On Hierarchical Scheduling in VxWorks

by

Mikael Åsberg (mag04002@student.mdh.se)

Master thesis
Supervisor: Moris Behnam
Examiner: Thomas Nolte

Mälardalen University
Department of Computer Science and Electronics
June 5, 2008
Västerås

Abstract

Theoretically, hierarchical scheduling has many advantages when it comes to integrating real-time applications on a single CPU. Even though each subsystem (real-time application) will get less resource in the form of CPU (as they are required to share the CPU), they will not be forced to have their properties changed (periods, priorities, etc.) during the integration and their scheduling policy is allowed to remain the same.
Each subsystem will still be isolated as if it would be executing in isolation and the error of one subsystem can not propagate and affect other subsystems.
A subsystem can be viewed as a virtual task which in turn have it's own properties. Virtual tasks will be scheduled by a global scheduler and virtual tasks execution includes the scheduling of it's local tasks (by the virtual tasks local scheduler) as well as the local task execution [18].

The cost of all the explained advantages is the overhead of the global scheduler. In facilitating the above mentioned benefits a more complex scheduler has been developed. A hierarchical scheduling framework (HSF) will only be of use and interest if it is time efficient (low overhead) and if it is possible to implement it on top of already existing industrial systems.

This thesis address these issues and specifically exploit the possibilities of designing and implementing a HSF on top of one of the most commonly used commercial industry real-time operating system, namely Vxworks.
The focus during the development of the HSF has been to find, evaluate and choose the most time efficient design solutions in order to impose as little system overhead as possible.

The work can be separated into two sections.

First we designed and implemented two extensions to the native vxworks priority preemptive scheduler. The first extension was to implement a module that had support to schedule tasks periodically. This module has an absolute time based multiplexed software timer event queue (TEQ). The second extension was an Earliest Deadline First (EDF) implementation. These two extensions together with the native vxworks scheduler form two new local schedulers, a Fixed Priority (FPS) periodic task scheduler and an EDF periodic task scheduler.

When the local schedulers were implemented, we moved on and designed and implemented two global schedulers (FPS and EDF). The HSF uses these two global schedulers (one of them) and schedules servers according to the periodic server model. Each subsystem is then internally scheduled by either our local FPS or EDF scheduler.
Server execution time (budget) and period, as well as tasks periods and deadlines, are handled by the TEQ. Each local system will each have their own TEQ and the servers will have one common TEQ.

Detailed scheduling overhead analysis for the local and global FPS scheduler was derived using the Response Time Analysis (RTA). Moreover, we did extensive time measurements of the local schedulers, the global schedulers and the HSF itself. Execution traces from tests are graphically shown using execution recording of tasks.

Acknowledgments

I want to thank Johan Kraft for his support and help with VxWorks.

Contents

1 Introduction

1.1 Introduction

Hard real-time applications have strict timing properties which most often are mathematically analyzed, based on the system properties, and thereby known to be schedulable or not. The schedulability of a system is strict dependant on the system properties and these should not be altered in order for the system to remain schedulable.

However, integrating different real-time systems on the same computer is positive due to a potential cost reduction inherent in lowering the amount of hardware needed.
The execution of different real-time applications in one system have a great impact on the timing requirements of each application. These application's timing requirements are dependant on the amount of resources (for example CPU) available for the application the be schedulable. This makes it difficult to share resources among real-time systems while at the same time preserving temporal behavior [19]. It might be very difficult, time consuming or in the worst case impossible to adopt real-time software to new environments where it must share resources with other applications. Merging several real-time systems together, sharing a CPU, is a big challenge. All systems must have their properties re-configured and re-analyzed to be able to execute together. The re-configurations of the systems might be very complex. Moreover, there might be other downsides as well, inherent in that all systems will affect all other systems. We will not have any isolation between applications which means that an unexpected error in one application might affect other applications [18].

We see that it is a big challenge to let real-time applications share resources with eachother. A solution to this problem is to introduce hierarchical scheduling [19].

1.2 Related work

We did not (during the 6 months of this project) find any hierarchical scheduling implementations on top of VxWorks operating system.

VxWorks have developed a special operating system version 653 which provides extra safety and security properties for mission critical applications. A global scheduler (in kernel space) schedules partitions in a predefined time basis. Each partition schedules tasks in a priority-preemptive fashion using the VxWorks, POSIX or ARINC API. [16]

In [15] the authors present the design of a HSF for which support the scheduling of systems at an arbitrary amount of levels. The implementation is done in RED-Linux, which is a modified linux kernel with support for real-time scheduling. The scheduling framework in RED-Linux consists of a *Dispatcher* and *Allocator*. The first mentioned component is in charge of choosing a job among jobs for execution. The second component translates a scheduling policy to specific parameters for each job in the group of jobs. The authors managed to implement so that each group had it's own scheduling policy and that jobs were either just jobs or a group of jobs. Note that in this implementation, the authors did not modify the kernel but extended the existing scheduler.

[21] describes an algorithm to handle time representation in conjunction with an EDF implementation. The technique used is the circular time model in which the time is absolute and at some point overflows.

An hierarchical implementation is presented in [22] where the target operating system is Linux/RK, an extended linux version with resource kernel functionality. Linux/RK supports partitioned reserves where the budgets are replenished at every period. The authors extend the hierarchical implementation by supporting multilevel hierarchy where each reserve has a list of child reserves.

In [17], the authors design and implement a two level HSF in the real-time kernel pSOS. A virtual task, defined as RCE (Resource Consuming Entity), was designed to have a group of tasks, a budget (execution time) and replenishment period. At the top level, scheduling was done with the Rate Monotonic (RM) scheduling policy. Each subsystem was scheduled according to priority based preemptive scheduling (a task has a priority).
A RCE was scheduled periodically with execution time equal to it's budget. A special timer component was developed that multiplexed several software timers against one or more hardware timers. Budgets were set with a so called one-shot timer and RCE periods with periodic timers. Each timer (periodic or one-shot) was sorted in a queue, the lowest value in this queue was set as expiration on the hardware timer. The timer values in the queue were relative so at each expiration, the first timer value would be reset and all the others decremented with the elapsed time.
When the global scheduler executed a RCE, all tasks in that RCE would get their priority raised while the tasks in the RCE before would be set to low priority values.
The execution time of the global scheduler was measured and subtracted from the next executing RCE's budget. This would decrease the execution time of RCE's but eliminate drift caused by the global scheduler.
Scheduling at the bottom layer was not explained in detail but it is assumed that the existing OS scheduler takes care of this, based on the task priorities and perhaps also their periods.

Our design and implementation of the HSF in VxWorks is similar to the work presented in [17], although their framework is based on scheduling soft tasks while our system is suited for hard tasks. Another difference is that we also implemented local task schedulers (FPS and EDF) which is extensions to the native VxWorks task scheduler. Also, our TEQ component and server taskset switching strategy differs (ours are more time efficient). Reducing drift by subtracting server budgets is a strategy we also used, this is of course included in the scheduling overhead formulas that

we have derived.

The preliminary results of this thesis was published in [24] and was accepted to the OSPERT 2008 conference in Prague, Czech Republic.

1.3 Thesis goal

The aim of this thesis is to investigate and explore the possibilities and limitations to design and implement hierarchical scheduling in the commercial real-time operating system VxWorks.
The chosen design must by all means be as efficient as possible (cause minimum overhead) and not cause extensive drift. The advantages of hierarchical scheduling is not worth anything if the implementation causes to much overhead. That is why this thesis has the strong requirement of investigating and choosing the most time efficient strategies for the design of this system.

Because hard tasks are meant to be scheduled by this HSF, the chosen design must have a relating mathematical analysis that express the overhead caused by our system.

VxWorks, as many other RTOS, lacks the property of periodic scheduling of tasks. This aspect must be thought of in the design.

The report is organized as follows:

Chapter 2
Theoretical background is given to real-time systems in general, how hierarchical scheduling works, introduction to the RTOS VxWorks and theories about scheduling analysis for both local systems as well as hierarchical systems.

Chapter 3
We first explain the design of the local schedulers and then the HSF. The design of the local scheduler has great impact on the design of the HSF.

Chapter 4
The results of both the local scheduling systems as well as the hierarchical system is presented. Time measurements of the schedulers and comparison of the scheduling analysis against execution measurements are shown. In many of the tests we record task executions with a special tool and then show the recorded traces graphically.

Chapter 5
We discuss and conclude our work and give suggestions to improvements of our system.

Appendix A
Time measurement graphs for both the local and global schedulers are shown.

Appendix B
Pseudocode for a function related to local scheduling analysis is presented.

Appendix C
Screenshots from task execution traces are shown.

2 Theoretical background

2.1 Real time systems

A real-time system is a system that does not only operate correctly, but also does it within a specified time. A key feature in these kind of systems is that operations are executed before their deadlines.

High level timing requirements for a real-time system can be re-defined as timing properties of system tasks, which are sequential programs that perform an assignment. Tasks run together according to some schedule.
Scheduling of tasks is important because it affects when a task start and end it's execution. Scheduling algorithms can be categorized as follows. [1]

Preemptive or non-preemptive
Preemptive scheduling is when tasks interrupt eachother during execution. Tasks always completes their execution without being interrupted by other tasks if the scheduling is non-preemptive.

Static or dynamic scheduling
Static scheduling means that scheduling decisions are based on fixed parameters, it is dynamic when the parameters change during system execution.

Off-line or on-line scheduling
Off-line scheduling means that is it is based on a pre-defined time table, where each task has a defined start and end time. On-line is when scheduling decisions are made during runtime based on task parameters.

Off-line scheduling schedules according to a pre-defined task schedule (*Figure 1*).

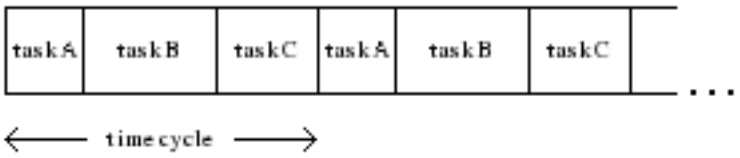

Figure 1: Off-line scheduling

Because real-time systems have time requirements, tasks typically have a time before it should end it's execution called *deadline*. Tasks can be classified based on the importantness of finishing it's execution before it's deadline.

Hard and soft deadlines
A hard deadline means that if a task misses it's deadline then the consequences could be catastrophic. Missing a soft deadline can at most just decrease system performance.

Tasks can be characterized by the following parameters.

Period (T)
The time interval in which the task is to be executed.

Relative deadline (D)
The time relative to the start time of the period in which the task must finish it's execution.

Worst case execution time (C)
The execution time in one period.

Release time (RT)
The time relative to the start of the period in which the task starts executing.

Priority (P)
The priority level of the task, the task with the highest priority of all tasks is scheduled to execute.

The above parameters are illustrated in *Figure 2*, where T=5, D=3, C=2 and RT=0 for a task.

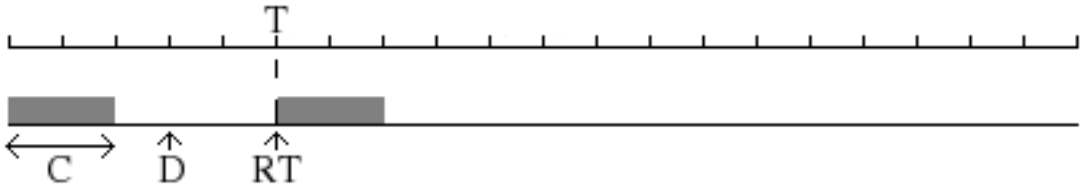

Figure 2: Task parameters

There exist three kinds of tasks, periodic, sporadic and aperiodic tasks. The first mentioned executes periodically according to the period parameter. The second has a minimum inter-arrival time and a hard deadline where as the third one arrives at irregular intervals and may have hard or soft deadlines.
Aperiodic tasks can be handled in the scheduling by introducing a special task called server. A server is essentially a periodic task that services aperiodic requests, the server execution time is normally referred to as the budget. Servers are normally used to service aperiodic tasks but it is of course possible to service periodic or sporadic tasks as well.
[2]

2.2 VxWorks task scheduling

2.2.1 Native and POSIX scheduler

VxWorks can be configured with either the Native VxWorks scheduler or the POSIX scheduler. These can in turn be customized by the developer. [10]
Both VxWorks schedulers can schedule tasks and pthreads in a preemptive priority based fashion or according to the round robin policy. In the first mentioned case, the highest priority task or pthread in the ready queue is always allocated for the CPU. With round robin, each task with the same priority executes according to it's time slice in round robin fashion. Higher priority tasks may preempt another task during it's time slice. It will resume and execute the rest of it's time slice after the high priority task is removed from the ready queue.

Tasks and pthreads are actually threads but they are scheduled globally according to their priority no matter if they belong to a process or execute in kernel mode.
VxWorks 6.0 introduced the notion of Real-Time Processes (RTP), before that all user tasks executed in kernel mode.
Tasks and pthreads are similar in that they are scheduled in the same way by the Native VxWorks scheduler, pthreads have more scheduling options if they run in a real time process (RTP) and the POSIX scheduler is used.
Tasks and pthreads that are created from within a RTP share the same virtual memory. ISR:s, kernel memory space and direct hardware access (among other) are prohibited. Tasks and pthreads that are created outside of a RTP execute in kernel mode, these tasks are allowed in VxWorks version 6.0 and forward.
Pthreads in RTP's can only be scheduled by the POSIX scheduler and vice versa pthreads can not be created in RTP:s if VxWorks is not configured with the POSIX scheduler. In all other cases, the Native and POSIX scheduler can schedule pthreads and tasks (in user or kernel mode) in preemptive priority-based fashion or round robin (*Figure 3*).

Execution enviroment	POSIX thread scheduler		Native vxworks scheduler	
	Tasks	Pthreads	Tasks	Pthreads
Kernel	Priority-based preemptive, or round robin scheduling	Priority-based preemptive, or round robin scheduling	Priority-based preemptive, or round robin scheduling	Priority-based preemptive, or round robin scheduling
Process	Priority-based preemptive, or round robin scheduling	POSIX FIFO, round robin sporadic, or other	Priority-based preemptive, or round robin scheduling	Not possible

Figure 3: VxWorks scheduler properties

All pthreads and tasks in a VxWorks system are scheduled according to the default scheduling policy except pthreads running in RTP's. These may have other scheduling policies than the rest of the system, i.e. round robin, preemptive priority-based or sporadic scheduling. Sporadic scheduling is best fitted for aperiodic tasks. The pthread is scheduled periodically at a high priority at specific time intervals, and at a low priority the rest of the time.
The POSIX scheduler has it's own round robin and preemptive priority-based scheduling, these are however similar to the Native VxWorks scheduler. Note that, in VxWorks versions before 6.0, neither POSIX nor the Native VxWorks scheduler can schedule tasks periodically.

2.2.2 Custom scheduler

In order to implement a HSF, the native VxWorks or POSIX scheduler must be used, but with extended functionality. The goal is to develop a middleware (hierarchical extension) between the actual VxWorks scheduler and the applications. This middleware must execute in kernel mode in order to manipulate scheduler data structures, execute code in interrupt level and have access to hardware. Having a middleware is a modular and general solution considering that it will be able to run on any VxWorks version.
The VxWorks scheduling framework is flexible so that it easily can be customized. The customization is actually to extend the Native VxWorks or POSIX scheduler with either a customized ready queue structure or to define a interrupt handler that is executed at every clock tick. [10]
The Native VxWorks or POSIX scheduler allocates the highest priority task in the ready queue to the CPU. Having a custom interrupt routine running before the scheduler at every instant in time when the ready queue must be re-ordered is actually enough. The ready queue can be manipulated by resuming or suspending tasks, priorities can be changed etc. with system calls (taskLib).
The frequency of the clock tick interrupts can be set and read, this notion of time together with task system calls are good tools for customizing the Native VxWorks or POSIX scheduler.

2.2.3 Interrupts and VxWorks scheduler

The VxWorks scheduler is always invoked after the execution of an interrupt handler no matter which kind of interrupt. The interrupt handler is actually a program function that is connected to a specific interrupt vector in a interrupt vector table. An exception is that some CPU and board support packages (BSP) only have one interrupt vector, so all interrupts are connected to the same program function.
The program function is wrapped in between interrupt initialization code and interrupt exit code. The first part does the context switch (switch from task execution to interrupt handler execution) and saves the context among other operations which means that CPU registers are manipulated. After the interrupt handler has executed, the last part invokes the scheduler if necessary or otherwise restores the saved context to the CPU. If the interrupt handler has not re-ordered the ready queue then the scheduler will not be invoked, the interrupted tasks data will be restored in CPU registers.
The exit code will restore the interrupted interrupt handler if interrupts are nested. If the scheduler has been locked by the interrupted task then the scheduler will not be invoked.
A notation is that troughout this paper, we will define a context switch (CS) as the switch from a task to an interrupt handler or opposite. An interesting fact is that VxWorks context switches are known to have very low overhead.
[11]
Figure 4 below illustrates how the interrupt handler is wrapped.

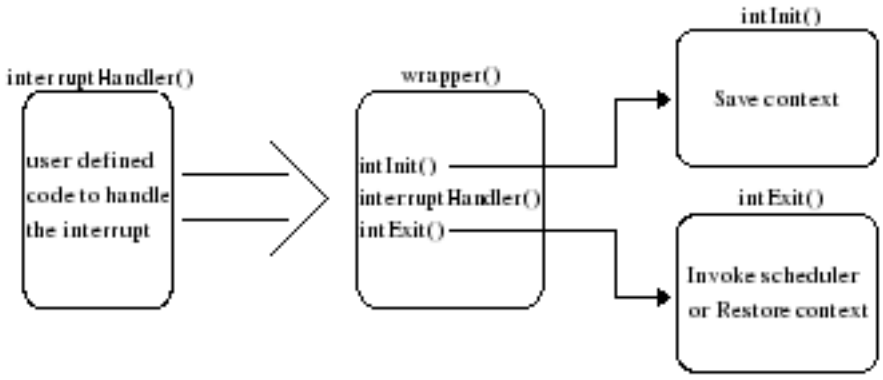

Figure 4: Interrupt handler wrapping

2.2.4 Tasks in VxWorks

A task in VxWorks can be created either in conjunction with the native or POSIX scheduler. No matter which scheduler is used, VxWorks will allocate a task control block (TCB) for each task. All relating task data is saved in this structure, for example priority, program counter, stack pointer etc.
VxWorks tasks can have the states shown in *Figure 5* below or combinations of them. The status flag in the TCB holds information about the task state.
A task in **suspend** state is unavailable for execution but not delayed or suspended, in **pended** state the task is blocked waiting for some resource other than the CPU and in **delayed** state it is asleep for some time. If a task is in the **ready** state then it is ready to execute. Tasks that are ready are sorted in the VxWorks ready queue based on task priority. Each node in the ready queue has a reference to the corresponding tasks TCB. The first task in the ready queue (highest priority) is allocated the CPU. [10]

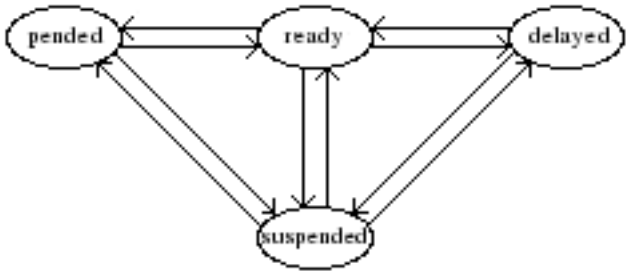

Figure 5: Task transitions

2.3 Scheduling analysis

In this section, we will present scheduling analysis for local and hierarchical systems.

2.3.1 Fixed priority scheduling analysis

Response time analysis (RTA) is a method for analysing the schedulability of each task in a set of fixed priority tasks. [20]
The response time (R) of a task is the point in time, relative to the time when it is released, when it has finished it's execution. The time when it is released is the worst case scenario (critical instant), that is when all tasks are released at the same time. The response time of a task is it's execution time added together with the execution time of all higher priority tasks times the amount of instances they execute during the whole response time.
The formula is iterative, starting with the task execution time. The response time will increase after an iteration, which may collide with new period instances from higher priority tasks, this will again increase the response time and so on.
Tasks properties such as execution time, period, deadline and priority are input to RTA which then determines if the response time is less than a tasks deadline. If a tasks response time is smaller than it's deadline then it is schedulable. If this is true for all tasks in a task set then the whole system is schedulable.
To sum it up, all tasks in a system must have the sum of their execution time and interference time smaller than their deadline.

$$\forall i \in \{1, 2, .., n\}, R_i < D_i$$

where n is the number of tasks

The RTA formula has the following appearance.

