Timing analysis of compound scheduling policies: application to Posix1003.1b

Jörn Migge[2], Alain Jean-Marie[1], Nicolas Navet[2]

[1] LIRMM
Département IFA
161 Rue Ada
F-34392 Montpellier Cedex 05

[2] LORIA - CNRS UMR 7503
TRIO Team - ENSEM
2, Avenue de la forêt de Haye
F-54516 Vandoeuvre-lès-Nancy

January 24, 2002

Abstract

The analysis of fixed priority preemptive scheduling has been extended in various ways to improve its usefulness for the design of real-time systems. In this paper, we define the *layered preemptive priority* scheduling policy which generalizes fixed preemptive priorities by combination with other policies in a layered structure. In particular, the combination with the Round Robin scheduling policy is studied. Its compliance with Posix 1003.1b requirements is shown and its timing analysis is provided. For this purpose and as a basis for the analysis of other policies, the concept of *majorizing work arrival function*, is introduced to synthesize essential ideas used in existing analysis of the fixed preemptive priority policy.
If critical resources are protected by semaphores, the Priority Ceiling Protocol (PCP) can be used under fixed preemptive priorities to control resulting priority inversions. An extension of the PCP is proposed for Round Robin, to allow a global control of priority inversions under the layered priority policy and to prevent deadlocks. The initial timing analysis is extended to account for the effects of the protocol. The results are illustrated by a small test case.

1 Introduction

A real-time system is typically composed of a controlling system and a controlled system. The timing constraints on the controlling system's activities arise from the dynamics of the controlled system. For the satisfaction of these constraints a correct schedule of the system's activities needs to be found.

In this paper we introduce the *layered preemptive priority* scheduling policy, LPP for short. It allows to combine different policies in a layered structure, based on the fixed preemptive priority scheduling policy, FPP for short. The techniques used in the timing analysis of FPP have evolved with the attempts at accounting more precisely for the characteristics of real-world systems. In this paper we propose a framework that synthesizes the essential ideas of these techniques. We apply the framework to the analysis of the LPP policy, of which the FPP policy is a special case.

The FPP policy was initially analyzed by Liu & Layland [1] for periodic tasks with deadlines equal to their period. It has been shown that the maximal response time of a task occurs after the *critical instant*, defined as a time where all tasks are released simultaneously. Only this response time needs to be analyzed to decide upon feasibility. A task set is said to be feasible if each task always meets its deadlines at run-time. A sufficient feasibility test has been derived from this property, based on a bound of the tasks's average processor utilization. A necessary and sufficient feasibility test has been derived by Lehoczky et al. [2], based on a test-function, that must be evaluated at certain times between the release and the deadline of a task. A different approach was introduced by Joseph & Pandya in [3]. Their idea is to compute the response time of a task after the critical instant and to compare it with the deadline. The actual computations however are quite similar in both cases.

In order to apply the test-function based approach in the case where deadlines are set beyond the task periods, Lehoczky introduced in [4] the concept of *busy period*, which was then also used for the analysis based on response time computation in [5] by Tindel et al. Basically, the busy period that starts with the critical instant has to be analyzed to determine the worst case response time. More generally, the critical instant can be seen as the beginning of a *worst case release pattern*.

In this paper we formalize these *worst case release patterns* by introducing the concept of *(majorizing) work arrival function*. The purpose is to provide a timing analysis framework that allows to easily integrate new types of tasks and policies with results already known. Work arrival functions (WAF, for short) allow in particular to write response time formulas and response time bounds for a given policy independently of a specific type of task used as model for a real-world task. As a result, timing analysis can be split into two parts, one concerned with the analysis of worst-case behaviors of tasks in terms of majorizing work arrival functions (MWAF, for short) and a second which uses MWAF and is only concerned with analysis of the specific properties of the scheduling policy. The analysis of a new policy can then rely on known MWAF, and the MWAF of a new task model can be integrated into the analysis of a known policy. This approach is especially useful in the analysis of more complex policies such as the layered preemptive priority scheduling policy, that we introduce in this paper. This scheduling policy uses the layered structure of fixed priorities to combine different policies. We shall restrict our study to the combination of FPP and the Round Robin policy (RR, for short), motivated by the Posix 1003.1b standard. According to this standard, (real-time) operating systems have to provide at least these two scheduling policies for a side-by-side use. With Round Robin, the question of efficiently handling semaphores has

to be reconsidered. Under preemptive policies, semaphores are used to prevent preemption during the use of critical resources. This leads to longer response times for higher priority tasks, because of *priority inversions*. To limit this effect and to avoid possible deadlocks with nested access to resources, the priority ceiling protocol has been proposed by Sha et al. in [6] for FPP. Applied "as is" to Round Robin, the Priority Ceiling Protocol (PCP) does not prevent deadlocks. Furthermore, semaphores distort the processor access rates which are usually guaranteed by the Round Robin scheduling policy. This is similar to the priority inversions under FPP. In this paper we define a first extension of the PCP that prevents from deadlocks and a second that controls the access rates. The first extension can be implemented under Posix. The second requires some modifications, but allows a more accurate timing analysis.

The paper is organized as follows. In Section 2, the Posix 1003.1b standard is briefly described and in Section 3 the layered preemptive priority policy is defined and its link with the Posix 1003.1b standard is explained. In Section 4 the concept of *majorizing work arrival function* is introduced to derive general response time bounds under FPP and also RR, in Section 5. The definition of the priority ceiling protocol for Round Robin and the corresponding timing analysis are given in Section 6. A numerical example is proposed in Section 7 and a conclusion is given in Section 8.

2 Posix 1003.1b scheduling

Posix 1003.1b standard [7], formerly Posix.4, defines real-time extensions to Posix.1 mainly concerning signals, inter-process communication, memory mapped files, memory locking, synchronized and asynchronous I/O, timers and scheduling policies. Most of today's real-time operating systems conform, at least partially, to this standard.
Posix 1003.1b specifies three scheduling policies : SCHED_RR, SCHED_FIFO and SCHED-OTHER. These policies apply on a process-by-process basis : each process runs with a particular policy and a given priority. Each process inherits its scheduling parameters from its father but may also change them at run-time.

- SCHED_FIFO : fixed preemptive priority with FIFO ordering among same priority processes.
- SCHED_RR : round-robin policy which allows processes of same priority to share the processing unit. Note that a process will not get the CPU until all higher priority ready-to-run processes are executed. The quantum value may be a system-wide constant, process specific or fixed for a given priority level.
- SCHED_OTHER is an implementation-defined scheduler. It could map onto SCHED_RR or SCHED_FIFO or also implement a classical Unix time-sharing policy. The standard merely mandates its presence and its documentation. Because it is not possible to expect the same behavior of SCHED_OTHER under all Posix compliant operating systems, it is strongly suggested not to use it if portability is a matter of concern and we will not consider it in our analysis.

Associated with each policy is a priority range. Depending on the implementation, these priority ranges may or may not overlap. In the case where priority ranges overlap, tasks with policy flags SCHED_FIFO or SCHED_RR may have the same priority. Our analysis can

handle this case if these SCHED_FIFO flags are changed to SCHED_RR and the quantum is set to the tasks *worst case execution time* (WCET).

Note that these scheduling mechanisms similarly apply to Posix threads standardized by Posix 1003.1c standard. Threads, defined in [7] as single flows of control within a process, are a means of providing multiple "lightweight" processes within one process address space. The problem with threads is that some aspects of their scheduling are not covered by the standard. The concept of the scheduling contention scope of a thread defines the set of threads with which it has to compete for the use of processing resources :

- A thread created with the *system contention scope*, regardless of the process in which it resides, competes with all processes and all other threads which have the system scheduling contention scope. The analysis that is further developed equally applies to system-wide contention threads.

- A thread created with the *process contention scope* only competes with other threads within its process having the process scheduling contention scope. In the standard, it is unspecified how such "process contention scope" threads are scheduled relative to threads belonging to other processes and relative to other processes. Because the process contention scope scheduling is OS-dependent, we will not further consider its use.

In the rest of the paper, we define a task as a recurrent activity. A task may be implemented either by a process (resp. thread) which is repetitively launched or by a unique process (resp. thread) that runs in cycles.

3 Layered Priorities

We now define the *layered preemptive priority* scheduling policy for a set of recurrent tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$. A task is recurrent if at any time there is a next release of the task that will occur in the future. The simplest example are periodic tasks. Since for each release the task may have a different execution time and the time since the previous release may be different too, we wish to be able to consider each release individually. Therefore, we call a release an *instance* of the task; the nth instance of τ_k is denoted $\tau_{k,n}$.

The policy is defined in two steps. The first step specifies the layer structure and the second the internal structure of the layers.

Let the set of tasks be partitioned into layers $\lambda_l = \{\tau_k | m_{l-1} < k \leqslant m_l\}$, where $l = 1, 2, \ldots$ and $1 \leqslant m_{l-1} < m_l \leqslant m$, with $m_0 = 0$, see Table 1. Suppose they are scheduled according to the global rule: an instance $\tau_{k,n}$ of a task τ_k in a layer λ_l is executed as soon as and as long as no instance in the (higher priority) layers $\lambda_1, \ldots, \lambda_{l-1}$ is pending.

So far the scheduling rule is incomplete since it does not tell how tasks are scheduled inside of a layer. The fixed preemptive priority scheduling policy (FPP) can be realized inside of a layer λ_l by applying the same rule again, i.e. an instance of $\tau_k \in \lambda_l$ is executed as soon as and as long as no instance of a (higher priority) task τ_j, $m_{l-1} < j < k$ in the same layer λ_l is pending. Under POSIX this can be implemented by giving to each task of the layer a different priority with the SCHED_FIFO attribute in a reserved subrange of priorities. The Round Robin scheduling policy can be realized inside of a layer by assigning the same priority to all tasks of the layer with the SCHED_RR attribute and some time quantum to each task

Layer	m_l	Tasks	Policy-Flag	Priority	Quantum
λ_1 (FPP)		τ_1	SCHED_FIFO	**1**	-
		τ_2	SCHED_FIFO	**2**	-
		τ_3	SCHED_FIFO	**3**	-
	$m_1 = 4$	τ_4	SCHED_FIFO	**4**	-
λ_2 (RR)		τ_5	SCHED_RR	**5**	15
		τ_6	SCHED_RR	**5**	5
	$m_2 = 7$	τ_7	SCHED_RR	**5**	10
λ_3 (FPP)		τ_8	SCHED_FIFO	**6**	-
		τ_9	SCHED_FIFO	**7**	-
		τ_{10}	SCHED_FIFO	**8**	-
		τ_{11}	SCHED_FIFO	**9**	-
	$m_3 = 12$	τ_{12}	SCHED_FIFO	**10**	-

Table 1: Example of Layered Priorities (under Posix 1003.1b)

that tells how long it is allowed to execute without being interrupted by the RR-scheduler. The priority must of course be reserved for tasks of the RR-layer.

