UNIVERSITY OF CALIFORNIA,
IRVINE

Time-Triggered Function Support in a Real-Time Linux Environment

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Francesc Calvo Serra

Thesis Committee:
Professor Kane Kim, Chair
Professor Stephen Jenks
Professor Rainer Doemer

2007

The thesis of Francesc Calvo Serra is approved:

_____

_____

_____

Committee Chair

University of California, Irvine

2007

to my grandfather

Contents

List of Figures

List of Tables

Acknowledgments

This document was possible due to the support I received from many people. I would like to acknowledge their contribution and thank them for the help provided. First of all, I want to express my appreciation towards my advisor Professor Kane Kim for giving me such an opportunity and for his guidance and thoughtful advice. Thanks also, to my committee members Professor Stephen Jenks and Professor Rainier Dömer for being an active part of this work.

I also want to specially thank Mr. Pete Balsells with who I share the love towards my homeland, Catalunya. Pete gave me the opportunity to pursue my graduate studies in the University of California in Irvine, which would not have been possible without the financial support provided through the Balsells fellowship. My gratitude extends to Professor Roger Rangel, the director of the California-Catalonia engineering programs which manages the fellowship.

I am also grateful to my colleagues in the DREAM lab who I have shared many hard working long days with.

I would also like to mention the support received from the catalan community in Irvine where I found many friends who made me feel home in a foreign country; thanks Xevi, Asli, Sergio, Laura, Pep, Albert, Jose, Raul, Markitus, Itsi, Albertet, Alfred and Mercè, Aleix, Xavi and Patri, Sergi, Quimi and Anna, Raimon and Brenda, Marc and Nesveh, Marcel, Roger, Marc and Dolo... thanks to all of you for the experiences shared during this last year. From my time in Irvine though, I would particularly like to mention a person who made me feel very special and warmed my heart with hers, thanks Suus.

També vull donar les gràcies als amics de sempre, doncs malgrat la distància

formeu una part important de la meva vida. I finalment, agraeixo a la meva família, i en especial als meus pares, el suport incondicional que sempre he rebut per la vostra part. Sense vosaltres, res hauria estat possible.

Vull acabar recordant al meu padrí, qui ja no podrà llegir aquesta dedicatòria, però al qual segueixo portant amb mi.

Abstract of the Thesis

Time-Triggered Function Support in a Real-Time Linux Environment

by

Francesc Calvo Serra

Master of Science in Electrical and Computer Engineering

University of California, Irvine, 2007

Professor Kane Kim, Chair

The open source nature of Linux has led to its wide adoption in highly heterogeneous hardware platforms, including embedded devices such as cell phones or GPS devices where real time guarantees have to be provided. Due to the original Linux design as a general purpose operating system it lacks key features that a real-time operating system as such, should have. This thesis presents the design of TS-Linux; an implementation of the existing time-triggered function support OS (TSOS) architecture introduced by Kane Kim into the Linux operating system in order to overcome these deficiencies. The TSOS architecture provides a programming model based on the scheduling of time-triggered functions (TTFs). This model is specially suited for distributed networked environments such as sensor networks thanks to the definition of a global time framework which allows the execution of synchronized distributed actions. The original Linux scheduler has been modified accordingly in order to support the execution of tasks based on time properties

instead of priority levels. As a result, applications using the TS-Linux framework present a deterministic flow of execution which can be easily analyzed in terms of timing behavior. Therefore, guarantees for tight execution time bounds can be more easily provided. The combined execution of time-triggered functions and original Linux processes is enforced by the creation of timeslots which ensures a fair CPU share to each scheduling class.

Chapter 1

Introduction

1.1 Time support in embedded systems

In the last years, the increase in performance together with the cost reduction attached to embedded devices has boosted their popularity from marginal gadgets to being present in almost every type of device available. Cell phones, digital cameras, routers, GPSs or music players are just a few examples of embedded devices that have reached mainstream adoption. From their relatively simple initial software, their complexity increased significantly, leading them to being controlled by full size operative systems. Embedded systems require a tight coupling with the underlying hardware in order to guarantee the strict timing requirements imposed by their functionalities, i.e. a GPS device has to be synchronized with an accuracy in the order of nanoseconds in order to function properly.

Therefore, timing analysis is one of the main aspects involving the software design of embedded devices, whereas on the desktop computer segment, this factor can usually be ignored since the timing requirements are more lax on a general basis, relegating the timing analysis for low level hardware support. Real-time operating systems providing a thoroughly tested deterministic behavior are used in this environments. General purpose operating systems do not provide the deterministic behavior required to guarantee the requirements under all circumstances; on the

other hand real-time operating systems are usually tailored solutions that fulfill the requirements, yet suffer from a loss of generality which makes third-party applications development complicate. It only seems natural to balance both scenarios in a middle ground where a deterministic flow of execution is possible and timing constraints can be easily specified.

Such functionalities for dependable real-time and distributed systems can be provided by the Time-triggered function Support Operating System (TSOS) layer, based on the higher level Time-triggered Message-triggered Object (TMO) framework proposed by Kim and Kopetz in 1994 [3]. TSOS has already been widely employed in sensor networks [4–6]. Other prototype implementations under WindowsCE also show how TSOS can be efficiently merged into an existing operating system in the form of a middleware support library.

1.2 Motivation

Current TSOS implementations on existing operating systems show satisfactory results; the new framework eases application development specially when involving timing constraints. However, on the embedded devices market where real-time operations are crucial, Linux is the main player. As shown in table 1.1, it boasts a market share close to 50%; and what is even more impressive: it has a huge momentum. It grew from 32% to the current situation in the last five years.

Table 1.1: Real-time OS market snapshot for embedded devices (Source: [1]).

	2003	2004	2005	2006	2007
Linux	32%	37%	44%	46%	49%
MS Windows	12%	13%	16%	13%	12%
VxWorks	10%	10%	8%	6%	7%
DOS	9%	5%	5%	3%	3%
eCos	2%	3%	3%	5%	4%
Other UNIX	4%	5%	3%	4%	4%

The success of Linux in the embedded market is manifest. The reasons for that are numerous; one of the most important ones may well be its open source nature

which makes it the most adaptable operating system. Developers can modify the system according to their needs and remove or modify unnecessary parts. On the other hand, Linux was originally designed as a general purpose operative system for desktop computers; and as such, it lacks several of the key features a Real-Time OS should have. The strength of the embedded market (estimated in more than 6$ billion dollars by 2010 [7]), is a powerful force though, and the number of linux-based commercial distributions for embedded devices is constantly growing; companies like ARM, MontaVista, Sysgo, VirtualLogix and WindRiver are just some of the major players trying to accommodate Linux into this segment.

All these companies as well as non-profit initiatives are trying to optimize Linux for portable devices; Mainly, focusing on the following areas:

- Smaller system footprint
- Higher stability and reliability
- Ease of development
- Predictable system
- Reduced response time

As stated in earlier works [8–10], we believe the TSOS programming model can overcome some of the liabilities present in existing embedded solutions. TSOS model's assets, such as time-referencing, time-triggered actions, deadline imposition for method execution or nonblocking invocation of methods, can successfully improve crucial aspects of real-time operating systems. Therefore, its functionalities can become a great benefit for the embedded devices market.

1.3 Related Work

Given the current flourishing embedded market situation, real time operating systems are a hot topic in the development community. Numerous efforts are being

made in order to improve existing solutions. Adapting these methods to provide new functionalities presents several advantages. This is why existing high-level programming languages such as Java, have been added real-time functionalities. Regular Java is far from being real-time capable, but this was not a restriction to create the Java Real Time specifications [11].

In the operating systems domain, the adaptation of regular desktop Linux kernels to satisfy real-time requirements has been the target of efforts from both commercial and non-commercial parties. On one hand, companies such as Montavista [1] have successfully completed that adaptation process and they are already selling their real-time verified Linux version. As such, it is suitable for dependable systems and it is already being used in avionics and automotive industries. On the other hand, regular Linux kernel developers are also pushing for merges into the OS that guarantee hard real-time capabilities. Ingo Molnar is one of the most active in this area, and he is responsible for the Linux Real Time preemption patch [12], which provides a Linux version with much lower latencies than the mainstream release.

Open source solutions are not the only ones. Some of the most used real-time operating systems come from proprietary vendors which created their product from the bottom up. QNX is one of these companies which offers reliable solutions for the embedded devices industry [2]; however, due to their closed nature, the techniques used are not disclosed and they lack interest from the academic point of view.

Returning to the TSOS framework, there are prior implementations in the form of middleware for non real-time systems such as Windows NT [9], yet due to the nature of the underlying OS, time guarantees can only be provided on a best-effort basis.

1.4 Contribution

This thesis addresses the design and implementation of the existing TSOS architecture into the Linux operating system. Due to its open source nature, the im-

[1] http://www.mvista.com
[2] http://www.qnx.com

plementation is not just a middleware support library, but the kernel has been modified accordingly to fully support the TSOS specifications. In addition, several community-based patches have been included, in order to transform Linux into a hard-real time operating system, since this is an important requirement for most embedded systems. The timing specifications resulting from the TSOS environment under the patched kernel represent an advantage over the traditional scheduled method execution. As a result, applications using this framework can be easily analyzed in terms of timing behavior. The main contribution of the thesis is the prototype implementation and evaluation of the TSOS framework on Linux. However, the prototype still lacks certain features such as global time support, therefore it corresponds to a partial implementation of the TSOS architecture.

1.5 Overview

This thesis contains five chapters. The order is arranged as follows: Chapter 2 introduces fundamental background information to understand the decision making procedure and the benefits of the new framework. Chapter 3 goes over the Linux structure and the implementation details, as well as the main inconveniences found. Then, chapter 4 assesses the prototype implementation by performing several benchmarks which validate the prototype's timing behavior. The thesis conclusion and future research areas are suggested in chapter 5.

Chapter 2

Background

2.1 Real-Time Operating Systems

At this point the real-time operating system (RTOS) concept already came out. Contrary to the most extended belief, real-time does not refer to doing things very fast, but scheduling computations in a time predictable way. RTOSs are used in systems where time is a critical factor; therefore having an entirely predictable execution flow can lead to the formal proof of execution before an actual deadline. It is not possible to give guarantees without a deterministic behavior. In many systems, formal proof is a must; for example, think about a car accident situation where the airbag deployment within a few milliseconds is critical. Such behavior needs to be guaranteed because a late airbag deployment is as bad as no deployment at all. These are time-critical systems.

Real-time tasks have a defined timing behavior which makes RTOSs time-driven systems. This allows to compute beforehand whether a task can be effectively scheduled or on the other hand if its deadline cannot be met. Since the CPU execution flow is already established, the task is immediately rejected and the actual user can be notified before the task deadline. For example, let's take a look at an imaginary processor that is only dealing with two tasks. Task A, has to be executed for two time units every three time units and task B, should be executed during one

time unit every four units. At this point, the CPU will be be running on average 92% of the time (2/3 + 1/4=11/12 = 92%). Then, if task C arrives to this virtual system with execution requirements of one unit every five time units, the system will reject the task after reaching the conclusion that it cannot be scheduled. It is not possible since the CPU should run 111% of the time (2/3 + 1/4 + 1/5 = 67/60 = 111%).

