
**Sharif University of Technology**
**Department of Computer Science and Engineering**

Lec. 8:

# Limited Preemptive Scheduling

## Real-Time Computing

S. Safari

2023

# Introduction

- The question whether preemptive systems are better than non-preemptive systems has been debated for a long time, but only partial answers have been provided in the real-time literature and some issues still remain open to discussion.
  - Each approach has advantages and disadvantages, and no one dominates the other when both predictability and efficiency have to be taken into account in the system design.
- This lecture presents and compares some existing approaches for reducing preemptions and describes an efficient method for minimizing preemption costs by removing unnecessary preemptions while preserving system schedulability.

# Introduction (2)

- In particular, the following issues have to be taken into account when comparing the two approaches:
  - In many practical situations, such as I/O scheduling or communication in a shared medium, either preemption is impossible or prohibitively expensive.
  - Preemption destroys program locality, increasing the runtime overhead due to cache misses and pre-fetch mechanisms. As a consequence, worst-case execution times (WCETs) are more difficult to characterize and predict [LHS+98, RM06, RM08, RM09].
  - Non-preemptive execution allows using stack sharing techniques [Bak91] to save memory space in small embedded systems with stringent memory constraints [GAGB01].

# Introduction (3)

- The mutual exclusion problem is trivial in non-preemptive scheduling, which naturally guarantees the exclusive access to shared resources. On the contrary, to avoid unbounded priority inversion, preemptive scheduling requires the implementation of specific concurrency control protocols for accessing shared resources, as those presented in Lecture 6, which introduce additional overhead and complexity.
- In control applications, the input-output delay and jitter are minimized for all tasks when using a non-preemptive scheduling discipline, since the interval between start time and finishing time is always equal to the task computation time [BC07]. This simplifies control techniques for delay compensation at design time.

# Introduction (4)

- Arbitrary preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, so degrading system predictability.
- At least four different types of costs need to be taken into account at each preemption:
  1. **Scheduling cost:** It is the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task.
  2. **Pipeline cost:** It accounts for the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed.

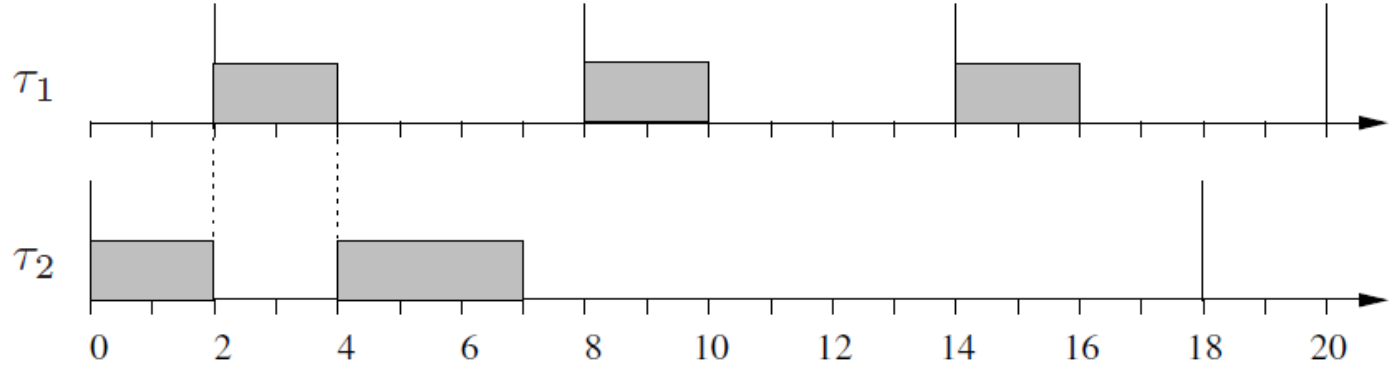
# Introduction (5)

3. **Cache-related cost:** It is the time taken to reload the cache lines evicted by the preempting task. This time depends on the specific point in which preemption occurs and on the number of preemptions experienced by the task [AG08, GA07].
4. **Bus-related cost:** It is the extra bus interference for accessing the RAM due to the additional cache misses caused by preemption.

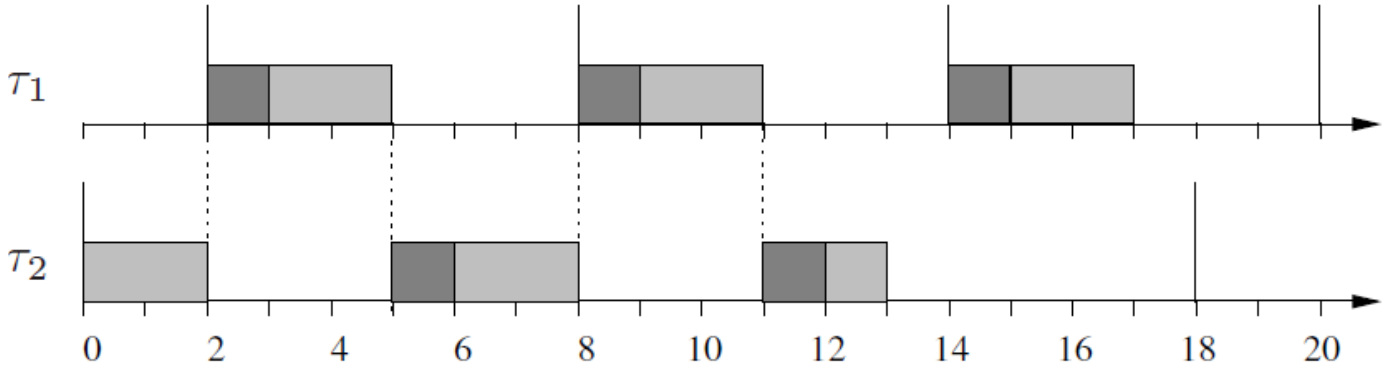
- The cumulative execution overhead due to the combination of these effects is referred to as *Architecture related cost*.
  - Unfortunately, this cost is characterized by a high variance and depends on the specific point in the task code when preemption takes place [AG08, GA07, LDS07].

