IMPLEMENTATION AND PERFORMANCE OF POSIX SPORADIC SERVER SCHEDULING IN RTLINUX

Name: Weijuan Shi
Department: Department of Computer Science
Major Professor: Theodore P. Baker
Degree: Master of Science
Term Degree Awarded: Summer, 2001

Real-time systems may contain periodic and aperiodic tasks with hard or soft deadlines. The Sporadic Server scheduling algorithm, supporting a mix of such tasks, gives a fast average response time for aperiodic tasks while still guaranteeing that periodic tasks meet their hard deadlines. However, Sporadic Server scheduling is not supported on any readily available operating systems, perhaps because of the concerns about complexity and overhead.

This paper describes an implementation of the POSIX standard Sporadic Server scheduling by modifications to the Realtime Linux (RTLinux) operating system. The implementation consists of three key components: capture of activation times, maintenance of pending replenishments, and accounting for execution time.

The implementation was tested, to validate it and to measure its performance. The results indicated there is no need to worry about the complexity and overhead of supporting Sporadic Server scheduling. Additionally, a model of scheduling overhead,

suitable for use in *a priori* schedulability analyses, was fitted to the data from the performance tests.

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

IMPLEMENTATION AND PERFORMANCE OF POSIX SPORADIC SERVER SCHEDULING IN RTLINUX

By

WEIJUAN SHI

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2001

The members of the Committee approve the thesis of Weijuan Shi defended on June, 13th, 2001.

Theodore P. Baker
Professor Directing Thesis

David B. Whalley
Committee Member

Xiuwen Liu
Committee Member

Approved:

Theodore P. Baker, Chairman
Department of Computer Science

To my parents in China ...

ACKNOWLEDGEMENTS

I give my sincerest and heartfelt thanks to my major professor, Dr. Theodore P. Baker, for suggesting the topic of this thesis and for his continuous encouragement and constructive criticism towards the completion of this manuscript. It has been a great opportunity for me to have my thesis conducted under Dr. Baker's supervison. His rigorous and diligent attitude in pursuing research played a role of motivation and model for me to overcome the difficulties I met in this thesis work. Without his great help, I do not think I could have made it.

I also want to express my thanks to Dr. David B. Whalley and Dr. Xiuwen Liu for serving as my committee members and for their helpful comments and advice.

Last, but not least, I want to give my thanks to the whole department and the university for providing me the facilities to finish this master degree program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Real-time systems may contain periodic and aperiodic tasks with hard or soft deadlines. The Sporadic Server scheduling algorithm, supporting a mix of such tasks, gives a fast average response time for aperiodic tasks while still guaranteeing that periodic tasks meet their hard deadlines. However, Sporadic Server scheduling is not supported on any readily available operating systems, perhaps because of the concerns about complexity and overhead.

This paper describes an implementation of the POSIX standard Sporadic Server scheduling by modifications to the Realtime Linux (RTLinux) operating system. The implementation consists of three key components: capture of activation times, maintenance of pending replenishments, and accounting for execution time.

The implementation was tested, to validate it and to measure its performance. The results indicated there is no need to worry about the complexity and overhead of supporting Sporadic Server scheduling. Additionally, a model of scheduling overhead, suitable for use in *a priori* schedulability analyses, was fitted to the data from the performance tests.

CHAPTER 1

INTRODUCTION

1.1 Introduction

The central concerns in the design of real-time systems are timing predictability and verifiability of the satisfaction of timing constraints. Perhaps the most successful and well known approach to addressing these concerns is Rate Monotonic Analysis (RMA), based on the foundational work of Liu and Layland[8] on preemptive scheduling in systems of independent periodic tasks. Several researchers, notably including the team of Sha, Rajkumar, and Lehoczky[14][7][16], extended these ideas to allow schedulability analysis of a wide range of task systems, including systems containing a mix of periodic and aperiodic tasks.

One of the most general scheduling techniques that are amenable to Rate Monotonic Analysis is the Sporadic Server[14]. This technique was originally designed to support the scheduling of a few aperiodic soft-deadline tasks within a system that contains periodic hard-deadline tasks. However, it is sufficiently general that it can also be used to schedule all the tasks in such a system.

The Sporadic Server algorithm is a very good algorithm, that ought to be widely used. The fact that it is not widely used is mainly due to lack of support for it in readily available operating systems. Among the reasons sometimes cited for not supporting this algorithm are that it is complex and would impose undesirable execution time overhead.

This paper describes an implementation of the POSIX Sporadic Server scheduling algorithm for the Realtime Linux (RTLinux) operating system. The goals of this implementation were twofold: (1) to produce a useful freely-available implementation of the Sporadic Server scheduler; (2) to determine whether concerns about the complexity and the overhead of supporting Sporadic Server scheduling are justified.

A substantial amount of testing and measurement was performed on the implementation. The functional testing relates the average response time of the scheduler to that of a simple queueing model. The performance testing provides data that indicates there is no need to worry about the complexity and the overhead is not very significant. Additionally, we have constructed an overhead model fitting the data from the performance tests. The model can be employed in *a priori* schedulability analyses.

1.2 Organization

The rest of this paper is organized as follows: Section 2 introduces some background on real-time systems, Rate Monotonic Analysis, Sporadic Server algorithms and some real-time operating systems. Section 3 explains the design choice of the POSIX Sporadic Server Scheduler and RTLinux operating system. It also gives the specifications of the POSIX Sporadic Server scheduler. Section 4 addresses the technical issues of the implementation. Section 5 discusses the asymptotic overhead analysis of the implementation, design and analysis of functional testing and performance testing, and the construction of the implementation overhead model. Section 6 summarizes the implementation achievements and discusses the possibilities for improvement and ideas for future work.

CHAPTER 2

BACKGROUND

2.1 Real-Time System Scheduling

Real-time systems are used to control physical processes, varying from automobile brake systems to aircraft guidance systems and even nuclear power plants. Tasks in real-time systems are categorized by arrival pattern and deadline, including periodic and/or aperiodic tasks with hard or soft deadlines. The correctness of the systems depends not only on the logical results of computations, but also on the time at which the results are produced. Therefore, a scheduler is needed to coordinate the use of all system resources.

The quality of a scheduling algorithm is judged by how well the algorithm meets the objectives below as given in [14]:

- Guarantee that tasks with hard timing constraints will always meet their deadlines.

- Attain a high degree of schedulable utilization for hard deadline tasks. Schedulable utilization is the degree of resource utilization at or below which all hard deadlines can be guaranteed.

- Provide fast average response times for tasks with soft deadlines.

- Ensure scheduling stability under transient overload. A scheduler is said to be stable if during overload it can guarantee the deadlines of critical tasks even though it is impossible to meet all task deadlines.

On the other hand, a scheduling algorithm relies on the CPU scheduling policy which is a rule to select a task to run next[11]. Some well known scheduling policies are cyclic executive, priority-driven and deadline-driven.

In the following sections, we will discuss some well-known scheduling algorithms in preemptive priority-driven system and explain why we conducted our research on the Sporadic Server algorithm.

2.2 Rate Monotonic Analysis

Rate Monotonic Analysis (RMA) is a well-known and very successful approach to addressing the timing concerns of real-time systems. It is a method for analyzing the *schedulability* [1] of sets of real-time tasks. It originated in Liu and Layland's paper "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment", which appeared in the Journal of the ACM in 1973[8]. At first the theory dealt only with Rate Monotonic Scheduling (RMS). As time went on it was developed to apply to many other scheduling policies. Much of the work of taking RMA from academia into practice was done at the Software Engineering Institute at Carnegie Mellon University.

The main scheduling model of RMA is *Rate Monotonic Scheduling (RMS)* which is one of the priority-driven algorithms. RMS assigns fixed priorities to tasks based on their periods: the shorter the period, the higher the priority. The rate of a task is the inverse of its period. Hence, the higher its rate, the higher its priority. Of the set of tasks that are ready at any time, the one with the shortest period executes before all others.

[1] *Schedulability* is the ability of tasks to meet all hard deadlines

2.2.1 Schedulability Tests

2.2.1.1 Utilization-Bound Test (UB)

The RMA *"Utilization-Bound Test"(UB Test)* [11] which determines whether a set of tasks will be schedulable, requires that the total task utilization be less than or equal to the worst-case utilization bound. The total utilization U of a task set is

$$U = \sum_{i=1}^{N} C_i/T_i$$

where C_i is the execution time of each task i, T_i is the period of task i, and i ranges from 1 to N. The utilization bound, $U(N)$, for a system of N independent periodic tasks is

$$U(N) = N(2^{1/N} - 1)$$

Any set of tasks is schedulable if $U \leq U(N)$. We know if the total utilization U is greater than 1, the system can not be schedulable. Thus, the schedulability of the system with utilization in the range of $U(N)$ and 1 is not decidable using the UB test. In this sense, the UB test is simple but conservative.

2.2.1.2 Response-Time Test (RT)

A more precise but also more complicated test is called the *"Response-Time Test"(RT Test)* [11]. The theorem on which this test is based says that for a set of independent, periodic tasks, if each task meets its first deadline when all tasks are released simultaneously, the deadline will always be met. The response time of task i can be computed as the fixed point of the following recursive formula:

$$a_{n+1} = C_i + \sum_{j=1}^{i-1} \lceil a_n/T_j \rceil * C_j$$

where

$$a_0 = \sum_{j=1}^{i} C_j$$

The fixed point occurs when $a_{n+1} = a_n$. Task i is schedulable if its response time is before its deadline: $a_n \leq T_i$.

2.2.2 An Example

Here is an example with three periodic tasks for the two tests described above:

$$\tau_1 = (3, 0.5), \tau_2 = (6, 1.5), \tau_3 = (12, 5)$$

where each pair consists of the task period and execution time (T_i, C_i).

For the UB test, we have

$$U = 0.5/3 + 1.5/6 + 5/12 \approx 0.834$$

So

$$U(3) \approx 0.779 \leq U \leq 1$$

Therefore the three tasks are not guaranteed to be schedulable by the UB test. However, the first two tasks only are schedulable since their total utilization (0.417) is less than $U(2)$ (≈ 0.828).

For the RT test, since we already know τ_1 and τ_2 can be scheduled from the UB test, we only need to check the schedulability of τ_3. For τ_3,

$$a_0 = 0.5 + 1.5 + 5 = 7,$$

$$a_1 = 5 + \lceil 7/3 \rceil * 0.5 + \lceil 7/6 \rceil * 1.5 = 9.5,$$

$$a_2 = 5 + \lceil 9.5/3 \rceil * 0.5 + \lceil 9.5/6 \rceil * 1.5 = 10,$$

$$a_3 = 5 + \lceil 10/3 \rceil * 0.5 + \lceil 10/6 \rceil * 1.5 = 10.$$

We have $a_3 = a_2$, so the test terminates. Since the final response time $a_3 = 10 \leq C_3 = 12$, τ_3 is schedulable together with the other two tasks, i.e., that entire task system is schedulable.

This example demonstrates that the UB test is simple but conservative and the RT test is more precise but also more complicated.

Since RMA is an analytic approach that can be used before system integration to determine if timing requirements will be met, it can result in significant savings in both system resources and development time. For this thesis, we also applied this technique in the design of tests for the implementation.

2.3 Sporadic Server Algorithms

There has been plenty of theoretical and practical research on the scheduling of periodic task systems. Therefore, we started this thesis project by focusing on scheduling of aperiodic tasks, especially in priority-driven systems.

2.3.1 Bandwidth-Preserving Algorithms

Three commonly used and straight-forward approaches to scheduling aperiodic tasks are background, polling and interrupt-driven executions. Because of the simple nature of those scheduling algorithms, sometimes they may not be able to meet hard deadlines of periodic tasks or offer a fast enough response time for soft deadline aperiodic tasks.

The *bandwidth-preserving* algorithms are an improvement over the polling execution approach. They have the property of allowing aperiodic tasks to be executed with good average-case response time, while preserving adequate bandwidth to guarantee that a set of periodic hard-deadline tasks in the system will still meet their deadlines. The basic idea is to limit the CPU time consumption pattern of the server (the process running the aperiodic tasks) to resemble that of one or more periodic tasks, by imposing an execution time budget. Therefore, each type of bandwidth preserving server is governed by a set of *consumption rules* and *replenishment rules.* The consumption rules define the conditions under which the execution budget is preserved and consumed. The replenishment rules specify when and by how much the budget is replenished.

In Jane Liu's book "Real-Time Systems"[9], a task that behaves more or less like a periodic task and is created (usually of high priority) for the purpose of executing aperiodic jobs is named as a *periodic server*. A periodic server (p_s, e_s) is defined partially by its period p_s and execution time e_s. The parameter e_s is called the *execution budget* (or simply *budget*) of the server. The ratio $u_s = e_s/p_s$ is the *size* of the server. When the budget is replenished by a certain amount, we call that time instant as *replenishment time*.

For a periodic server (p_s, e_s), a minor difference in the consumption rules and replenishment rules may make a significant difference in the resulting scheduling behavior. As a consequence, a variety of bandwidth-preserving servers have been developed for both priority-driven[14] and deadline-driven[6] scheduling disciplines. Each of them has its own strengths and weaknesses.

2.3.2 SpSL Sporadic Server

The *Sporadic Server (SS)*, originally introduced by Sprunt, Sha, and Lehoczky (SpSL)[14], is one of the bandwidth-preserving algorithms for fixed-priority systems. It was an improvement over the two other previously published bandwidth-preserving algorithms, Deferrable Server (DS) and Priority Exchange (PE). It has "an aperiodic response time performance comparable to the DS and PE algorithms, the low implementation complexity of the DS algorithm, and the larger server size of the PE algorithm"[14].

