8

LIMITED PREEMPTIVE SCHEDULING

8.1 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. In fact, 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 chapter 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.

Preemption is a key factor in real-time scheduling algorithms, since it allows the operating system to immediately allocate the processor to incoming tasks with higher priority. In fully preemptive systems, the running task can be interrupted at any time by another task with higher priority, and be resumed to continue when all higher priority tasks have completed. In other systems, preemption may be disabled for certain intervals of time during the execution of critical operations (e.g., interrupt service routines, critical sections, and so on.). In other situations, preemption can be completely forbidden to avoid unpredictable interference among tasks and achieve a higher degree of predictability (although higher blocking times).

The question to enable or disable preemption during task execution has been investigated by many authors with several points of view and it has not a trivial answer. A general disadvantage of the non-preemptive discipline is that it introduces an additional blocking factor in higher priority tasks, thus reducing schedulability. On the other hand, however, there are several advantages to be considered when adopting a non-preemptive scheduler.

G.C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Real-Time Systems Series 24, DOI 10.1007/978-1-4614-0676-1_8,

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].
- 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 Chapter 7, 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.
- Non-preemptive execution allows using stack sharing techniques [Bak91] to save memory space in small embedded systems with stringent memory constraints [GAGB01].

In summary, arbitrary preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, so degrading system predictability. In particular, 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.
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].

Bui et al. [BCSM08] showed that on a PowerPC MPC7410 with 2 MByte two-way associative L2 cache the WCET increment due to cache interference can be as large as 33% of the WCET measured in non-preemptive mode.

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].

The total increase of the worst-case execution time of a task τ_i is also a function of the total number of preemptions experienced by τ_i, which in turn depends on the task set parameters, on the activation pattern of higher priority tasks, and on the specific scheduling algorithm. Such a circular dependency of WCET and number of preemptions makes the problem not easy to be solved. Figure 8.1(a) shows a simple example in which neglecting preemption cost task τ_2 experiences a single preemption. However, when taking preemption cost into account, τ_2's WCET becomes higher, and hence τ_2 experiences additional preemptions, which in turn increase its WCET. In Figure 8.1(b) the architecture related cost due to preemption is represented by dark gray areas.

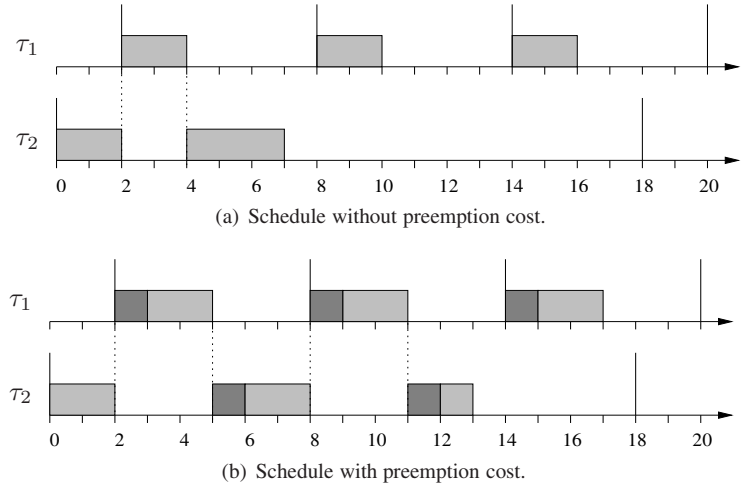


Figure 8.1 Task τ_2 experiences a single preemption when preemption cost is neglected, and two preemptions when preemption cost is taken into account.

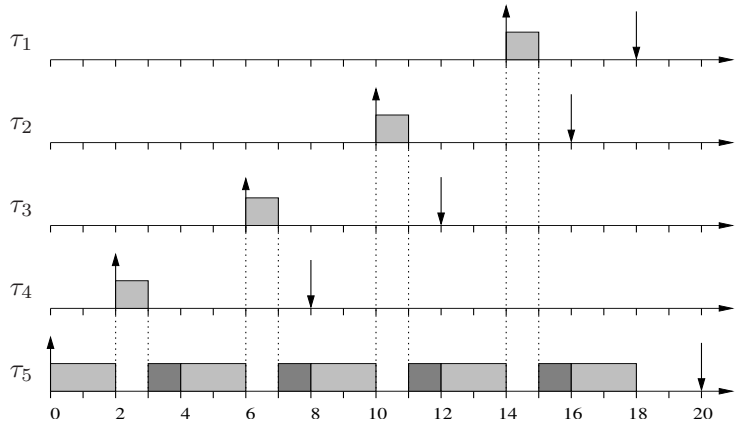
Some methods for estimating the number of preemptions have been proposed [ERC95, YS07], but they are restricted to the fully preemptive case and do not consider such a circular dependency.

Often, preemption is considered a prerequisite to meet timing requirement in real-time system design; however, in most cases, a fully preemptive scheduler produces many unnecessary preemptions. Figure 8.2(a) illustrates an example in which, under fully preemptive scheduling, task τ_5 is preempted four times. As a consequence, the WCET of τ_5 is substantially inflated by the architecture related cost (represented by dark gray areas), causing a response time equal to $R_5 = 18$. However, as shown in Figure 8.2(b), only one preemption is really necessary to guarantee the schedulability of the task set, reducing the WCET of τ_5 from 14 to 11 units of time, and its response time from 18 to 13 units.

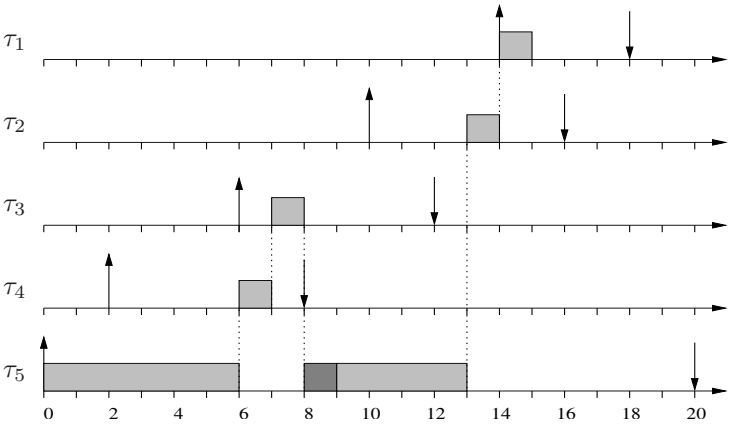
To reduce the runtime overhead due to preemptions and still preserve the schedulability of the task set, the following approaches have been proposed in the literature.

- *Preemption Thresholds*. According to this approach, proposed by Wang and Saksena [WS99], 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.
- *Deferred Preemptions*. According to this method, each task τ_i specifies the longest interval q_i that can be executed non-preemptively. Depending on how non-preemptive regions are implemented, 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. Since the start time of each region is not specified in the model, non-preemptive regions cannot be identified off-line and, for the sake of the analysis, are considered to be "floating" in the code, with a duration $\delta_{i,k} \leq q_i$.
  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 q_i units of time (unless the task finishes earlier), after which preemption is enabled. Once a timer is set at time t, additional activations arriving before the timeout $(t + q_i)$ do not postpone the preemption any further. After the timeout, a new high-priority arrival can trigger another non-preemptive region.



(a) τ_5 is preempted 4 times.



(b) Only one preemption is really necessary for τ_5.

Figure 8.2 Fully preemptive scheduling can generate several preemptions (a) although only a few of them are really necessary to guarantee the schedulability of the task set (b).

- *Task splitting*. According to this approach, investigated by Burns [Bur94], 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.

To better understand the different limited preemptive approaches, the task set reported in Table 8.1 will be used as a common example throughout this chapter.

	C_i	T_i	D_i
τ_1	1	6	4
τ_2	3	10	8
τ_3	6	18	12

Table 8.1 Parameters of a sample task set with relative deadlines less than periods.

Figure 8.3 illustrates the schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set of Table 8.1. Notice that the task set is not schedulable, since task τ_3 misses its deadline.

