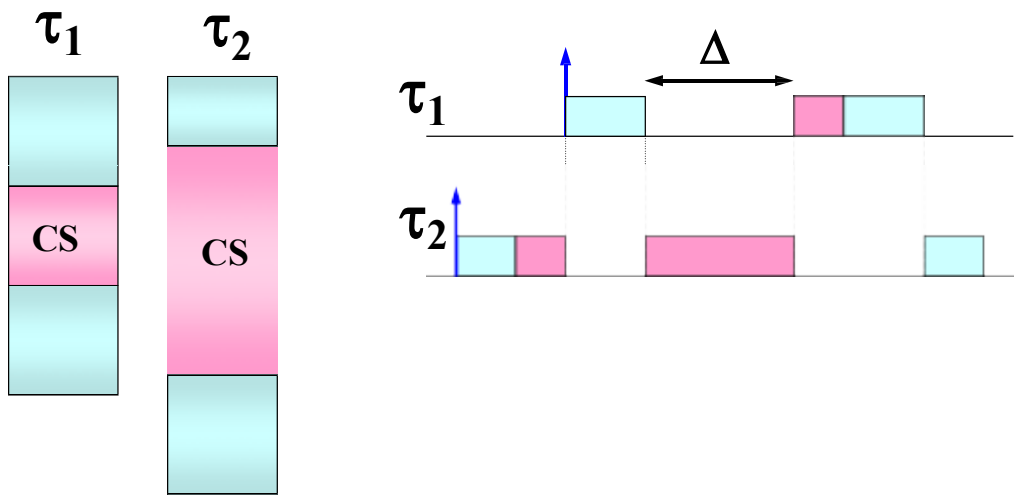
- Some shared resources do not allow simultaneous accesses but require mutual exclusion. A piece of code executed under mutual exclusion constraints is called a critical section.
- When two tasks access shared resource, semaphores are used to protect critical sections.
- Each shared resource R_i must 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

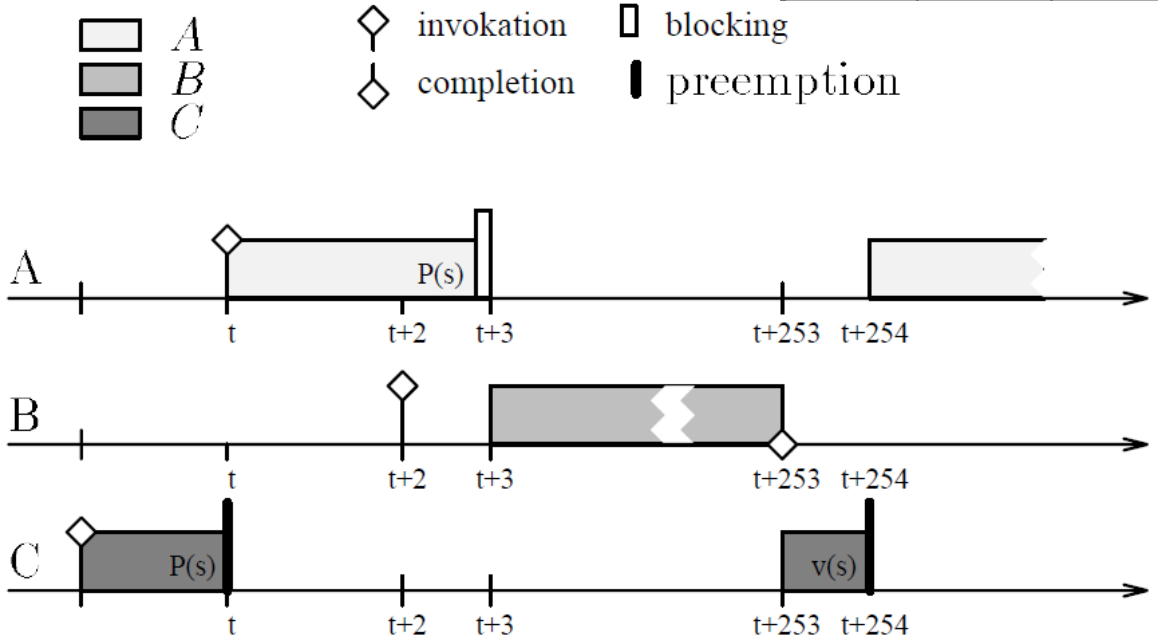
Task	T	D	C	Prio
A	50	10	5	H
B	500	500	250	M
C	3000	3000	1000	L

- Utilization=5/50+250/500+1000/3000=0.93
- WCRT (without blocking delays):
  - R_A=5, R_B=280, R_C=2500

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!



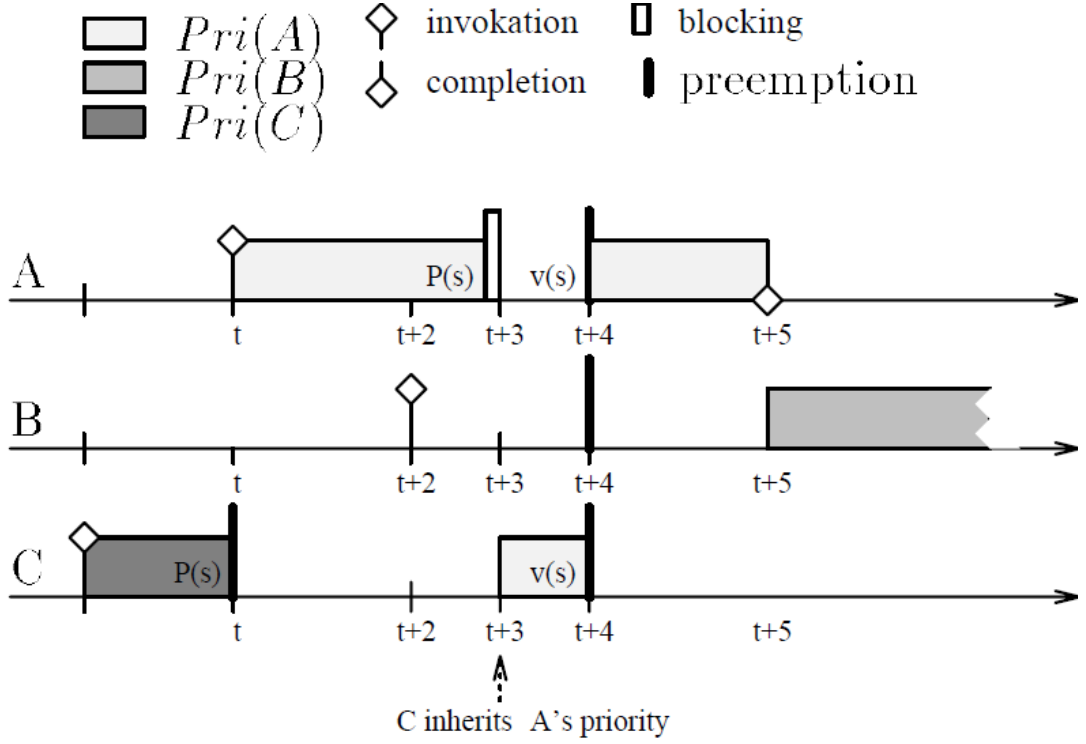
Priority Inversion and Priority Inheritance

- 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

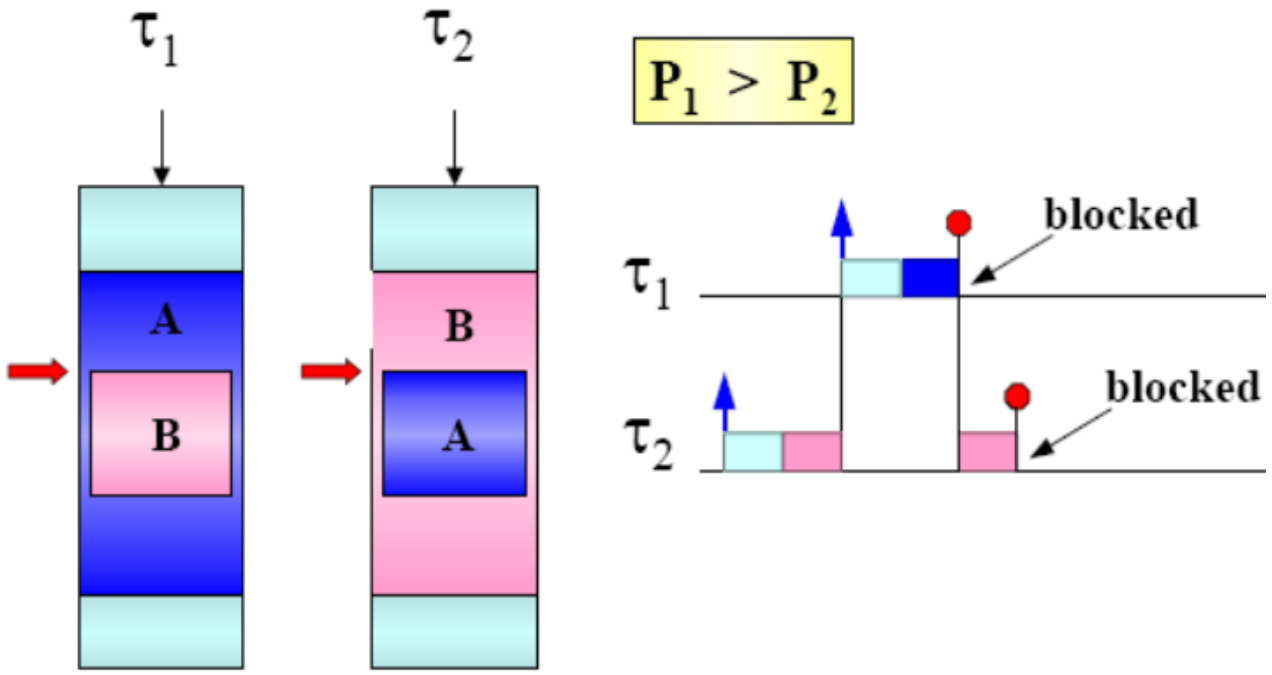
Task	T	D	C	Prio
A	50	10	5	H
B	500	500	250	M
C	3000	3000	1000	L

