CSC 112: Computer Operating Systems
Lecture 6

Real-Time Scheduling II

Department of Computer Science,

Hofstra University

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

Outline

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

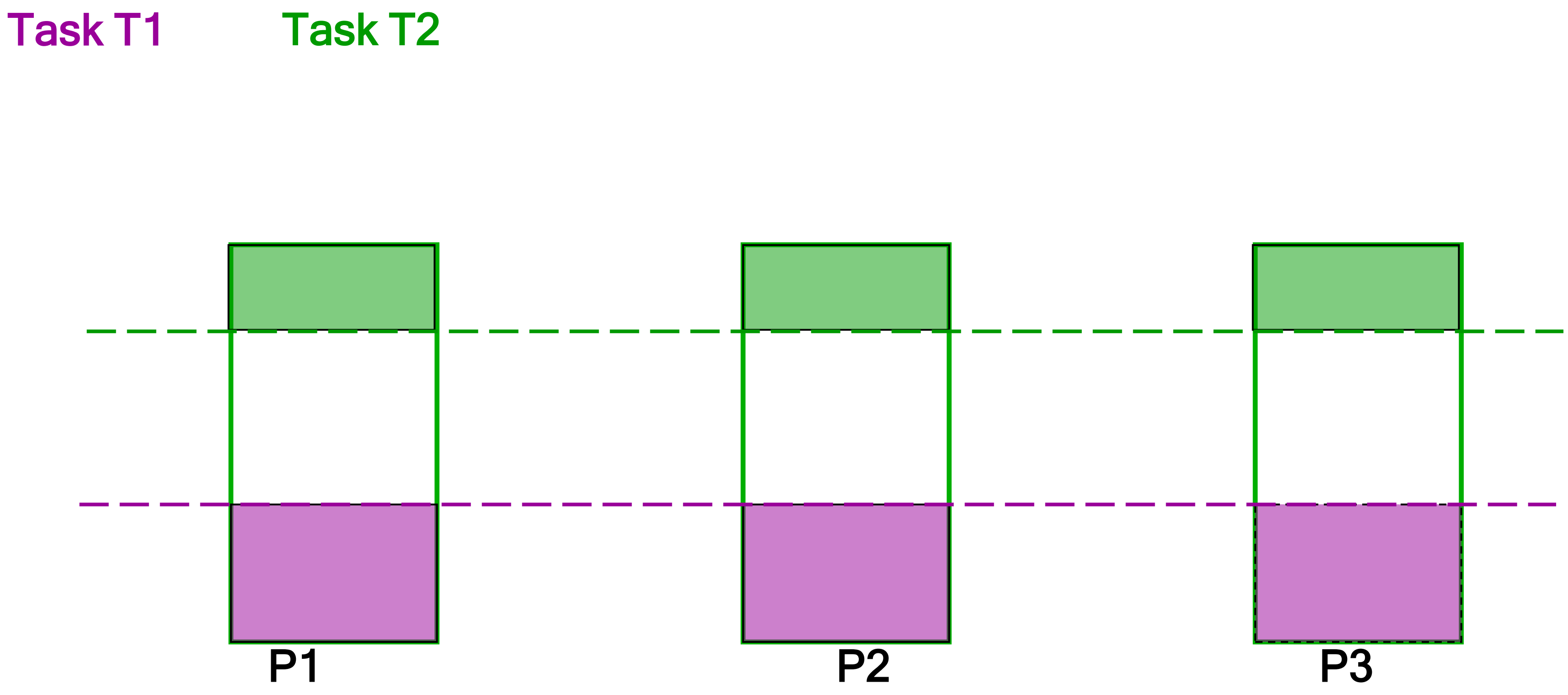
Multiprocessor Scheduling

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

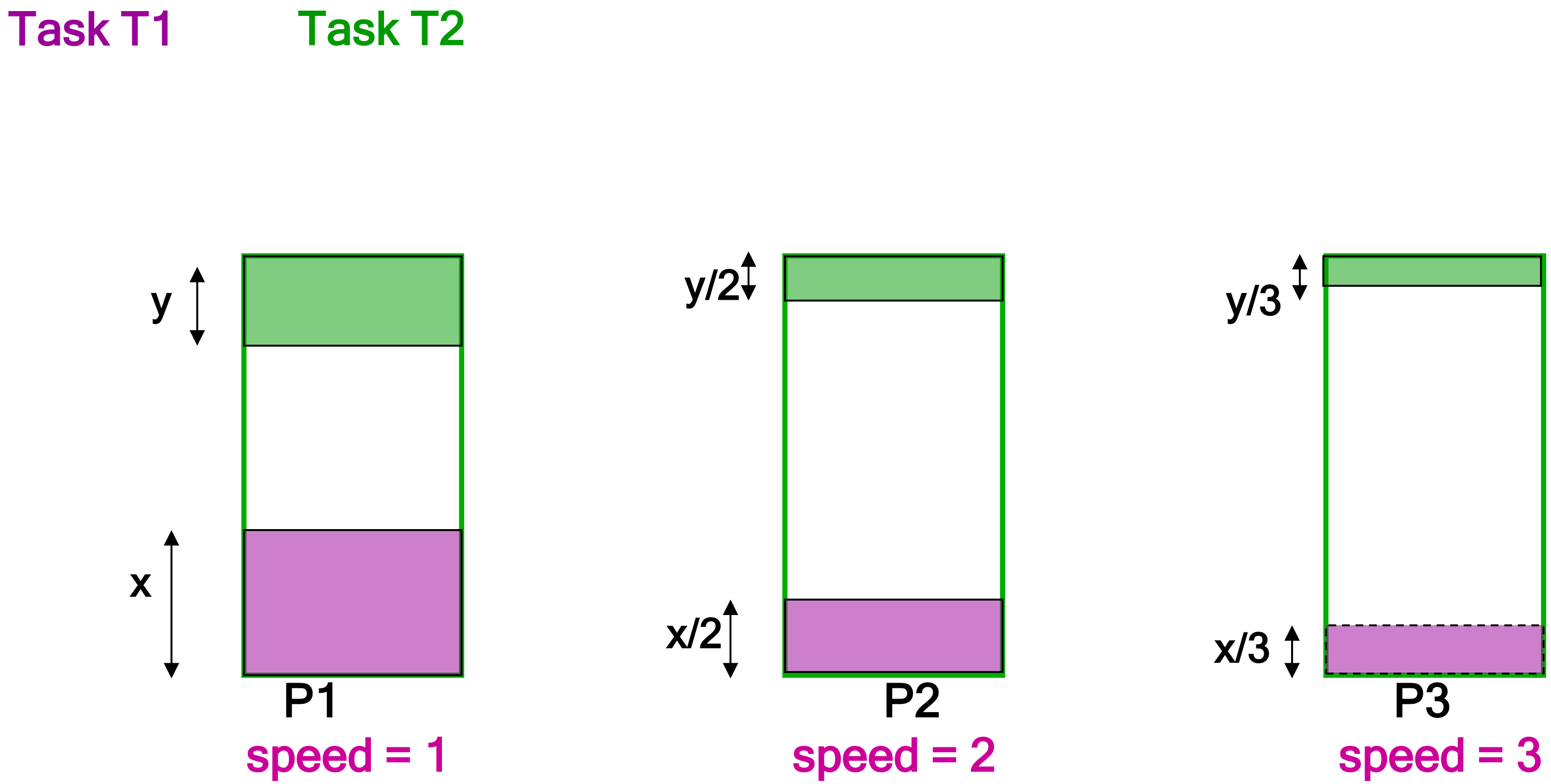
Multiprocessor Models

Identical multiprocessors: each processor has the same speed



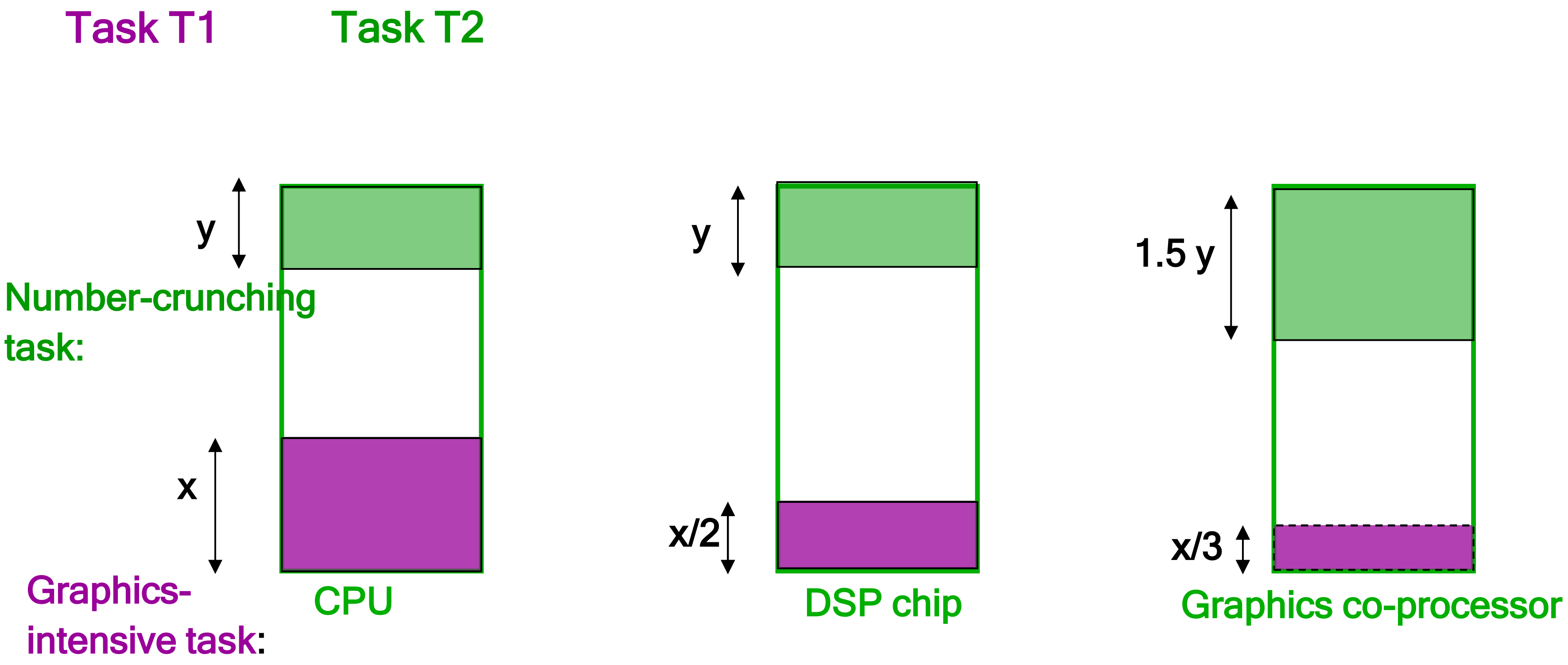
Multiprocessor Models

Uniform multiprocessors: different processors have different speeds



Multiprocessor Models

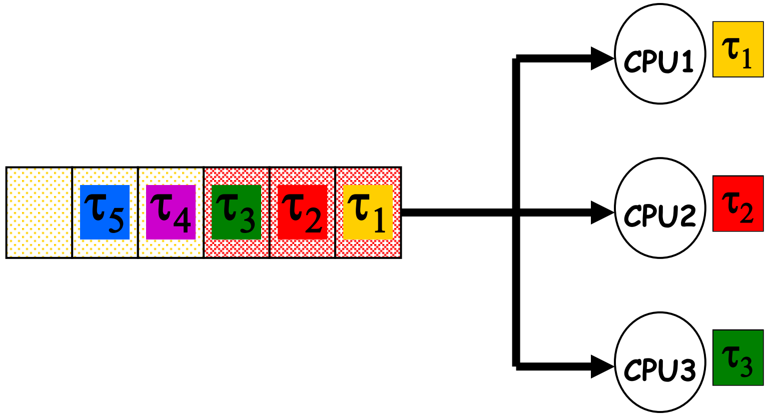
Heterogeneous multiprocessors: each (task, processor) pair may have a different relative speed, due to specialized processor architectures



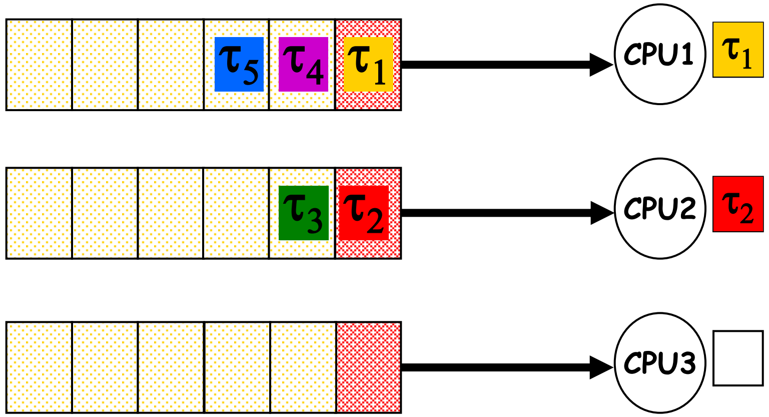
Global vs partitioned scheduling

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

- Partitioned Scheduling
- Pros:
  - Mature scheduling framework
  - Uniprocessor scheduling theory 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
- Cons:
  - No runtime load-balancing; surplus CPU time cannot be shared among processors

Partitioned Scheduling

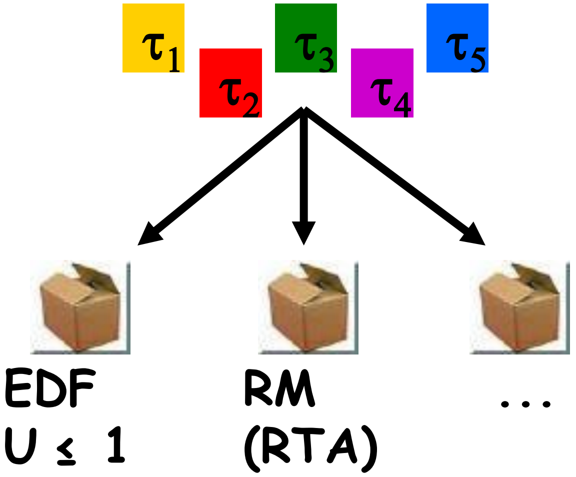
- Scheduling problem reduces to:

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

NP-hard **Well-known**

⇓

Various heuristic algorithms
First Fit (FF)
Best Fit (BF)
Worst Fit (WF)
Next Fit (NF)
…



EDF
U ≤ 1

RM
(RTA)

...