# An Example of Schedule with preemption cost

- A simple example in which neglecting preemption cost task τ2 experiences a single preemption.



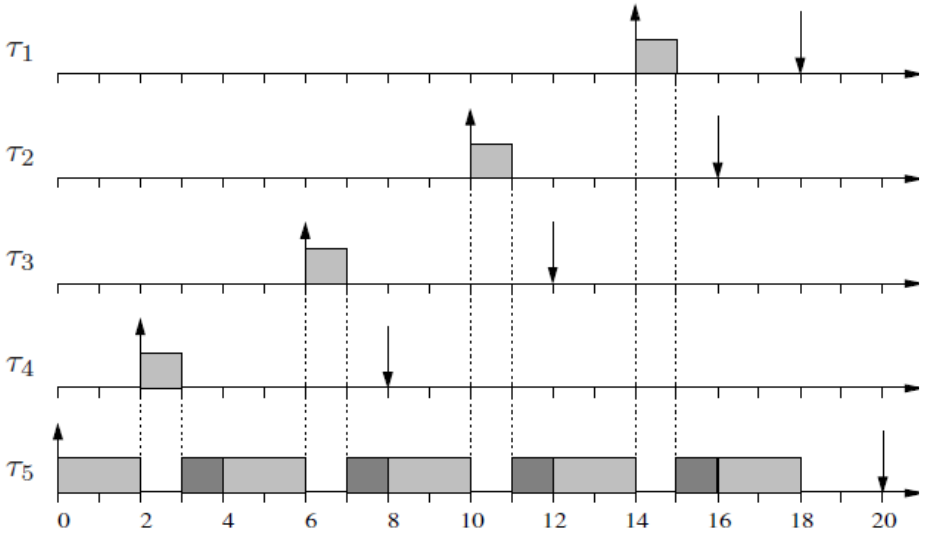

(a) Schedule without preemption cost.



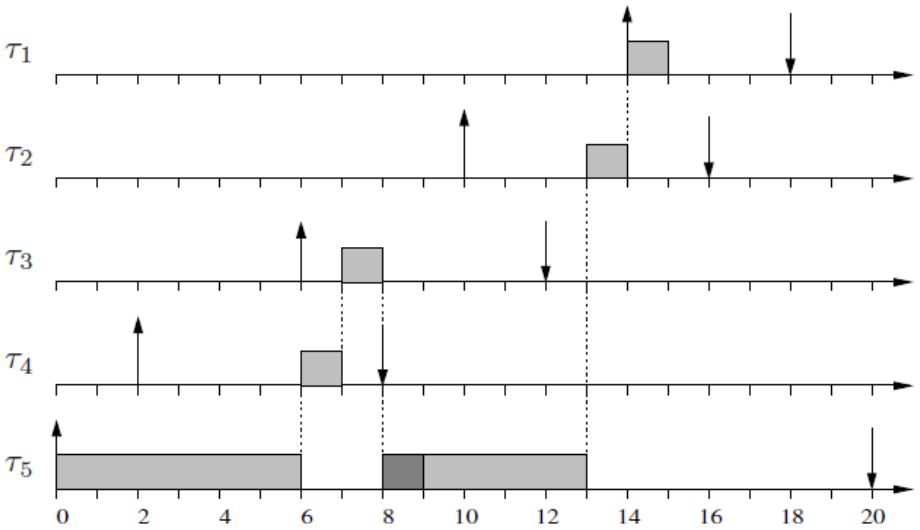

(b) Schedule with preemption cost.

# An Example of Unnecessary Preemptions

- An example in which, under fully preemptive scheduling, task $\tau_5$ is preempted four times.




(a) $\tau_5$ is preempted 4 times.




(b) Only one preemption is really necessary for $\tau_5$.

# Reducing the Runtime Overhead due to Preemptions

- To reduce the runtime overhead due to preemptions and still preserve the schedulability of the task set, the following approaches have been proposed in the literature:
  - **Preemption Thresholds:** According to this approach, a task is allowed to disable preemption up to a specified priority level, which is called preemption threshold. Thus, each task is assigned a regular priority and a preemption threshold, and the preemption is allowed to take place only when the priority of the arriving task is higher than the threshold of the running task.

# Reducing the Runtime Overhead due to Preemptions (2)

- **Deferred Preemptions:** According to this method, each task $\tau_i$ specifies the longest interval $q_i$ that can be executed non-preemptively. This model can come in two slightly different flavors:

  1. *Floating model*. In this model, non-preemptive regions are defined by the programmer by inserting specific primitives in the task code that disable and enable preemption.

  2. *Activation-triggered model*. In this model, non-preemptive regions are triggered by the arrival of a higher priority task and enforced by a timer to last for *qi* units of time after which preemption is enabled.