- 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.



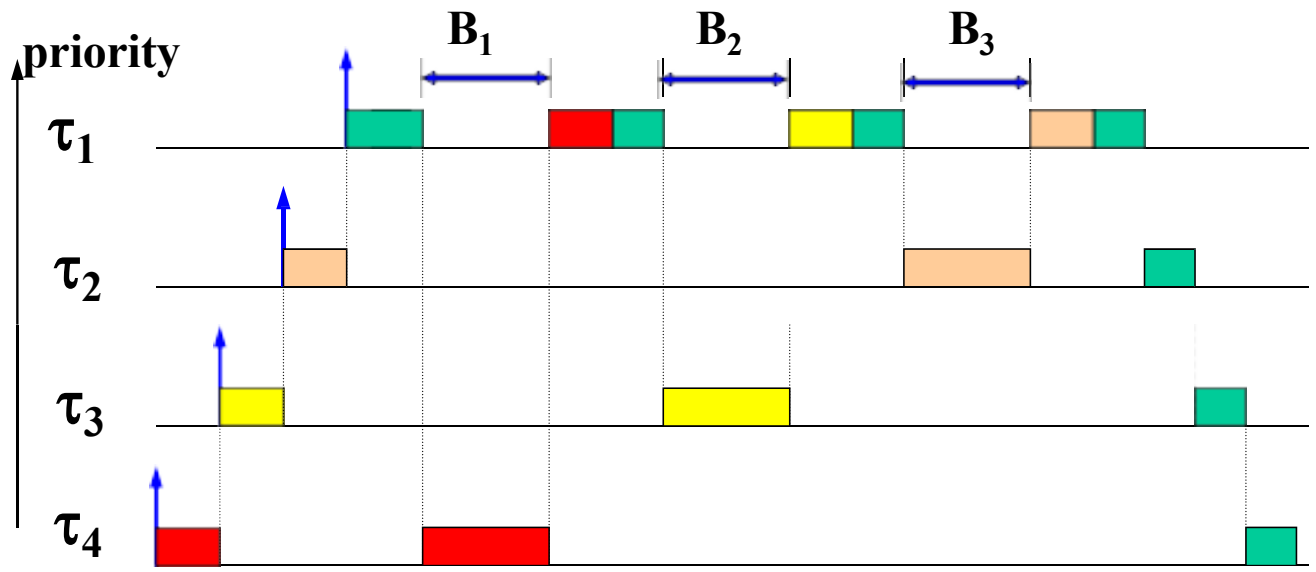
PIP Properties

- 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 (discussions omitted)



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 $U_i = \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i+B_i}{T_i} \leq i(2^{1/i} - 1)$

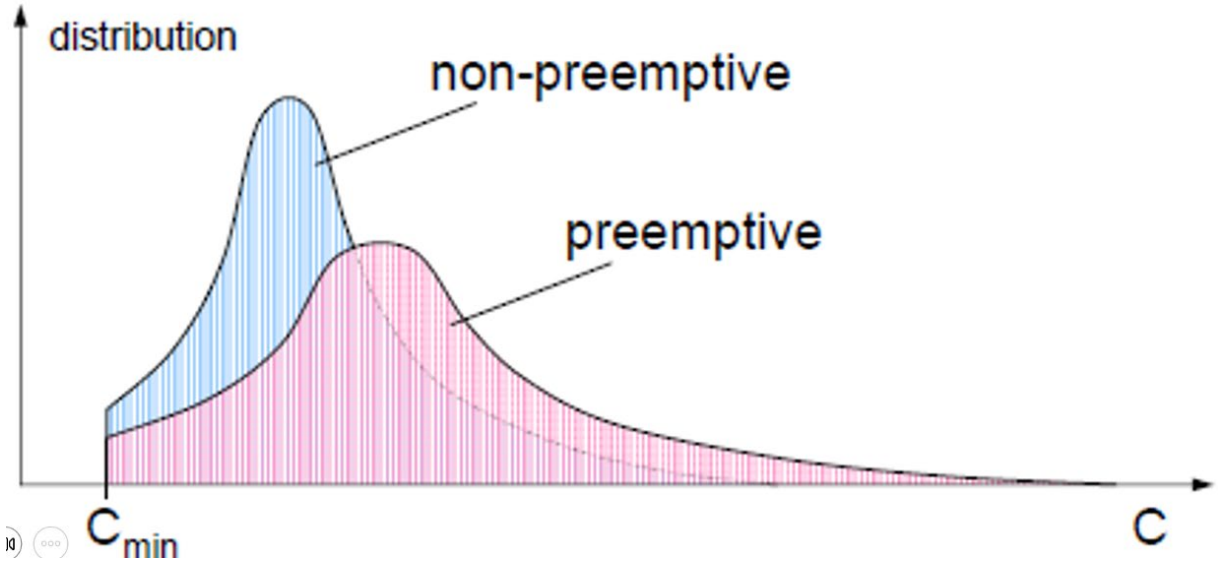
- Response Time Analysis (RTA) for RM scheduling with blocking time:
- WCRT R_i is computed by solving the following recursive equation:
- $R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$

Preemptive vs. Non-Preemptive Scheduling

Preemptive vs. Non-Preemptive Scheduling

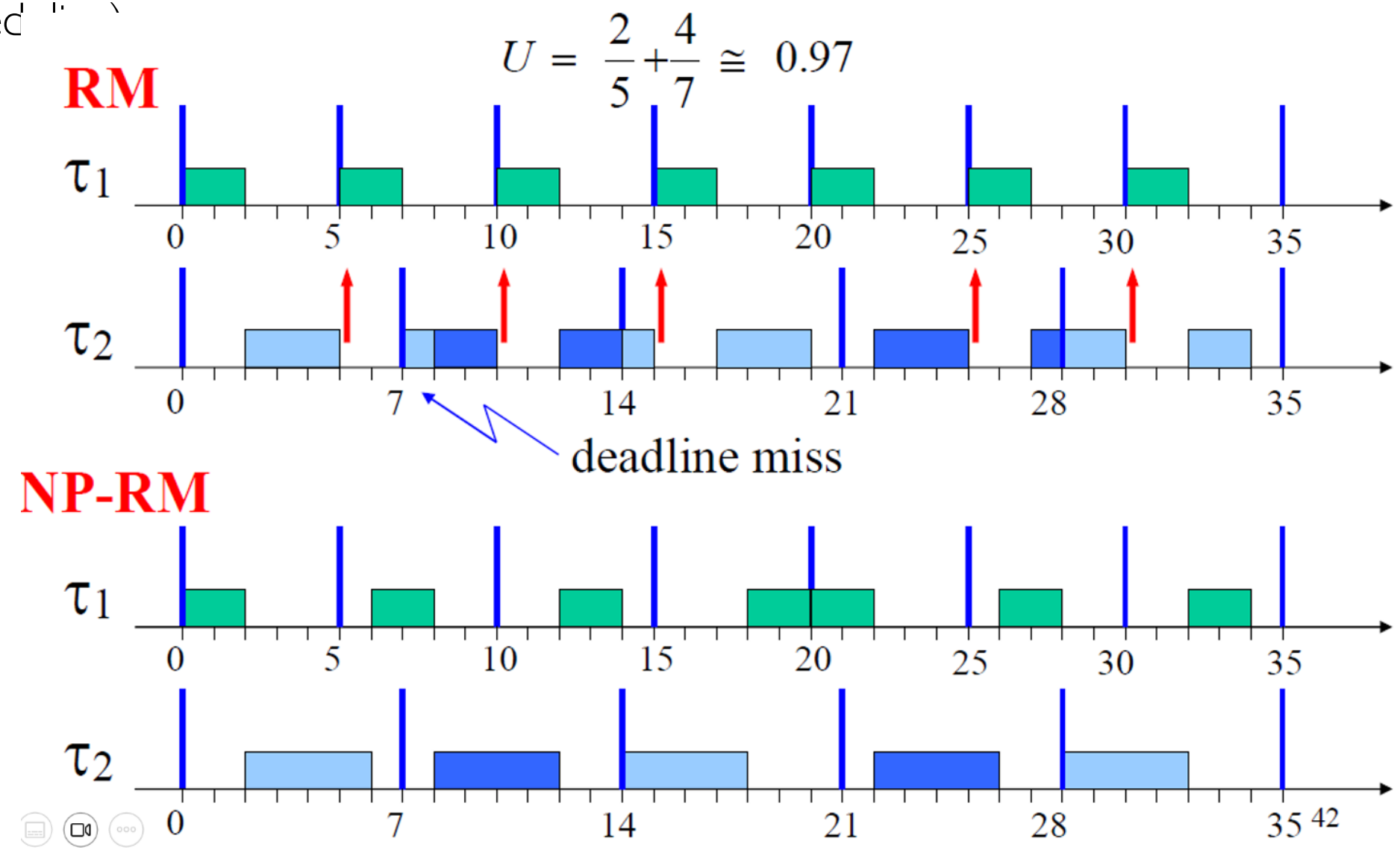
- Non-preemptive scheduling pros:
  - It reduces runtime overhead
    - Less context switches
    - No mutex locks needed for critical sections
  - It preserves program locality, improving the effectiveness of CPU cache
    - As a result, task WCET becomes smaller and execution time distribution becomes 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