Partitioned Scheduling

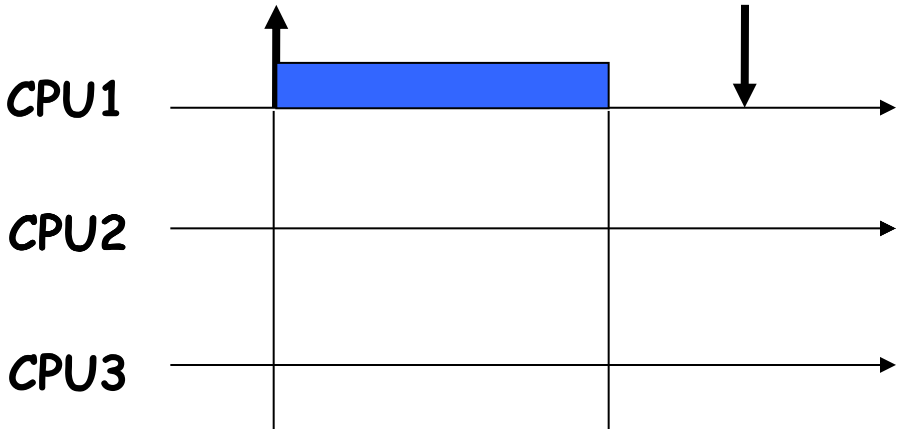
- Bin-packing algorithms:
  - The problem concerns packing objects of varying sizes in boxes (”bins”) with some optimization objective, e.g., minimizing number of used boxes (best-fit), or minimizing the maximum workload for each box (worst-fit)
- Application to multiprocessor scheduling:
  - Bins are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

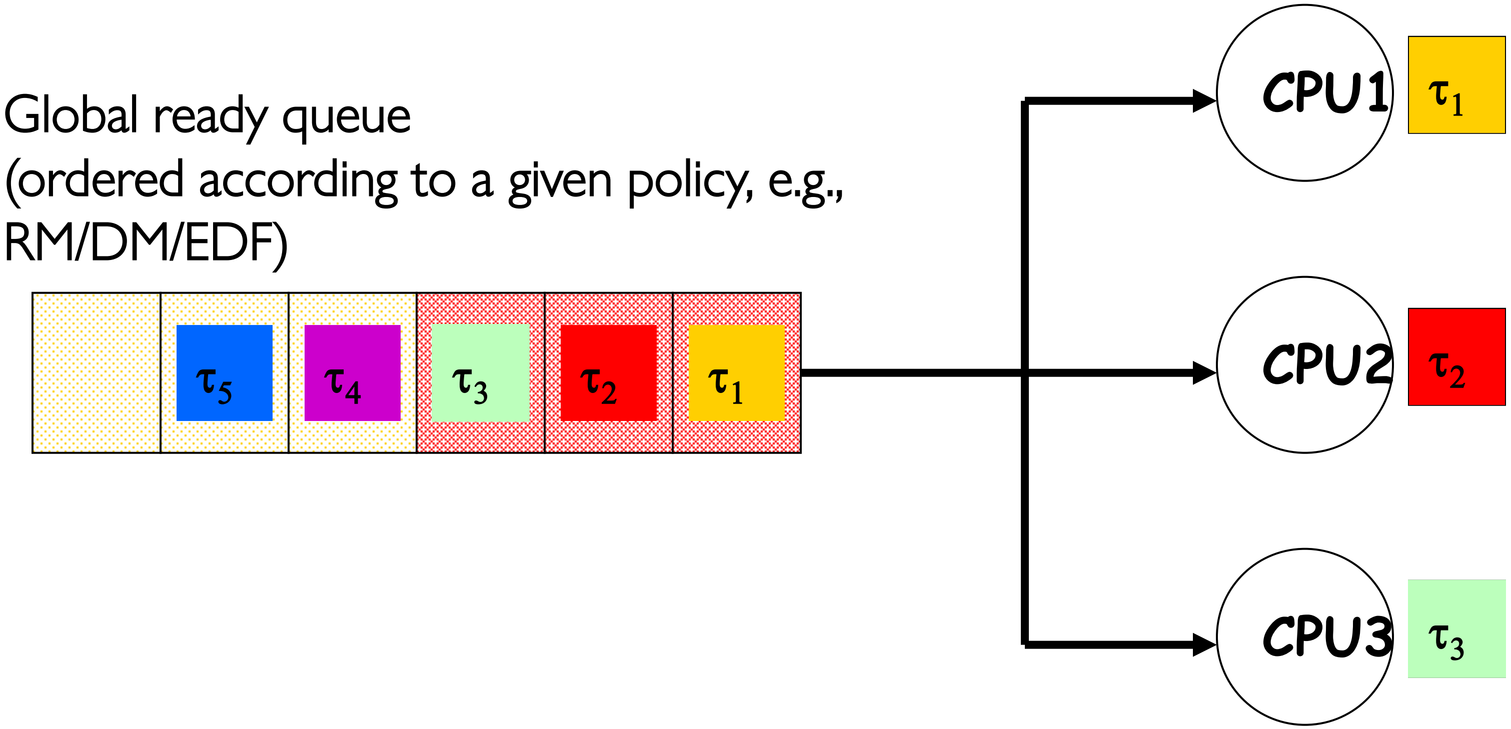
Source of Difficulty

- 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]



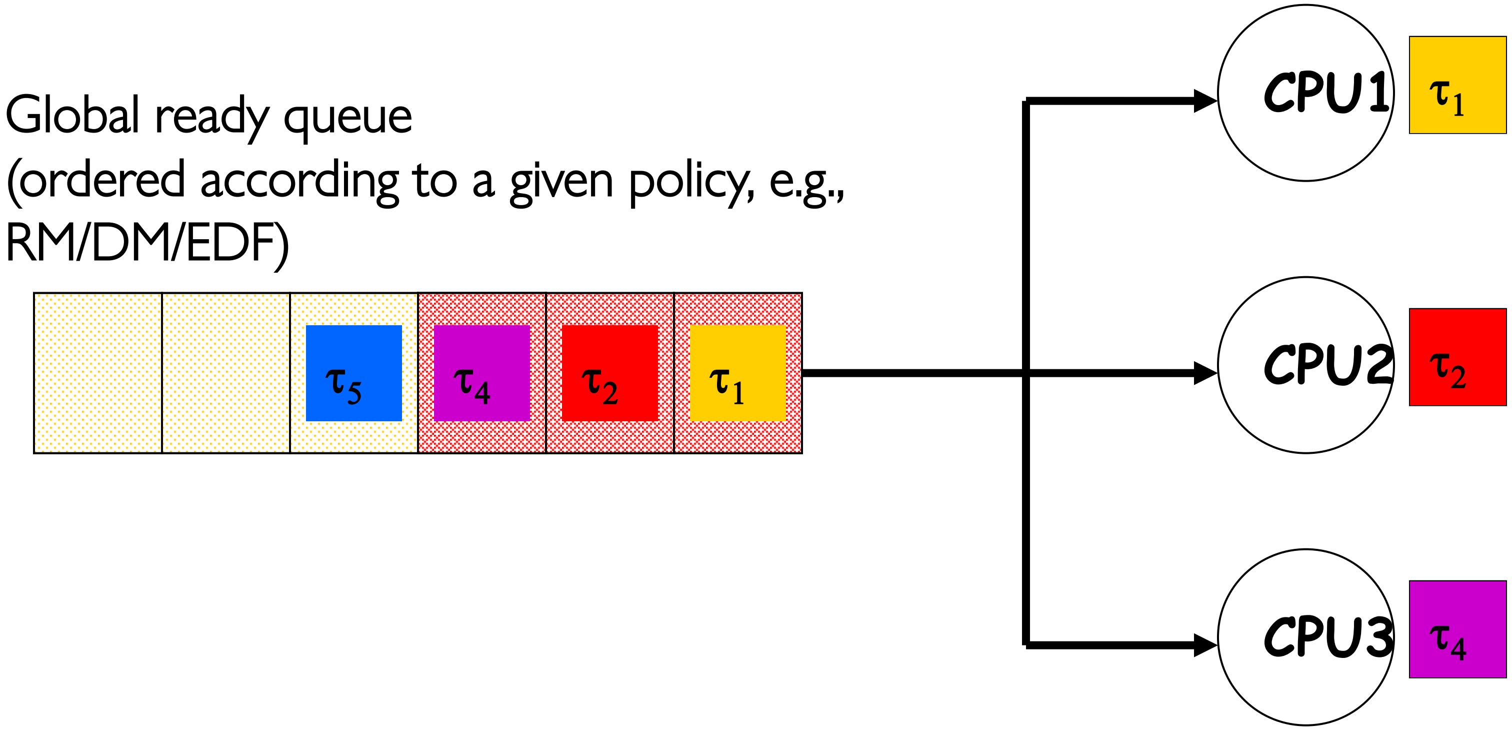
Global scheduling example

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



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

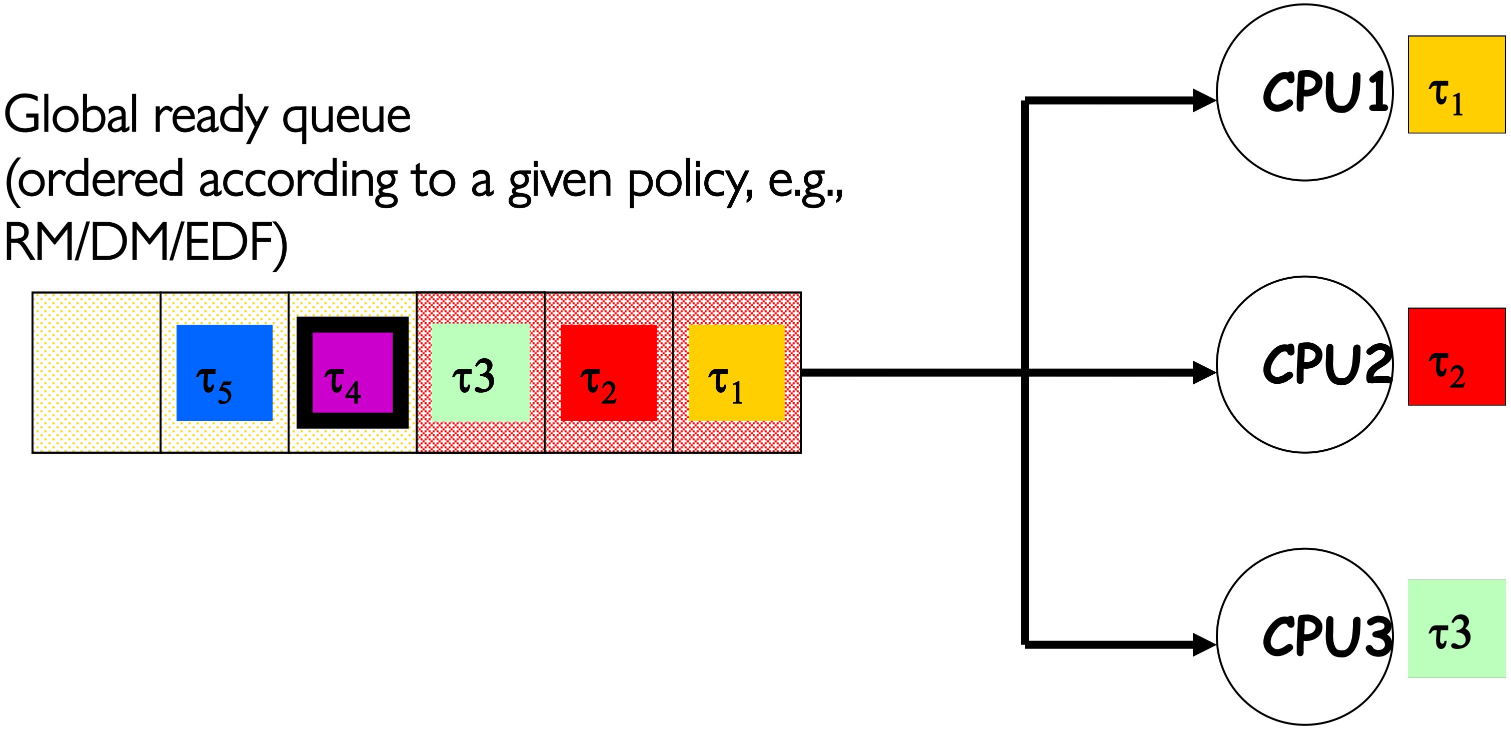
Global scheduling example



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

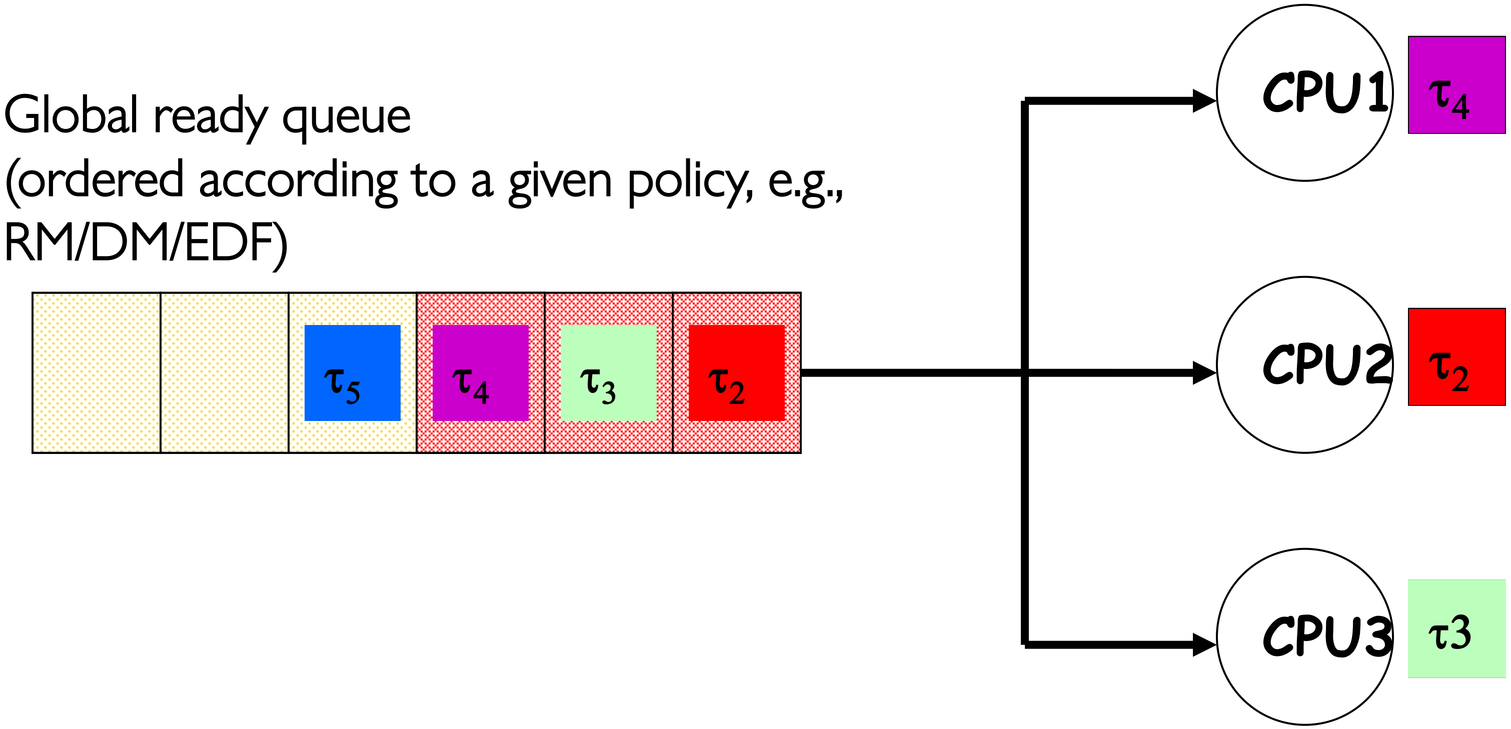
Global scheduling example

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



When a new higher-priority job τ_3 arrives in its next period T_3, 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

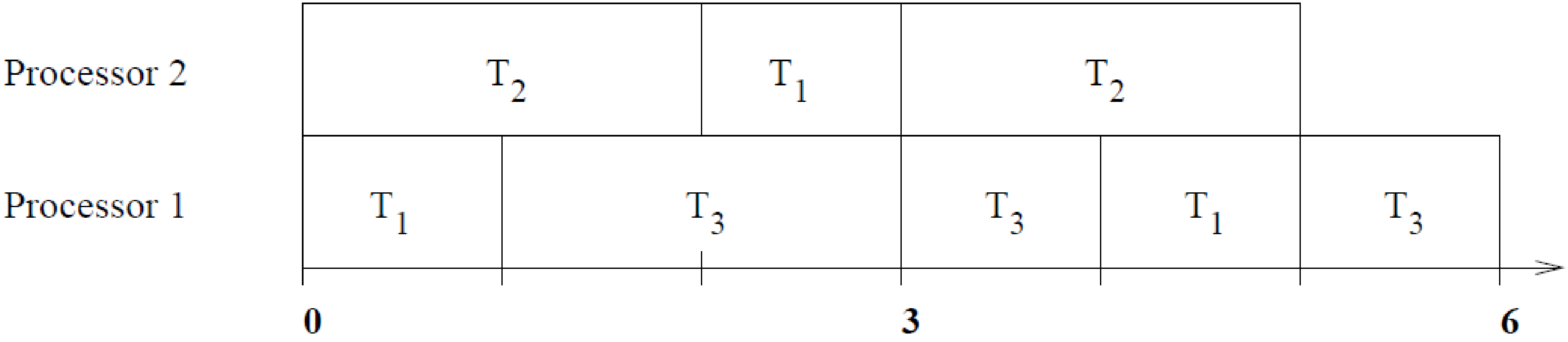
Global vs. Partitioned

- 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. 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



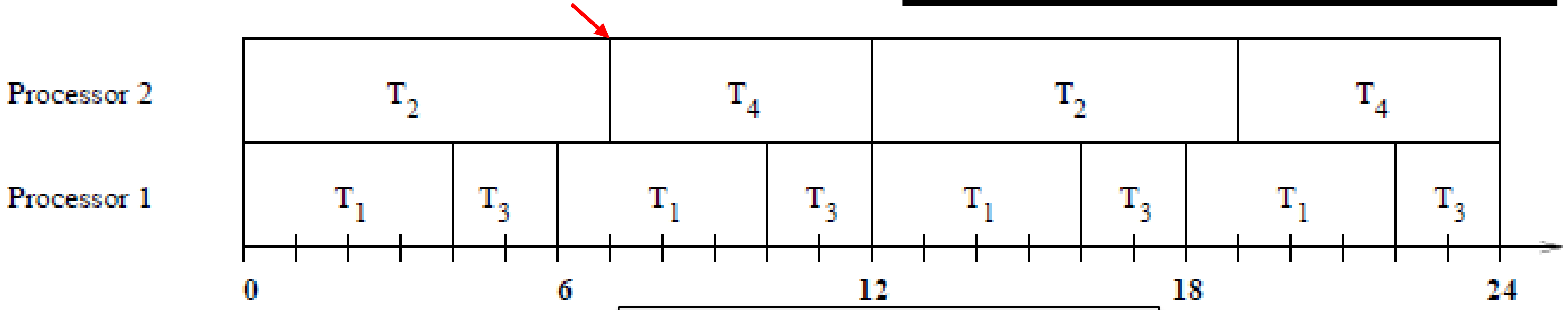
A feasible execution trace under global scheduling
T2 runs always on P2, T3 runs always on P1, T1 runs on both P1 and P2 with task migration across different periods