As discussed in [14], like other bandwidth-preserving algorithms, the SS algorithm creates a high-priority task for serving aperiodic tasks. It preserves its server execution time at its high priority level until an aperiodic request occurs. The SS algorithm differs from the DS and PE algorithms in the way it replenishes its server execution time. The DS and PE algorithms periodically replenish their server execution time to full capacity. The SS algorithm only replenishes its server execution time after some or all of the execution time is consumed by aperiodic task execution. This method of replenishing server execution time sets the SS algorithm apart from the DS and PE algorithms and is central to understanding the operation of the SS algorithm, as said in that paper.

In the original SpSL definition of the Sporadic Server algorithm, they introduced the following terms and rules[14]:

> P_s The task priority level at which the system is currently executing.
>
> P_i One of the priority levels in the systems. Priority levels are consecutively numbered in priority order, with P_1 being the highest priority level and P_2 being the next highest.
>
> *active* This term is used to describe a period of time with respect to a particular priority level. A priority level P_i is considered to be active if the current priority of the system, P_s, is equal to or higher than the priority of P_i.
>
> *idle* This term has the opposite meaning of the term *active*. A priority level P_i is *idle* if the current priority of the system P_s, is less than the priority of P_i.
>
> RT_i The replenishment time for priority level P_i. This is the time at which consumed execution time for the Sporadic Server of priority level P_i will be replenished.
>
> The replenishment of consumed execution time for a Sporadic Server executing at priority level P_i proceeds as follows:
>
> If the server has execution time available, the replenishment time, RT_i, is set when priority level P_i becomes active. The value of RT_i is set equal to the current time plus the period of P_i. If the server capacity has been exhausted, the next replenishment time can be set when the server's capacity becomes nonzero and P_i is active. Replenishment of any consumed server execution time is scheduled to occur at RT_i if either the priority level P_i becomes idle or the server's execution time has been exhausted. The amount to be replenished is equal to the amount of server execution time consumed.

2.3.3 Other Sporadic Servers

Since the original publication of the Sporadic Server (SS) algorithm in [14], several variants of the SS algorithm have been developed. Most of the difference is at the above-mentioned "method of replenishing server execution time", the number of pending replenishments allowed at any time, and what the SS should do if no replenishment is allowed to be scheduled. Jane Liu in her book "Real-Time Systems"[9] reported some of the variants, such as the *Simple Sporadic Server*.

The simplicity of the Simple Sporadic Server is that all the budget a server has at any time is replenished at one time. In other words, a Simple Sporadic Server is not allowed to accumulate its execution budget from replenishment to replenishment. An enhancement can be done to the Simple Sporadic Server if we give more budget to the server by allowing the server to keep any budget that remains unconsumed. Therefore, rather than setting the budget to e_s, we increment the budget by e_s units at each replenishment time. Hence the budget may exceed e_s. This enhancement can give a little bit faster response time for aperiodic tasks than the Simple Sporadic Server does.

If we compare the Simple Sporadic Server with the SpSL Sporadic Server, we may observe that the former allows a server to consume and replenish its budget as a whole while the latter allows replenishment in chunks. That also explains SpSL's original design, that is, to preserve unconsumed chunks of budget whenever possible and replenish the consumed chunks as soon as possible. In this sense, we can say a SpSL server with period p_s and execution budget e_s emulates several periodic tasks with the same period and total execution time equal to e_s.

The SS variant we implemented in this thesis is the POSIX Sporadic Server. It is slightly different from the SpSL Sporadic Server. Basically, in the POSIX specification, the Sporadic Server (which is a thread) is switched between its high and low priority levels based on the budget availability and pending replenishments allowance. The consumption and replenishment of budget are processed in chunks. More details on the POSIX SS will be given in the next chapter.

CHAPTER 3

DESIGN

3.1 Design Choice

3.1.1 RTLinux

There are several real-time Linux operating systems that have been published, including Realtime Linux (RTLinux), Real-Time and Embedded (RED) Linux[17], Kansas University Real-Time (KURT) Linux[15] and Open Real-Time Linux (ORiS Linux)[21]. From these, we chose Real-Time Linux (RTLinux) as our implementation platform for several reasons. The scheduler in RTLinux is quite simple and modular. It is easy to replace the scheduler with any other desired one[4], since the code associated with the scheduling is mainly contained in just two files: `rtl_sched.h` and `rtl_sched.c`. These features promised to make the job of adding Sporadic Server scheduling fairly simple. Another important reason is that RTLinux is well known and is becoming a popular platform for real-time applications[25]. Therefore, we had some basis to hope that our implementation would encourage wider use of Sporadic Server scheduling.

3.1.2 POSIX Sporadic Server

As mentioned earlier, since the original publication of the Sporadic Server (SS) algorithm several variants have been developed. For this study of scheduling complexity and overhead, we needed to choose which version(s) to implement. We originally hoped to eventually implement and evaluate several versions.

As discussed in the previous chapter, the Simple Sporadic Server is simple in the sense of there being no accumulation of its execution budget from one replenishment to another. However, on the other hand, to implement the consumption rules and replenishment rules we have to locate the *effective replenishment time* and the beginning and the end of the *server busy interval.* The latter is a time interval that begins when an aperiodic job arrives at an empty aperiodic job queue (the sporadic server is not running and no other aperiodic job is pending) and ends when the queue becomes empty again (the server finishes all pending aperiodic jobs). Keeping track of the beginnings and ends of the server busy intervals at various priorities appears to require a cumbersome stack-like data structure.

After a period of reading and comparison, we decided to start with the SS algorithm defined in the POSIX 1003.1 standard[22]. (This specification is also incorporated into the new draft combined POSIX.1 and Single Unix Specification[23], which is being jointly developed by the IEEE, the Open Group and ISO.) We believe the POSIX SS specification is more understandable than the other published versions and more likely to be widely accepted. Morever, RTLinux developers have been moving the RTLinux API towards a POSIX style.

3.2 RTLinux

Realtime Linux (RTLinux) is a hard real-time kernel add-on to the well-known Linux operating system. It was originally developed by Michael Barabanov[3] at New Mexico Institute of Technology as a master thesis, and now is widely distributed as open-source software. "In RTLinux, a small hard real-time kernel shares one or more processors with standard Linux. This allows the system to run accurately timed applications to perform data acquisition, systems control and robotics, while still serving as a standard Linux workstation."[26] This is attained by scheduling the

RTLinux threads ahead of all the processes and device drivers of the normal Linux operating system.

The RTLinux kernel offers a pure priority-driven scheduler, supporting symmetric multiprocessing (SMP) with static binding of each RTLinux thread to a single CPU. In this scheduler, the ready thread with highest priority is always chosen to run, for each CPU. RTLinux implements the scheduler using a static linked list of all the threads on each CPU, which includes the RTLinux threads, plus one thread per CPU to execute the entire regular Linux operating system and all its processes. Whenever the scheduler is called, it runs through the thread list and wakes up any delayed threads it finds. Among all the threads ready to execute, it then selects the one with the highest priority as the new running thread. Then, the scheduler checks the next nearest in time preemptor of the new thread and sets a timer interrupt to schedule the preemptor.

3.3 POSIX Sporadic Server

3.3.1 Specifications

There are four scheduling policies defined in POSIX:

- `SCHED_FIFO` : Strict preemptive priority-driven scheduling with equal priority threads being served in FIFO order.
- `SCHED_RR` : Strict preemptive priority-driven scheduling with equal priority threads being served in round robin order.
- `SCHED_SPORADIC` : Sporadic Server scheduling.
- `SCHED_OTHER` : Local implementation-dependant scheduling

RTLinux currently supports only the `SCHED_FIFO` policy. Our goal was to extend RTLinux to implement the `SCHED_SPORADIC` policy.

In POSIX, under the `SCHED_SPORADIC` policy, the following members are defined for the scheduling parameters structure:

- `sched_ss_init_budget` This is the initial CPU time budget of the thread. It is also the maximum CPU time that a thread may use within any time interval of the length `sched_ss_repl_period`.

- `sched_ss_repl_period` This is the replenishment period, which defines the separation in time between consumption of a block of CPU time budget and replenishment of the budget.

- `sched_ss_max_repl` This is the maximum number of replenishments that may be pending at any time.

- `sched_priority` This is the "normal" priority of the thread, *i.e.*, the priority at which the thread runs when it has nonzero available budget.

- `sched_ss_low_priority` This is the priority at which the thread runs when it has used up its available budget, or has `sched_ss_max_repl` replenishments pending, until the next replenishment.

The budget consumption and replenishment rules are defined as follows:

> 1. When the thread at the head of the `sched_priority` list becomes a running thread, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation-defined.
> 2. Each time the thread is inserted at the tail of the list associated with `sched_priority` – because as a blocked thread it became runnable with priority `sched_priority` or because a replenishment operation was performed – the time at which this operation is done is posted as the `activation_time`.
> 3. When the running thread with assigned priority equal to `sched_priority` becomes a preempted thread, it becomes the head of the thread list for its priority, and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.
> 4. When the running thread with assigned priority equal to `sched_priority` becomes a blocked thread, the execution time consumed is subtracted from

the available execution capacity, and a replenishment operation is scheduled, as described in 6 and 7.

5. When the running thread with assigned priority equal to `sched_priority` reaches the limit imposed on its execution time, it becomes the tail of the thread list for `sched_ss_low_priority`, the execution time consumed is subtracted from the available execution capacity (which becomes zero), and a replenishment operation is scheduled, as described in 6 and 7.

6. Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, `replenish_amount`, is set equal to the execution time consumed by the thread since the `activation_time`. The replenishment is scheduled to occur at `activation_time` plus `sched_ss_repl_period`. If the scheduled time obtained is before the current time, the replenishment operation is carried out immediately. Several replenishment operations may be pending at the same time, each of which will be serviced at its respective scheduled time. With the above rules, the number of replenishment operations simultaneously pending for a given thread that is scheduled under the sporadic server policy shall not be greater than `sched_ss_max_repl`.

7. A replenishment operation consists of adding the corresponding `replenish_amount` to the available execution capacity at the scheduled time. If as a consequence of this operation, the execution capacity would become larger than `sched_ss_init_budget`, it shall be rounded down to a value equal to `sched_ss_init_budget`. Additionally, if the thread was runnable or running, and had assigned priority equal to `sched_ss_low_priority`, then it becomes the tail of the thread list for `sched_priority`.

In short, in POSIX Sporadic Server scheduling, the server's priority is switched between normal and low priority levels based on the budget availability and pending replenishments allowance; the server is allowed to execute at most the execution budget time; the budget is scheduled to be replenished when it is exhausted or the server is blocked; replenishment is performed at the time which is equal to activation time plus replenishment period; consumption and replenishment are processed in chunks.

3.3.2 Illustration

Fig. 3.1 gives an example of how the POSIX Sporadic Server scheduling policy operating in a system with both periodic tasks and aperiodic tasks.

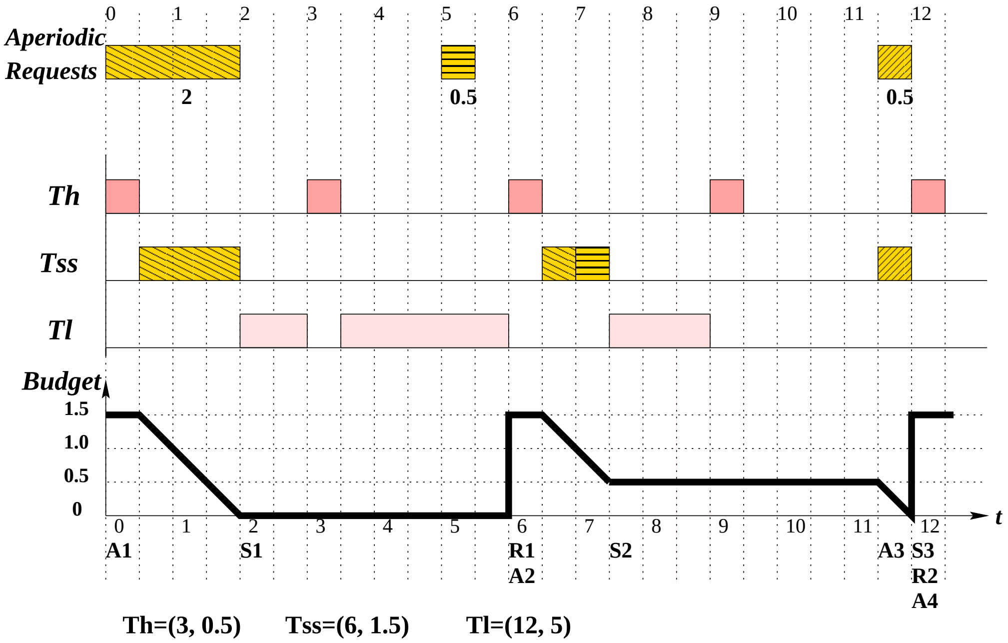

Figure 3.1. Illustration of POSIX Sporadic Server Scheduling

This example has almost the same task parameters as the one we have in the example for RMA tests in Section 2.2.2 except that one of them now is a POSIX sporadic server. Initially the budget of that sporadic server T_{ss} (6, 1.5) is 1.5. It is scheduled rate-monotonically with two periodic tasks: $T_h = (3, 0.5)$ and $T_l = (12, 5)$. Therefore, T_h has the highest priority, T_{ss} the next and T_l the lowest. We already know from Section 2.2.2 that they should be schedulable even when the aperiodic task queue is always busy (i.e., the server is always either executing or pending, never idle). This is referred as "the worst case" for the sporadic server, in which it makes the maximum demand for high priority time.