In real-time systems task priority is essential. A job with an imminent deadline should run with the highest available priority. High priority jobs can even be granted execution time before some of the own operating system tasks. Interrupting an existing task, either user or system based, in order to execute a higher priority one is known as preemption. The arriving new task doesn't let the existing ones to complete their execution since it has a higher priority; thus it preempts them. Switching from one task to another requires performing some extra operations not useful from the task point of view, therefore these are seen as a 'task switching overhead'. The more switches occur, the higher the overhead will be. Since task switching occurs more frequently in real-time based systems (due to preemption), their throughput as a general basis is lower than with regular desktop or server systems. However, throughput is not one of the key factors for a RTOS; the features to focus on are the response time to events, i.e. latency, predictability, met deadlines. These are the parameters that will be measured to evaluate the proposed prototype. On a general basis, it can be stated that a general purpose OS tries to optimize the average case while a real-time OS attempts to improve the worst case scenario.

Real-time operationg systems are more complex than regular systems: They work under most of the same principles but they have an added constraint: timing behavior. Therefore, its design and implementation become more restrictive. A good example can be found in the memory domain. Several widespread memory management techniques, such as dynamic memory or garbage collection cannot be used in the strict timing domain of real-time systems due to their unpredictability. This section only pretends to be an introduction to the RTOS domain and the most important concepts required for our design. Yet, a thorough presentation on these systems can be found on [13].

The determinism of a real-time operating system is what allows the inclusion of the TSOS architecture. As it will be seen in section 2.3, TSOS provides a simple and powerful way to specify the timing behavior of tasks. Taking that into consideration, these can be later on scheduled by the operating system with timing guarantees. On the other hand, the benefits would be much lower on a non-real-time OS since although the timing specifications are there, the system cannot assure their proper execution.

2.1.1 Timing Requirements

Timing is the most important factor in real-time operating systems. Nevertheless, the fact that these guarantee a predictable execution does not translate into particular requirements. Concrete numbers depend on the environment where the system will be deployed. Obviously, a 0 ns latency would be desirable but it is a just an unachievable theoretical bound; in practice, the smaller the latency the better the system will behave, and the more difficult it will be to implement. Currently, computer clocks which establish the time base for a system, are in the order of the Gigahertz. Such high frequency implies cycles in the order of nanoseconds. Due to the complexity of existing implementations, even the simplest operations easily reach magnitudes of microseconds, which is the point where latencies for highly optimized embedded systems are. MontaVista, one of the commercial Linux vendors previously mentioned, boasts a worst case measured latency under 50 μs [14], which is a very low. Their products have the lowest latency measured on a commercially available Linux based system. On a completely different project, the real-time Java developers claim their implementation can schedule periodical threads with switching task times as low as 15 ∓ 5 μs. This optimal case can only be achieved in a minimalistic environment which lacks important features such as dynamic memory, yet again, it is a surprisingly low value.

Regular Linux latencies are considerably larger. Several benchmarks are presented in table 2.1 and the results come as expected. On the average, Linux behavior is not bad at all with a measured task latency of 132 μs under a very heavy load. This is a good result for a general purpose operating system but the problem is in

the worst case scenario. In this case, the latency rises to 4508 μs which is a little bit high, but for the 2.4 kernel the worst case latency is of 158 ms which cannot be employed in a dependable embedded system. For example, when dealing with audio applications, a 20 ms delay would result in a noticeable glitch. The values introduced should be taken as an initial reference to compare to when analyzing the improvements to the existing architecture.

Table 2.1: Measured worst case Linux Kernel latency (Source: [2]).

	Average	Worst case
kernel 2.4	1133 μs	158560 μs
kernel 2.6	132 μ	4508 μs

2.1.2 Hard and Soft requirements

One of the most straightforward concepts of time based systems is the deadline. It is defined as a time until which the completion of a task has more utility to the system, and after which its completion has less utility. From this definition several types of deadlines follow. The special case where the utility to the system takes a binary form is known as hard deadline or hard requirement. It simply means that completing that job before the deadline is useful and doing it after the deadline results in zero utility. An example of such deadline would be the case of an airbag deployment in an accident: it has to be done at the right time otherwise it results in no benefit at all. It might even be harmful to do so at an inadequate time. Hard-deadlines are quite restrictive on their timing behavior. On the other hand, deadlines which have laxer requirements are known as soft. The utility associated to a task completion with a soft-deadline decreases linearly as the actual task ending time (posterior to the deadline) increases[1]. Even in the most strict real-time devices, forcing all constraints to be hard will reduce the system flexibility and will make the overall design stage more complicate.

[1]More information on deadlines and other RT related concepts can be found on [15]

2.1.3 Timing Constraints

The precision and latency expected from the operating system are based on the environment where it is going to run. However, since the TSOS architecture is not initially constrained to any domain but it is conceived as a general framework, a set of fixed specifications is not available. The only constrain which can be originally established is the hard real-time nature of the underlying OS, in order for the TSOS layer to be able to provide timing guarantees. Apart from that, a high boundary for tolerable latencies is a priori unknown. It is important to note though, that due to the TSOS nature, distributed applications are encouraged without making assumptions about the underlying network. Whether it is a local area network, a wireless network or the internet, the smallest delays experienced are in the order of the low milliseconds. Therefore, a local scheduling latency smaller than those transmission delays would be desirable.

2.2 Time Reference

Up to this point, several references have been made to the strict timing requirements of embedded systems and the importance of a highly responsive / low latency OS. However, any time reference will depend on a clock source which is finally based on the measurement of a cyclic event. There is no such thing as an ideal universal time; there are no ideal clocks either. Therefore, time references will have to consider the jitter introduced by imperfect clocks such as the ones used in modern computers. In case the clock precision were much higher than the timing requirements, clock imperfections could be simply neglected, yet as it will be shown in the next sections, clocks are far from perfect and in consequence, accurate timing analysis has to be performed.

2.2.1 Physical Clock

A physical clock is a device based on an oscillator of some kind that generates a continuous and regular event. This event is used to establish a time base which will allow time measuring; therefore, it should occur as frequently and regularly as possible in order to minimize the jitter. The exact specifications of the clock will depend on the application domain. Computer clocks are based on a cheap quartz oscillator contained within the 8254 timer chip included in most motherboards. The specifications state this oscillator works at an approximate frequency of 1.193 MHz. and the time cycle associated to this frequency is 0.84 us. Then, temporal intervals can be measured as a multiple of this time base, which is known as granularity. Therefore, the smallest measurable time event associated to an oscillator will be equal to its granularity which in this case is slightly below one microsecond. Such granularity is good enough when dealing with embedded devices since it is much smaller than any of the time intervals that need to be measured. It is important to note that by using a cycle based clock, the continuous concept of time becomes discrete: As far as clocks are concerned, time increases in very small amounts and at a constant pace.

2.2.2 Drift (Uncertainty)

In the previous section it was assumed that a clock ticked at a constant (and ideal) rate. However, reality is far from this scenario. Clocks are based on oscillators and these may vibrate at a slightly different frequency than the specified one. Moreover, their vibration is not constant and it is subject to some degree of uncertainty. How reliable an oscillator is, should be specified by the maker. For example, the quartz oscillator previously mentioned typically suffers from en error smaller or equal to 50 parts per million (ppm) [16]. This means that after one million ticks, the clock can deviate from an ideal clock as much as 50 ticks. It certainly seems a low amount, but in practice that means that each second a computer clock can skew 50 us from an ideal timer. With such uncertainty, having a resolution smaller than 1 us is certainly useless. This drift accumulates along time, and after a single day, the clock may

be more than 4 seconds wrong. When trying to maintain a global time within a sub-millisecond precision, such delays cannot be tolerated.

There are several alternatives to improve this accuracy. The synchronization with a higher precision clock is the most frequent. The new clock reference will have to be based on a more reliable oscillator with an uncertainty, at least lower than 1 ppm. Such oscillators are not easy nor cheap to acquire. Typically, this adjustment can be either done locally with a GPS device or remotely over a network which is connected to an atomic resonance frequency standard clock. The new precision obtained will be based on the the accuracy of the reference clock and the synchronization algorithm used.

2.2.3 Synchronization

Since there is no perfect clock, all physical devices suffer from some degree of uncertainty. How small it is will depend on the quality of the device used; however, different clocks will eventually drift apart until a significant deviation is reached. Therefore, if the time difference between clocks has to be kept within a certain threshold, the need to periodically synchronize them appears. Many algorithms have been proposed to do so: Precision time protocol, TIME protocol, etc. and they are based on different approaches: master-slave, distributed, fault-tolerant amongst other techniques. In the internet, the most extended one is the Network Time Protocol (NTP) which synchronizes against world time servers and can typically obtain an accuracy of 10 ms on the internet or around 200 μs in a local area network [17].

The TSOS framework requires a reliable time base, but the algorithm used to provide it falls outside its design and depending on the environment some approaches might be more suitable than others. Therefore, the synchronization algorithm will be merged as a module to the existing TSOS framework. The approach taken in the prototype implementation is a simple master-slave algorithm where every slave node will synchronize with the master at the beginning of each second. This design was performed assuming a scenario with several nodes connected through a local area network where communication latencies can be kept under a few milliseconds.

2.3 TSOS Framework

The main distinctive trait characterizing the TSOS approach is its time based nature. Traditional systems are generally event driven; on the other hand TSOS is based on the execution of methods triggered by time, while still supporting the conventional way. This new approach supports method scheduling based on priority but also based on time properties such as period length, deadline proximity, etc.. Moreover, synchronization between distributed nodes is simplified thanks to the notion of global time. Consequently, timing analysis, both at local and global level, becomes much easier, since time is an inherent design factor in TSOS.

2.3.1 Time-Triggered Functions (TTF)

A time-triggered function can be seen as a regular thread of execution to which timing behavior has been attached. The timing properties can be something as simple as: "execute task A every x seconds, from time t1 to t2". From such statement, the TSOS subsystem would extract the execution deadlines and proceed to schedule the task accordingly, assuming an Earliest Deadline First scheduler [18]. Obviously, in order to interpret the timing properties embedded in a TTF, the default OS scheduler has to be modified and enhanced.

This was just a trivial example, yet since there many ways to stipulate timing constraints a proper timing syntax has to be established, which is why the Autonomous Activation Conditions were defined.

Autonomous Activation Conditions (AAC)

The AAC syntax is both simple and powerful. As shown in table 2.2, it is based on six parameters that completely define the timing specifications of a task. These conditions were introduced in [8] and [10].

The following statement is equivalent to the AAC parameters from the table: "*Execute task X from T1 until T2 every P seconds. For each cycle P, start at some*

Table 2.2: Autonomous Activation Conditions

From:	T1	This is the start time.
To:	T2	This is the end time.
Every:	P	This is the period.
Earliest Start Time:	EST	This is the Earliest Activation Time.
Latest Start Time:	LST	This is the Latest Activation Time.
By:	D	This is the deadline for each execution.