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)

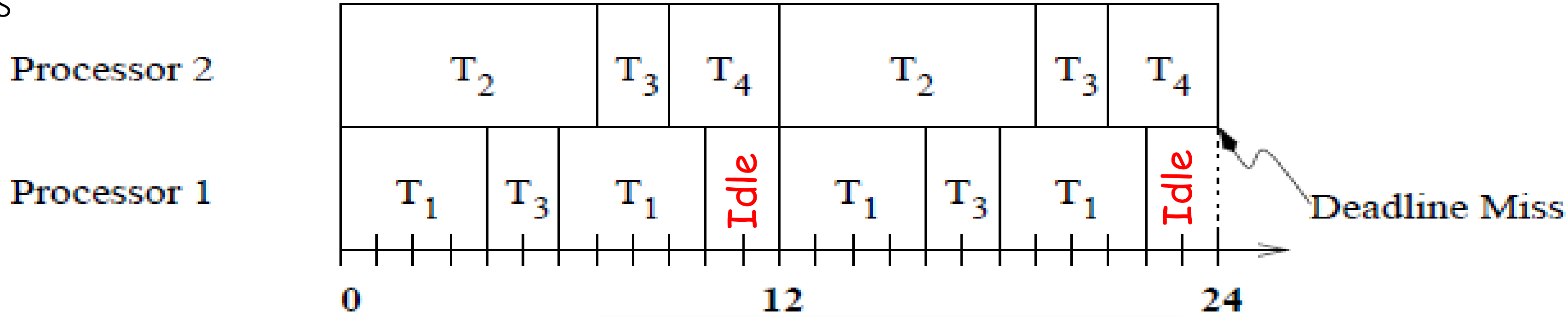
- A taskset schedulable with partitioned scheduling, but not global scheduling. System utilization $U = \frac{4}{6} + \frac{7}{12} + \frac{4}{12} + \frac{10}{24} = 2.0$, hence the two processors must be fully utilized with no possible idle intervals
- Partitioned FP scheduling with RM priority assignment ($p_1 > p_2 > p_3 > p_4$) 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



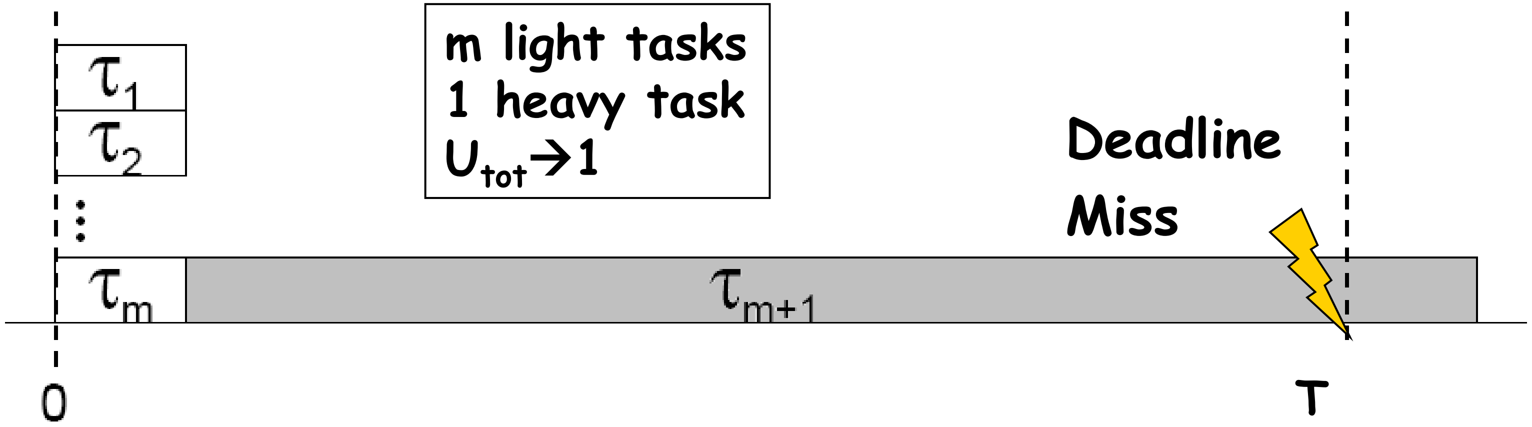
An infeasible execution trace under global scheduling

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

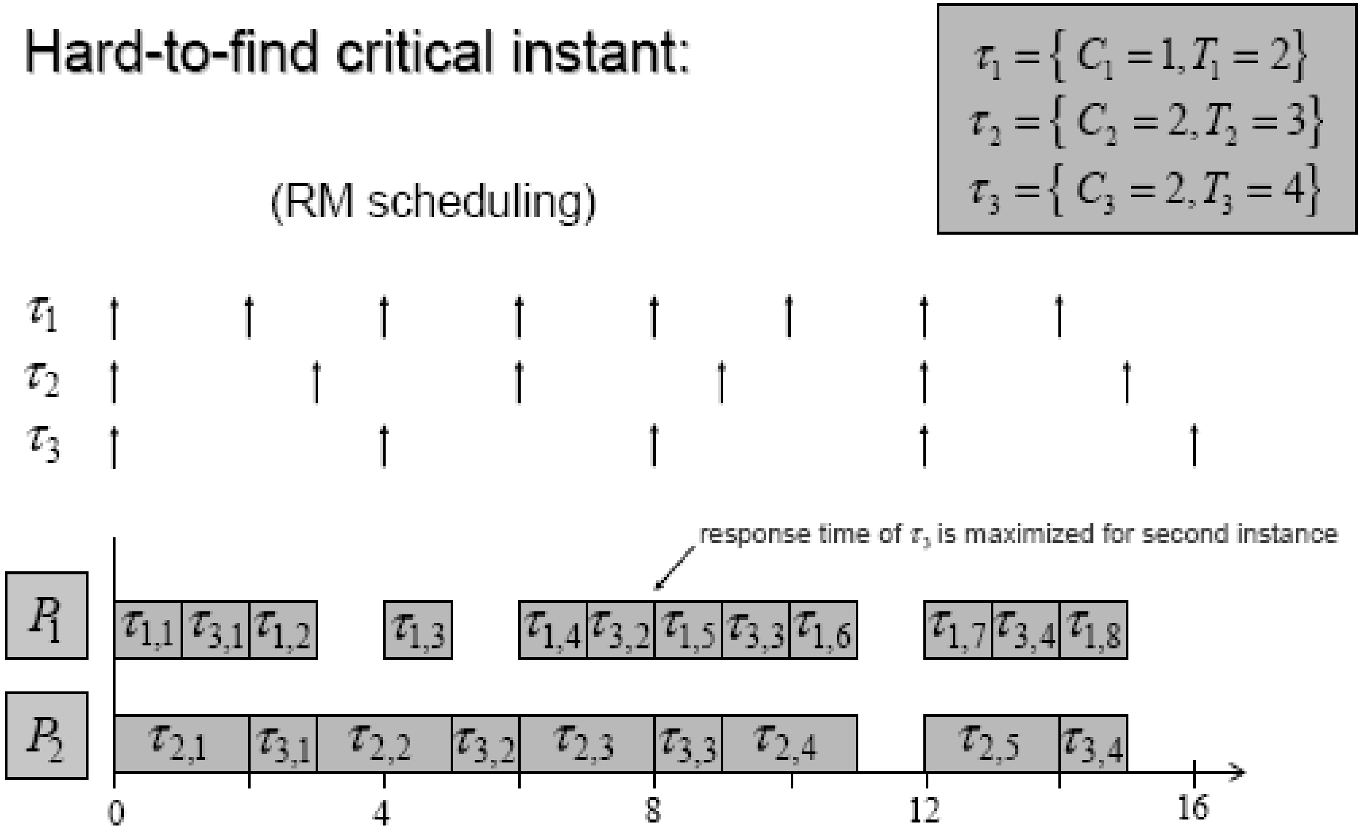
Dhall's effect

- Global RM/DM/EDF can fail at very low utilization
- Example: m processors, n=m+1 tasks. Tasks $\tau_1, \dots, \tau_m$ 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 $\tau_1, \dots, \tau_m$ must run on m processors starting at time 0, causing τ_{m+1} to finish at time $T + 1$, miss its deadline at T
- 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)” in L6-RTScheduling I)
- This is no longer true for multiprocessor scheduling, as the worst-case interference for a task does not always occur at the critical instant time 0, when all tasks are initially released 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)

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

response time of τ_3 is maximized for second instance

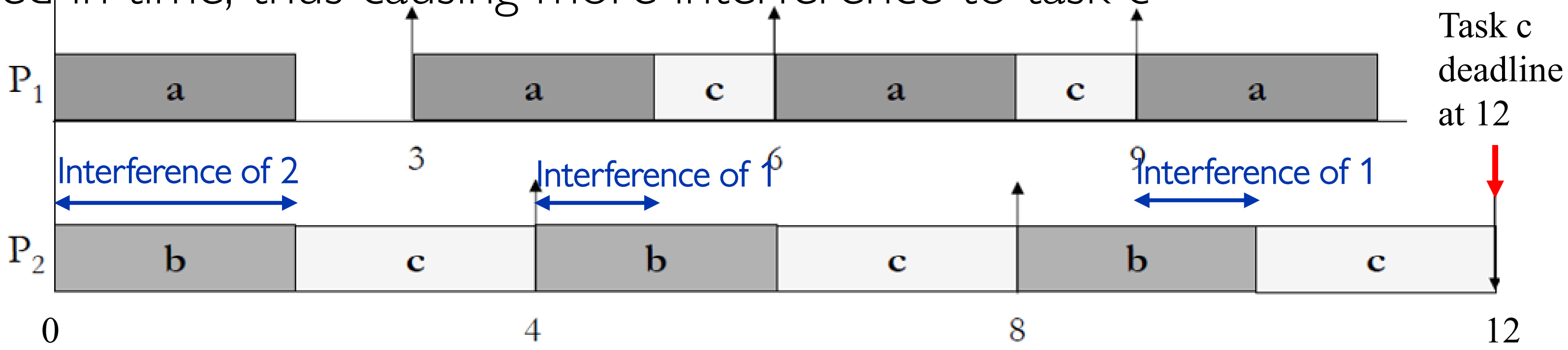
MP Scheduling Anomalies

- 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

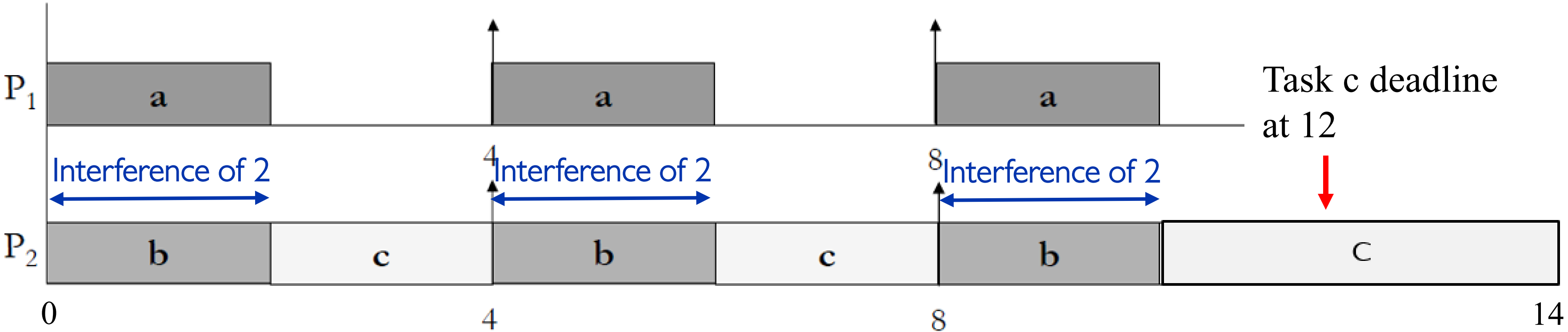
Scheduling Anomaly Example 1

- 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



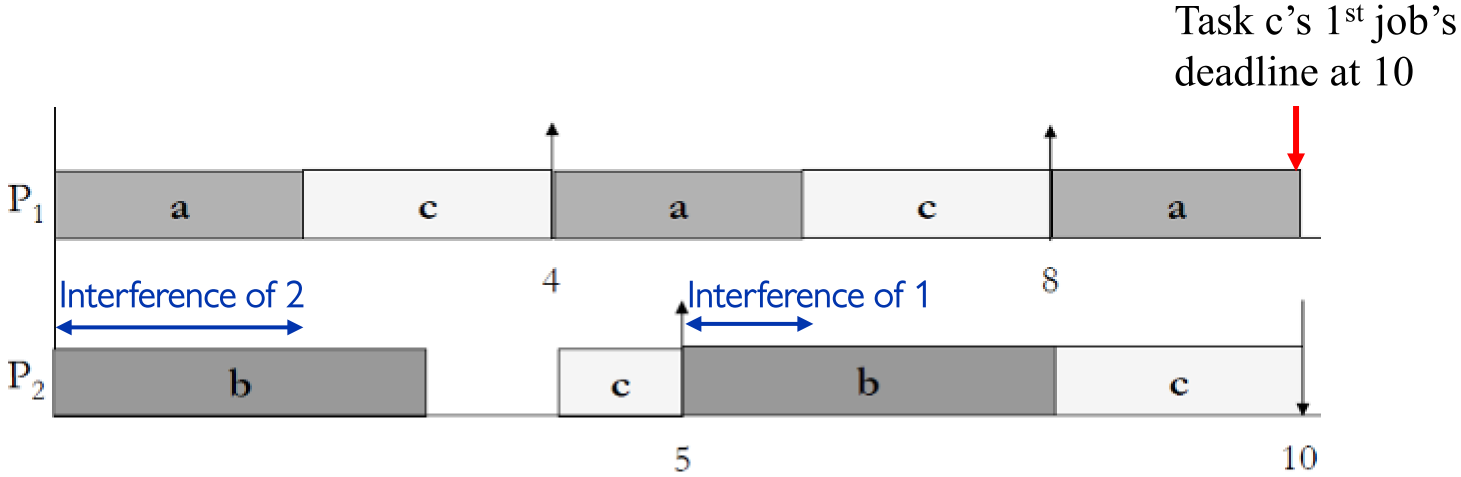
Task	T=D	C	Util	Prio
a	4	2	0.5	H
b	4	2	0.5	M
c	12	8	0.67	L



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


Task c's 1st job's deadline at 10

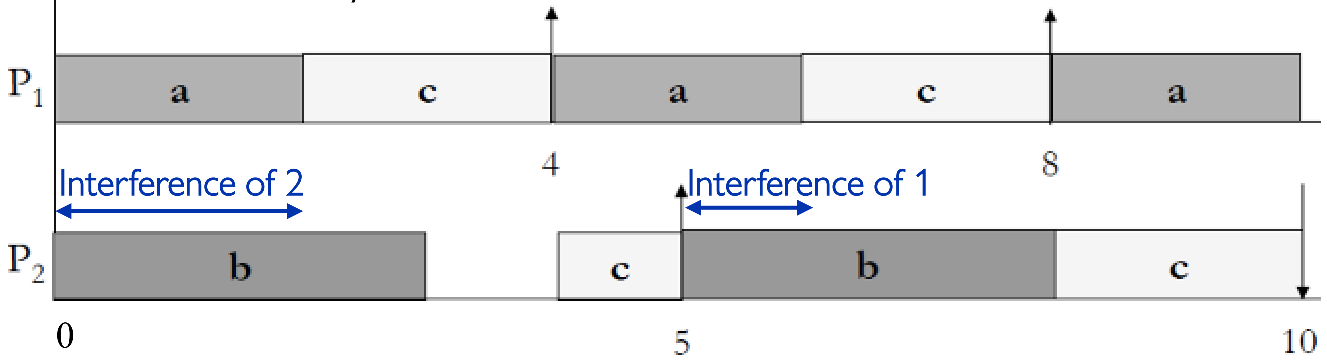
Interference of 2

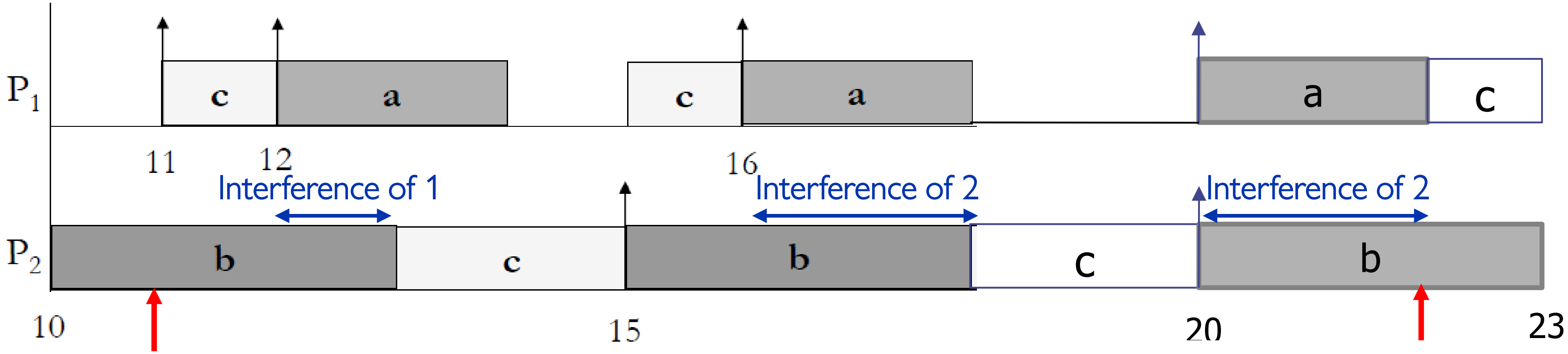
Interference of 1

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 $C_c + I_c = 7 + 3 = 10 \le D_c = 11$, 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