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j \tag{1}$$

Where n is the iteration number, i and j are the task numbers, R is the response time, T is the period, C is the execution time and $hp(i)$ is the set of tasks that have higher priority than task τ_i.

The formula start value is: $R_i^0 = C_i$. The iteration stops either when
$R_i^{n+1} > D_i$ or when $R_i^{n+1} = R_i^n$.

This is how the formula can be interpretted. The start is that R_i^0 is equal to task τ_i:s execution time, then this value is brought to the next iteration. $\frac{R_i^n}{T_j}$ means that we derive the amount of executions of other tasks during the execution time of task τ_i. When all tasks τ_j have preempted task τ_i then we have a new value for our R_i^0, which now is R_i^1.

The calculation of task τ_2's response time is as follows, the task set is shown in *Table 1.*

Name	*Period*	*Executiontime*	*Deadline*	*Priority* ($4 = highest$)
τ_1	15	1	15	1
τ_2	13	3	13	2
τ_3	5	1	5	3
τ_4	4	2	4	4

Table 1: Task set

$$R_2^0 = 3$$

$$R_2^1 = 3 + \left(\left\lceil \frac{3}{5} \right\rceil \cdot 1\right) + \left(\left\lceil \frac{3}{4} \right\rceil \cdot 2\right) = 6$$

$$R_2^2 = 3 + \left(\left\lceil \frac{6}{5} \right\rceil \cdot 1\right) + \left(\left\lceil \frac{6}{4} \right\rceil \cdot 2\right) = 9$$

$$R_2^3 = 3 + \left(\left\lceil \frac{9}{5} \right\rceil \cdot 1\right) + \left(\left\lceil \frac{9}{4} \right\rceil \cdot 2\right) = 11$$

$$R_2^4 = 3 + \left(\left\lceil \frac{11}{5} \right\rceil \cdot 1\right) + \left(\left\lceil \frac{11}{4} \right\rceil \cdot 2\right) = 12$$

$$R_2^5 = 3 + \left(\left\lceil \frac{12}{5} \right\rceil \cdot 1\right) + \left(\left\lceil \frac{12}{4} \right\rceil \cdot 2\right) = 12$$

The calculation is illustrated in *Figure 6.*

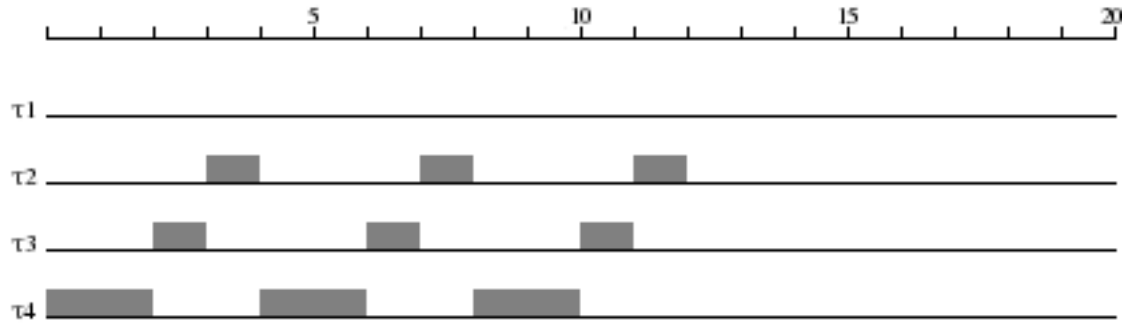

Figure 6: τ_2's response time

2.3.2 EDF scheduling analysis

When analysing task schedulability for a task system scheduled under EDF, the following formula can be used. [1]

$$\forall\ 0\ <\ t\ \leq\ LCM_{tasks} : \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i\ \leq\ t \tag{2}$$

where LCM_{tasks} is defined as the least common multiplier of all tasks in the system.

In the case when the period is not equal to the deadline, the last instance will not require an entire period for the execution to complete. This is shown in the figure (*Figure 7*).

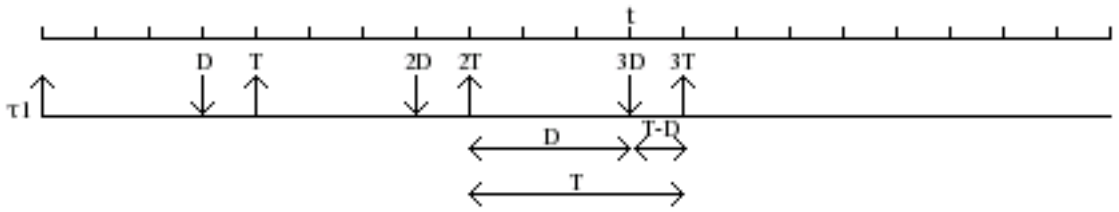

Figure 7: Period instances when $T\ \neq\ D$

The task will finish it's last execution instance before the deadline, not the period. That is why the expression T_i - D_i is added to the numerator, so it is accounted for.

The equation (2) can be used for tasks that have deadline less or equal to it's period.
Every tasks execution demand during a certain time period t are added together. If they request more execution time than available (t), then they will not be schedulable.

Table 2 below shows a task set which is not schedulable according to the formula (2).

Name	*Period*	*Executiontime*	*Deadline*
τ_1	3	1	3
τ_2	5	1	5
τ_3	2	1	2

Table 2: Non-schedulable task set

$$\left(\left(\left\lfloor \frac{30+3-3}{3} \right\rfloor 1\right)\ +\ \left(\left\lfloor \frac{30+5-5}{5} \right\rfloor 1\right)\ +\ \left(\left\lfloor \frac{30+2-2}{2} \right\rfloor 1\right)\right)\ \leq\ 30$$

$$(10\ +\ 6\ +\ 15)\ \leq\ 30$$

$$31\ \leq\ 30$$

Figure 8 shows the execution scenario. At time 28, all tasks request 3 time units when there is only 2 time units left before their deadlines expire at time 30.

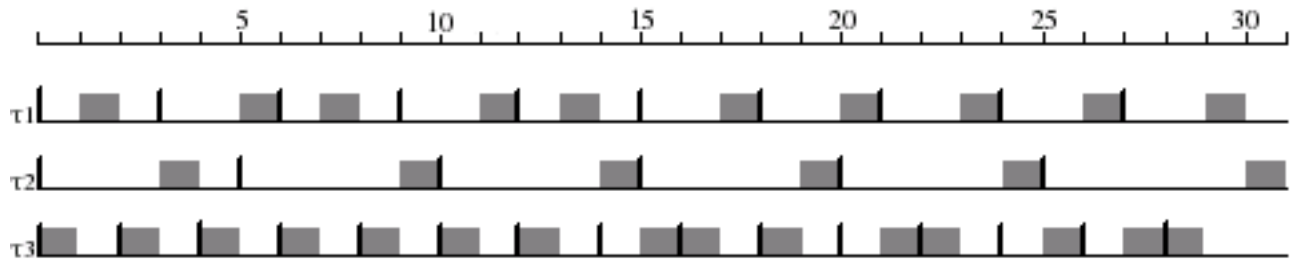

Figure 8: EDF scheduling example

2.4 Hierarchical scheduling framework

2.4.1 Hierarchical scheduling

Hierarchical scheduling can be seen as a number of separate schedulers scheduling nodes in a tree like structure (*Figure 9*). A node can represent either a task or a scheduler (which in turn schedules it's nodes). The tree may have an arbitrary number of levels and each node may have an arbitrary number of children. [19]

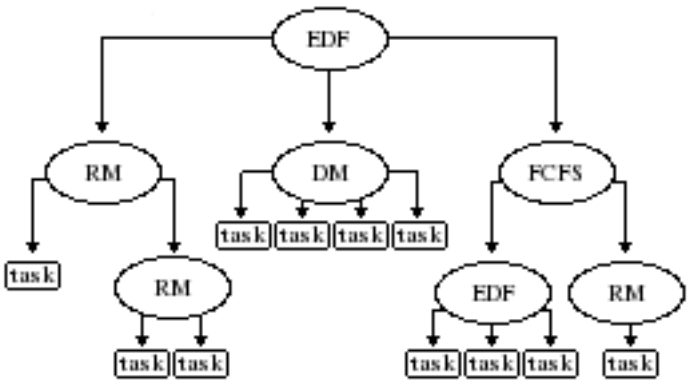

Figure 9: Hierarchical scheduling tree

The remainder of this chapter will describe various ways of constructing two-level hierarchical scheduling systems, since this thesis implements such a scheduler. Some constructs can of course be applied to multi-level hierarchical systems, since two-level systems are just special cases of multi-level systems.

A common way of constructing a two-level system is by having a global on-line scheduler that schedules servers (*Figure 10*). These in turn consists of an application with a local scheduler that schedules a group of tasks. The global scheduler may be either Dynamic Priority Scheduling (DPS) [8] or FPS [9].
Depending on the server algorithm that is used, the server parameters may be either static or dynamic.
The server period and budget form the CPU utilization of the server, the sum of all server utilization may of course not exceed 100 % on a single CPU.

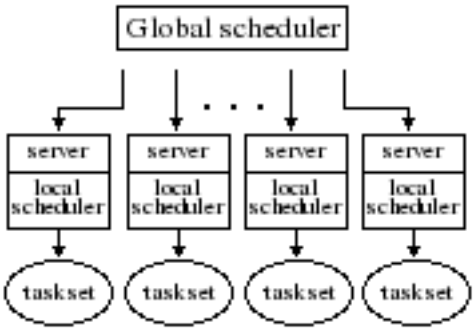

Figure 10: Two level scheduling system

Another way of constructing a two-level system is to use off-line global scheduling. Examples of such systems can be found in [4], [7] and [6].
Off-line scheduling can of course be realized with an on-line scheduler, but then the server must be of type that does not have dynamic parameters.

No matter what kind of global scheduler that is used, the local schedulers may be DPS or FPS. Local schedulers can also be of type non-preemptable. An example of a non-preemptable local scheduler can be found in [5], where EDF and a re-designed CBS [1] is used in the global level and FCFS (first come first serve) is used as a local scheduler.

Not all two-level hierarchical systems fall in the categories mentioned so far. Most often, the global scheduler use the server parameters when doing scheduling decisions. This is however not true for all two-level systems. For example, in [3], a two-level hierarchical scheduling system is presented which do not incorporate any notion of servers.
The global scheduler is a FPS scheduler which schedules applications based on their fixed priority levels and the task priorities. Each application has either a FPS scheduler or an EDF scheduler and a priority level.
The scheduling algorithm changes depending on the local application scheduler. When an application is introduced to the system, the tasks may be split up into different system applications.

2.4.2 Hierarchical scheduling analysis

A server's resource is defined as how much budget it executes each period. When analysing the schedulability of a hierarchical system, two aspects are looked upon. First, the sum of all server resource's may not exceed the available system resource. Second, a task set resource demand may not exceed it's servers resource. This must be true for all task sets and their servers.
If these two aspects are complete then the hierarchical system is schedulable.

There exists different ways to perform hierarchical scheduling analysis. The authors in [23] present a approach to perform analysis of FPS at both global and local level. The analysis assumes that all server parameters are known, the schedulability is therefore less pessimistic because the execution trace of the whole server system is known. However, this approach is not suitable for systems that are developed independently.

A more simple approach can be found in [14]. Server parameters are not considered, so analysis is not dependant on knowledge of other servers in the system. The results are more pessimistic than the other approach, but it is easier to develop subsystems independently.
We have chosen to express our scheduling overhead together with the analysis in [14]. However, it is easy to extend our analysis to make it suitable for the approach in [23].

The following describes definitions of a hierarchical model. [14]

A set of servers in a hierarchical system is defined as

$$S$$

A server S_i, where $S_i \in S$, is described as

$$S_i(W_i, \Gamma_i, A_i)$$

where W_i is a task set, Γ_i is the server resource and A_i is the local scheduling algorithm

A task set W_i is further described as

$$W_i(\tau_1, \tau_2, \tau_3, ..., \tau_n)$$

where n is the number of tasks in S_i

Each server has a resource defined as

$$\Gamma_i(\Pi_i, \Theta_i)$$

where Π_i is the server period and Θ_i is the server budget

First, let's look at how to analyze wether a task set W_i is schedulable according to it's servers resource Γ_i.
We first define the resource supply function sbf which, for a resource Γ, gives the minimum amount of resource as a function of the time interval t. The supply function is defined as

$$sbf_\Gamma(t) = \left\lfloor \frac{t - (\Pi - \Theta)}{\Pi} \right\rfloor \cdot \Theta + \epsilon_s \tag{3}$$

where ϵ_s is defined as

$$\epsilon_s = max\left(t - 2(\Pi - \Theta) - \Pi \left\lfloor \frac{t - (\Pi - \Theta)}{\Pi} \right\rfloor , 0 \right) \tag{4}$$

The minimum supply function is illustrated in *Figure 11.*

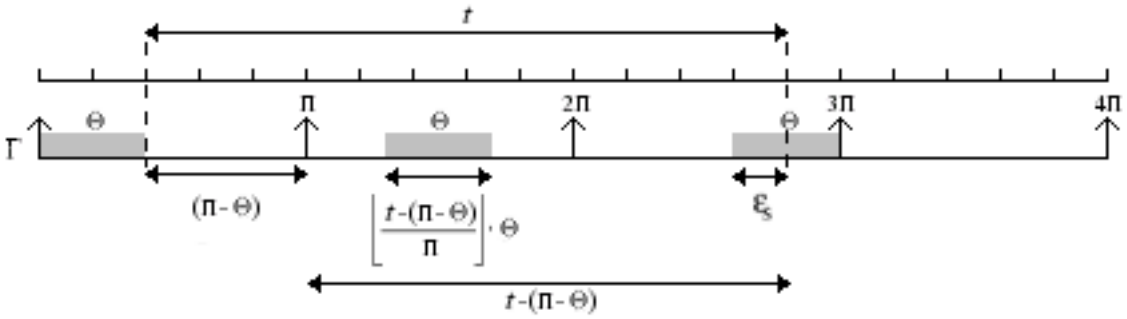

Figure 11: Minimum supply function

The part $(\Pi - \Theta)$ makes the formula assume that the interval t starts right after a budget instant which in turn starts at the same time as the corresponding period, no budget will occur in this interval so it is subtracted from the formula. The formula also assumes that the last budget instance starts as late as possible. The floor sign will not account for the fraction of the last budget, this fraction is calculated by the ϵ_s.
To sum it up, given any time interval t, the function will return the worst case minimum amount of resource.

The other way around is that given an amount of execution time, calculate the worst case (maximum) amount of time that it takes to supply that execution time. The service time function $tbf_\Gamma(t)$ does exactly this calculation.

$$tbf_\Gamma(t) \;=\; (\Pi \;-\; \Theta) \;+\; \Pi \cdot \left\lfloor \frac{t}{\Theta} \right\rfloor \;+\; \epsilon_t \tag{5}$$

where

$$\epsilon_t \;=\; \begin{cases} \Pi - \Theta + t - \Theta \cdot \lfloor \frac{t}{\Theta} \rfloor & \text{if } \left(t - \Theta \cdot \lfloor \frac{t}{\Theta} \rfloor\right) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The function $tbf_\Gamma(t)$ is illustrated in *Figure 12* below.

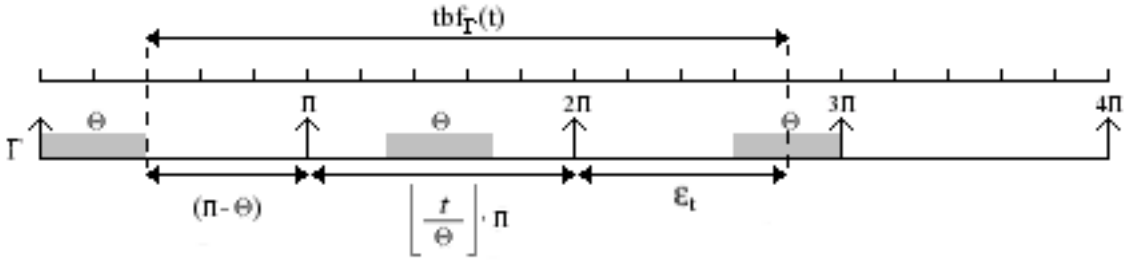

Figure 12: Service time function

$(\Pi - \Theta)$ assumes that we just missed a budget, $\lfloor \frac{t}{\Theta} \rfloor \cdot \Pi$ includes the number of periods to supply whole budgets and ϵ_t includes the last fraction budget if any.

The part ϵ_t is illustrated in *Figure 13*.

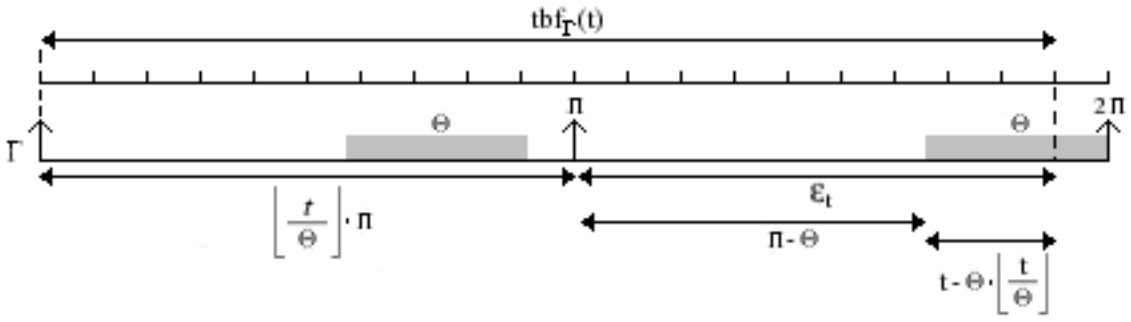

Figure 13: Illustration of ϵ_t

If we consider a local scheduler with EDF, that is a server $S_i(W_i, \Gamma_i, EDF)$, the task set W_i is schedulable with respect to the resource Γ_i if the following holds

$$\forall\ 0\ < t\ \leq\ LCM_{W_i}\ :\ \sum_{j\ \in\ W_i} \left\lfloor \frac{t}{T_j} \right\rfloor\ \cdot\ C_j\ \leq\ sbf_{\Gamma_i}(t) \tag{7}$$

where LCM_{W_i} is the least common multiplier (LCM) of all task periods

The equation states that the maximum (worst case) amount of execution time that the task set W_i demands during the time interval t must be less or equal to the minimum (worst case) amount of execution time that can be supplied by the resource Γ_i during that same time interval t. If this statement is true for $t = LCM_{W_i}$ then it is guaranteed that the task system W_i will always, no matter which time interval, have enough resources.
It is important to note that it is not sufficient to just look at the end values, that is, at time point equal to LCM_{W_i}. The demand curve must stay below the supply curve at all time instants until LCM_{W_i}, so all points where task period instants occur must be checked as illustrated in *Figure 14*.

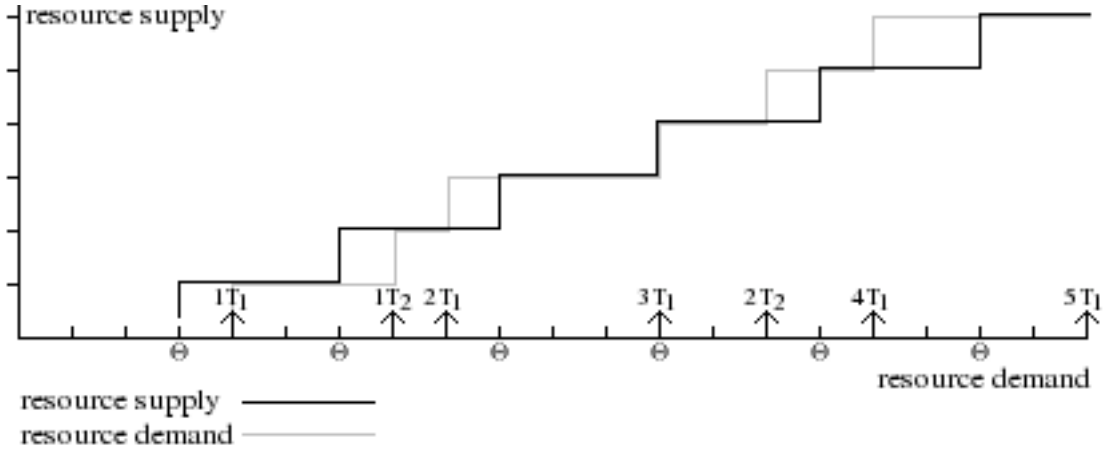

Figure 14: Demand and supply curve

Even though the resource supply may be larger than the demand in the end of the interval, there may be points in time where the opposite occurs. If all tasks period instances are checked within

the interval, then one can be sure that the entire interval is checked. The main reason why this checking is necessary is because the supply curve is not constant growing compared to when resource supply is 1, in this case the demand can never be more than supply during time instants before the end of t.