Notice that the policy in a layer could be any *non-idling* policy such as *earliest deadline first*, *last in first out*, etc. Following this idea it is possible to define LPP in a more general manner in terms of *priority functions*. The interested reader is referred to [8].

4 Fixed preemptive priorities

To be able to analyze a layered preemptive priority policy based on FPP and RR, we have to reformulate the existing FPP timing analysis in this section. Computation of response time bounds are always based on task models. At the beginning a simple model based on worst-case inter-release and execution times [3], [5] has been used. Some tasks however do not achieve the corresponding peak-rate for an arbitrarily long period of time and thus their worst-case demand is overestimated. The resulting response time bounds are rather pessimistic. With that kind of approach the worst-case behavior of a task is actually associated with that of a periodic task, even if its worst-case behavior is different. To improve the analysis, other task models have been defined, such as *sporadically periodic tasks* [5] and *multi-frame tasks* [9] (see Section 4.2. Response time bounds have been derived in [5] and a feasibility test in [9]. In this section, we present these known results in a general manner to give a synthetic view of the common underlying ideas and to extend the analysis to layered priorities. At the end, we provide a generic algorithm for computing response times, based on the general concept of *majorizing work arrival function*.

4.1 Interference periods and Response Times

In this section we present a response time formula for an instance of a task scheduled under FPP. The formula is based on *work arrival functions* and *interference periods*. It does not involve deadlines, and is therefore valid whatever their values are. For a detailed discussion the reader is referred to [8].

For the purpose of deriving a response time formula for an instance $\tau_{k,n}$ of a task τ_k we have to introduce some notations. Let $C_{k,n} \in \mathbb{R}_+$ and $A_{k,n} \in \mathbb{R}_+$ be the execution and release time of the nth instance of τ_k on a trajectory of the system. By definition, beyond its activation or *release* time, an instance is ready for execution, meaning that further delays are only due to the scheduler and no other reasons. The time between two releases is called cycle time and is denoted $T_{k,n}$ with the convention $A_{k,n+1} = A_{k,n} + T_{k,n}$. Notice that the cycle times of a task are not necessarily equal to a period, but may be different for each instance. We make however the assumption that $T_{k,n} > 0$, so that consecutive instances are never released simultaneously. The execution end of $\tau_{k,n}$ is denoted $E_{k,n}$ and its response time is $R_{k,n} = E_{k,n} - A_{k,n}$.

The execution end depends on several factors. First of all, the instance $\tau_{k,n}$ may start to execute only after the previous instances of τ_k have completed. Furthermore, while $\tau_{k,n}$ is executed it is preempted to satisfy incoming demands from the higher priority tasks $\tau_1, \ldots, \tau_{k-1}$. Notice that whether the tasks in higher priority layers are scheduled under Round Robin or FPP, all their demands must be satisfied first. This represents a total amount of work to be executed between $A_{k,n}$ and $E_{k,n}$, which can be subdivided into two parts.

One part consists in the remaining work of higher priority instances released *strictly* before $A_{k,n}$ and still pending at $A_{k,n}$. Let the remaining work of a task τ_i at some time t be denoted by $W_i(t)$. With $W_{1..i}(t) = \sum_{j=1}^{i} W_j(t)$, the pending higher priority work is $W_{1..k}(A_{k,n})$.

The second part is due to the releases of higher priority tasks during $[A_{k,n}, E_{k,n})$. For this kind of work we introduce *work arrival functions*, WAF for short. For a task τ_k, it is defined for every interval $[a, b)$ by

$$S_k(a, b) = \sum_{j \in \mathbb{N}} C_{k,j} \cdot \mathbb{I}_{[a \leqslant A_{k,j} < b]}. \tag{1}$$

The indicator function $\mathbb{I}_{[.]}$ takes the value 1 if the release time $A_{k,j}$ is situated in $[a, b)$, otherwise 0. Thus the WAF returns the sum of the execution times of all the task released in $[a, b)$. Its value is the sum of all demands from instances of τ_k released in the given interval $[a, b)$. Notice that WAF's are additive on adjacent intervals:

$$S_k(a, b) + S_k(b, c) = S_k(a, c). \tag{2}$$

With $S_{1..i}(a, b) = \sum_{k=1}^{i} S_k(a, b)$, the execution end of $\tau_{k,n}$ can now be written as

$$E_{k,n} = \min\{t > A_{k,n} \mid W_{1..k}(A_{k,n}) + C_{k,n} + S_{1..k-1}(A_{k,n}, t) = t - A_{k,n}\}.$$

It is the first time after $A_{k,n}$ where the sum of initially pending higher priority work $W_{1..k}(A_{k,n})$, the execution time $C_{k,n}$ of the instance and the arriving higher priority work $S_{1..k-1}(A_{k,n}, t)$ is equal to the available processing time $t - A_{k,n}$. Notice that we have defined WAF's so that $S_{1..k-1}(A_{k,n}, t)$ does not account for releases that may occur at t, because a higher priority instance arriving "at the end" of the interval $[A_{k,n}, t)$ inside of which $\tau_{k,n}$ is able to complete can not preempt $\tau_{k,n}$.

To be able to derive response time bounds from the execution end formula it is useful to express the pending work $W_{1..k}(A_{k,n})$ in terms of WAF. There is a time U^k before $A_{k,n}$, where pending work of the tasks $\tau_1, \ldots, \tau_k$ is zero:

Definition 1
A level-k interference period *is a time interval $[U^k, V^k)$ such that already activated instances of tasks with a priority higher or equal to k are pending neither at its beginning U^k nor at its end V^k, but at any other time between U^k and V^k there is at least one such instance pending.*

The tasks "with a priority higher than k" are all those from higher priority layers, i.e. $\tau_1, \ldots, \tau_{m_{l-1}}$, plus the higher priority tasks in the same layer, i.e. $\tau_{m_{l-1}+1}, \ldots, \tau_{k-1}$.

By "already activated" we mean that the activation time must be strictly before the considered time. Therefore, at any time $t \in [U^k, V^k)$, there is at least one pending instance, i.e. the demands are higher than the available processing time, and thus

$$V^k = \min\{t > U^k \mid S_{1..k}(U^k, t) = t - U^k\}. \tag{3}$$

We denote by $U_{k,n} = U^k$ the beginning and by $V_{k,n}$ the end of the level-k interference period in which $\tau_{k,n}$ is released, i.e. that contains $A_{k,n}$. Notice the following property:

$$A_{k,i} \in [U_{k,n}, V_{k,n}),\ \text{for } n \geqslant k+1 \qquad \Rightarrow \qquad A_{k,i} < E_{k,i-1}, \tag{4}$$

which means that inside a level-k interference period, all instances of τ_k, except the first one, are activated strictly before the previous instances has completed. The strict inequality is implied by the fact that the definition requests that "already activated instances of tasks with a priority higher or equal to k are pending". At a point where $A_{k,i} = E_{k,i-1}$, there is no "already activated instances of tasks with a priority higher or equal to k are pending" since $\tau_{k,i-1}$ is completing and because $\tau_{k,i}$ is activated just at that time - and not before. The term "interference" is precisely motivated by this fact: inside an interference period, at each moment, pending work from the past has an influence on present or future executions and the corresponding response times; this is not the case at a time where $A_{k,i} = E_{k,i-1}$, because $\tau_{k,i}$ is activated when no work from the past is pending, see also the remark below.

The pending higher priority work $W_{1..k}(A_{k,n})$ is the result of the arrivals since $U_{k,n}$ and the available processing time $A_{k,n} - U_{k,n}$. More precisely, since during the interval $[U_{k,n}, A_{k,n})$ only instances of the tasks $\tau_1, \ldots, \tau_k$ are able to execute,

$$W_{1..k}(A_{k,n}) = S_{1..k}(U_{k,n}, A_{k,n}) - (A_{k,n} - U_{k,n}).$$

For the execution time of previous instances of τ_k and the considered instance $\tau_{k,n}$ itself we introduce

$$S_k^n = S_k(U_{k,n}, A_{k,n}) + C_{k,n}. \tag{5}$$

Thus the execution end $E_{k,n}$ of $\tau_{k,n}$ satisfies

$$E_{k,n} = \min\{t > U_{k,n} \mid S_{1..k-1}(U_{k,n}, t) + S_k^n = t - U_{k,n}\}. \tag{6}$$

The execution end is the smallest solution of a fixed point equation, see Appendix A. If the term $S_{1..k-1}(U_{k,n}, t)$ is replaced by a function that majorizes the term for every time t, then the solution of the new fixed point equation is larger than $E_{k,n}$, i.e. it is a bound for $E_{k,n}$. This fact, and the the concept of *majorizing work arrival function* introduced in the next section will allows us to derive response time bounds.

Remark: In [4], Lehoczky has introduced the concept of busy periods, which has been used by Tindell et al. in [5] to derive response time bounds for (sporadically) periodic tasks in the general case where deadlines may be longer than periods. A busy period consists in one or several adjacent interference periods separated by times where the past has no influence on the future at the considered priority level [8], or in other words, see [10] footnote page 40, by idle periods of zero length. Interference periods are needed for equation (6). The derived response time bounds are however the same whether interference or busy periods are used. The interested reader is again referred to [8] for further details..

4.2 Majorizing work arrival functions

In this section, we discuss briefly several types of tasks encountered in the literature and show how to majorize their WAF by *majorizing work arrival functions* (MWAF for short, see Definition 2). The MWAF are formulae based on the worst-case characteristics of the tasks. We derive MWAF for each considered type of tasks but assume that their *worst case execution times* (WCET) are known. Techniques for finding WCET are for instance described in [11, 12].