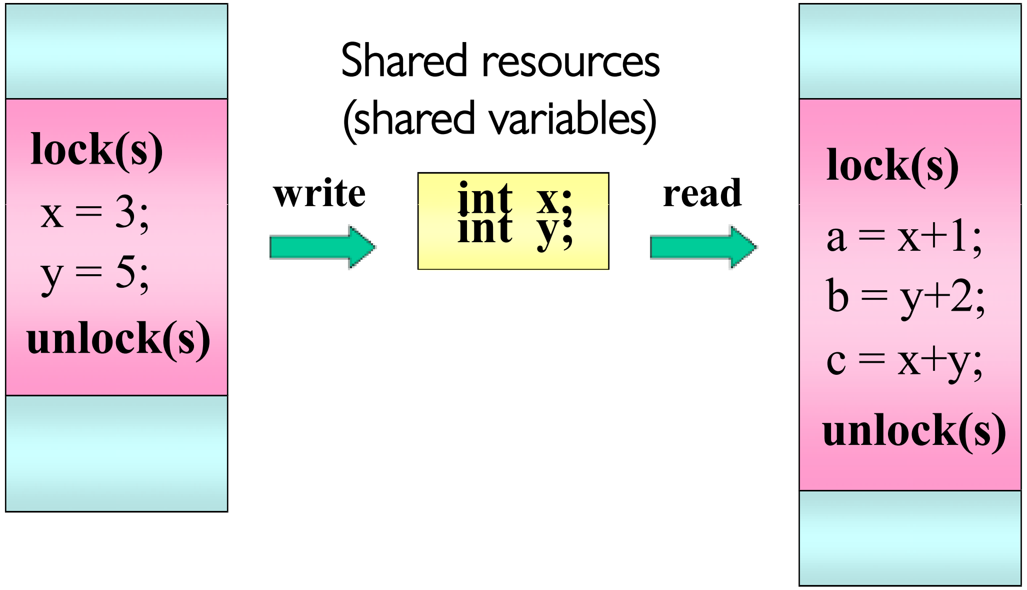
Task c's 1st job's deadline at 11

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

Resource Synchronization Protocols (for Fixed-Priority Scheduling)

Resource Sharing

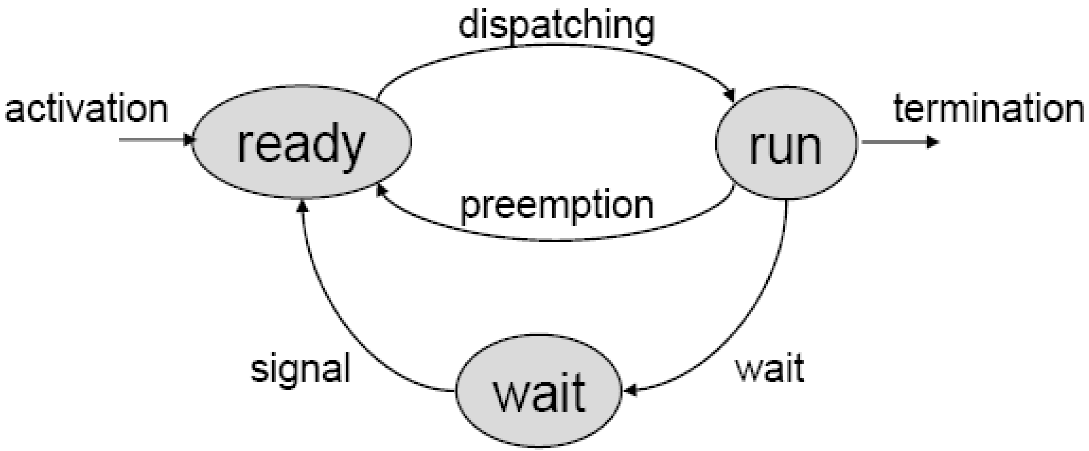
- When two tasks access shared resources (variables), mutexes (or binary semaphores) are used to protect critical sections. Each Critical Section (CS) must begin with lock(s) and end with unlock(s)
- A task waiting for a shared resource is blocked on that resource. Otherwise, it proceeds by entering the critical section and holds the resource
- Tasks blocked on the same resource are kept in a queue. When a running task invokes lock(s) when s is already locked, it enters the waiting state, until another task unlocks s



```
lock(s)
 x = 3;
 y = 5;
unlock(s)
```

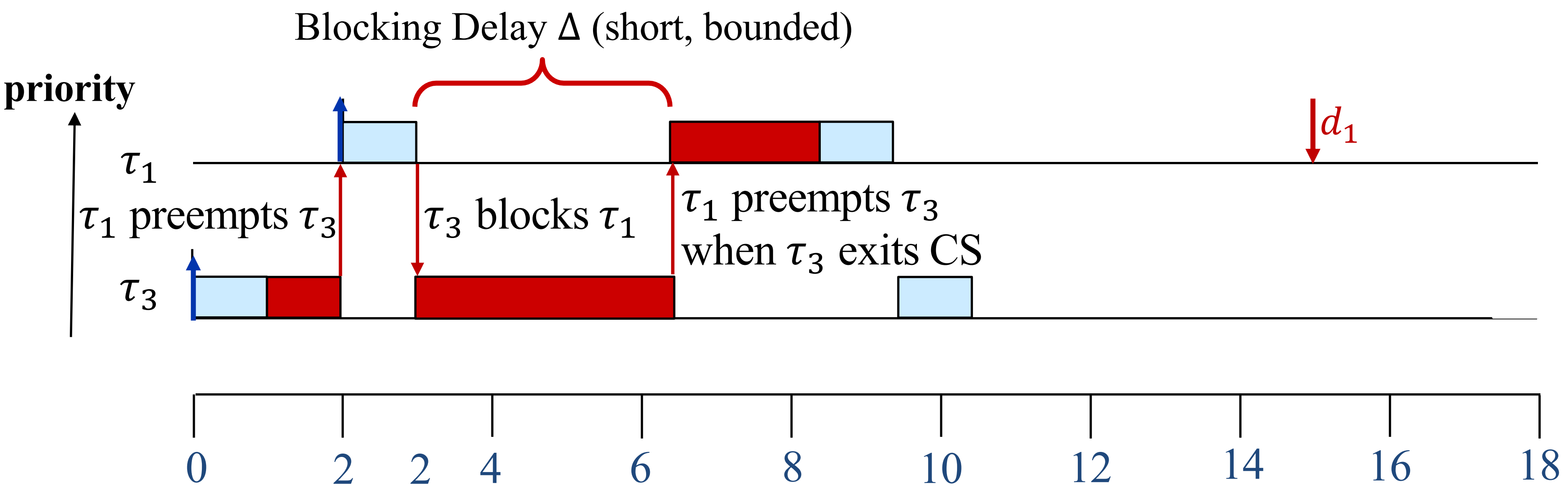
```
int x;
int y;
```

```
lock(s)
a = x+1;
b = y+2;
c = x+y;
unlock(s)
```



Blocking Delay

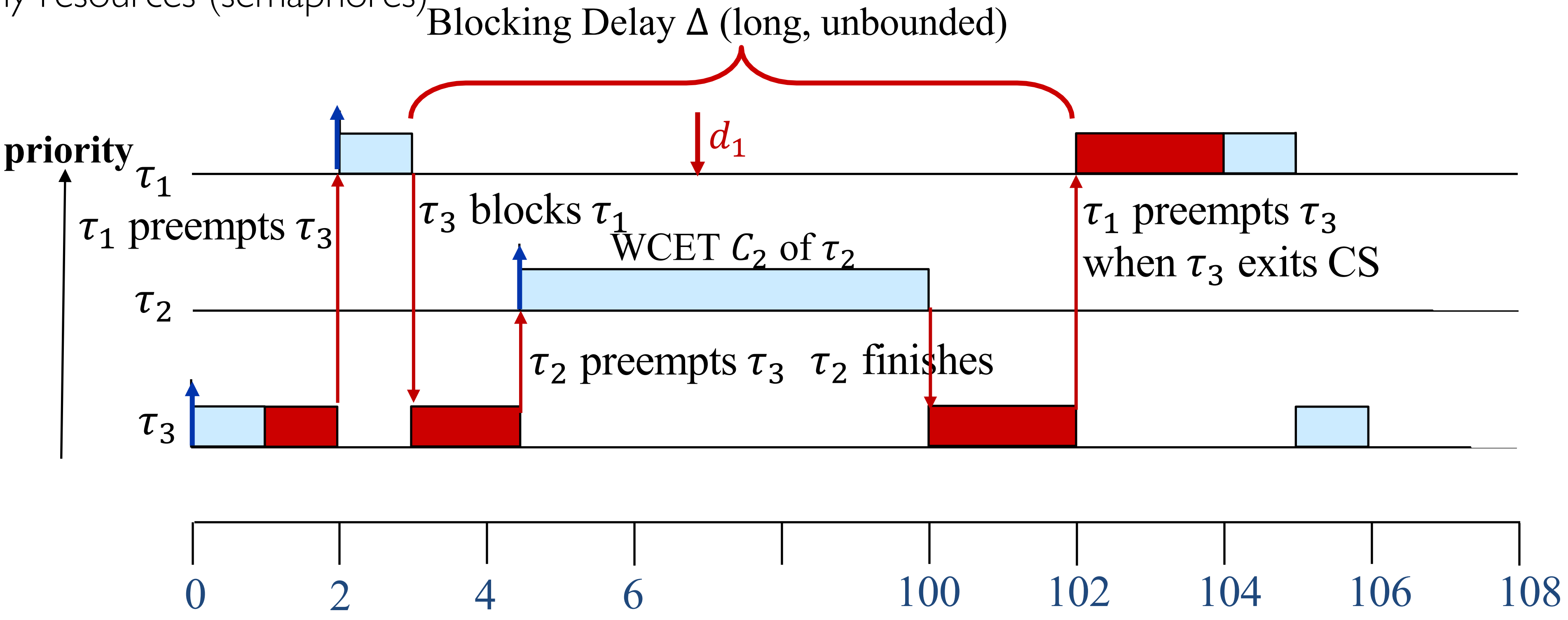
- Lower Priority (LP) tasks can cause blocking delay to Higher Priority (HP) tasks due to resource sharing
  - HP tasks may cause preemption delay to LP tasks, but not blocking delay
- Example: Two tasks τ_1, τ_3 with priority ordering $P_1 > P_3$. They both require semaphore s (which protects the red CS)
- If HP task τ_1 tries to lock s that is held by LP task τ_3, τ_1 is blocked until τ_3 unlocks s, so τ_1 experiences a blocking delay Δ.
  - Since CS is typically very short, it seems this blocking time delay Δ is bounded by the longest critical section in lower-priority tasks?
- No, blocking delay may be unbounded!



Blocking Delay Δ (short, bounded)
priority
τ_1
τ_3
τ_1 preempts τ_3
τ_3 blocks τ_1
τ_1 preempts τ_3 when τ_3 exits CS
d_1
0
2
2
4
6
8
10
12
14
16
18

Priority Inversion I

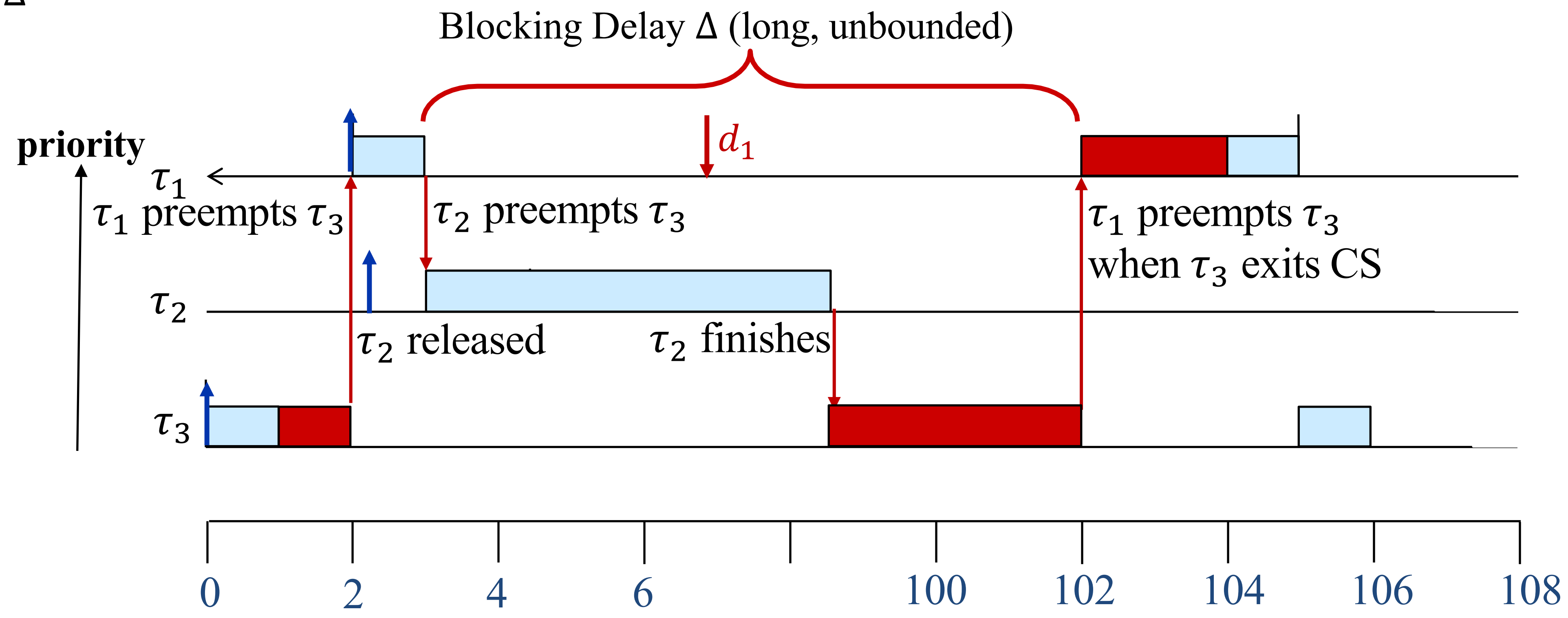
- Three tasks τ_1, τ_2, τ_3 with priority ordering $P_1 > P_2 > P_3$. τ_1, τ_3 both require semaphore s, and τ_2 does not require any semaphore
- t=1: LP task τ_3 locks s and enters CS
- t=2: HP task τ_1 is released and preempts τ_3
- t=3: HP task τ_1 tries to lock s, but gets blocked by τ_3 holding s
- t=4.2: Medium Priority (MP) task τ_2 is released and preempts τ_3
- t=100: MP task τ_2 finishes execution after running for its WCET C_2; τ_3 resumes execution in CS
- t=102: LP task τ_3 unlocks s; HP task τ_1 preempts τ_3 and finally locks s, after experiencing a long, unbounded blocking delay Δ, and misses its deadline d_1
- This is priority inversion, since MP task τ_2 causes a long blocking delay to HP task τ_1, even though they do not share any resources (semaphores)



Blocking Delay Δ (long, unbounded)
d_1
priority
τ_1
τ_2
τ_3
τ_1 preempts τ_3
τ_3 blocks τ_1
WCET C_2 of τ_2
τ_1 preempts τ_3 when τ_3 exits CS
τ_2 preempts τ_3
τ_2 finishes
0
2
4
6
100
102
104
106
108