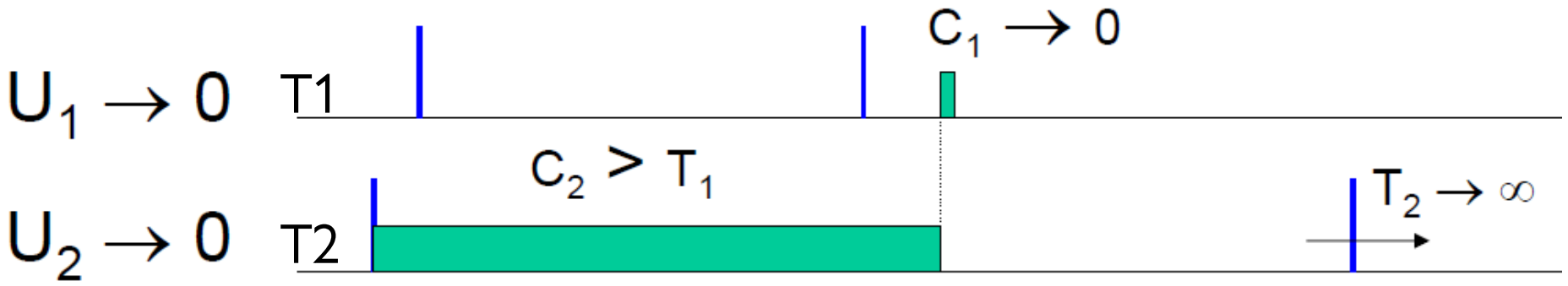
Sometimes NP Scheduling Improves Schedulability

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



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 non-schedulable.
  - If C_2 (WCET of T2) $\geq T_1$ (period of T1), then the taskset is non-schedulable 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: doubling the processor speed (reducing task execution times by half) makes task τ_1 miss its deadline