Without losing generality, we consider the case where the tasks are *in phase* [1]. The top of the figure shows the aperiodic requests coming on the sporadic server. Since the other two tasks are periodic ones, their work load patten is obvious. Therefore, we did noy show those task in the figure. There are three aperiodic requests released

[1] The release time of the first job of a task is called the *phase* of that task. Some tasks are *in phase*, meaning that they have the same phase. [9]

on the sporadic server, at times 0, 5 and 11.5, and work loads (execution times) 2, 0.5, and 0.5, respectively. We call them "aperiodic requests" because in the user (application) code for this example, these executions and sleeps would be scheduled within one thread only. (Of course, we can have more than one thread serving as sporadic servers.) Or put in real-time terms, we have only one aperiodic task, but we have a series of aperiodic jobs (requests) on that task.

In the middle of the figure, the resulting time lines of the three tasks are represented in different patterns (the three aperiodic requests are represented in three different patterns on one time line). Each time line is labeled by its task name on the left and shows the time intervals during which the corresponding task executes.

At the bottom of the figure is the temporal behavior of the sporadic server's budget:

Step 1 : At time 0, an aperiodic request is released but T_{ss} cannot execute because T_h has higher priority. However, according to rule 2, the scheduler sets this moment (A1) as T_{ss}'s activation time.

Step 2 : At time 0.5, T_{ss} begins its execution for the first aperiodic request.

Step 3 : At time 2, T_{ss} runs out of the budget. As to rule 1, T_{ss} is blocked. However, that aperiodic request is not finished yet, having 0.5 units of work to go. According to rule 4, the scheduler lowers T_{ss}'s priority and schedules a replenishment (S1) for T_{ss}. According to rule 6, the replenishment amount is the budget used from the last activation time A1 up to now, which is 1.5. The replenishment is scheduled to occur at time 6 which is activation time 0 plus replenishment period 6. Meanwhile, because T_{ss} in is in lower priority, the lowest priority task T_l begins to execute at time 2.

Step 4 : At time 5, the second aperiodic request is released, but it cannot be served because the sporadic server is blocked at its lower priority. No activation time is set at this time of aperiodic task release.

Step 5 : At time 6, the scheduler performs the replenishment (R1) on T_{ss} as scheduled in step 2. The replenishment will be processed as rule 7 defines. This (scheduled) time instant (which is equal to 6) is set as the new activation time (A2) for T_{ss}. T_{ss} now has budget 1.5. But because of the preemption by T_h it has to wait.

Step 6 : At time 6.5, T_{ss} resumes its execution on the first aperiodic request and finishes it at time 7. Then it continues on the second aperiodic request.

Step 7 : At time 7.5, after finishing the second aperiodic request, T_{ss} is blocked again. Therefore, according to rule 4, the scheduler schedules (S2) a replenishment for T_{ss} with amount 2 and time 12 (R2).

Step 8 : At time 11, the third aperiodic request with execution time 0.5 is released. The scheduler records this time as T_{ss}'s new activation time (A3). Since T_{ss} has budget 0.5 available, it begins to execute.

Step 9 : At time 12, the scheduled replenishment R2 occurs. At the same time the server's the budget is exhausted. Therefore, at this time instant, a replenishment (S3) is scheduled to happen at time 17 (11+6) and the old scheduled replenishment (R2) is performed. Accordingly, another activation time (A4) is set because of this replenishment.

CHAPTER 4

IMPLEMENTATION

4.1 General Consideration

In the implementation, all tasks are considered as POSIX threads. We have seen in the illustrative example of last chapter that there is one sporadic server which deals with several aperiodic requests. The thread behaves in this way: execute (busy) for a while and sleep (idle) for a while. However, the thread (as the task) is always there in the kernel, created at the beginning of the the user application. We have to take this into account because in RTLinux, real-time threads can only be created in RTLinux's `init_module()` function [4] which is like the `main()` function in a regular C program.

In Sporadic Server scheduling, there are several different kinds of aperiodic events coming in as the application runs, such as scheduling of replenishments and the processing of replenishments. A natural solution from programming experience is to have an event queue or timer queue which records all those events and when one event is due, a timer interrupt (with an event type ID) will be triggered. In this way, the scheduler will just handle that event only and switch back to the user application again. In the RTLinux kernel, there is a task queue (thread queue), but no timer queue. Two CPU clock modes are available there: `RTL_CLOCK_MODE_PERIODIC` mode and `RTL_CLOCK_MODE_ONESHOT` mode. The first mode actually serves like a periodic scheduler for the RTLinux scheduler. The RTLinux scheduler is called periodically. Our implementation uses the latter mode, i.e., the scheduler is called by the handler of timer that is set in the scheduler itself. Each time it is called for a certain event, the scheduler goes through the same routine (such as passing through the thread queue, which sometimes may not be necessary at all) and finally sets up the timer for the next event.

4.2 Scheduling Parameters

Structure `sched_param` is one important data structure which can be seen by both the user code and the kernel code. Users use it to configure the scheduling parameters of their application program. The kernel scheduler coordinates the threads based on their scheduling parameters, which are stored in the corresponding thread descriptor. Originally, there is only one field which contains the `sched_priority`. According to the POSIX specifications, the new structure with the name `rtl_sched_param` is defined as follows:

```
typedef struct
{
  int sched_priority;
  int sched_ss_low_priority;
  hrtime_t sched_ss_init_budget;
  hrtime_t sched_ss_repl_period;
  int sched_ss_max_repl;
}  rtl_sched_param;
```

Usually, the value of these parameters will be specialized in user application code when the user creates the server thread. From the user's point of view, they are static (users can change them somewhere in the code, but after they change it, they will be static again). However, inside the kernel scheduler, the priority is a dynamic one, switched between the normal and the lower level. Therefore, internally, in the thread descriptor we needed another field to store the current active priority.

Besides, since we were supporting two policies `SCHED_FIFO` and `SCHED_SPORADIC`, we also added in the field *policy* to the thread descriptor.

4.3 Technical Issues

After studying the rules of the POSIX SS scheduler for a period of time, we observed that the following technical issues needed to be addressed by an implementation:

- how to capture the activation time
- how to maintain the pending replenishments
- how to keep track of the available execution time budget (*i.e.*, how to do CPU execution time accounting)

4.3.1 Capture of Activation Time

In general, the thread of a Sporadic Server can be in one of the following four states: (1) running; (2) runnable, *i.e.* ready; (3) preempted; (4) blocked. Roughly speaking, the events that require scheduling action with the Sporadic Server algorithm are the following:

- a thread is created (*i.e.*, released)
- a thread is completed (*i.e.*, terminated)
- a thread is preempted
- a thread resumes execution from preemption
- a thread is blocked
- a thread is unblocked(*i.e.*, waked up)
- a thread runs out of budget
- a replenishment needs to be performed on a thread

As the POSIX standard defines, the activation time is the moment when (1) the Sporadic Server thread becomes runnable with its normal priority from being blocked or (2) when a replenishment operation is performed on it. Thus, the activation time is associated with two kinds of events: (1) a thread state transition; (2) a replenishment operation. Capturing the latter one is fairly simple: As to the POSIX specification, the first activation time is the time the task is released (so the first replenishment time is that task release time plus the replenishment period). The activation time associated with a later replenishment, would be the time the replenishment is scheduled to occur. That is to say, in processing a replenishment, we set the next activation time to be the time at which the replenishment was scheduled to occur. (In our earlier implementation, we used the current time obtained by reading the CPU clock during the replenishment, as the next activation time. Later we realized that allowed the overhead to be accumulated and would make the scheduler less accurate.) So the main problem becomes how to capture the state transition *to* and *from* being "blocked".

We observed in the RTLinux source code, the most state changes are processed as signals. Hence, the logic governing the transition between blocked and unblocked states is distributed over several different locations within the RTLinux source code. For convenience, we decided to perform all the replenishment-related actions in one place. For that reason, we added a `repl_state` component to the thread descriptor, to indicate what replenishment-related actions need to be performed for that thread the next time when the scheduler is called. The possible states of replenishment are the following:

```
enum rtl_repl_states {
  REPL_NO_ACTION_NEEDED, /* initial state */
  REPL_BLOCK,      /* became blocked recently */
  REPL_UNBLOCK,    /* became unblocked recently */
  REPL_BUDGET_OUT /* budget is zero */
}
```

The state REPL_UNBLOCK indicates that the activation time needs to be set, and the states REPL_BLOCK and REPL_BUDGET_OUT indicate that a replenishment needs to be scheduled. Once the required actions is performed, the state is reset to REPL_NO_ACTION_NEEDED.

4.3.2 Maintenance of Pending Replenishments

As we discussed in Chapter 2, POSIX SS allows execution budget consumption and replenishment in chunks, i.e., there can be more than one pending replenishment at a time and the amounts of the replenishments can vary. That is a strength from the point of view of the response time and CPU utilization, but in implementation, it adds to the complexity.

A mechanism is needed to keep track of the pending replenishments. For this purpose, we allocate an array for each thread at the time of the thread creation. The size of the array is determined by the scheduling parameter sched_ss_max_repl, the maximum number of pending replenishment operations allowed for that thread at any one time. The array is used as a circular FIFO queue. Future replenishment events for the thread are inserted in the queue. When replenishments are performed, they are removed from the queue. Each node of the replenishment queue array contains a data structure for replenishment time and amount, plus a flag field to indicate whether the node is occupied or not. The type definition is as follows:

```
typedef struct rtl_replenish_struct{
  int       scheduled;
  hrtime_t  time;
  hrtime_t  amount;
} RTL_REPLENISH_STRUCT;
```

Accordingly, in the descriptor of the thread we added the following parameters:

- pending_repls   A pointer to the replenishment queue array

- `next_scheduled_repl` A pointer to the next replenishment that has been scheduled, or if there are no replenishments scheduled, the same node as `next_free_repl` (and `scheduled` is set to zero to indicate that the node is marked as free)

- `next_free_repl` A pointer to the next available node for scheduling a replenishment, or if there are none left available, the same node as `next_scheduled_repl` (and `scheduled` is set to one to indicate that the node is scheduled)

Therefore, we have a thread queue, each node of which has a pending replenishment queue shown in Fig. 4.1.

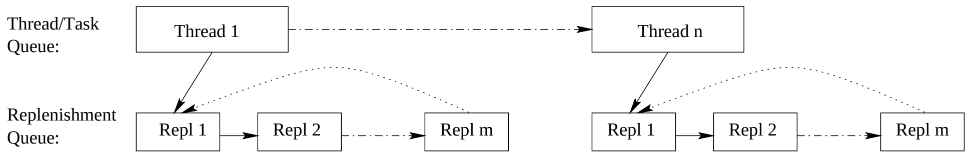

Figure 4.1. Task/Thread Queue with Pending Replenishment Queue

The nodes occupied by scheduled replenishment operations are inserted in a FIFO order. Consequently, when the scheduler looks at the replenishment queue of a thread, it will get the replenishment node with the earliest scheduled replenishment time at the head of the queue. If the replenishment time of the head of the queue is still in the future, the scheduler does not need to check any further for that thread.

4.3.3 Measurement of Execution Time

In order to determine the replenishment amount, the scheduler must keep track of how much execution time each thread has consumed since its last activation time. When the system schedules a replenishment, it can take this value as the amount of the replenishment.

The preemption of a Sporadic Server thread does not make its priority level idle, *i.e.*, the thread is still runnable, which means the consumption of CPU time may be discontinuous. The scheduler should allow the discontinuous execution time to be replenished as a single block for each round of activation, *i.e.*, the scheduler should have a "memory".

So we needed a field along with the owner thread to record the execution time since the activation time. Since this requirement is functionally very close to what is needed to implement the POSIX CPU time accounting feature, we decided to go an extra step to fully implement per-thread CPU time accounting. That is, in addition to recording the amount of CPU time used since its last activation time, we keep track of the total execution time of the thread after its creation (`total_exec_time`).

Adding these to other new fields explained in previous sections, we obtained the following new thread descriptor structure:

```
struct rtl_thread_struct {
  ... fields of original RTLinux ...

  /* added for Sporadic Server */
  int policy;   /* current scheduling policy */
  int priority; /* current scheduling priority */
  hrtime_t budget;
  hrtime_t start_exec_time;
  hrtime_t total_exec_time;
  RTL_REPLENISH_STRUCT *pending_repls;
  int next_sched_repl;
  int next_free_repl;
  hrtime_t activation_time;
  hrtime_t repl_amount;
  enum rtl_repl_states repl_state;
};
```

4.4 Pseudo Code

Most of the implementation work consists of modifications to two files `rtl_sched.h` and `rtl_sched.c`. The modifications are mainly to one function, `rtl_schedule()`. Roughly it has three steps as shown in Fig. 4.2.