Priority Inversion II

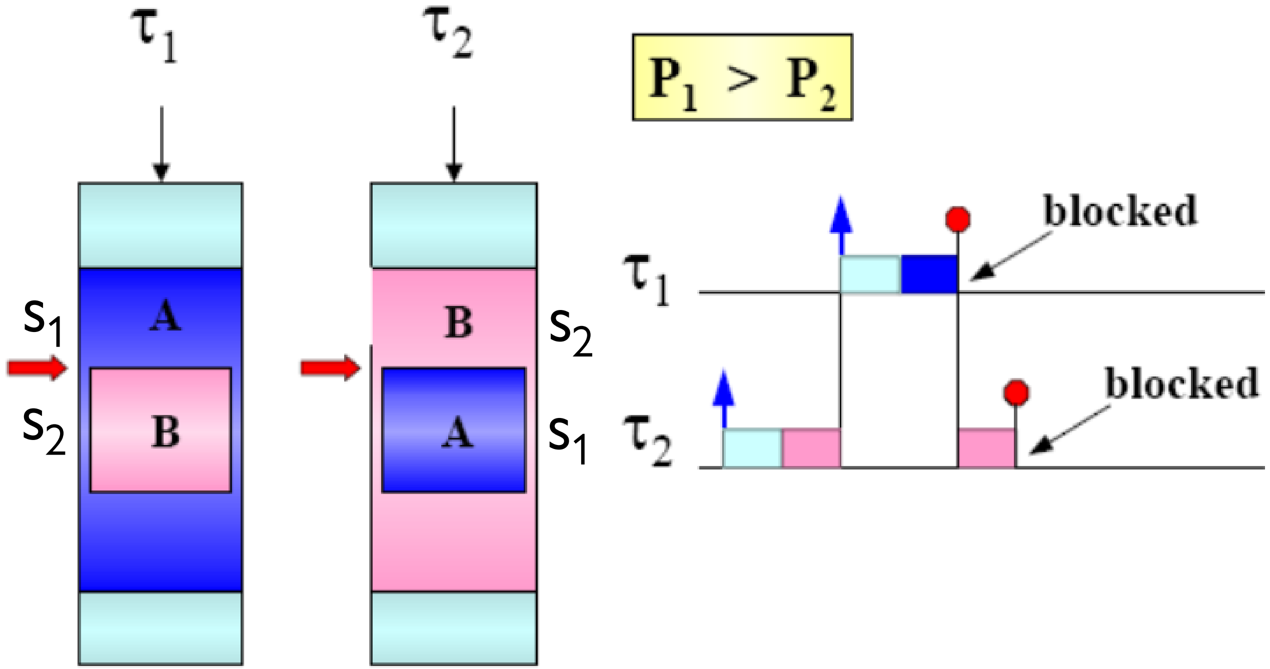
- (This scenario is more realistic and likely than previous one, as MP task τ_2 may be released anytime during τ_1's execution after it preempts τ_3. In the previous example, τ_2 is released during τ_3's critical section, which is very short.)
- t=1: LP task τ_3 locks s and enters CS
- t=2: HP task τ_1 is released and preempts τ_3
- t ∈ [2, 3]: MP task τ_2 is released, but cannot run since HP task τ_1 is running
- t=3: HP task τ_1 tries to lock s, but gets blocked by τ_3 holding s; MP task τ_2 starts running
- t=98.5: MP task τ_2 finishes execution after running for its WCET C_2; τ_3 resumes execution in the CS
- t=102: LP task τ_3 unlocks s; HP task τ_1 preempts τ_3 and finally locks s, after experiencing a long, unbounded blocking Delay Δ



Blocking Delay Δ (long, unbounded)
priority
d_1
τ_1
τ_1 preempts τ_3
τ_2 preempts τ_3
τ_1 preempts τ_3 when τ_3 exits CS
τ_2
τ_2 released
τ_2 finishes
τ_3
0
2
4
6
100
102
104
106
108

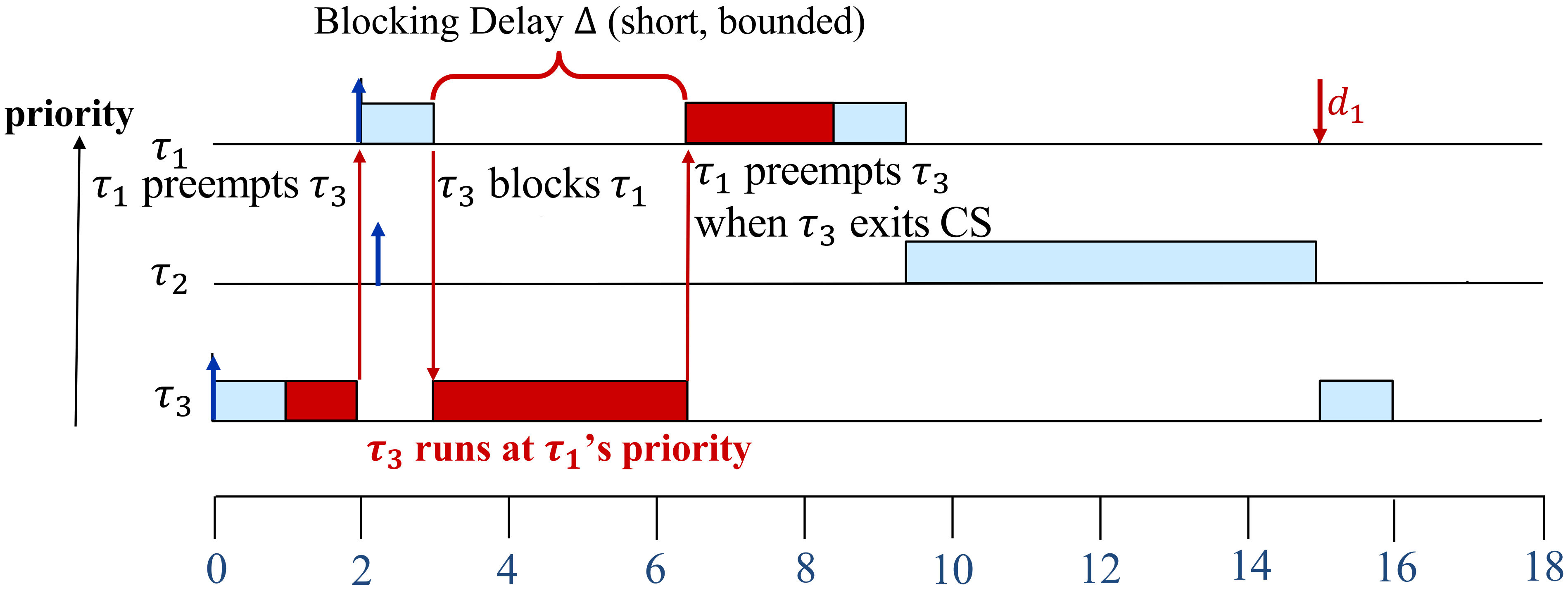
Deadlocks

- Classic deadlock scenario: Two tasks $\boldsymbol{\tau_1}$ and $\boldsymbol{\tau_2}$ lock two semaphores s_1, s_2 in opposite order (s_1 protects blue CS A and s_2 protects pink CS B)
  - HP task $\boldsymbol{\tau_1}$ enters blue CS A before pink CS B: …lock(s_1)…lock(s_2)… unlock(s_2)…unlock(s_1)…
  - LP task $\boldsymbol{\tau_2}$ enters pink CS B before blue CS A: …lock(s_2)…lock(s_1)… unlock(s_1)…unlock(s_2)…
  - LP task $\boldsymbol{\tau_2}$ runs first and locks s_2
  - HP task $\boldsymbol{\tau_1}$ starts running and locks s_1, then tries to lock s_2, gets blocked by $\boldsymbol{\tau_2}$
  - $\boldsymbol{\tau_2}$ starts running and tries to lock $s_{1,}$ but $\boldsymbol{\tau_1}$ holds s_1. Circular waiting → deadlock



Priority Inheritance Protocol (PIP)

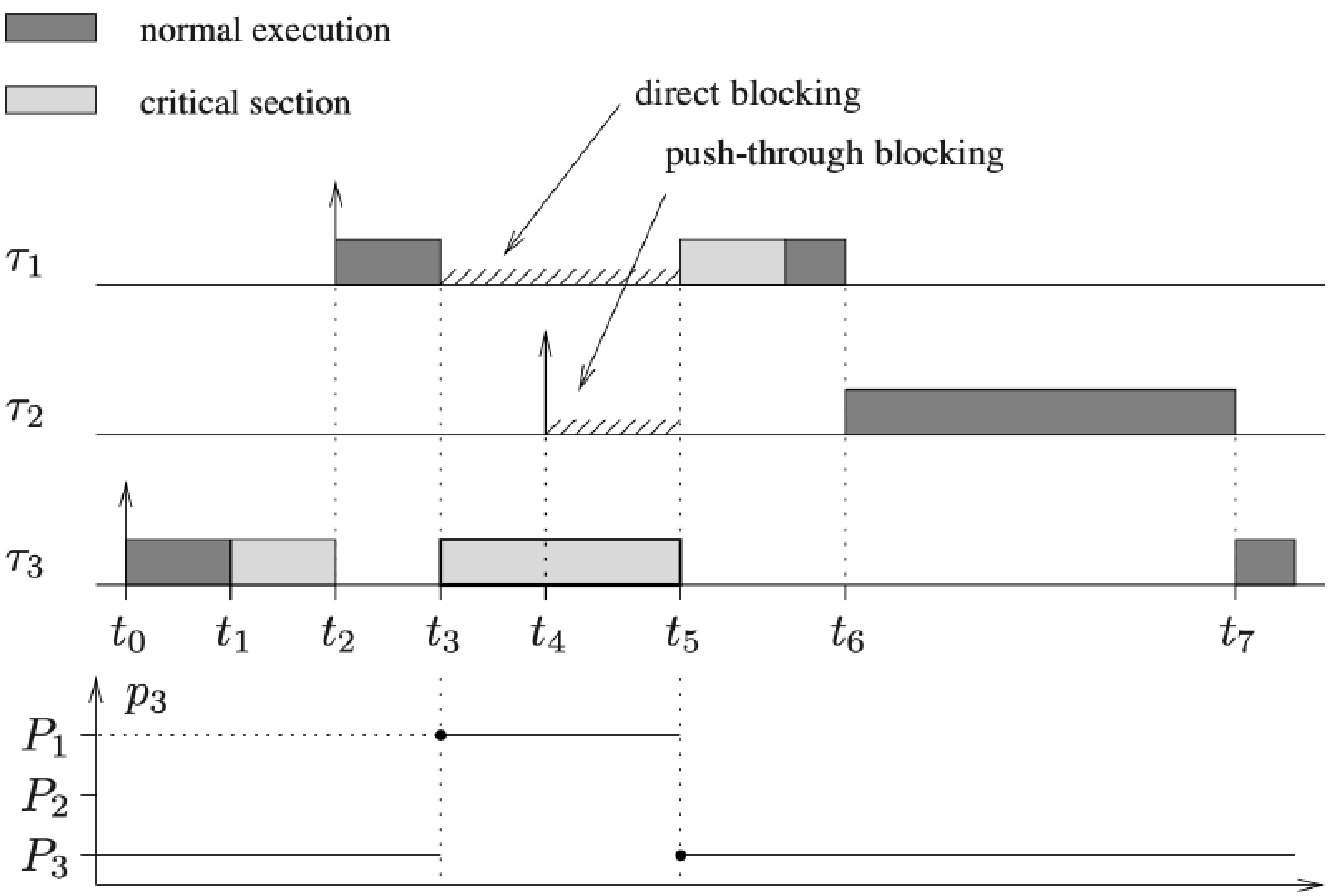
- In 1997, this bug caused the Mars pathfinder to freeze up occasionally and then starts working again. Fixed by uploading a software patch enabling Priority-Inheritance Protocol (PIP)
- A task $\boldsymbol{\tau_i}$ in a CS increases its priority, if it is holding a lock s and blocks other higher priority tasks, by inheriting the highest priority of all higher-priority tasks $\boldsymbol{\tau_k}$ blocked waiting for lock s
  - $P_{\tau_i \text{ holding } s} = \max\{P_k | \tau_k \text{ blocked on } s\}$
- t=3: HP task $\boldsymbol{\tau_1}$ tries to enter CS, gets blocked since LP task $\boldsymbol{\tau_3}$ is in CS; $\boldsymbol{\tau_3}$ inherits $\boldsymbol{\tau_1}$'s high priority, so MP task $\boldsymbol{\tau_2}$ cannot pre-empt $\boldsymbol{\tau_3}$, which finishes its CS, and then $\boldsymbol{\tau_1}$ can run after a short, bounded Blocking Delay Δ (regardless of when $\boldsymbol{\tau_2}$ is released at t ∈ [2, 3] or t>3)



Blocking Delay Δ (short, bounded)
priority
τ_1
τ_2
τ_3
d_1
τ_1 preempts τ_3
τ_3 blocks τ_1
τ_1 preempts τ_3 when τ_3 exits CS
τ_3 runs at τ_1's priority
0
2
4
6
8
10
12
14
16
18

Blocking Time under PIP

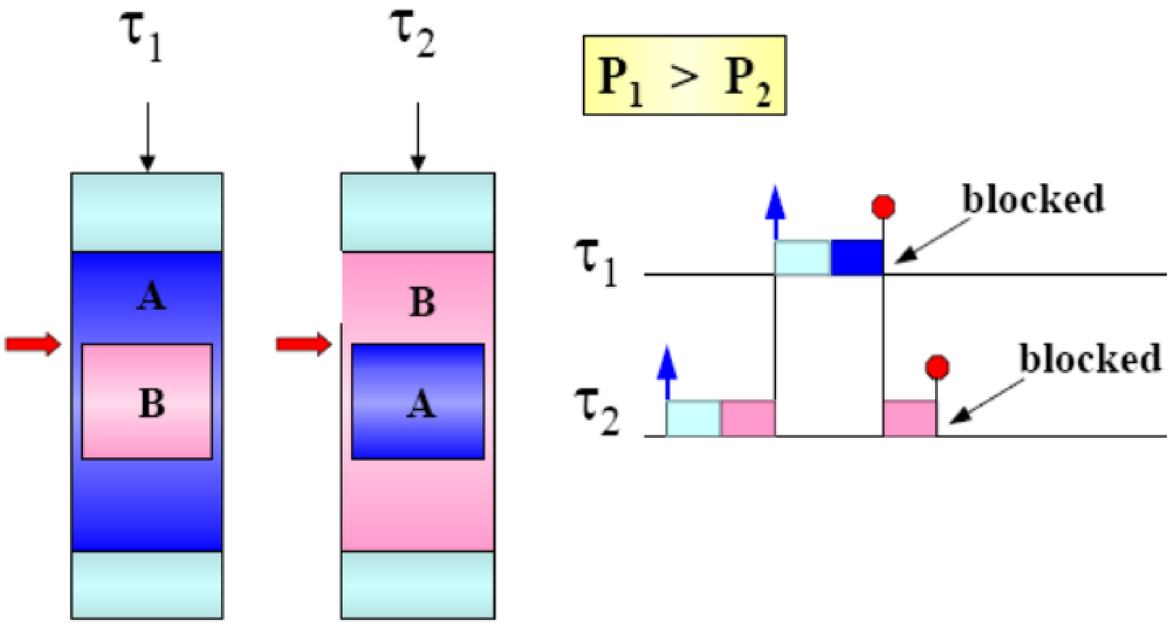
- Under PIP, task τ_i may experience two types of blocking delays:
  - Direct blocking: τ_i tries to lock semaphore s that is already locked
  - Push-through blocking: τ_i blocked by lower-priority task that has inherited a higher-priority (τ_i itself may not need any semaphores)
- Example:
  - HP task τ_1 experiences direct blocking by LP task τ_3 in time interval $[t_3, t_5]$
  - MP task τ_2 experiences push-through blocking by LP task τ_3 in time interval $[t_4, t_5]$
- PIP analogy: suppose you have checked out a book from the library and planned to read it in your spare time. But you got a message from the library that some VIP, say the university president, just got in the waiting queue for the book. You should then hurry up, give the book-reading task a high priority so it is not preempted by other daily chores, finish reading it, and return it to the library quickly, so the VIP is not delayed for a long time.
- Your book-reading task (critical section) initially had a low priority, but it inherits higher priority of the VIP as soon as the VIP gets blocked waiting for the book (shared resource)



LP task τ_3's priority is increased at time t_3 when HP task τ_1 tries to lock semaphore s but is blocked by τ_3; NOT when τ_3 lock s at time t_1