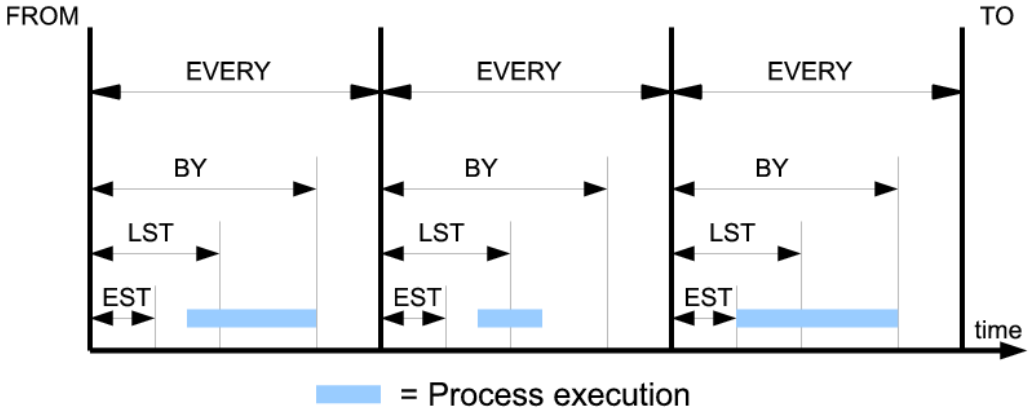

Figure 2.1: AAC specifications diagram.

point between $[n * P + EST, n * P + LST]$ *and finish before* $[n * P + D]$", where n is the cycle number. The first two parameters T1 and T2 are expressed in an absolute way, whereas P, EST, LST and D are relative time expressions that refer to the intra-cycle behavior. The reader may notice that such specifications can be used to express a one-time execution but they are specially suited for periodic events. Figure 2.1 shows a detailed timeline execution.

2.3.2 Local and Global Time

Assuming the existence of several nodes working on the same network, each one will have its own clock, and the execution of their TTFs will be based on that time reference. However, it could be very interesting to develop a collective sense of time so nodes can coordinate their actions. The existence of a global time notion allows the creation of what would be known as global TTFs. It is important to note that

a set of AAC specifications is not bounded to any specific time reference, therefore, coexisting TTF threads can be based on different time basis. As it was seen in section 2.2.2, due to the existing uncertainty associated to physical clocks, there is the need to synchronize nodes periodically within a network in order to keep clock jitters bounded. Due to its distributed nature, any synchronization procedure will be more complex than simply reading a local clock value; therefore the resolution and precision obtained will be lower than the local ones. However, these drawbacks are overshadowed by the benefits associated to having a global time notion.

Assuming several nodes communicating through an Ethernet-based local area network, the round trip time measured between two given nodes can be seen in table 2.3.

Table 2.3: Sample LAN (Ethernet) round trip time delay statistics

15 packets (0% loss), time 13999ms.	
Min:	206 μs
Avg:	211 μs
Max:	215 μs
Std:	8 μs

An average communication delay between nodes of 211 μs is small enough to maintain a global time notion with a sub-millisecond precision.

2.3.3 Scheduling

There is no predetermined scheduling policy associated to the TSOS framework; moreover, the underlying operating system already has an existing scheduler with its own policies. Since one of the main objectives of this approach is being able to provide timing guarantees; the original OS scheduler which suffers from an unpredictable timing behavior will have to be bypassed by the TSOS scheduler when necessary. In practice, this means that the OS scheduler will be scheduled by the TSOS scheduler unit; therefore, the latter has a higher priority and will be able to run as often as desired. On the other hand, this approach could strangle traditional Linux applications leaving them virtually with no CPU; this is why it is important to maintain certain degree of fairness.

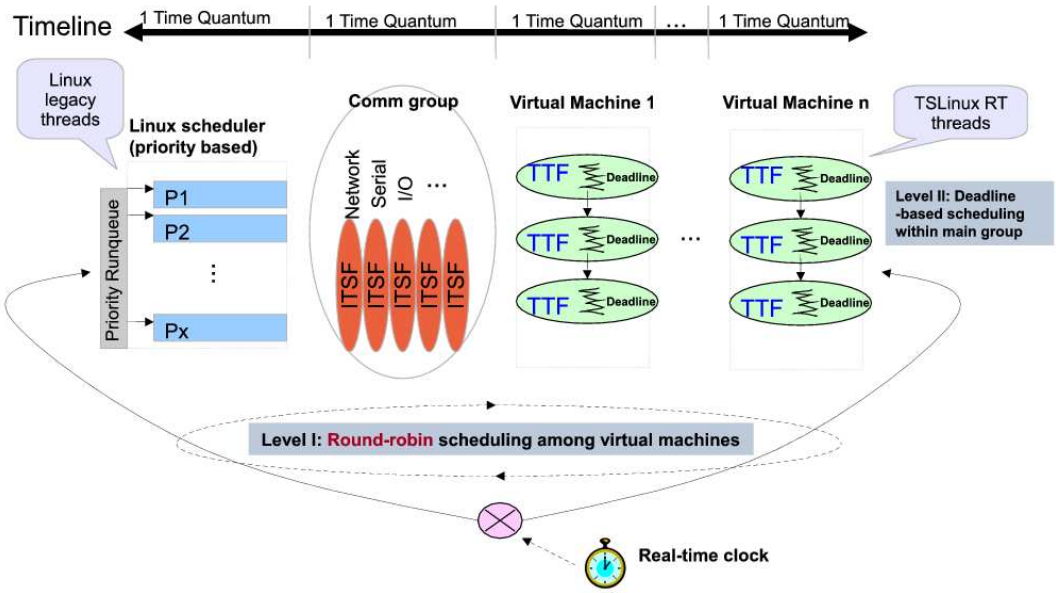

Figure 2.2: Round-Robin algorithm across VMs in the system

Virtual Machines

Having layered schedulers ensures having a clean division between tasks belonging to different groups. In that sense, the TSOS scheduler defines several virtual machines or scheduling groups. Each group will be independent from the others. They will all receive the same share of CPU, by periodically scheduling tasks from each group in a round-robin fashion. Opposite to other systems, TSOS defines as many runqueues as virtual machines are created. A schematic representation of the runqueues approach can be seen in figure 2.2.

The scheduler will go through all the runqueues in the system staying a constant amount of time in each one. When the runqueue is empty, that virtual machine will be simply skipped until the next round. The time assigned to each virtual machine (VM) is known as time-quantum. It does not necessarily have to be a predefined constant but it can be dynamically adjusted. Given this approach, a worst case response time threshold can be easily defined as:

$$rt = n * tq$$

where rt stands for the worst case response time, n represents the number of VMs in the system and tq refers to the time-quantum. Common time quantum values are found around a few milliseconds, typically between 1 and 10 ms. The shorter the time quantum is, a lower response time bound can be established. On the other hand, the overhead of going through all the runqueues too often will not be negligible anymore. When a strict worst case bound needs to be set, limiting the number of VMs running under TSOS is also a valid approach.

In TSOS, there are several tasks pivotal for maintaining a proper operation of the system such as network communication, clock synchronization, etc. These jobs will be running periodically, executing as regular TTFs. However, due to the importance of their nature, they will be contained in their own virtual machine. Therefore, TSOS will initially contain at least three independent scheduling groups. The first group are those threads belonging to the original operating system (in this case Linux); the second virtual machine corresponds to the TSOS upkeep tasks and the third one will be executing any TTF method that has been registered.This is the initial scenario, but at any time, more TTFs belonging to new scheduling groups can be created and more virtual machines will be added to the system.

Evidently, the more virtual machines there are in a system, the smallest CPU share each one of them will have. That could well become a problem in the extreme case that the number grows to an untenable point. However, due to the easily computable timing analysis, the decisions whether to allow plugging new machines into the scheduler can be taken on run-time.

Intra-VM scheduler

Once the CPU is granted to a particular virtual machine for a full time-quantum period, there will still be a runqueue populated with several time triggered tasks (TTFs) ready to be executed. A second scheduling policy is required in order to determine the order of execution between these tasks. The TSOS design does not impose any particular algorithm. The idea is to have a modular scheduler where diverse algorithms can be employed. Given the timing specifications attached to each TTF, a time based scheduling policy will be used. By default, the Earliest

Deadline First (EDF) approach is applied.

As explained in [18], EDF is a dynamic scheduling algorithm. It places processes in a priority queue. The closer the deadline of a process is, the higher its position will be in the queue. Whenever an event occurs (a task finishes, a new task arrives, etc.) the queue will be searched for the next process to run. Unfortunately, this search doesn't take a constant amount of time $O(1)$; it will be higher when more processes are present in the queue. The exact complexity depends on implementation details, but in general it will linearly increase with the number of processes; that is $O(n)$.

On the VM corresponding to the underlying Linux operating system, there are no TTFs but only regular threads which lack timing specifications. Therefore, in this case a regular priority based scheduling algorithm is used.

Independently from the algorithm used, once a task is selected to run it will be executed by the CPU as a regular thread until, either: it finishes, a task with a higher priority arrives (i.e. with a closer deadline) or the virtual machine time-quantum has expired.

2.4 Independently Threaded Service Functions

One of the reasons why Linux is not a real-time operating system is the fact that interrupts handlers are executed in what is known as '*interrupt context*'. This execution mode is triggered as soon as an interrupt signal arrives; therefore, having a higher priority than any user or kernel thread. This approach is logical, since interrupts need to be handled as soon as possible in order to avoid lost data, but this also delays the execution of user processes. This is an acceptable situation in a general scenario, but it is not in some specific circumstances. For instance, a real time process with strict timing requirements which does not use the hard drive, should not have to wait for the handling of interrupts issued by the disk, if that implies he will miss its own deadline.

Interrupt handling is typically separated in two phases; in Linux, these are known as hard and soft interrupt requests (IRQs).

Hard IRQ Since interrupts are generated by hardware devices, this phase deals with modifying data from its hardware buffers and / or copying it to system memory. The delay or improper handling of this operation would lead to the malfunction of the hardware device. Therefore, these type of operations cannot be interrupted and have to be completed as quickly as possible; thus, they have to be executed with the system interrupts disabled. On the other hand, by definition interrupts can only be disabled for very short amounts of time, therefore hard IRQ handling has to be very fast and all actions susceptible to be delayed don't have to be executed within a hard interrupt handler; they will be placed in a softIRQ. That is why, a hard IRQ handler does not defer a process execution significantly.

Soft IRQ The second phase of interrupt handling comprehends operations which are still important for the interrupt to work properly, but they can be executed with interrupts enabled and therefore they can be delayed for a longer period time. In the previous example, when the hard IRQ completes its execution the hardware device is ready to be used again, but the data provided by the device still has to be processed. Assuming it is a network card, the data to process might correspond to a packet received through the physical interface. Therefore, it has to be sent to the process waiting for it. The soft IRQ needs to take place before the data is available to processes, but its nature is not so urgent.

Given the clear distinction between hard and soft interrupt requests, it is feasible to execute softIRQs in a thread instead of interrupt context; this way they do not have absolute priority over all processes and they can be selected to run within the scheduler whereas this is the original priority-based Linux one or the new time based TS-Linux scheduler. In the TSOS model proposed by Kane Kim, the tasks performed by softIRQs run within a thread with time properties, known as Interrupt Threaded Service Function (ITSF). An ITSF will be selected by the scheduler according to its AAC, and it will compete with other TTFs to acquire the CPU. At this point, a process could be created with a higher priority than an ITSF, but this is the rare case. In general, ITSFs need to have guarantees about their prompt

execution; therefore, in TS-Linux there is a timeslot specifically assigned for the execution of ITSFs.