When looking at a local FPS system, the worst case response time of a task in such system $S_i(W_i, \Gamma_i, FPS)$ is calculated with the RTA formula

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i,W_k)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j \tag{8}$$

where $hp(i, W_k)$ denotes all higher priority tasks than τ_i in the task subset W_k

Because a periodic resource does not assume that an amount of execution time t will take an amount of time t to finish but maybe more, each iteration in the response time must be added some time (blackout duration) for which is the empty time between budgets.

$$r_i^{(k)}(\Gamma) \;=\; tbf_\Gamma(I_i^{(k)}) \tag{9}$$

where

$$I_i^{(k)} = C_i + \sum_{\forall j \in hp(i,W_k)} \left\lceil \frac{r_i^{(k-1)}(\Gamma)}{T_j} \right\rceil \cdot C_j \tag{10}$$

The task set W_i is schedulable with respect to it's servers resource supply Γ_i if the following holds

$$\forall\, j \;\in W_i : r_j^n \leq \; D_j \tag{11}$$

When all subsystems have a sufficient resource supply with respect to it's demand, these resource supplies must be added and checked so they do not exceed system utilization. The following must hold for the entire hierarchical system to be schedulable in the case of a global EDF scheduler

$$\forall\, 0 \;<\; t \;\leq\; LCM_S : \sum_{i=1}^{n} \left\lfloor \frac{t}{\Pi_i} \right\rfloor \cdot \Theta_i \;\leq\; t \tag{12}$$

where LCM_S is the LCM of all server periods and n is the number of servers

In order to analyze a hierarchical system with a global FPS scheduler, the RTA is used

$$R_i^{n+1} = \Theta_i + \sum_{\forall j \in hp(i,S)} \left\lceil \frac{R_i^n}{\Pi_j} \right\rceil \cdot \Theta_j \tag{13}$$

where $hp(i, S)$ is the set of servers for which has higher priority than server i in the system S

The servers in a global FPS system is schedulable if the following holds

$$\forall\, i \;\in\; S \; R_i^n \;\leq\; \Pi_i \tag{14}$$

3 Design & implementation

3.1 Custom periodic task scheduler

3.1.1 Motivation

As mentioned in the previous chapter, the VxWorks native scheduler does not have support for scheduling periodic tasks. The POSIX scheduler avaliable in version 6.0 and forward can schedule pthreads periodically [10]. Using this scheduling policy has a few drawbacks, it does not check deadlines and it has it's own priority mechanism (low and high priority depending on time intervals) together with fixed priority scheduling (FPS) only. The positive aspect is that one can specify which tasks that should have this scheduling policy, so a custom scheduling policy can reside together with this POSIX scheduling policy. This is important considering that a HSF should have support for several different local scheduling policies.
The problem of incorporating the POSIX periodic scheduling policy with a HSF is that old VxWorks systems (5.x) does not support it. Second, the POSIX scheduler is not flexible enough to fit as a local scheduler. Consider the scheduling timeline for servers in *Figure 15*.

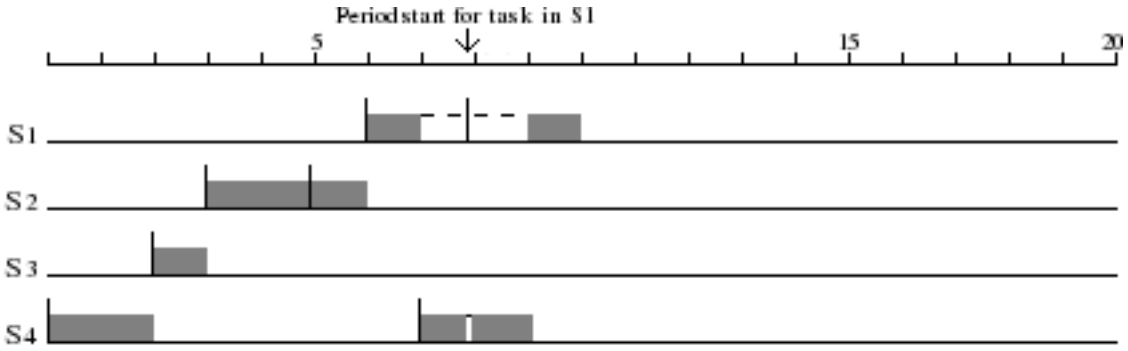

Figure 15: Server execution trace

Server S4 has the highest priority and a period of 7 time units, so server S1 (lowest priority) will be preempted at this time. If some task in server S1 is scheduled by the POSIX periodic scheduling policy, then an interrupt will occur at that tasks period start and insert the task in the ready queue. Thus, server S4 will be preempted and in worst case the task in S1 will execute if it has higher priority than the current running task in S4.
According to [12] (page 51) the POSIX periodic scheduler uses the system periodic timer for time accounting. Disabling the system timer interrupt might stop the scenario in *Figure 15* but other facilities that rely on the system clock such as watchdog timers and perhaps the hierarchical implementation itself will not work properly.
The POSIX periodic scheduler might work as a FPS local scheduler for the hierarchical system, but we would probably want other scheduling policies as well.
In the end, we decided to implement our own local schedulers which should support FPS and EDF.

3.1.2 Design

The main actions of the local FPS and EDF schedulers are to put tasks in the ready queue at every period start and in the case of EDF, set new task priorities according to the absolute deadline values. Setting new priorities means that we actually set new priority values in the task TCB's and also re-order the ready queue (according to the new priorities).
Our local scheduler should just re-order the ready queue, it should not take the highest priority task in the ready queue and put it to execution. Putting a task to execution means that CPU registers are set. Some data from the task TCB is copied to these registers, for example the program counter (PC) value. All of this is done by the VxWorks scheduler. Our FPS and EDF local schedulers should just execute before the VxWorks scheduler at each interrupt and manipulate the ready queue. So in a sense, we just extend the VxWorks scheduler so it can handle periodic tasks. The EDF scheduler will also re-assign priorities.

As we described in section 2, the VxWorks scheduler is invoked at every interrupt not matter which source that is responsible. The VxWorks scheduler will only do a task switch if there has been a change in the ready queue.
Our schedulers should have the following functionalities.

• Put tasks in the ready queue at every task period. In the case of EDF, re-assign priorities of all tasks (based on their absolute deadlines) current in the ready queue.

• Do a check at every task deadline.

The solution is to have a routine, executed at every task period or deadline, which manipulates the ready queue. The VxWorks scheduler will then take care of task switching if it is necessary.
The routine must be executed at every time instance when there is a period or deadline. A sorted Time Event Queue (TEQ), with the nearest event first in the queue, should hold all tasks period and deadline times. The routine should sleep the amount of time labeled in the first node of the TEQ. When the routine is awakened, the first node or node's with the same time should be updated. The appropriate action should then be taken considering if it is a period or deadline event.
The work flow is illustrated in *Figure 16*.

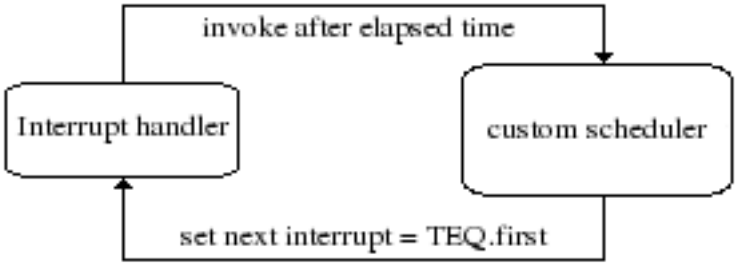

Figure 16: Workflow of local scheduler

3.1.3 Implementation

The implementation of the local schedulers were done by using an interrupt handler that was connected to the system hardware clock. The handler could be set to invoke after a specific amount of time. An ABB robotics controller with a Pentium 200 MHz processor was used, running a VxWorks version 5.2 operating system. The ABB controller's operating system was configured with the native VxWorks scheduler. Our local scheduler was therefore implemented as an extension to this scheduler, and not the POSIX scheduler.
We used system call routines and low level routines to changed task priorities and manipulate the ready queue.
The schedulers (FPS and EDF) follow an absolute timeline (starting at time 0) which wrap at some point so the time will restart again with the value of 0. Periods and deadlines may of course not exceed the wrap-around value.
We used system clock ticks as time base for our scheduler which can vary depending on OS configuration.
The ABB controller had a limitation of 32-bit integers, the counter would thereby wrap around at value $2^{32} - 1$. Having the ABB controllers maximum clock resolution of 4660 ticks per second would give a life span of approximately 256 hours before the conter would wrap around.

The handler has a local counter which hold the current absolute time. Each tasks absolute deadline and period times are stored in the TEQ. Updating the next event is simply a matter of getting the lowest value of all tasks deadlines and periods in the TEQ. These values wrap around in the same manner as the absolute timeline, thus the system must keep track of which absolute deadlines and periods have past the wrap-around value. The next interrupt will occur at the nearest absolute deadline or period time.
The TEQ itself is implemented as a sorted double linked list with two pointers pointing at the first and middle node (*Figure 17*). Extracting the first node is fast because of the first mentioned pointer. The second pointer is always adjusted so it always points at the middle node. Having a middle pointer speeds up insertions because the binary search method is used.
A sorted double linked list with binary search has the advantage that it has an even performance no matter which side of the middlepoint that the new node is inserted. An array with binary search is very effective when insertions are made in the beginning of the queue but rather slow in the opposite side of the queue. This is because it is fast in traversing but it has to move all nodes to the right of the new inserted node. Allocating extra memory could of course solve this matter, but the cost of vasting memory is not very appealing.
A second version of the sorted double linked list was implemented at a later point during this project. Two optimizations were added, two event queues were used with task deadlines in one of them and task periods in the other. The second optimization was to remove all nodes (with equal values) and then insert them all afterwards, instead of first remove and then insert each node. This is memory consuming but increases performance if there are nodes with equal values in the TEQ. Time measurements were made on both versions of the TEQ, the results can be found in the next chapter.

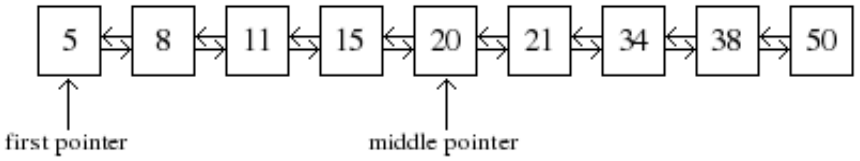

Figure 17: TEQ

The following pseudocode (*Figure 18*) describes the workflow for the FPS/EDF local scheduler interrupt handler.

```
function local_scheduler_interrupt_handler()

  // Deadline-event: check if the task is still in the ready queue
  for ALL_DEADLINE_EVENTS, i = 0, i = i + 1
    if (DEADLINE_EVENT_QUEUE[i].task = READY)
      log(deadline_miss, DEADLINE_EVENT_QUEUE[i].task)
    endif

    update_event_queue(DEADLINE_EVENT_QUEUE[i].task)
  endfor

  // Period-event: tasks are put in the ready queue at period starts
  for ALL_PERIOD_EVENTS, i = 0, i = i + 1
    insert_task_in_READY_queue(PERIOD_EVENT_QUEUE[i].task)

    // Tasks absolute deadlines and/or periods are updated in the sorted TEQ.
    // Updating the event queue is a matter of removing the node,
    // update the node's value and inserting the node.
    update_event_queue(PERIOD_EVENT_QUEUE[i].task)
  endfor

  // Set new priorities to tasks currently in the ready queue.
  // Task absolute deadlines are sorted in ascending order in the TEQ,
  // so priorities are set based on the order of the tasks deadlines in this queue.
  if EDF
    for ALL_TASKS_IN_READY_QUEUE, i = 0, i = i + 1
      set_priority(DEADLINE_EVENT_QUEUE[i].task, i)
    endfor
  endif

  // The nearest event is updated by extracting the smallest
  // absolute deadline or period in the TEQ's
  next_event = get_nearest_event(DEADLINE_EVENT_QUEUE, PERIOD_EVENT_QUEUE)

  expiration_time = next_event - ABSOLUTE_SYSTEM_TIME

  // Set next interrupt to invoke at nearest event
  set_next_interrupt(expiration_time, local_scheduler_interrupt_handler)

  // Update our local counter
  ABSOLUTE_SYSTEM_TIME = next_event

endfunction
```

Figure 18: Pseudocode for local scheduler

Figure 19 illustrates how the TEQ works with FPS (it works the same as with EDF). Tasks $\tau 1$, $\tau 2$ and $\tau 3$ data is shown in *Table 3* below.

Name	*Period*	*Executiontime*	*Deadline*	*Priority* ($0 = highest$)
τ_1	100	10	100	0
τ_2	250	10	250	1
τ_3	500	100	150	2

Table 3: Task set

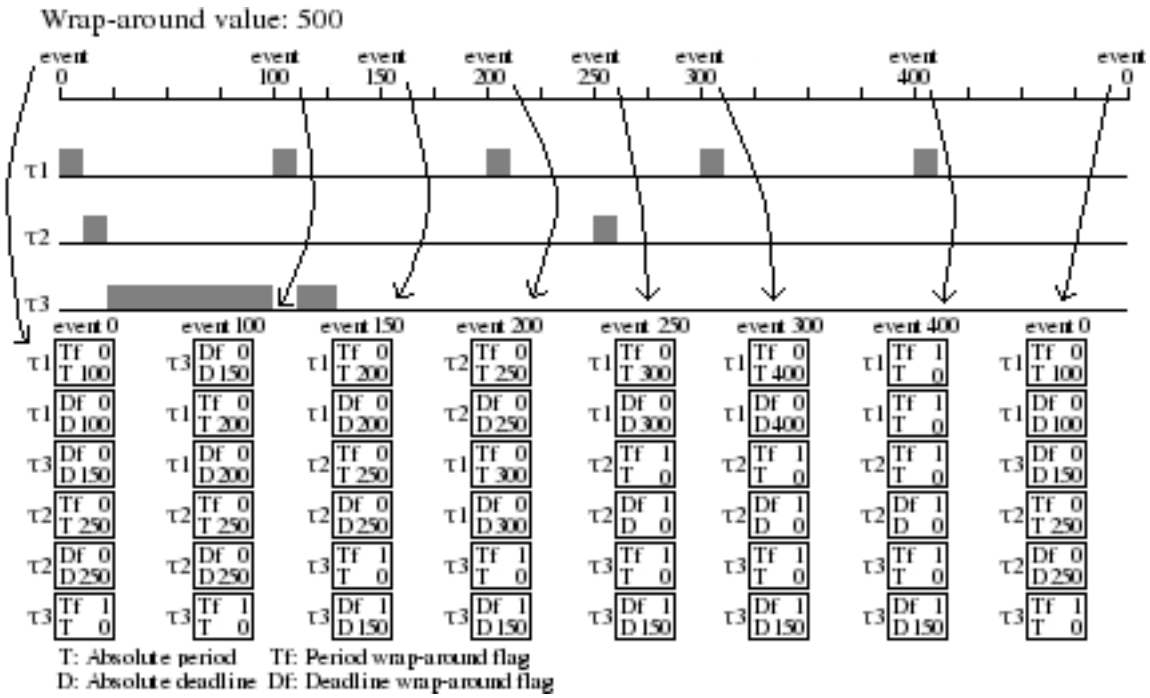

Figure 19: Example of how TEQ works

Note that if a wrap around flag (Tf or Df) in *Figure 19* is set to 1, then it's corresponding value is not considered when choosing the next event. When all flags have the value of 1, then they are all reset to 0.

3.1.4 Scheduling analysis with overhead

This chapter will only focus on the scheduling overhead for local FPS systems. The reason for this is that the EDF scheduling formula does not account for the number of preemptions as the RTA does. Calculating the number of preemptions in a EDF system is to complex to be included in this paper. Knowing the number of preemptions is crucial for expressing scheduler overhead. The rest of this chapter will show how to include overhead in the RTA formula.

Each task in a FPS system will have interrupts (local scheduler handler) associated with it. These interrupts will occur periodically at every task period and deadline. These interrupts are illustrated in *Figure 20*.

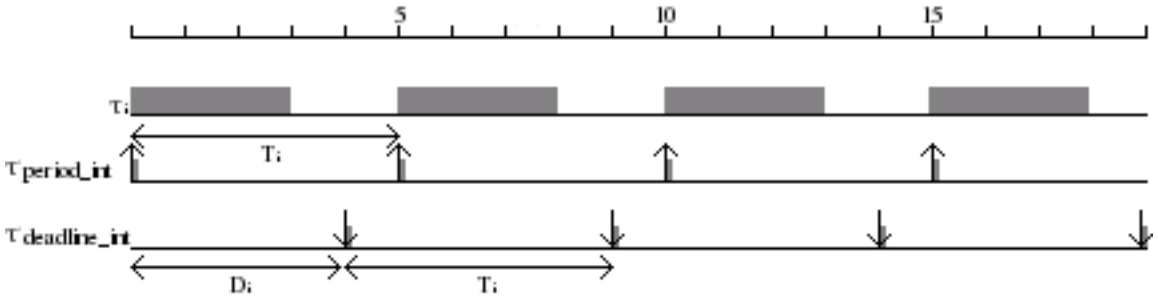

Figure 20: Interrupts associated with a task

These interrupts, that the scheduler is responsible for, can be seen as tasks themselves but with very short execution time (interrupt handler execution).
Definitions that are related to interrupts are the following.

• The execution time of an interrupt associated with a deadline check is defined as C_{Dint}

• The execution time of an interrupt associated with a period start is defined as C_{Tint}

• The amount of time to switch between a task and an interrupt routine (or vice versa) is defined as CS

• The amount of time to execute the VxWorks scheduler is defined as C_{OS}

When dealing with task systems that are scheduled with FPS, we analyse them with RTA. As stated earlier, each task will have one or two periodic interrupt executions associated with it. Interrupt executions related to a low priority task can preempt a high priority task because interrupts have higher priority than tasks (interrupts do not consider task priorities). This is why we include, in equation (15), that even lower priority tasks associated interrupt executions may preempt higher priority system tasks.
If deadline is equal to period then the deadline interrupt interference is the number of interrupt period instances (floor) times the execution time of C_{Dint} and two context switches CS.
If deadline is not equal to the period then the deadline interrupt interference is the number of deadline instances (floor) times the deadline interrupt execution and the context switches.
Period interrupts are counted with the mathematical ceiling. If there is room for a preempting task execution then there will also be room for a period interrupt execution. This is however not true for deadline interrupt executions.

RTA with scheduling overhead

$$R_i^n = C_i + C_{Tint} + 2CS \sum_{j \in hp(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil \cdot (C_j + 2C_{OS} + 4CS + C_{Tint}) + \sum_{j \in lp(i)} (C_{Tint} + 2CS) + Z \tag{15}$$

$$Z = \begin{cases} \sum_{j \in hp(i)} \left\lfloor \frac{R_i^{n-1} + T_j - D_j}{T_j} \right\rfloor \cdot (C_{Dint} + 2CS) & \text{if } T_i \neq D_i \\ \sum_{j \in hp(i)} \left\lfloor \frac{R_i^{n-1}}{T_j} \right\rfloor \cdot (C_{Dint} + 2CS) & \text{otherwise} \end{cases}$$

Figure 21 ($T = D$) and *Figure 22* ($T \neq D$) below illustrates how parts of the RTA formula corresponds to the execution scenario.