PIP Pros and Cons

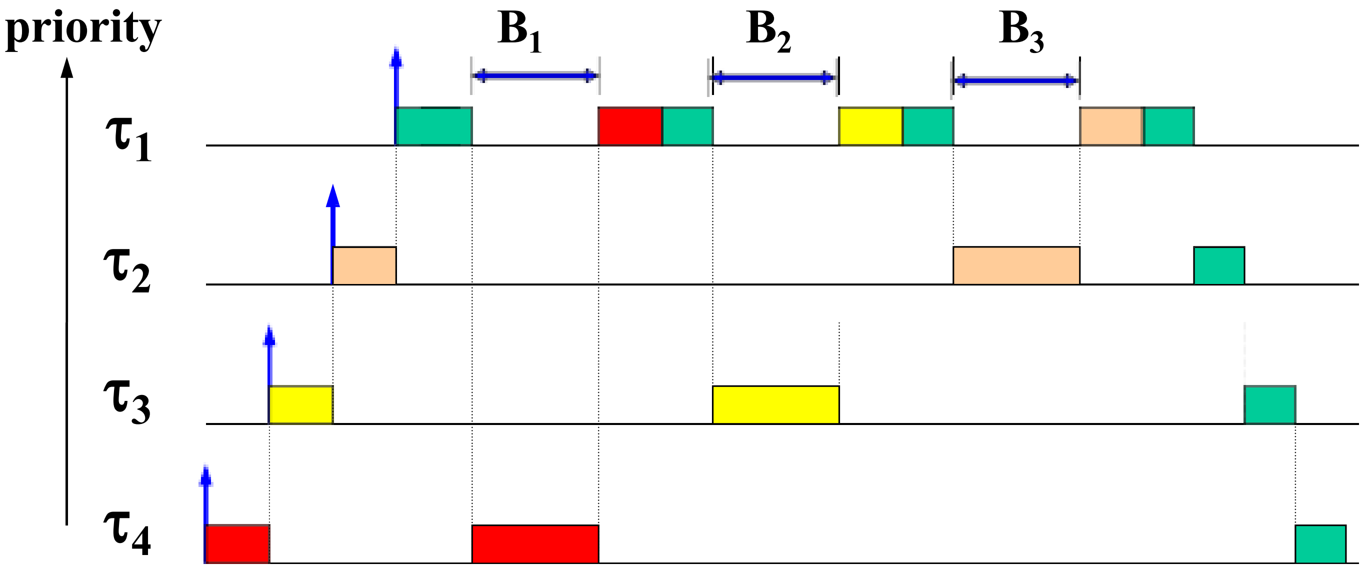
- Pros:
  - It prevents priority inversion
  - It is transparent to the programmer
- Cons:
  - It does not prevent deadlocks and chained blocking



Deadlock still occurs under PIP

PIP Causes Chained Blocking

- Chained blocking: task τ_i can be blocked at most once by each lower priority task
- Theorem: Task τ_i can be blocked at most for the duration of $\min(n, m)$ critical sections
  - n is the number of tasks with priority lower than τ_i
  - m is the number of locks/semaphores on which τ_i can be blocked
- In this example, Four tasks and three semaphores (s_1 protects red CS, s_2 protects yellow CS, s_3 protects beige CS). Task τ_i is blocked for the duration of $\min(3,3) = 3$ critical sections

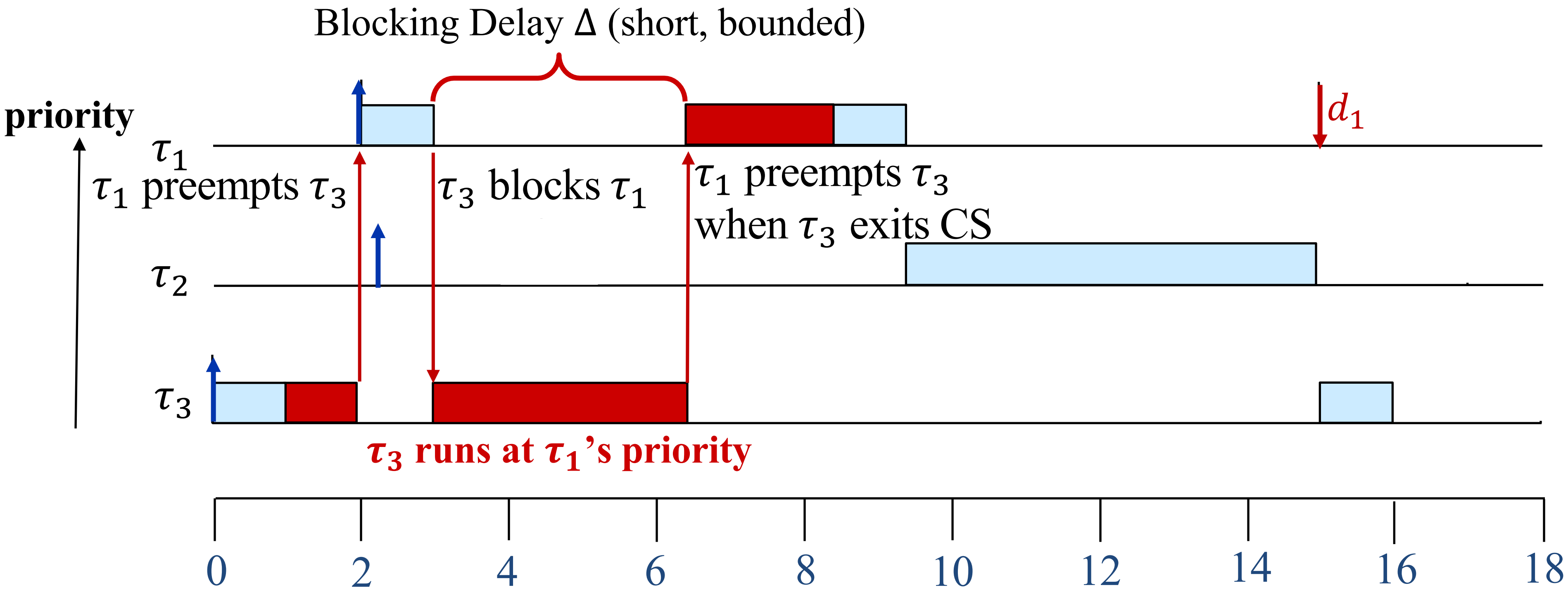


Priority Ceiling Protocol (PCP)

- Assumptions: fixed-priority scheduling; resources required by all tasks are known a priori at design time (not required by PIP)
- Priority Ceiling Protocol PCP = PIP + ceiling blocking
- PIP still holds: When τ_i is blocked on s_k, the lower-priority task currently holding s_k inherits τ_i's priority
- Each semaphore is assigned a ceiling, equal to maximum priority of all tasks that require it: $C(s_k) = \max\{P_j : \tau_j \text{ uses } s_k\}$
- Task τ_i can acquire semaphore s_j and enter CS only if
  - $P_i > \max\{C(s_k) : s_k \text{ locked by other tasks} \neq \tau_i\}$, that is, its priority P_i is strictly higher than the maximum ceiling of all semaphores (s_k) currently held by other tasks; otherwise it is blocked due to ceiling blocking. (s_k may or may not be the same as s_j)
  - Corollary: If s_j itself is currently held by some task, then τ_i cannot lock s_j, since $P_i \leq C(s_j)$, as ceiling of s_j is at least the priority of τ_i by definition
- Under PCP, a task τ_i may experience ceiling blocking, in addition to direct blocking and push-through blocking under PIP:
  - τ_i tries to lock s_j, but its priority P_i is not strictly higher than the maximum ceiling of all semaphores (s_k) currently held by other tasks (s_j itself may be free)
  - Ceiling blocking is "preventive blocking", since a task may be blocked even though the semaphore it tries to lock is free. This helps to prevent potential deadlocks and chained blocking

PCP Example I

- Three tasks τ_1, τ_2, τ_3 with priority ordering $P_1 > P_2 > P_3$. τ_1, τ_3 both require semaphore s, and τ_2 does not require any semaphore
  - $C(s) = \max\{P_j : \tau_j \text{ uses } s\} = \max\{P_1, P_2\} = P_1$
- The execution trace is the same as PIP, since PCP includes PIP as part of the protocol



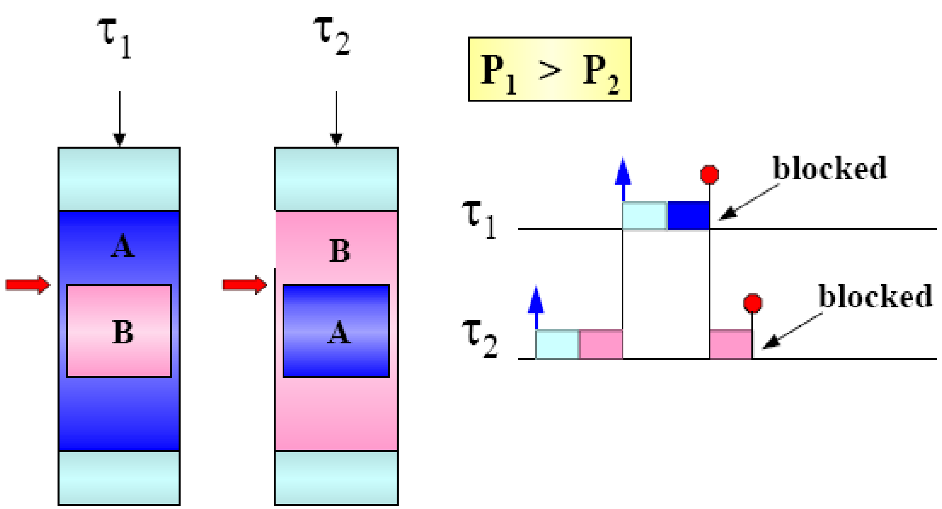
Blocking Delay Δ (short, bounded)
priority
τ_1
τ_2
τ_3
d_1
τ_1 preempts τ_3
τ_3 blocks τ_1
τ_1 preempts τ_3 when τ_3 exits CS
τ_3 runs at τ_1's priority
0
2
4
6
8
10
12
14
16
18

PCP Prevents Deadlocks

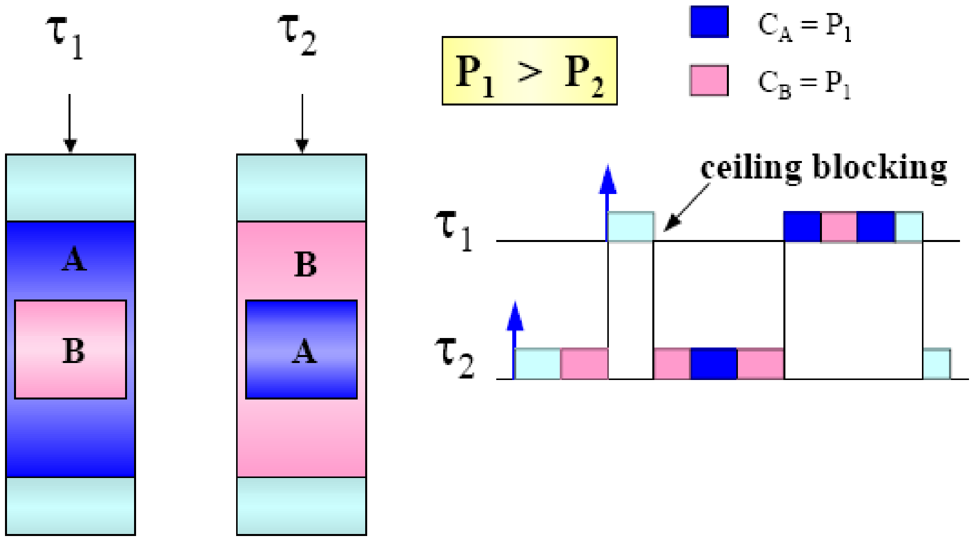
- Semaphore s_1 protects blue CS A and s_2 protects pink CS B
- Classic deadlock scenario (with or without PIP): Two tasks τ_1 and τ_2 lock two semaphores in opposite order:
  - LP task τ_2 runs first and locks s_2
  - HP task τ_1 starts running and locks s_1, then tries to lock s_2, gets blocked by τ_2
  - τ_2 starts running and tries to lock s_1 but τ_1 holds s_1. Circular waiting → deadlock

- Under PCP, $C(s_1) = C(s_2) = \max\{P_1, P_2\} = P_1$. Both semaphores s_1 and s_2 have ceiling equal to P_1, since they are all required by the higher priority task τ_1.
  - LP task τ_2 runs first and locks s_2
  - HP task τ_1 runs and preempts τ_2. When τ_1 tries to lock s_1, it is blocked since its priority does not exceed ceiling of s_2, i.e., $P_1 \leq ceil(s_2) = P_1$
  - τ_2 will lock both s_2 and s_1, and exit both CSes before τ_1 can lock s_1 and s_2. This prevents circular waiting and deadlock
- Analogous to requiring a philosopher to pick up both forks in one atomic operation to prevent deadlocks

Typical Deadlock

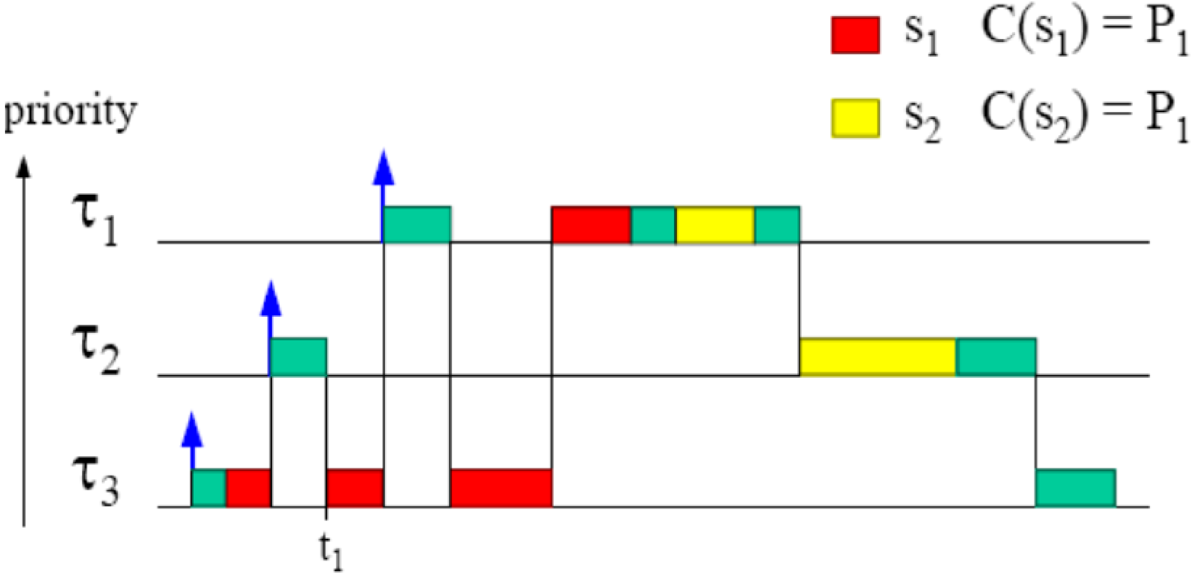


Deadlock avoidance with PCP



PCP Prevents Chained Blocking

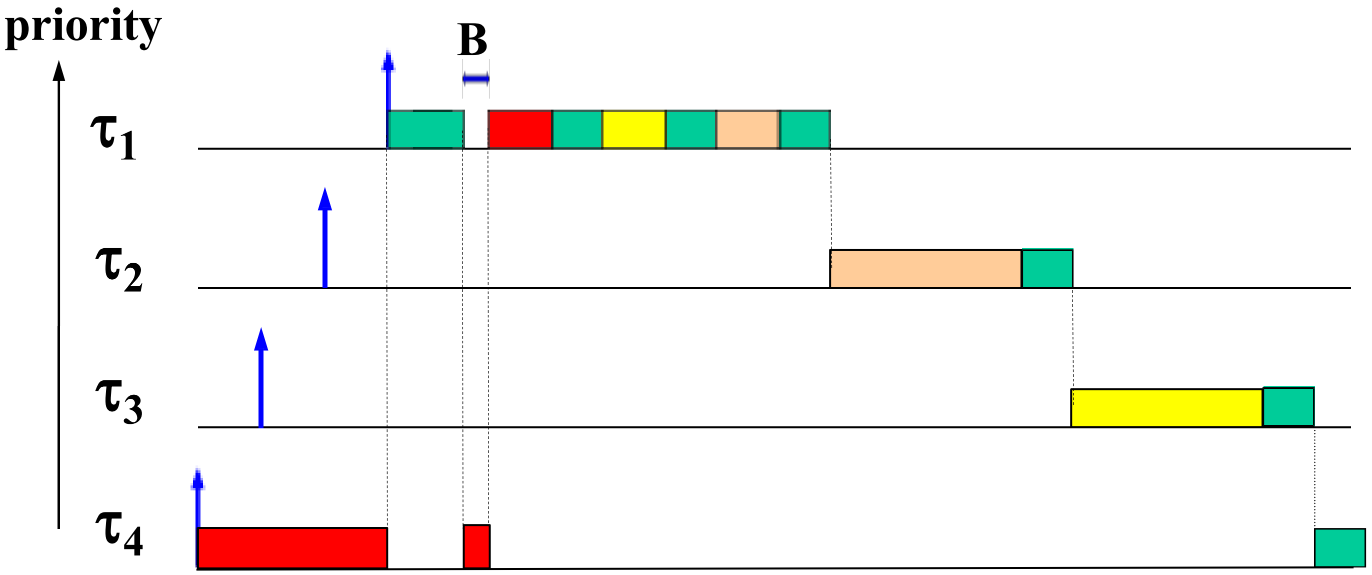
- Three tasks and two semaphores (s_1 protects red CS and s_2 protects yellow CS)
- $C(s_1) = \max\{P_1, P_3\} = P_1,\ C(s_2) = \max\{P_1, P_2\} = P_1$. Both semaphores s_1 and s_2 have ceiling equal to P_1, since they are all required by the highest priority task τ_1. At time t_1, LP task τ_3 is holding s_1 (in red CS). When MP task τ_2 tries to lock s_2 and enter yellow CS, it is blocked since its priority does not exceed ceiling of s_1, $P_2 \leq C(s_1) = P_1$ (ceiling blocking)
- Hence τ_3 must unlock s_1 before τ_2 can lock s_2. This prevents possible chained blocking, so τ_1 is blocked only once by the red CS. (Under PIP, τ_2 will enter the yellow CS, at time t_1, so τ_1 may be blocked twice by both red CS and yellow CS.)