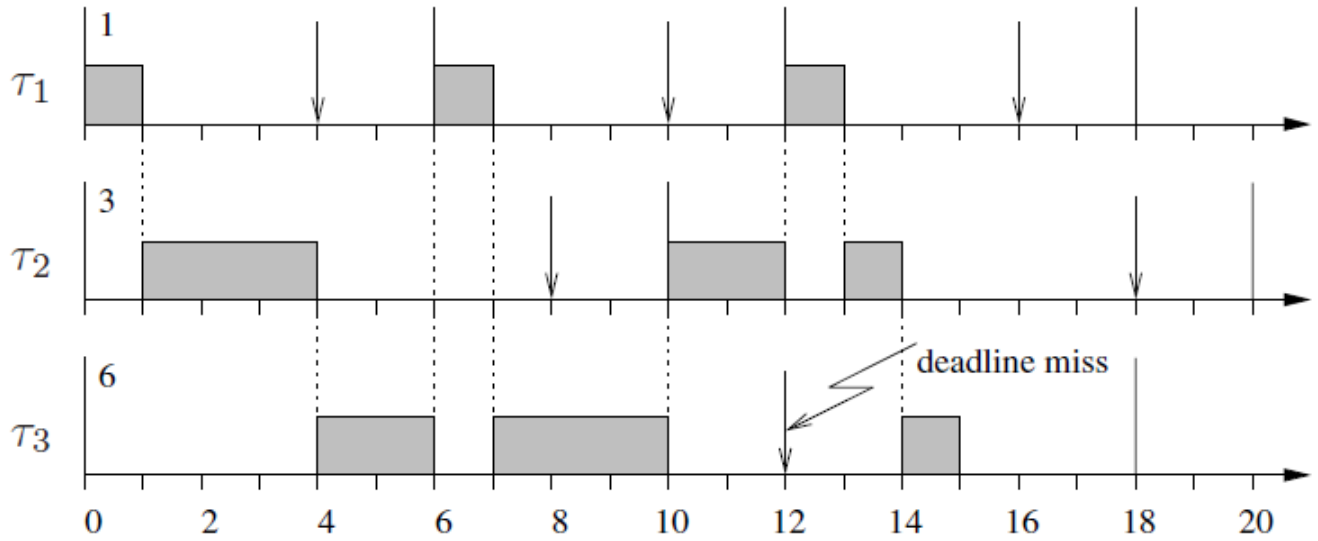
# Reducing the Runtime Overhead due to Preemptions (3)

- **Task splitting:** According to this approach, a task implicitly executes in non-preemptive mode and preemption is allowed only at predefined locations inside the task code, called *preemption points*. In this way, a task is divided into a number of non-preemptive chunks (also called subjobs). If a higher priority task arrives between two preemption points of the running task, preemption is postponed until the next preemption point. This approach is also referred to as *cooperative scheduling*, because tasks cooperate to offer suitable preemption points to improve schedulability.

# An Example of Unnecessary Preemptions

- To better understand the different limited preemptive approaches, the task set reported in following table will be used as a common example throughout this lecture.
- Schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set.

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 1 | 6 | 4 |
| $\tau_2$ | 3 | 10 | 8 |
| $\tau_3$ | 6 | 18 | 12 |



# Terminology and Notation

- A set of $n$ periodic or sporadic real-time tasks will be considered to be scheduled on a single processor.
- Each task $\tau_i$ is characterized by a worst-case execution time (WCET) $C_i$, a relative deadline $D_i$, and a period (or minimum inter-arrival time) $T_i$.
- A constrained deadline model is adopted, so $D_i$ is assumed to be less than or equal to $T_i$.
- For scheduling purposes, each task is assigned a fixed priority $P_i$, used to select the running task among those tasks ready to execute.
- Tasks are indexed by decreasing priority, that is, $\forall i \mid 1 \leq i < n : P_i > P_{i+1}$.

# Non-Preemptive Scheduling

- The most effective way to reduce preemption cost is to disable preemptions completely.
- In this condition, however, each task $\tau_i$ can experience a blocking time $B_i$ equal to the longest computation time among the tasks with lower priority. That is:

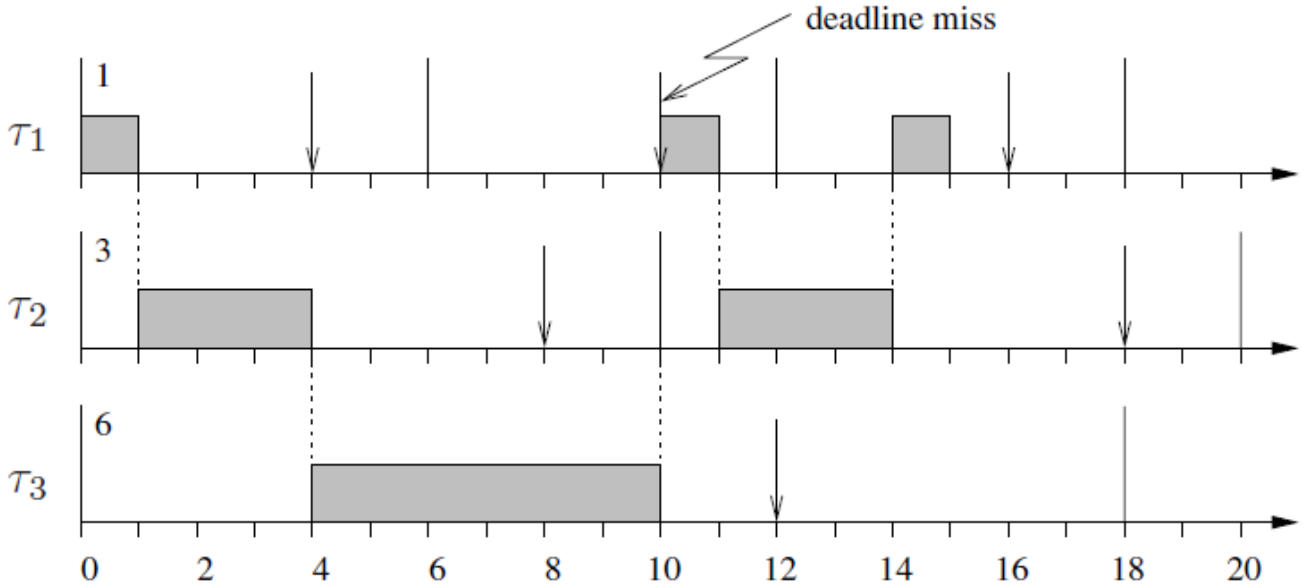
$$B_i = \max_{j:P_j<P_i} \{C_j - 1\}$$

- where the maximum of an empty set is assumed to be zero. Note that one unit of time must be subtracted from $C_j$ to consider that to block $\tau_i$, the blocking task must start at least one unit before the critical instant.
- Such a blocking term introduces an additional delay before task execution, which could jeopardize schedulability.
- High priority tasks are those that are most affected by such a blocking delay, since the maximum in the above equation is computed over a larger set of tasks.

# An Example of Non-Preemptive Deadline Monotonic

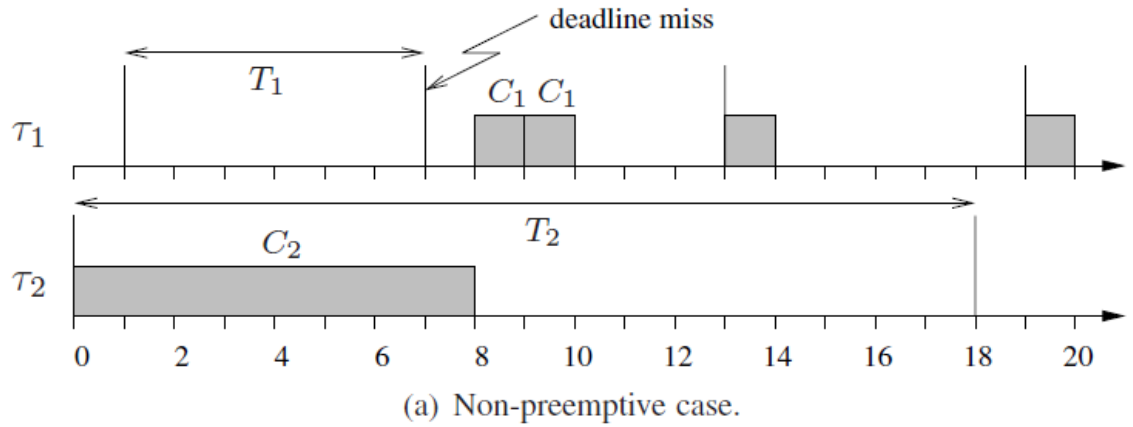
- The schedule generated by Deadline Monotonic on the task set of the mentioned table when preemptions are disabled.

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 1 | 6 | 4 |
| $\tau_2$ | 3 | 10 | 8 |
| $\tau_3$ | 6 | 18 | 12 |

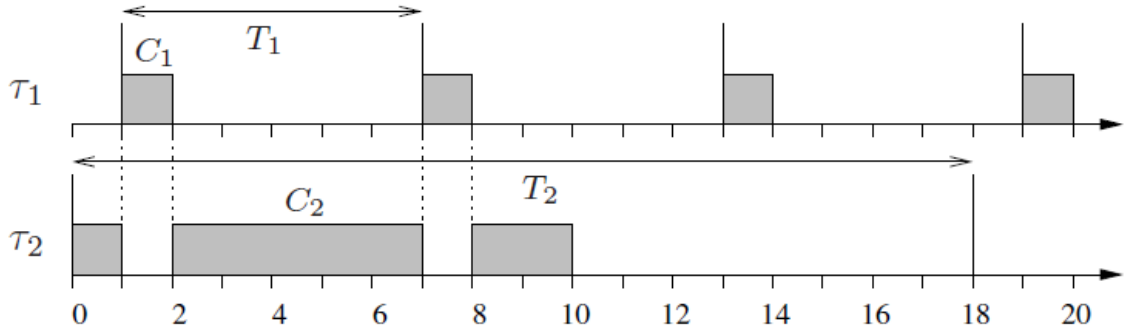


# The least upper bounds of both RM and EDF drop to zero!

- This means that there are task sets with arbitrary low utilization that cannot be scheduled by RM and EDF when preemptions are disabled.