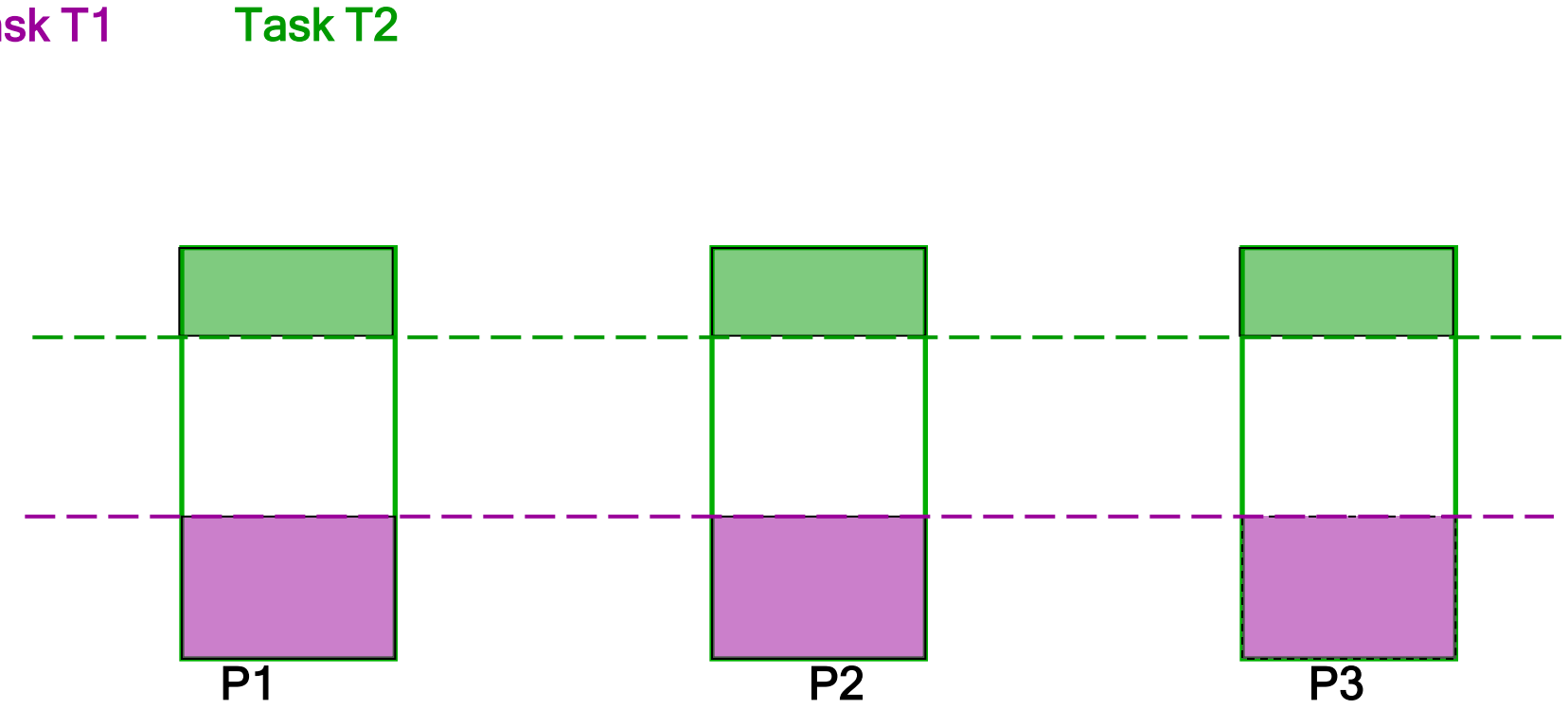
Multiprocessor Scheduling

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

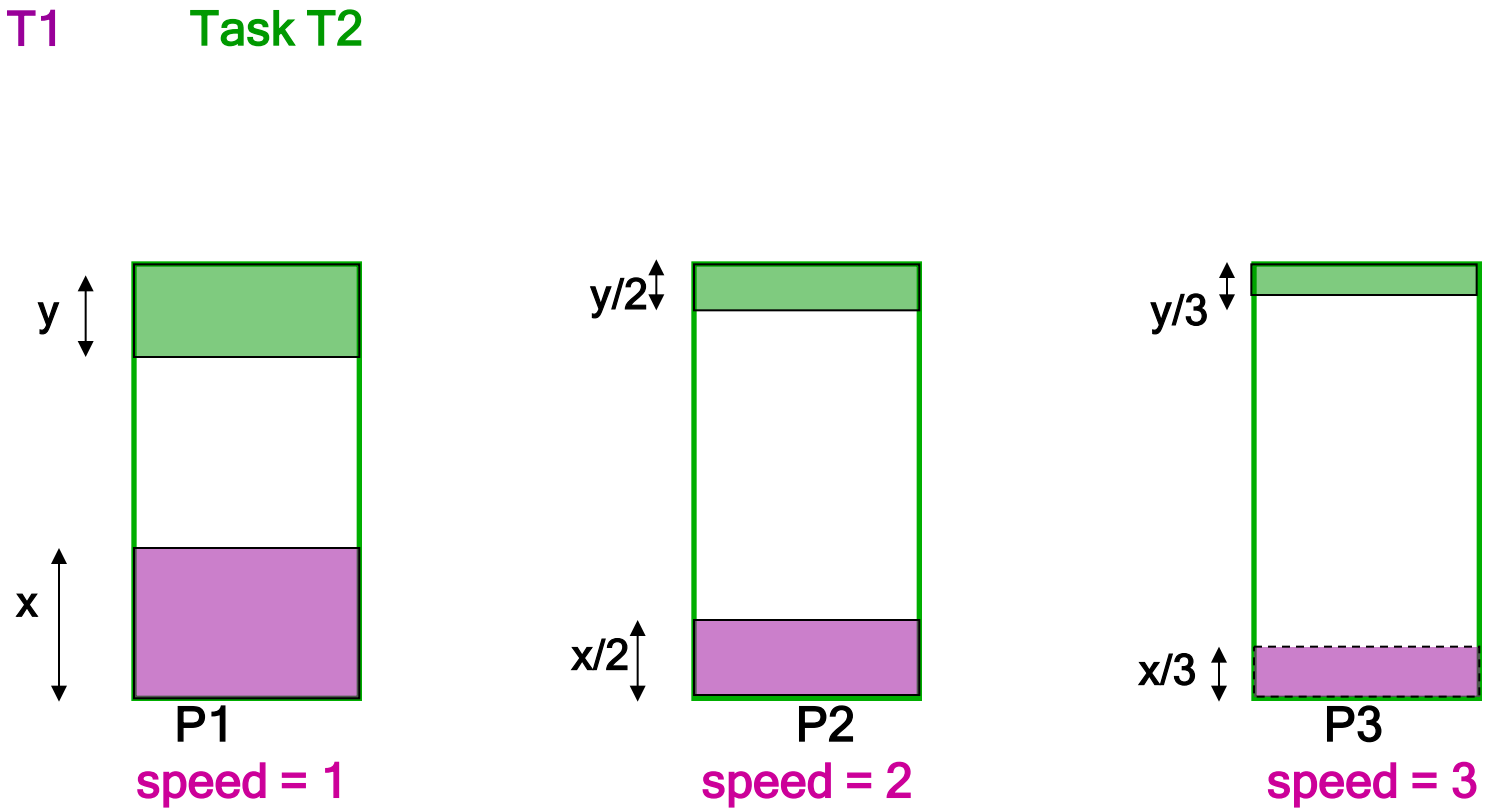
Multiprocessor Models

Identical multiprocessors: each processor has the same speed



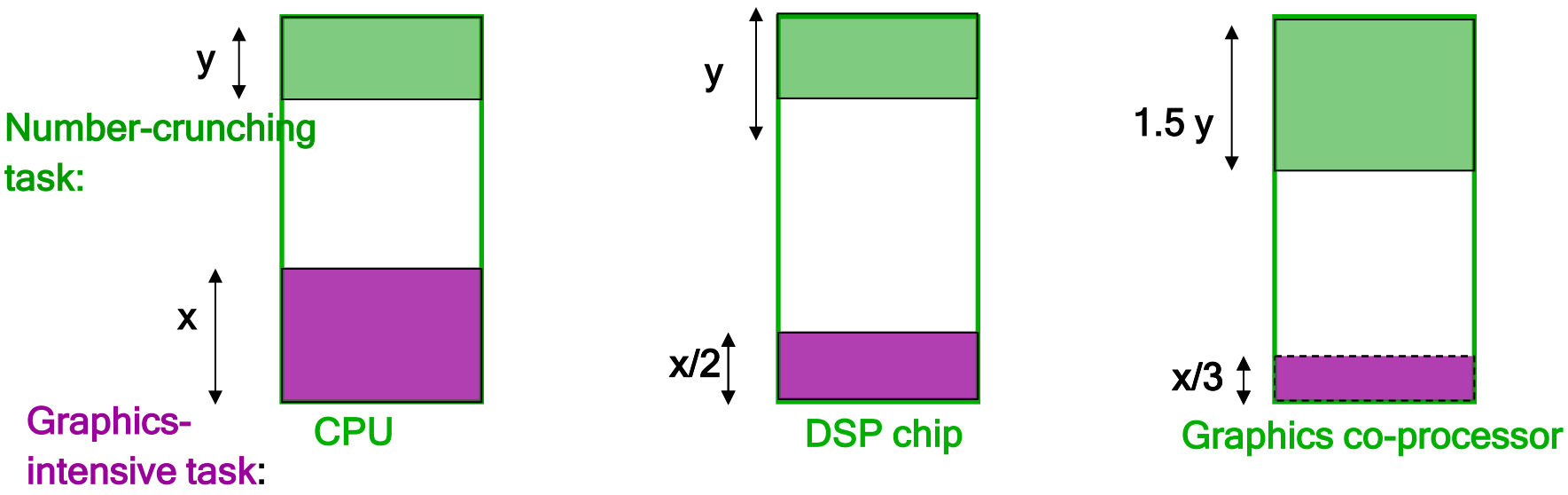
Multiprocessor Models

Uniform multiprocessors: different processors have different speeds



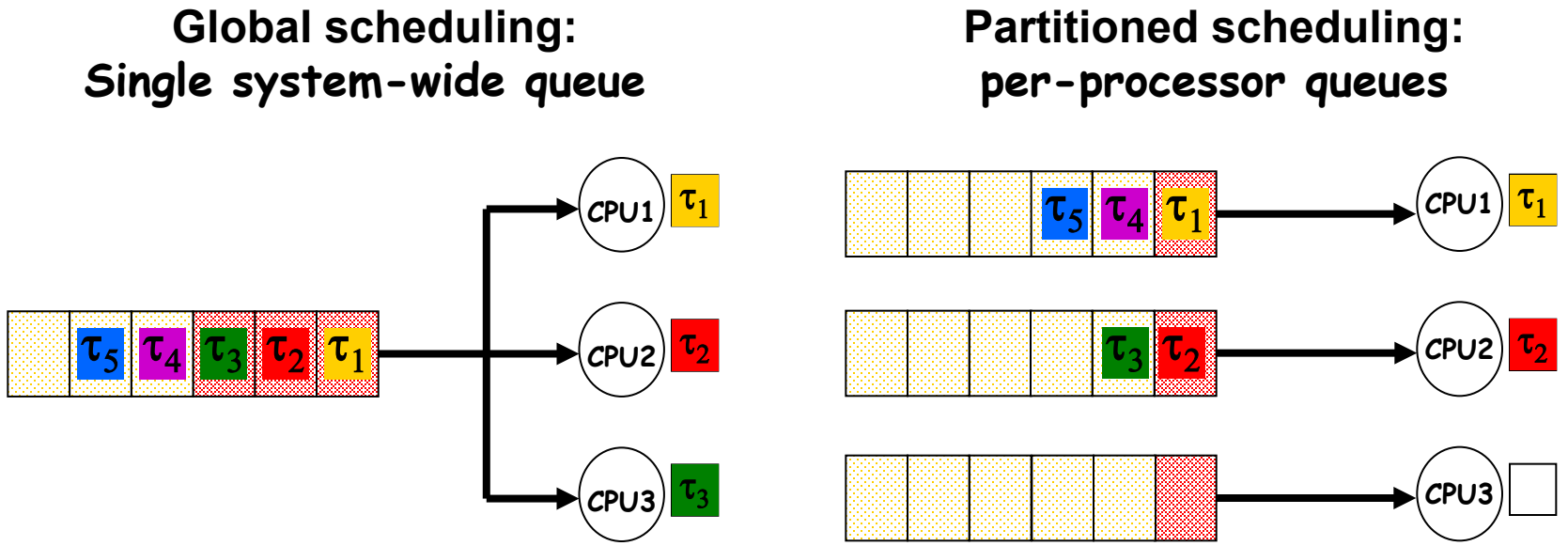
Multiprocessor Models

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 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