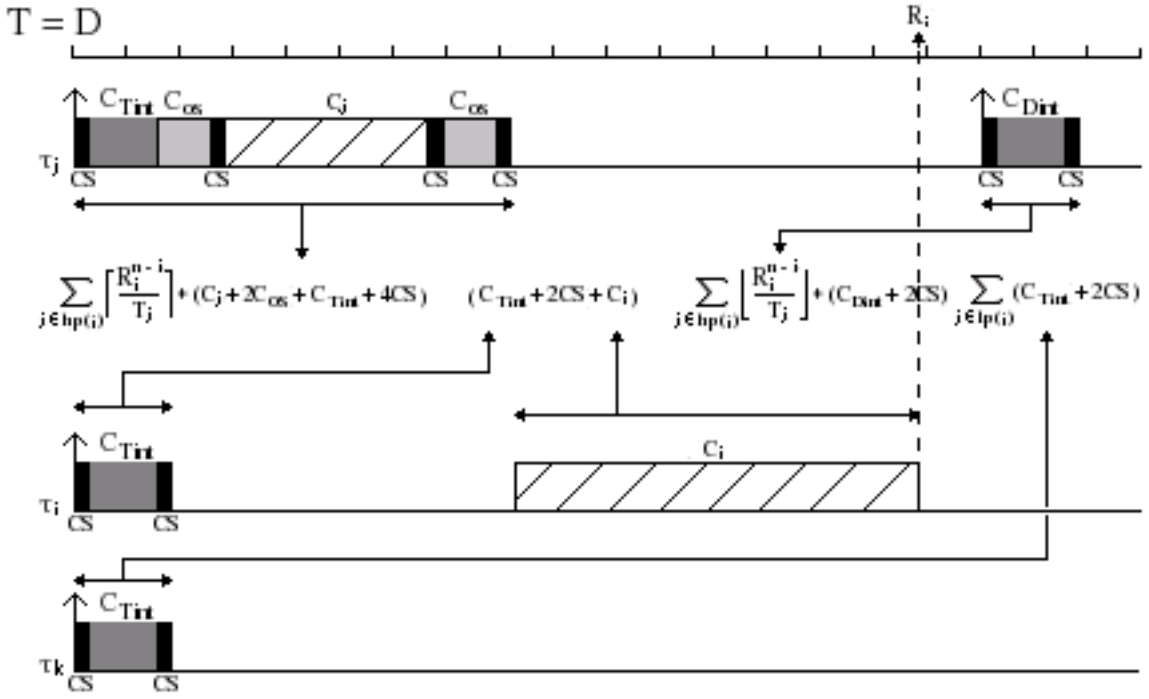

Figure 21: RTA with scheduling overhead ($T = D$)

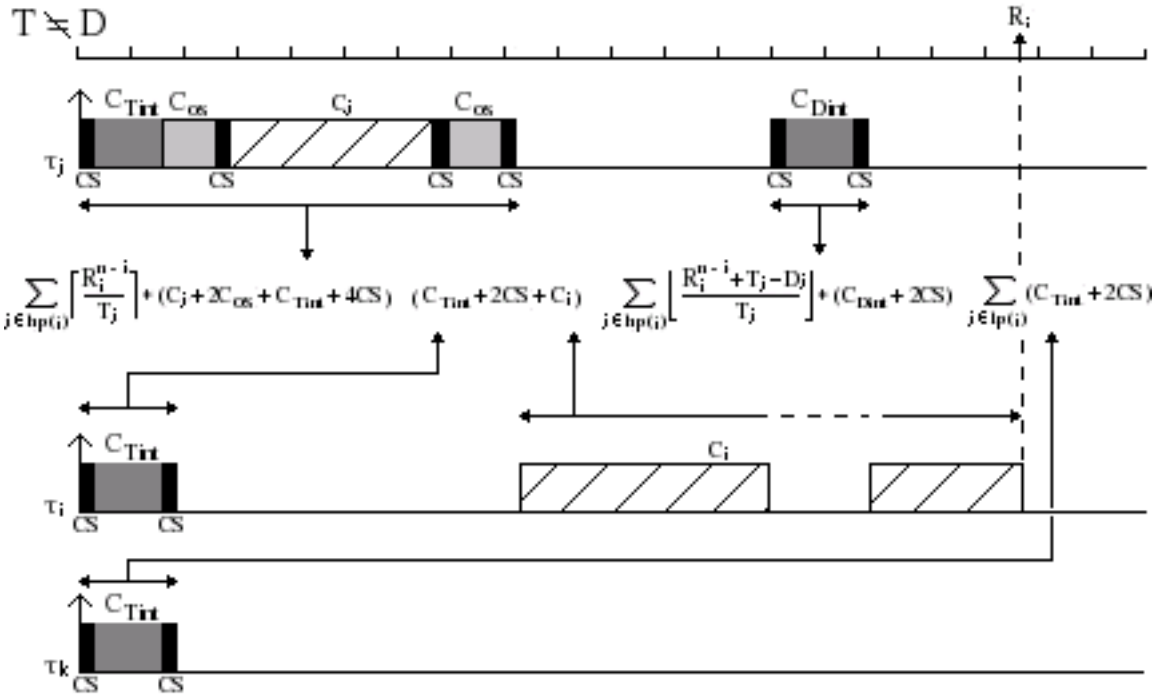

Figure 22: RTA with scheduling overhead ($T \neq D$)

3.1.5 Calculation of amount of interrupts

The RTA scheduling formula overhead calculations have overestimations. A more accurate calculation, which is easy and that may have great effect if interrupts are of high frequency and they often occur at the same time, can be done by subtracting interrupts that occur at the same time. If the interrupt handlers execution will handle more than one event (period or deadline events) then there will still only be two context switches. There will maybe also be a shorter execution if there is much execution that is shared and not dependant on each event handling, for example setting the timer for the next expiration. For simplicity, we will only subtract the extra context switches that the overhead calculation overestimates.
The algorithm *Count_Double_Hit* (*Appendix B*) compares two tasks at a time and searches for time instances where they both have interrupts. Every instance where two or more interrupts occur at the same time will be counted.

Figure 23 below is a simple example of how the algorithm traverses the tasks in the search for double instances. First, task τ_1 is compared to τ_2 and τ_3, then τ_2 against τ_3.

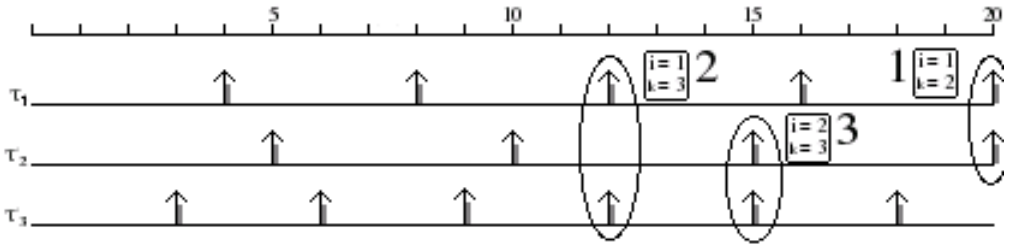

Figure 23: Algorithm traversing

This function can be integrated to the RTA scheduling formula to reduce the overestimation of the amount of interrupt context switches. Below is the formula (16) with the function *Count_Double_Hit* inserted.

RTA with scheduling overhead and optimized interrupt counting

$$R_i^n = C_i + C_{Tint} + 2CS \sum_{j \in hp(i)} \left\lceil \frac{R_i^{n-1}}{T_j} \right\rceil \cdot (C_j + 2C_{OS} + 4CS + C_{Tint}) + \sum_{j \in lp(i)} (C_{Tint} + 2CS) + Z - (Count\_Double\_Hit(R_i^{n-1}) \cdot (2CS)) \quad (16)$$

3.2 Hierarchical framework

3.2.1 Overview

Having the local schedulers implemented as well as the interrupt handling and time event queue handling, we now look into how to put the global scheduler on top of this and introduce the notion of servers. *Figure 24* below illustrates how these subsystems are layered.

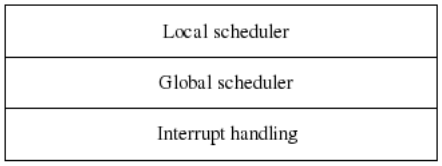

Figure 24: Subsystem layering

The bottom layer *Interrupt handling* is responsible for how interrupts are administrated. This layer could use the VxWorks watchdog facility or some other more hardware specific solution such as manipulating hardware timers directly. The important aspect is that the interface is not affected, so the other subsystems do not need changes.
The *Global scheduling* layer supports both FPS and EDF scheduling together with the periodic server model.
Global scheduler is responsible for handling all events in the system which can be local task subsystem events, server period events or server budget events.
Local scheduler is not aware of the fact that it schedules different task systems, it's interface to *Global scheduler* is a server which has it's own task tcb and task TEQ's.

Figure 25 illustrates how parameters in the hierarchical framework are related.

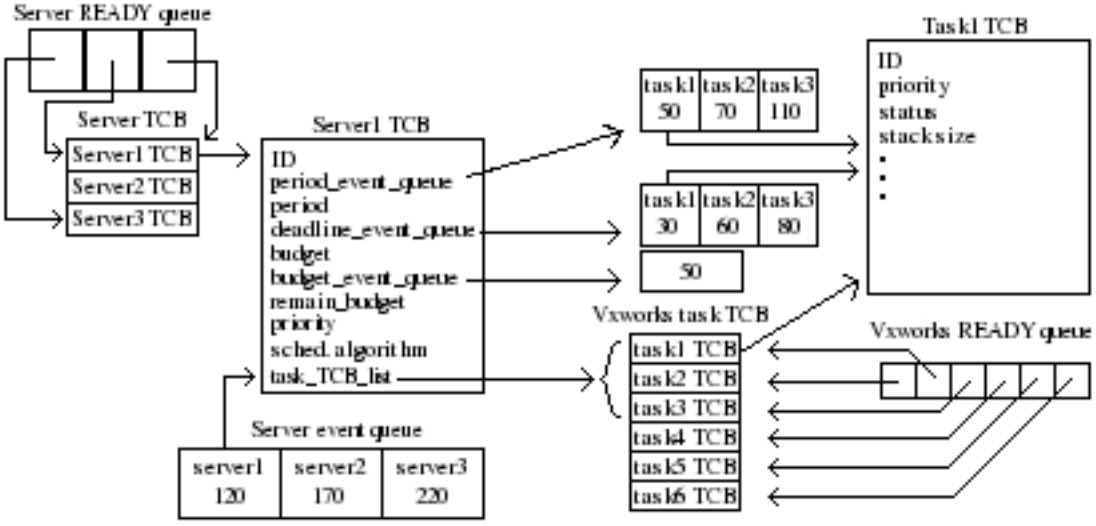

Figure 25: Overview of HSF parameters

In *Figure 25*, there exists a server ready queue (*Server READY queue*) with references to server TCB's. These TCB's contain the following.

ID is a unique number associated with each server.

period_event_queue is a reference to the servers tasks TEQ which holds task periods.

period is the actual period of the server.

deadline_event_queue is a reference to the servers tasks TEQ which holds task deadlines.

budget is the servers defined budget.

budget_event_queue is the servers TEQ holding one budget node.

remain_budget is the current remaining budget of the server. In the case of server preemptions, the server will store the amount of budget left to execute.

priority is the servers priority, this parameter will change if the global scheduler is EDF.

sched_algorithm is the servers local scheduling algorithm.

task_TCB_list is a reference to a part of the VxWorks TCB list. It references those task TCB's that are associated with the server.

The *Server event queue* stores server periods, these nodes also store a reference to it's associated server. In the case of a server period interrupt execution, the server event queue is examined. The server period events can easily be handled due to the reference to the server TCB. The server TCB is referenced into the server ready queue when the server has a period start.
Those tasks that are in the VxWorks ready queue have a reference from the queue to it's TCB. Each node in the *period_event_queue* and *deadline_event_queue* have each a reference to it's associated tasks TCB. In the case of task period interrupt execution, the TCB can easily be referenced and put in the task ready queue.

The global scheduler chooses the closest event of the running servers task deadline TEQ, the task period TEQ, the server budget TEQ (only one node) and the server period TEQ (we assume that server period is equal to server deadline). In the example in *Figure 25* above, the next event would be a task deadline at absolute time 30 if server 1 was the active server at that point.
Figure 25 also illustrates that each server has a reference to a part of the VxWorks task ready queue, this is hidden from the local scheduler which schedules as if this is the whole task tcb of the system.

Even the scheduler absolute time and each server budget is implemented as a TEQ. The reason for this is because it makes it easier to keep track of absolute time wrap arounds. The TEQ module keeps track of all TEQ's in the system, it wraps the absolute time back to zero in all queue's when all the TEQ's has past time zero. If a queue pass the absolute time zero it will not actually wrap around to zero, but keep on going without wrapping. This is illustrated in *Figure 26* below.

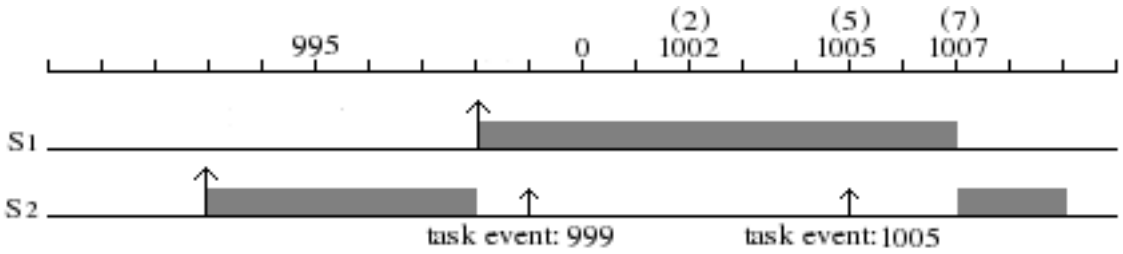

Figure 26: Time wrap around example

In *Figure 26* above server S2 has two task events at time 999 respectively 5 which can not be handled due to the preemption by server S1. At time 7 server S2 will run again, at this point there might be TEQ's that have wrapped and which may have not. Those TEQ's that have wrapped must still keep the high time value 1007 as well as the absolute system time, otherwise it will be difficult to compare events in different queue's. Also, handling event with time 999 can not be done if the system time is 7 because it is in the future and will not be handled. The solution here is to keep all TEQ's and system time high until the last event in the last TEQ has past time zero.

3.2.2 Detailed design

This section describes more precise how the hierarchical framework is designed and implemented. Some changes were done on the local scheduler to be able to work correct in the hierarchical system. The fact that task events might not always be handled in time but some time later forced some changes to the implementation of the local scheduler. The difference is that it now can handle events in the past.

The TEQ module was also modified, now it can handle more than one TEQ and it also keeps track of all TEQ's wrap arounds. All queue's will wrap when all queue's has wrapped including the system time as well.

A server has three TEQ's, task periods, task deadlines and the server budget. Having separate budget TEQ's for each server is necessary otherwise we do not know if the budget TEQ has wrapped. This is because it will store budgets from different servers, the previous value is from another server which even might make it become smaller.

There is a separation between the global and local scheduler because the server keeps it's own task tcb, task TEQ's and information about which local scheduling algorithm that is used.

The static view of the HSF's modules is shown in *Figure 27* below.

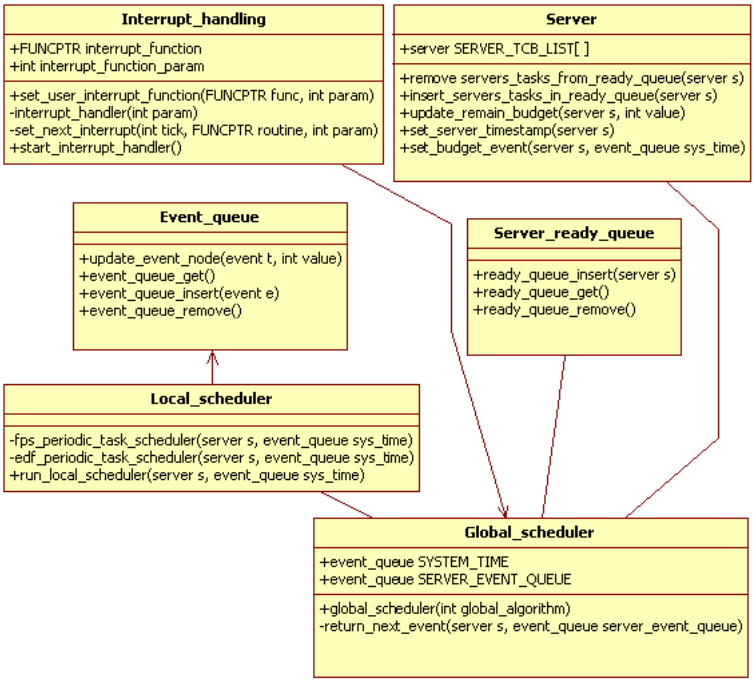

Figure 27: Static view of HSF

In the subsystem *Interrupt_handling*, the function *interrupt_handler* is connected to whatever library or timer that is used. It is called at the specified delay for which the registered interrupt function *interrupt_function* will return. The only demand on the user interrupt function is that it returns the amount of ticks for which the next interrupt should occur.
set_user_interrupt_function and *start_interrupt_handler* is used at system initialization to register an interrupt function and start the interrupt handling. *global_scheduler* is the function to register to the interrupt handling if our HSF is to be used. The function *return_next_event* is used internally by *global_scheduler* to fetch the nearest event. The subsystem *Global_scheduler* uses the *Server_ready_queue* to organize *Server*'s that are ready.
The *Local_scheduler* is used by *Global_scheduler* to schedule the tasks associated with a *Server*. The local and global scheduler uses *Event_queue* (TEQ) for handling task and server time events.

The user function registration is illustrated in *Figure 28* below.

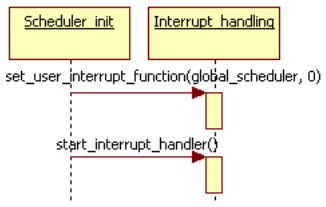

Figure 28: Function registration

There are three different types of events that the global scheduler must be able to handle:

- Task period or deadline event
- Server period event
- Budget exhaustion event

The following sequence diagram (*Figure 29*) illustrates a task event.

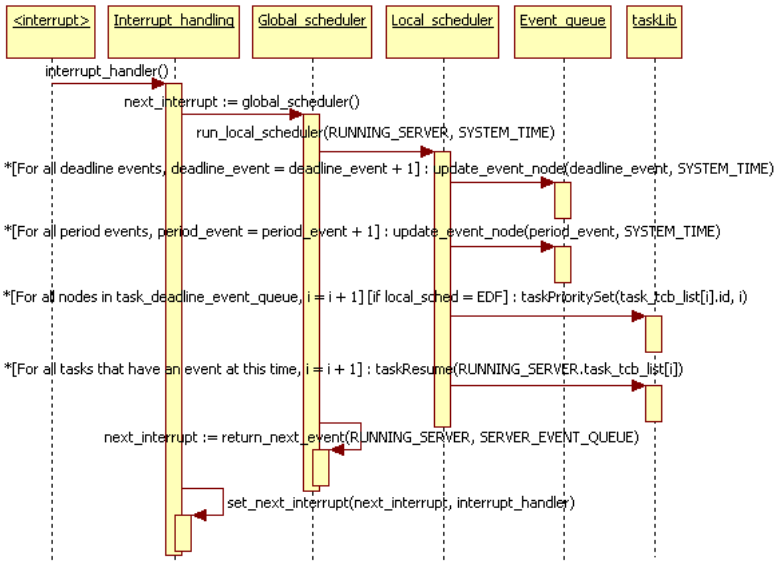

Figure 29: Task event

In the case of an interrupt, *interrupt_handler* calls it's registered function *global_scheduler*. The global scheduler runs the local scheduler in case of task event. All period and deadline events at the current and past absolute time are handled. Deadlines misses are logged and tasks are put in the VxWorks ready queue in case of period start. In the case of a local EDF subsystem, all tasks in the ready queue are sorted according to their deadlines.
At the end, the nearest event is fetched and returned to *interrupt_handler*.

A server period event is illustrated in *Figure 30* below.