Consider a task τ_k with smallest cycle time T_k and longest execution time C_k. Suppose t is a time inside a level-k interference $[U^k, V^k)$. Let A_{k,n_0} be the first release of τ_k after or at U^k and A_{k,n_1} be the last before t. Suppose there are i instances in the interval $[U^k, t)$, i.e. $i = n_1 - n_0 + 1$. The amount of work $\sum_{n=n_0}^{n_1} C_{k,n}$ is thus bounded by $i \cdot C_k$. Assuming there is at least one release, the last release takes place at

$$A_{k,n_1} = A_{k,n_0} + \sum_{n=n_0}^{n_1-1} T_{k,n} \geqslant U^k + (i-1) \cdot T_k.$$

Furthermore $t \geqslant A_{k,n_1}$ and hence $i \leqslant 1+(t-U^k)/T_k$. Thus there are at most $i = \lceil (t-U^k)/T_k \rceil$ releases implying that the demand of the task is bounded by

$$s_k(t - U^k) = \left\lceil \frac{t - U^k}{T_k} \right\rceil \cdot C_k. \tag{7}$$

This is exactly the demand of a task with execution times $c_{k,n} = C_k$, activated first at $a_{k,0} = 0$ and then at $a_{k,n} = n \cdot T_k$. In a set of periodic tasks, each task has a MWAF of the form (7) and thus all task start at the same time $a_{k,0} = 0 \ \forall k$ in corresponding majorizing release pattern. Thus, we recognize the *critical instance* [1], where all tasks are activated at the same time and with the shortest inter-release times. The function $s_k(x)$ is a bound on the quantity of work arrived from task τ_k in any interval of length x. It can also be seen as a work arrival function, hence the name “majorizing work arrival function”.

In a similar way, one can prove that for a task with a *sporadically-periodic* [5] worst-case profile a majorizing function is given by [8]:

$$s_k(x) = C_k \cdot \left(\left(\left\lceil \frac{x}{T_k^{(2)}} \right\rceil - 1 \right) \cdot N_k + \left\lceil \frac{x - (\lceil x/T_k^{(2)} \rceil - 1) \cdot T_k^{(2)}}{T_k^{(1)}} \right\rceil \wedge N_k \right), \tag{8}$$

where $T_k^{(1)}$ is the smallest *inner cycle* time, $T_k^{(2)}$ is the smallest *outer cycle* time, N_k the largest number of releases in an outer cycle and C_k the longest execution time of an instance. The outer cycle is the time between two batches of instances with the inner cycle as time between releases. The right-hand side of (8) is equivalent to the expression given in [5] for the maximal demand from a sporadically periodic task on an interval of length x.

Multi-frame tasks [9] have been introduced to model more accurately the demands of tasks that decode MPEG video packets. The authors have derived feasibility conditions for this new task model. Their result can easily be extended to the computation of response time bounds, as we will see now. A multi-frame task is based on a repeating sequence of execution times $\vec{C} = (C_k^0, \ldots, C_k^{M_k-1})$ with smallest inter-release time P_k:

$$T_{k,n} \geqslant P_k \qquad C_{k,n} = C_k^{(n \bmod M_k)}.$$

In a similar way to that for the periodic tasks one can prove that $S_k(U^k,t) \leqslant \sum_{i=n_0}^{\lceil (t-U^k)/P_k \rceil} C_{k,i}$. A bound for this expression must take into account the fact that C_{k,n_0} could be any element of $\vec{C}$. To obtain a bound valid in every case, one must take the maximum over all possibilities. This is achieved by the vector $(\check{C}_k^0, \ldots, \check{C}_k^{M_k-1})$:

$$n = 0, \ldots, M_k - 1: \quad \sum_{i=0}^{n} \check{C}_k^i = \max_{0 \leqslant j < M_k} \sum_{i=j}^{j+n} C_k^{(i \bmod M_k)} .$$

Therefore, the function

$$s_k(x) = \sum_{i \in \mathbb{N}} \check{C}_k^{(i \bmod M_k)} \cdot \mathbb{1}_{[i \cdot P_k < x]} = \lfloor \lceil x/P_k \rceil / M_k \rfloor \cdot \sum_{i=0}^{M_k-1} \check{C}_k^i + \sum_{i=0}^{(\lceil x/P_k \rceil \bmod M_k)-1} \check{C}_k^i \tag{9}$$

is a bound on the quantity of work arrived in any interval $[u, u+x)$.

A task with $C_k^i = \check{C}_k^i$ is called *accumulatively monotonic* [9]. In that case, the bound $s_k(x)$ represents the worst case pattern, otherwise it is a conservative bound.

Consider finally the case of *tick-scheduling*, where jitter can occur between the time when an instance is ready for execution and the time where it is actually taken into account by the scheduler [5]. The latter time must be considered as the actual release time, to comply with the non-idling hypothesis. The effect of the jitter can be taken into account by the majorizing release pattern. With the length of time between two ticks T_{tick} a bound on the jitter, Equation (7) for example would become

$$s_k(x) = \lceil (x + T_{tick})/T_k \rceil \cdot C_k. \tag{10}$$

In the case where $T_{tick} > T_k$, there would be several simultaneous release at the beginning of the pattern: $a_{k,n} = 0$ for $n = 0, \ldots \lceil T_{tick}/T_k \rceil - 1$.

In all these examples appears a function $s_k(.)$ with the property that $s_k(x)$ is a bound for the amount of work due to released instances of the task τ_k in any interval of length x, i.e. a bound for $S_k(u, u+x)$. Furthermore, $s_k(.)$ and the actual WAF have similar properties. Recall that $S_k(U,t)$ does not account for eventual releases at t. It implies that $S_k(U,t)$ is a left-continuous function in t. Also, $S_k(U,t)$ is a step function in t, which increases just after the release of an instance of τ_k. It can be checked that in each example above, $s_k(.)$ is also a left-continuous step function. These considerations motivate the general definition of majorizing work arrival functions.

Definition 2
A majorizing work-arrival function *of a task τ_k, is a left-continuous step-function s_k, such that for all intervals starting at some time u, the WAF of τ_k satisfies the bound*

$$S_k(u, u+x) \leqslant s_k(x) \qquad \forall\, u \geqslant 0, x \geqslant 0. \tag{11}$$

Notice that a MWAF can represent the actual worst case demands of a task or only a (conservative) bound. This is of no importance at the current level of abstraction. Furthermore, choosing a task model could already imply bounds on demands that can never be realized by the real-world task. Sporadically periodic tasks and multi-frame task have been introduced, in order to obtain tighter bounds on the demands of the modeled real-world task.

If necessary, it is possible to consider a *set* of MWAF's to characterize more precisely the demands of a task. For instance, with the multi-frame tasks discussed above, it is possible to use one MWAF for each possibility $C_{k,n_0} = C_k^i$, $i = 0, \ldots, M_k - 1$. Another application of this idea is the case of offset relations [13]. For more details on families of MWAF, the reader is referred to [8].

Since MWAF's have the same properties as ordinary WAF's, a sequence of virtual release times $a_{k,n}$ with execution times $c_{k,n}$ can be found such that

$$s_k(x) = \sum_n c_{k,n} \cdot \mathbb{1}_{[a_{k,n}<x]},$$

i.e. it can be seen as a WAF on the interval $[0, x)$, see Figure 1. The corresponding cycle times are $t_{k,n} = a_{k,n+1} - a_{k,n}$. We will refer to the set of majorizing WAF's of tasks in $\mathcal{T}$ as *majorizing release pattern*. The execution end of an instance of τ_k in the first interference period of a majorizing release pattern is of course

$$e_{k,n} = \min\{x > 0 \mid s_{1..k-1}(x) + s_k^n = x\} \tag{12}$$

and the corresponding response time is

$$r_{k,n} = e_{k,n} - a_{k,n}. \tag{13}$$

The major advantage of introducing the concept of MWAF is the resulting separation of the analysis of tasks from the analysis of scheduling policies. On one hand response time equations, which essentially depend on the scheduling policy, can be established independently of a particular type of task, see equation (12). On the other hand, the worst case behavior of a certain kind of task can be derived independently of the applied scheduling policy, as we have seen in this section. This reduces the overall complexity of the timing analysis by breaking it into separate smaller problems. In subsequent sections we apply this principle to the timing analysis of layered priorities, by considering tasks only under the form of (majorizing) WAF, without referring to any particular kind of task.

4.3 Response time bounds

Based on the concepts of MWAF's and interference periods defined in the previous sections,, the fundamental argument for finding response time bounds can be stated as follows.

Theorem 1
Every response time of a task τ_k in an FPP-layer is bounded by the response time of some of its instances in the first level-k interference period of a majorizing release pattern:

$$R_{k,n} \leqslant \max_{j=0,1,2,\ldots,j^*} r_{k,j}, \qquad j^* = \min\{j > 0 \mid e_{k,j} \leqslant a_{k,j+1}\}.$$

Proof: Let $\tilde{n}$ be the index of the instance in the majorizing release pattern, that satisfies

$$a_{k,\tilde{n}} \leqslant A_{k,n} - U_{k,n} < a_{k,\tilde{n}+1}. \tag{14}$$

Furthermore, let $s_k^n = \sum_{i=0}^n c_{k,i}$. As illustrated in Figure 1, we have $S_k^n \leqslant s_k^{\tilde{n}}$. To show this, we have to distinguish two cases. Notice first that $S_k^n = S_k(U_{k,n}, A_{k,n}) + C_{k,n} = S_k(U_{k,n}, A_{k,n+1})$.