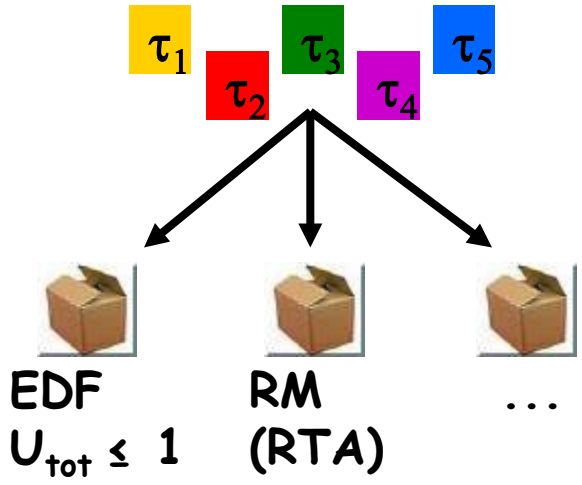
Partitioned Scheduling

- 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

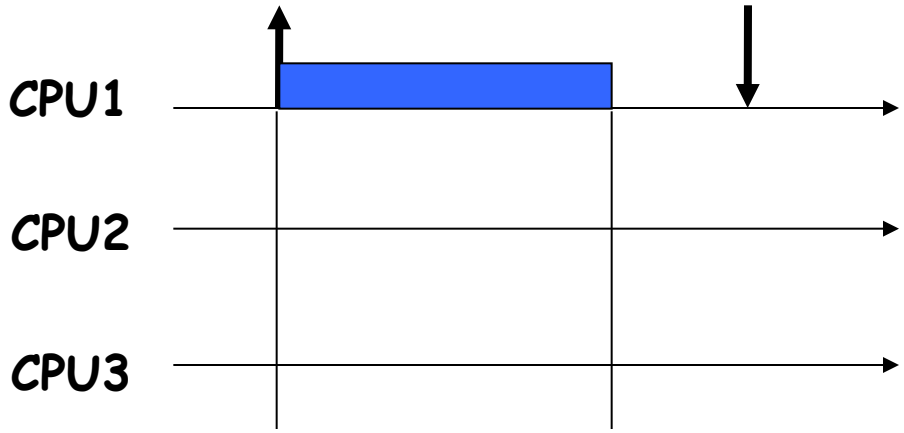
- Bin-packing algorithms:
  - The problem concerns packing objects of varying sizes in boxes (”bins”) with some optimization objective, e.g., minimizing number of used boxes (best-fit), or minimizing the maximum workload for each box (worst-fit)
- Application to multiprocessor scheduling:
  - Bins are 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

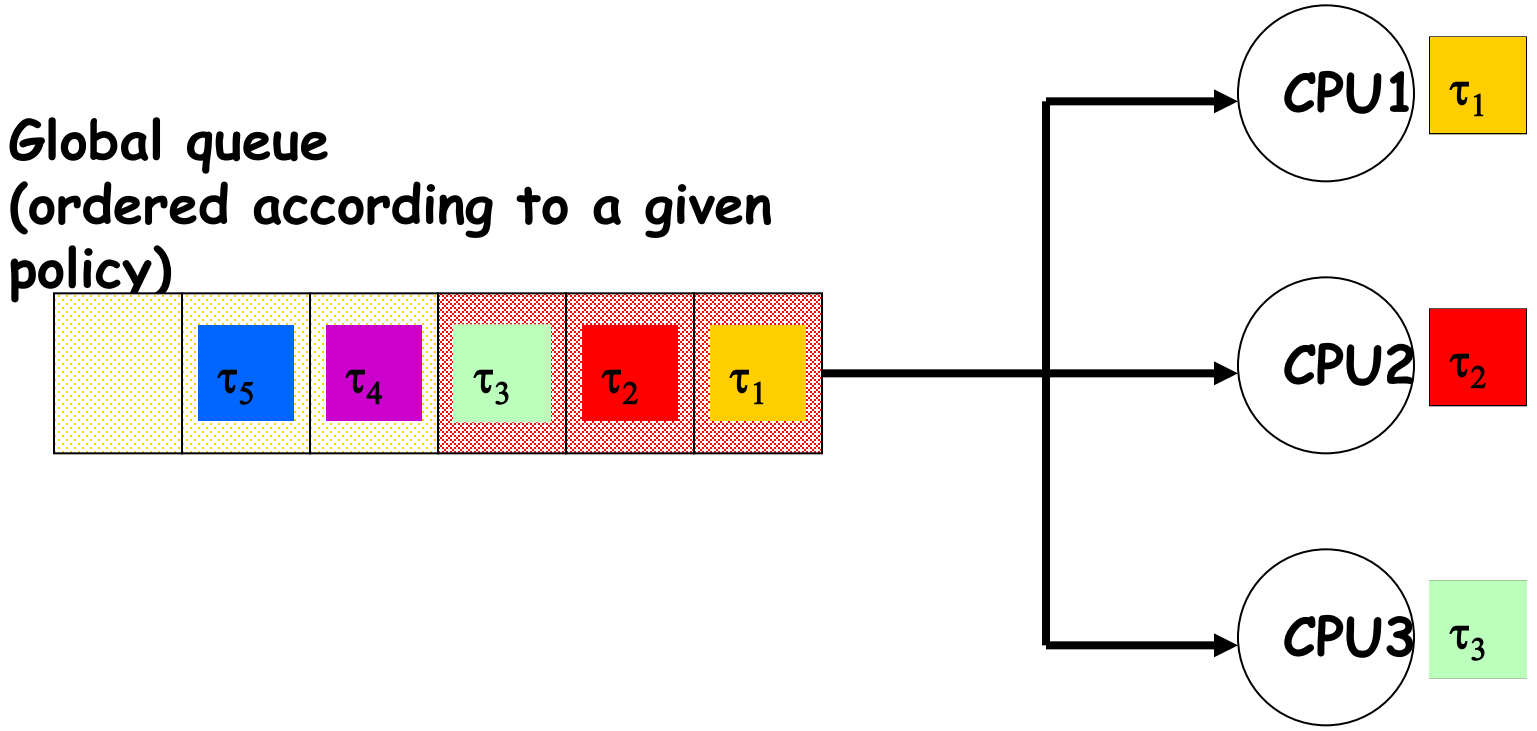
- Identical multiprocessors
- Work-conserving:
  - At each instant, the highest-priority jobs that are eligible to execute are selected for execution upon the available processors
  - No processor is ever idle when the ready queue is non-empty
- Preemption and Migration support
  - For global schedulers, a preempted task can resume its execution on a different processor with 0 overhead
  - Cost of preemption/migration integrated into task WCET
- No job-level parallelism
  - the same job cannot be *simultaneously* executed on more than one processor

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]