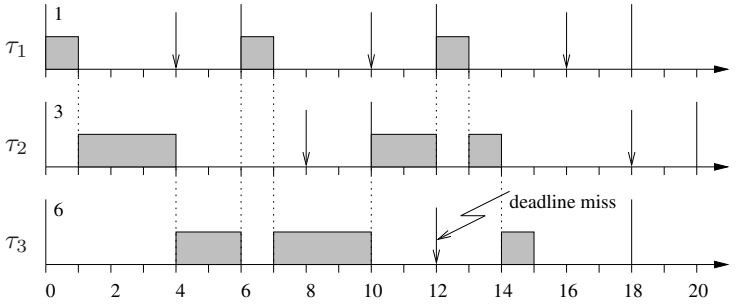


Figure 8.3 Schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set of Table 8.1.

8.1.1 TERMINOLOGY AND NOTATION

Throughout this chapter, a set of n periodic or sporadic real-time tasks will be considered to be scheduled on a single processor. Each task τ_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. A higher value of P_i corresponds to a higher priority. Note that task activation times are not known a priori and the actual execution time of a task can be less than or equal to its worst-case value C_i. Tasks are indexed by decreasing priority, that is, $\forall i \mid 1 \leq i < n : P_i > P_{i+1}$. Additional terminology will be introduced below for each specific method.

8.2 NON-PREEMPTIVE SCHEDULING

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

$$B_i = \max_{j:P_j<P_i} \{C_j - 1\} \tag{8.1}$$

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 τ_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 Equation (8.1) is computed over a larger set of tasks. Figure 8.4 illustrates the schedule generated by Deadline Monotonic on the task set of Table 8.1 when preemptions are disabled. With respect to the schedule shown in Figure 8.3, note that τ_3 is now able to complete before its deadline, but the task set is still not schedulable, since now τ_1 misses its deadline.

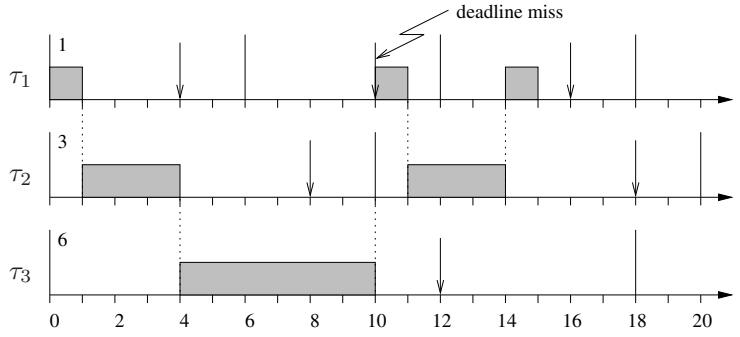


Figure 8.4 Schedule produced by non-preemptive Deadline Monotonic on the task set of Table 8.1.

Unfortunately, under non-preemptive scheduling, 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. For example, the task set illustrated in Figure 8.5(a) is not feasible under non-preemptive Rate Monotonic scheduling (as well as under non-preemptive EDF), since $C_2 > T_1$, but its utilization can be set arbitrarily low by reducing C_1 and increasing T_2. The same task set is clearly feasible when preemption is enabled, as shown in Figure 8.5(b).

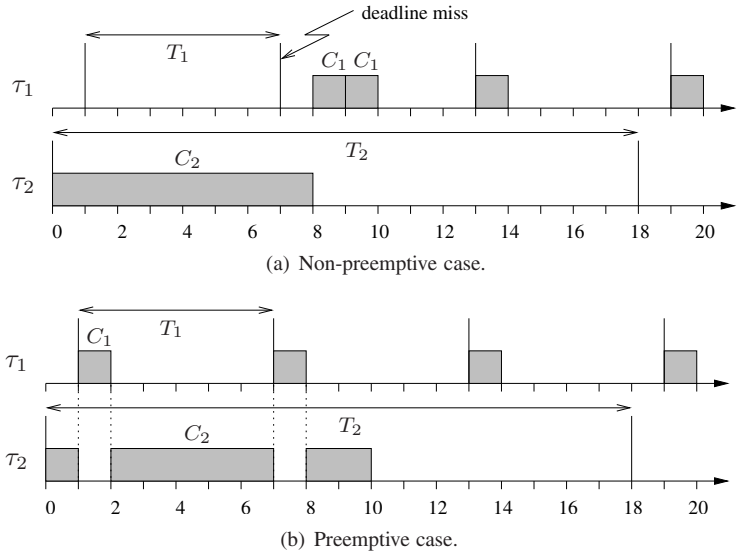


Figure 8.5 A task set with low utilization that is unfeasible under non-preemptive Rate Monotonic scheduling, and feasible when preemption is enabled.

8.2.1 FEASIBILITY ANALYSIS

The feasibility analysis of non-preemptive task sets is more complex than under fully preemptive scheduling. Bril et al. [BLV09] showed that in non-preemptive scheduling the largest response time of a task does not necessarily occur in the first job, after the critical instant. An example of such a situation is illustrated in Figure 8.6, where the worst-case response time of τ_3 occurs in its second instance. Such a scheduling anomaly, identified as *self-pushing phenomenon*, occurs because the high priority jobs activated during the non-preemptive execution of τ_i's first instance are pushed ahead to successive jobs, which then may experience a higher interference.

The presence of the self-pushing phenomenon in non-preemptive scheduling implies that the response time analysis for a task τ_i cannot be limited to its first job, activated at the critical instant, as done in preemptive scheduling, but it must be performed for multiple jobs, until the processor finishes executing tasks with priority higher than or equal to P_i. Hence, the response time of a task τ_i needs to be computed within the longest Level-i Active Period, defined as follows [BLV09]:

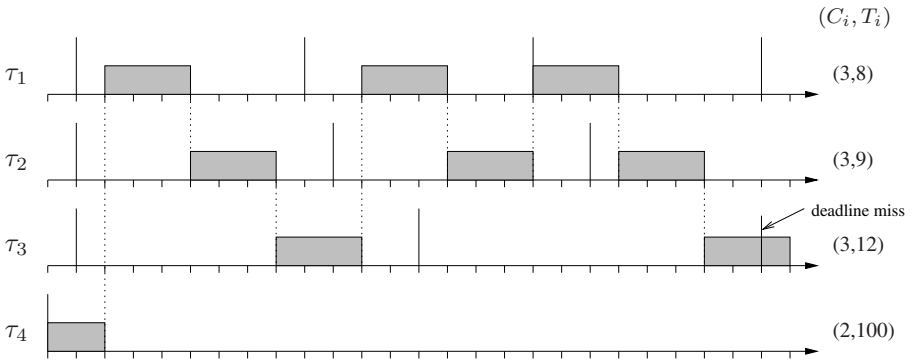


Figure 8.6 An example of self-pushing phenomenon occurring on task τ_3.

Definition 8.1 *The* Level-i pending workload $W_i^p(t)$ *at time* t *is the amount of processing that still needs to be performed at time* t *due to jobs with priority higher than or equal to* P_i *released strictly before* t*.*

Definition 8.2 *A* Level-i Active Period L_i *is an interval* $[a, b)$ *such that the Level-*i *pending workload* $W_i^p(t)$ *is positive for all* $t \in (a, b)$ *and null in* a *and* b*.*

The longest Level-i Active Period can be computed by the following recurrent relation:

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \displaystyle\sum_{h:P_h \geq P_i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{8.2}$$

In particular, L_i is the smallest value for which $L_i^{(s)} = L_i^{(s-1)}$.

This means that the response time of task τ_i must be computed for all jobs $\tau_{i,k}$, with $k \in [1, K_i]$, where

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{8.3}$$

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ can then be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs and the interference of the tasks with priority higher than P_i.

Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} = B_i + \sum_{h:P_h>P_i} C_h \\ s_{i,k}^{(\ell)} = B_i + (k-1)C_i + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.4}$$

Since, once started, the task cannot be preempted, the finishing time $f_{i,k}$ can be computed as

$$f_{i,k} = s_{i,k} + C_i. \tag{8.5}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1,K_i]} \{f_{i,k} - (k-1)T_i\}. \tag{8.6}$$

Once the response time of each task is computed, the task set is feasible if and only if

$$\forall i = 1, \ldots, n \quad R_i \leq D_i. \tag{8.7}$$

Yao, Buttazzo, and Bertogna [YBB10a] showed that the analysis of non-preemptive tasks can be reduced to a single job, under specific (but not too restrictive) conditions.