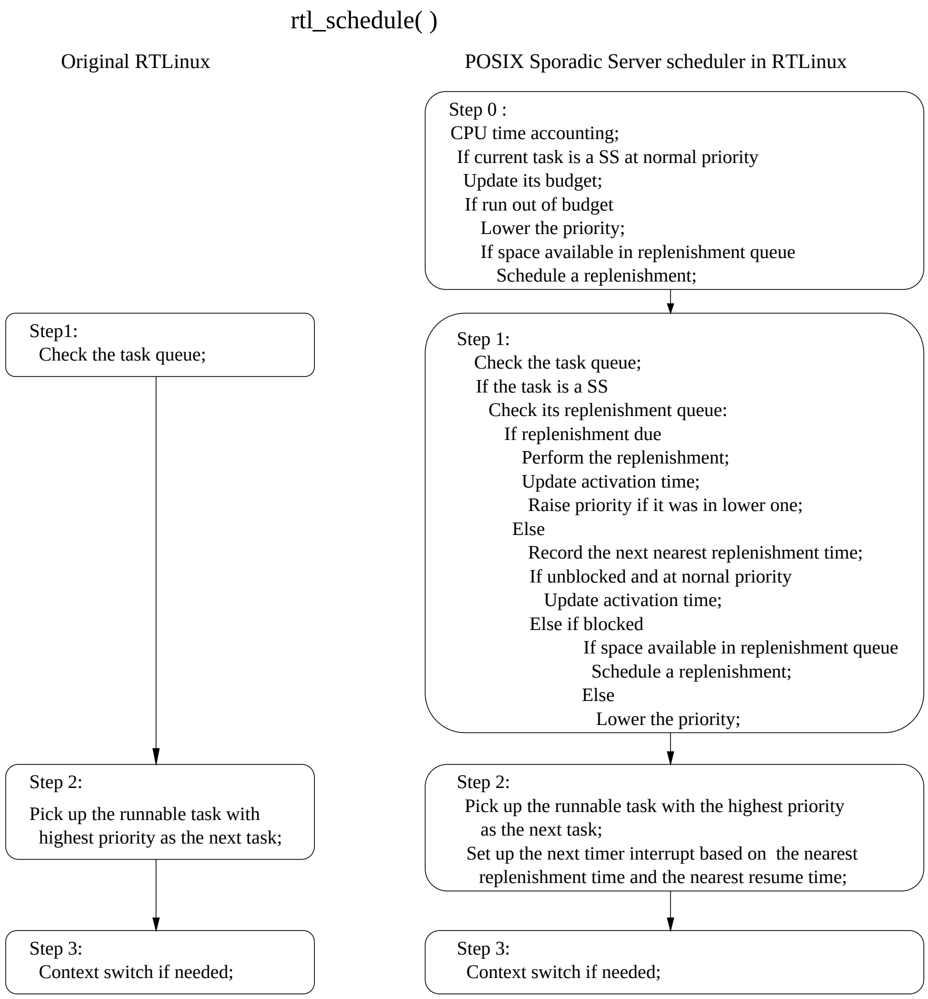

Figure 4.2. Scheduling Function

Below is the pseudo code of our implementation of the scheduling function `rtl_schedule()`, which refers to the POSIX Sporadic Server Scheduler consumption and replenishment rules given in Section 3.3.1:

```
rtl_schedule(){
 CPU time accounting;
 if current thread is a SS running at its normal priority then
   update the budget; (rule 3, 4, and 5)
   if run out of budget then (rule 5)
     lower the priority;
     if space available in replenishment queue then
       schedule a replenishment; (rule 6)

 for each thread in the system{
   check pending signals of thread;
   check timer overrun if thread is delayed;

   if thread is a SS then
     if unblocked and at its normal priority then
       update activation time; (rule 2)
     else if blocked then (rule 4)
            if space available in replenishement queue then
              schedule a replenishment; (rule 6)
            else
              lower the priority;

     for each node in replenishment queue{
       if replenishment time is now or earlier then
         perform replenishment right away; (rule 7)
         update activation time; (rule 2)
         raise the priority if it was at lower priority;
       else
         check the next nearest replenishment time;
     }
   select the highest priority ready thread as the new thread;
 }

 check if the time when the new thread will run out of budget
(if sporadic and at normal priority) is sooner than
the next nearest replenishment time;
 check if the next nearest resume time of a higher priority thread
       (preemptor) is even sooner;
 set the timer for the next time the scheduler needs to be called;
 if the new task is not the current one then
    do context switch;
}
```

CHAPTER 5

PERFORMANCE ANALYSIS AND TESTING

5.1 Asymptotic Overhead Analysis

An asymptotic upper bound on the worst-case overhead of the POSIX Sporadic Server scheduler implementation can be estimated *a priori*. The original RTLinux scheduler makes several passes over the list of all threads, taking $\mathcal{O}(n)$ execution time, where n is the number of all threads in the system. Suppose there are n sporadic server threads in the system and a maximum of m pending replenishments allowed for each thread at any time. One might suppose that all of the replenishments of all the threads could occur and be performed at the same time, in which case the SS scheduler would take time $\mathcal{O}(n * m)$. However, that cannot happen if the overhead is much less than (e.g., less than 10% of) the execution time. That is the only case we are interested in because if the overhead is too significant, the implementation would be not of much use. We observed that the replenishments of each thread are scheduled to occur at different times. Since the timer interrupt is set to call the scheduler whenever the next replenishment is due, there should never be more than one replenishment processed per thread for a single call of the scheduler.

Fig. 5.1 demonstrates one of the worst cases with a sporadic server $(p_s,\ e_s) =$ (10, 4). The dotted line is the budget behavior in theory. The solid line is the actual behavior of the budget under POSIX Sporadic Server scheduling on RTLinux. To make both of the two lines visible, the solid line is moved a little bit higher than it is supposed to be. The small circles are time instants that the scheduler is called because of the state change (such as from execution to sleep) on that server.

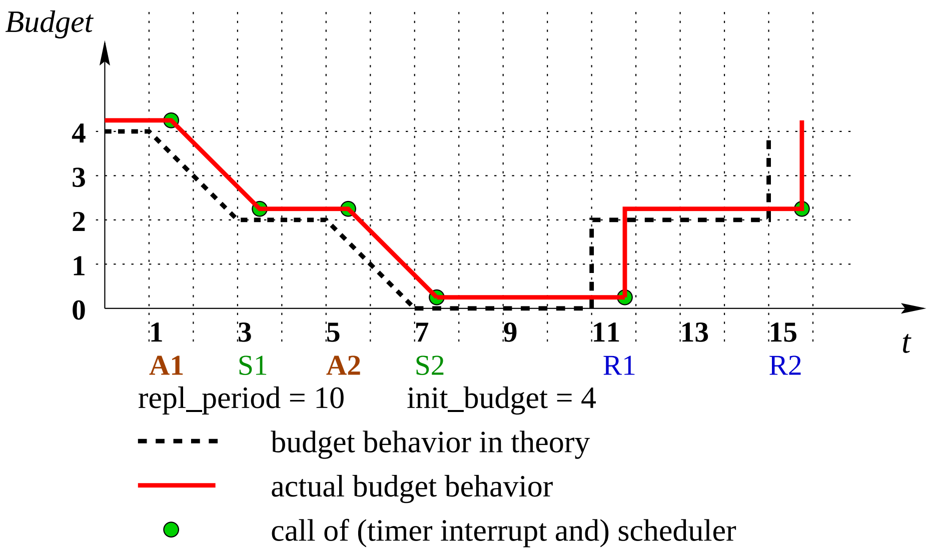

Figure 5.1. Asymptotic Overhead Analysis

Suppose at time 1, an aperiodic request with an execution time of 2 units is released (A1). The server executes 2 units (E) and then it is blocked (e.g., because of no work load). Then a corresponding replenishment (R1) is scheduled (by the scheduler) to be performed at time 11 (S1). After 2 units (S), at time 5, the server resumes (A2). The server continues the execution until time 7 when it runs out of budget and is blocked again. Then another replenishment (R2) is scheduled to be performed at time 15 (S2). For simplicity, we only consider activations A1 and A2, and their replenishments R1 and R2. At time 11 and time 15, as scheduled, a timer interrupt for each replenishment comes and the scheduler is called.

Because of the overhead in implementation, the actual update to the budget is processed slightly later than in theory. Intuitively, one might assume that because of the overhead, the first replenishment R1 might be delayed until the time the second replenishment R2 is processed, therefore the scheduler would at one call need to pass along the replenishment queue twice. However, that is not true. Suppose the average scheduling overhead is ε units. Since that overhead is charged to the thread to which the scheduler will give the CPU after scheduling, the actual execution time E' is equal to $E - \varepsilon$, and the actual sleep time S' is equal to $S - \varepsilon$. Then we have:

$A1 + E + S = A2 \Longrightarrow$

$A1 + (\varepsilon + E') + (\varepsilon + S') = A2 \Longrightarrow$

$A2 - A1 = 2\varepsilon + E' + S' \Longrightarrow$

As said above, R1 and R2 are the scheduled times for replenishments, i.e., $R1 = A1 + p_s$, and $R2 = A2 + p_s$, so we have:

$R2 - R1 = A2 - A1 = 2\varepsilon + E' + S' \Longrightarrow$

$R2 = (R1 + \varepsilon) + (\varepsilon + E' + S') \Longrightarrow$

$R1 + \varepsilon \ll R2$

Since R1 is triggered by the timer interrupt as scheduled, $R1 + \varepsilon$ is the time when the scheduler for that replenishment finishes. From the deduction above, we can see that time is much earlier than the time when R2 is due. Therefore, there cannot be two or more replenishments due in one call of the scheduler (there might be variation in the actual value of ε, but since we assume that ε is a relatively small value compared to the actual execution time E' and the actual sleep time S', it will not have much impact on the overhead evaluation).

Thus, the asymptotic complexity of Sporaidic Server scheduling is no different from that of the original RTLinux scheduler. It still depends on the number of tasks in the system, no matter whether the tasks are scheduled as periodic or aperiodic. Therefore, the overhead of adding support for POSIX Sporadic Server scheduling can be characterized by a constant factor k (the complexity is $\mathcal{O}(n * k)$), which can be estimated by performance testing.

5.2 Debugging and Testing

To validate the implementation, to verify the asymptotic overhead analysis above, and to measure the performance, we spent a substantial amount of time on debugging and testing. The debugging and testing turned out to be very time consuming. Because real-time threads on RTLinux run in kernel memory as kernel modules, it is easy to crash the system, leaving little or no information available at all to identify the bug, and requiring that the system be rebooted before work can resume. Failures may come from: (1) our implementation in the kernel, (2) our testing code as user-level application, (3) the design of the original RTLinux, and (4) the hardware. All these factors add to the complexity of debugging. To aid in debugging, we set up a second computer to act as a console for the other. In this way, we could log and preserve the console output even when the machine being tested crashed.

The implementation, tests and testing environment evolved by trial and error. Things we didn't notice because of partial understanding or even misunderstanding in the earlier stages of the design and implementation were improved as the testing went on. In this fashion, we made the implementation more and more conformant to the POSIX specifications.

The version of RTLinux that we chose for the implementation and performance evaluation is RTLinux Version 2.2, based on Linux Kernel Version 2.2.14. At the time, that was the most recent stable version. The computer we first used for testing has two Intel Pentium processors, since we wanted to be sure that the POSIX Sporadic Server scheduler implementation works on multiple CPUs (SMP) as well as on a single CPU. The CPU speed is 133MHz and the memory is 64MB. Sometimes in testing, especially in performance testing with a high work load (such as 95% CPU utilization), it would give kernel error messages such as "stuck on TLB IPI wait (CPU#0)". Some online Linux documents said that is "a bug triggered by lots of memory and network use"[24]. We then tried different computers with more memory and faster CPU speed. Finally we ended up with a computer which has two Intel 333Hz Pentium II processors and 400MB memory. That solved the problem. All the following results were obtained on that computer.

5.3 Functional Testing

5.3.1 Basic Testing

In this phase of testing, we needed to answer two questions: (1) Does the modified RTLinux kernel still support periodic tasks as the original one does? (2) Does it support aperiodic tasks?

As a user of the kernel scheduler we implemented, we tested the examples released with the RTLinux operating system. They ran perfectly. We also wrote several simple applications which included periodic tasks or aperiodic tasks. They were working fine too. Therefore, both of the answers were "Yes".

5.3.2 Worst Case Testing

In this phase of testing, the question was: Does it still allow periodic tasks to meet their deadlines with aperiodic tasks co-existing in the same system?

The test scheme for this case was based on the proof given by Liu and Layland in their paper of 1973[8] and lecture notes from T.P.Baker's course "Real-Time Systems" in 1998[2]. Essentially, it constructed a worst case system for a set of periodic tasks in which : (1) all tasks are first released at time zero, and time zero is the *critical instant* [1] for all tasks; (2) increasing the execution time or decreasing the period of any task in the set will make the system unschedulable; (3) the utilization of the task set is the minimum of all such systems of tasks. The construction of that task system, i.e. the selection of the worst case periods and execution times, is as follows:

[1] A *critical instant* for a task is defined to be an instant at which a request for that task will have the largest response time. A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks. [9]

Suppose there are n tasks in the system. Given x as the shortest period of those tasks in the system, the period p_i of task i is :

$$p_i = x2^{\frac{i-1}{n}}, i = 1, 2, ..., n$$

The execution time of task i is :

$$e_i = p_{i+1} - p_i, i = 1, 2, ..., n - 1$$

and

$$e_n = p_n - 2\sum_{i=1}^{n-1} e_i = 2p_1 - p_n$$

In theory, all periodic tasks in such a system should meet their deadlines according to Rate Monotonic Scheduling as discussed in Chapter 2. Therefore, in this test case, the priorities of the tasks are assigned according to their periods, i.e., the one with shortest period gets the highest priority.

To answer the question for this test case, we would replace the highest priority periodic task with an aperiodic task served by a sporadic server. The sporadic server's initial budget is the periodic task's execution time. The replenishment period is the periodic task's period. We know a sporadic server has two priority levels. In this case, its normal priority is the same as that of the periodic task. Its lower priority would be lower than the priority of any of the periodic tasks but still higher than that of the whole Linux operating system. We would let the sporadic server try to execute all the time (of course, if the implementation is correct, the sporadic server should be allowed to execute only when it has non-zero budget and is in its normal priority, or when all other periodic tasks are idle and it is in its lower priority). We wanted to see in this case whether all the other periodic tasks (which have lower priorities than the sporadic server) would still make their deadlines.