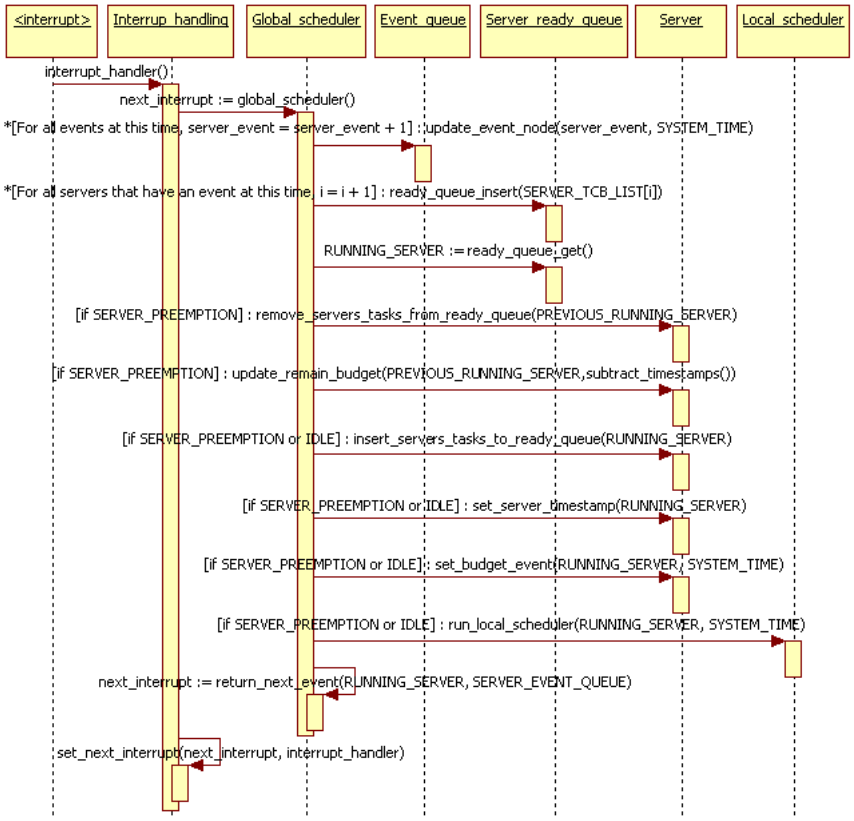

Figure 30: Server period event

The sequence diagram shows that the server events are handled first, then the corresponding servers are also put in the server ready queue. The first server in the ready queue is fetched, at this point, there are three cases:

- There is a server context switch (server preemption), remove previous server and insert new
- It was idle before, insert new server
- No server context switch (do nothing)

The first two cases are illustrated in *Figure 30*. Removing a server means that we have to remove the servers tasks from the ready queue, these tasks are flagged so we know which tasks to insert later. Tasks that are in a blocked state will get it's priority lowered to the lowest possible. The removed server will also have it's remaining budget updated. This means that it's saved timestamp value (which it got when it started to execute) will be subtracted with the current timestamp, this value in turn is subtracted from the remaining budget.
Activating a server means that it's tasks are inserted in the ready queue, it will get the current timestamp value, it's budget timer will be updated (system time added with it's budget) and last it's tasks will be scheduled by the local scheduler.

In the last case, the global scheduler will do nothing after the server events are handled.

The third and last event is when a server budget is exhausted, this is shown in the sequence diagram below (*Figure 31*).

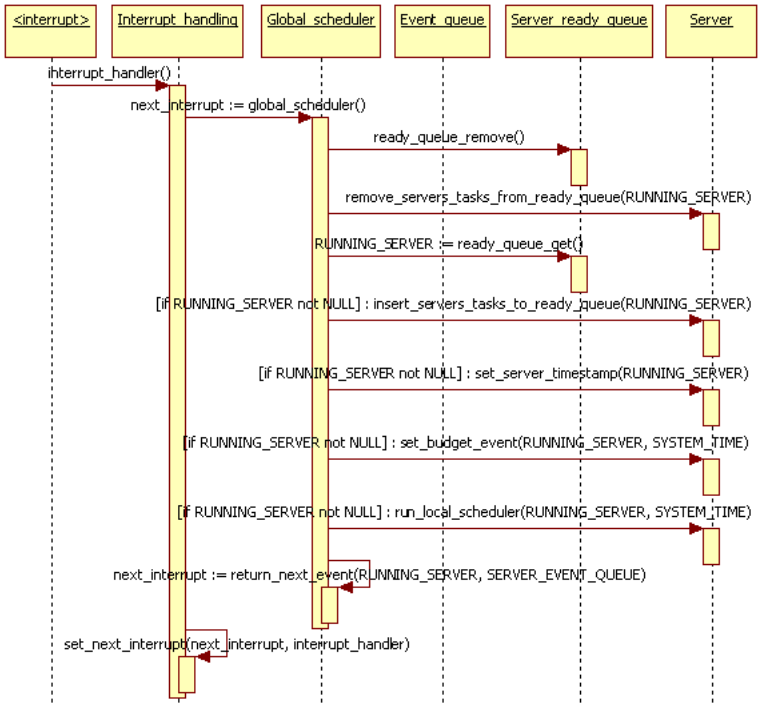

Figure 31: Server budget exhaustion

As shown in *Figure 31*, the running server is removed from the server ready queue. Also, it's tasks are removed from the task ready queue. A new server is then fetched from the ready queue if it is not empty. Activating a new server is the same procedure as shown in *Figure 30*. There may be a new server that starts to execute or the idle task will run if no server is ready to execute.

3.2.3 Scheduling analysis with overhead

The extra overhead that the global scheduler generates as well as the local scheduling overhead is modelled as a part of server budget in order to minimize system drift caused by the HSF.

The execution time of the global scheduler can be separated by:

• Server period start interrupt execution (including period starts from all servers in a hierarchical system) and budget interrupt execution (including activating and deactivating a server) ($C_{\Pi int}$ and $C_{\Theta int}$).

• Inserting or removing a server task from the task ready queue (C_Q).

The distinction from different types of server execution time and the dependancy to server task sets is included in the global scheduler overhead analysis.

The following definitions are related to the global scheduling overhead analysis.

The number of tasks belonging to server S_i is defined as: N_i

To analyze wether a taskset is schedulable with it's server and our implemented local scheduler, we use the formula derived in section 3.1.3.

Scheduling analysis for a system

$$S_i(W_i, \Gamma_i(\Pi_i, \Theta_i^{new}), FPS)$$

with respect to it's server is done with the new budget definition Θ^{new}

$$\Theta_i^{new} \;=\; \Theta_i \;-\; O_i^{global} \tag{17}$$

where O_i^{global} is the global scheduling overhead for server S_i defined as

$$O_i^{global} \;=\; C_{\Pi int} + 2CS + (N_i{+}M){\cdot}\, C_Q + C_{OS} + \sum_{j \in hp(i,S)} \left\lceil \frac{R_i^n}{\Pi_j} \right\rceil \cdot\, (C_{\Theta int} + 2CS + (N_i + N_j){\cdot}\, C_Q) \tag{18}$$

where R_i^n is the response time of server S_i and M is defined as

$$M \;=\; max(N_j \mid \forall\; j \;\in\; lp(i,S)) \tag{19}$$

where $lp(i,S)$ is the set of servers with lower priority than server S_i

M is the maximum number of tasks for a server (S_j) for which has lower priority than server S_i.

Our local scheduling overhead formula for FPS (15) together with hierarchical scheduling for local FPS systems looks as follows.

$$r_i^{(k)}(\Gamma_i) \;=\; tbf_{\Gamma_i}(R_i^{(n)})$$

where $R_i^{(n)}$ is the response time of our local scheduling overhead formula

A local FPS system is schedulable if the following holds.

$$\forall\; j \;\in W_i : \; r_j^n \leq \; D_j$$

The budget interface Θ_i^{new} is the 'real' budget that tasks actually can use and for which is used for the service time function tbf_{Γ_i} when analysing local tasks systems against the server parameters.

Each preemption of higher priority servers and the server period start will decrease the budget Θ_i^{new}. The cause of this is that any scheduling overhead will be subtracted from the next running server. This fact is illustrated in *Figure 32* below.

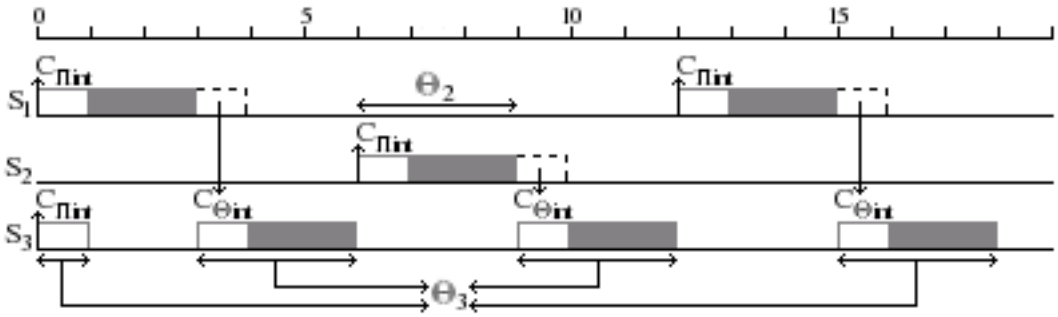

Figure 32: Budget decrease caused by preempting servers

The global scheduling analysis with overhead has the same appearance as equation (13) and (14). The global FPS scheduler analysis assumes that switching tasks to/from the task ready queue is done on all respectively servers tasks when a server preemption occurs and not only on those tasks that are in the ready queue.

Including scheduling overhead in the budget will have a big affect on decreasing the drift that the HSF causes. This is implemented by measuring the global and local scheduler execution and subtract that value from the next running servers budget.
Figure 33 below illustrates HSF overhead included in server budgets to reduce drift.

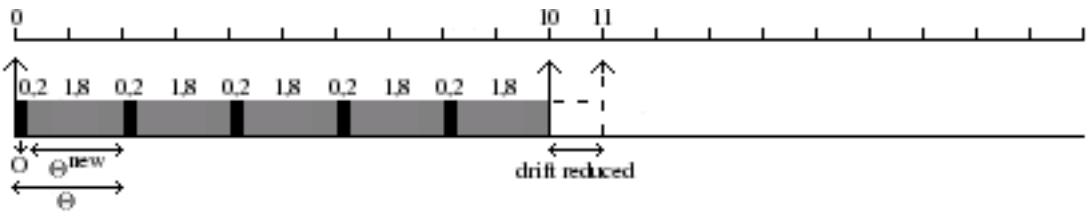

Figure 33: Drift reducing

4 Results

4.1 Interrupt overhead measurements

There are different ways to use timer interrupts in VxWorks. We can use different hardware timers such as the system clock, auxiliary clock or other external timers. VxWorks provide libraries that can be used to connect interrupt handlers to hardware timers. The system and the auxiliary clock timers can be used through these libraries. The library in VxWorks 5.2 in the ABB robotics controller did not support the auxiliary clock at frequencies higher than 60 ticks/second, so we did not use this timer. Our remaining options were to use the watchdog timer facility (uses the system clock) or connect a handler to the system clock. The system clock library supported frequencies up to 4660 ticks/second, this clock rate could be changed through library calls during runtime.
We want to be able to set the time between each interrupt. Changing the clock rate is the way to do it, but it is not accurate enough. Suppose we want the interrupt to invoke in exactly 66 milliseconds, the clock rate of 15 ticks/second will cause an interrupt in 1000/15 = 66,666... milliseconds. The solution (using libraries) is to either use watchdog timers (*Figure 35*) where you can set the next expiration in number of ticks, or have a interrupt handler that is invoked at every tick and uses a local counter (*Figure 34*).

```
/* Handler that is connected to the system clock (without tick announce) */
void handler() {

   static int counter = 0;
   static int next_expiration = 0;

   if (counter == next_expiration) {
      // User code
      next_expiration = ...;
      counter = 0;
   }

   counter++;
}
```

Figure 34: Handler connected to system clock

```
/* Handler that is connected to the system clock through the watchdog library */
void handler() {

   int next_expiration;

   // User code
   next_expiration = ...;
   wdStart(handler, next_expiration);
}
```

Figure 35: Watchdog

The above solutions (*Figure 34* and *Figure 35*) causes overhead between each expiration. The first solution executes an if-statement and a counter increment at each tick between the user specified expirations. The second solution uses the watchdog facility which in turn uses the system clock

interrupt routine (usrClock) to count the time to the next expiration (*Figure 36*).

```
/* System clock interrupt routine */
void usrClock() {

   tickAnnounce();
}
```

Figure 36: System clock interrupt routine

The system clock interrupt routine announces a tick to the kernel tick counter at every clock interrupt. The watchdog uses this kernel tick counter to administrate the time delay between each watchdog interrupt. This can be proved by disabling clock interrupts, the kernel tick counter will not increment and watchdog timers will not work.
We measured the overhead that these two solutions causes between each expiration. The measurement was done by reading a timestamp value at the start and end of a task, and then subtracting these two values. The timestamp reading was done by reading a memory mapped address of a dsq500 hardware timer counter register. The dsq500 hardware counter had a frequency of 12000000 ticks/second, approximately 83 nanoseconds per counter tick. We first measured the task execution time with interrupt disable, and then with interrupt enable. The difference in values is the overhead from the system clock interrupt routine. Finally we measured the overhead from a watchdog timer (*Figure 35*) and an interrupt handler connected to the system clock (*Figure 34*). The ABB robotics controller was running with a system clock rate of 4500 ticks/second during these tests.

System configuration	*Number of test runs*	*Mean time (microsec)*
Interrupt disable	10	997911,73
Interrupt enable	80	1015942,91
Watchdog	80	1015929,53
Interrupt handler (without tick announce)	80	1012461,68
Interrupt handler (with tick announce)	80	1016057,12

Table 4: Interrupt overhead

The variation in execution time for the task was low when running at interrupt disable, so we were satisfied with only 10 test runs.
From the result (*Table 4*) we see that the watchdog facility overhead is so small compared to interrupt enable that it does not show in the tests.
When installing an interrupt handler, the system clock interrupt routine is detached. The difference in time between interrupt disable and having an interrupt handler (without tick announce) is the handler execution time only. Note that some system facilities will not work due to that the kernel tick counter will not increment. We get the same functionality as having the system clock interrupt routine attached if we include the tick announce functionality to the interrupt handler. The test results show that there is a very small difference in overhead between watchdog and interrupt handler (with tick announce). The interrupt handler may still function properly without the kernel tick counter, where as the watchdog will not work without it. If the application is not dependant on the kernel tick counter then using the interrupt handler (without tick announce) is the best choice considering overhead.

The best interrupt routine solution is to manipulate hardware registers directly through memory mapped registers. Hardware timers usually have two registers, a terminal count register and a counter register. When the counter register value reaches zero or the terminal count value (depending on hardware), an interrupt will fire at the specified interrupt vector. Setting the terminal count register value is equal to setting the time to the next interrupt. The counter register ticks at very high frequencies so it is very accurate.
This solution will not cause overhead between expirations, the downside is that the implementation is hardware dependant. Watchdogs or interrupt handlers are completely hardware independent, but the cost is the extra overhead between expirations.

Our implementation uses the watchdog library.

4.2 Local scheduler

4.2.1 Time measurements of the local schedulers

The interrupt execution of the deadline check (C_{Dint}) and the period start (C_{Tint}) was measured on the ABB robotics controller with a clock rate of 1000 ticks/second. Measurements were made with the dsq 500 hardware timer, the timer was read in the beginning and at the end of of the interrupt handler function. All tasks had the same period, deadline and execution time, T=200 ms, D=150 ms and C=1 ms. Each C_{Tint} and C_{Dint} handles all tasks simultaneously.

Scheduler	*Number of tasks*	*Number of test runs*	*Mean time* (µs)
FPS	10	45	79,63
FPS	20	45	147,76
FPS	30	45	217,66
FPS	40	45	298,98
FPS	50	45	378,31
FPS	60	45	479,43
FPS	70	45	571,91
FPS	80	45	686,68
FPS	90	45	792,68
FPS	100	45	916,12

Table 5: Period start interrupt execution for FPS (C_{Tint})

Scheduler	*Number of tasks*	*Number of test runs*	*Mean time* (µs)
EDF	10	45	83,11
EDF	20	45	155,69
EDF	30	45	234,74
EDF	40	45	315,59
EDF	50	45	399,71
EDF	60	45	510,48
EDF	70	45	611,03
EDF	80	45	743,69
EDF	90	45	852,55
EDF	100	45	990,93

Table 6: Period start interrupt execution for EDF (C_{Tint})

Scheduler	*Number of tasks*	*Number of test runs*	*Mean time* (µs)
FPS	10	45	74,20
FPS	20	45	138,23
FPS	30	45	208,65
FPS	40	45	288,38
FPS	50	45	359,87
FPS	60	45	458,51
FPS	70	45	548,48
FPS	80	45	657,74
FPS	90	45	760,12
FPS	100	45	873,30

Table 7: Deadline check interrupt execution for FPS (C_{Dint})

Scheduler	*Number of tasks*	*Number of test runs*	*Mean time* (µs)
EDF	10	45	72,58
EDF	20	45	135,47
EDF	30	45	206,35
EDF	40	45	283,07
EDF	50	45	353,59
EDF	60	45	448,39
EDF	70	45	540,38
EDF	80	45	651,97
EDF	90	45	751,84
EDF	100	45	864,51

Table 8: Deadline check interrupt execution for EDF (C_{Dint})

Here are the handler interrupt execution time measurements where two optimizations are included to the TEQ that the handler uses.

Scheduler	*# tasks*	*# test runs*	*Mean time* (µs)	*Max time* (µs)	*Min time* (µs)
FPS	10	50	65,75	71	63
FPS	20	50	109,75	119	106
FPS	30	50	158,25	172	155
FPS	40	50	202,00	214	197
FPS	50	50	256,08	266	249
FPS	60	50	305,83	318	299
FPS	70	50	352,00	367	341
FPS	80	50	404,42	422	397
FPS	90	50	459,42	473	453
FPS	100	50	516,50	527	511

Table 9: Period start interrupt execution with optimized TEQ for FPS (C_{Tint})

Scheduler	*# tasks*	*# test runs*	*Mean time* (µs)	*Max time* (µs)	*Min time* (µs)
EDF	10	50	69,75	74	68
EDF	20	50	118,58	131	115
EDF	30	50	172,75	187	169
EDF	40	50	228,00	241	220
EDF	50	50	280,58	296	275
EDF	60	50	338,75	359	331
EDF	70	50	396,83	415	390
EDF	80	50	453,58	476	444
EDF	90	50	523,08	539	515
EDF	100	50	589,33	600	583

Table 10: Period start interrupt execution with optimized TEQ for EDF (C_{Tint})

Scheduler	*# tasks*	*# test runs*	*Mean time* (µs)	*Max time* (µs)	*Min time* (µs)
FPS	10	50	60,75	70	57
FPS	20	50	100,25	111	95
FPS	30	50	141,92	151	137
FPS	40	50	180,33	192	175
FPS	50	50	225,75	236	219
FPS	60	50	268,00	282	262
FPS	70	50	309,50	324	304
FPS	80	50	354,17	371	349
FPS	90	50	398,08	415	393
FPS	100	50	442,25	459	436

Table 11: Deadline check interrupt execution with optimized TEQ for FPS (C_{Dint})

Scheduler	*# tasks*	*# test runs*	*Mean time* (µs)	*Max time* (µs)	*Min time* (µs)
EDF	10	50	61,25	72	57
EDF	20	50	99,42	112	95
EDF	30	50	138,83	155	132
EDF	40	50	185,08	197	179
EDF	50	50	227,67	241	222
EDF	60	50	268,50	279	263
EDF	70	50	309,08	325	303
EDF	80	50	353,67	365	346
EDF	90	50	396,83	416	390
EDF	100	50	444,08	461	437

Table 12: Deadline check interrupt execution with optimized TEQ for EDF (C_{Dint})

The corresponding graphs and equations of the table data above (*Table 5-12*) can be found in *Appendix A*.

4.2.2 Comparison of scheduling overhead formula with execution measurements

In order to compare the formula against execution measurements, we need to approximate C_{Dint}, C_{Tint}, CS and C_{OS}. With these values estimated, we can then calculate response time for a task system using the overhead formula (15). Further, the same task system can be executed and measured in a VxWorks system. The comparison of the calculated and measured values reveal how accurate the overhead formula is.

The context switch (CS) was measured by running a task with execution time of 1000,006 ms with interrupt enable and interrupt disable. When interrupts were enabled, an empty interrupt handler was executed. The clock rate on the ABB robotics controller was set to 1000 ticks/second. The difference in time between interrupt disable and enable should correspond to the two context switches at each interrupt. Dividing this difference in time with each interrupt (1000) times two context switches (per interrupt) gives a value of:

$$CS: \ 1,573 \ \mu s$$

Three test runs were done, the above value is the mean time of these test run values.

The value of C_{OS} was estimated by having one task restarted (inserted in the ready queue) by an interrupt routine. If the ready queue is altered then the VxWorks scheduler will be invoked, as in this case. The dsq 500 hardware timer was read at the end of the interrupt routine and at the beginning of the task. This is illustrated in *Figure 37* below.

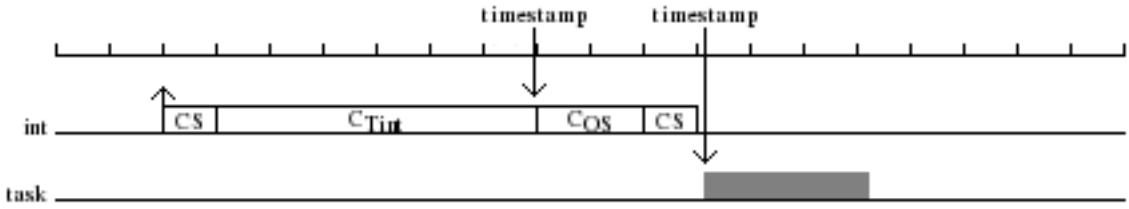

Figure 37: How to measure C_{OS}

The context switch in between these two timestamps is subtracted (1,573 µs). The clock rate on the ABB controller was set to 1000 ticks/second and the execution time of the task was less than 1 ms. The following value was measured:

$$C_{OS}: \ 14,57 \ \mu s$$

500 test runs were done, the above value is the mean time of these test run values.