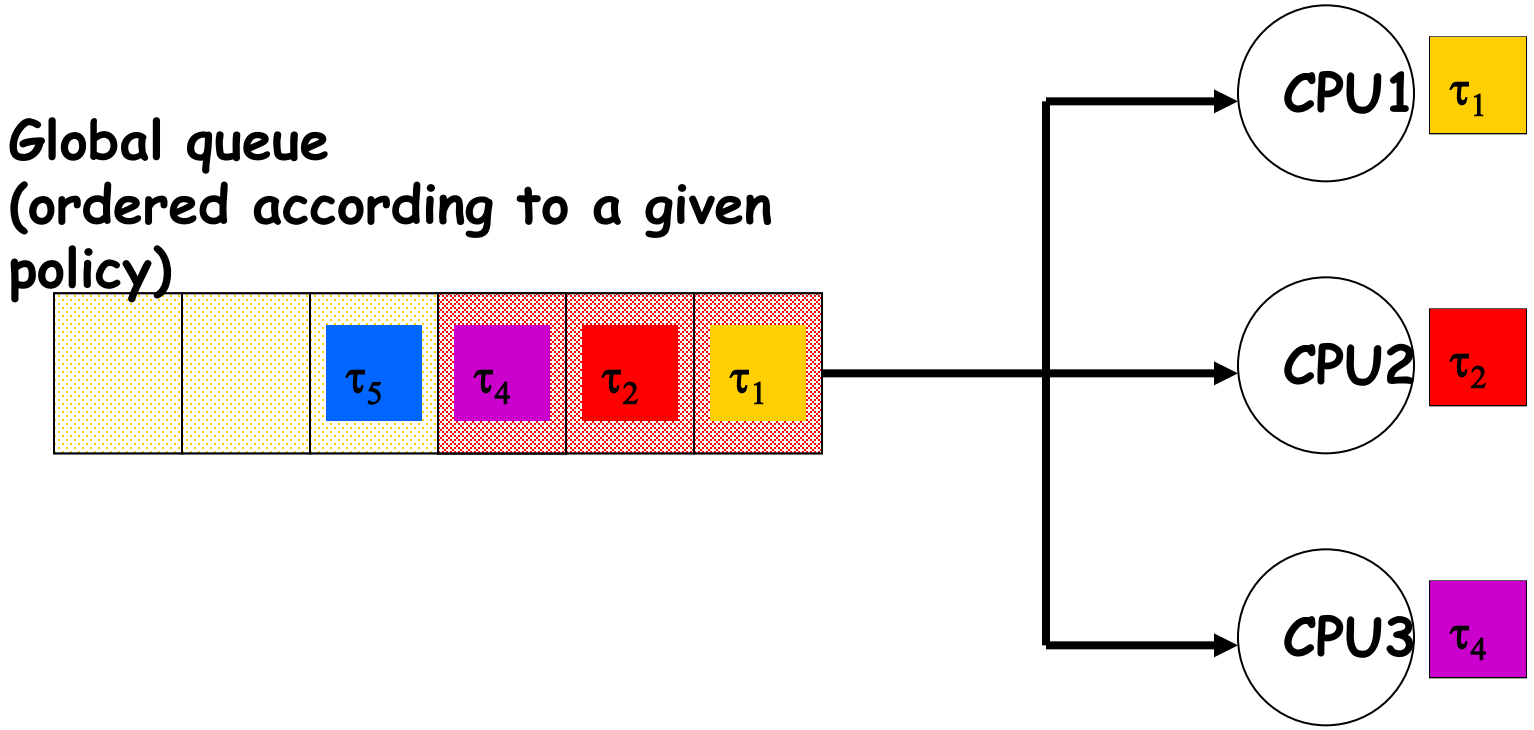


Global scheduling example



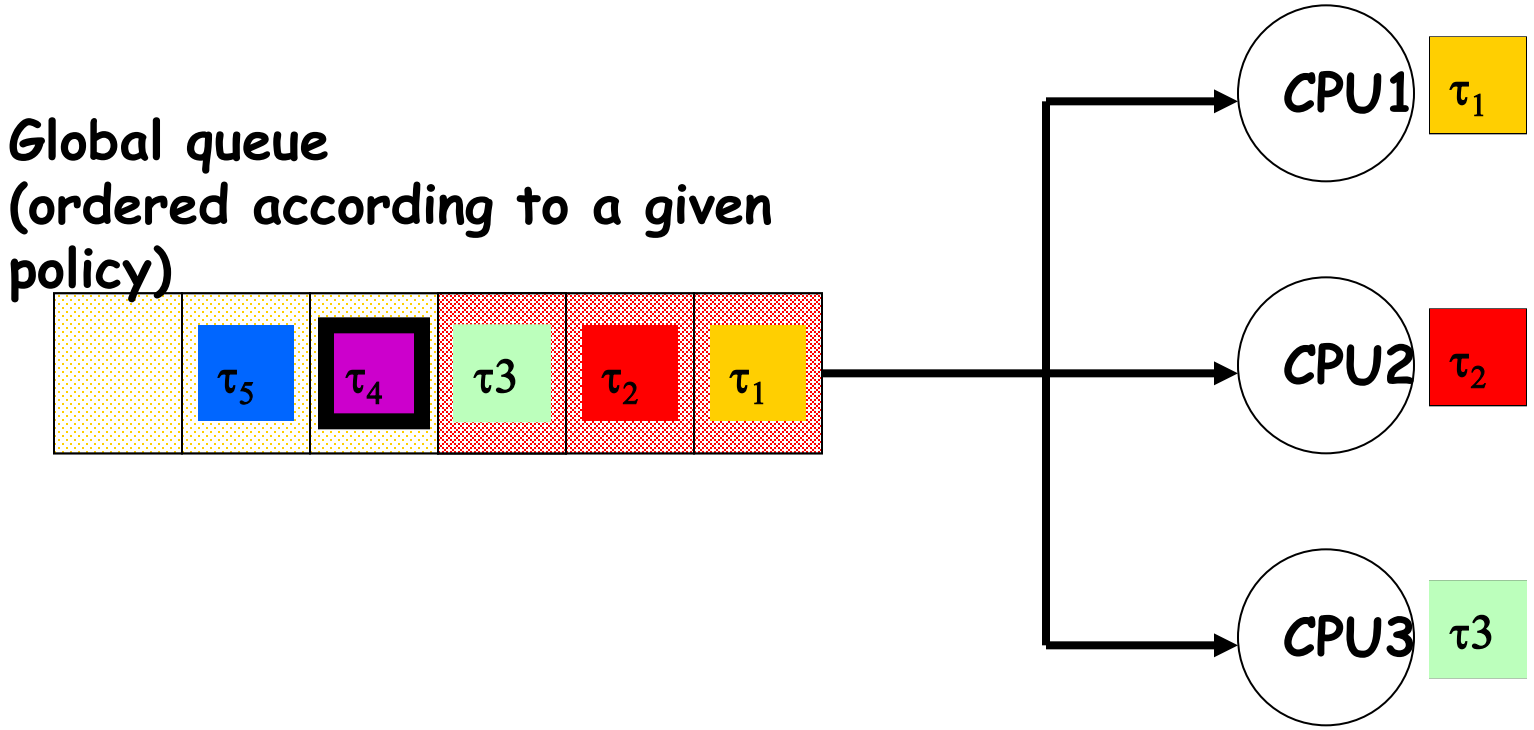
The first m jobs in the queue are scheduled upon the m CPUs

Global scheduling example



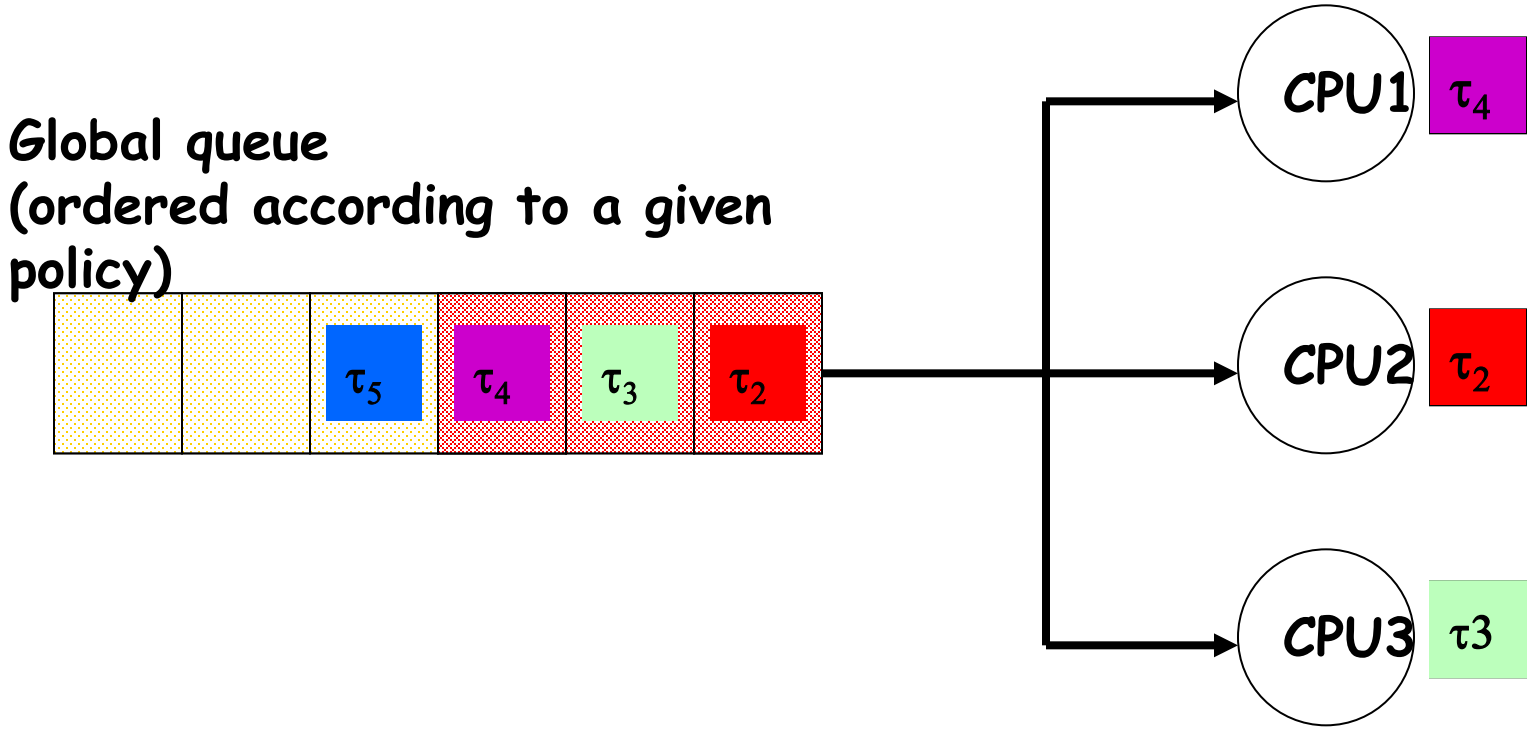
When a job τ_3 finishes its execution, the next job in the queue τ_4 is scheduled on the available CPU

Global scheduling example



When a new higher priority job τ_3 arrives, it preempts the job with lowest priority τ_4 among the executing ones