- $A_{k,n+1} - U_{k,n} \leqslant a_{k,\tilde{n}+1}$: in this case $s_k(A_{k,n+1} - U_{k,n}) \leqslant s_k(a_{k,\tilde{n}+1})$ since s_k is increasing. With (11), we obtain

$$S_k^n = S_k(U_{k,n}, A_{k,n+1}) \leqslant s_k(A_{k,n+1} - U_{k,n}) \leqslant s_k(a_{k,\tilde{n}+1}) = s_k^{\tilde{n}}.$$

- $A_{k,n+1} - U_{k,n} > a_{k,\tilde{n}+1}$: in this case $S_k(U_{k,n}, A_{k,n+1}) = S_k(U_{k,n}, U_{k,n} + a_{k,\tilde{n}+1})$, since by definition of $\tilde{n}$, $a_{k,\tilde{n}+1} > A_{k,n} - U_{k,n}$. With (11), we obtain

$$S_k^n = S_k(U_{k,n}, A_{k,n+1}) = S_k(U_{k,n}, U_{k,n} + a_{k,\tilde{n}+1}) \leqslant s_k(a_{k,\tilde{n}+1}) = s_k^{\tilde{n}}.$$

The execution end of $\tau_{k,\tilde{n}}$ is

$$e_{k,\tilde{n}} = \min\{x > 0 \mid s_{1..k-1}(x) + s_k^{\tilde{n}} = x\} \ . \tag{15}$$

Inequality (11) for tasks $\tau_1, \ldots, \tau_{k-1}$ and $S_k^n \leqslant s_k^{\tilde{n}}$ implies $S_{1..k-1}(U_{k,n}, t) + S_k^n \leqslant s_{1..k-1}(U_{k,n} - t) + s_k^{\tilde{n}}$ and hence Proposition 1 (see Appendix A) implies

$$E_{k,n} - U_{k,n} \leqslant e_{k,\tilde{n}}. \tag{16}$$

Now, (16) and (14) imply $R_{k,n} \leqslant r_{k,\tilde{n}}$. So far, we have identified for each response time $R_{k,n}$ a bound $r_{k,\tilde{n}}$. If we take the maximum of these bounds, we obtain a bound for all response times. Recall that $\tilde{n}$ is the smallest index, i.e. $\tau_{k,\tilde{n}}$ is the first instance, such that $S_k^n \leqslant s_k^{\tilde{n}}$. Thus, to be able to determine the maximum, we need to know how large $\tilde{n}$ can be.

Suppose $\tau_{k,n}$ is the first instance in a level-k interference period. In the interval $[U_{k,n}, A_{k,n})$ the processor is occupied by the tasks $\tau_1, \ldots, \tau_{k-1}$. Because of (11), the same will be true for the majorizing release pattern during the interval $[0, A_{k,n} - U_{k,n})$. Since $a_{k,\tilde{n}} \leqslant A_{k,n} - U_{k,n}$, $\tau_{k,\tilde{n}}$ is part of the first level-k interference period.

Suppose now, that $\tau_{k,n}$ is not the first instance, i.e. $\tau_{k,n-1}$ and $\tau_{k,n}$ are part of the same level-k interference period, then $E_{k,n-1} > A_{k,n}$. Suppose $R_{k,n-1}$ and $R_{k,n}$ are both bounded by two different response times of the majorizing release pattern, i.e. suppose $\widetilde{n-1} < \tilde{n}$. Then

$$a_{k,\tilde{n}} \leqslant A_{k,n} - U_{k,n} < E_{k,n-1} - U_{k,n} \leqslant e_{k,\widetilde{n-1}} \leqslant e_{k,\tilde{n}-1},$$

meaning that $\tau_{k,\tilde{n}}$ starts before $\tau_{k,\tilde{n}-1}$, i.e. the bounds we are looking for are the response times of instances in the first level-k interference period, remember (4). Thus we only need to compute $r_{k,j}$ for $j = 0, 1, 2, \ldots$ while $a_{k,j} < e_{k,j-1}$. ■

An overview of the algorithm implementing this result is given in Table 2. The function $f(k,t)$ (on the right) returns the execution demands $s_{1..k-1}(t)$ on the interval $[0,t)$. The variable c contains always the appropriate value of s_k^n. Both are used in lines 5- 8 to recursively compute the execution end $e_{k,n} = \min\{t > 0 \mid f(k,t) + c = t\}$ (see the discussion in appendix A). Convergence is guaranteed, under the following conditions of stability. If for each τ_k there exist $\sigma_k, \rho_k \in \mathbb{R}_+$ such that $s_k(x) \leqslant \sigma_k + \rho_k \cdot x$ and $\sum_{i=1}^{m} \rho_i < 1$ then the majorizing release pattern is *stable*, which means that the processor is not overloaded in the long run. If the majorizing pattern is based on worst-case execution and inter-release times, then $\sigma_k = C_k$ and $\rho_k = C_k/T_k$ is the choice with the smallest possible value for ρ_k. For sporadically periodic tasks it would be $\sigma_k = N_k \cdot C_k$ with $\rho_k = N_k \cdot C_k/T_k^{(2)}$ and for a multi-frame task $\sigma_k = \sum_{i=0}^{M_k-1} C_k^i$ with $\rho_k = \sigma_k/(M_k \cdot P_k)$. The rate ρ_k is a bound on the average demand of a task. For a multi-frame task one could also use its worst execution and

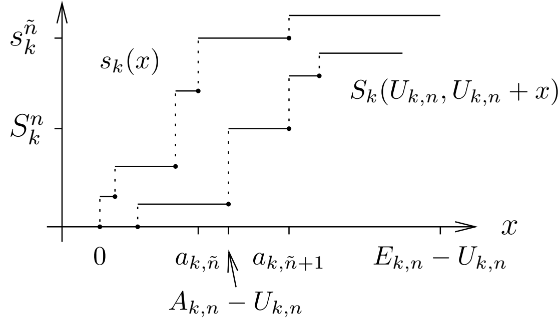

Figure 1: Illustration of (11) and (14).

inter-release time, but the corresponding ρ_k could be higher than 1, although the task alone would never overload the processor. In such a case, the worst-case demand of the task would drastically be overestimated, which underlines the gain of using majorizing release functions that are as tight as possible.

In Table 2 the function $s(k,t)$ implements a generic algorithm for computing values of work arrival functions. For most tasks more efficient versions can be derived, based on the formulas given in Section 4.2. This one has to be used if the majorizing WAF is derived from a sample.

The level-k interference period ends as soon as an instance finishes execution before or at the release time of the next instance, see line 14 of Table 2.
We conclude this section on FPP with some remarks:

1. Majorizing WAF's and thus release patterns are not unique. Their tightness has a direct impact on the tightness of the response time bounds. These bounds are maxima if the majorizing release pattern can actually be realized.

2. The scheduling policies in other layers have no effect on the response time bound computation, only majorizing release patterns are needed.

5 Round Robin

To the best of our knowledge, response time bounds under Round Robin have not yet been derived. In this section we derive these bounds which are also usable in the context of Posix 1003.1b (layered preemptive priority policy, based on FPP and RR).

The underlying idea of the Round Robin policy is to reserve a certain rate of the available processor capacity for each task. This is realized by allowing repeatedly each task to execute for a certain amount of time (quantum) without interruption - if it has pending instances. As a result, a period of preemption, i.e. a period where the other tasks are allowed to execute, lasts at most as long as the sum of all other quanta. The part of the processor capacity reserved for a task is then the ratio of its quantum to the sum off all quanta. This is not a typical real-time policy, but if response time bounds are known, a feasibility test can be performed. Furthermore, if tasks in a layer are also feasible under a different policy than FPP, say RR, this policy can be chosen in order to satisfy additional properties (regularity of

```
1  funct time RespTime(task k)
2    max := 0; n := 0; a := 0;
3    c := c_{k,0}; w := c_{k,0};
4    repeat
5      repeat
6        e := w;
7        w := f(k, e) + c;
8      until w = e;
9      r := e − a;
10     if r > max then max := r; fi
11     a := a + t_{k,n};
12     n := n + 1;
13     c := c + c_{k,n};
14   until a ⩾ e;
15   return max;
16 end
```

```
funct time f(task k, time t)
  w := 0
  for i := 1 to k − 1 do
      w := w + s(i, t);
  od
  return w;
end

funct time s(task i, time t)
  w := 0; a := 0; j := 0;
  while a < t do
        w := w + c_{i,j};
        a := a + t_{i,j};
        j := j + 1;
  od
  return w;
end
```

Table 2: Computing a response time bound.

service, fairness, etc). This is true in particular because the alternative choice for some layer has no effect on the response time bounds of tasks in other layers. The aim of this section is to derive bounds for tasks scheduled in a Round Robin layer under LPP and to give reasons for using Round Robin.

First we give a precise description of the policy. Let τ_k be a task in an RR-layer λ_l. Repeatedly, it gets the opportunity to execute during a time quantum of at most Ψ_k. If the task has no pending instance or less pending work than the quantum amounts, then the rest of the quantum is lost for that task and it has to await the next cycle. When an instance finishes without completely using a quantum, then the next instance of the same task is allowed to use the rest of the quantum - if it is already pending at the time when the previous instances is completing.

As a result, the time between two consecutive opportunities for the instances of a task to execute may vary, depending on the actual demands of other tasks, but it is bounded by $\Psi^l = \sum_{\tau_k \in \lambda_l} \Psi_k$ in any interval where the considered task has pending instances at any moment.

5.1 Interference periods

Under RR, level-k interference periods make no sense, since any task of the layer can preempt any other. A different kind of period must be found. The discussion below shows that with the following definition response time bounds can be found.

Definition 3
A τ_k-RR-interference period *is a time interval* $[U_k^{RR}, V_k^{RR})$ *such that started instances of* τ_k *or of tasks from higher priority layers, that is* $\tau_1, \ldots, \tau_{m_{l-1}}$*, are pending neither at its beginning* U_k^{RR} *nor at its end* V_k^{RR} *but at any other time between* U^k *and* V^k *there is at least one such instance pending.*

Such a period ends with the execution end of an instance of τ_k but can start with the execution beginning of any instance of the task or of higher priority layers. At U_k^{RR} started instances of other tasks from the layer can be pending.

Furthermore, the processing unit is busy at any moment for tasks of the layer or for tasks from higher priority layers. For the considered task τ_k it implies in particular that all its quanta which are situated inside of $[U_k^{RR}, V_k^{RR})$, except for the last, are completely used, i.e. have the maximal size Ψ_k. To see this, suppose there is a quantum in $[U_k^{RR}, V_k^{RR})$, which is shorter than Ψ_k. Then, at its end an instance finishes while the next has not yet been released, implying it is the end of the τ_k-RR-interference period.

Recall that each instance of a task in a FPP layer is part of some level-k-interference period. Here, each instance of a task is part of some τ_k-RR-interference period. A special case is that of periods containing only one quantum because the instance's execution time is shorter than Ψ_k.

Basic response time bound We now turn to the timing analysis of RR. Under RR, execution times of instances are subdivided into quanta with release times that depend on RR-cycles of the scheduler in the past. RR-cycles in return depend on the way tasks actually used their quanta in the past. Because of this kind of feedback, an exact response-time formula like (6) is rather complex [8]. To avoid this difficulty and because an exact formula is not necessarily needed for finding response time bounds, we first derive a *basic* response time bound, for a particular instance in some interference period and adapt it in a second step to the majorizing release pattern.