Theorem 8.1 (Yao, Buttazzo, and Bertogna, 2010) *The worst-case response time of a non-preemptive task occurs in the first job if the task is activated at its critical instant and the following two conditions are both satisfied:*

1. *the task set is feasible under preemptive scheduling;*
2. *relative deadlines are less than or equal to periods.*

Under these conditions, the longest relative start time S_i of task τ_i is equal to $s_{i,1}$ and can be computed from Equation (8.4) for $k = 1$:

$$S_i = B_i + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) C_h. \tag{8.8}$$

Hence, the response time R_i is simply:

$$R_i = S_i + C_i. \tag{8.9}$$

8.3 PREEMPTION THRESHOLDS

According to this model, proposed by Wang and Saksena [WS99], each task τ_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, τ_i can be preempted by τ_h only if $P_h > \theta_i$. Figure 8.7 illustrates how the threshold is used to raise the priority of a task τ_i during the execution of its k-th job. At the activation time $r_{i,k}$, the priority of τ_i is set to its nominal value P_i, so it can preempt all the tasks τ_j with threshold $\theta_j < P_i$. The nominal priority is maintained as long as the task is kept in the ready queue. During this interval, τ_i can be delayed by all tasks τ_h with priority $P_h > P_i$. When all such tasks complete (at time $s_{i,k}$), τ_i is dispatched for execution and its priority is raised at its threshold level θ_i until the task terminates (at time $f_{i,k}$). During this interval, τ_i can be preempted by all tasks τ_h with priority $P_h > \theta_i$. Note that when τ_i is preempted its priority is kept to its threshold level.

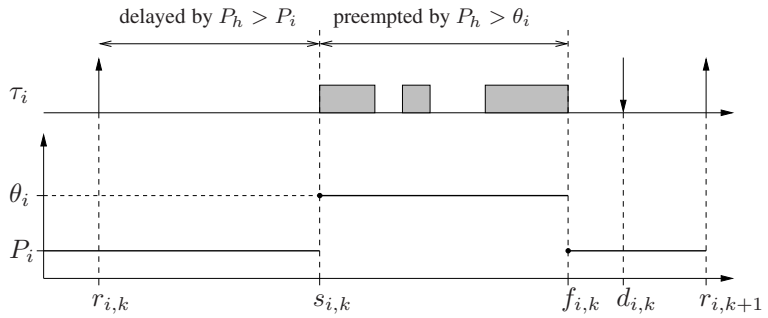


Figure 8.7 Example of task executing under preemption threshold.

Preemption threshold can be considered as a trade-off between fully preemptive and fully non-preemptive scheduling. Indeed, if each threshold priority is set equal to the task nominal priority, the scheduler behaves like the fully preemptive scheduler; whereas, if all thresholds are set to the maximum priority, the scheduler runs in non-preemptive fashion. Wang and Saksena also showed that by appropriately setting the thresholds, the system can achieve a higher utilization efficiency compared with fully preemptive and fully non-preemptive scheduling. For example, assigning the preemption thresholds shown in Table 8.2, the task set of Table 8.1 results to be schedulable by Deadline Monotonic, as illustrated in Figure 8.8. [1]

[1] Note that the task set is not schedulable under sporadic activations; in fact, τ_2 misses its deadline if τ_1 and τ_2 are activated one unit of time after τ_3.

	P_i	θ_i
τ_1	3	3
τ_2	2	3
τ_3	1	2

Table 8.2 Preemption thresholds assigned to the tasks of Table 8.1.

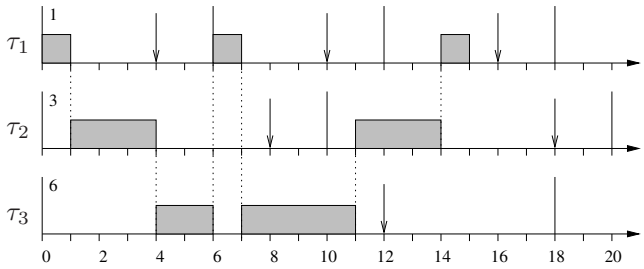


Figure 8.8 Schedule produced by preemption thresholds for the task set in Table 8.1.

Note that at $t = 6$, τ_1 can preempt τ_3 since $P_1 > \theta_3$. However, at $t = 10$, τ_2 cannot preempt τ_3, being $P_2 = \theta_3$. Similarly, at $t = 12$, τ_1 cannot preempt τ_2, being $P_1 = \theta_2$.

8.3.1 FEASIBILITY ANALYSIS

Under fixed priorities, the feasibility analysis of a task set with preemption thresholds can be performed by the feasibility test derived by Wang and Saksena [WS99], and later refined by Regehr [Reg02]. First of all, a task τ_i can be blocked only by lower priority tasks that cannot be preempted by it; that is, by tasks having a priority $P_j < P_i$ and a threshold $\theta_j \geq P_i$. Hence, a task τ_i can experience a blocking time equal to the longest computation time among the tasks with priority lower than P_i and threshold higher than or equal to P_i. That is,

$$B_i = \max_j \{C_j - 1 \mid P_j < P_i \leq \theta_j\} \tag{8.10}$$

where the maximum of an empty set is assumed to be zero. Then, the response time R_i of task τ_i is computed by considering the blocking time B_i, the interference before its start time (due to the tasks with priority higher than P_i), and the interference after its start time (due to tasks with priority higher than θ_i), as depicted in Figure 8.7. The analysis must be carried out within the longest Level-i active period L_i, defined by the following recurrent relation:

$$L_i \;=\; B_i \;+\; \sum_{h:P_h \geq P_i} \left\lceil \frac{L_i}{T_h} \right\rceil C_h. \tag{8.11}$$

This means that the response time of task τ_i must be computed for all the jobs $\tau_{i,k}$ ($k = 1, 2, \ldots$) within the longest Level-i active period. That is, for all $k \in [1, K_i]$, where

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{8.12}$$

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ can be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs, and the interference of the tasks with priority higher than P_i. Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} \;=\; B_i \;+ \sum\limits_{h:P_h > P_i} C_h \\ s_{i,k}^{(\ell)} \;=\; B_i \;+ (k-1)C_i \;+ \sum\limits_{h:P_h > P_i} \left(\left\lfloor \dfrac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.13}$$

For the same job $\tau_{i,k}$, the finishing time $f_{i,k}$ can be computed by summing to the start time $s_{i,k}$ the computation time of job $\tau_{i,k}$, and the interference of the tasks that can preempt $\tau_{i,k}$ (those with priority higher than θ_i). That is,

$$\begin{cases} f_{i,k}^{(0)} \;=\; s_{i,k} \;+\; C_i \\ f_{i,k}^{(\ell)} \;=\; s_{i,k} \;+\; C_i \;+ \sum\limits_{h:P_h > \theta_i} \left(\left\lceil \dfrac{f_{i,k}^{(\ell-1)}}{T_h} \right\rceil - \left(\left\lfloor \dfrac{s_{i,k}}{T_h} \right\rfloor + 1 \right) \right) C_h. \end{cases} \tag{8.14}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1, K_i]} \{ f_{i,k} - (k-1)T_i \}. \tag{8.15}$$

Once the response time of each task is computed, the task set is feasible if

$$\forall i = 1, \ldots, n \quad R_i \leq D_i. \tag{8.16}$$

The feasibility analysis under preemption thresholds can also be simplified under the conditions of Theorem 8.1. In this case, we have that the worst-case start time is

$$S_i \;=\; B_i \;+ \sum_{h:P_h > P_i} \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) C_h \tag{8.17}$$

and the worst-case response time of task τ_i can be computed as

$$R_i \;=\; S_i \;+\; C_i \;+ \sum_{h:P_h > \theta_i} \left(\left\lceil \frac{R_i}{T_h} \right\rceil - \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) \right) C_h. \tag{8.18}$$

8.3.2 SELECTING PREEMPTION THRESHOLDS

The example illustrated in Figure 8.8 shows that a task set unfeasible under both preemptive and non-preemptive scheduling can be feasible under preemption thresholds, for a suitable setting of threshold levels. The algorithm presented in Figure 8.9 was proposed by Wang and Saksena [WS99] and allows assigning a set of thresholds to achieve a feasible schedule, if one exists. Threshold assignment is started from the lowest priority task to the highest priority one, since the schedulability analysis only depends on the thresholds of tasks with lower priority than the current task. While searching the optimal preemption threshold for a specific task, the algorithm stops at the minimum preemption threshold that makes it schedulable. The algorithm assumes that tasks are ordered by decreasing priorities, τ_1 being the highest priority task.