The task set in *Table 13* below was calculated with the RTA scheduling overhead formula (15).

Task	*C (ms)*	*T (ms)*	*D (ms)*	*Priority (0 = highest)*
$\tau 1$	10	100	60	0
$\tau 2$	10	100	70	1
$\tau 3$	10	100	80	2
$\tau 4$	10	100	90	3
$\tau 5$	10	100	100	4
$\tau 6$	10	100	100	5
$\tau 7$	10	100	100	6
$\tau 8$	10	100	100	7
$\tau 9$	10	100	100	8
$\tau 10$	1000	15000	15000	9

Table 13: Task set

The following overhead constants (*Table 14*) were used.

Overhead constant	*Value (µs)*
C_{Tint}	25
C_{Dint}	25
CS	1,5
C_{OS}	15

Table 14: Overhead constants

Note that the values for C_{Tint} and C_{Dint} are overestimated. A part of the execution time is independent, no matter how many tasks that are handled at the same time. This shared execution time is, for example, setting the next expiration time for the timer.
We calculated the response time of task $\tau 10$ with formula (15). The measurements of the task set were made on the ABB controller with the software tool Tracealyzer [13] which in turn uses the dsq 500 timer as time base. The system clock rate was set to 1000 ticks/second. Task execution times were measured with the dsq500 hardware timer with interrupt disable.
We got the following results (*Table 15*).

Theoretical value (ms)	*Calculated value (ms)*	*Measured value (ms)*
10000	$R_{\tau 10}^{33} = 10893$	10506,2

Table 15: Response time results

Screenshots from the Tracealyzer software can be found in *Appendix C*.
We used a watchdog interrupt routine which means that system interrupt (1000 ticks/second) executions are present. These interrupts will cause some overhead which will have some effect on the measurement.

4.3 Hierarchical framework

4.3.1 Time measurements of the global schedulers

The interrupt execution of the budget exhaustion ($C_{\Theta int}$) and the period start ($C_{\Pi int}$) was measured on the ABB robotics controller with a clock rate of 1000 ticks/second. Measurements were made with the dsq 500 hardware timer, the timer was read in the beginning and at the end of of the interrupt handler function exclusive the local scheduler execution and inserting/removing servers tasks. All servers had the same period and budget, Π=200 ms, Θ=1 ms. The measurements are presented in *Table 16* and *Table 17*.

Scheduler	*# servers*	*# test runs*	*Mean time* (µs)	*Max time* (µs)	*Min time* (µs)
FPS	10	100	89	91	85
FPS	20	100	145,92	149	139
FPS	30	100	205,67	212	189
FPS	40	100	267,58	274	243
FPS	50	100	333,08	344	318
FPS	60	100	399,67	412	388
FPS	70	100	466,17	483	417
FPS	80	100	534	548	509
FPS	90	100	604,67	630	525
FPS	100	100	667,5	689	570

Table 16: Period start interrupt execution for FPS ($C_{\Pi int}$)

Scheduler	*# servers*	*# test runs*	*Mean time* (µs)	*Max time* (µs)	*Min time* (µs)
EDF	10	100	106,42	109	103
EDF	20	100	184,67	189	175
EDF	30	100	267,5	273	253
EDF	40	100	353,25	361	334
EDF	50	100	444,92	454	435
EDF	60	100	539,33	545	522
EDF	70	100	632,33	642	587
EDF	80	100	732,83	740	716
EDF	90	100	834,08	856	760
EDF	100	100	929,83	955	841

Table 17: Period start interrupt execution for EDF ($C_{\Pi int}$)

Comparing the scheduler execution time (*Table 16* and *Table 17*) for FPS and EDF, we see a big difference. The cause of this is that changing priorities of servers is time consuming because every server must be removed and inserted in the server ready queue. In other words, our server ready queue is not time efficient.
The graphs of *Table 16* and *Table 17* can be found in *Appendix A*.

The budget exhaustion execution $C_{\Theta int}$ measurements include both activating and deactivating a server (server switch). FPS and EDF global scheduler have the same values because we do not update priorities during a budget exhaustion execution (*Table 18*).

# test runs	*Mean time* (µs)	*Max time* (µs)	*Min time* (µs)
100	30,92	35	14

Table 18: Budget exhaustion interrupt execution ($C_{\Theta int}$)

4.3.2 Comparison of scheduling overhead formula with execution measurements

The ABB robotics controller is limited to maximum 4660 ticks/second which gives a bad resolution for setting the next handler expiration. Due to this problem, we instead used a computer with a Pentium 4 processor running a VxWorks operating system version 6.6. All tests on this computer was done with a clock rate of 200000 ticks/second. We hardcoded the subtraction of handler execution time from server budget because the Pentium 4 computer did not have an external hardware timer as the ABB controller has (dsq500 timer) which can measure handler execution time accurately. The system clock (or auxiliary clock) could not be used for timestamping in the interrupt routine because the handler **usrClock** did not run during our handler which means that the tick counter will not be updated.
The following table (*Table 19*) presents constants that were measured on the Pentium 4 computer.

Measured constant	*Measured value* (µs)
C_Q	1
CS	1
C_{OS}	8
C_{Tint}	12
C_{Bint}	12
$C_{\Pi int}$	20
$C_{\Theta int}$	15

Table 19: Measured constants

We measured a server set of 100 servers. These are presented in *Table 20*.

Server	Π (ms)	Θ (ms)
$Server1 - Server99$	1000	10
$Server100$	10000	10

Table 20: Server set

Server100's respons time was calculated with equation (13) and measured on the Pentium 4 computer (*Table 21*).

Calculated value (ms)	*Measured value* (ms)
$R^1_{S_{100}} = 1000$	998,525

Table 21: Response time results

Server100's budget was calculated with equation (17) Θ^{new} and measured on the Pentium 4 computer (*Table 22*).

Calculated value (µs)	*Measured value* (µs)
$\Theta^{new}_{100} = 8087$	9985

Table 22: Server budget

Next, we measured a taskset of three tasks (τ_1, τ_2, τ_3) belonging to a server of total three servers (*Table 23* and *Table 24*).

Task	*Server*	T (ms)	C (ms)	*Priority* $(0 = highest)$
τ_1	S_3	10000	10	0
τ_2	S_3	10000	20	1
τ_3	S_3	300	20	2
τ_4	S_1	-	∞	0
τ_5	S_2	-	∞	0

Table 23: Task set

Server	Π (ms)	Θ (ms)	*Priority* $(0 = highest)$
S_1	100	40	0
S_2	100	40	1
S_3	100	20	2

Table 24: Server set

We measured and calculated the response time of task τ_3, it's execution trace is illustrated in *Figure 38* below.

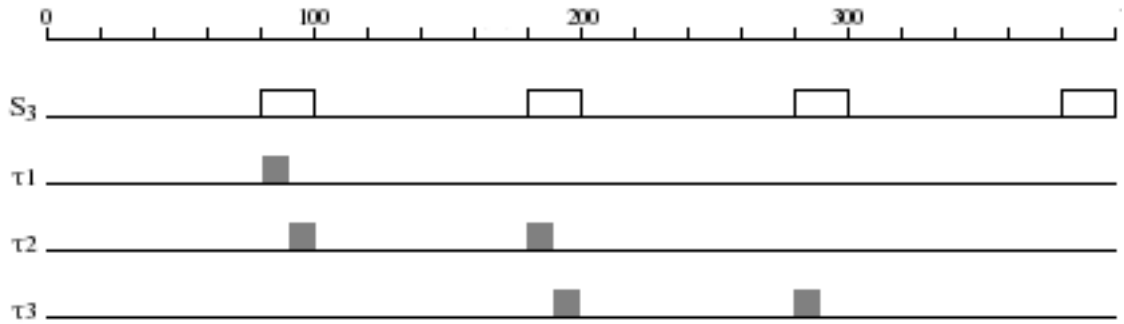

Figure 38: Execution trace of τ_3

Task τ_3's response time was calculated with equation (20) and (21) and the execution of task τ_3's was done on the Pentium 4 computer (*Table 25*).

Calculated value (ms)	*Measured value* (ms)
$r_3^3 = 370,07$	289,695

Table 25: Response time results

The *tbf* formula assumes that the task will start executing right after it's servers budget, this

will add 80 ms to the task response time ($\Pi - \Theta$). This scenario was not present in our test, τ_3 started executing at time 0 which is when the system starts. If the taskset should have been released at time -80 it would theoretically just miss the budget. So adding 80 ms to the test result will match the calculated response time of task τ_3:

$$80 + 289,695 = 369,695\ Ms$$

Server S_1 and S_2 had each a task that had infinite execution time. The test showed that the taskset of server S_3 was not affected by these two erroneous tasksets of servers S_1 and S_2.

Task τ_3's execution trace is included in *Appendix C*.

4.3.3 Jitter

We measured a hierarchical system with the following servers and tasks (*Table 26* and *Table 27*).

Task	*Server*	T (ms)	C (ms)	*Priority* $(0 = highest)$
τ_1	S_3	140	7	0
τ_2	S_3	150	7	1
τ_3	S_3	320	30	2
τ_4	S_1	-	∞	0
τ_5	S_2	-	∞	0

Table 26: Task set

Server	Π (ms)	Θ (ms)	*Priority* $(0 = highest)$
S_1	5	1	0
S_2	6	1	1
S_3	70	20	2

Table 27: Server set

We timestamped the period start, release time and finish time for task τ_3 during 100 period instances. The following tables (*Table 28*, *Table 29*) presents the data of τ_3' jitter.

Mean time (µs)	*Max time* (µs)	*Min time* (µs)	*Absolute release jitter* (µs)
17547	62335	1057	61278

Table 28: Release time

Mean time (µs)	*Max time* (µs)	*Min time* (µs)	*Absolute finishing jitter* (µs)
166406	208945	98664	110281

Table 29: Finish time

4.3.4 Test execution of tasks in delayed and pended state

The last tests were done with tasks that were in delayed and pended state. A requirement of the hierarchical framework is that no task should ever execute during other servers budgets but their own. Also, a task should be able to be in delayed or pended state between it's servers budgets without any side effects. By side effects we mean that it should not execute during other servers budgets and the task should execute normally during the next budget of it's server.
If a task is in delayed (for example sleeping) or pended (for example waiting for a semaphore) state when it's servers budget gets exhausted, that task will get the minimum VxWorks priority (255). If the VxWorks scheduler should by some reason put this task in the ready queue during another servers execution, it will not execute assuming that the tasks in that server has higher priority than 255. When the tasks servers budget gets reloaded and starts executing again, any task that had it's priority lowered will be raised to it's normal priority.

We performed three tests.

• (1) Set a task to sleep and let the VxWorks scheduler wake it up during another servers budget.

• (2) Set a task to sleep and let the VxWorks scheduler wake it up during it's servers next budget.

• (3) Make a task block for a semaphore between two of it's servers budgets.

The last testcase (3) is equal to testcase (2), the motivation for this test is to see that nothing unexpected happens when a task starts blocking waiting for a resource and then gets unblocked during it's servers next budget.

The three test cases (1), (2) and (3) are illustrated in *Figure 39*, *Figure 40* respectively *Figure 41* below. Task τ_1 has higher priority than τ_2 and both tasks belong to server S_1.

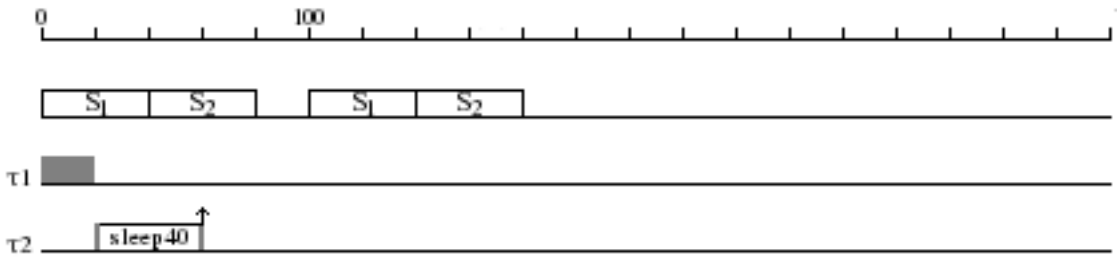

Figure 39: Task wakeup during another servers budget

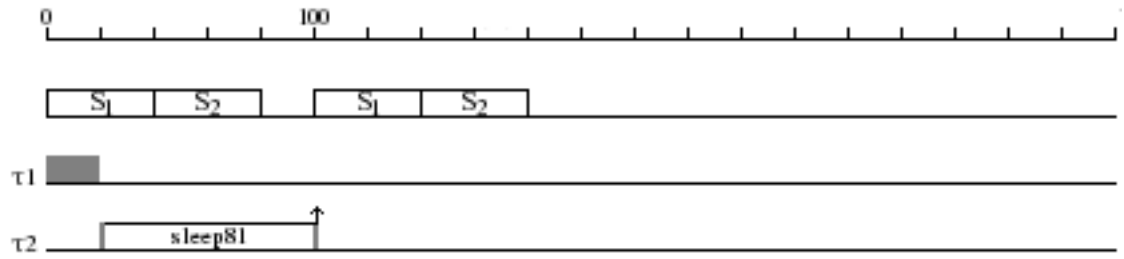

Figure 40: Task wakeup during it's servers budget

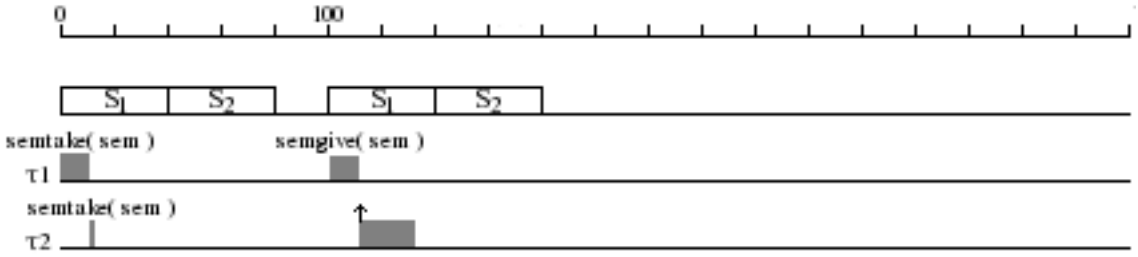

Figure 41: Semaphore blocking

Testcases (1), (2) and (3) was monitored with the Tracealyzer tool, screenshots from all three tests with the Tracealyzer tool can be found in *Appendix C*.

5 Conclusion & Future work

5.1 Conclusion

In this thesis we have presented the design and implementation of a hierarchical scheduling framework.

We have concluded that, in VxWorks, the best way to implement a custom scheduler is to use interrupt handlers connected to some hardware timer. The reason for this is that the VxWorks scheduling framework is well suited for this kind of solution and the implementation will be efficient and easy to implement. The most time efficient implementation is to program directly against memory mapped hardware timer registers. In this way we will not get any interrupt overhead between interrupt expirations, which would be the case with system library solutions.
Another benefit with direct hardware access is that timestamping can be done by just reading the timer register. Libraries do not support any kind of high resolution timing in interrupt routines. Timestamping is a crucial and important function in the HSF. Firstly, the scheduler execution time must be measured and subtracted from server budgets in order to keep the drift down at a low level. Secondly, server preemptions will force the preempted server to calculate the executed budget so far. This is done by reading the current time and subtracting it with the timestamp that the server got when it started to execute. In all, direct access to a hardware timer instead of using VxWorks interrupt libraries is a requirement for the HSF to function efficiently.

System calls to set tasks priorities or to manipulate task status (ready, suspend, etc.) was shown to be inefficient. We experienced interrupt crashes when running many servers or tasks using system calls. Switching to low level routines instead worked fine. Most probably, the system calls were too time consuming to be called from an interrupt handler.
The downside with using low level routines and directly manipulate VxWorks task TCB's in order to change task properties is that it may not be system independent. Newer versions of VxWorks will most likely have the same system call routines but may change low level primitives such as those we used and the task TCB structure.

The single most important part of this thesis, considering having a time efficient implementation, is the time event queue TEQ. Every subsystem has two TEQ's (task periods and deadlines), each server has one budget TEQ and all servers share one TEQ for their periods. The most time consuming part of every HSF interrupt execution, wether it is a task event or server event, is updating the TEQ. The TEQ is used often and by many entities and should thereby be extremely efficient. This was not thought of in the beginning of this project. Looking back now, more work should have been allocated to the design and construction of the TEQ. Our implementation is neither the best or worst. There is no single implementation that is best, the efficiency of the TEQ is dependant on the time properties of servers and tasks.

Finally, we have shown that it is possible to implement a hierarchical scheduling framework on top of VxWorks. The system may at this stage not be optimal considering overhead. However, we know which parts that can be improved. With continued work on these known bottlenecks, the system can be very useful.

5.2 Future work

The work in this thesis can be improved and extended in many ways. Here are a few suggestions for improvements.

As we have shown, interrupt handling and timestamping is most efficient if libraries are not used. These VxWorks libraries are useless for the HSF. The downside is that our system will not be as platform independent. Implementing a new library for interrupts and timestamping for the most used CPU's to go with the HSF may be a good solution. The HSF will be more hardware independent and more time efficient as well.

As we have discussed so far, the TEQ is the single most important module of the HSF because it is the overhead bottleneck. Some research should be done considering alternative efficient solutions for administrating software timers. Also, a notation is that no single implementation is better than all others. Investigations should be done on matching solutions against different types of task and server time properties (periods and deadlines).

We have implemented EDF at both the local and global levels but we have not derived any scheduling overhead formulas for these schedulers. The reason for this was that the original EDF scheduling analysis does not express the number of preemptions as the RTA does. Solving this problem and deriving a scheduling overhead formula for EDF would make this work more complete. Also, it might make our EDF schedulers more attractive to use.

Considering the overhead included in the hierarchical scheduling analysis, we added scheduling overhead caused by preempting servers on the preempted server. The downside is that the development phase of each subsystem will be dependant on other subsystem parameters. A new approach, considering overhead in the hierarchical scheduling analysis, should be derived in order to keep scheduling analysis of each subsystem apart from eachother.

The work so far has not considered shared logical resources. Most, if not all, real-time systems use resource sharing in some way. The system, at this stage, can not by any means guarantee that applications that share logical resources will work correctly. We do not support synchronization protocols such as PCP, SRP, PIP etc. An extension to the HSF could be to look in to how to adapt our system to resource sharing.

References

[1] G. Buttazzo *Hard Real-Time Computing Systems*, University of Pavia, Italy, 2005.

[2] B. Sprunt *Aperiodic Task Scheduling for Real-Time Systems*, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[3] M. González Harbour J.C. Palencia *Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities*, Departamento de Electrónica y Computadores, Universidad de Cantabria, Spain, 2003.

[4] L. Almeida *Response-time analysis and server design for hierarchical scheduling*, DET/IEETA, Universidade de Aveiro, Portugal, 2003.

[5] {G. Lipari, S. Baruah} *A hierarchical extension to the constant bandwidth server framework*, {Scuola Superiore S. Anna, The University of North Carolina}, {Italy, USA}, 2001.

[6] {M. Takashi H. Kei, S. Shigero} *On the Schedulability Conditions on Partial Time Slots*, {Department of Information Science The University of Tokyo, PRESTO Japan Science and Technology Corporation}, Japan, 1999.

[7] {A. Mok X. Feng, D. Chen} *Resource Partition for Real-Time Systems*, {Department of Computer Sciences University of Texas at Austin, Fisher-Rosemount Systems Inc.}, Austin, 2001.

[8] Z. Deng J. Liu J. Sun *A Scheme for Scheduling Hard Real-Time Applications in Open System Environment*, Department of Computer Science, University of Illinois at Urbana-Champaign, USA, 1997.

[9] T. Kuo C. Li *A Fixed-Priority-Driven Open Environment for Real-Time Applications*, Real-Time and Embedded System Laboratory, Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, 1999.

[10] *Kernel Programmer's Guide*, Part #: DOC-16075-ND-00, 14 nov 2007.