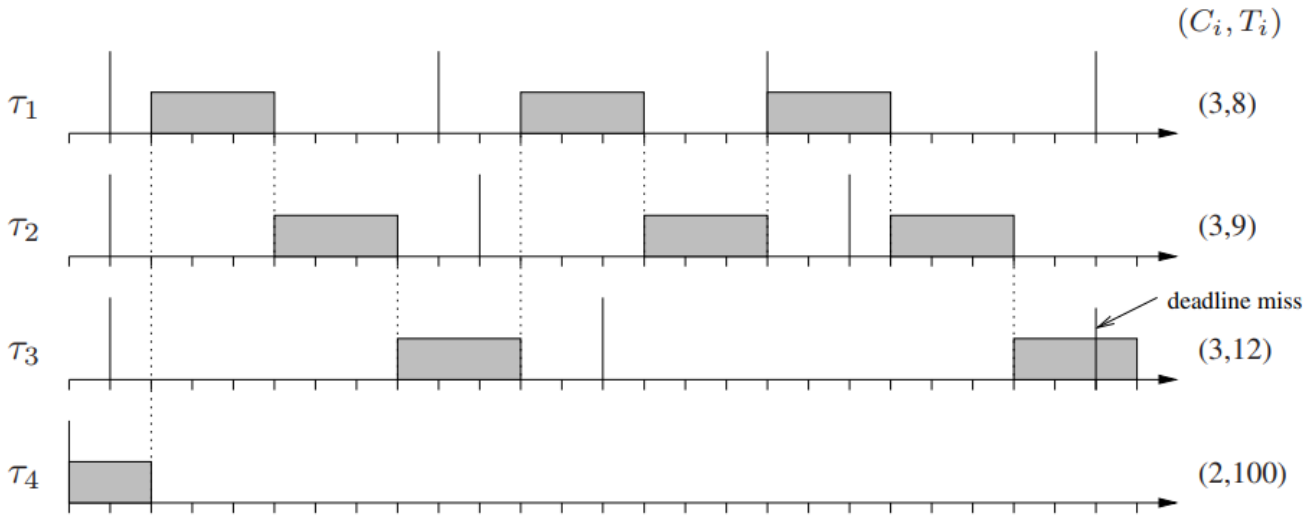

(a) Non-preemptive case.




(b) Preemptive case.

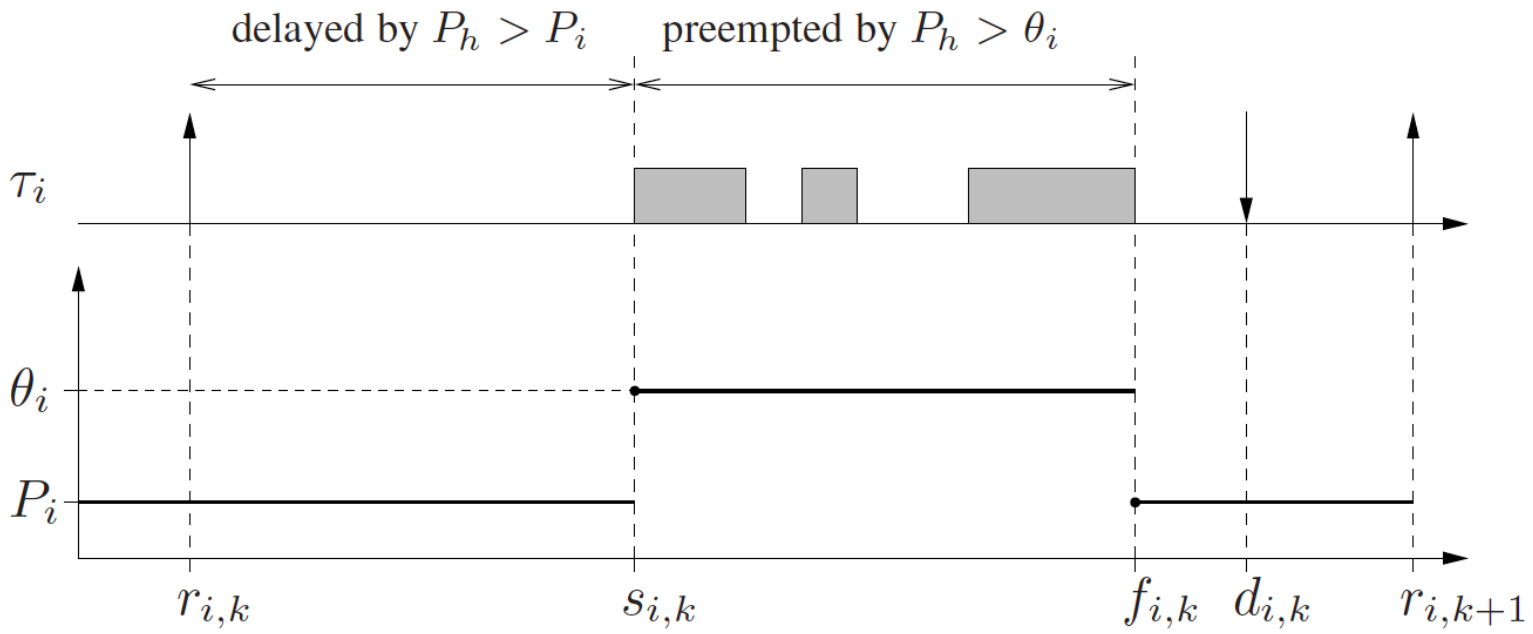
# Self-pushing phenomenon

- The high priority jobs activated during the non-preemptive execution of $\tau_i$'s first instance are pushed ahead to successive jobs, which then may experience a higher interference.



# Preemption Thresholds

- According to this model, proposed by Wang and Saksena [WS99], each task $\tau_i$ is assigned a nominal priority $P_i$ (used to enqueue the task into the ready queue and to preempt) and a *preemption threshold* $\theta_i \geq P_i$ (used for task execution).
- Then, $\tau_i$ can be preempted by $\tau_h$ only if $P_h > \theta_i$.