Algorithm: Assign Minimum Preemption Thresholds
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: Task set feasibility and $\theta_i, \forall \tau_i \in \mathcal{T}$
// Assumes tasks are ordered by decreasing priorities

```
(1)  begin
(2)      for (i := n to 1) do        // from the lowest priority task
(3)          θ_i := P_i;
(4)          Compute R_i by Equation (8.15);
(5)          while (R_i > D_i) do          // while not schedulable
(6)              θ_i := θ_i + 1;           // increase threshold
(7)              if (θ_i > P_1) then       // system infeasible
(8)                  return (INFEASIBLE);
(9)              end
(10)             Compute R_i by Equation (8.15);
(11)         end
(12)     end
(13)     return (FEASIBLE);
(14) end
```

Figure 8.9 Algorithm for assigning the minimum preemption thresholds.

Note that the algorithm is optimal in the sense that if a preemption threshold assignment exists that can make the system schedulable, the algorithm will always find an assignment that ensures schedulability.

Given a task set that is feasible under preemptive scheduling, another interesting problem is to determine the thresholds that limit preemption as much as possible, without jeopardizing the schedulability of the task set. The algorithm shown in Figure 8.10, proposed by Saksena and Wang [SW00], tries to increase the threshold of each task up to the level after which the schedule would become infeasible. The algorithm considers one task at the time, starting from the highest priority task.

Algorithm: Assign Maximum Preemption Thresholds
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: Thresholds $\theta_i, \forall \tau_i \in \mathcal{T}$
// Assumes that the task set is preemptively feasible

```
(1)  begin
(2)      for (i := 1 to n) do
(3)          θ_i = P_i;
(4)          k = i;                    // priority level k
(5)          schedulable := TRUE;
(6)          while ((schedulable := TRUE) and (k > 1)) do
(7)              k = k − 1;            // go to the higher priority level
(8)              θ_i = P_k;            // set threshold at that level
(9)              Compute R_k by Equation (8.15);
(10)             if (R_k > D_k) then   // system not schedulable
(11)                 schedulable := FALSE;
(12)                 θ_i = P_{k+1};    // assign the previous priority level
(13)             end
(14)         end
(15)     end
(16) end
```

Figure 8.10 Algorithm for assigning the maximum preemption thresholds.

8.4 DEFERRED PREEMPTIONS

According to this method, each task τ_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); whereas, under the activation-triggered model, preemption is enabled by a timer interrupt after exactly q_i units (unless the task completes earlier).

Since, in both cases, the start times of non-preemptive intervals are assumed to be unknown a priori, non-preemptive regions cannot be identified off-line, and for the sake of the analysis, they are considered to occur at the worst possible time (in the sense of schedulability).

For example, considering the same task set of Table 8.1, assigning $q_2 = 2$ and $q_3 = 1$, the schedule produced by Deadline Monotonic with deferred preemptions is feasible, as illustrated in Figure 8.11. Dark regions represent intervals executed in non-preemptive mode, triggered by the arrival of higher priority tasks.

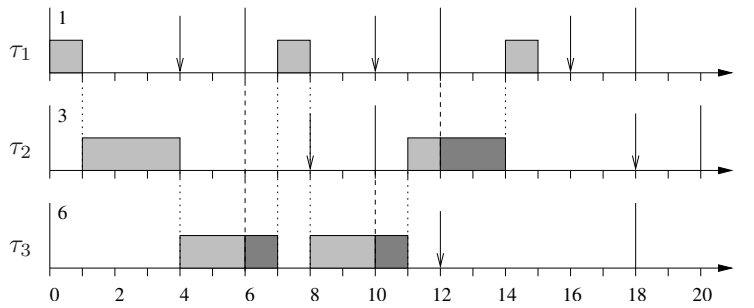


Figure 8.11 Schedule produced by Deadline Monotonic with deferred preemptions for the task set reported in Table 8.1, with $q_2 = 2$ and $q_3 = 1$.

8.4.1 FEASIBILITY ANALYSIS

In the presence of non-preemptive intervals, a task can be blocked when, at its arrival, a lower priority task is running in non-preemptive mode. Since each task can be blocked at most once by a single lower priority task, B_i is equal to the longest non-preemptive interval belonging to tasks with lower priority. In particular, the blocking factor can be computed as

$$B_i = \max_{j:P_j<P_i} \{q_j - 1\}. \tag{8.19}$$

Note that under the floating model one unit of time must be subtracted from q_j to allow the non-preemptive region to start before τ_i. Under the activation-triggered model, however, there is no need to subtract one unit of time from q_j, since the non-preemptive interval is programmed to be exactly q_j from the task arrival time. Then schedulability can be checked through the response time analysis, by Equation (7.22), or through the workload analysis, by Equation (7.23). Note that under the floating model the analysis does not need to be carried out within the longest Level-i active period. In fact, the worst-case interference on τ_i occurs in the first instance assuming that τ_i can be preempted an epsilon before its completion.

On the other hand, the analysis is more pessimistic under the activation-triggered model, where non-preemptive intervals are exactly equal to q_i units and can last until the end of the task. In this case, the analysis does not take advantage of the fact that τ_i cannot be preempted when higher periodic tasks arrive q_i units (or less) before its completion. The advantage of such a pessimism, however, is that the analysis can be limited to the first job of each task.

8.4.2 LONGEST NON-PREEMPTIVE INTERVAL

When using the deferred preemption method, an interesting problem is to find the longest non-preemptive interval Q_i for each task τ_i that can still preserve the task set schedulability. More precisely, the problem can be stated as follows:

> Given a set of n periodic tasks that is feasible under preemptive scheduling, find the longest non-preemptive interval of length Q_i for each task τ_i, so that τ_i can continue to execute for Q_i units of time in non-preemptive mode, without violating the schedulability of the original task set.

This problem has been first solved under EDF by Baruah [Bar05], and then under fixed priorities by Yao et al. [YBB09]. The solution is based on the concept of *blocking tolerance* β_i, for a task τ_i, defined as follows:

Definition 8.3 *The blocking tolerance β_i of a task τ_i is the maximum amount of blocking τ_i can tolerate without missing any of its deadlines.*

A simple way to compute the blocking tolerance is from the Liu and Layland test, which according to Equation (7.19) becomes:

$$\forall i = 1, \ldots, n \quad \sum_{h:P_h \geq P_i} \frac{C_h}{T_h} + \frac{B_i}{T_i} \leq U_{lub}(i)$$

where $U_{lub}(i) = i(2^{1/i} - 1)$ under RM, and $U_{lub}(i) = 1$ under EDF. Isolating the blocking factor, the test can also be rewritten as follows:

$$B_i \leq T_i \left(U_{lub}(i) - \sum_{h:P_h \geq P_i} \frac{C_h}{T_h} \right).$$

Hence, considering integer computations, we have:

$$\beta_i = \left\lfloor T_i \left(U_{lub}(i) - \sum_{h:P_h \geq P_i} \frac{C_h}{T_h} \right) \right\rfloor. \tag{8.20}$$

A more precise bound for β_i can be achieved by using the schedulability test expressed by Equation (7.23), which leads to the following result:

$$\exists t \in \mathcal{TS}_i : B_i \leq \{t - W_i(t)\}.$$

$$B_i \leq \max_{t \in \mathcal{TS}_i} \{t - W_i(t)\}.$$

$$\beta_i = \max_{t \in \mathcal{TS}_i} \{t - W_i(t)\}. \tag{8.21}$$

where set $\mathcal{TS}_i$ has been defined by equation (4.21).