2.4.1 Atomic Directives

In a temporal programming environment atomic directives become of utter importance. Most operations involving external devices or synchronization between several threads are done via some type of blocking primitive such as semaphores. The Linux kernel is full of them, as well as spinlocks. Most of them have a very short response time, but some operations can delay for a few milliseconds. For example, when reading data from a file stored on disk, the access time can be as high as 100 milliseconds. This wouldn't be a problem if the calling process had the option to wait for the information while doing some other task. However, the wait time is a priori unknown: the calling process will be blocked and put to sleep when necessary for an unbounded amount of time. This is a significant problem, not just for TS-Linux but for real-time operating systems in general.

One way to solve the issue would be for user applications to be careful when issuing instructions that access critical sections suitable to be blocked and delayed for a long time. On the other hand, some blocking calls cannot be further postponed and have to be eventually made, therefore, the application could be divided in a real-time section and a non real-time one. Creating separate threads to accomplish it seems a sound approach. Threads can then transfer information via non-blocking buffers. This is a valid working solution; however, transferring the problem to the user domain does not seem to be the best approach since it can easily become error prone.

The other way to address the problem which is employed in TS-Linux, is based on creating new non blocking atomic directives for certain operations. [19] introduces several mechanisms in order to access lock-free structures in a synchronized way. From those, TS-Linux uses the *trylock* approach. The concept behind this method relies on having an atomic operation that checks whether a lock is free or not; in case it is free, the lock will be acquired as a usual semaphore. On the other hand, when

it is busy, the caller returns instantly without having acquired it. In TS-Linux, the trylock will be used to replace some of the critical semaphores used in networking and filesystem. Implementation details will be further discussed later on chapter 3.

In this way, timing guarantees can be provided by using the newly provided primitives; but user applications will have to limit to only using the non-blocking directives.

2.4.2 Network Communication

The I/O subsystem is one of the most significant components from an operating system as this is the part which allows data transmission to external devices; whereas these are magnetic disks, serial ports or network nodes. However, disk and network operations can be several orders of magnitude slower than modern processors. Therefore, proper management of I/O operations is crucial in order to avoid CPU stalls which might lead to time deadlines not met.

In a TSOS-based system, network operations become specially important since TSOS is conceived as a design solution for distributed nodes which communicate over a network. The underlying type of network is not defined a priori, yet original implementations were focused on simple nodes belonging to unreliable meshes such as sensor networks [20].

In order to coordinate the nodes belonging to a single network, there are several tasks which have to be performed periodically. One of them, which has been previously mentioned in section 2.2.3, is time synchronization. On unreliable networks, fault tolerance is another of the tasks which have to be performed regularly. Due to the importance of these maintenance tasks TSOS guarantees their execution by establishing a virtual machine exclusively for their completion.

Figure 2.2 shows how a particular time slot is assigned for network tasks. This period is established beforehand, therefore when the moment arrives the network driver has to be available for TSOS, otherwise there is the possibility for the time slot to expire; and, in that event, time guarantees could not be provided. In order to avoid conflicts between the underlying OS and TSOS operations, the network

interrupt handler shall be modified in a way that guarantees the TSOS execution and defers OS operations. However, the drawback from this approach is obvious: the solution is not portable as drivers have to be modified in a one-by-one basis.

Chapter 3

TS-Linux Implementation

As previously introduced, TS-Linux intends to extend the original Linux capabilities in order to support the time-driven execution of processes according to the TSOS design model. One of the first aspects to define regarding its implementation, is whether timing guarantees can be provided within the Linux kernel or on the other hand an external micro-kernel is necessary (as employed by [21] and [22]) . Given the long-term Linux development, current kernels based on the 2.6 version were initially released on 2003 (and developed since 2001); therefore, after being under public scrutiny and test for more than 5 years, they are considered mature. Initial timing tests presented on Chapter 4 confirmed that current Linux kernels can successfully provide sub-millisecond latencies, therefore TS-Linux was implemented within the Linux version 2.6 kernel. Another significant reason to discard a separate kernel implementation, is the loss of generality and compatibility. TS-Linux aims to simultaneously provide, timing guarantees for processes which need them and minimize interference with regular tasks; therefore the major part of the original kernel will remain intact and changes will be mainly contained within the scheduler.

3.1 Real-Time preemption patch

Due to the open nature of Linux, developers unsatisfied with current features or performance can create their own flavor of Linux by adding the modifications deemed necessary. Initial kernel-2.4 based Linux, suffered from high worst case latencies which can cause problems even for simple time constrained applications such as audio playback. Ingo Molnar (at the present time one of the main Linux kernel developers) deemed those timing properties insufficient and created the real-time preemption patch [12]. The current TS-Linux implementation is based on a vanilla 2.6.21 kernel [1] plus the real time preemption patch [2].

The bottom-line idea of this patch is to minimize the latency by making everything (including the kernel) premptible. By looking at an example it is easier to understand how this can be done. If there are two tasks A with a high priority of 10 and B with a lower priority 5; when A is ready to run, it should never be waiting because of B. Original Linux supports this behavior but has several limitations. One of these occurs when B executes a system call. Then, the execution path goes to kernel mode. At this instant, if A becomes ready it cannot execute until the system call summoned by B finishes. When using the preemption patch, the kernel will check for the existence of higher priority processes and yield the CPU to them even though it did not finish the tasks it was performing. These will be resumed when A finishes its execution and B is selected to run again. This is just an example, but virtually everything was made preemptible by this patch, including critical sections and even interrupt handlers. Having the option to execute a task with more priority than an interrupt handler is limited to privileged users due to the negative consequences such as lost interrupts and in consequence, lost data it can have.

[1] Available since April 2007 on http://www.kernel.org/pub/linux/kernel/v2.6/

[2] Available since May 2007 on http://www.kernel.org/pub/linux/kernel/projects/rt/older/

3.2 Clock Source

In order to keep track of time locally, a clock source is necessary. In section 2.2.1, the fact that each motherboard has a constant frequency clock was introduced. However, this is not the only time source available in a computer. There is also the CPU clock which has a much higher frequency, therefore, it can provide a better resolution. TS-Linux uses this clock as the local time source. The CPU clock presents new difficulties, though. It doesn't have a constant frequency accross systems which leads to a different cycle length for every computer; in modern portable computers, the frequency can be dynamically lowered and the CPU will execute at a lower rate to save energy.

TS-Linux assumes that dynamic frequency mechanisms such as Advanced Configuration and Power Interface (ACPI) are disabled; therefore the clock frequency remains constant. Its exact value is calibrated at each boot-up and it can be read from the file /proc/cpuinfo. All Intel and AMD processors since the original Pentium2 released in 1997 provide a register known as Time Stamp Counter which contains the number of cycles that took place since the system boot-up. The value of this register can be accessed with very low overhead via the assembler instruction *rdtsc*. Given the current frequencies in the order of gigahertz, the available clock resolution is higher than 1 nanosecond. TS-Linux performs the necessary scaling and keeps track of time in a 64 bit variable exactly with 1 nanosecond resolution:

$$localtime = \frac{rdtsc * 10^9}{freq}[ns]$$

The *localtime* value is frequently updated. At least every time the scheduler is called or the timer interrupt occurs. The use of a 64 bit counter is imposed by having a nanosecond resolution. If only 32 bits were used, the counter would overflow every 4 seconds, making it virtually impossible to compare times more than four seconds apart. By using 64 bits, this problem is resolved since the overflow point is not reached until 585 years after the computer booted up.

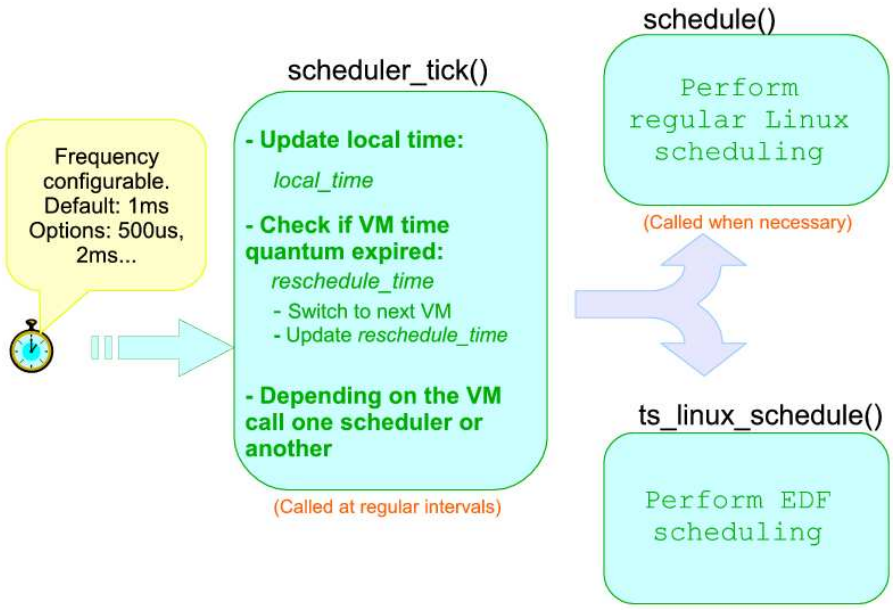

Figure 3.1: Timer interrupt workflow.

3.3 Timer Interrupt

The timer interrupt is generated periodically by a physical clock located on every computer. This interrupt is used by the operating system to keep track of the time assigned to the several tasks running. When a task is set to run by the OS, it has control over the CPU, therefore it can execute arbitrary code on it. When the timer interrupt occurs the proper interrupt handler is called and the OS can regain control and decide whether the task can keep using the CPU or it should be assigned to another job. The frequency of the timer interrupt is set at compile time; the default value in Linux is 1 KHz, i.e. the timer cycles are 1 millisecond long. This cycle length is still valid for TS-Linux, but the interrupt handler itself has to be modified in order to ensure the next virtual machine is executed when its timeslot arrives.

As seen in figure 3.1, the original interrupt handler updates the time counters and checks that the current process is still the highest priority one. TS-Linux forces an extra check: whether the active virtual machine has consumed its timeslot or not. Whenever any of these checks fails, the scheduler is called to select again the highest priority task.

3.4 Scheduler

The new scheduler is based on a modular approach where several schedulers coexist. The proper scheduler will be called by the timer interrupt depending on the active virtual machine time slot. A priori, each time slot has the same length; by definition, this length is a multiple of the interrupt cycle. As a starting point, TS-Linux uses a VM timeslot of 2 ms. Therefore, the timer interrupt will go through all the slots in a round-robin fashion, switching from one virtual machine to the next every 2 ms. In case a virtual machine has nothing to run, its timeslot will be ended prematurely and the next one will be selected in a regular way.

3.4.1 Native priority-based scheduler

The original linux scheduler is assigned to the virtual machine 0. Therefore, it will only be called when its timeslot is the active one. With the exception of how often it is called, the scheduler structure and behavior remain the same. It is a priority based scheduler. In practice, it supports 140 different priority levels; but it makes the conceptual distincton between the first 100 priorities which are considered real time and the last 40 which are considered non-real time or regular. The implementation is based on a 140 position array, where each position holds a pointer to the head of linked list. Each list, will contain all the processes in the system with the same priority. Selecting the next process to run is as easy as finding the first non-null linked list and selecting its head process [23]. This process requires an amount of time independent from the number of actual tasks in the system, which is known as $O(1)$. It is one of the main advantages of this scheduler; however, big O notation refers only to how well the scheduling algorithm scales and not to its actual speed. For real-time tasks, the higher priority one will always be selected starving the other ones if necessary. On the other hands, regular tasks are selected based on an heuristic algorithm that combines priority, execution time, and sleeping time ensuring no task is starved from CPU. The fact that some tasks are considered real-time by the scheduler does not provide any guarantee about their time behavior; it is reduced to the fact they will be executed more often than the other tasks.