We chose the number of tasks $n = 5$ and $x = 100{,}000{,}000$ (nanoseconds). The execution times and periods of such a task set are listed in Table 5.1 (all times are in the unit of nanoseconds) and the total utilization is

$$U = \sum_{i=1}^{n} e_i/p_i \;\; = 0.74349177...$$

The schedulable utilization bound for a system of five tasks (see Chapter 2) is

$$U(n) = n(2^{1/n} - 1) = 5(2^{1/5} - 1) = 0.74349177...$$

U is very close to but still less than $U(n)$ (which guarantees it is schedulable in the UB test). In this sense, it is a "worst case".

Table 5.1. Execution Times and Periods of Tasks in Worst Case Testing

task i	p_i	e_i	priority	lower priority
1	100,000,000	14,869,835	10	4
2	114,869,835	17,080,956	9	-
3	131,950,791	19,620,866	8	-
4	151,571,657	22,538,456	7	-
5	174,110,113	25,889,887	6	-

Experiments showed that all periodic tasks, in both the original task set test and the modified task set test (in which the highest priority periodic task is replaced by an aperiodic task/a sporadic server), met their deadlines. Therefore, the implemented scheduler passed the worst case testing, i.e., it does allow periodic tasks to meet their deadline with aperiodic tasks in the same system.

5.3.3 Average Response Time Testing

In this phase of testing, the question was: What is the average response time for aperiodic requests scheduled by the POSIX Sporadic Server Scheduler?

The test scheme is as follows: The system contains only one thread, which is a POSIX sporadic server. The size of that sporadic server (see Chpater 2), which is its execution budget time e_s over its replenishment period p_s, is given as c ($c = \frac{e_s}{p_s}$, $0 < c \leq 1$). That ratio is the maximum CPU utilization for the aperiodic request load, i.e., the service is as if the CPU speed was slowed down by a factor of c. A series of aperiodic requests with random aperiodic interarrival times and random execution times (work load) are generated to be scheduled on this sporadic server. The mean interarrival time and the work load percentage (in the interval of $(0, c)$) are given.

In the test, we let the mean interarrival time of the aperiodic requests be 4 milliseconds. The sporadic server's size, the maximum CPU utilization, is $c = 0.5$. The replenishment period of the sporadic server is the same as the mean aperiodic interarrival time. Therefore, the execution budget of the sporadic server is 2 milliseconds. The interarrival time and execution time of the random aperiodic requests follow an exponential distribution $A(x) = 1 - e^{-\lambda x}$, λ is the mean value of the distribution. Fig. 5.2 gives the result of the average response time plotted as a function of the aperiodic load. Lower values indicates better performance, i.e., a faster response time. The two smooth lines indicate the predicted performance based on queueing models.

The lower smooth line with the label "M/M/1" corresponds to the function

$$\mathcal{L}(\rho) = \frac{\rho}{\lambda(1 - \rho)}$$

where ρ is the mean aperiodic work load percentage (the horizontal axis of the figure), and $\frac{1}{\lambda}$ is the mean aperiodic interarrival time. This is the expected response time for a pure M/M/1 model[5], in which the ratio c is 1, i.e. 100% of the CPU capacity is available for the sporadic server to serve the aperiodic requests.

The higher smooth line with the label "M/M/0.5" corresponds to the function

$$\mathcal{U}(\rho) = \frac{\rho}{\lambda(c - \rho)}$$

where $c = 0.5$. This is the expected response time for a scaled M/M/1 model in which the CPU speed is scaled down to c.

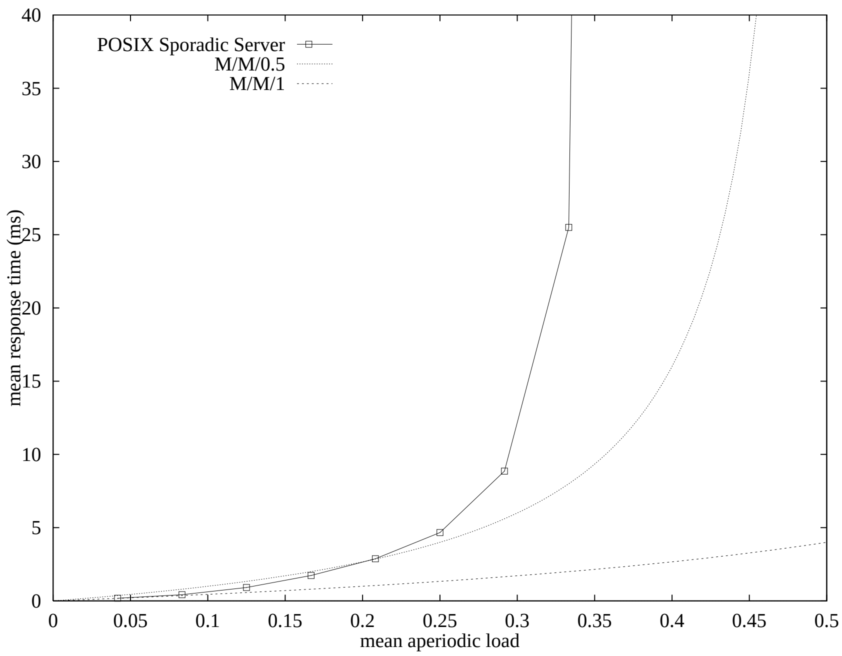

Figure 5.2. Average Response Time Test

The line labled "POSIX Sporadic Server" is the experiment result of the POSIX Sporadic Server scheduler in RTLinux. It has an obvious intersection with the line M/M/0.5. Before that intersection, its average response is lower than that of the M/M/0.5 model and higher than that of the M/M/1 model. That is true because at the low load, the server usually has enough budget for the incoming aperiodic requests to be served right away. It is as if that the server has more than the assigned available CPU utilization (50%), so the response time is quicker than that of the M/M/0.5 model (but still should be no better than that of the M/M/1 model because it can never have more than 100% utilization). After that intersection, the sporadic server's response time grows very quickly as the aperiodic request load increases and the average response time becomes much higher than that of the M/M/0.5 model. That is true because as the load increases, the request waiting queue is getting longer and longer, and there are more and more pending replenishments. That delays the processing of the requests.

5.4 Performance Testing

5.4.1 Test Design

This phase of testing is very important because the main motivation of this thesis is to determine the validity of concerns about the scheduling overhead. Naturaly, one would ask questions like these:

(1) How much is the overhead of the original RTLinux scheduler?

(2) How much is the overhead of the POSIX Sporadic Server scheduler implemented in RTLinux?

(3) Relatively speaking, how much new overhead is added due to the implementation of the POSIX Sporadic Server scheduler ?

As we know, the original RTLinux supports periodic tasks with the policy `SCHED_FIFO`. In our implementation, we added in the support for the policy `SCHED_SPORADIC`. In addition to that, we also added CPU time accounting as mentioned in Section 4.3.3. So the schedulers we have available are:

1. RTLinux (S1)
2. RTLinux + CPU time accounting (S2)
3. RTLinux + CPU time accounting + POSIX Sporadic Server (S3)

Tasks that can execute on some or all of these schedulers are:

1. Periodic tasks (T1)
2. Aperiodic tasks that act like periodic tasks (release/arrival times are periodic, execution times are all the same, but the execution is bound to the budget availability) (T2)
3. True aperiodic tasks (T3)

With the three schedulers and the three types of tasks above, we observed:

(1) We can have T1 executed with policy `SCHED_FIFO` on S1 and S2 . The comparison between them will show the overhead coming from the CPU time accounting.

(2) We can have T1 executed with policy `SCHED_FIFO` on S2 and S3. The comparison between them will show the overhead coming from the Sporadic Server with policy `SCHED_FIFO`.

(3) We can have T1 and T2 executed on S3 (using policy `SCHED_FIFO` and `SCHED_SPORADIC` respectively). Comparison between them will show the overhead coming from the Sporadic Server with policy `SCHED_SPORADIC` with processing on budget and replenishments.

(4) By comparison of the results from (1), (2) and (3), we can estimate the new overhead due to adding support for the POSIX Sporadic Server.

Next, we will discuss two similar but separate test schemes which are to evalute the implementation overhead based on the observations above. Since in those tests, we have no interest in the third type of task T3 (that is because a true aperiodic task can only be scheduled on scheduler S3 with policy `SCHED_SPORADIC` and no comparison can be done), when we say "aperiodic task" in the following tests, we mean the second type of task T2 (aperiodic tasks that act like periodic tasks).

5.4.2 Breakdown Utilization Test

5.4.2.1 Design

This test design idea came from the so-called "Breakdown Utilization Test", originally attributed to Douglas Locke, introduced By T.P.Baker's in his course "Real-Time System" of 2000 [1]. In this test, the test program models the execution of a set of tasks so as to measure the implementation overhead of the underlying scheduling (in our testing, it is one of the three schedulers discussed earlier). The test program repeatedly runs a set of tasks for a time interval. If any of the tasks misses its deadline, the execution times of all tasks are scaled down. Otherwise, the execution times of all tasks are scaled up. The scaling up and scaling down is done in successively smaller steps, so as to converge in a bisection fashion to a threshold at which all hard-deadline tasks are making their deadlines but any further increase in utilization will cause a missed deadline. As mentioned in Chapter 2, Rate Monotonic Analysis (RMA) can be used to predict the utilization bound of a task set. Therefore, comparing the predicted value with the measured value from the test can provide us some information about the implementation overhead of the underlying scheduling (in this case, it is the POSIX Sporadic Server scheduler in RTLinux) and the degree to which the system faithfully implements the Rate Monotonic Scheduling execution model.

Here a problem came up. As we mentioned before, real-time threads can only be created in the very initial function `init_module()` of a kernel module program [4]. But in our testing, each time when the scale factor is changed, we want to rerun the test with the states of all the threads clean as if they were just created (such as zero pending replenishments). Therefore, we had to write some new system APIs which will allow the user application to reset the related fields (such as scheduling parameters) in the thread descriptor in the kernel. In this way, for each round of testing with a new scale factor, we forged a new "fresh" copy of each thread.

Generally in a real-time system, there are more than one task. Therefore, for testing on such as a system, besides the working threads, we also needed another main control thread which would do the test synchronization work, such as letting all tasks release at the same time. Obviously, this thread would impose some overhead, which also added to the complexity of testing. Later on, for simplicity but without losing generality, we tested a system with only one task. In this way, not much synchronization work is needed. That improved the testing accuracy.

5.4.2.2 Result

Table 5.2 gives the result of the "Breakdown Utilization Test" of different types of tasks executed on different schedulers as the three observations described in Section 5.4.1. In each test, there is only one task $(p,\ e) = (100\text{ms},\ 1\text{ms})$, i.e., the period is 100 milliseconds and execution time is 1 millisecond. Therefore, the utilization before scaling is $1/100 = 1\%$. What the table shows are the utilization values after scaling.

Table 5.2. Breakdown Utilization Test Result

Task	RTLinux	RTLinux+CPU accounting	RTLinux+CPU accounting+SS
Periodic	100%	97%	95%
Aperiodic	-	-	95%

From the table, we could see:

- CPU time accounting alone adds 3% (100%-97%) overhead.
- The implementation of the POSIX Sporadic Server scheduler adds 5% (100%-95%) overhead when scheduling periodic tasks.
- The implementation of the POSIX Sporadic Server scheduler adds 5% (100%-95%) overhead when scheduling aperiodic tasks.

There is no significant overhead coming from the scheduling operations on the server's budget and replenishments, as we can see the overheads on both periodic tasks and aperiodic tasks are the same (5%). We concluded from this test scheme that in the term of utilization, the implementation of the POSIX Sporadic Server Scheduler in RTLinux adds on overhead of about 5% of the task period.

5.4.3 Breakdown Budget Test

5.4.3.1 Design

The task set we were most interested in is the above-mentioned second type task: an aperiodic task served on a sporadic server thread simulating a periodic task. In that case, we give `SCHED_SPORADIC` as the thread's scheduling policy. As to the other scheduling parameters: `sched_ss_init_budget` is equal to the peer periodic task's execution time, `sched_ss_repl_period` is equal to the peer periodic task's period, and `sched_ss_max_repl` is equal to 1 (that is because theoretically, in one period, the budget time the server needs is the periodic execution time since the server is simulating a periodic task). This test would be compared with the test in which the thread is given the policy `SCHED_FIFO` running as a periodic task.

The original "Breakdown Utilization Test" works well for periodic tasks. However, as to this test, because of the unknown scheduling overhead (actually that is what we want to know in performance testing), we needed another converging bisection process to set up its appropriate initial budget. We call it as "Breakdown Budget Test". Otherwise, if we give the exact execution time as the initial budget, one could expect that the sporadic server's budget will be always short of the amount of the scheduling overhead time. That is because once the budget is all consumed (by the scheduling overhead and useful execution), it is supposed to be replenished after one replenishment period. Therefore, that task can never meet its deadline with those scheduling parameters. On the other hand, using the "Breakdown Budget Test" , once the budget reaches a value in which the thread can meet its deadline, the scheduling overhead can be considered as difference between the breakdown budget and the initial budget.

5.4.3.2 Result

For this case, we let a POSIX sporadic server (p_s, e_s) schedule an aperiodic tasks. During the test, the minimum workable initial budget e'_s was found between the interval p_s and e_s. Table 5.3 shows the results of two tests, (100ms, 1ms) and (1000ms, 10ms).