Given the blocking tolerance, the feasible test can also be expressed as follows:

$$\forall i = 1, \ldots, n \quad B_i \leq \beta_i$$

and by Equation (8.19), we can write:

$$\forall i = 1, \ldots, n \quad \max_{j:P_j<P_i} \{q_j - 1\} \leq \beta_i.$$

This implies that to achieve feasibility we must have

$$\forall i = 1, \ldots, n \quad q_i \leq \min_{k:P_k>P_i} \{\beta_k + 1\}$$

Hence, the longest non-preemptive interval Q_i that preserves feasibility for each task τ_i is

$$Q_i = \min_{k:P_k>P_i} \{\beta_k + 1\}. \tag{8.22}$$

The Q_i terms can also be computed more efficiently, starting from the highest priority task (τ_1) and proceeding with decreasing priority order, according to the following theorem:

Theorem 8.2 *The longest non-preemptive interval Q_i of task τ_i that preserves feasibility can be computed as*

$$Q_i = \min\{Q_{i-1}, \beta_{i-1} + 1\} \tag{8.23}$$

where $Q_1 = \infty$ and $\beta_1 = D_1 - C_1$.

Proof. The theorem can be proved by noting that

$$\min_{k:P_k>P_i} \{\beta_k + 1\} = \min\{\min_{k:P_k>P_{i-1}} \{\beta_k + 1\}, \beta_{i-1} + 1\},$$

and since from Equation (8.22)

$$Q_{i-1} = \min_{k:P_k>P_{i-1}} \{\beta_k + 1\}$$

we have that

$$Q_i = \min\{Q_{i-1}, \beta_{i-1} + 1\},$$

which proves the theorem. □

Note that in order to apply Theorem 8.2, Q_i is not constrained to be less than or equal to C_i, but a value of Q_i greater than C_i means that τ_i can be fully executed in non-preemptive mode. The algorithm for computing the longest non-preemptive intervals is illustrated in Figure 8.12.

Algorithm: Compute the Longest Non-Preemptive Intervals
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: $Q_i, \forall \tau_i \in \mathcal{T}$
// Assumes $\mathcal{T}$ is preemptively feasible and $D_i \leq T_i$

```
(1) begin
(2)     β1 = D1 − C1;
(3)     Q1 = ∞;
(4)     for (i := 2 to n) do
(5)         Qi = min{Qi−1, βi−1 + 1};
(6)         Compute βi using Equation (8.20) or (8.21);
(7)     end
(8) end
```

Figure 8.12 Algorithm for computing the longest non-preemptive intervals.

8.5 TASK SPLITTING

According to this model, each task τ_i is split into m_i non-preemptive chunks (subjobs), obtained by inserting $m_i - 1$ preemption points in the code. Thus, 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:

$$\begin{cases} q_i^{max} = \max\limits_{k \in [1, m_i]} \{q_{i,k}\} \\ q_i^{last} = q_{i,m_i} \end{cases} \tag{8.24}$$

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

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

For example, consider the same task set of Table 8.1, and suppose that τ_2 is split in two subjobs of 2 and 1 unit, and τ_3 is split in two subjobs of 4 and 2 units. The schedule produced by Deadline Monotonic with such a splitting is feasible and it is illustrated in Figure 8.13.

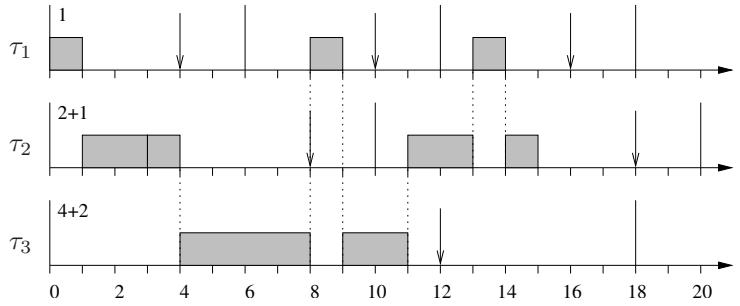


Figure 8.13 Schedule produced by Deadline Monotonic for the task set reported in Table 8.1, when τ_2 is split in two subjobs of 2 and 1 unit, and τ_3 is split in two subjobs of 4 and 2 units, respectively.

8.5.1 FEASIBILITY ANALYSIS

Feasibility analysis for task splitting can be carried out in a very similar way as the non-preemptive case, with the following differences:

- The blocking factor B_i to be considered for each task τ_i is equal to the length of longest subjob (instead of the longest task) among those with lower priority:

$$B_i = \max_{j:P_j<P_i} \{q_j^{max} - 1\}. \tag{8.25}$$

- The last non-preemptive chunk of τ_i is equal to q_i^{last} (instead of C_i).

The response time analysis for a task τ_i has to consider all the jobs within the longest Level-i Active Period, which can be computed using the following recurrent relation:

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \displaystyle\sum_{h:P_h \geq P_i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{8.26}$$

In particular, L_i is the smallest value for which $L_i^{(s)} = L_i^{(s-1)}$. This means that the response time of τ_i must be computed for all jobs $\tau_{i,k}$ with $k \in [1, K_i]$, where

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil . \tag{8.27}$$

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ of the last subjob can be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs, those subjobs preceding the last one ($C_i - q_i^{last}$), and the interference of the tasks with priority higher than P_i. Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} &= B_i + C_i - q_i^{last} + \sum\limits_{h:P_h>P_i} C_h \\ s_{i,k}^{(\ell)} &= B_i + kC_i - q_i^{last} + \sum\limits_{h:P_h>P_i} \left(\left\lfloor \dfrac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.28}$$

Since, once started, the last subjob cannot be preempted, the finishing time $f_{i,k}$ can be computed as

$$f_{i,k} = s_{i,k} + q_i^{last}. \tag{8.29}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1,K_i]} \{ f_{i,k} - (k-1)T_i \}. \tag{8.30}$$

Once the response time of each task is computed, the task set is feasible if

$$\forall i = 1, \ldots, n \quad R_i \leq D_i. \tag{8.31}$$

Assuming that the task set is preemptively feasible, the analysis can be simplified to the first job of each task, after the critical instant, as shown by Yao et al. [YBB10a]. Hence, the longest relative start time of τ_i can be computed as the smallest value satisfying the following recurrent relation:

$$\begin{cases} S_i^{(0)} &= B_i + C_i - q_i^{last} + \sum\limits_{h:P_h>P_i} C_h \\ S_i^{(\ell)} &= B_i + C_i - q_i^{last} + \sum\limits_{h:P_h>P_i} \left(\left\lfloor \dfrac{S_i^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.32}$$

Then, the response time R_i is simply:

$$R_i = S_i + q_i^{last}. \tag{8.33}$$

8.5.2 LONGEST NON-PREEMPTIVE INTERVAL

As done in Section 8.4.2 under deferred preemptions, it is interesting to compute, also under task splitting, the longest non-preemptive interval Q_i for each task τ_i that can still preserve the schedulability. It is worth observing that splitting tasks into subjobs allows achieving a larger Q_i, because a task τ_i cannot be preempted during the execution of the last q_i^{last} units of time.

If tasks are assumed to be preemptively feasible, for Theorem 8.1 the analysis can be limited to the first job of each task. In this case, a bound on the blocking tolerance β_i can be achieved using the following schedulability condition [YBB10a]:

$$\exists t \in \mathcal{TS}_i^* : B_i \leq \{t - W_i^*(t)\}, \tag{8.34}$$

where $W_i^*(t)$ and the testing set $\mathcal{TS}_i^*$ are defined as

$$W_i^*(t) = C_i - q_i^{last} + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{t}{T_h} \right\rfloor + 1 \right) C_h, \tag{8.35}$$

$$\mathcal{TS}_i^* \stackrel{\text{def}}{=} \mathcal{P}_{i-1}(D_i - q_i^{last}) \tag{8.36}$$

where $\mathcal{P}_i(t)$ is given by Equation (4.22).

Rephrasing Equation (8.34), we obtain

$$B_i \leq \max_{t \in \mathcal{TS}^*(\tau_i)} \{t - W_i^*(t)\}.$$

$$\beta_i = \max_{t \in \mathcal{TS}^*(\tau_i)} \{t - W_i^*(t)\}. \tag{8.37}$$

The longest non-preemptive interval Q_i that preserves feasibility for each task τ_i can then be computed by Theorem 8.2, using the blocking tolerances given by Equation (8.37). Applying the same substitutions, the algorithm in Figure 8.12 can also be used under task splitting.

As previously mentioned, the maximum length of the non-preemptive chunk under task splitting is larger than in the case of deferred preemptions. It is worth pointing out that the value of Q_i for task τ_i only depends on the β_k of the higher priority tasks, as expressed in Equation (8.22), and the blocking tolerance β_i is a function of q_i^{last}, as clear form equations (8.35) and (8.37).