3.4.2 New time-based scheduler

The new scheduler is based on the time properties of a process in order to decide who will execute next. The processes are ordered by increasing deadlines; that is, the process with the earliest deadline will execute first, which is equivalent to saying it has the highest priority. This scheduling approach is known as earliest deadline first (EDF). EDF is a dynamic scheduling algorithm; this means decisions are taken dynamically as processes enter or leave the scheduler and not beforehand. EDF is an optimal scheduling algorithm on preemptive systems with a single CPU [18]. An optimal algorithm can successfully schedule a collection of jobs whenever there is a valid schedulable order. Since EDF is only optimal on uniprocessor systems, the execution of TTFs will only take place on the first CPU of the system, in case there is more than one available.

The approach used by EDF is different than a priority based scheduler, but in this case priorities are also used, yet infinite levels are available. Such degree of freedom comes with a cost though. The scheduler access time is not constant anymore and depends on the number of processes in the queue. When adding a new process to the deadline-sorted queue, the proper insertion position has to be found which requires scanning on average half of the queue. Therefore, the cost associated to this approach is $O(n)$.

In practice, this means that if the number of jobs to be scheduled is very high, the scheduler may take a significant amount of time to decide who should run next, which implies an unnecessary overhead not present in the regular Linux scheduler. Since TS-Linux tries to provide timing guarantees, it seems reasonable to set a threshold on the number of processes which can be placed into the scheduler. Then, the worst case scenario will be the time the scheduler takes to select the next process when the threshold has been reached.

Limiting the number of processes that can enter the scheduler seems a quite restrictive solution, but there is a point where more processes would not be able to be scheduled, not because of the overhead caused by the scheduler, but due to their overlapping timing specifications. Therefore, imposing this restrictions works as a

solution for overload situations. In case more processes still need to be added to the system, it can still be done by creating a new virtual machine.

Data Structures

Autonomous Activation Conditions As it was explained in section 2.3.1, the scheduling properties of a time-triggered function are defined by a set of specifications known as autonomous activation conditions (AAC). Therefore, the following data structure was created to hold them:

```
struct aac{
    u64 from;
    u64 to;
    u64 every;
    u64 est;
    u64 lst;
    u64 by;
};
```

u64 is a synonym for the *unsigned long long* type which stores numbers using 8 bytes.

Process information In Linux, each process or thread is an independent entity which can run on its own. It has a *task_struct* data structure associated which fully defines its properties. This structure is defined in the Linux header file sched.h[3]. It is a key structure for managing every aspect of a process: scheduling properties, virtual memory, current state, siblings, sleeping time, etc. Due to its length (it contains more than 100 different fields with a total size of 2536 bytes) it will not be copied in this document. Time triggered functions are a specific type of processes, and as such, they need all the information contained in a *task_struct*, but they require some extra data; therefore, TS-Linux redefines the structure by adding the required fields. Here is a simplified representation containing the added fields and

[3]The exact location in Linux v2.6.21 is: include/kernel/sched.h, line 822

the original Linux ones which will no longer be used when scheduling TTF methods:

```
struct task_struct{
    //Original Linux fields not used in TTFs
    int prio, static_prio, normal_prio;
    unsigned long rt_priority;
    struct prio_array *array;
    //...

    //Added TS-linux fields
    struct aac *tspec;
    int vmID;
    int first_iteration;
    int next_iteration;
    int last_iteration;
};
```

The new fields are mostly self-explanatory: *tspec* is a pointer to the AAC specifications that will be used to schedule the process; *vmID* is the virtual machine id which this process runs under; *first_iteration*, *next_iteration* and *last_iteration* are used by the scheduler as boolean flags to keep track of the running status of the process and decide what queue it should be placed into. Regular Linux processes will ignore these added fields which in their case are simply set to 0.

Virtual machine The virtual machine layer is an abstract concept defined by TS-Linux; it can be seen as a guaranteed share of CPU. For instance, a configuration with two virtual machines will guarantee a 50% share to each one of them. Processes belonging to a virtual machine cannot be freely scheduled by Linux, therefore they have to be taken off from the system runqueue and be placed in the virtual machine queue. A virtual machine is defined by the following data structure:

```
struct vmachine{
    int enabled;
    spinlock_t lock;
```

```
    struct task_struct *active_ttf;

    struct list_head rq; //running queue
    int nr_running;
    u64 earliest_deadline;

    struct list_head wq;  //waiting queue
    u64 earliest_est;
    int nr_waiting;

    struct list_head dq;  //disabled queue
    int nr_disabled;
};
```

The first field *enabled* determines whether the virtual machine is enabled or not, that is, if it will receive its share of CPU or will simply be bypassed. This field is necessary since the number of virtual machines in the system is dynamic and can be set during runtime. Iniatially when Linux starts up, an array containing the maximum number of virtual machines is allocated, and only the active ones are enabled. The next field is a *lock* that provides serial access to the whole data structure. TS-Linux only schedules time-triggered functions on the first CPU, but in a multiprocessor environment other processors can also create TTFs; therefore they need to access the virtual machine structure. The spinlock type used for the lock is equivalent to a semaphore but the calling process remains blocked in a busy waiting fashion. This means that it should only be employed for very short blocking times. *active_ttf* is a pointer to the currently active process in the virtual machine, in case there is one running, otherwise it is just a null pointer; having a pointer to the active process simplifies the scheduler implementation.

As it can be seen, a virtual machine contains three different queues:

- Running queue
- Waiting queue

- Disabled queue

Each queue is implemented using a circular double linked list where each item points to the previous and next elements. Since linked-list have to be traversed in order to discover how many elements are in them, an additional field holding that information is created: *nr_running*, *nr_waiting*, *nr_disabled*. A TTF belonging to the virtual machine is necessarily contained in one of these three queues. If the TTF is disabled it will never be scheduled and it will be placed in the disabled queue. Items are only placed / removed from this queue when the *enableTTF() / disableTTF()* system call is invoked. On the other hand, processes which do have to be scheduled will be placed in one of the other two queues. The waiting queue contains TTFs which are active, but that cannot be scheduled yet due to their timing conditions. This is a priority queue sorted by increasing *est* field, which means the method that can be set to run earliest will be in the first position. The third and last queue is the running one. Processes contained in it, are not necessarily running but they can be set to run given their time specifications. Since the scheduling algorithm chosen is EDF, the queue is sorted by increasing deadlines, that is, the *by* field. The first field from this queue will be the one selected to run when the virtual machine timeslot comes. Items are moved from one queue to the other by the TS-Linux scheduler given the current time instant.

Scheduler execution

Here is the logic which TS-Linux scheduler is based on. On each execution the following procedure takes place.

```
//PART1: Update running process
If TTF first iteration:
        1) Compute AAC specifications in absolute time.
        2) Set first_iteration = 0;
        3) Put the TTF in the right VM (based on vmID)
        Place the task_struct in the proper queue
                Running queue if now > est
```

```
                Waiting queue if now < est

If TTF last iteration:
        1)Remove the TTF from its queue.
        2)Deallocate the task_struct structure

If TTF Next iteration:
        1)Compute AAC specifications for next iteration.
        2)Set next_iteration = 0;
        3)Remove the TTF from the running queue.
        4)Place it in the proper queue based on the new AAC.
                Running queue if now > est
                Waiting queue if now < est

Check if VM timeslot expired
        If now > vm_reschedule
                - Update vm_active = vm_active+1
                - Compute vm_reschedule += TIME_SLOT
                - Remove running TTF

//PART2: Select next process
Look at TTFs in the active VM wait_queue:
If now > ttf.est
        - Remove from wait_queue and place into run_queue
        - Repeat until no TTFs are placed in the run_queue

Look at TTFs in the run_queue
- Select the TTF with a closest deadline
- Put the selected TTF to run
```

3.5 ITSF Implementation

The implementation of threaded interrupts in TS-Linux is based on the original handling of softIRQs performed by Linux. As explained in section 2.4, softIRQs are executed in interrupt context; however in order to provide some degree of fairness Linux limits the number of soft IRQs that can be handled in a row to 10 [4]. After that, their execution will be handled by a regular kernel thread. A kernel thread behaves like a user thread, but it performs tasks for the Linux kernel. The thread in charge of softIRQs is known as *ksoftirqd*. When 10 soft interrupts have been handled, the execution context is returned to user processes even though there still are pending interrupts and the *ksoftirqd* thread is awaken. This thread is initialized with the minimum scheduling priority available, therefore, it will only be selected by the scheduler in case there are no other processes ready to run.

The typical interrupt workflow is the following: A hardware device generates an interrupt signal; the proper hard IRQ handler is executed; once its execution is done, soft IRQ handlers (if any) are called. At most ten soft handlers can be called in a row. If there still are tasks to be performed, the ksoftirqd thread is awaken and execution flow is returned to user context. It is important to remember that at least once every millisecond an interrupt signal will be generated by the system clock; therefore softIRQs are most likely to be handled in interrupt context than by the background low priority kernel thread.

TS-Linux moves all the softIRQ interrupt handling to the background thread, and all that needs to be performed by the original softIRQ handler is to wake up that thread. Since at this point soft interrupt handling is no longer performed in interrupt context, it can no longer delay user processes in an undertermined way, but the scheduler will select the proper task. Due to the original low priority associated to the ksoftirqd thread, softIRQ handling could be delayed dangerously; therefore, in TS-Linux its priority is switched to AAC specifications and it will be selected by the new time-based TS-Linux scheduler instead of the original one. Table 3.1 shows the used values by the soft interrupts handler.

[4] This value is MAX_SOFTIRQ_RESTART, as defined in file kernel/softirq.c at line 206.

Table 3.1: AAC specifications for ITSFs.

TTF1	
FROM:	0
TO:	forever
EVERY:	5 ms
EST:	0 ms
LST:	1 ms
BY:	5 ms

The execution of ITSFs is guaranteed by assigning a virtual machine timeslot exclusively to their execution, as it can be seen in figure 2.2. If ITSFs are the only TTFs running in the virtual machine, its timing specifications are not relevant. They will only matter if other TTFs are included in the same timeslot. In that case, the TS-Linux scheduler will use the AAC specifications in order to select the next process to be executed.