Consider therefore a τ_k-RR-interference period $[U_k^{RR}, V_k^{RR})$. As explained above, the task τ_k completely uses all of the quanta the scheduler proposes to it, except perhaps for the last one, where the last instance finishes at V_k^{RR}. Before an instance $\tau_{k,n}$ is executed, the previous instance $\tau_{k,n-1}$ must first complete. Thus a total amount of $S_k^n = S_k(U_k^{RR}, A_{k,n}) + C_{k,n}$ units of work will be executed for τ_k in the interval $[U_k^{RR}, E_{k,n})$, which corresponds to $\lceil S_k^n/\Psi_k \rceil$ RR cycles. In each RR cycle the other tasks of the layer may use the processor for up to $\overline{\Psi}_k = \Psi^l - \Psi_k$ units of time. Thus their demand in the interval $[U_k^{RR}, E_{k,n})$ is bounded by $\lceil S_k^n/\Psi_k \rceil \cdot \overline{\Psi}_k$. On the other hand it cannot exceed the actual demands, which is due to instances of other tasks of the layer that might be pending at U_k^{RR}, and the instances arriving during $[U_k^{RR}, t)$:

$$\overline{W}_k(U_k^{RR}) = \sum_{\tau_i \in \lambda_l, \tau_i \neq \tau_k} W_i(U_k^{RR}) \quad \text{and} \quad \overline{S}_k(U_k^{RR}, t) = \sum_{\tau_i \in \lambda_l, \tau_i \neq \tau_k} S_i(U_k^{RR}, t)$$

In addition, there is the demand $S_{1..m_{l-1}}(U_k^{RR}, t)$ from tasks in higher priority layers. Notice that by Definition 3, $W_{1..m_{l-1}}(U_k^{RR}) = 0$. Hence the total demand of tasks other than τ_k is bounded by

$$\overline{\Psi}_k(U_k^{RR}, t) = \min\left(\left\lceil \frac{S_k^n}{\Psi_k} \right\rceil \cdot \overline{\Psi}_k, \quad \overline{W}_k(U_k^{RR}) + \overline{S}_k(U_k^{RR}, t) \right) + S_{1..m_{l-1}}(U_k^{RR}, t). \qquad (17)$$

With (17) as a bound for demands of other tasks of the layer, we deduce using Proposition 1 that

$$E_{k,n} \leqslant E^*_{k,n} = \min\{t > U_k^{RR} \mid \overline{\Psi}_k(U_k^{RR}, t) + S_k^n = t - U_k^{RR}\}$$

is a bound for the actual execution end $E_{k,n}$. Thus, $R^*_{k,n} = E^*_{k,n} - A_{k,n}$ is a bound for the response time $R_{k,n}$. We call $R^*_{k,n}$ a *basic* response time bound. As before we use small letters for the counterpart in the majorizing release pattern:

$$e^*_{k,n} = \min\{x > 0 \mid \overline{\psi}_k(x) + s_k^n = x\}. \tag{18}$$

and

$$r^*_{k,n} = e^*_{k,n} - a_{k,n}. \tag{19}$$

5.2 Response time Bounds

We prove in this section that the basic response time bounds in the first τ_k-RR-interference period of a majorizing release pattern, are also bounds for the actual response times of τ_k.

Theorem 2
The response time of any instance $\tau_{k,n}$ in an RR-layer is bounded by the basic response time bound $r^*_{k,\tilde{n}}$ *of some instance $\tau_{k,\tilde{n}}$ in the first τ_k-RR-interference period of the majorizing release pattern. Therefore:*

$$R_{k,n} \leqslant \max_{j=0,1,2,\ldots,j^*} r^*_{k,j} \qquad \textit{where} \qquad j^* = \min\{j \mid e_{k,j} \leqslant a_{k,j+1}\}.$$

Remark: This theorem is very similar to Theorem 1, because the method for finding response time bounds is basically the same. The difference lies in the specific definition of the response times (*basic response time bounds*) and the interference periods (τ_k*-RR-interference period*).
Proof: The preemption from other tasks (17) consists in two parts, which we will bound separately. First we derive a bound for

$$\overline{W}_k(U_k^{RR}) + \overline{S}_k(U_k^{RR}, t) + S_{1..m_{l-1}}(U_k^{RR}, t)\ . \tag{20}$$

The workload $\overline{W}_k(U_k^{RR})$ is the result of the scheduling since the begin U^{m_l} of the level-m_l interference period in which $\tau_{k,n}$ is executed. At U^{m_l} there is no pending work at level m_l, but at each moment until U_k^{RR} there is such work pending, implying that the processor only executes $\tau_1, \ldots, \tau_{m_l}$ during $[U^{m_l}, U_k^{RR})$. Therefore, $W_{1..m_l}(U_k^{RR}) = S_{1..m_l}(U^{m_l}, U_k^{RR}) - (U_k^{RR} - U^{m_l})$. On the other hand,

$$W_{1..m_l}(U_k^{RR}) = W_{1..m_{l-1}}(U_k^{RR}) + \overline{W}_k(U_k^{RR}) + W_k(U_k^{RR})$$

and by Definition 3, $W_k(U_k^{RR}) = 0$ and $W_{1..m_{i-1}}(U_k^{RR}) = 0$. Thus

$$\overline{W}_k(U_k^{RR}) = S_{1..m_l}(U^{m_l}, U_k^{RR}) - (U_k^{RR} - U^{m_l}).$$

Since $S_{1..m_l}(U^{m_l}, U_k^{RR}) = S_{1..m_{l-1}}(U^{m_l}, U_k^{RR}) + S_k(U^{m_l}, U_k^{RR}) + \overline{S}_k(U^{m_l}, U_k^{RR})$ and because of the additivity (2) of WAF's, the term (20) becomes

$$\begin{aligned}
&\overline{W}_k(U_k^{RR}) + \overline{S}_k(U_k^{RR}, t) + S_{1..m_{l-1}}(U_k^{RR}, t) \\
&= S_k(U^{m_l}, U_k^{RR}) + \overline{S}_k(U^{m_l}, t) + S_{1..m_{l-1}}(U^{m_l}, t) - (U_k^{RR} - U^{m_l}) \\
&\leqslant s_k(U_k^{RR} - U^{m_l}) + \overline{s}_k(t - U^{m_l}) + s_{1..m_{l-1}}(t - U^{m_l}) - (U_k^{RR} - U^{m_l}) \\
&= s_k(u) + \overline{s}_k(t - U_k^{RR} + u) + s_{1..m_{l-1}}(t - U_k^{RR} + u) - u.
\end{aligned} \tag{21}$$

where we have introducing MWAF's (11) and $u = U_k^{RR} - U^{m_l}$. On the other hand, it is easily seen from the identity $v^{m_l} = \min\{x > 0 \mid s_{1..m_l}(x) = x\}$ and using $S_{1..m_l}(U^{m_l}, t) \leqslant s_{1..m_l}(t - U^{m_l})$, that $V^{m_l} - U^{m_l} \leqslant v^{m_l}$, i.e. any level-$m_l$ interference period is shorter than the first level-m_l period of the majorizing release pattern. Since $[U^{m_l}, U_k^{RR}) \subset [U^{m_l}, V^{m_l})$, we have therefore $u = U_k^{RR} - U^{m_l} \leqslant v^{m_l}$. Thus a bound for (21) is $s_k^*(t - U_k^{RR})$, where s_k^* is given by

$$s_k^*(x) = \max_{0 \leqslant u \leqslant v^{m_l}} s_k(u) + \overline{s}_k(u + x) + s_{1..m_{l-1}}(u + x) - u. \tag{22}$$

We turn now to the other part of (17). From the definition of $\tilde{n}$, given by equation (14), it follows immediately that

$$\left\lceil \frac{S_k^n}{\Psi_k} \right\rceil \cdot \overline{\Psi}_k + S_{1..m_{l-1}}(U_k^{RR}, t) \leqslant \left\lceil \frac{s_k^{\tilde{n}}}{\Psi_k} \right\rceil \cdot \overline{\Psi}_k + s_{1..m_{l-1}}(t - U_k^{RR}).$$

Thus a bound for (17) is given by a similar expression

$$\overline{\psi}_k(t - U_k^{RR}) = \min\left(\left\lceil \frac{s_k^{\tilde{n}}}{\Psi_k} \right\rceil \cdot \overline{\Psi}_k + s_{1..m_{l-1}}(t - U_k^{RR}),\ s_k^*(t - U_k^{RR}) \right). \tag{23}$$

From the proof of Theorem 1, we know that $A_{k,n} \geqslant U_k^{RR} + a_{k,\tilde{n}}$. Consider then

$$e_{k,\tilde{n}}^* = \min\{x > 0 \mid \overline{\psi}_k(x) + s_k^{\tilde{n}} = x\}. \tag{24}$$

We call $r_{k,\tilde{n}}^* = e_{k,\tilde{n}}^* - a_{k,\tilde{n}}$ *basic* response time bound in the majorizing release pattern. With $s_k^{\tilde{n}}$ defined by (14), Proposition 1 applies to $E_{k,n}^*$ and $e_{k,\tilde{n}}^*$. Thus $E_{k,n} \leqslant U_k^{RR} + e_{k,\tilde{n}}^*$, implying finally with (14)

$$R_{k,n} \leqslant E_{k,n}^* - A_{k,n} \leqslant r_{k,\tilde{n}}^*.$$

As in Theorem 1, it can be proven that it is sufficient to compute $r_{k,j}^*$ for $j = 0, 1, 2, \ldots$ while $a_{k,j} < e_{k,j-1}^*$, to determine the response time bound. ■

Given the MWAF's, $s^*(.)$ can be determined algorithmically.

The basic algorithm of Table 2 remains the same. Only the function that returns the relevant execution time demands must be adapted. The changes concern the other tasks of the layer containing the task under study, see Table 3.

```
1 funct time f(task k, instance n, time t)
2   w := 0;
3   for i := 1 to m_{l-1} do
4       w := w + s(i, t);
5   od
6   w := w + ceil(s_k^n / Ψ_k) * Ψ̄_k;
7   return min(w, s_k^*(t));
8 end
```

Table 3: Total amount of work: τ_k in a RR-layer

In the literature, it is sometimes claimed that SCHED_RR is mainly useful for the scheduling of tasks in the lower priority ranges with soft timing constraints (see [14] p. 163). One reason is perhaps the lack of schedulability analysis. Consider a subset λ_l of tasks that perform some work of equal importance from the point of view of the application, such as transmitting data to other stations through the network or controlling external devices. We claim that if the complete set of tasks is schedulable whether SCHED_RR and SCHED_FIFO is used for the tasks of the subset λ_l, then using SCHED_RR may be better.