# Preemption Thresholds (2)

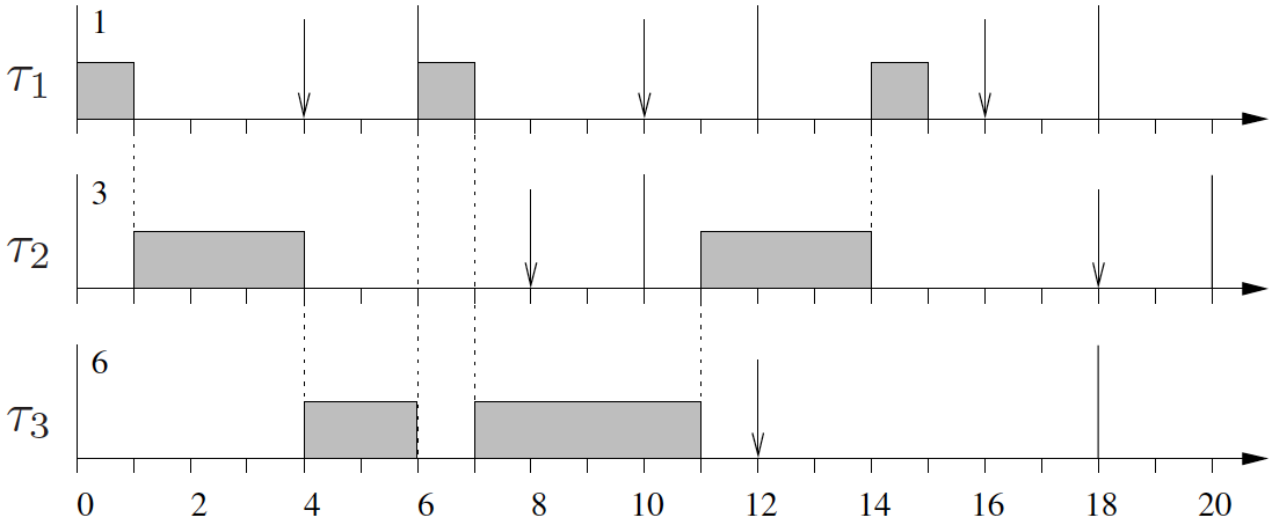
- Preemption threshold can be considered as a trade-off between fully preemptive and fully non-preemptive scheduling.
  - If each threshold priority is set equal to the task nominal priority, the scheduler behaves like the fully preemptive scheduler;
  - If all thresholds are set to the maximum priority, the scheduler runs in non-preemptive fashion.
- By appropriately setting the thresholds, the system can achieve a higher utilization efficiency compared with fully preemptive and fully non-preemptive scheduling.

# An Example of Preemption Thresholds

- Assigning the preemption thresholds shown in following table, the task set of Slide 12 results to be schedulable by Deadline Monotonic.

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 1 | 6 | 4 |
| $\tau_2$ | 3 | 10 | 8 |
| $\tau_3$ | 6 | 18 | 12 |

| | $P_i$ | $\theta_i$ |
|---|---|---|
| $\tau_1$ | 3 | 3 |
| $\tau_2$ | 2 | 3 |
| $\tau_3$ | 1 | 2 |