[11] J. Gordon *VxWorks Cookbook*,
http://www.bluedonkey.org/cgi-bin/twiki/bin/view/Books/VxWorksCookBook,
15 apr 2003.

[12] *Application api reference 6.6*, Part #: DOC-16103-ND-00, 16 nov 2007.

[13] J. Kraft *Tracealyzer*,
http://www.tracealyzer.se

[14] I. Shin I. Lee *Periodic Resource Model for Compositional Real-Time Guarantees*, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, USA, 2003.

[15] Y. Wang K. Lin *The Implementation of Hierarchical Schedulers in the RED-Linux Scheduling Framework*, Department of Electric aland Computer Engineering, University of California, Irvine, USA, 2000.

[16] *VxWorks 653 – DO-178B Certified ARINC 653 Real-Time Operating System*,
http://www.windriver.com/products/run-time_technologies/Real-Time_Operating_Systems/VxWorks_653

[17] L. Papalau P. Samalik *Design of an Efficient Resource Kernel for Consumer Devices*, Stan Ackermans Institute, Eindhoven University of Technology, Eindhoven, Holland, 13 dec 2000.

[18] {Y. Lee D. Kim, M. Younis J. Zhou} *Partition Scheduling In APEX Runtime Environment for Embedded Avionics Software*, {Real Time Systems Research Laboratory CISE Department University of Florida, Advanced System Technology Group AlliedSignal}, {Gainesville, Columbia}, {USA}, 1998.

[19] G. Lipari E. Bini *Resource Partitioning among Real-Time Applications*, Scuola Superiore S. Anna Pisa, Italy, 2003.

[20] N. Audsley A. Burns M. Richardson A. Wellings *Applying new scheduling theory to static priority pre-emptive scheudling* Software Engineering Journal, 1993.

[21] G. Buttazzo P. Gai *Efficient Implementation of an Edf Scheduler for small Embedded Systems* In Proceedings of the 2nd International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'06) in conjunction with the 18th Euromicro International Conference on Real-Time Systems (ECRTS'08), Dresden, Germany, July 2006.

[22] S. Saewong R. Rajkumar *Hierarchical Reservation Support in Resource Kernels* Real-time and Multimedia Systems Laboratory, Carnegie Mellon University, Pittsburgh, 2001.

[23] R. Davis A. Burns *Hierarchical Fixed Priority Pre-emptive Scheduling* Real-Time Systems Research Group, Department of Computer Science, University of York, UK, 2005.

[24] {M. Behnam T. Nolte I. Shin M. Åsberg, R. Bril} *Towards Hierarchical Scheduling on top of VxWorks* {MRTC Mälardalen University, Technische Universiteit Eindhoven}, {Västerås, Eindhoven}, {Sweden, Holland}, 2008.

6 Appendix A

6.1 A.1 Mean time measurement graphs of local scheduler interrupt routines with non optimized TEQ

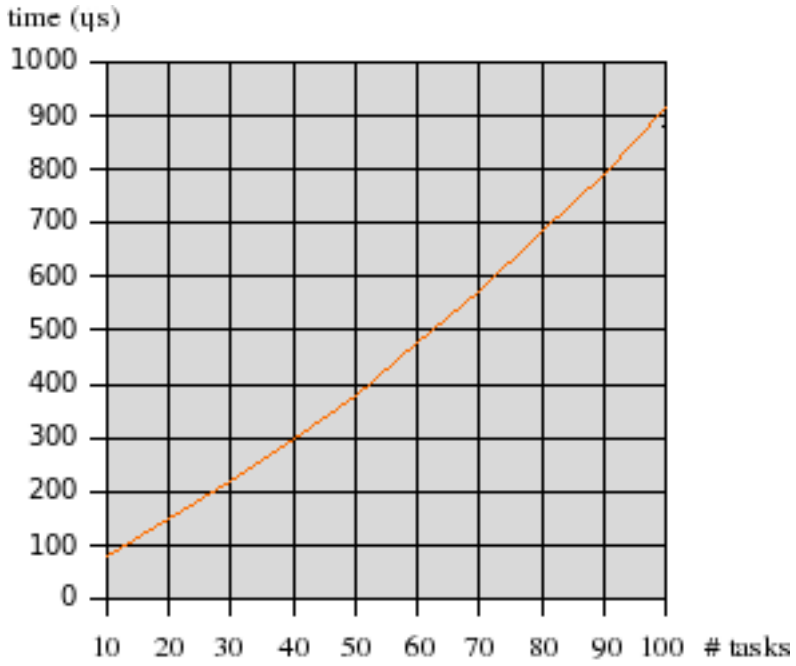

Figure 42: C_{Tint} for FPS
(Regression curve: $f(x) = 90,47 * 1,287^{x/10}$)

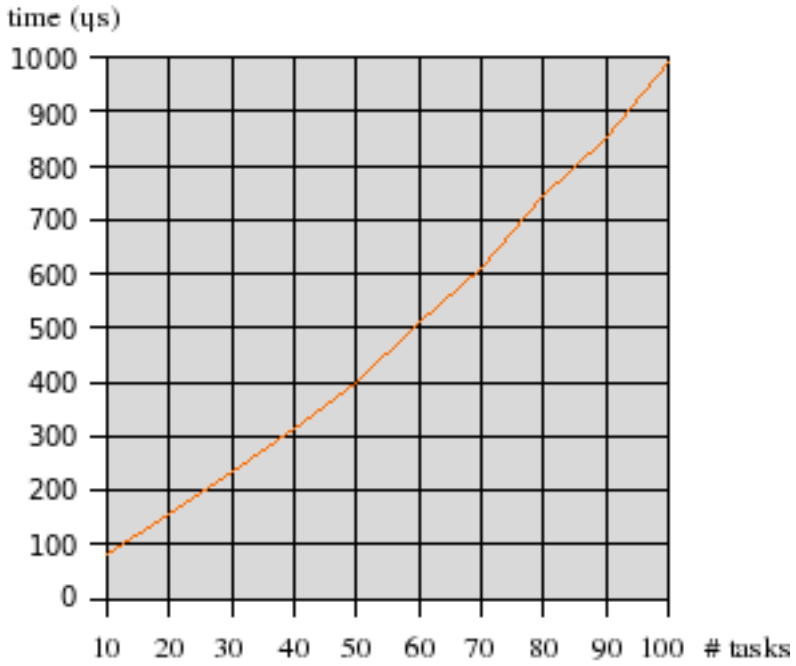

Figure 43: C_{Tint} for EDF
(Regression curve: $f(x) = 94,76 * 1,291^{x/10}$)

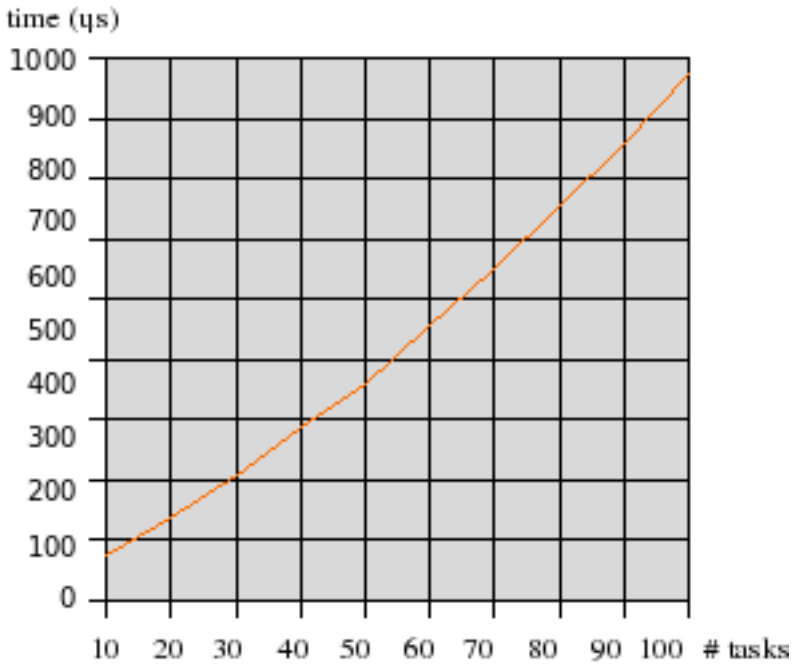

Figure 44: C_{Dint} for FPS
(Regression curve: $f(x) = 85,16 * 1,29^{x/10}$)

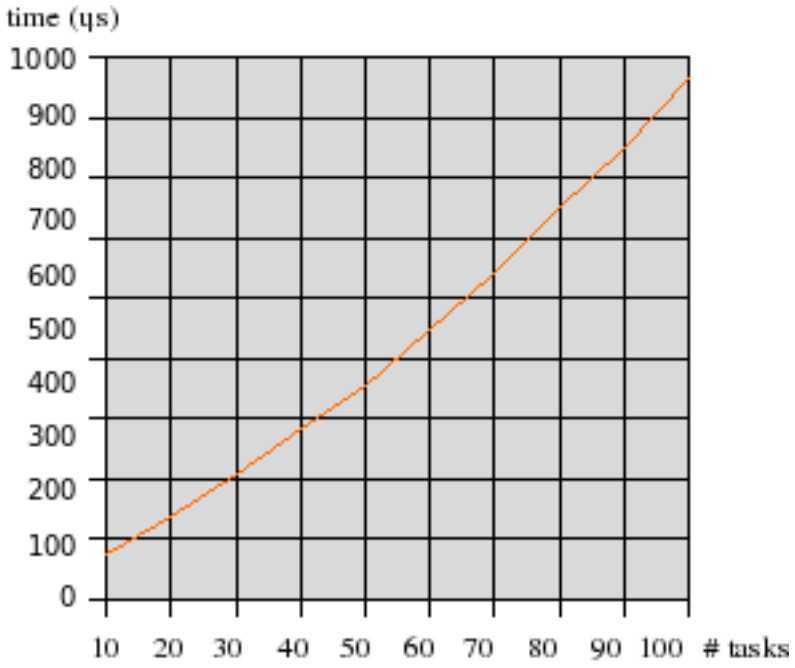

Figure 45: C_{Dint} for EDF
(Regression curve: $f(x) = 83,31 * 1,292^{x/10}$)

6.2 A.2 Mean time measurement graphs of local scheduler interrupt routines with optimized TEQ

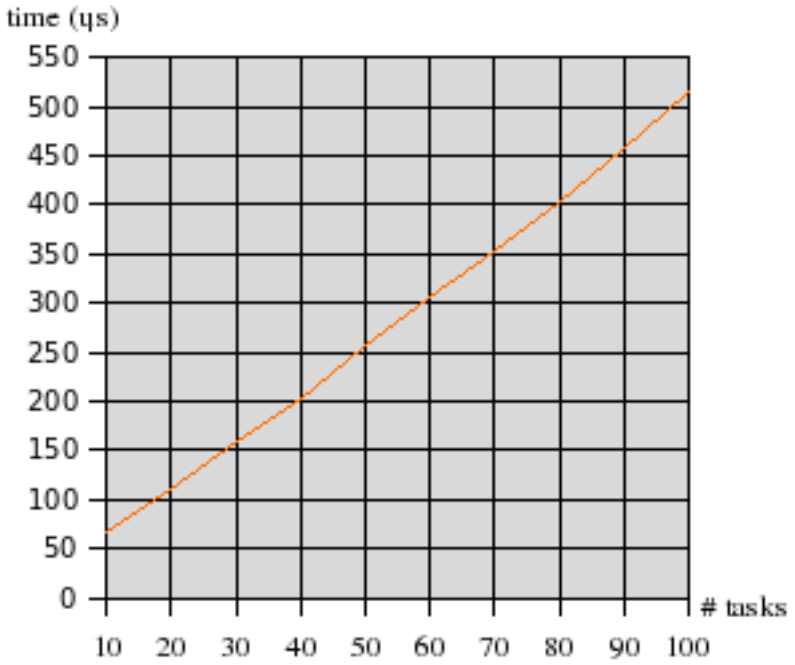

Figure 46: C_{Tint} for FPS
(Regression curve: $f(x) = 49,91x + 8,499$)

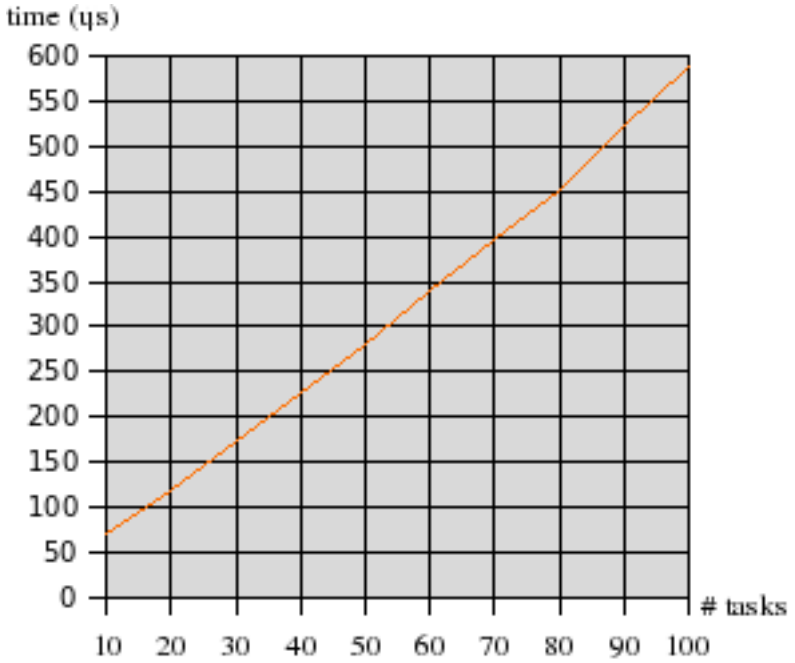

Figure 47: C_{Tint} for EDF
(Regression curve: $f(x) = 57,43x + 1,239$)

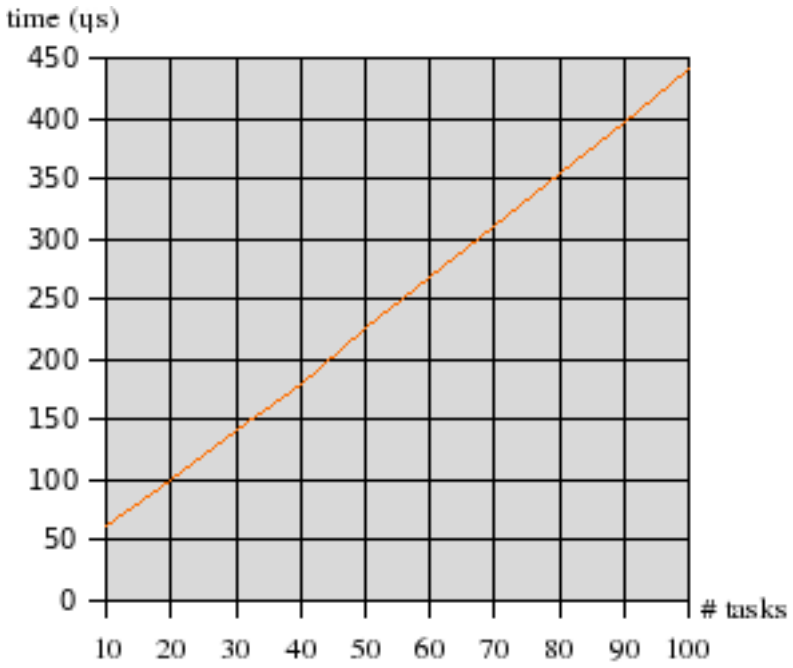

Figure 48: C_{Dint} for FPS
(Regression curve: $f(x) = 42,48x + 14,46$)

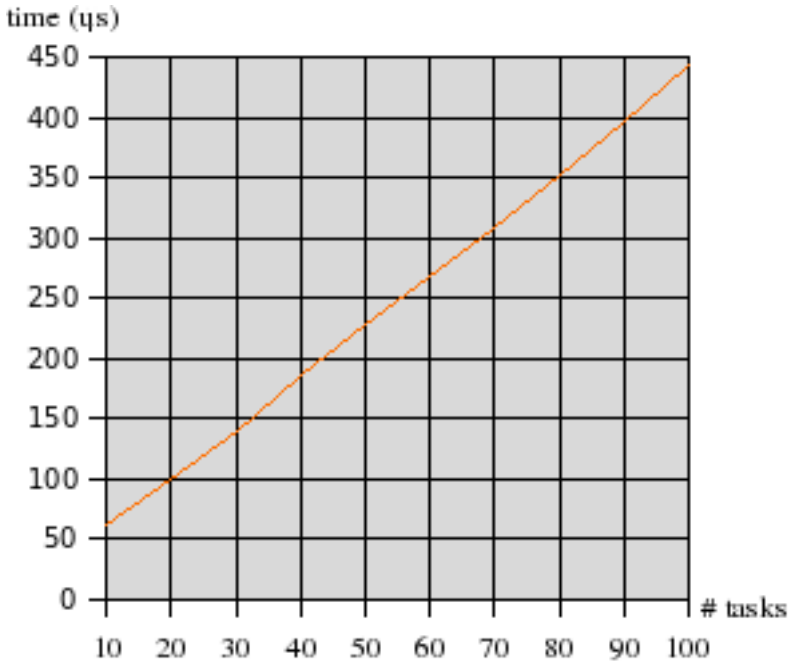

Figure 49: C_{Dint} for EDF
(Regression curve: $f(x) = 42,51x + 14,63$)

6.3 A.3 Mean time measurement graphs of global scheduler interrupt routines with optimized TEQ

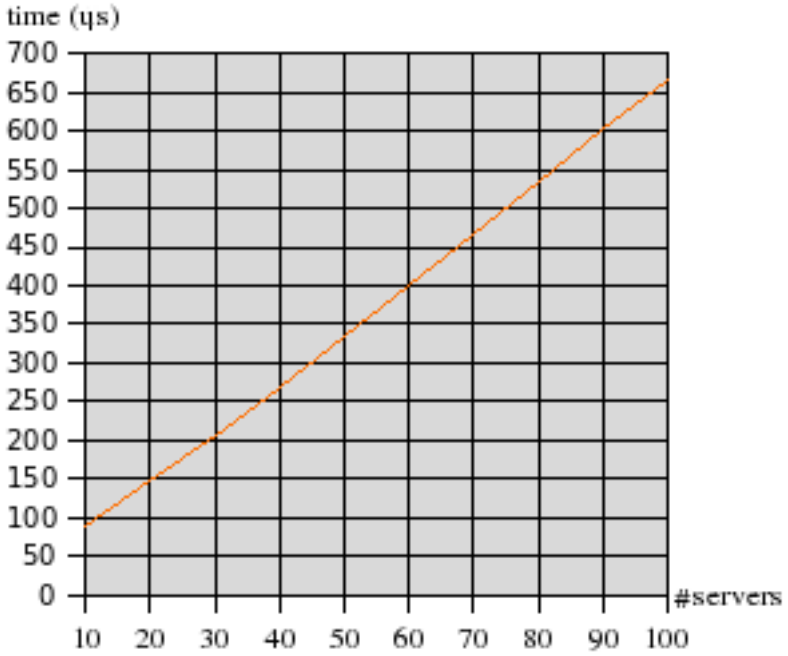

Figure 50: $C_{\Pi int}$ for FPS
(Regression curve: $f(x) = 64,98x + 13,93$)

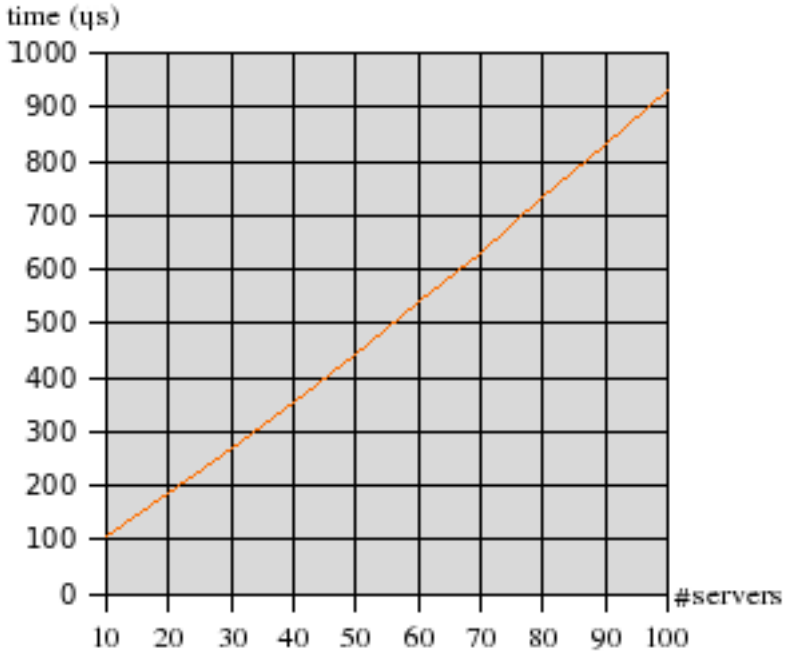

Figure 51: $C_{\Pi int}$ for EDF
(Regression curve: $f(x) = 92,18x - 4,446$)