We illustrate this with the comparison reported in Table 4. The data includes the description of the tasks using symbols already introduced (except for the relative deadlines D_k), their priority level `p`, their response time bounds and the maximum, average and standard deviation of the response time, obtained by simulation. Also displayed is the average CPU utilization of the task (ρ_k) and the cumulated utilizations of tasks with a higher or equal priority ($\rho_{1..k}$).

The response times have been computed with TkRts [15]. The column "bound" gives the response time bounds computed by our algorithm whereas the column "max" gives the maximal response times obtained by simulation. It can be noticed that in FPP-layers the maxima are always equal to the bounds. This is due to two factors: on the one hand the FPP algorithm actually computes maximal response times and on the other the worst-case scenario is known and was simulated (*critical instant* see [1]). In the case of RR-layers the bounds computed by our algorithm are larger than the maxima obtained by simulation. There are two reasons: our algorithm is only able to compute upper bounds and the worst-case scenario is not known and was perhaps not simulated. In our example the difference is about 15%.

However, the use of RR has an interesting advantage: the maximum of standard deviations of the response times in the layer λ_l is lower. It means that there is less "jitter" in the availability date of the results produced by instances of task in λ_l. It provides, to some degree, more predictability to the behavior of the application. In the case of *distributed* applications, this may result in less bursty WAF's and reduced response time bounds at the receiving end.

6 Critical sections and semaphores

While a task is writing into a shared memory zone, it must not be preempted because it could leave the zone in an incoherent state, that could then be read by another task. More generally there are time periods, called *critical sections*, where a task should not be preempted by other tasks that need the same resource. To implement this, semaphores are commonly used to signal that a resource is currently being used. But resource contentions can modify preemptive scheduling policies. Under fixed preemptive priorities it induces *priority inversions*: consider as an example three tasks τ_1, τ_2, τ_3, which are given in decreasing order of priority. If after the beginning of a critical section of the lowest priority task τ_3, the highest priority task τ_1 needs the resource locked by τ_3, then τ_1 is blocked, which is as having a lower priority than τ_3. The medium priority task τ_2 is then able to preempt τ_1, by preempting the critical section of τ_3, which is a contradiction with the intended preemptive fixed priority policy. As a result, τ_1 can have longer response times than it would usually have. Normally, a transient overload at level j, has no effect on the response times of τ_1, but in the case of the priority inversion just described, it would be blocked as long as τ_2 is blocked by the execution of higher priority tasks. A prominent example is given by the initial difficulties with the scheduler in the "Mars

Layer	p	τ_k	C_k	T_k	D_k	bound	max	mean	stdev	ρ_k	$\rho_{1..m}$
FPP	10	T1	3	20	20	3	3.0	3.0	0.0	15.0	15.0
	9	T2	5	30	30	8	8.0	6.1	1.3	16.7	31.7
	8	T3	2	40	40	10	10.0	6.1	2.7	5.0	36.7
	7	T4	4	55	55	14	14.0	6.4	**2.9**	7.3	43.9
	6	T5	7	70	70	24	24.0	12.5	**4.4**	10.0	53.9
	5	T6	15	125	125	49	49.0	31.8	**7.8**	12.0	65.9
	4	T7	6	150	150	55	55.0	24.4	**13.2**	4.0	69.9
	3	T8	10	200	200	89	89.0	37.1	**18.9**	5.0	74.9
	2	T9	11	250	250	108	108.0	70.6	17.7	4.4	79.3
	1	T10	15	250	250	190	190.0	113.6	26.2	6.0	85.3

Layer	p	Ψ_k	τ_k	C_k	T_k	D_k	bound	max	mean	stdev	ρ_k	$\rho_{1..m}$
FPP	10		T1	3	20	20	3	3.0	3.0	0.0	15.0	15.0
	9		T2	5	30	30	8	8.0	6.1	1.3	16.7	31.7
	8		T3	2	40	40	10	10.0	6.1	2.7	5.0	36.7
RR	7	4	T4	4	55	50	50	46.0	10.4	**6.3**	7.3	43.9
	7	7	T5	7	70	70	50	45.0	14.2	**5.8**	10.0	53.9
	7	8	T6	15	125	125	89	72.0	32.3	**9.1**	12.0	65.9
	7	3	T7	6	150	150	89	72.0	22.5	**10.0**	4.0	69.9
	7	5	T8	10	200	200	89	72.0	27.6	**11.4**	5.0	74.9
FPP	6		T9	11	250	250	108	108.0	70.6	17.7	4.4	79.3
	5		T10	15	250	250	190	190.0	113.6	26.2	6.0	85.3

Table 4: Fixed priorities versus Round Robin.

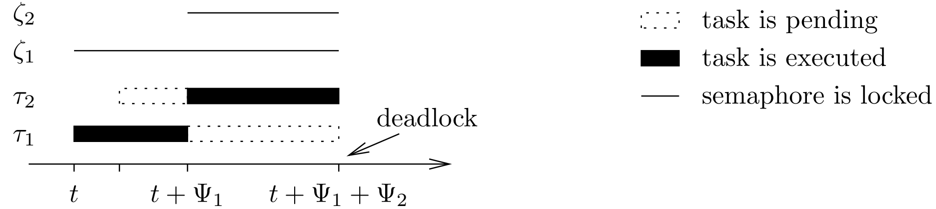

Figure 2: Deadlock with nested resources.

Path finder Robot Software" (see the RTSS'97 keynote speech and subsequent articles in sci.space.news).

To limit the length where a higher priority task has to wait for a resource to be freed, and to avoid deadlock problems with nested access to critical resources the *priority ceiling protocol* (PCP) [6] can be used with fixed preemptive priorities.

PCP-FPP: *When an instance of a task enters a critical section it locks the corresponding semaphore and its priority is changed to the associated ceiling, unless its current priority is higher than the ceiling. At the end of the critical section, the priority of the instance is set back to the normal priority associated with its task.*

In the example described above, it implies that during its critical section, τ_3 can not be preempted by τ_2 and thus, the priority inversion period is limited. For the response time analysis it implies that the execution of an instance of τ_1 can be blocked by at most one critical section of a lower priority task, that could need the same resource, see [6].

The question that arises now is how Round Robin layers are affected by semaphores and the PCP, designed for FPP. Recall that tasks which are scheduled in the same Round Robin layer have the same priority under Posix. Hence, if for example all tasks are scheduled in one Round Robin layer, then all priority ceilings are the same, equal to the priority of the layer. Increasing a priority to a ceiling has then strictly no effect. But semaphores without any additional mechanism, may induce deadlocks also under Round Robin. To understand how, let us consider a schematic example. Let a Round Robin layer consist in two tasks τ_1 and τ_2 (i.e. with the same priority) with nested access to the same resources ζ_1 and ζ_2 but in a different order. Suppose τ_1 needs ζ_1 immediately and ζ_2 after Ψ_1 units of execution time. Symmetrically, τ_2 needs ζ_2 immediately and ζ_1 after Ψ_2 units of execution time. Now, if τ_1 is released at t, and τ_2 just thereafter, then τ_1 locks ζ_1 at t and is executed until $t + \Psi_1$. Then τ_2 executes and locks ζ_2. At $t + \Psi_1 + \Psi_2$, τ_1 should again be executed, but it needs ζ_2, which is locked by τ_2. Thus τ_1 is blocked. On the other hand τ_2 is also blocked since it needs ζ_1 which is locked by τ_1, i.e. a deadlock occurs, see Figure 2.

Such a deadlock situation can not occur if tasks in a RR layer are not interrupted by the RR scheduler during a critical section. The RR scheduler usually interrupts a task as soon as it has used the processor for Ψ_k units of time. In order to prevent the interruption the tasks quantum Ψ_k must be ignored (or set to a very large value) while the task is inside a critical section. As a result, a task can then be interrupted only when it has released all locks. Thus, a task of the layer accesses the processor only when no semaphore is locked by a task of the RR layer. This implies that no deadlock can occur, because at any time at most one task of the RR layer is holding locks.

Notice that when a task of the RR layer enters a critical section and its priority is set to

a ceiling that is higher than the priority of the RR layer, then it is also not interrupted by the RR scheduler while it executes its critical section. On the other hand, if a ceiling is equal to the priority of a RR layer, the usual PCP rule is still applicable, if the RR scheduler does not interrupt the execution of a task whose priority would be set to the priority of the layer.

Thus, the deadlock preventing rule can more generally be stated as follows:

PCP-RR.1: *The time quantum of a task is ignored, while it executes a critical section.*

Notice that this rule may easily be implemented under Posix, by setting the time quantum to a very large value while the task executes its critical section. In Section 6.2 we derive response time bounds that account for the effects of PCP-RR.1, see (28). These bounds are rather pessimistic, because with PCP-RR.1 the behavior of the policy becomes difficult to analyse. The longer are the critical sections the less accurate is the bound (28).

Figure 3 shows a small example where the execution of two tasks can be compared in different cases. In Figure 3.a both tasks have no critical sections. It can be seen that the scheduler regularly alternates between pending tasks. In Figure 3.b, τ_1 has two critical sections during which it is not interrupted according to PCP-RR.1. As a result, τ_1 requires less RR cycles than before and completes its execution earlier to the disadvantage of τ_2. This example shows a phenomenon similar to priority inversions. Under Round Robin, while τ_1 and τ_2 are both pending, they should share the processor in the proportion of Ψ_1 to Ψ_2 time units. But critical sections distort this proportion, which the Round Robin scheduling policy tries to impose. The phenomenon can be limited as follows.