3.6 Added System Calls

Linux provides a set of system calls to manipulate and establish processes configuration. In order to interact with the new available virtual machines and time-based scheduler, additional system calls were added. The following list goes over all of them and provides a simple explanation of the function they provide.

long SetVirtualMachines(unsigned int number) As its name states, this system call is used to set the number of virtual machines in the system. Initially, there is only one virtual machine which corresponds to the original linux system. Therefore the parameter number can be in the range [1,maxVM]. maxVM is initially set to 10. Higher values are not encouraged due to the long time it would take to go around a full round-robin scheduling round. On success, the function will return the current number of virtual machines in the system. In case of error, -1 will be the value returned.

long RegisterTTF((f*)(args), void *args, int vm, struct aac time) This is the function used to create time-triggered functions, that is, the core of TS-

Linux. It receives several parameters. The first one is a pointer to the function to execute and the second one a pointer to its arguments. The third parameter is the virtual machine number where the TTF is going to execute and the final parameter contains the time specifications that are going to control the execution of the TTF. After testing the validity of all the parameters passed, the system call will create a new thread ready to run the function f(args). This new thread will be placed in the proper virtual machine runqueue and will be executed based on the AAC specifications. On success, the system call returns the process id of the new created thread and in case of error, -1 will be returned.

long DisableTTF(int pid) This is a simple system call. As its name states, it will disable the execution of the given TTF, which will no be scheduled again until it is not re-enabled. Since many time-triggered functions are executed periodically without a stopping point, at some intervals it might be necessary to disable them temporarily.

long EnableTTF(int pid) This system call enables the scheduling of the given time-triggered functions. By default, TTFs are enabled; therefore, this function is only necessary after having disabled the execution with *DisableTTF(int pid)*.

long getTStime(unsigned long long *time) This system call can be used by TTFs to retrieve the current time value used by TS-Linux to schedule tasks. It will receive a pointer to a 64 bits integer where it will place the value of the current time with a nanosecond precision.

This small set of added system calls is enough to manage the execution of TTFs. However, since time-triggered functions are just an extension of regular linux processes, original system calls can still be used on them. The actions performed by some of these system calls may cause large delays, though. Therefore, by using them strict time guarantees may no longer be provided.

3.7 Latency sources

There are many actions which delay the execution of processes. In a regular environment, this is a simple nuisance, however in a real-time scenario identifying these latency sources and avoiding them is crucial. Some delays are predictable such as reading data from a hard-drive. However, delays caused by other situations like memory allocation cannot be initially foreseen. When designing TS-Linux all of them had to be considered and avoided by using different techniques.

The following sections go over the most significant ones and how they have been solved.

3.7.1 Dynamic Memory

Physical memory is a scarce resource, therefore by using dynamic memory, programs can request memory only when they need it and only the amount needed. Typically, dynamic memory is not allocated until the very last moment; that is, when it actually needs to be used. The typical flow of execution goes like this: process A asks for new memory, i.e. by using *malloc()*, Linux notes the petition and tells process A its new memory is ready to be used. At this point, process A can access the new memory regions. However, it is not until A tries to write something on them, that Linux is forced to physically allocate the memory regions that need to be modified. This approach is known as Copy On Write (COW). The allocation process depends on several factors: amount of free memory, size of free pieces, fragmentation, etc., therefore it does not take a constant amount of time. By looking at it from the perspective of A, a new memory region was requested and granted quite fast, but then, when accessing its new memory, process A is put on hold for an undetermined amount of time, which can be fatal if timing guarantees need to be provided by A.

The only solutions seems to be either not to use free memory or trying to write immediately on each newly allocated page to force its real allocation. Solution one is quite restrictive because it implies that each process has to know the amount of memory it will need to use at compile time. On the other hand, the second solution

shifts the problem to a later stage. Dynamic memory can be used but every single page has to be written before being used, which in case of big memory regions can also take a long time.

But this is not the only problem. As explained in [24], Linux has an *overcommitment* memory policy in order to maximize concurrency; that means, that processes can be assigned more memory than it is actually physically available on the system. The only possible way of doing that is to back up some of the memory information on disk in a transparent way to processes. However, if processes are not aware of what memory regions are actually held on memory they cannot predict beforehand how long it will take them to access their data, since disk access is several orders of magnitude slower than memory access, and again, no timing guarantees can be provided. The overcommitment policy, ruins the previously mentioned solutions, because even though a memory region has been physically assigned to process A, there is no guarantee that a later stage, if memory is tight, that information is copied to disk and the memory assigned to some other process, leading to a significant delay for A when accessing its data again.

The solution used by TS-Linux allows the use of dynamic memory while still providing time guarantees. The system call *mlockall()* provided by Linux allows disabling memory backing from a process to disk. This way, when a memory region is assigned to a process it is guaranteed that it will remain that way until the process terminates. By calling it at the creation of a TTF and touching each memory page, it is guaranteed that there will be no memory faults after that point. Using this technique, the problem is solved partially, because no delays due to page-faults will occur, but still no dynamic memory support is provided.

Dynamic memory allocation interfaces such as *malloc()* are provided by the glibc library present in every Linux system [5]. The library interfaces will trigger the proper underlying Linux system calls. glibc provides the call mallopt() which allows the configuration of its behavior. By setting the following parameters: $M\_TRIM\_THRESHOLD = -1$ and $M\_MMAP\_MAX = 0$, the library is configured to never release memory to the operating system. In practice, this means that

[5]The GNU C library is available from http://www.gnu.org/software/libc/

a given proces can allocate a private memory pool when starting up, from which it will be the only process allocating and releasing memory. As long as the private memory pool is not exhausted, dynamic memory can be used with 0 page faults.

The following code snippet provides the explained functionalities:

```
//Startup phase
mlockall(MCL_CURRENT | MCL_FUTURE);
mallopt (M_TRIM_THRESHOLD, -1);
mallopt (M_MMAP_MAX, 0);
buffer = malloc(PRIVATE_MEMORY_POOL);

// Touch each page in the pool
for (i=0; i < PRIVATE_MEMORY_POOL; i+=page_size)
    buffer[i] = 0;
free(buffer);

//RT phase
mem1 = malloc(4096); //dynamic memory allocation
mem2 = malloc(8192); //without page-faults
...
```

Memory access time tests were performed and presented in chapter 4, showing the validity of this approach. Other real time solutions can be implemented by fully redesigning the memory management stack and using an $O(1)$ memory allocator. This is the approach used in [25]; however, this solution was not pursued in TS-Linux due to the significant changes it would require to the whole operating system.

3.7.2 Process communication

Having an environment that provides timing guarantees is advantageous, but it cannot be fully employed unless some sort of communication can be established either with other processes in the system or external devices. In Linux, there are several options to establish a data transfer between different entities, but independently

of the nature of the involved entities, the approach relies on *read()* and *write()* directives. The default behavior for these functions is blocking. For example, when a given process wants to read n bytes from a pipe, it will call the read() function on the pipe. If those n bytes are available, the function immediately returns; on the other hand, if they are not ready yet, the calling process will be blocked. The same behavior occurs when trying to write on a full buffer. As it has already been explained, real-time threads cannot be blocked under any circumstance. In order to overcome this situation, it is not necessary to create new system calls for TS-Linux, but regular Linux already provides workarounds for processes which do not desire to be blocked. This is accomplished by opening the given file, pipe, socket, etc. with the corresponding *O_NON_BLOCK* flag, then when trying to read on an empty device, the function will return immediately stating that 0 bytes could be read.

However, TS-Linux, provides its own library to perform some of these basic actions. It basically contains wrapper functions that switch the default blocking behavior to the non-blocking one. Developers, can freely use any function whether it belongs to TS-Linux or not; if called with the proper parameters, the behavior will be the same on both ends, and blocking will never occur. By using the provided library, this nuisance is taken off from developers.

Interprocess communication

Communication between independent threads or even processes is provided by two different means in TS-Linux. The first one is by using a shared memory region and the second one is based on message queues.

Shared memory Independent processes have their own virtual memory space. Even the same address in different virtual spaces, points to a different physical memory location. This approach provides extra security due to the isolation, yet it makes communication between processes more difficult. The virtual machine concept has been introduced before, but up until now, it only referred to the scheduling policies which guaranteed a CPU share. TS-Linux also creates a virtual memory region for each virtual machine. Any process belonging to

the virtual machine can access its memory region. Initially, 8 MB have been reserved per virtual machine. These are the directives created for this purpose:

- void *TS_shm_attach(int size)
- int TS_shm_dettach(void *address)

These functions are similar to an allocation procedure, but the difference relies on the fact that the virtual machine shared memory pool is already created (and allocated). Therefore, this calls are just used to attach / dettach the calling process to that region. A size parameter is used because, access to only a reduced section of the memory region is also allowed. The pointer returned by *TS_shm_attach()* is the address of the first location inside the memory region. Due to the memory addressing approach employed by Linux, two unrelated processes calling the same function will receive different value for that first address; however, both will refer to the same physical location. TS-Linux is able to provide these functionalities based on the underlying standard library functions: *ftok()*, *shmget()*, *shmat()*, *shmdt()* and *shmctl()*.

It is left to the user, to overcome the concurrency problems inherent by the fact of having a shared memory region between many threads. If no synchronization measures are taken, overwritten or corrupted data will probably occur.

Message queues The other approach to communicate processes is by transmitting information through a similar approach to pipes, but based on discrete packets instead of an information stream. It can be seen as a similar approach to internet sockets, but only available within a single computer. TS-Linux defines the following functions to transmit and receive packets:

- TS_msg_snd(channel, packet, size)
- TS_msg_rcv(channel, packet, size)

These message queues defined are similar to non-blocking buffers (NBB). Conceptually, it is like there are many pipes available in the system; when invoking any of the previous functions, the channel parameter states what pipe we want

to access, packet is the address of the data structure that we want to send or receive, and size stands for the number of bytes the transmitted structure has. TS-Linux is able to provide these functionalities based on the underlying standard library functions: *ftok()*, *msgget()*, *msgrcv()*, *msgsnd()* and *msgctl()*.

I/O devices

Communication with external devices is provided by the I/O subsystem. External devices are usually associated with large latencies due to their slower nature. Even when using non-blocking strategies, these devices work at a slower pace; therefore, real-time threads will eventually have to wait for them whether they are sleeping or busy waiting.

Parallel port This is a special case of I/O device. It is currently becoming deprecated because it can transmit data at a lower rate than newer alternatives such as USB. However, since it is a memory-mapped device, it becomes very easy to use and it is not very different than setting (or reading) the value of a variable in memory. TS-Linux does not provide a particular directive to read or write data to the parallel port because it can be accessed natively in a non-blocking manner and with a very low latency, as seen in table 3.2, which shows the maximum parallel port access delay when writing a single byte, as measured by 5000 samples taken 10 seconds apart from each other.

Table 3.2: Parallel port access time.

5000 samples, taken 10 seconds apart.	
Min:	869 ns
Avg:	1215 ns
Max:	3908 ns
Std:	400 ns

The functions used in Linux to access the parallel port are *outb()* and *inb()* in order to write and read respectively. The parallel port, connected to an external oscilloscope, was used to check the temporal behavior of TS-Linux, by creating TTFs that modified the port output periodically.

Disks and Network Although disk and network access belong to different device families, their read and write operations can be unified because both are reached through the Linux virtual filesystem (VFS). For instance, the same *write()* function provided by Linux can be used to write to a hard drive or to a network socket. However, typically sockets are accessed through the system calls family *send()*, *sendto()* and *sendmsg()*, but both are based on the same underlying implementation. Since the function used becomes more an issue of developer preferences, when designing TS-Linux no new non-blocking wrapper functions were created to read or write network / disk devices. The following new function was created though:

- *TS_IOnoblock(int descriptor)*

Given any already open file, pipe or socket, *TS_IOnoblock()* will transforms its behavior into non-blocking, thus making sure it can be used in a real-time framework. Its implementation is based on the underlying Linux function *fctl()*.