Table 5.3. Breakdown Budget Test Result

test id	p_s	e_s	e'_s	$(e'_s - e_s)/e_s$
1	100ms	1ms	1.05ms	5%
2	1000ms	10ms	10.5ms	5%

Therefore, the overhead for the scheduling from this test is also about 5% of the work load. This result is consistent with the result from the "Breakdown Utilization Test".

5.5 Overhead Model

As mentioned in asymptotic overhead analysis (Section 5.1), the implementation overhead within the kernel scheduler for supporting POSIX Sporadic Server can be characterized by a constant factor k and accordingly, the complexity is considered to be $\mathcal{O}(n * k)$ where n is the number of all tasks (periodic one or aperiodic one) in the system. However, the performance testing shows the new overhead does not rely on the number of tasks n very much (i.e., there is no such constant factor k). Instead, the overhead depends on the aperiodic task itself, i.e., if the task with period p_s and execution time e_s scheduled on a sporadic server (p_s , e_s), then the actual initial budget the server needs to make the task schedulable is

$$e'_s = e_s * (1 + 5\%)$$

That is the overhead model. It can be applied when we design a task set to be schedulable. We have to take into account of the extra 5% overhead load when constructing the parameters of execution times and periods of tasks scheduled on the POSIX Sporadic Server Scheduler in RTLinux.

CHAPTER 6

SUMMARY

6.1 Conclusion

In this thesis, several scheduling algorithms for aperiodic tasks in real-time systems have been discussed. Among them, the Sporadic Server algorithm, first introduced by Sprunt, Sha, and Lehoczky[14], distinguished itself by the property of supporting the scheduling of aperiodic soft-deadline tasks within a system that contains periodic hard-deadline tasks and providing a good average response time for aperiodic requests. To determine the validity of concerns about the complexity and the overhead of supporting the Sporadic Server scheduling in a practical operating system, an implementation of the POSIX standard Sporadic Server scheduler in the kernel of the Real-Time Linux (RTLinux) operating system was accomplished in this thesis. The implementation consists of three key components: capture of activation times, maintenance of pending replenishments, and accounting for execution time. A substantial amount of functional and performance testing showed that the implementation exhibits correct behavior and the implementation overhead is tolerable.

6.2 Future work

We have been thinking, since the beginning of the thesis research, about the trade-off of overhead *vs.* accuracy in scheduling. The RTLinux developers are also struggling with that. Therefore, in future work we would be interested in making improvements to the data structures and implementation mechanisms to get better execution efficiency and reduce the overhead. Another interesting subject for further study is the relationship between the maximum number of pending replenishments allowed at any time and the overhead and schedulability. Beyond that, we may implement other variants of the Sporadic Server algorithm or other aperiodic scheduling algorithms and compare them with this implementation, with regard to scheduling overhead and aperiodic request response time.

Meanwhile, the RTLinux operating system is being updated all the time. Therefore, our implementation has to be updated to keep up with the latest version of RTLinux. We have contacted the RTLinux developers for contributing the code and they are glad to have it shared with the RTLinux community.

APPENDIX A

IMPLEMENTATION CODE

A.1 Modification in Header File *rtl_sched.h*

```
enum rtl_task_states {
  RTL_THREAD_READY = 1, RTL_THREAD_DELAYED = 2,
  RTL_THREAD_DORMANT = 4, RTL_THREAD_SUSPENDED = 8,
  RTL_THREAD_ACTIVE = 16, RTL_THREAD_ZOMBIE = 32,
  RTL_THREAD_WAIT_NEEDED = 64 };

enum rtl_repl_states {
  REPL_NO_ACTION_NEEDED = 0, REPL_BLOCK = 1,
  REPL_UNBLOCK = 2, REPL_BUDGET_OUT = 3 };

#define SS_REPL_MAX 1024

/*typedef struct sched_param rtl_sched_param;*/
typedef struct
{
  int sched_priority;
  int sched_ss_low_priority;
  hrtime_t sched_ss_init_budget;
  hrtime_t sched_ss_repl_period;
  int sched_ss_max_repl;
} rtl_sched_param;

typedef struct rtl_replenish_struct{
  int scheduled; /* boolean */
  hrtime_t  time;
  hrtime_t  amount;
} RTL_REPLENISH_STRUCT;

struct rtl_thread_struct {
int *stack; /* hardcoded */
```

```
int uses_fp;
enum rtl_task_states state;
int *stack_bottom;
struct rtl_thread_struct *next;
RTL_FPU_CONTEXT fpu_regs;
int cpu;
hrtime_t resume_time;
hrtime_t period;
void *retval;
int pending_signals;
struct tq_struct free_task;
void *user[4];
int errno_val;
int pad [64];

/* added for SS */
rtl_sched_param sched_param;
int sched_ss_high_priority;
int policy;
int priority;
hrtime_t budget;
hrtime_t start_exec_time;
hrtime_t total_exec_time;
RTL_REPLENISH_STRUCT *pending_repls; /* pointer to the address
of the first element of the replenishment queue, allocated at the
creation of the thread */
int next_sched_repl; /*index for next scheduled repl */
int next_free_repl; /*index of the next available repl*/
hrtime_t activation_time;
hrtime_t repl_amount;
enum rtl_repl_states repl_state;
};

extern inline hrtime_t pthread_get_totalexectime_np(pthread_t thread){
  return thread->total_exec_time;
}

extern int pthread_reset_np(pthread_t thread, hrtime_t budget);

extern int pthread_repl_init_np(pthread_t thread);

extern inline int pthread_setschedparam(pthread_t thread, int policy,
const rtl_sched_param *param) {
  thread->policy = policy;
```

```
  if( param == NULL ) return 0;
  thread->sched_param = *param;
  thread->priority = param->sched_priority;
  if(policy == SCHED_SPORADIC)
  {
    if(param->sched_ss_max_repl > SS_REPL_MAX ||
         param->sched_ss_max_repl < 1 ||
         param->sched_ss_init_budget > param->sched_ss_repl_period )
      return -1;
    if(!thread->pending_repls)
    {
      thread->activation_time=clock_gethrtime(rtl_getschedclock());
      pthread_repl_init_np(thread);
    }
  }
  return 0;
}

extern inline int pthread_getschedparam(pthread_t thread, int *policy,
rtl_sched_param *param) {
  if( param != NULL )    *param = thread->sched_param;
  if( policy != NULL )   *policy = thread->policy;
  return 0;
}

/* should be called AFTER pthread_attr_setschedpolicy() */
extern inline int pthread_attr_setschedparam(pthread_attr_t *attr,
const rtl_sched_param *param){
  if(attr->schedpolicy == SCHED_SPORADIC)
    if(param->sched_ss_max_repl > SS_REPL_MAX ||
       param->sched_ss_max_repl < 1 ||
       param->sched_ss_init_budget > param->sched_ss_repl_period )
      return -1;

  attr->sched_param = *param;
  return 0;
}

extern inline int pthread_attr_getschedpolicy(pthread_attr_t *attr,
int *policy){
   *policy = attr->schedpolicy;
   return 0;
}
```

```
extern inline int pthread_attr_setschedpolicy(pthread_attr_t *attr,
int policy){
   attr->schedpolicy = policy;
   return 0;
}
```