7 Appendix B

7.1 B.1 Pseudocode for algorithm Count_Double_Hit

```
FUNCTION Count_Doubel_Hit(Interval)
   FOR i = 1  i < NumberOfTasks
      Global := Global UNION Temp
      FOR k = (i + 1)  k < (NumberOfTasks + 1)

         IF Ti = Di AND Tk != Dk
            lcm = temp = LCM(Ti, Tk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR

            lcm = temp = LCM(Ti, Dk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR
         ENDIF

         IF Ti = Di AND Tk = Dk
            lcm = temp = LCM(Ti, Tk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR
         ENDIF

         IF Ti != Di AND Tk != Dk
            lcm = temp = LCM(Ti, Tk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR
```

```
            lcm = temp = LCM(Ti, Dk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR

            lcm = temp = LCM(Di, Tk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR

            lcm = temp = LCM(Di, Dk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR
         ENDIF

         IF Ti != Di AND Tk = Dk
            lcm = temp = LCM(Ti, Tk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR

            lcm = temp = LCM(Di, Tk)
            FOR lcm <= Interval
               IF lcm NOT_BELONG Global
                  Temp := Temp UNION lcm
                  Hit = Hit + 1
               ENDIF
               lcm = lcm + temp
            ENDFOR
         ENDIF

      ENDFOR
   ENDFOR
   RETURN Hit
ENDFUNCTION
```

8 Appendix C

8.1 C.1 Screenshot from Tracealyzer test of the local FPS scheduler

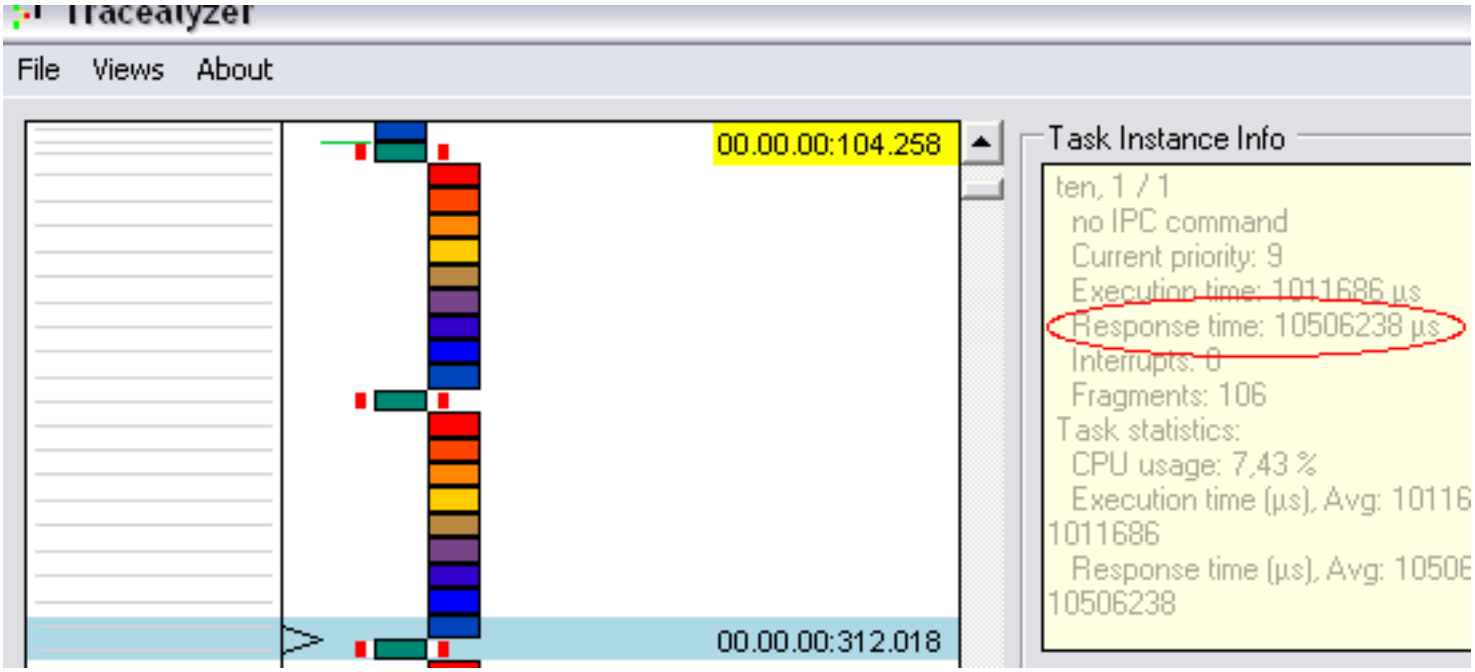

Figure 52: FPS test

8.2 C.2 Screenshot from Tracealyzer test of task executing in a server

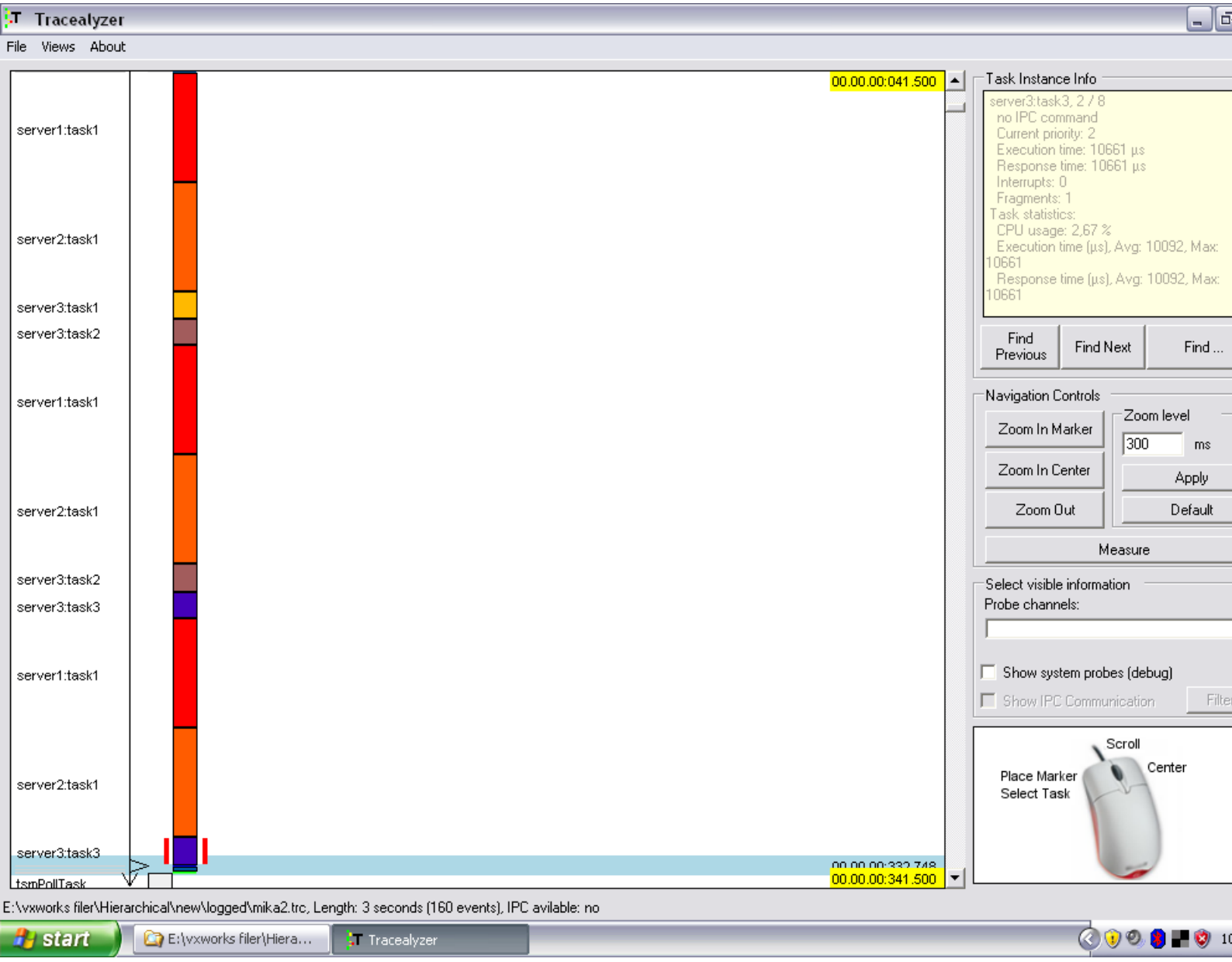

Figure 53: Task3 (server 3) execution trace

8.3 C.3 Screenshot from Tracealyzer test of task in delayed and pended state

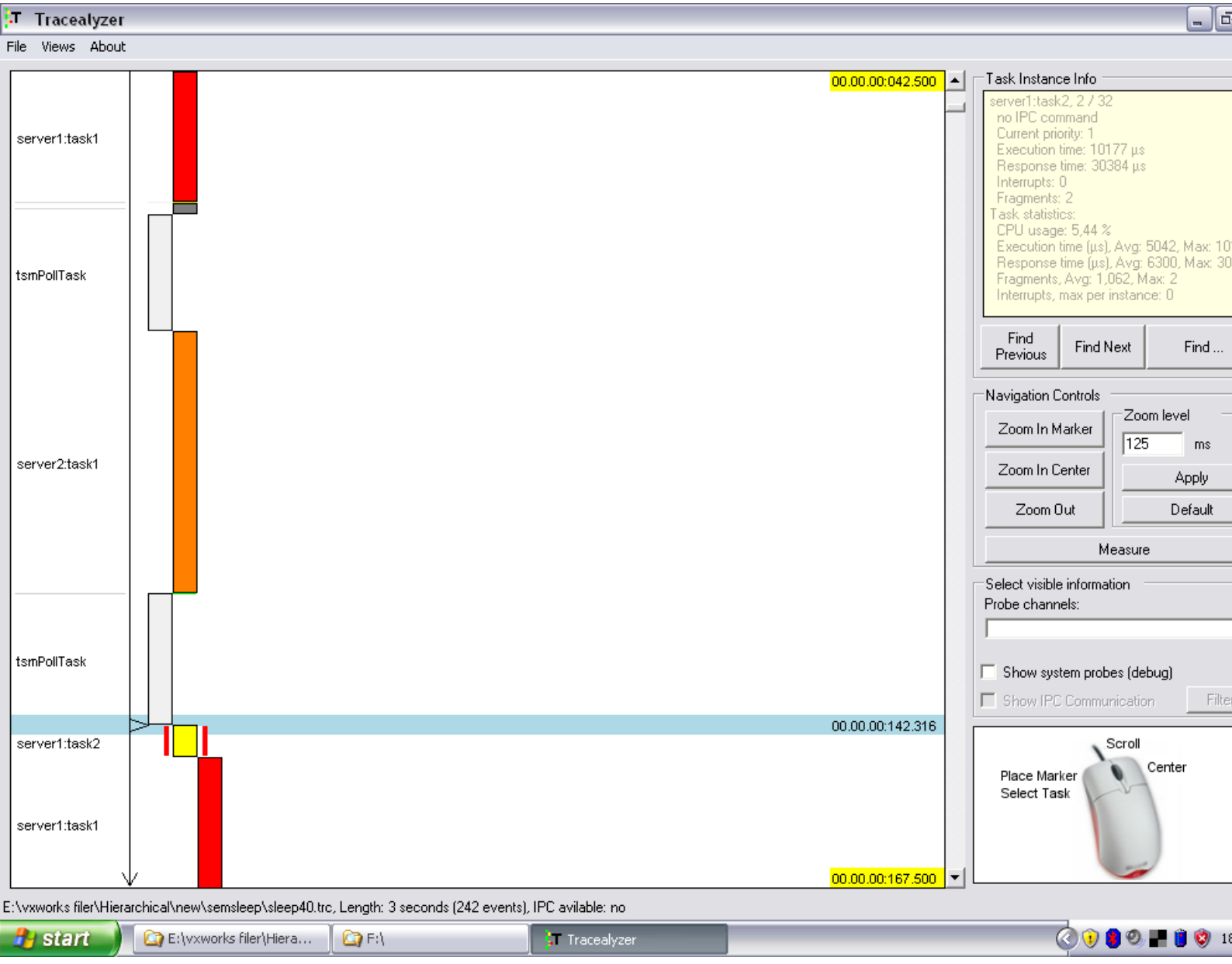

Figure 54: Task sleeps in 40 ticks

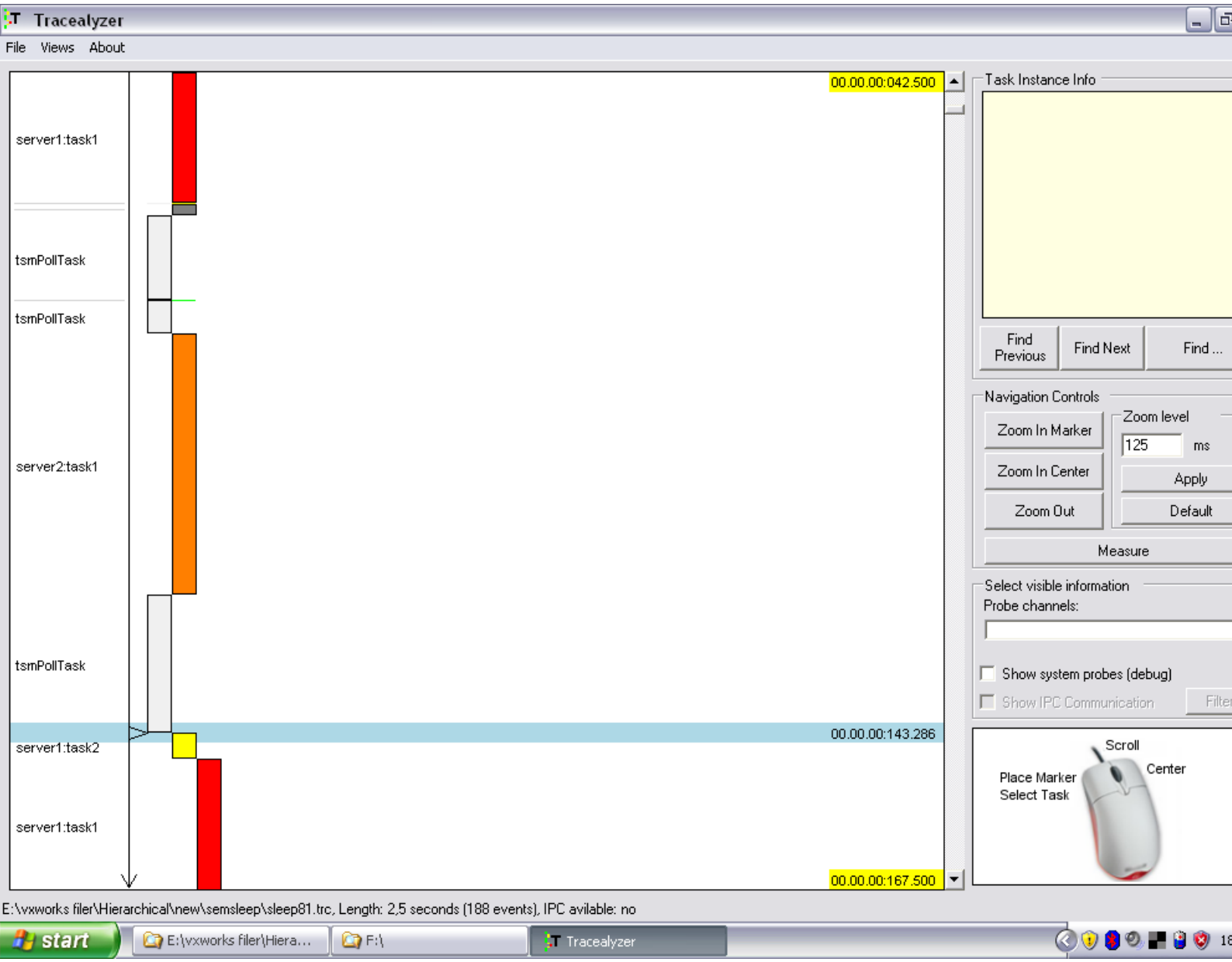

Figure 55: Task sleeps in 81 ticks

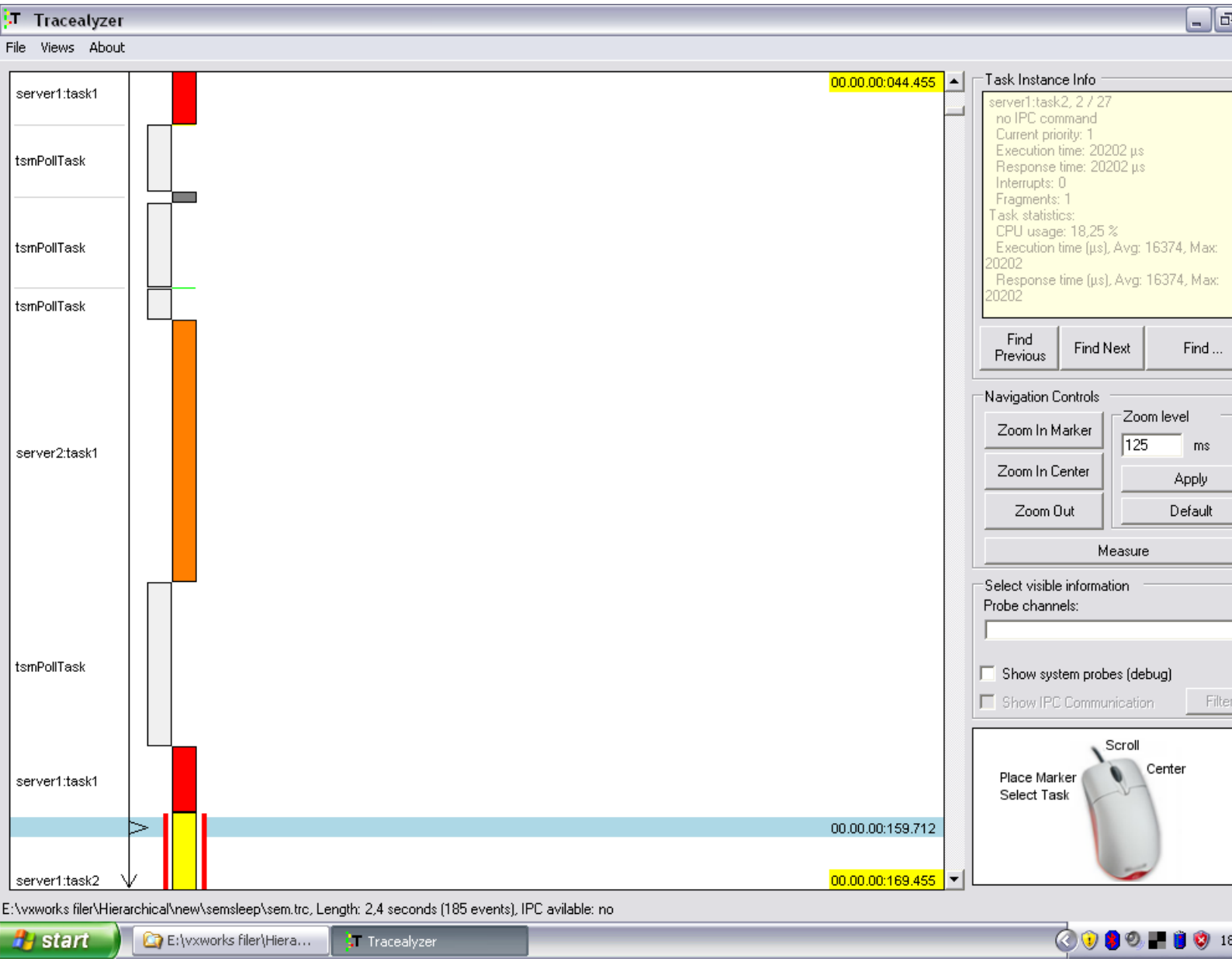

Figure 56: Task gets blocked waiting for a semaphore