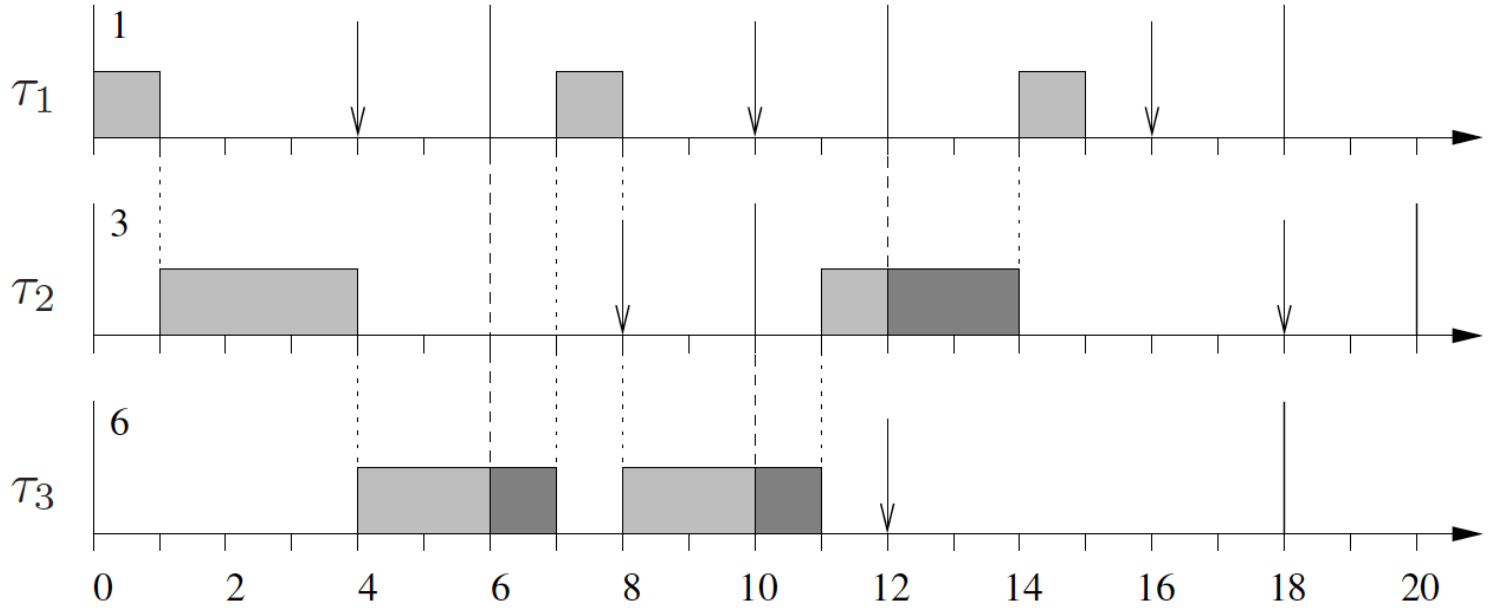
# Deferred Preemptions

- According to this method, each task $\tau_i$ defines a maximum interval of time $q_i$ in which it can execute non-preemptively.
- Depending on the specific implementation, the non-preemptive interval can start after the invocation of a system call inserted at the beginning of a non-preemptive region (floating model), or can be triggered by the arrival of a higher priority task (activation-triggered model).
- Under the floating model, preemption is resumed by another system call, inserted at the end of the region (long at most $q_i$ units).
- Under the activation-triggered model, preemption is enabled by a timer interrupt after exactly $q_i$ units (unless the task completes earlier).

# An Example of Deferred Preemptions

- Assigning $q_2$=2 and $q_3$=1, the schedule produced by Deadline Monotonic with deferred preemptions is feasible.

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 1 | 6 | 4 |
| $\tau_2$ | 3 | 10 | 8 |
| $\tau_3$ | 6 | 18 | 12 |



# Task Splitting

- According to this model, each task $\tau_i$ is split into $m_i$ non-preemptive chunks (subjobs), obtained by inserting $m_i - 1$ preemption points in the code.
- Preemptions can only occur at the subjobs boundaries.
- All the jobs generated by one task have the same subjob division.
- The $k^{th}$ subjob has a worst-case execution time $q_{i,k}$; hence:

$$C_i = \sum_{k=1}^{m_i} q_{i,k}.$$

- Among all the parameters describing the subjobs of a task, two values are of particular importance for achieving a tight schedulability result:

$$\begin{cases} q_i^{max} = \max\limits_{k \in [1, m_i]} \{q_{i,k}\} \\ q_i^{last} = q_{i,m_i} \end{cases}$$

# Task Splitting (2)

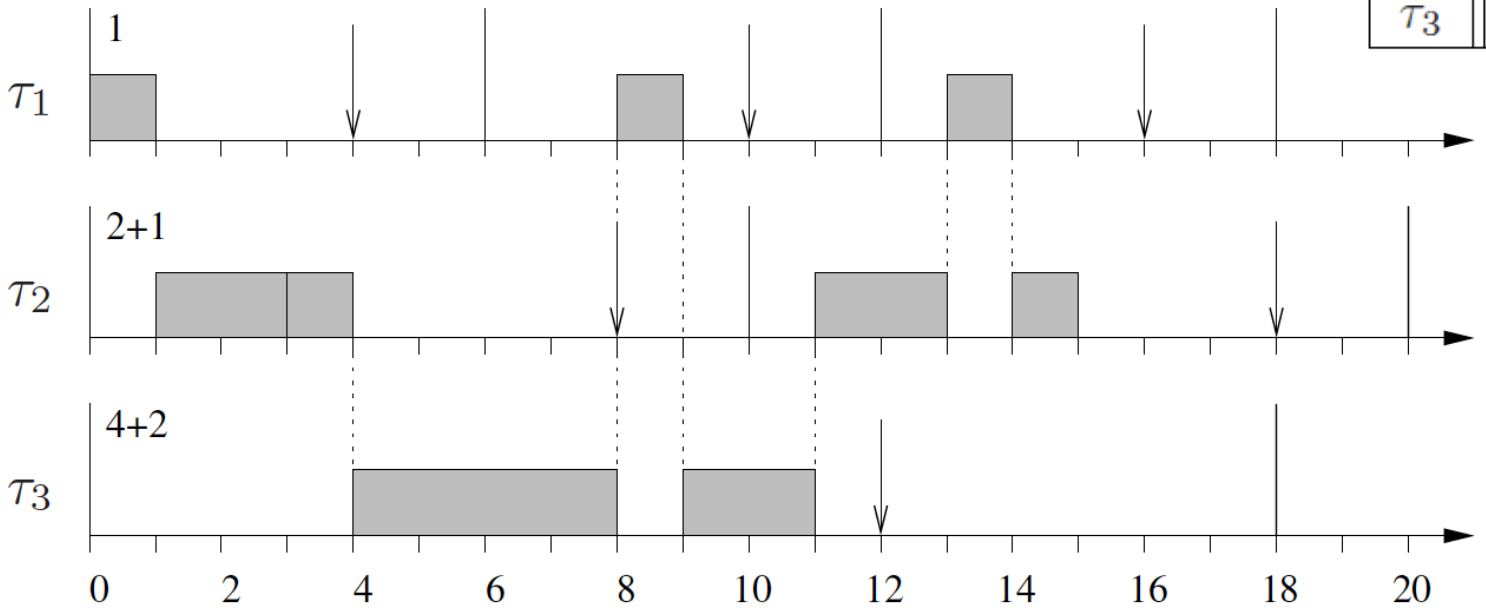
- In fact, the feasibility of a high priority task $\tau_k$ is affected by the size $q_j^{max}$ of the longest subjob of each task $\tau_j$ with priority $P_j < P_k$.
- Moreover, the length $q_i^{last}$ of the final subjob of $\tau_i$ directly affects its response time.
  - All higher priority jobs arriving during the execution of $\tau_i$'s final subjob do not cause a preemption, since their execution is postponed at the end of $\tau_i$.
- In this model, each task will be characterized by the following 5-tuple:

$$\{C_i, D_i, T_i, q_i^{max}, q_i^{last}\}.$$

# An Example of Task Splitting

- Suppose that $\tau_2$ is split in two subjobs of 2 and 1 unit.
- Suppose that $\tau_3$ is split in two subjobs of 4 and 2 units.
- The schedule produced by Deadline Monotonic with such a splitting is feasible.