3.7.3 Blocking system calls

Although most latency issues are caused by the I/O subsystem, there are other sources which can delay the outcome of a given program. Several system calls, can significantly delay the execution flow in a real-time environment. Some of them are quite obvious due to the functionality they provide such as *nanosleep()*, which is the system call used in Linux to put the active thread to sleep for a given amount of time. Other calls, like *read()* or *write()* can also be foreseen as a latency source due to their involvement with external devices. It was already explained how to avoid them in the previous section. On the other hand, system calls like *mmap()* can delay the execution of a program but they might not be as easily identified as such. *mmap* stands for memory mapping, and it is the function that performs the allocation of memory resources for a given process. Incidentally, it is the underlying function called by the standard library when using *malloc()*. The latency problems caused by dynamic memory explained in section 3.7.1, were caused by *mmap*. Therefore,

real-time application developers should be aware of the underlying implementation of the system calls used in order to avoid unforeseen latency issues. Eventually, all existing system calls should be replaced by their equivalent non-blocking versions.

Chapter 4

Prototype Assessment

When implementing TS-Linux, several modifications were performed to the Linux kernel and many design decisions were taken in order to provide timing guarantees. In this chapter, numerous measures are taken in order to validate the work done. The configuration used to perform these measurements is a Pentium 4 microprocessor at 1.7 GHz. The RAM memory available in the system is 768 MB and the Linux distribution used is Fedora Core version 7 with a modified kernel which supports the TS-Linux extensions.

4.1 Dynamic Memory

One of the serious problems to overcome was the dynamic memory latency issue. By using the locking approach introduced in section [reference], memory accesses become almost instantaneous. Figure 4.1 was obtained by writing random bytes to dynamic memory pages previously allocated. Due to the Copy On Write allocation approach used in Linux, in one case, they were not really allocated and it can be easily seen, due to a write time almost 100 times higher than in the previously locked case. However, this case is still acceptable, since memory accesses only take around 20 microseconds per page accessed. In case, of higher memory fragmentation, these numbers should grow, but this behavior could not be seen. On the other hand,

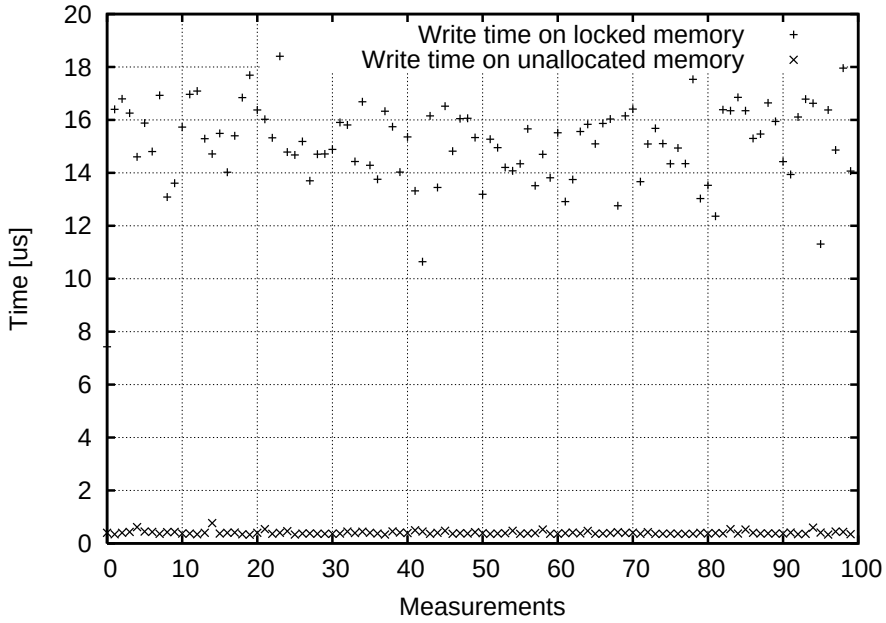

Figure 4.1: Memory write access time.

reading from memory pages which were backed to disk turns out to be disastrous. This behavior is shown in figure 4.2. Reading a recently used (or locked) full page can be done in only 12 microseconds. If the pages were not accessed in a long time and they have been moved to disk, this time can rise to about 10 milliseconds. This behavior was triggered by running two independent programs; one of them allocates a small memory region and sleeps for a long time; meanwhile the other one starts allocating high amounts of memory stressing the whole operating system. At this point, the first process is awaken and measures the access time to its memory regions.

Since locked dynamic memory can be used with a very low and constant latency, the approach used in TS-Linux has proved to be successful.

4.2 I/O Access

In order to asses that TS-Linux is behaving properly from an external source, the I/O subsystem has to be employed. The following measurements were taken using

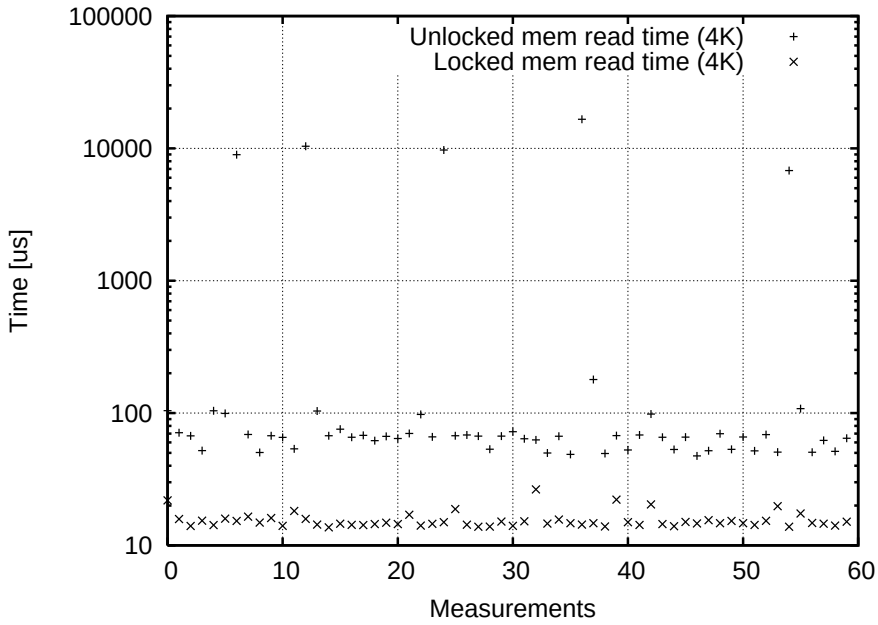

Figure 4.2: Memory read access time.

the parallel port due to its simplicity and low latency. The approach used was to write 2 consecutive bytes to the parallel port from a time-triggered function (TTF). In the first write, a single bit is set in the output and in the second one a second bit plus the original one are set. These bits are connected to different input channels in a digital oscilloscope. The first channel is used as the trigger and the delay between them is measured. The results obtained can be seen in figure 4.3.

The latencies obtained are very low which means the parallel port can be successfully accessed from within TTF methods.

4.3 TTF scheduling

The next step is to show that a TTF can be properly executed according to their timing specifications. A TTF with the specifications in table 4.1.

which renders one hundred executions of the given method. The body of the function besides performing the timing measurements contains a dummy loop which goes through a random number of iterations between 10 and 30 thousand. In or-

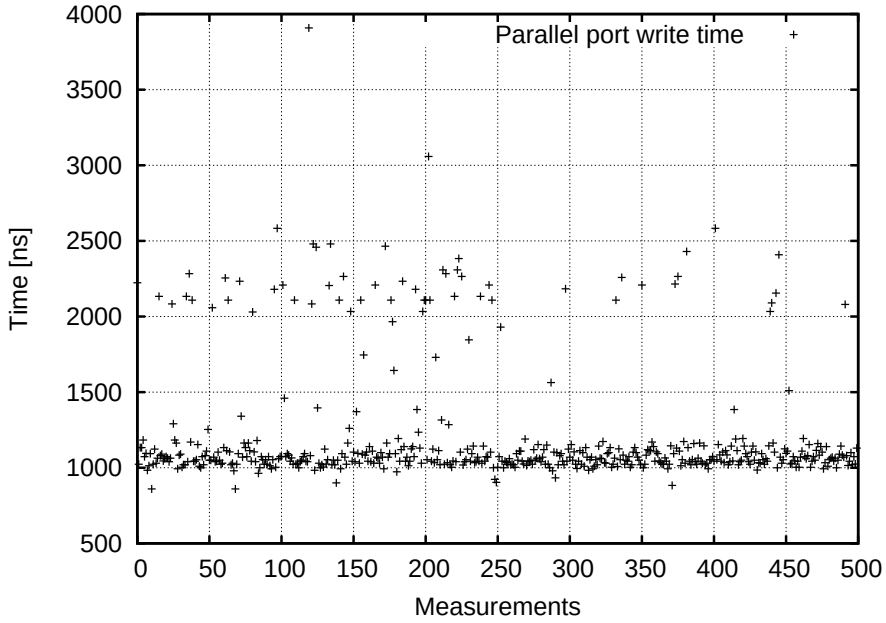

Figure 4.3: Parallel port access time.

Table 4.1: Single TTF, AAC specifications.

TTF	
FROM:	now
TO:	now + 50 ms
EVERY:	5 ms
EST:	0 ms
LST:	4 ms
BY:	4 ms

der to check the independence of virtual machines, a regular Linux process going through an infinite loop which leads to 100% CPU consumption, is set to run on the background. The beginning of all periods have been aligned and they are shown in figure 4.4. The system executing this methods contains two independent virtual machines (plus the original Linux one) with a time-quantum of 2 milliseconds. The timer interrupt occurs with a frequency of 1 millisecond.

As it can be seen, the behavior is not deterministic but all the time specifications of the method are fulfilled, since there is no single execution that finishes more than 5 milliseconds after the origin of the cycle. Due to the existence of three virtual machines with a 2 milliseconds time quantum, the behavior presented in the

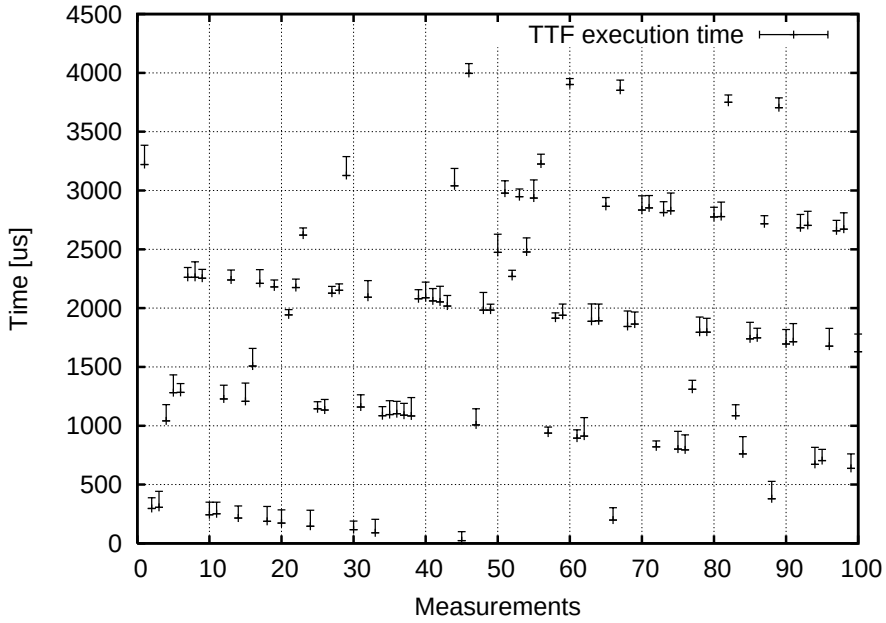

Figure 4.4: TTF execution.

figure seems logical since in the worst case, the TTF will begin its execution after 4 milliseconds ($2 * timequantumlength$). It also seems that most TTF executions are approximately aligned in 1 millisecond intervals, which is the timer interrupt frequency.