Global scheduling example



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

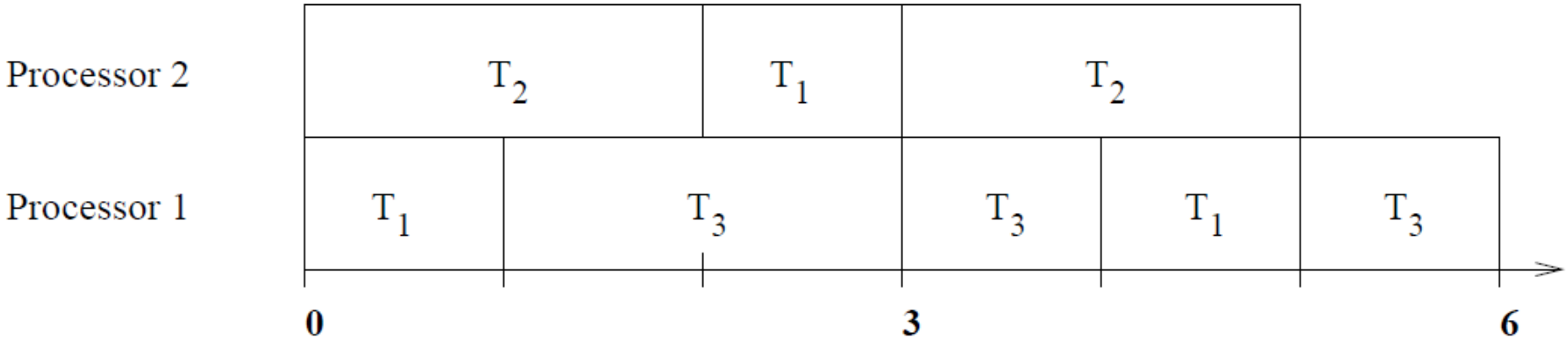
Global vs. Partitioned

- 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
- Global FP scheduling is feasible with priority assignment $p_1>p_2>p_3$ (or $p_2>p_1>p_3$)
- Partitioned scheduling is not feasible, since assigning any two tasks to the same processor will cause the utilization to exceed 1.



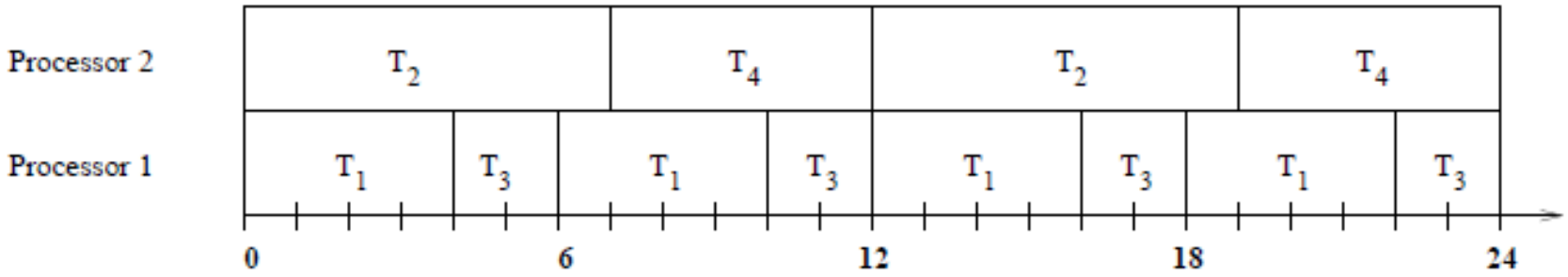
A feasible execution trace under global scheduling

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)

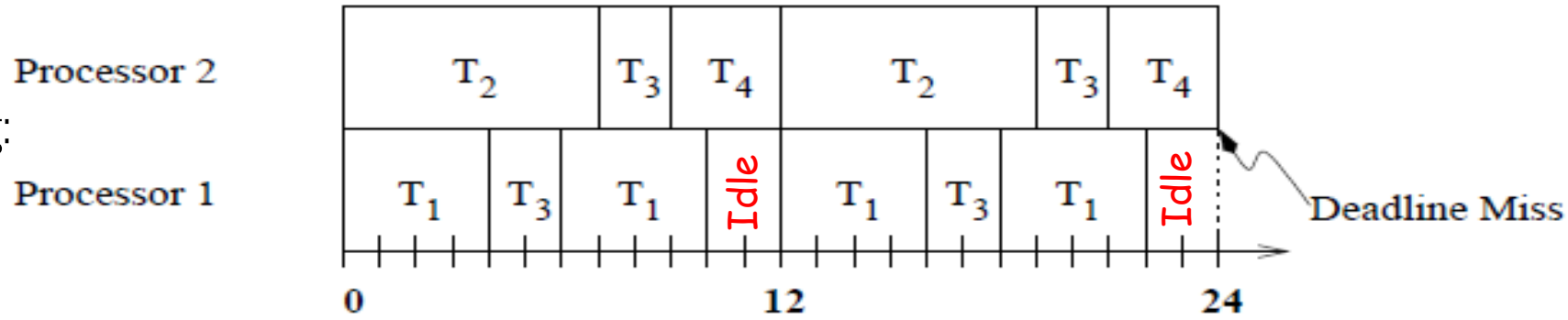
- A taskset schedulable with partitioned scheduling, but not global scheduling

- Tasks T1, T3 assigned to Processor 1; T2, T4 assigned to Processor 2
- Partitioned FP scheduling with RM priority assignment ($p_1>p_2>p_3>p_4$) is feasible. Both processors have utilization 1.0, and harmonic task periods



A feasible execution trace under partitioned scheduling

- Global FP scheduling with RM priority assignment $p_1>p_2>p_3>p_4$
- Compare to partitioned scheduling: the difference is at time 7, when global FP scheduling must run T3, which has higher priority than T4; this causes Processor 1 to be idle in time intervals [10,12] and [22,24], since only one task T4 can be run



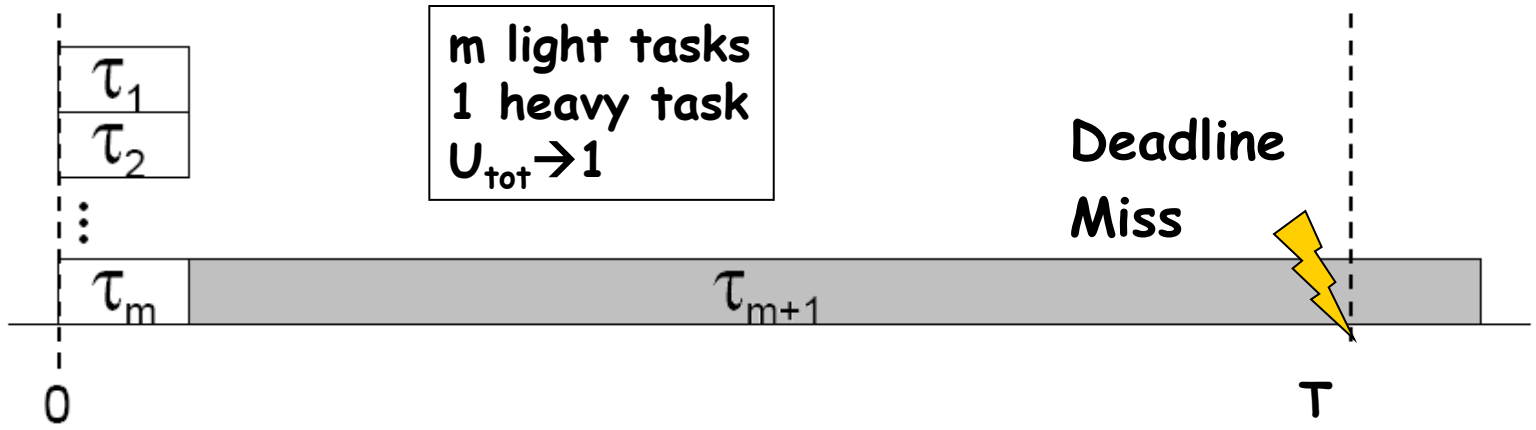
A feasible execution trace under partitioned scheduling

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
- Dependence on relative priority ordering (omitted)
  - Changing the relative priority ordering among higher-priority tasks may affect schedulability for a lower-priority task.
- Hard-to-find critical instant (omitted)
  - A critical instant does not always occur when a task arrives at the same time as all its higher-priority tasks.