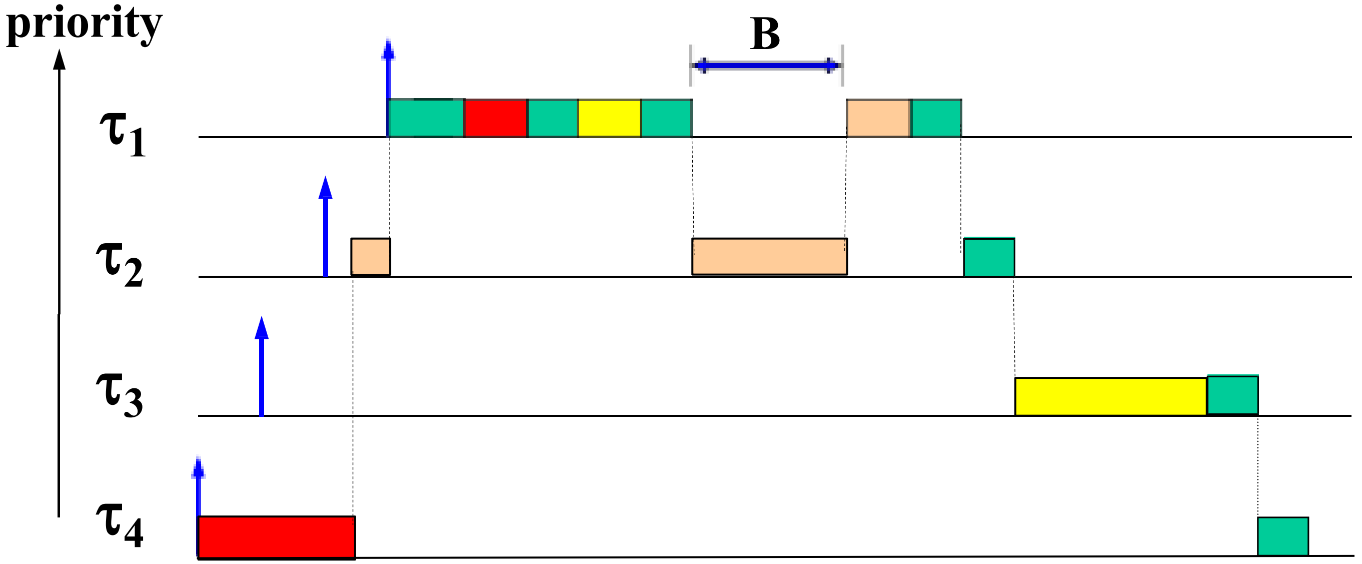


t_1: τ_2 is blocked by the PCP, since $P_2 < C(s_1)$

PCP Prevents Chained Blocking

- Recall the example with chained blocking under PIP
- Four tasks and three semaphores (s_1 protects red CS, s_2 protects yellow CS, s_3 protects beige CS)
- Under PCP: $C(s_1) = \max\{P_1, P_4\} = P_1$, $C(s_2) = \max\{P_1, P_3\} = P_1$, $C(s_3) = \max\{P_1, P_2\} = P_1$. All semaphores s_1, s_2, s_3 have ceiling equal to P_1, since they are all required by the highest priority task τ_1.
- Upper fig: While τ_4 is holding s_1 (in the red CS), τ_3 cannot lock s_2, since $P_3 \leq C(s_1) = P_1$; and τ_2 cannot lock s_3, since $P_2 \leq C(s_1) = P_1$ (ceiling blocking)
- Lower fig: While τ_4 is holding s_1 (in the red CS), τ_3 cannot lock s_2. Later when τ_4 releases s_1, both τ_2 and τ_3 are ready, and τ_2 locks s_3 since it has higher priority than τ_3
- Hence PCP prevents chained blocking, since task τ_1 is blocked at most once by a lower-priority task (either τ_4, or τ_3, or τ_2)



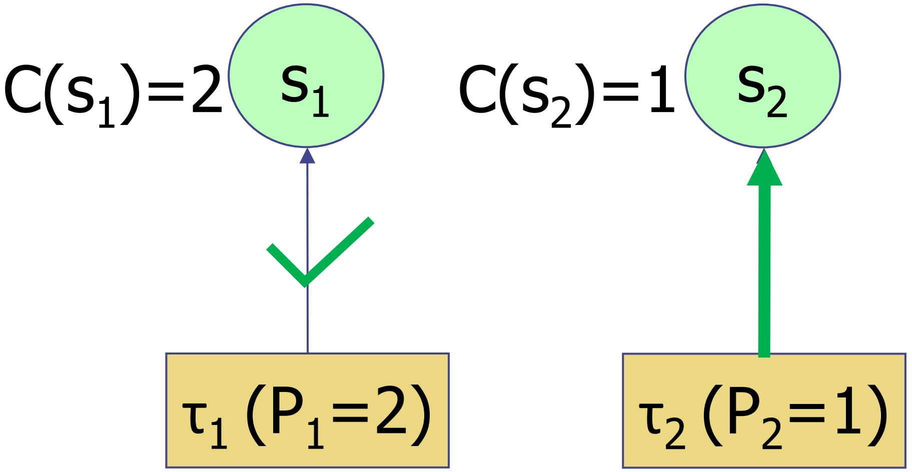


PCP Example II

- Two tasks τ_1, τ_2 with priority ordering $P_1 = 2$ (higher) and $P_2 = 1$ (lower) and two semaphores s_1, s_2. (In the figure below, a thin blue arrow indicates that a task requires a semaphore during its execution; a solid green arrow indicates that a task is currently holding the required semaphore.)
  - $C(s_1) = \max\{P_j: \tau_j \text{ uses } s_1\} = \max\{P_1\} = 2$
  - $C(s_2) = \max\{P_j: \tau_j \text{ uses } s_2\} = \max\{P_2\} = 1$
- While τ_1 is holding s_1, τ_2 cannot lock s_2, since $P_2 = 1 \leq C(s_1) = 2$ (ceiling blocking)
  - In this case PCP is over-conservative, and there are no bad consequences even if we allow τ_1 to hold s_1 and τ_2 to hold s_2 simultaneously
- While τ_2 is holding s_2 (indicated by the thick green arrow), τ_1 can lock s_1, since $P_1 = 2 > C(s_2) = 1$

Task	Prio	sems
τ_1	2	s_1
τ_2	1	s_2

sem	Ceil
s_1	2
s_2	1

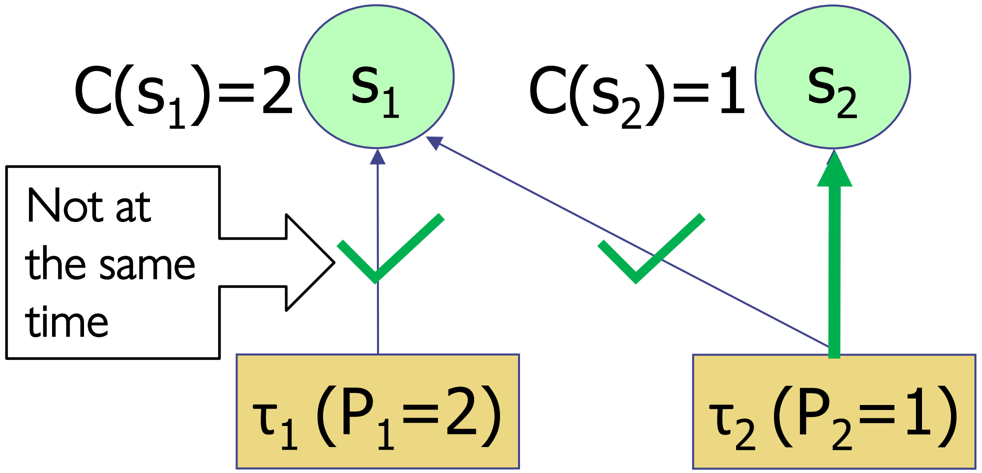


PCP Example II

- Two tasks **τ_1, τ_2** and two semaphores s_1, s_2
  - $C(s_1) = \max\{P_j: \tau_j \text{ uses } s_1\} = \max\{P_1, P_2\} = 2$
  - $C(s_2) = \max\{P_j: \tau_j \text{ uses } s_2\} = \max\{P_2\} = 1$
- While **τ_1** is holding s_1, **τ_2** cannot lock s_2, since $P_2 = 1 \le C(s_1) = 2$ (ceiling blocking)
  - In this case PCP is over-conservative, and there are no bad consequences even if we allow **τ_1** to hold s_1 and **τ_2** to hold s_2 simultaneously
- While **τ_2** is holding s_2, **τ_1** can lock s_1, since $P_1 = 2 > C(s_2) = 1$

Task	Prio	sems
τ_1	2	s_1
τ_2	1	s_1, s_2

sem	Ceil
s_1	2
s_2	1

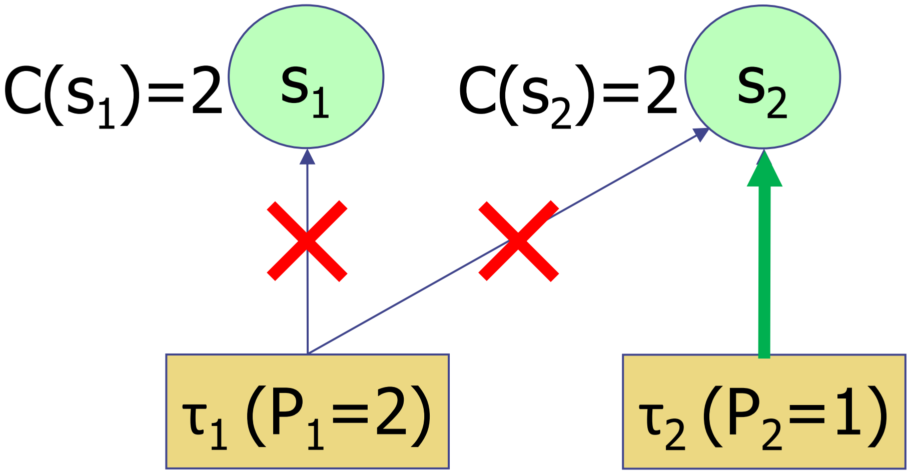


PCP Example II

- Two tasks $\boldsymbol{\tau_1, \tau_2}$ and two semaphores s_1, s_2
  - $C(s_1) = \max\{P_j: \tau_j \text{ uses } s_1\} = \max\{P_1\} = 2$
  - $C(s_2) = \max\{P_j: \tau_j \text{ uses } s_2\} = \max\{P_1, P_2\} = 2$
- While $\boldsymbol{\tau_1}$ is holding s_1, $\boldsymbol{\tau_2}$ cannot lock s_2, since $P_2 = 1 \le C(s_1) = 2$ (ceiling blocking)
- While $\boldsymbol{\tau_2}$ is holding s_2, $\boldsymbol{\tau_1}$ cannot lock s_1, since $P_1 = 2 \le C(s_2) = 2$ (ceiling blocking)
  - In this case PCP is over-conservative, and there are no bad consequences even if we allow $\boldsymbol{\tau_1}$ to hold s_1 and $\boldsymbol{\tau_2}$ to hold s_2 simultaneously

Task	Prio	sems
τ_1	2	s_1, s_2
τ_2	1	s_2

sem	Ceil
s_1	2
s_2	2

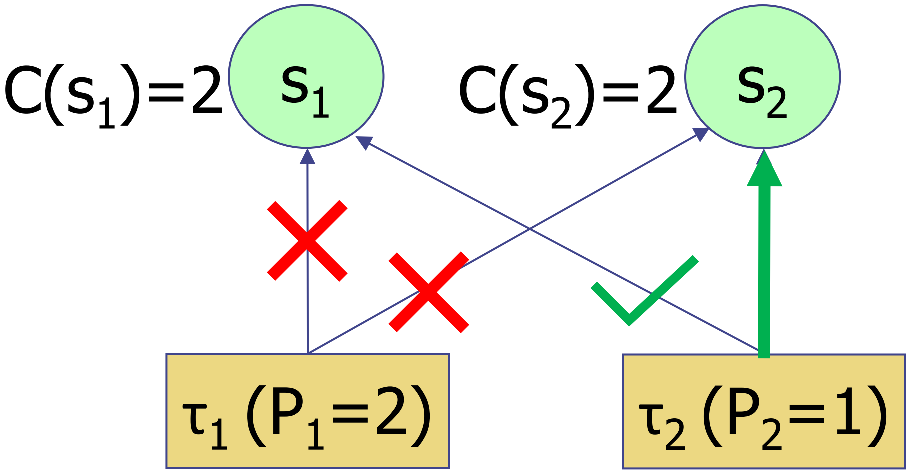


PCP Example II

- Two tasks **τ_1, τ_2** and two semaphores s_1, s_2
  - $C(s_1) = \max\{P_j: \tau_j \text{ uses } s_1\} = \max\{P_1, P_2\} = 2$
  - $C(s_2) = \max\{P_j: \tau_j \text{ uses } s_2\} = \max\{P_1, P_2\} = 2$
- While **τ_1** is holding s_1, **τ_2** cannot lock s_2, since $P_2 = 1 \le C(s_1) = 2$ (ceiling blocking)
- While **τ_2** is holding s_2, **τ_1** cannot lock s_1, since $P_1 = 2 \le C(s_2) = 2$ (ceiling blocking)
  - This prevents any potential deadlocks in the future, when **τ_1**, **τ_2** each holds one of s_1, s_2 and tries to lock the other (circular waiting)

Task	Prio	sems
τ_1	2	s_1, s_2
τ_2	1	s_1, s_2

sem	Ceil
s_1	2
s_2	2

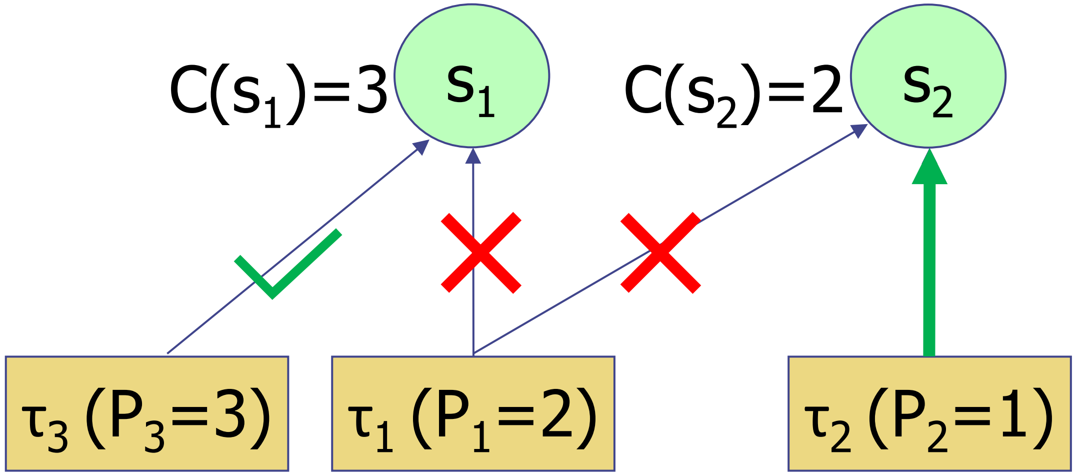


PCP Example II

- Three tasks τ_1, τ_2, τ_3 and two semaphores s_1, s_2
  - $C(s_1) = \max\{P_j\!: \tau_j \text{ uses } s_1\} = \max\{P_1, P_3\} = 3$
  - $C(s_2) = \max\{P_j\!: \tau_j \text{ uses } s_2\} = \max\{P_2, P_3\} = 2$
- While τ_2 is holding s_2, τ_1 cannot lock s_1, since $P_1 = 2 \le C(s_2) = 2$ (ceiling blocking)
  - In this case PCP is over-conservative, and there are no bad consequences even if we allow τ_1 to lock s_1
- While τ_2 is holding s_2, τ_3 can lock s_1, since $P_3 = 3 > C(s_2) = 2$