8.6 SELECTING PREEMPTION POINTS

When a task set is not schedulable in non-preemptive mode, there are several ways to split tasks into subtasks to generate a feasible schedule, if one exists. Moreover, as already observed in Section 8.1, the runtime overhead introduced by the preemption mechanism depends on the specific point where the preemption takes place. Hence, it would be useful to identify the best locations for placing preemption points inside each task to achieve a feasible schedule, while minimizing the overall preemption cost. This section illustrates an algorithm [BXM+11] that achieves this goal.

Considering that sections of code exist where preemption is not desirable (e.g., short loops, critical sections, I/O operations, etc.), each job of τ_i is assumed to consist of a sequence of N_i non-preemptive Basic Blocks (BBs), identified by the programmer based on the task structure. Preemption is allowed only at basic block boundaries; thus each task has $N_i - 1$ *Potential Preemption Points* (PPPs), one between any two consecutive BBs. Critical sections and conditional branches are assumed to be executed entirely within a basic block. In this way, there is no need for using shared resource protocols to access critical sections.

The goal of the algorithm is to identify a subset of PPPs that minimizes the overall preemption cost, while achieving the schedulability of the task set. A PPP selected by the algorithm is referred to as an *Effective Preemption Point* (EPP), whereas the other PPPs are disabled. Therefore, the sequence of basic blocks between any two consecutive EPPs forms a Non-Preemptive Region (NPR). The following notation is used to describe the algorithm:

N_i denotes the number of BBs of task τ_i, determined by the $N_i - 1$ PPPs defined by the programmer;

p_i denotes the number of NPRs of task τ_i, determined by the $p_i - 1$ EPPs selected by the algorithm;

$\delta_{i,k}$ denotes the k-th basic block of task τ_i;

$b_{i,k}$ denotes the WCET of $\delta_{i,k}$ without preemption cost; that is, when τ_i is executed non-preemptively;

$\xi_{i,k}$ denotes the worst-case preemption overhead introduced when τ_i is preempted at the k-th PPP (i.e., between δ_k and δ_{k+1});

$q_{i,j}$ denotes the WCET of the j-th NPR of τ_i, including the preemption cost;

$q_i^{\max}$ denotes the maximum NPR length for τ_i:

$$q_i^{\max} = \max\{q_{i,j}\}_{j=1}^{p_i}.$$

To simplify the notation, the task index is omitted from task parameters whenever the association with the related task is evident from the context. In the following, we implicitly refer to a generic task τ_i, with maximum allowed NPR length $Q_i = Q$. As shown in the previous sections, Q can be computed by the algorithm in Figure 8.12. We say that an EPP selection is *feasible* if the length of each resulting NPR, including the initial preemption overhead, does not exceed Q.

Figure 8.14 illustrates some of the defined parameters for a task with 6 basic blocks and 3 NPRs. PPPs are represented by dots between consecutive basic blocks: black dots are EPPs selected by the algorithm, while white dots are PPPs that are disabled. Above the task code, the figure also reports the preemption costs ξ_k for each PPP, although only the cost for the EPPs is accounted in the q_j of the corresponding NPR.

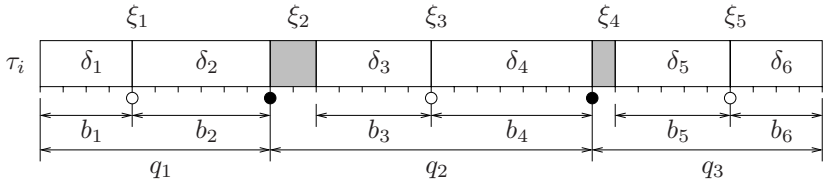


Figure 8.14 Example of task with 6 BBs split into 3 NPRs. Preemption cost is reported for each PPPs, but accounted only for the EPPs.

Using the notation introduced above, the non-preemptive WCET of τ_i can be expressed as follows:

$$C_i^{\text{NP}} = \sum_{k=1}^{N_i} b_{i,k}.$$

The goal of the algorithm is to minimize the overall worst-case execution time C_i of each task τ_i, including the preemption overhead, by properly selecting the EPPs among all the PPPs specified in the code by the programmer, without compromising the schedulability of the task set. To compute the preemption overhead, we assume that each EPP leads to a preemption, and that the cache is invalidated after each context switch (note that EPP selection is optimal only under these assumptions). Therefore,

$$C_i = C_i^{\text{NP}} + \sum_{k=1}^{N_i-1} \mathsf{selected}(i,k) \cdot \xi_{i,k}$$

where $\mathsf{selected}(i,k) = 1$ if the k-th PPP of τ_i is selected by the algorithm to be an EPP, whereas $\mathsf{selected}(i,k) = 0$, otherwise.

8.6.1 SELECTION ALGORITHM

First of all, it is worth noting that minimizing the number of EPPs does not necessarily minimize the overall preemption overhead. In fact, there are cases in which inserting more preemption points, than the minimum number, could be more convenient to take advantage of points with smaller cost.

Consider, for instance, the task illustrated in Figure 8.15, consisting of 6 basic blocks, whose total execution time in non preemptive mode is equal to $C_i^{NP} = 20$. The numbers above each PPP in Figure 8.15(a) denote the preemption cost; that is, the overhead that would be added to C_i^{NP} if a preemption occurred in that location. Assuming a maximum non-preemptive interval $Q = 12$, a feasible schedule could be achieved by inserting a single preemption point at the end of δ_4, as shown in Figure 8.15(b). In fact, $\sum_{k=1}^{4} b_k = 3+3+3+2 = 11 \leq Q$, and $\xi_4 + \sum_{k=5}^{6} b_k = 3+3+6 = 12 \leq Q$, leading to a feasible schedule. This solution is characterized by a total preemption overhead of 3 units of time (shown by the gray execution area). However, selecting two EPPs, one after δ_1 and another after δ_5, a feasible solution is achieved with a smaller total overhead $\xi_1 + \xi_5 = 1 + 1 = 2$, as shown in Figure 8.15(c). In general, for tasks with a large number of basic blocks with different preemption cost, finding the optimal solution is not trivial.

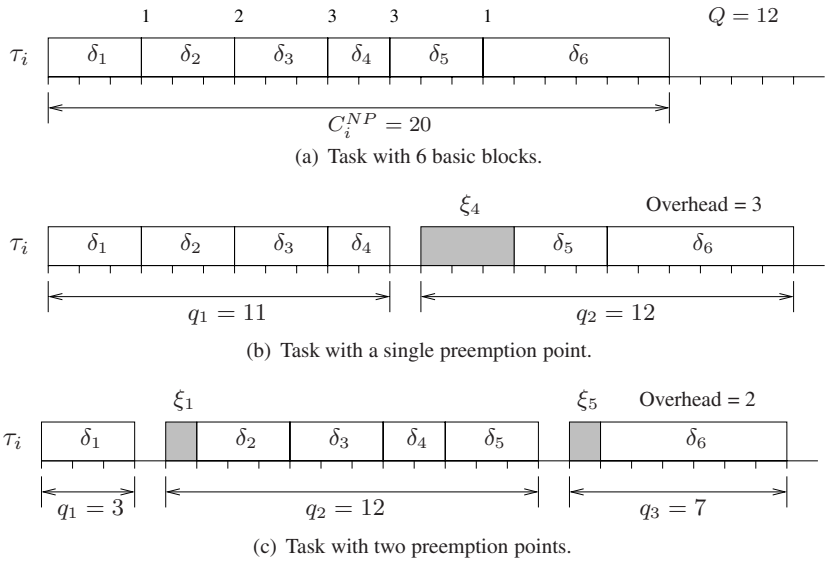


Figure 8.15 Two solutions for selecting EPPs in a task with $Q = 12$: the first minimizes the number of EPPs, while the second minimizes the overall preemption cost.