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 $C_i = T$, $T_i = D_i = T$;
- 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 tasks with higher utilization



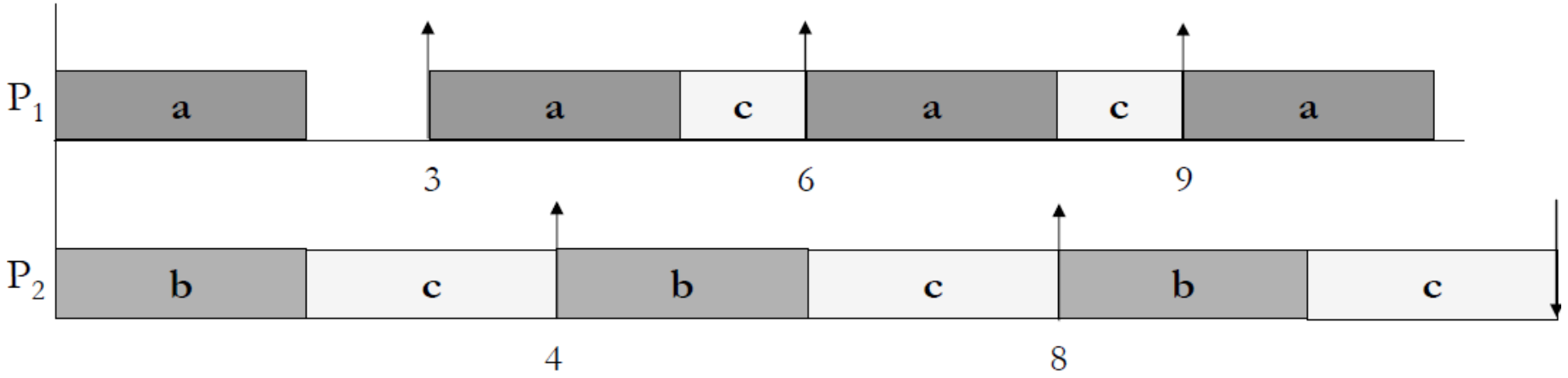
MP Scheduling Anomalies

- 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

Anomaly 1

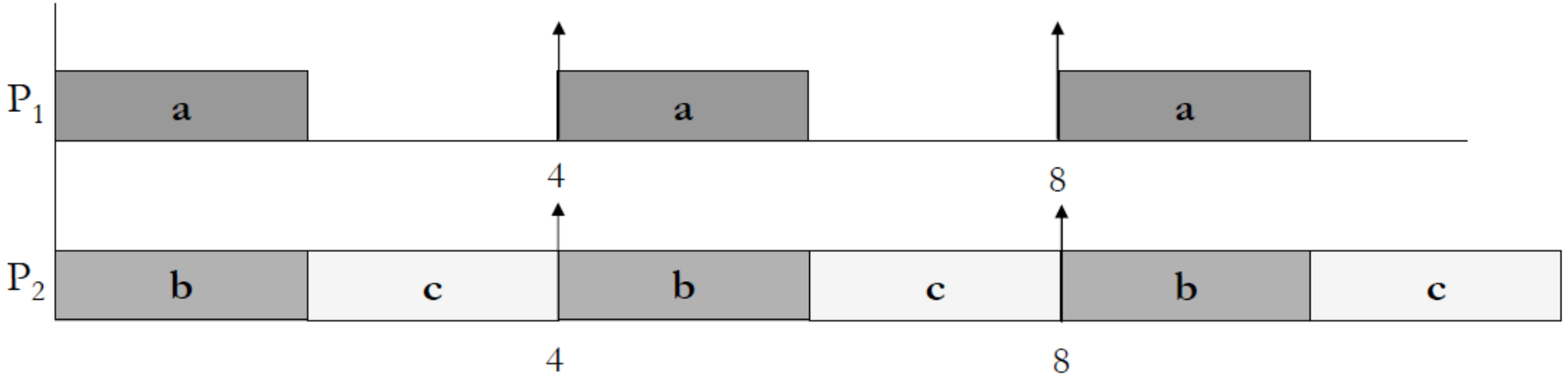
Task	T	D	C	U
a	3	3	2	0.67
b	4	4	2	0.50
c	12	12	8	0.67

m = 2 processors and $\sum U_i = 1.83$ but task c is *saturated* because $C_c + I_c = D_c$ and thus any increase in C_c would makes it unschedulable



Anomaly 1

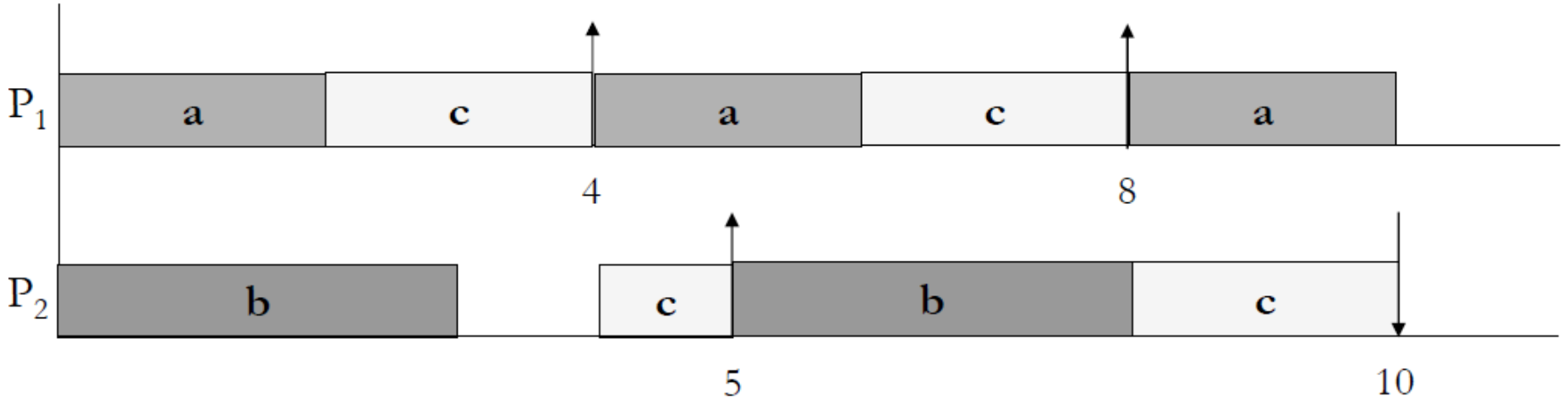
- If we reduce T_a to 4 we *decrease* system load to $\sum U_i = 1.67$
- But then I_c *increases* from 4 to 6 and task c misses its deadline (!)



Anomaly 2

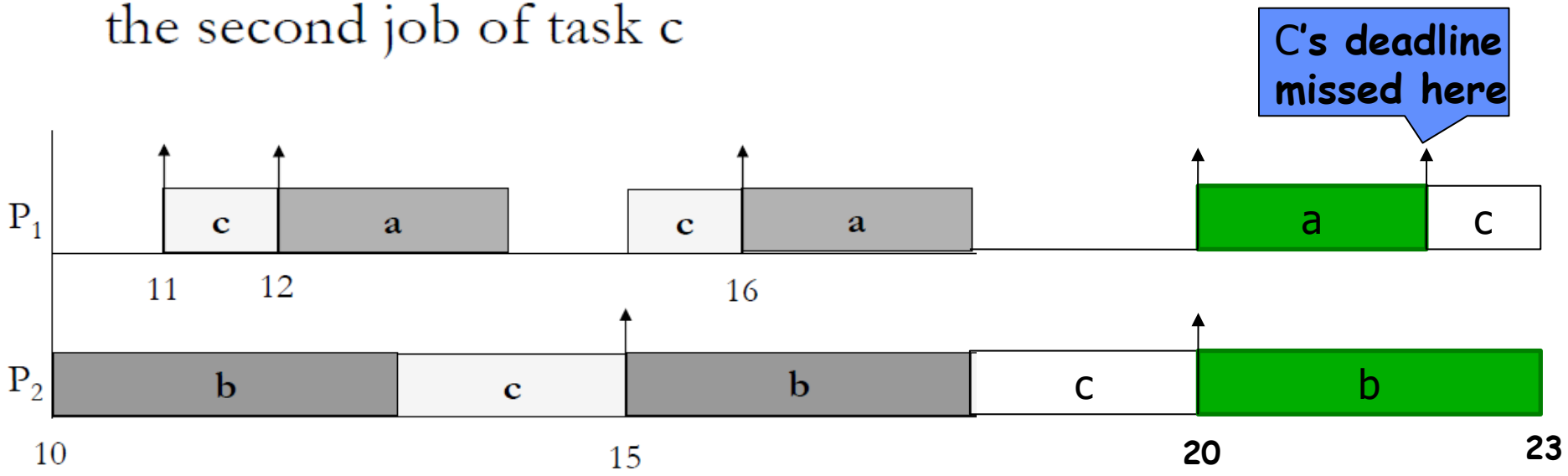
Task	T	D	C	U
a	4	4	2	0.5
b	5	5	3	0.6
c	10	10	7	0.7

m = 2 processors and $\sum U_i = 1.8$ but task c is *saturated*



Anomaly 2

- If we extend T_c to 11 we *decrease* system load to $\sum U_i = 1.74$
- But then I_c *increases* from 3 to 5 (!) as becomes visible in the second job of task c



- Another example where the critical instant for task c does not occur at time 0 when it arrives at the same time as all its higher-priority tasks.