Task	Prio	sems
τ_1	2	S_1
τ_2	1	s_1, s_2
τ_3	3	s_2

sem	Ceil
s_1	3
s_2	3

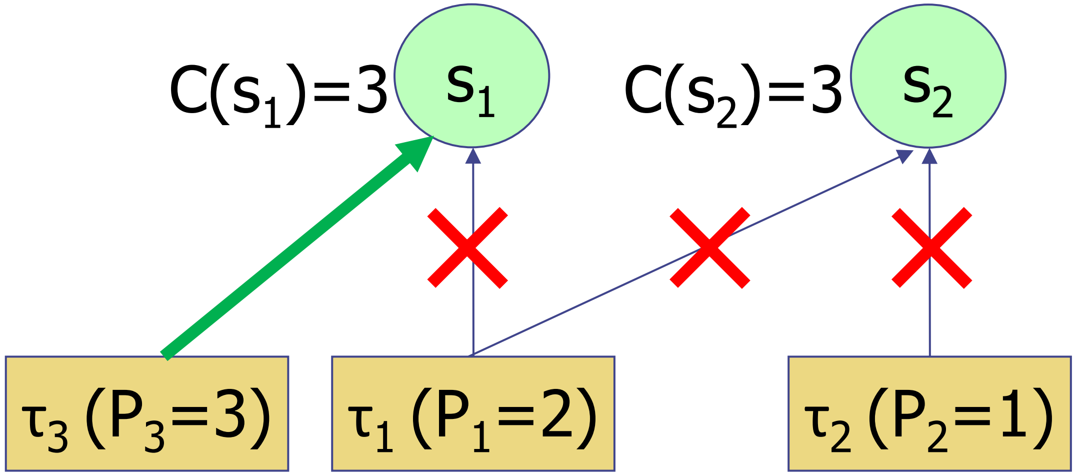


PCP Example II

- Three tasks **τ_1, τ_2, τ_3** and two semaphores s_1, s_2
  - $C(s_1) = \max\{P_j : \tau_j \text{ uses } s_1\} = \max\{P_1, P_3\} = 3$
  - $C(s_2) = \max\{P_j : \tau_j \text{ uses } s_2\} = \max\{P_2, P_3\} = 3$
- While **τ_3** is holding s_1, **τ_1** cannot lock s_2, since $P_1 = 2 \leq C(s_1) = 3$ (ceiling blocking)
- While **τ_3** is holding s_1, **τ_2** cannot lock s_2, since $P_2 = 1 \leq C(s_1) = 3$ (ceiling blocking)
  - In this case PCP is over-conservative, and there are no bad consequences even if we allow **τ_1** to lock s_2 and **τ_2** to lock s_2

Task	Prio	sems
τ_1	2	S_1
τ_2	1	s_2
τ_3	3	s_1, s_2

sem	Ceil
s_1	3
s_2	3

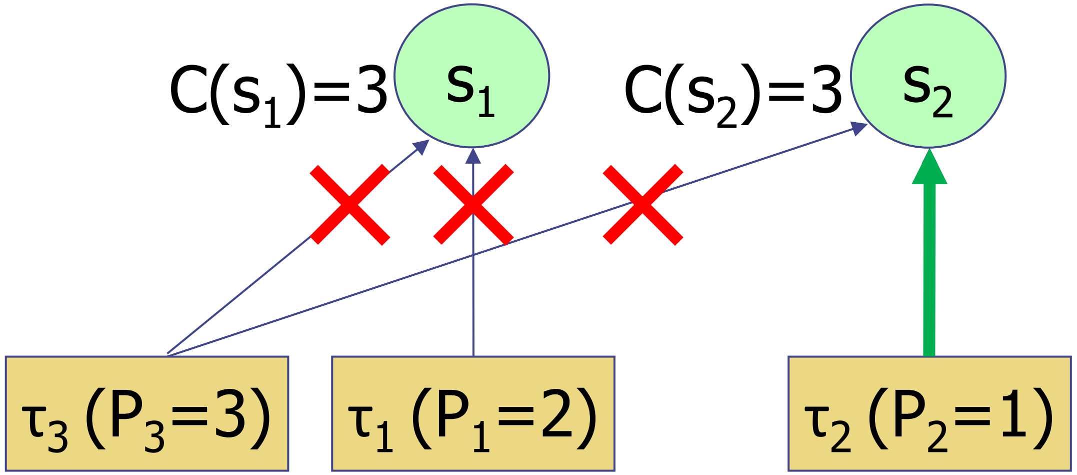


PCP Example II

- Three tasks τ_1, τ_2, τ_3 and two semaphores s_1, s_2
  - $C(s_1) = \max\{P_j : \tau_j \text{ uses } s_1\} = \max\{P_1, P_3\} = 3$
  - $C(s_2) = \max\{P_j : \tau_j \text{ uses } s_2\} = \max\{P_2, P_3\} = 3$
- While τ_2 is holding s_2, τ_1 cannot lock s_1, since $P_1 = 2 \le C(s_2) = 3$ (ceiling blocking)
  - This prevents potential chained blocking in the future, when τ_1, τ_2 each holds one of s_1, s_2 , and τ_3 tries to lock both s_1 and s_2, and get blocked twice
- While τ_2 is holding s_2, τ_3 cannot lock s_1, since $P_3 = 3 \le C(s_1) = 3$ (ceiling blocking)
  - In this case PCP is over-conservative, and there are no bad consequences even if we allow τ_3 to lock s_1

Task	Prio	sems
τ_1	2	S_1
τ_2	1	s_2
τ_3	3	s_1, s_2

sem	Ceil
s_1	3
s_2	3



IMPORTANT

PCP Blocking Time

A given task i is blocked (or delayed) by at most one critical section of any lower priority task locking a semaphore with priority ceiling greater than or equal to the priority of task i. We can explain that mathematically using the notation:

$$B_i = \max_{\{k,s \,\mid\, k\in lp(i) \,\wedge\, s\in used\_by(k) \,\wedge\, ceil(s)\geq pri(i)\}} cs_{k,s}$$

- Consider all lower-priority tasks ($k\in lp(i)$), and the semaphores they can lock (s)
- Select from those semaphores (s) with ceiling higher than or equal to $pri(i) = P_i$
- Take max length of all tasks (k)'s critical sections that lock semaphores (s)
- (The blocking time is valid even for a task that does not require any semaphores/critical sections, as it may experience push-through blocking.)

PCP Pros and Cons

- Pros:
  - It prevents priority inversion, deadlocks, and chained blocking
  - Any given task is blocked at most once by a lower-priority task
- Cons:
  - It is not transparent to the programmer, as shared resources required by all tasks must be known a priori at design time, and programmer needs to calculate priority ceilings of all semaphores and pass them to the OS (PIP does not need this step)

	Deadlock Prevention	Number of blockings	Programmer Transparency
PIP	No	$\min(n, m)$	Yes
PCP	Yes	1	No

# blockings under PIP: n is the number of tasks with priority lower than τ_i; m is the number of locks/semaphores on which τ_i can be blocked

Schedulability Analysis under PIP and PCP

- Let B_i denote the maximum blocking time experienced by task τ_i by lower-priority tasks due to shared resources
- Schedulable utilization bound for RM scheduling with blocking time (sufficient condition):
  - A taskset is schedulable under RM scheduling with blocking time if
  - $\forall i,$ priority level i utilization $U_i = \sum_{\forall j \in hp(i)} \frac{C_j}{T_j} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$
  - Assumptions: task period equal to deadline ($P_i = D_i$); task with smaller period P_i is assigned higher priority (RM priority assignment)
- Response Time Analysis (RTA) for RM scheduling with resource sharing (necessary and sufficient condition):
  - Task τ_i's WCRT R_i is computed by solving the following recursive equation to find the minimum fixed-point solution, where task WCRT is sum of its WCET, blocking time caused by LP tasks, and preemption delay caused by HP tasks:
  - $R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$
  - τ_i is schedulable iff $R_i \leq D_i$
  - c.f. RTA for RM scheduling without resource sharing (Slide 35 in "L6-RT Scheduling I"), where $B_i = 0$

Example Taskset (without shared resources)

- System utilization $U = \frac{5}{50} + \frac{250}{500} + \frac{1000}{3000} = 0.933 > 0.780$
  - 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:
  - $R_1 = C_1 + 0 = 5 + 0 = 5 \leq D_1 = 50$
  - $R_2 = C_2 + \left\lceil \frac{R_2}{T_1} \right\rceil \cdot C_1 = 250 + \left\lceil \frac{R_2}{50} \right\rceil \cdot 5 = 280 \leq D_2 = 500$
  - $R_3 = C_3 + \left\lceil \frac{R_3}{T_1} \right\rceil \cdot C_1 + \left\lceil \frac{R_3}{T_2} \right\rceil \cdot C_2 = 1000 + \left\lceil \frac{R_3}{50} \right\rceil \cdot 5 + \left\lceil \frac{R_3}{500} \right\rceil \cdot 250 = 2500 \leq D_3 = 3000$

Task	T	D	C	Prio	R
1	50	50	5	H	5
2	500	500	250	M	280
3	3000	3000	1000	L	2500

Example Taskset (with shared resources under PCP) I

- 3 semaphores s$_1$, s$_2$, s$_3$
  - Task 1 requires semaphore s$_1$, with CS length 1
  - Task 2 requires semaphores s$_2$ and s$_3$, with CS lengths 2 and 5, respectively
  - Task 3 requires semaphores s$_2$ and s$_3$, with CS lengths 3 and 4, respectively
- Ceilings $C(s_1) = P_1 = H;\ C(s_2) = C(s_3) = \max(P_2, P_3) = M$
- Blocking times:
  - Task 1: $B_1 = 0$ (Task 1 does not experience any blocking since its priority is higher than ceilings of s$_2$ and s$_3$: $P_1 > C(s_2) = C(S_3) = M$), so it remains schedulable
  - Task 2: $B_2 = \max(3, 4) = 4$ (maximum CS length of LP Task 3 since $P_2 \leq C(s_2) = C(S_3) = M$)
    - » Utilization $U_2 = \sum_{\forall j \in hp(2)} \frac{C_j}{T_j} + \frac{C_2+B_2}{T_2} = \frac{5}{50} + \frac{250+4}{500} = 0.608 \leq 0.828$ (utilization bound for 2 tasks under RM)
    - » Or WCRT: $R_2 = C_2 + B_2 + \left\lceil \frac{R_2}{T_1} \right\rceil \cdot C_1 = 250 + 4 + \left\lceil \frac{R_2}{50} \right\rceil \cdot 5 = 284 \leq D_2 = 500$
  - Task 3: $B_3 = 0$ (Task 3 is the lowest priority task, so it does not experience any blocking), so it remains schedulable
- The taskset remains schedulable with shared resources under PCP

Task	T	D	C	Prio	sems	CS Len	B	R
1	50	50	5	H	s$_1$	1	0	5
2	500	500	250	M	s$_2$, s$_3$	2, 5	4	284
3	3000	3000	1000	L	s$_2$, s$_3$	3, 4	0	2500

sem	Ceiling
s$_1$	H
s$_2$	M
s$_3$	M

Example Taskset (with shared resources under PCP) II

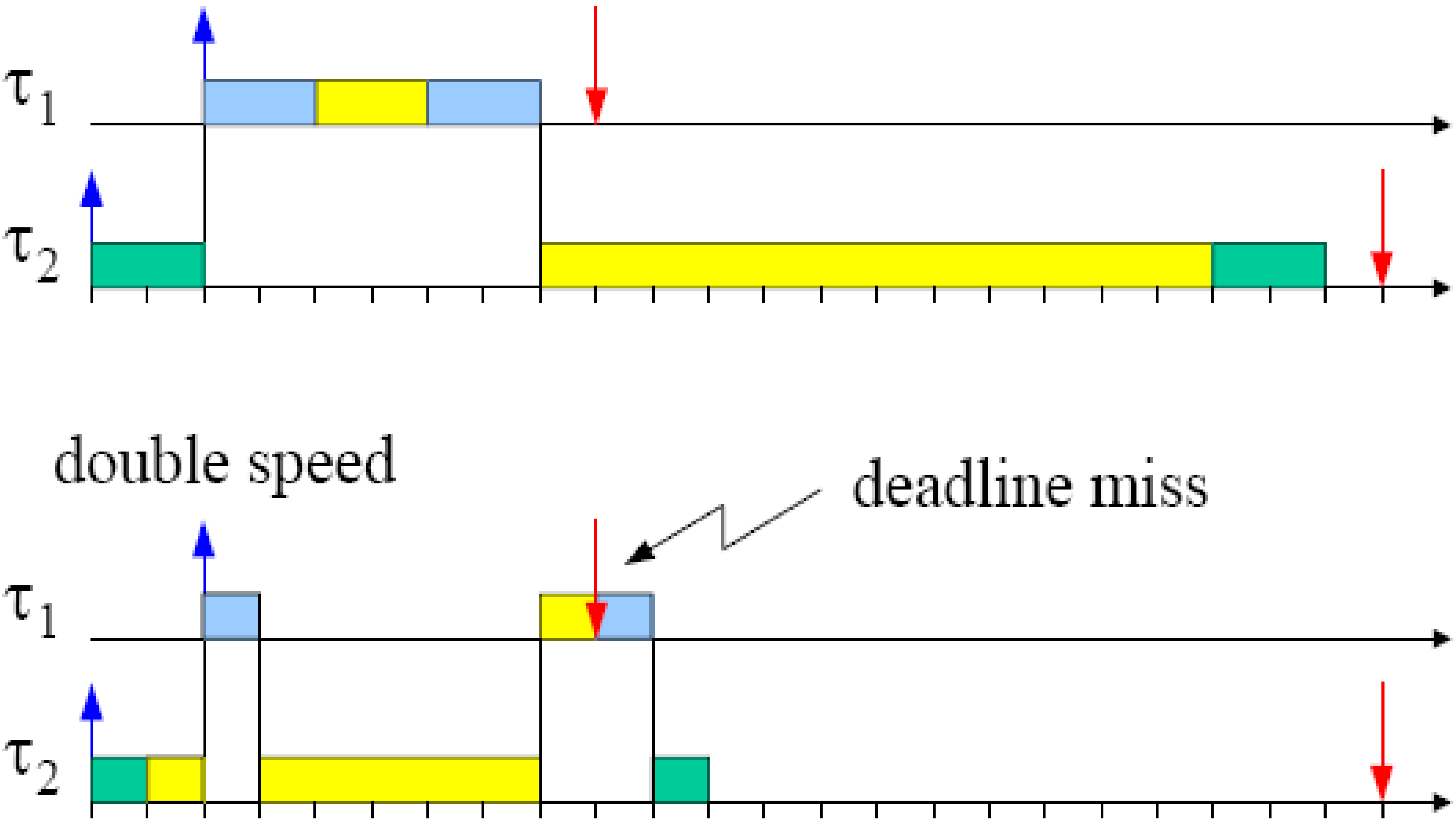
- 3 semaphores s_1, s_2, s_3
  - Task 1 requires semaphores s_1, s_2 and s_3 with CS lengths 1, 1, 1
  - Task 2 requires semaphores s_2 and s_3, with CS lengths 2 and 5, respectively
  - Task 3 requires semaphores s_2 and s_3, with CS lengths 3 and 4, respectively
- Ceilings $C(s_1) = P_1 = H;\ C(s_2) = C(s_3) = \max(P_1, P_2, P_3)\ = H$
- Blocking times:
  - Task 1: $B_1 = \max(2, 5, 3, 4) = 5$ (maximum CS length of LP Tasks 2 and 3, since $P_1 \le C(s_2) = C(S_3) = H$)
    - » Utilization $U_1 = \sum_{\forall j \in hp(1)} \frac{C_j}{T_j} + \frac{C_1+B_1}{T_1} = 0 + \frac{5+5}{50} = 0.2 \le 1$ (Utilization bound for 1 task under RM), so Task 1 remains schedulable
    - » Or WCRT: $R_1 = C_1 + B_1 = 5 + 5 = 10 \le D_1 = 50$
    - » (Task 1's CS lengths (1, 1, 1) do not matter since it is the highest priority task and does not block any other task)
  - Task 2: $B_2 = \max(3, 4) = 4$ (maximum CS length of LP Task 3, since $P_2 \le C(s_2) = C(S_3) = H$)
  - Task 3: $B_3 = 0$ (Task 3 is the lowest priority task, so it does not experience blocking), so it remains schedulable
    - » Same calculation of utilization and WCRT for Tasks 2 and 3 as before
- The taskset remains schedulable with shared resources under PCP

Task	T	D	C	Prio	sems	CS Len	B	R
1	50	50	5	H	s_1,s_2,s_3	1, 1, 1	5	10
2	500	500	250	M	s_2, s_3	2, 5	4	284
3	3000	3000	1000	L	s_2, s_3	3, 4	0	2500

sem	Ceiling
s_1	H
s_2	H
s_3	H

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)



τ_1
τ_2
double speed
deadline miss
τ_1
τ_2