A.2 Modification in Source File *rtl_sched.c*

```
int pthread_wait_np(void)
{
  long interrupt_state;
  schedule_t *s = LOCAL_SCHED;
  rtl_no_interrupts(interrupt_state);
  if (!IS_BLOCKED(s->rtl_current->state))
     s->rtl_current->repl_state = REPL_BLOCK;
    s->rtl_current->state = RTL_THREAD_DELAYED;

  rtl_schedule ();
  rtl_restore_interrupts(interrupt_state);
  return 0;
}

#define CURRENT (sched->rtl_current)
#define SCHED_PRIO sched_param.sched_priority
#define LOW_PRIO sched_param.sched_ss_low_priority
#define IS_BLOCKED(state)\
  ((state == RTL_THREAD_DELAYED) ||\
   (state == RTL_THREAD_SUSPENDED) ||\
   (state == RTL_THREAD_DORMANT) ||\
   (state == RTL_THREAD_ZOMBIE))

/* main scheduling function */
void rtl_schedule (void)
{
  schedule_t *sched;
  RTL_THREAD_STRUCT *t;
  RTL_THREAD_STRUCT *new_task;
  unsigned long interrupt_state;
  int cpu_id = rtl_getcpuid();
  hrtime_t now;

  RTL_REPLENISH_STRUCT  *repl;
  hrtime_t elapsed_time;
  hrtime_t next_schedule_time = HRTIME_INFINITY;
```

```
#ifdef CONFIG_RTL_FAST_RESCHED
  clear_bit (RTL_RESCHEDULE_BIT, &rtl_get_linux_thread(cpu_id)
     ->pending_signals);
#endif
  new_task = rtl_get_linux_thread(cpu_id);
  sched = &rtl_sched[cpu_id];
  rtl_no_interrupts(interrupt_state);
  now = sched->clock->gethrtime(sched->clock);

  /* account for execution time used */
  elapsed_time = now - CURRENT->start_exec_time;
  CURRENT->total_exec_time += elapsed_time;
  /* Check current running task: sporadic server and
     running in its regular(higher) priority .
     Cover case 3,4,5. */
  if (CURRENT->policy == SCHED_SPORADIC
      && CURRENT->priority == CURRENT->SCHED_PRIO)
  {
     /* case 3,4,5: no matter preempted or blocked  */
     CURRENT->repl_amount += elapsed_time;
     /* case 4 is done below when scanning the task list;
        case 5: run out of budget */
    if(CURRENT->budget > elapsed_time)
       CURRENT->budget -= elapsed_time;
    else
    {
       CURRENT->budget = 0;
       CURRENT->priority = CURRENT->LOW_PRIO;
       CURRENT->repl_state = REPL_BUDGET_OUT;
       /* schedule the replenishment */
        repl = CURRENT->pending_repls + CURRENT->next_free_repl;
       if (repl->scheduled == 0)
       { /*  space available for replenishment */
         repl->time = CURRENT->activation_time +
                   CURRENT->sched_param.sched_ss_repl_period;
         repl->amount = CURRENT->repl_amount;
         repl->scheduled = 1;
         CURRENT->next_free_repl = (CURRENT->next_free_repl + 1)
             % CURRENT->sched_param.sched_ss_max_repl;
         CURRENT->repl_amount = 0;
       }
       /* no need to have an 'else' to lower the priority
          because it's in low priority already */
```

```
    }
  } /* end of if(CURRENT->policy == SCHED_SPORADIC &&
       CURRENT->priority == CURRENT->SCHED_PRIO) */

  /* check time overrun,<SS>repl_state,then do replenishment</SS>
     and finally find the highest priority task */
  for (t = sched->rtl_tasks; t; t = t->next) {
    /* handle signals */
    if(t->pending_signals) {  /* do_signal */
      if (test_and_clear_bit(RTL_SIGNAL_SUSPEND, &t->pending_signals)) {
        if (t->policy == SCHED_SPORADIC && !IS_BLOCKED (t->state)
         && t->repl_state != REPL_BUDGET_OUT)
          t->repl_state = REPL_BLOCK;
        t->state = RTL_THREAD_SUSPENDED;
      }
      if(test_and_clear_bit(RTL_SIGNAL_WAKEUP, &t->pending_signals)) {
        if (t->policy == SCHED_SPORADIC && IS_BLOCKED (t->state))
          t->repl_state = REPL_UNBLOCK;
        t->state = RTL_THREAD_READY;
      }
    }
    /* expire timers */
    if(t->state == RTL_THREAD_DELAYED) {
      if (now >= t->resume_time) {
        /* unblock the thread */
        t->state = RTL_THREAD_READY;
        if(t->policy == SCHED_SPORADIC)
          t->repl_state = REPL_UNBLOCK;
        if(t->period != 0) { /* periodic */
          /* schedule the next wakeup time */
          t->resume_time += t->period;
#ifndef CONFIG_RTL_OLD_TIMER_BEHAVIOUR
          while (now >= t->resume_time) {
           /* recover from timer overrun */
           t->resume_time += t->period;
          }
#endif
        } else { /* we don't yet know the next wakeup time */
           t->resume_time = HRTIME_INFINITY;
        }
     }/*end of now >= t->resume_time*/
   }/* end of if (t->state == RTL_THREAD_DELAYED) */

   /*if sporadic server */
```

```
if( t->policy == SCHED_SPORADIC)
{
  /*case 7: do replenishment */
  repl = t->pending_repls + t->next_sched_repl;
  while (repl->scheduled) {
    if(repl->time <= now){/* need to replenish right now*/
      t->activation_time = repl->time; /* case 2b */
      if(!IS_BLOCKED (t->state)&&t->priority == t->LOW_PRIO)
        t->priority = t->SCHED_PRIO;
      t->budget += repl->amount;
      if(t->budget > t->sched_param.sched_ss_init_budget)
        t->budget = t->sched_param.sched_ss_init_budget;
      repl->scheduled = 0;
      t->next_sched_repl = (t->next_sched_repl + 1) %
                   t->sched_param.sched_ss_max_repl;
      repl = t->pending_repls + t->next_sched_repl;
     } else {
       /* the replenishement is not to be performed right
          away, but scheduled only */
         if(next_schedule_time > repl->time)
            next_schedule_time = repl->time;
         break;
         /*there may still be some scheduled replenishment,
          but no more needed to be performed right away*/
     }
  }/*end of while( repl->scheduled ) */

  if (t->repl_state == REPL_UNBLOCK)
  {
    if (t->priority == t->SCHED_PRIO)
       t->activation_time = now;  /* (case 2a) */
  }
  else if(t->repl_state == REPL_BLOCK)/* case 4 */
  {
     /*schedule a replenishment
     -if replenishment time is now or in the past,
      do it right away (merged in case 7 below)
     -if replenishment allows the task to raise priority,
      we need to update that (case 7) and
      update activation_time (case 2b)
     -otherwise, schedule a replenishment in the future*/
     /* case 6 */
     repl = t->pending_repls + t->next_free_repl;
     if (repl->scheduled == 0)
```

```
      { /*  space available for replenishment */
        repl->time = t->activation_time +
             t->sched_param.sched_ss_repl_period;
        repl->amount = t->repl_amount;
        repl->scheduled = 1;
        t->repl_amount = 0;
        t->next_free_repl = (t->next_free_repl + 1) %
               t->sched_param.sched_ss_max_repl;
      }
      else
         t->priority = t->LOW_PRIO;/* lower priority */
    }

   t->repl_state = REPL_NO_ACTION_NEEDED;
 } /* end of if case for t->policy == SCHED_SPORADIC */

 /* and find highest priority runnable task */
 if(t->state == RTL_THREAD_READY &&
   ((!rtl_stop_threads_flag && t != debug_thread) ||
   (rtl_stop_threads_flag && t == debug_thread)) &&
   (t->priority > new_task->priority))
     new_task = t;
} /* end of for(t = sched->rtl_tasks;..; ) */

/*check if next time of running out of budget is
  sooner than next replenishment time*/
if(new_task->policy == SCHED_SPORADIC &&
   new_task->priority == new_task->sched_param.sched_priority
   && next_schedule_time > now + new_task->budget)
     next_schedule_time = now + new_task->budget;

/* check if next resume time for some higher-priority
    periodic tasks is sooner */
if (sched->clock->mode == RTL_CLOCK_MODE_ONESHOT)
{
   RTL_THREAD_STRUCT *tt;
   for (tt = sched->rtl_tasks; tt; tt = tt->next) {
     if ( tt->state == RTL_THREAD_DELAYED )
       /* search for preemptor */
       if (tt->priority > new_task->priority &&
                tt->resume_time < next_schedule_time)
         next_schedule_time = tt->resume_time;
   }
   if(next_schedule_time != HRTIME_INFINITY)
```

```
       {
         sched->timerset = next_schedule_time;
         (sched->clock)->settimer(sched->clock,
                    next_schedule_time - now );
       }
  }

  new_task->start_exec_time = now;
  if (new_task != sched->rtl_current)
  { /* switch out old, switch in new */
      if (new_task ==  &sched->rtl_linux_task) {
        rtl_make_rt_system_idle();
       } else {
           rtl_make_rt_system_active();
       }

      rtl_switch_to(&sched->rtl_current, new_task);
      /*delay switching the FPU context until it is really needed*/
#ifdef CONFIG_RTL_FP_SUPPORT
      if (sched->rtl_current-> uses_fp && sched->rtl_task_fpu_owner !=
          sched->rtl_current)
      {
        if (sched->rtl_task_fpu_owner)
        {
          rtl_fpu_save (sched,sched->rtl_task_fpu_owner);
        }
        rtl_fpu_restore (sched,sched->rtl_current);
        sched->rtl_task_fpu_owner = sched->rtl_current;
      }
#endif CONFIG_RTL_FP_SUPPORT
  }

  rtl_restore_interrupts(interrupt_state);
}

/* initialize the data struct of budget and pending replenishments,
   called by pthread_create() or pthread_setschedparam() only */
int pthread_repl_init_np(pthread_t task)
{
  int len;
  RTL_REPLENISH_STRUCT *repls;

  if( task->policy == SCHED_SPORADIC )
   {
```

```
      task->budget = task->sched_param.sched_ss_init_budget;
      task->repl_amount = 0;
      task->repl_state = REPL_NO_ACTION_NEEDED;
      task->next_sched_repl = 0;
      task->next_free_repl = 0;

      /*init the array for the replenishment*/
      len = task->sched_param.sched_ss_max_repl
            * sizeof(RTL_REPLENISH_STRUCT);
      repls = (RTL_REPLENISH_STRUCT *) kmalloc(len, GFP_KERNEL);
      if(!repls)
      {
          kfree(task->stack_bottom);
          kfree(task);
          return EAGAIN;
      }
      /* initialize pending_repls->scheduled to zero,
         along with the other fields */
      memset(repls, 0, len);
      task->pending_repls = repls;
      return 0;
    }
    return -1;
}

int pthread_reset_np(pthread_t task, hrtime_t budget)
{
  hrtime_t now;
  schedule_t *sched;
  int cpu_id = rtl_getcpuid();
  sched = &rtl_sched[cpu_id];

  now = sched->clock->gethrtime(sched->clock);
  task->pending_signals = 0;
  task->retval = 0;
  task->free_task.next = 0;
  task->free_task.sync = 0;
  task->free_task.data = task;
  task->free_task.routine = rtl_task_free_memory;
  task->cpu = rtl_getcpuid();
  task->resume_time = HRTIME_INFINITY;
  task->period = 0;
  task->state = RTL_THREAD_READY;
  task->priority = task->sched_param.sched_priority;
```

```
  /*task->start_exec_time = now;*/
  task->total_exec_time = 0;

  if(task->policy == SCHED_SPORADIC)
    {
      int len;

      if( budget > task->sched_param.sched_ss_repl_period ) return -1;
      if( budget > 0) task->sched_param.sched_ss_init_budget = budget;

      task->budget = task->sched_param.sched_ss_init_budget;
      task->repl_amount = 0;
      task->repl_state = REPL_NO_ACTION_NEEDED;
      task->activation_time = now;
      task->next_sched_repl = 0;
      task->next_free_repl = 0;
      len = task->sched_param.sched_ss_max_repl
              * sizeof(RTL_REPLENISH_STRUCT);
      memset(task->pending_repls, 0,len);
    }
  rtl_schedule();
  return 0;
}

int pthread_create (pthread_t *thread, pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg)
{
  int *st;
  long interrupt_state;
  RTL_THREAD_STRUCT *task;
  pthread_attr_t default_attr;
  int stack_size;

  hrtime_t now;
  schedule_t *sched;
  int cpu_id = rtl_getcpuid();
  sched = &rtl_sched[cpu_id];
  now = sched->clock->gethrtime(sched->clock);
  if (!attr) {
    pthread_attr_init(&default_attr);
    attr = &default_attr;
  }
#ifdef __SMP__
  if (!rtl_cpu_exists(attr->cpu)) {
```

```
    return EAGAIN;
  }
#endif
  stack_size = attr->stack_size;
  task = (RTL_THREAD_STRUCT *)kmalloc(sizeof(RTL_THREAD_STRUCT),
          GFP_KERNEL);
  if (!task) {
     return EAGAIN;
  }
  st = (int *) kmalloc(stack_size, GFP_KERNEL);
  if (!st) {
    kfree (task);
    return EAGAIN;
  }
  *thread = task;
  task->pending_signals = 0;
  task->retval = 0;
  task->free_task.next = 0;
  task->free_task.sync = 0;
  task->free_task.data = task;
  task->free_task.routine = rtl_task_free_memory;
  task->cpu = attr->cpu;
  task->resume_time = HRTIME_INFINITY;
  task->period = 0;
  task->sched_param = attr->sched_param;
  task->stack_bottom = st;
  task->stack = st + stack_size / sizeof(int);
  task->state = attr->initial_state;
  /*task->state = RTL_THREAD_READY; */
  task->uses_fp = 0;

  /* modified here for SS */
  task->policy = attr->schedpolicy;
  task->priority = task->sched_param.sched_priority;
  task->start_exec_time = now;
  task->total_exec_time = 0;
  if(task->policy == SCHED_SPORADIC)
  {
    task->activation_time = now;
    pthread_repl_init_np(task);
  }
  else
    task->pending_repls = 0;
```

```
  rtl_init_stack(task,start_routine, arg,rtl_startup);
  rtl_no_interrupts(interrupt_state);
  mb();
  add_to_task_list(task);
  mb();
  rtl_schedule();
  rtl_restore_interrupts(interrupt_state);
  return 0;
}
```