For a generic task, the worst-case execution time q of a NPR composed of the consecutive basic blocks $\delta_j, \delta_{j+1}, \ldots, \delta_k$ can be expressed as

$$q = \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell, \tag{8.38}$$

conventionally setting $\xi_0 = 0$. Note that the preemption overhead is included in q. Since any NPR of a feasible EPP selection has to meet the condition $q \leq Q$, we must have

$$\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \leq Q. \tag{8.39}$$

Now, let $\hat{C}_k$ be the WCET of the chunk of code composed of the first k basic blocks – that is, from the beginning of δ_1 until the end of δ_k – including the preemption overhead of the EPPs that are contained in the considered chunk. Then, we can express the following recursive expression:

$$\hat{C}_k = \hat{C}_{j-1} + q = \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell. \tag{8.40}$$

Note that since δ_N is the last BB, the worst-case execution time C_i of the whole task τ_i is equal to $\hat{C}_N$.

The algorithm for the optimal selection of preemption points is based on the equations presented above and its pseudo-code is reported in Figure 8.16. The algorithm evaluates all the BBs in increasing order, starting from the first one. For each BB δ_k, the feasible EPP selection that leads to the smallest possible $\hat{C}_k$ is computed as follows.

For the first BBs, the minimum $\hat{C}_k$ is given by the sum of the BB lengths $\sum_{\ell=1}^{k} b_\ell$ as long as this sum does not exceed Q. Note that if $b_1 > Q$, there is no feasible PPP selection, and the algorithm fails. For the following BBs, $\hat{C}_k$ needs to consider the cost of one or more preemptions as well. Let $Prev_k$ be the set of the preceding BBs $\delta_{j \leq k}$ that satisfy Condition (8.39), i.e., that might belong to the same NPR of δ_k. Again, if $\xi_{k-1} + b_k > Q$, there is no feasible PPP activation, and the algorithm fails. Otherwise, the minimum $\hat{C}_k$ is given by

$$\hat{C}_k = \min_{\delta_j \in Prev_k} \left\{ \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \tag{8.41}$$

Let $\delta^*(\delta_k)$ be the basic block for which the rightmost term of Expression (8.41) is minimum

$$\delta^*(\delta_k) = \min_{\delta_j \in Prev_k} \left\{ \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \tag{8.42}$$

If there are many possible BBs minimizing (8.41), the one with the smallest index is selected. Let $\delta_{Prev}(\delta_k)$ be the basic block preceding $\delta^*(\delta_k)$, if one exists. The PPP at the end of $\delta_{Prev}(\delta_k)$ – or, equivalently, at the beginning of $\delta^*(\delta_k)$ – is meaningful for the analysis, since it represents the last PPP to activate for minimizing the preemption overhead of the first k basic blocks.

A feasible placement of EPPs for the whole task can then be derived with a recursive activation of PPPs, starting with the PPP at the end of $\delta_{Prev}(\delta_N)$, which will be the last EPP of the considered task. The penultimate EPP will be the one at the beginning of $\delta_{Prev}(\delta_{Prev}(\delta_N))$, and so on. If the result of this recursive lookup of function $\delta_{Prev}(k)$ is δ_1, the start of the program has been reached. A feasible placement of EPPs has therefore been detected, with a worst-case execution time, including preemption overhead, equal to $\hat{C}_N$. This is guaranteed to be the placement that minimizes the preemption overhead of the considered task, as proved in the next theorem.

Theorem 8.3 (Bertogna et al., 2011) *The PPP activation pattern detected by procedure **SelectEPP**(τ_i, Q_i) minimizes the preemption overhead experienced by a task τ_i, without compromising the schedulability.*

The feasibility of a given task set is maximized by applying the **SelectEPP**(τ_i, Q_i) procedure to each task τ_i, starting from τ_1 and proceeding in task order. Once the optimal allocation of EPPs has been computed for a task τ_i, the value of the overall WCET $C_i = \hat{C}_N$ can be used for the computation of the maximum allowed NPR Q_{i+1} of the next task τ_{i+1}, using the technique presented in Section 8.5.

The procedure is repeated until a feasible PPP activation pattern has been produced for all tasks in the considered set. If the computed Q_{i+1} is too small to find a feasible EPP allocation, the only possibility to reach schedulability is by removing tasks from the system, as no other EPP allocation strategy would produce a feasible schedule.

```
Algorithm: SelectEPP(τ_i, Q_i)
Input: {C_i, T_i, D_i, Q_i} for task τ_i
Output: The set of EPPs for task τ_i
(1)  begin
(2)      Prev_k := {δ_0}; Ĉ_0 := 0;      // Initialize variables
(3)      for (k := 1 to N) do            // For all PPPs
(4)          Remove from Prev_k all δ_j violating (8.39);
(5)          if (Prev_k = ∅) then
(6)              return (INFEASIBLE);
(7)          end
(8)          Compute Ĉ_k using Equation (8.41);
(9)          Store δ_Prev(δ_k);
(10)         Prev_k := Prev_k ∪ {δ_k};
(11)     end
(12)     δ_j := δ_Prev(δ_N);
(13)     while (δ_j ≠ ∅) do
(14)         Select the PPP at the end of δ_Prev(δ_j);
(15)         δ_j ← δ_Prev(δ_j);
(16)     end
(17)     return (FEASIBLE);
(18) end
```

Figure 8.16 Algorithm for selecting the optimal preemption points.

8.7 ASSESSMENT OF THE APPROACHES

The limited preemption methods presented in this chapter can be compared under several aspects, such as the following

- Implementation complexity.
- Predictability in estimating the preemption cost.
- Effectiveness in reducing the number of preemptions.

8.7.1 IMPLEMENTATION ISSUES

The preemption threshold mechanism can be implemented by raising the execution priority of the task, as soon as it starts running. The mechanism can be easily implemented at the application level by calling, at the beginning of the task, a system call that increases the priority of the task at its threshold level. The mechanism can also be fully implemented at the operating system level, without modifying the application tasks. To do that, the kernel has to increase the priority of a task at the level of its threshold when the task is scheduled for the first time. In this way, at its first activation, a task is inserted in the ready queue using its nominal priority. Then, when the task is scheduled for execution, its priority becomes equal to its threshold, until completion. Note that if a task is preempted, its priority remains at its threshold level.

In deferred preemption (floating model), non-preemptive regions can be implemented by proper kernel primitives that disable and enable preemption at the beginning and at the end of the region, respectively. As an alternative, preemption can be disabled by increasing the priority of the task at its maximum value, and can be enabled by restoring the nominal task priority. In the activation-triggered mode, non-preemptive regions can be realized by setting a timer to enforce the maximum interval in which preemption is disabled. Initially, all tasks can start executing in non-preemptive mode. When τ_i is running and a task with priority higher than P_i is activated, a timer is set by the kernel (inside the activation primitive) to interrupt τ_i after q_i units of time. Until then, τ_i continues executing in non-preemptive mode. The interrupt handler associated to the timer must then call the scheduler to allow the higher priority task to preempt τ_i. Note that once a timer has been set other additional activations before the timeout will not prolong the timeout any further.

Finally, cooperative scheduling does not need special kernel support, but it requires the programmer to insert in each preemption point a primitive that calls the scheduler, so enabling pending high-priority tasks to preempt the running task.

8.7.2 PREDICTABILITY

As observed in Section 8.1, the runtime overhead introduced by the preemption mechanism depends on the specific point where the preemption takes place. Therefore, a method that allows predicting where a task is going to be preempted simplifies the estimation of preemption costs, permitting a tighter estimation of task WCETs.

Unfortunately, under preemption thresholds, the specific preemption points depend on the actual execution of the running task and on the arrival time of high priority tasks;

hence, it is practically impossible to predict the exact location where a task is going to be preempted.

Under deferred preemptions (floating model), the position of non-preemptive regions is not specified in the model, thus they are considered to be unknown. In the activation-triggered model, instead, the time at which the running task will be preempted is set q_i units of time after the arrival time of a higher priority task. Hence, the preemption position depends on the actual execution of the running task and on the arrival time of the higher priority task. Therefore, it can hardly be predicted off-line.

On the contrary, under cooperative scheduling, the locations where preemptions may occur are explicitly defined by the programmer at design time; hence, the corresponding overhead can be estimated more precisely by timing analysis tools. Moreover, through the algorithm presented in Section 8.6.1, it is also possible to select the best locations for placing the preemption points to minimize the overall preemption cost.

8.7.3 EFFECTIVENESS

Each of the presented methods can be used to limit preemption as much as desired, but the number of preemptions each task can experience depends of different parameters.