PCP-RR.2: *With each task τ_k of a RR layer is associated a processor utilization counter c_k. The counter is initially set to zero. It increases while an instance of the task is executed. The counters are used as follows:*

1. *Task selection: let τ_i be the pending task considered by the RR scheduler for being executed next in the current RR cycle:*
   - (a) *If c_i is smaller than Ψ_i, then (the oldest instance of) τ_i is chosen.*
   - (b) *If c_i exceeds or is equal to Ψ_i, then c_i is decreased by Ψ_i and τ_i is not selected. The next pending task is considered.*
2. *Task interruption (a task finishes or exceeds its quatum): let $\tau_{k,n}$ be the instance currently selected for execution:*
   - (a) *If c_k is smaller than or equal to Ψ_k when $\tau_{k,n}$ completes and $\tau_{k,n+1}$ is not pending, then c_k is set to zero.*
   - (b) *If τ_k does not execute a critical section when c_k exceeds Ψ_k then it is immediately interrupted. The counter c_k is set to zero.*
   - (c) *If τ_k does execute a (nested) critical section, when c_k exceeds Ψ_k, then it is interrupted as soon as it exits all critical sections. The counter c_k is then decreased by Ψ_k.*
3. *When the RR scheduler idles, i.e. no task of the RR layer is pending then all counters are reset to zero.*

Figure 3.c shows the effect of PCP-RR.2. After less than $\Psi_1 = 3$ units of time, τ_1 begins to execute a critical section. According to PCP-RR.1 it can than not be interrupted before the end of the critical section, where its counter is equal to $c_1 = 3 \cdot \Psi_1 = 9$. According

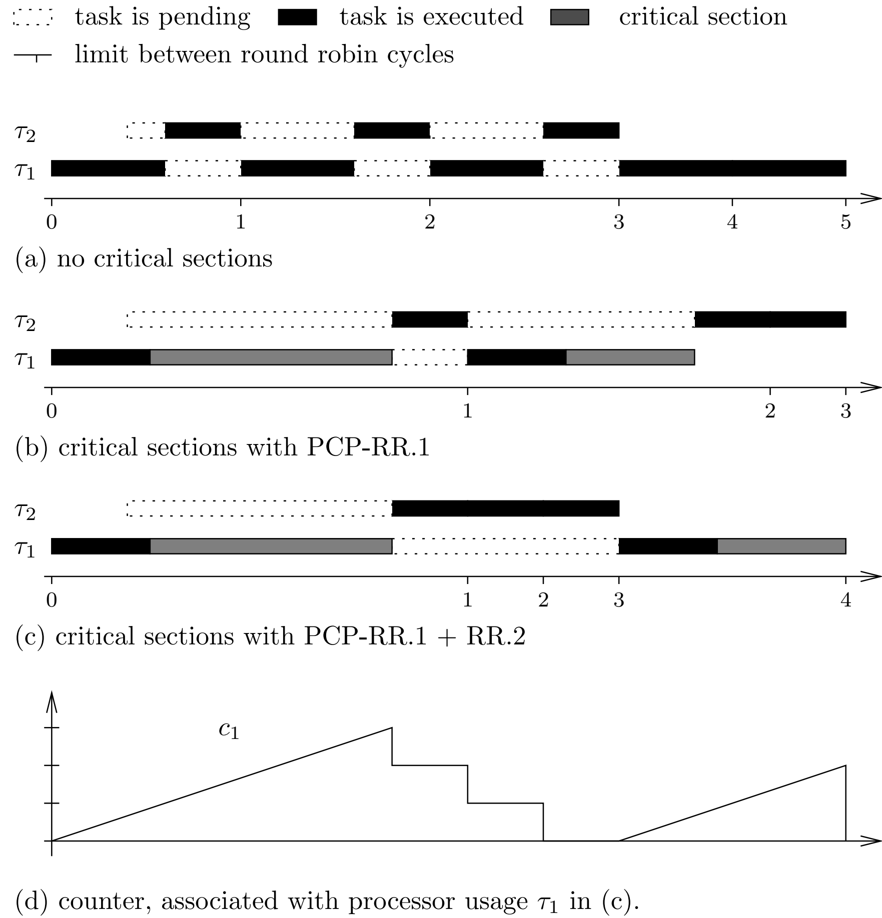

Figure 3: Critical sections under Round Robin: $C_1 = 15$, $\Psi_1 = 3$, $C_2 = 6$, $\Psi_2 = 4$.

to PCP-RR.2 (1.b), τ_1 will not be selected by the RR scheduler in the following 2 cycles (Figure 3.d shows the evolution of the counter c_1). During that time τ_2 is able to "recover" the lost quanta.

As can be seen from this example, PCP-RR.2 tends to reestablish the distorted access rates: (c) ressembles more to (a) than (b) does.

It is possible to define the Round Robin scheduling policy in terms of time dependent *priority functions* [8]. With the help of priority functions it can be seen that the PCP-RR 1+2 is designed according to the same principle as the PCP [6] for FPP and the dynamic PCP [16] for Earliest deadline first.

6.1 Response time bounds of tasks in FPP-layers

We have described above how a task can preempt a higher priority task by executing its critical section. The PCP does not prevent this from happening. Its aim is to limit the duration of such periods. It has been shown in [6] that under the FPP policy with PCP, during a level-k busy period a task τ_k can be preempted by only one critical section of a lower priority task. Since interference periods are sub-intervals of busy periods, the same holds true for them. To account for critical sections the execution end formula (6) changes to

$$E_{k,n} = \min\{t > U_{k,n} \mid S_{1..k-1}(U_{k,n}, t) + B_{k,n} + S_k^n = t - U_{k,n}\} \tag{25}$$

with $B_{k,n}$ being the length of the unique preempting critical section if it exists and $B_{k,n} = 0$ otherwise. Its value is bounded by the maximum length of critical sections of lower priority task that can lock the same semaphores as τ_k. This maximum is denoted b_k. Thus, the formula of execution end bounds needs to be changed to

$$e_{k,\tilde{n}} = \min\{t > 0 \mid s_{1..k-1}(t) + b_k + s_k^n = t\}. \tag{26}$$

From the point of view of a task in a FPP-layer, tasks with critical sections in lower priority RR-layers behave exactly like other lower priority tasks, because of PCP-RR.1. PCP-RR.2 only concerns tasks in RR-layers. Since (25) is still valid for a task in a FPP-layer, (26) gives a response time bound.

6.2 Response time bounds for tasks in RR-layers

PCP-RR.1 implies that a task τ_i may use the processor for more than Ψ_i units of time without interruption if it has critical sections. If τ_i starts to execute a critical section just before its quantum expires, then it may use the processor for up to $\Psi_i + z_i$ units of time. It depends on the number and locations of the critical sections in the code of τ_i, when the task is actually using the processor for longer than foreseen by RR. It is therefore difficult to exactly account for the resulting effect on the response time of an other task of the RR layer. A response time bound may nevertheless be obtained by assuming a pessimistic case where each other task uses the processor during $\Psi_i + z_i$ units of time in each RR cycle but the task τ_k under study only for during Ψ_k units. It is an acceptable overestimation if z_i is relatively small compared to Ψ_i.

The blocking factor b_k is the same for all tasks in the Round Robin layer. It is the longest critical section of a task with a lower priority than the layer and due to a resource which is also needed by tasks of the RR layer or higher priority layers. For the critical sections of the

RR layer, we introduce $\overline{z}_k = \sum_{\tau_i \in \lambda_l, \tau_i \neq \tau_k} z_i$. Thus, a response time bound is obtained from the (pessimistic) execution end:

$$e^*_{k,n} = \min\{x > 0 \mid \overline{\psi}_k(x) + b_k + s^n_k = t\} \tag{27}$$

with

$$\overline{\psi}_k(x) = \min\left(\left\lceil \frac{s^{\tilde{n}}_k}{\Psi_k} \right\rceil \cdot \left(\overline{\Psi}_k + \overline{z}_k\right) + s_{1..m_l-1}(x),\ s^*_k(x) \right). \tag{28}$$

The effect of PCP-RR.2 is that when a task τ_i uses the processor during $\Psi_i + z_i$ units of time in a single RR cycle, then it is not allowed to use the following quanta until the excess is compensated. It implies that it can only enter a critical section when it has not used the processor more than allowed in the past.

Suppose a task τ_i executes a critical section in the jth RR cycle after U^{RR}_k. According to PCP-RR.2, at beginning of this cycle, τ_i has used the processor for at most $(j-1) \cdot \Psi_i$ units of time and at the end of the RR cycle, for at most $z_i + j \cdot \Psi_i$. If in the jth RR cycle τ_i is not inside a critical section then it has used the processor for at most $j \cdot \Psi_i$ units of time at the end of the cycle. Thus,

$$\overline{\psi}_k(x) = \min\left(\left\lceil \frac{s^{\tilde{n}}_k}{\Psi_k} \right\rceil \cdot \overline{\Psi}_k + \overline{z}_k + s_{1..m_l-1}(x),\ s^*_k(x) \right). \tag{29}$$

Notice that the blocking factor b_k appears as an isolated additional constant in equation (27). With PCP-RR.2, the same is true for $\overline{z}_k$, which accounts for the critical sections of the tasks in the RR layer.

7 Case-study

We will consider an application, running on a uniprocessor machine with a Posix 1003.1b compliant OS, that monitors and controls a real-time device (inspired from [14] p. 152, see Figure 4). The aim of this case study is to demonstrate that a timing analysis of a complex RR/FPP interaction is indeed possible. For a more detailed study the reader is referred to [17].

The application is structured as 7 tasks. The informations on the controlled system's state are gathered by task A. Task B analyses the data, takes the appropriate decisions and sends them back to the controlled system. Task C is a background computation task that keeps statistics up to date or implements some artificial intelligence techniques. Informations on the system's state evolution are displayed by task D and logged onto a file by task E. Task D also takes care of user input. In the case of an abnormal system state, an alarm condition is triggered and task F is executed for urgent data collecting, task G will then take decisions and send back the control data to the system. Each task is either periodic or sporadic with a worst-case time between successive invocations and each task owns a scheduling policy with a given priority fixed for the entire system's lifetime. In this case-study, in order to stay consistent with the notations from the previous sections, we will adopt the priority numbering scheme "the smaller the number, the higher the priority" which is the inverse of Posix.