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 1 | 6 | 4 |
| $\tau_2$ | 3 | 10 | 8 |
| $\tau_3$ | 6 | 18 | 12 |



# Summary

- Introduction to limited preemptive scheduling
- Reducing the runtime overhead due to preemptions
  - Preemption thresholds
  - Deferred preemptions
  - Task splitting

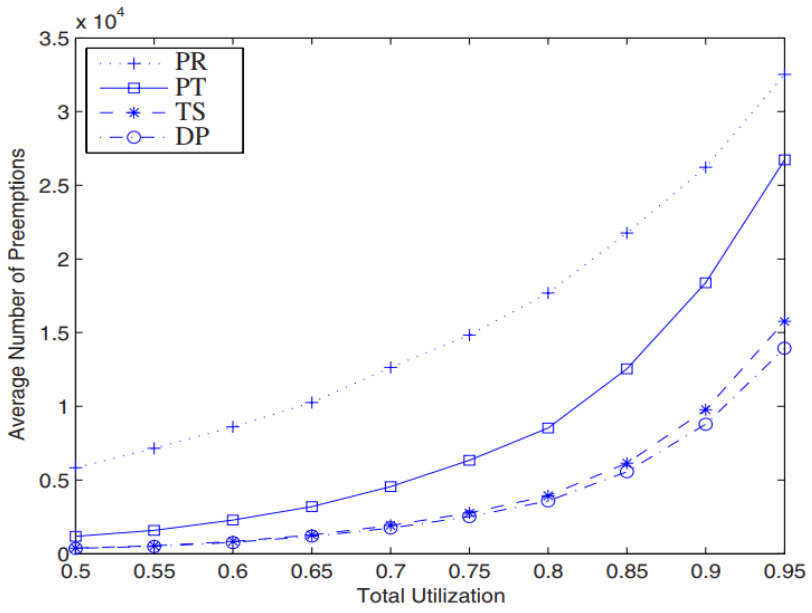
| | Implementation cost | Predictability | Effectiveness |
|---|---|---|---|
| Preemption Thresholds | Low | Low | Medium |
| Deferred Preemptions | Medium | Medium | High |
| Cooperative Scheduling | Medium | High | High |

# Summary (cont'd)

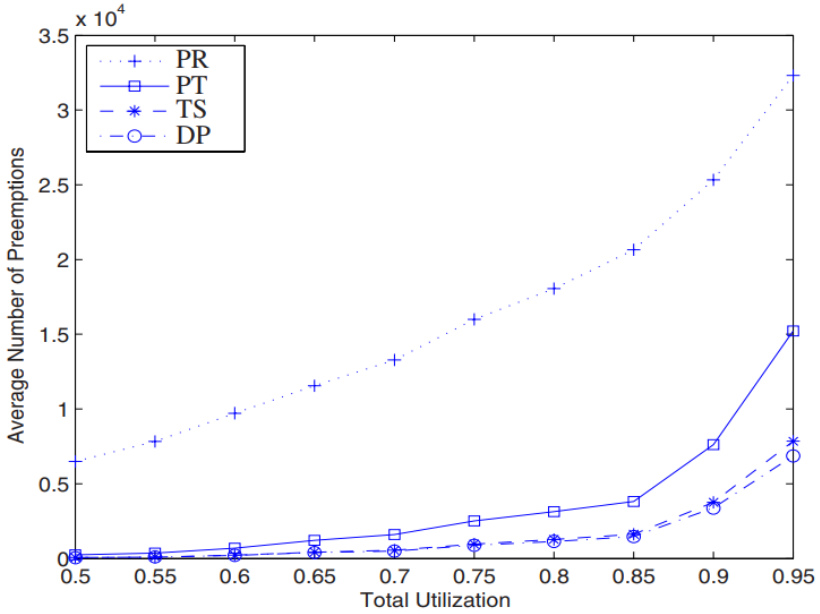
- Simulations experiments with randomly generated task sets have been carried out by Yao, Buttazzo, and Bertogna [YBB10b] to better evaluate the effectiveness of the considered algorithms in reducing the number of preemptions.
- The simulation results obtained for a task set of 6 and 12 tasks, respectively, and report the number of preemptions produced by each method as a function of the load.
- Each simulation run was performed on a set of $n$ tasks with total utilization $U$ varying from 0.5 to 0.95 with step 0.05.
- Individual utilizations $Ui$ were uniformly distributed in [0,1], using the UUniFast algorithm.
- Each computation time $Ci$ was generated as a random integer uniformly distributed in [10, 50], and then $Ti$ was computed as $Ti = Ci/Ui$. The relative deadline $Di$ was generated as a random integer in the range $[Ci + 0.8 \cdot (Ti - Ci), Ti]$.

# Summary (cont'd)

- Fully preemptive scheduling (PR)
- Preemption thresholds (PT)
- Deferred preemptions (DP)
- Task splitting (TS)




(a) Number of tasks: $n$=6




(b) Number of tasks: $n$=12