The next step is to execute two TTFs in the same virtual machine and check that their execution flow is correct and the presence of multiple time triggered threads does not interfere with each other. The timing specifications are presented in table 4.2 and 4.3:

Table 4.2: Specifications for TTF1.

TTF1	
FROM:	now
TO:	now + 100 ms
EVERY:	10 ms
EST:	0 ms
LST:	5 ms
BY:	8 ms

Since both TTFs have the exact same specifications but the second one has a

Table 4.3: Specifications for TTF2.

TTF2	
FROM:	now
TO:	now + 100 ms
EVERY:	10 ms
EST:	0 ms
LST:	5 ms
BY:	6 ms

shorter deadline, they should be scheduled in an interlaced fashion starting by the second one, and each one should be executed 10 times. Here is the execution flow obtained:

The behavior exhibited by these tests is the correct one. As a side note, it is recommended not to set the *every* parameter in a TTF to a lower value than $time_quantum * numberofvirtualmachines$. Otherwise the specifications will most likely not be fulfilled.

4.4 Clock uncertainty

The TSOS principle is based on a global time definition. A time synchronization protocol between nodes has been created, but it is necessary to know how often it should be scheduled, that is, the specifications of the TTF which runs the *clock_synch()* method. Computer makers do not provide specifications about the reliability of their clocks. Since no specifications are available for any of the clocks used, it is assumed they have a similar degree of uncertainty. The clock used for testing is the CPU one which has a nominal frequency of 1.7 GH. However, it does not remain constant. Several frequency measurements were performed as seen in figure 4.5.

The difference between the highest and lowest measured frequencies is of 175 KHz. Assuming different computers have clock sources with the same precision, the

Table 4.4: 2 TTFs running in the same virtual machine.

Time units in μs			
Pid:	Start exec:	End exec:	Exec time:
1720	0	154	154
1719	204	352	148
1720	9998	10088	90
1719	10123	10212	89
1720	19992	20128	136
1719	20170	20306	136
1720	30549	30688	139
1719	30731	30869	138
1720	39975	40127	152
1719	40162	40313	151
1720	49995	50064	69
1719	50112	50178	66
1720	10123	10212	89
1719	50112	50178	66
1720	60036	60119	83
1719	60162	60243	81
1720	69969	70102	133
1719	70145	70276	131
1720	80522	80597	75
1719	80643	80717	74
1720	89947	90054	107
1719	90092	90199	107
1720	99964	100063	99
1719	100517	100616	99

maximum expected drift comes from the following formula:

$$\begin{aligned} \Delta f &= f_{max} - f_{min} \\ f_{avg} &= \frac{\Sigma_{i=1}^{n} f_i}{samples} \\ Drift &= \frac{\Delta f}{f_{avg}} \\ &= 102us \end{aligned}$$

If a precision in global timing below 1 millisecond wants to be achieved, clocks have to be resynchronized at least once every 10 seconds. Therefore, in TS-Linux the

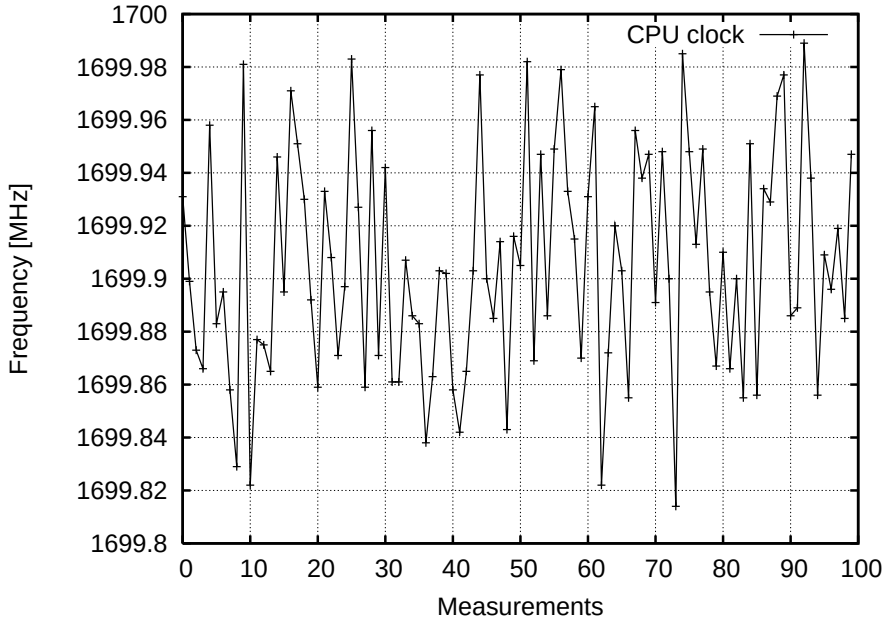

Figure 4.5: CPU frequency.

synchronization TTF is set to be executed every 5 seconds.

Chapter 5

Conclusions and future work

5.1 Conclusions

TS-Linux is based on the TSOS design introduced by [9]. It functions as an operating system extension that aims to provide timing guarantees on a regular Linux system. The original scheduler logic has been modified in order for the architecture to support the new functionalities. Such significant changes in the core of the OS have been done trying to minimize the impact on regular Linux processes. In an attempt to provide fairness, TS-Linux defines the concept of a virtual machine which is based on a scheduling hierarchy that ensures access to an equal share of microprocessor time between virtual machines.

In TS-Linux, the basic execution block is based on a process with attached time specifications, which is known as a time-triggered function (TTF). The mentioned timing specifications are set by using absolute time parameters which completely define when and how often a process should be executed; these are known as autonomous activation conditions (AAC). The scheduler can immediately detect when a process does not fulfill its specifications and issue the proper exception.

In a strictly time constrained scenario, additional measures are taken in order to reduce latency sources. Dynamic memory is allocated beforehand in order to avoid costly page faults. System calls which trigger blocking actions are forbidden; and

an additional set of directives to access I/O devices in a non-blocking manner is provided. The design is not tied to any specific hardware platform, but like Linux, it can be used across several architectures.

The performance of the current design has been measured in a prototype implementation and it has been shown that the timing guarantees are met in all expected situations. The scheduling precision has been shown to be consistently under 2 milliseconds given a timeslot equal to the timer interrupt cycle length and two virtual machines. Communication in a timely controlled way has also been proved to be successful using I/O ports such as the parallel port.

5.2 Future work

TS-Linux provided a successful prototype implementation of a design architecture: the possibilities of the framework have been presented, but they still have to be shown by developing applications which benefit from them.

The reduced API provided by TS-Linux can be further developed in order to provide more functionalities, such as fault-tolerant systems support. There is still room for improvement in the network communication subsystem which turns out to be a key block of the distributed approach presented by TSOS. Global clock synchronization is still in a very early stage, and it is based on a master-slave method; other synchronization algorithms have been shown to perform better and do not suffer from the single point of failure that comes with having a master node.

The easy analyzable time properties of TTFs have been shown, but there is always room to improve the scheduling precision supported by the TS-Linux scheduler. The timer interrupt frequency could be risen in order to allow shorter timeslots or acceptable device latencies could be lowered. After all, the more precision provided by the system means that the framework can be employed in new fields.

Bibliography

[1] ZDM, 2007. Snapshot of the embedded linux market. Ziff Davis Media Inc.; Electronic publication. http://www.linuxdevices.com/articles/AT7065740528.

[2] White, B., 2003. Linux 2.6. a breakthrough for embedded systems. Ziff Davis Media Inc.; Electronic Publication. http://www.linuxdevices.com/articles/AT7751365763.

[3] Kopetz, H., and Kim, K., 1994. "A real-time object model rto.k and an experimental investigation of its potentials". *IEEE Computer Society's 1994 Int'l Computer Software and Applications Conf.*, Jan.

[4] Kim, K., and Li, Y., 2003. "Toward easily analyzable sensor networks via structuring of time-triggered tasks". In FTDCS '03: Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03), IEEE Computer Society, p. 344.

[5] Kim, K. H. K., Im, C. S., Kim, M. C., Li, Y., Yoo, S.-M., and Zheng, L. C., 2004. "A software architecture and supporting kernel for largely synchronously operating sensor networks". In DIPES, pp. 133–144.

[6] Kim, K. H. K., Fujiwara, K., Kim, M.-C., Zheng, L., Watanabe, K., and Takizawa, M., 2007. "A ttf-based programming model and a support kernel running on a communicating sensor platform". In ISADS '07: Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems, IEEE Computer Society, pp. 368–376.

[7] ZDM, 2007. Embedded systems market. Ziff Davis Media Inc.; Electronic publication. http://linuxdevices.com/news/NS9103141896.

[8] Kim, K. H. K., 1997. "Object structures for real-time systems and simulators". *Computer,* **30**(8), pp. 62–70.

[9] Kim, K. H. K., Ishida, M., and Liu, J., 1999. "An efficient middleware architecture supporting time-triggered message-triggered objects and an nt-based implementation". In ISORC '99: Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society, p. 54.

[10] Kim, K. H. K., 2000. "Apis for real-time distributed object programming". *Computer,* **33**(6), pp. 72–80.

[11] Dibble, P. C., 2002. *Real-Time Java Platform Programming.* Prentice Hall.

[12] Molnar, I., 2007. Linux real time preemption patch. Electronic Publication. http://www.kernel.org/pub/linux/kernel/projects/rt/.

[13] Labrosse, J. J., 1998. *Microc/OS-II: The Real-Time Kernel.* CMP.

[14] Montavista, 2007. Montavista linux professional edition 5.0. Electronic Publication. http://www.mvista.com/downloads/.

[15] Liu, J. W. S. W., 2000. *Real-Time Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[16] Maxim, 2001. Application note 569. accuracy: What do you really need? Maxim Integrated Products.

[17] Mills, D. L., 1994. "Internet time synchronization: The network time protocol". In *Zhonghua Yang and T. Anthony Marsland (Eds.), Global States and Time in Distributed Systems, IEEE Computer Society Press.*

[18] Stankovic, J. A., Ramamritham, K., and Spuri, M., 1998. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA.

[19] Valois, J. D., 1996. “Lock-free data structures”. PhD thesis, Troy, NY, USA.

[20] Fujiwara, K., 2006. “Design and implementation of an operating system kernel supporting global-time-triggered functions”. Master’s thesis, UC Irvine.

[21] Dipartimento di Ingegneria Aerospaziale, P. d. M., 2005. Rtai - the realtime application interface for linux. https://www.rtai.org.

[22] Lackorzynski, A. L4linux, running linux on top of l4. http://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[23] Bovet, D. P., and Cassetti, M., 2005. *Understanding the Linux Kernel*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, December.

[24] Gorman, M., 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[25] Masmano, M. Tlsf: Memory allocator for real-time. http://rtportal.upv.es/rtmalloc/.