In the development process of a real application, one is likely to be working with libraries from third-part suppliers or from the OS vendor for functions like network communication,

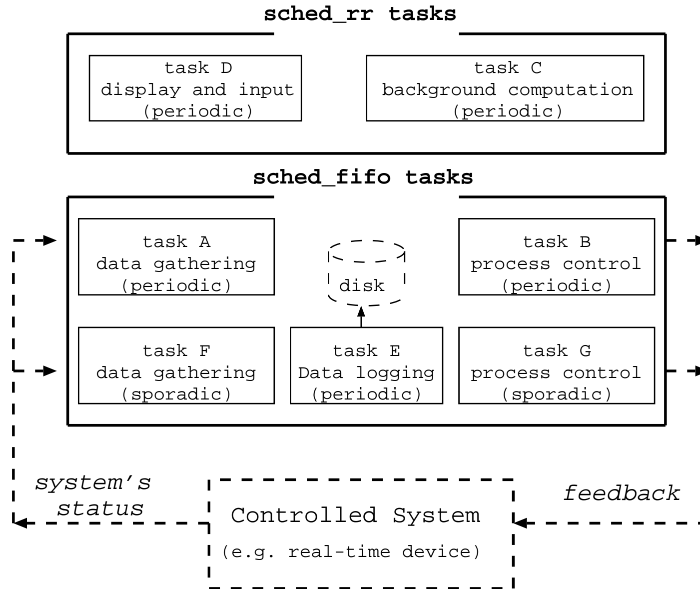

Figure 4: Application structure

resource management or graphic display. When writing an application, it may a priori seem reasonable to run the most important activity (either a task or a thread) within our code using SCHED_FIFO policy at the maximum priority. One must be aware that such a priority choice may prevent other libraries and even the OS (e.g. kernel I/O drivers) from performing work on which the application relies [18]. That is why the priorities must be chosen with respect to all components of an application and not only considering the code that one actually writes.

Note that tasks may share the same priority level even if they are scheduled according to different policies. This case is tractable, in the case of SCHED_RR and SCHED_FIFO tasks, by considering a unique SCHED_RR layer and by giving the SCHED_FIFO tasks a time quantum equal to their processing time.

Tasks C and E share a common resource which is a memory mapped file with task E updating periodically the underlying file. The maximum duration of the critical section, previously denoted z_k, is 6ms for task C and 15ms for task E. The characteristics of the tasks and the schedulability analysis, performed with TkRts [15], are summarized in Table 5. The symbols in the table have been described for Table 3.

For comparison, we report in Table 6 the response times bounds in the case of FPP, with both possible priority assignments, and with semaphores (left) or without them (right). In the case FPP-1, C has a higher priority than D and the other way round in case FPP-2. Under RR the parameters Ψ_k of task C and D are chosen in order to obtain bounds smaller than the deadlines.

The Round Robin policy can be seen as being between the two extreme cases FPP-1 and FPP-2. It appears that the averages of the bounds in the two FPP cases are smaller than under RR, although one would expect them to be similar. We believe that the timing analysis of RR can still be improved by deriving an analysis from the exact equation of response times under RR, that we have mentioned in Section 5.1.

Layer	p	Ψ_k	τ_k	C_k	T_k	D_k	z_k	bound	ρ_k	$\rho_{1..m}$
FPP	7		F	3	15	6		3	20.0	20.0
	6		G	3	15	7		6	20.0	40.0
	5		A	7	50	50		13	14.0	54.0
	4		B	6	50	50		25	12.0	66.0
RR	3	5	C	10	100	150	6	120	10.0	76.0
	3	4	D	40	500	700		277	8.0	84.0
FPP	2		E	20	500	500	15	282	4.0	88.0

Table 5: Application set of tasks and timing analysis results (time unit : ms)

Task	FPP-1	FPP-2	RR
D	277	133	277
C	87	195	120

Task	FPP-1	FPP-2	RR
D	190	133	190
C	41	149	74

Table 6: Bounds for other priority assignments, with semaphores (left) and without (right).

8 Conclusion

Posix 1003.1b, [7] on page xvii, defines real-time in operating systems as

> "the ability of the operating system to provide a required level of service in a bounded response time".

This requirement of *predictable* response also applies to real-time in applications. In this paper, we have given elements of a theoretical framework (work arrival functions, formulas for response times) that enable the timing analysis of a wide class of scheduling algorithms. This class includes *hybrid* policies within which multiple scheduling policies may coexist. In particular, the combination of fixed preemptive priorities and the Round Robin policy has been studied here. An extensive account of the other applications of this framework can be found in [8]. Future work in this direction will consist in analyzing combinations of Round Robin or fixed priority policies with the *earliest deadline first* scheduling policy.

References

[1] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20(1):40–61, February 73.

[2] J. Lehozky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[3] M. Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[4] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, December 1990.

[5] K. Tindell, A. Burns, and A.J. Wellings. An extendible approach for analysing fixed priority hard real time sytems. *Real-Time Systems*, 6(2), 1994.

[6] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[7] (ISO/IEC). *9945-1:1996 (ISO/IEC)[IEEE/ANSI Std 1003.1 1996 Edition] Information Technology - Portable Operating System Interface (POSIX) - Part 1 : System Application : Program Interface*. IEEE Standards Press, 1996. ISBN 1-55937-573-6.

[8] J. Migge. *Scheduling of recurrent tasks on one processor: A trajectory based Model*. PhD thesis, Université Nice Sophia-Antipolis, 1999. `http://www.migge.net/jorn/thesis/`.

[9] A. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, oktober 1997.

[10] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report 2966, INRIA, september 1996.

[11] P. Pushner. Worst-case execution-time analysis at low cost. *Control Eng. Practice*, 6:129–135, 1998.

[12] C.Y. Park. Predicting program execution times by analyzing static and dynamic programs paths. *Real-Time Systems*, 5(1):31–62, 1993.

[13] K.W. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, December 1993.

[14] B.O. Gallmeister. *Programming for the Real World - Posix 4*. O'Reilly & Associates, 1995. ISBN 1-56592-074-0.

[15] J. Migge. Tkrts: A tool for computing response time bounds with a trajectory based model., 1998. `http://www.migge.net/jorn/rts/`.

[16] M. Chen and K. Lin. Dynamic priority ceiling: A concurrency control protocol for real-time systems. *RTS*, 2, 1990.

[17] N. Navet and J.M. Migge. Fine tuning the scheduling of tasks on posix1003.1b compliant systems. Research Report 3730, INRIA, 1999. ftp://ftp.inria.fr/INRIA/publication/RR/RR-3730.ps.gz.

[18] D.R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

A Fixed point equations and response time bounds

It has been shown above that response times can be expressed as particular solutions of fixed point equations. These equations generally have more than one fixed point such that in their neighborhood, the involved function is not necessarily a contraction. This partially explains why response times appear as “first fixed point after some point”, see (30).

Let $x_0 \in \mathbb{R}$ and f be a function $\mathbb{R} \longmapsto \mathbb{R}$ with $f(x_0) > 0$ such that

$$x_1 = \min\{x > x_0 \mid f(x) = x - x_0\} \tag{30}$$

exists. If f can be bounded by another function $\hat{f}$ then the first fixed point either remains unchanged or is shifted towards the future, seen relative to their respective reference points x_0 and 0:

Proposition 1
Let $\widehat{f}$ be a function $\mathbb{R} \longmapsto \mathbb{R}$ with $\widehat{f}(\hat{x}_0) > 0$ such that

$$\widehat{x}_1 = \min\{x > 0 \mid \widehat{f}(x) = x\} \tag{31}$$

exists and

$$\forall\, x \in [x_0, x_1) \quad f(x) \leqslant \widehat{f}(x - x_0). \tag{32}$$

Then $x_1 - x_0 \leqslant \hat{x}_1$

Proof: Equation (30) and (32) imply

$$\forall\, x \in [x_0, x_1) \quad x - x_0 < f(x) \leqslant \widehat{f}(x - x_0)$$

and thus $\widehat{x}_1 \notin [0, x_1 - x_0)$. ■

This is typically used when deriving response time bounds with $x_0 = U_{k,n}$ being the beginning of the interference period containing the instance under study and the beginning $u = 0$ of the first interference period of the majorizing release pattern: $E_{k,n} - U_{k,n} \leqslant r_{k,\tilde{n}}$.

B Notations

$A_{k,n}$	activation or release time of $\tau_{k,n}$
b_k	maximum length of critical sections of lower priority task that can lock the same semaphores as τ_k.
$a_{k,n}, c_{k,n}$	activation and execution times in the worst case activation pattern
$C_{k,n}$	execution time of $\tau_{k,n}$
$E_{k,n}$	execution end of $\tau_{k,n}$
λ_l	layer = subset of tasks $\tau_{m_{l-1}}, \tau_{m_{l-1}+1}, \tau_{m_{l-1}+2}, \ldots, \tau_{m_l}$
m_l	index of the last task in λ_l
Ψ_k	time quantum for τ_k under Round Robin
Ψ^l	sum of the quanta of the tasks in layer λ_l.
$\overline{\Psi}_k$	sum of the quanta of the task in layer λ_l except that of τ_k.
$S_i(t_1, t_2)$	work due to instances of τ_i activated in $[t_1, t_2)$: work arrival function (WAF)
$S_{1..i}(t_1, t_2)$	work due to instances of $\tau_1, \ldots, \tau_i$ activated in $[t_1, t_2)$
S_k^n	work due to instances of τ_k in $[U_{k,n}, A_{k,n})$.
$s_i(x)$	work due to instances of τ_i activated in $[0, x)$ of the worst case pattern: (MWAF)
$s_{1..i}(x)$	work due to instances of $\tau_1, \ldots, \tau_i$ activated in $[0, x)$ of the worst case pattern
$\underline{s}_k^n$	bound for S_k^n derived from $s_k(x)$.
$\overline{S}_k(t_1, t_2)$	work due to tasks of τ_k's layer other than τ_k, activated in $[t_1, t_2)$
$T_{k,n}$	inter-arrival time $T_{k,n+1} - T_{k,n}$
$\mathcal{T}$	task set
τ_k	task number k
$\tau_{k,n}$	n^{th} instance or release of τ_k
$U_{k,n}$	beginning of the level-k interference period containing $A_{k,n}$
U^k	beginning of a level-k interference
U_k^{RR}	beginning of a τ_k-RR-interference period
$V_{k,n}$	end of the level-k interference period containing $A_{k,n}$
V^k	end of a level-k interference
V_k^{RR}	end of a τ_k-RR-interference period
$W_k(t)$	pending work at t, due to instances of τ_k activated strictly before t.
$W_{1..m_l}(t)$	pending work at t, due to instances of $\tau_1, \tau_2 \ldots, \tau_{m_l}$ activated strictly before t.
$\overline{W}_k(t)$	pending work at t, due to instances of the task in a RR-layer except that of τ_k.
z_k	longest critical section of τ_k

C Acronyms

FPP	Fixed Preemptive Priorities
LPP	Layered Preemptive Priorities
RR	Round Robin
WAF	Work Arrival Function
MWAF	Majorizing Work Arrival Function
POSIX	Portable Operating System Interface
WCET	Worst Case Execution Time