Under preemption thresholds, a task τ_i can only be preempted by tasks with priority greater than its threshold θ_i. Hence, if preemption cost is neglected, an upper bound ν_i on the number of preemptions that τ_i can experience can be computed by counting the number of activations of tasks with priority higher than θ_i occurring in $[0, R_i]$; that is

$$\nu_i = \sum_{h:P_h>\theta_i} \left\lceil \frac{R_i}{T_h} \right\rceil .$$

This is an upper bound because simultaneous activations are counted as if they were different, although they cause a single preemption.

Under deferred preemption, the number of preemptions occurring on τ_i is easier to determine, because it directly depends on the non-preemptive interval q_i specified for the task. If preemption cost is neglected, we simply have

$$\nu_i = \left\lceil \frac{C_i^{NP}}{q_i} \right\rceil - 1.$$

However, if preemption cost is not negligible, the estimation requires an iterative approach, since the task computation time also depends on the number of preemptions.

Considering a fixed cost ξ_i for each preemption, then the number of preemptions can be upper bounded using the following recurrent relation:

$$\begin{cases} \nu_i^0 = \left\lceil \frac{C_i^{NP}}{q_i} \right\rceil - 1 \\ \nu_i^s = \left\lceil \frac{C_i^{NP} + \xi_i \nu_i^{s-1}}{q_i} \right\rceil - 1 \end{cases}$$

where the iteration process converges when $\nu_i^s = \nu_i^{s-1}$.

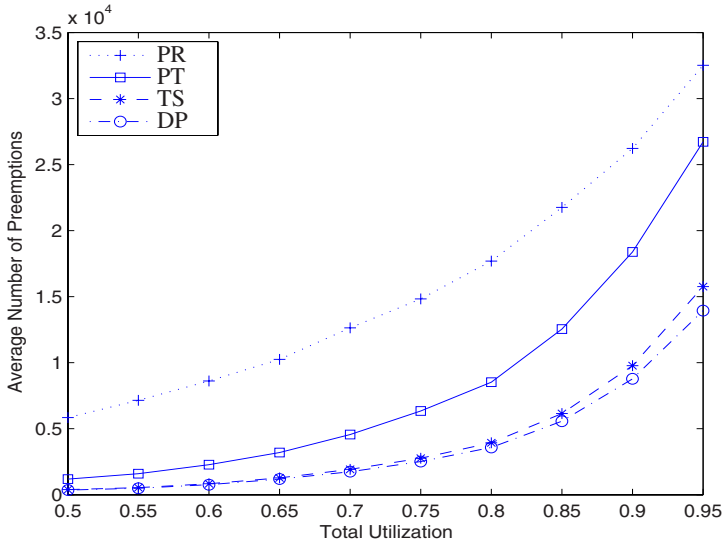
Finally, under cooperative scheduling, the number of preemptions can be simply upper bounded by the number of effective preemption points inserted in the task code.

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. Figures 8.17(a) and 8.17(a) show 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.

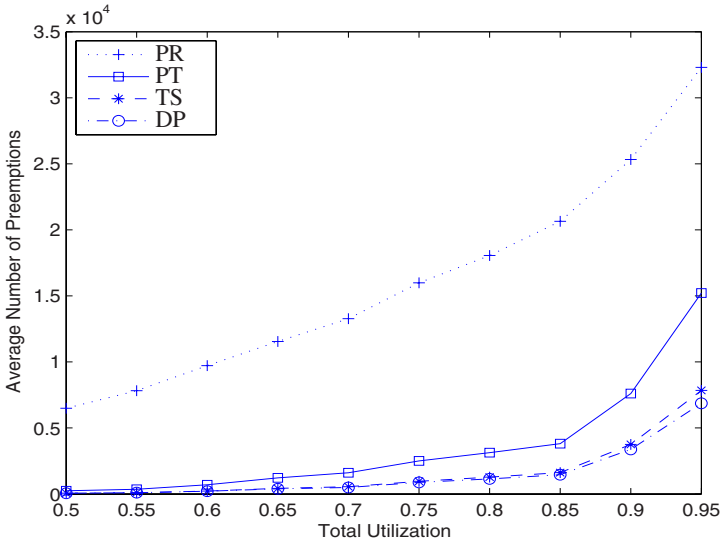
Each simulation run was performed on a set of n tasks with total utilization U varying from 0.5 to 0.95 with step 0.05. Individual utilizations U_i were uniformly distributed in [0,1], using the UUniFast algorithm [BB05]. Each computation time C_i was generated as a random integer uniformly distributed in [10, 50], and then T_i was computed as $T_i = C_i/U_i$. The relative deadline D_i was generated as a random integer in the range $[C_i + 0.8 \cdot (T_i - C_i),\ T_i]$. The total simulation time was set to 1 million units of time. For each point in the graph, the result was computed by taking the average over 1000 runs.

All the task sets have been generated to be preemptive feasible. Under preemption thresholds (PT), the algorithm proposed by Saksena and Wang [SW00] was used to find the maximum priority threshold that minimizes the number of preemptions. Under deferred preemptions (DP) and task splitting (TS), the longest non-preemptive regions were computed according to the methods presented in Sections 8.4.2 and 8.5.2, respectively. Finally, under task splitting, preemption points were inserted from the end of task code to the beginning.

As expected, fully preemptive scheduling (PR) generates the largest number of preemptions, while DP and TS are both able to achieve a higher reduction. PT has an intermediate behavior. Note that DP can reduce slightly more preemptions than TS, since, on the average, each preemption is deferred for a longer interval (always equal to Q_i, except when the preemption occurs near the end of the task).



(a) Number of tasks: n=6



(b) Number of tasks: n=12

Figure 8.17 Average number of preemptions with different number of tasks.

However, it is important to consider that TS can achieve a lower and more predictable preemption cost, since preemption points can be suitably decided off-line with this purpose. As shown in the figures, PR produces a similar number of preemptions when the number of tasks increases, whereas all the other methods reduce the number of preemptions to an even higher degree. This is because, when n is larger, tasks have smaller individual utilization, and thus can tolerate more blocking from lower priority tasks.

8.7.4 CONCLUSIONS

The results reported in this chapter can be summarized in Table 8.3, which compares the three presented methods in terms of the metrics presented above. As discussed in the previous section, the preemption threshold mechanism can reduce the overall number of preemptions with a low runtime overhead; however, preemption cost cannot be easily estimated, since the position of each preemption, as well as the overall number of preemptions for each task, cannot be determined off-line. Using deferred preemptions, the number of preemptions for each task can be better estimated, but the position of each preemption still cannot be determined off-line. Cooperative scheduling is the most predictable mechanism for estimating preemption costs, since both the number of preemptions and their positions are fixed and known from the task code. Its implementation, however, requires inserting explicit system calls in the source code that introduce additional overhead.

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

Table 8.3 Evaluation of limited preemption methods.

Exercises

8.1 Given the task set reported in the table, verify whether it is schedulable by the Rate-Monotonic algorithm in non-preemptive mode.

	C_i	T_i	D_i
τ_1	2	6	5
τ_2	2	8	6
τ_3	4	15	12

8.2 Given the task set reported in the table, verify whether it is schedulable by the Rate-Monotonic algorithm in non-preemptive mode.

	C_i	T_i	D_i
τ_1	3	8	6
τ_2	3	9	8
τ_3	3	14	12
τ_4	2	80	80

8.3 Given the task set reported in the table, compute for each task τ_i the longest (floating) non-preemptive region Q_i that guarantees the schedulability under EDF. Perform the computation using the Liu and Layland test.

	C_i	T_i
τ_1	2	8
τ_2	2	10
τ_3	5	30
τ_4	5	60
τ_5	3	90

8.4 For the same task set reported in Exercise 8.3, compute for each task τ_i the longest (floating) non-preemptive region Q_i that guarantees the schedulability under Rate Monotonic. Perform the computation using the Liu and Layland test.

8.5 Compute the worst case response times produced by Rate Monotonic for the sporadic tasks illustrated below, where areas in light grey represent non-preemptive regions of code, whereas regions in dark grey are fully preemptable. The number inside a region denotes the worst-case execution time (WCET) of that portion of code, whereas the number on the right represents the WCET of the entire task.

Task periods are $T_1 = 24$, $T_2 = 40$, $T_3 = 120$, and $T_4 = 150$. Relative deadlines are equal to periods.