APPENDIX B

TESTING CODE SAMPLE

B.1 Breakdown Utilization/Budget Test

```
#include <rtl.h>
#include <time.h>
#include <rtl_fifo.h>
#include <rtl_sched_ss.h>
#include <rtl_sync.h>
#include <unistd.h>
#include <rtl_debug.h>
#include <rtl_mutex.h>

struct MY_TASK
{
  int policy;
  int priority;
  int low_priority;
  hrtime_t computation_time;
  hrtime_t init_budget;
  hrtime_t period;
  hrtime_t repl_period;
  int max_repl;
};
struct MY_TASK my_task = {SCHED_SPORADIC, 5, -3,1000000,1000000,
100000000,100000000, 1} ;

#define Iterations              100
#define Loops                   1
#define Second 1000000000
#define Real_Accuracy 100
#define Accuracy  1000

hrtime_t  Seconds_To_Run;
pthread_t main_thread, thread;
```

```
volatile int Missed_Deadlines=0, done_all=0,times=0;
int             otimes=0, utimes=0;
hrtime_t Hi, Lo, Load_Factor=1,Computation_Load;
hrtime_t   start_time,stop_time,total_time=0,total_comp_time=0,
                Unit_Work_Execution_Time, Total_Utilization,
Deadline_Check_Time;
hrtime_t        last_total_time=0, last_total_comp_time=0;
int             last_times=0;
hrtime_t  t1, t2, delay, bound, base, start, stop;
int  a[2] = {0, 1}; /* for Do_Unit_Work() */

void Do_Unit_Work(long long dummy);
void estimate_resolution_bound(void);
void dual_loop_benchmark(void);
void Compute_Loads(unsigned int load_factor);
void *Periodic_Task(void *arg);
int  Run_Tasks(void);
void compute_utilization(int scaled);
void *Find_Utilization_Limit(void *arg);
void print_with_time(const char msg[], hrtime_t time);
void Set_Budgets (unsigned int load_factor);
void test_budget_and_load(void);

void Do_Unit_Work(long long dummy)
{
  int i,t,j;
  i = dummy - dummy / 2 * 2;
  t = a[i];
  j = (i + 1) - (i + 1) / 2 * 2;
  a[i] = a[j];
  a[j] = t;
  if( a[1] > 1 ){
      rtl_printf("Should never happen");   }
} /* Do_Unit_Work( int ) */

void estimate_resolution_bound()
{
  int i;
  bound = 100000;
  for(i = 0; i < 1000; i++)
    {
      t1 = clock_gethrtime(rtl_getschedclock());
      do{
         t2 = clock_gethrtime(rtl_getschedclock());
```

```
        delay = t2 - t1;
        if( delay > 0 ) break;
      } while ( 1 );
      if( delay < bound ) bound = delay;
    }
} /* estimate_resolution_bound() */

void dual_loop_benchmark()
{
  long long j, k = 10000;
  hrtime_t start, stop;
  estimate_resolution_bound();
  do{
      t1 = clock_gethrtime(rtl_getschedclock());
      for( j = 0; j < k; j++ ) Do_Unit_Work(j);
      t2 = clock_gethrtime(rtl_getschedclock());
      base = t2 - t1;
      if( base >  (Real_Accuracy * bound) )  break;
      if( k * 10 < k ) {
        rtl_printf("k overflows dual loop\n");
        break;
      }
      k = k * 10;
  } while ( 1 );
  Unit_Work_Execution_Time = base / k;
} /* dual_loop_benchmark() */

void Compute_Loads(unsigned int load_factor)
{
  Computation_Load = ( load_factor * my_task.computation_time )
                     / Unit_Work_Execution_Time;
  if( (load_factor *  my_task.computation_time  - Computation_Load *
       Unit_Work_Execution_Time) >= ( Unit_Work_Execution_Time / 2 ))
    Computation_Load ++;
} /* Compute_Loads() */

void * Periodic_Task(void *arg) /* for each task */
{
  long long i;
  hrtime_t wakeup_time;
  while( !done_all ) {
    wakeup_time = start_time;
    if( pthread_make_periodic_np(thread, start_time, my_task.period) )
      return 0;
```

```
    while( 1 ) {
      pthread_wait_np();
      for(i = 0; i < Computation_Load; i++)
        Do_Unit_Work(i);
      t2 = clock_gethrtime(rtl_getschedclock());
      if (t2 > (wakeup_time + my_task.period) ) {
        Missed_Deadlines = 1;
        break; }
      times ++;
      wakeup_time += my_task.period;
      if (wakeup_time >= stop_time)
         break;
    }/* while (1) */
    pthread_wakeup_np(main_thread);
  }  /* end of while ( !done_all ) */
  return 0;
} /* Periodic_Task() */

int Run_Tasks( ) /* return 0 or 1 to mean missed deadlines or not */
{
  Missed_Deadlines = 0;
  times = 0;
  start_time = clock_gethrtime(rtl_getschedclock()) + (hrtime_t)Second;
  stop_time = start_time + (hrtime_t) (Seconds_To_Run * Second) ;
  /*rtl_printf("main thread suspended\n");*/
  pthread_suspend_np( main_thread );
  total_time = pthread_get_totalexectime_np(thread);
  return (!Missed_Deadlines);
} /* Run_Tasks() */

void test_budget_and_load(void) {
  hrtime_t Lb, Hb, Budget;
  Hb = my_task.repl_period;
  Lb = Hb;

  /* finding initial budget */
  Compute_Loads(1);
  if(my_task.policy == SCHED_SPORADIC)
  {
    otimes =0;
    utimes =0;
    rtl_printf(" -initializing bound of budgets-\n");
    do {
      pthread_reset_np(thread, Lb);
```

```
  if( Run_Tasks() ) {
    utimes ++;
    otimes = 0;
    if( utimes < Loops + 3) continue;
    rtl_printf(" -underloaded at budget %ld-\n",Lb);
    utimes = 0;
    last_times = times;
    last_total_time = total_time;
    Hb = Lb;
    Lb = Lb / 16;
    if( Lb <= 0 )
    {
      Lb = 0;
      rtl_printf("budget reaches zero!\n");
      break;
    }
  }
  else {
    otimes ++;
    utimes = 0;
    if(otimes < Loops) continue;
    rtl_printf(" -overloaded at budget %ld-\n",Lb);
    break;
  }
} while ( 1 );

/* bisection for limiting budget */
otimes = 0;
utimes = 0;
rtl_printf(" -using bisection to find limiting budget-\n");
do {
  Budget = (Lb + Hb) / 2;
  pthread_reset_np( thread, Budget);
  if( Run_Tasks() ) {
    otimes = 0;
    utimes ++;
    if( utimes < Loops + 3) continue;
    rtl_printf(" -underloaded at budget %ld-\n", Budget);
    utimes = 0;
    last_times = times;
    last_total_time = total_time;
    Hb = Budget;
  }
  else {
```

```
      otimes ++;
      utimes = 0;
      if( otimes < Loops ) continue;
      rtl_printf(" -overloaded at budget %ld-\n", Budget);
      otimes = 0;
      Lb = Budget;
   }
  } while ( (Hb - Lb) > Budget/Accuracy );
  if( Budget < Hb )
    {
      Budget = Hb;
      times = last_times;
      total_time = last_total_time;
    }
  rtl_printf(" -limiting budget: %ld-\n", Budget);
  if(times != 0 && total_time !=0 )
    {
      if(total_time >= times * (Computation_Load *
         Unit_Work_Execution_Time))
        print_with_time("at init_budget, overhead per iteration:",
          (total_time -  times * Computation_Load *
          Unit_Work_Execution_Time) / times - Deadline_Check_Time);
      else
        rtl_printf("error in total_exec_time!\n");
    }
  my_task.init_budget = Budget;
  pthread_reset_np( thread, my_task.init_budget);
  rtl_printf(" -set init budget: %ld-\n", my_task.init_budget);
} /* if SCHED_SPORADIC */

/* init load factor */
{
  rtl_printf("initializing bound of load factor\n");
  Lo = 1;
  Hi = 1;
  otimes = 0;
  utimes = 0;
  do {
    Compute_Loads(Hi);
    Set_Budgets(Hi);
    if( Run_Tasks() ) {
      otimes = 0;
      utimes ++;
      if(utimes < Loops+2) continue;
```

```
      rtl_printf("underloaded at load factor %ld\n",Hi);
      Lo = Hi;
      if( Hi * 16 < Hi )
      {
         rtl_printf("load factor overflows in initialize_bound()\n");
         return ;
      }
      Hi = Hi * 16;
      utimes = 0;
      last_times = times;
      last_total_time = total_time;
    }
    else {
      otimes ++;
      utimes = 0;
      /*if(otimes == Loops+2)*/
      if(otimes < Loops+2) continue;
      rtl_printf("overloaded at load factor %ld\n",Hi);
      break;
    }
  } while ( 1 );
}

/* bisection for limiting load factor */
{
  otimes = 0;
  utimes = 0;
  rtl_printf("using bisection to find limiting load factor\n");
  do
    {
      Load_Factor = (Lo + Hi) / 2;
      Compute_Loads(Load_Factor);
      Set_Budgets(Load_Factor);
      if( Run_Tasks() )
      {
          utimes ++;
          otimes = 0;
          if(utimes < Loops + 2) continue;
          rtl_printf("underloaded at load factor %ld\n", Load_Factor);
          Lo = Load_Factor;
          last_times = times;
          last_total_time = total_time;
           utimes = 0;
      }
```

```
        else
        {
            otimes ++;
            utimes = 0;
            if(otimes < Loops+2) continue;
            rtl_printf("overloaded at load factor %ld\n", Load_Factor);
            otimes = 0;
            Hi = Load_Factor;
        }
      } while ( (Hi - Lo) > (Load_Factor + Accuracy) /Accuracy );
    if(Load_Factor > Lo)
      {
        times = last_times;
        total_time = last_total_time;
        Load_Factor = Lo;
        Compute_Loads(Load_Factor);
      }
    rtl_printf("limiting load factor: %ld\n", Load_Factor);
  }
} /* test_budget_and_load() */

void Set_Budgets (unsigned int load_factor) {
  /* assume that repl_period = period */
  hrtime_t budget;
  if(my_task.policy != SCHED_SPORADIC)
  {
    pthread_reset_np( thread, -1);
    return;
  }
  budget = load_factor * my_task.init_budget;
  if(budget > my_task.repl_period)  budget = my_task.repl_period;
  pthread_reset_np( thread, budget);
}

void compute_utilization(int scaled)
{
  if( !scaled )
    Total_Utilization = my_task.computation_time * Second;
  else
    Total_Utilization = Computation_Load * Second *
                        Unit_Work_Execution_Time;
  Total_Utilization = Total_Utilization / my_task.period * 100 / Second;
} /* compute_utilization(int scaled) */
```

```
void *Find_Utilization_Limit(void *arg)
{
  rtl_sched_param p;
  p . sched_priority = 10;
  if( pthread_setschedparam (pthread_self(), SCHED_FIFO,&p) )
    return 0;

  done_all = 0;
  pthread_getschedparam(thread,NULL, &p);
  p.sched_priority = my_task.priority;
  p.sched_ss_low_priority = my_task.low_priority;
  p.sched_ss_repl_period = my_task.repl_period;
  p.sched_ss_max_repl = my_task.max_repl;
  pthread_setschedparam(thread,my_task.policy, &p);

  dual_loop_benchmark();
  test_budget_and_load();
  done_all = 1;

  rtl_printf("\n--Testing Result--\n");
  if(my_task.policy == SCHED_SPORADIC)
    rtl_printf("tested a sporadic task\n");
  else
    rtl_printf("tested a periodic task\n");
  compute_utilization(0);
  rtl_printf("utilization without scaling : %ld%%\n",Total_Utilization);
  compute_utilization(1);
  rtl_printf("utilization with scaling    : %ld%%\n",Total_Utilization);
  print_with_time("period            :", my_task.period);
  print_with_time("repl period       :", my_task.repl_period);
  print_with_time("computation time  :", my_task.computation_time);
  print_with_time("init budget       :", my_task.init_budget);
  print_with_time("total_exec_time   :", total_time);
  if(times != 0 && total_time !=0 ){
      if(total_time >= times * (Computation_Load *
                       Unit_Work_Execution_Time))
        print_with_time("overhead per iteration:",
               (total_time -  times * Computation_Load *
               Unit_Work_Execution_Time) / times - Deadline_Check_Time);
      else
       rtl_printf("error in total_exec_time!\n");
  }
  stop =  clock_gethrtime(rtl_getschedclock()) ;
  rtl_printf("********** End of RM Test *********\n");
```

```
  print_with_time("totally testing time cost: ", stop - start);
  rtl_printf("\n\n");
  return 0;
}

int init_module(void)
{
  rtl_sched_param p;
  pthread_attr_t attr;
  Seconds_To_Run = my_task.period * Iterations / Second;
  start =  clock_gethrtime(rtl_getschedclock());
  rtl_printf("\n\n*********** RM TEST of SS **************\n");
  pthread_attr_init(&attr);
  p.sched_priority = my_task.low_priority;
  p.sched_ss_low_priority = my_task.low_priority-1;
  if(my_task.repl_period > my_task.period) return -1;
  p.sched_ss_repl_period = HRTIME_INFINITY;
  p.sched_ss_init_budget = HRTIME_INFINITY;
  p.sched_ss_max_repl = 32;
  if( pthread_attr_setschedpolicy(&attr,my_task.policy) )
    return -1;
  if( pthread_attr_setschedparam(&attr,&p) )
    return -1;
  if( pthread_create (&thread,&attr,Periodic_Task,0) )
    return -1;
  if( pthread_create(&main_thread,NULL,Find_Utilization_Limit,0) )
    return -1;
  return 0;
}

void cleanup_module(void)
{
  pthread_delete_np(thread) ;
  pthread_delete_np(main_thread);
}
```

BIBLIOGRAPHY

[1] T. P. Baker, lecture notes of "COP4613/CIS5930 Real Time Systems", Department of Computer Science, Florida State University, *http://www.cs.fsu.edu/ baker/cop4613.A00/*, Summer 2000.

[2] T. P. Baker, lecture notes of "COP4613 Real-Time Systems", Department of Computer Science, Florida State University, *http://www.cs.fsu.edu/ baker/cop4613.S98/lectures/L16/*, Spring 1998.

[3] M. Barabanov, master thesis "A Linux-based Real-Time Operating System", New Mexico Institute of Technology, June 1996.

[4] M. Barabanov, "Getting Started with RTLinux", document with release of RTLinux version 2.2.

[5] E. G. Jr. Coffman and P. J. Denning, *Operating System Theory* , Prentice-Hall, New Jersey, 1973.

[6] T. M. Ghazalie, T. P. Baker, "Aperiodic Servers in Deadline Scheduling Environment", *Real-Time Systems Journal, vol.9, no.1*, 1995.

[7] J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", *Proceedings of Real-Time Systems Symposium*, Dec. 1989.

[8] C. L. Liu, and J. W. Layland, "Scheduling Algorithm for Multiprogramming in a Hard Real-time Environment", *Journal of the Association for Computing Machinery, vol.20*, Jan. 1973.

[9] J. W. S. Liu, "Real-Time Systems", *Prentice Hall*, 2000.

[10] T. Motylewski, "Sharing Memory Between Kernel and User Space in Linux", Institute of Physical Chemistry, University of Basel, Switzerland, *2nd Real-time Linux Workshop*, Orlando, Florida, Nov. 27 - 28, 2000.

[11] R. Obenza, "Guaranteeing Real-Time Performance Using Rate Monotonic Analysis", Software Engineering Institute, Carnegie Mellon University, presentation on *Embedded Systems Conference*, Sep. 1994.

[12] H. Shen, T. P. Baker, master thesis "A Linux Kernel Module Implementation of Restricted Ada Testing", Department of Computer Science, Florida State University, 1998.

[13] C. Shih, Jiang Qian,*et al*, "Open Real-Time Linux ", Department of Computer Science, University of Illinois at Urbana-champaign, *2nd Real-time Linux Workshop*, Orlando, Florida, Nov. 27 - 28, 2000.

[14] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems", *The Journal of Real-Time Systems, 1, 1*, June 1989.

[15] B. Srinivasan, *et al*, "A Firm Real-Time System Implementation Using Commercial Off-the Shelf Hardware and Free Software", *Proceedings of IEEE Real-Time Technology and Application Symposium*, 1998.

[16] J. K. Strosnider, J. L. Lehoczky, and L. Sha, "The Deferable Server Algorithm For Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", *IEEE Transactions on Computers.*

[17] Y. C. Wang, K. Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel", *Proceedings of IEEE Real-Time Systems Symposium*, 1999.

[18] V. Yodaiken, "The RT-Linux Approach to Hard Real-time", *RT-Linux Whitepaper*, Department of Computer Science, New Mexico Institute of Technology.

[19] V. Yodaiken, and M. Barabanov, "RTLinux Version Two", *VJY Associates LLC*, 1999.

[20] V.Yodaiken, "The RTLinux Manifesto", *http://www.rtlinux.org*, 1999.

[21] Real-Time Systems Laboratory (RTSL), Department of Computer Science, University of Illinois at Urbana-champaign, "Oris Linux - An Open Real-Time Scheduling Linux", *http://pertsserver.cs.uiuc.edu/ cshih/ORIS.*

[22] IEEE Std 1003.1d-1999, *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) – Amendment 4: Additional Realtime Extensions [C Language].*

[23] IEEE Draft Std. 1003.1-200x, *System Interfaces, Issue 6*, Draft 4, July 31, 2000.

[24] *http://www.ca.postgresql.org/mhonarc/pgsql-interfaces/2000-02/msg00156.html*

[25] *http://www.rtlinux.com.*

[26] *http://www.rtlinux.org.*

BIOGRAPHICAL SKETCH

Weijuan Shi

The author was born in Jiangsu Province, China on May 3rd, 1975. She got her bachelor degree of Computer Science from Fudan University, Shanghai, China in July 1997. After that, she continued the master's degree program in the same department for two years, working as a Research Assistant on monitoring and visualization of parallel and distributed systems in the Computer Aided Technology (CAT) lab. During that period, she had a paper "Visualization of Performance Data in Distributed Computing" published in *Chinese Software Journal (Supplement)* in June 1999. In August 1999, she came to the Department of Computer Science in Florida State University as a master student and Teaching Assistant. Her paper "Implementation of Sporadic Server Scheduling in RTLinux" [1], part of this thesis work, was accepted by the 39th Annual ACM Southeast Conference held at the University of Georgia, Athens, Georgia on March 16-17, 2001.

From May 1997 to August 1998, the author worked as part-time Visual C++ programmer for an industry automation software *"SuperGuard"* in Softfoundry Inc. (Shanghai), Singapore. In summer 2000, she worked on intern as Network and Unix System Administrator for the FAMU/FSU College of Engineering, Florida State University.

[1]Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. 39th ACM-SE Conference 3/01 Athens, GA, USA. * 2001 ACM ISBN 1-58113-360-X/01/0003