Real-Time Syst (2009) 43: 177–210
DOI 10.1007/s11241-009-9086-5

Enhancing a dependable multiserver operating system with temporal protection via resource reservations

Antonio Mancina · Dario Faggioli · Giuseppe Lipari · Jorrit N. Herder · Ben Gras · Andrew S. Tanenbaum

Published online: 5 August 2009

Abstract Nowadays, microkernel-based systems are getting studied and adopted with a renewed interest in a wide number of IT scenarios. Their advantages over classical monolithic solutions mainly concern the dependability domain. By being capable of dynamically detect and solve non-expected behaviours within its core components, a microkernel-based OS would eventually run forever with no need to be restarted. Dependability in this context mainly aims at isolating components from a spatial point of view: a microkernel-based system may definitely not be adopted in the context of real-time environments, simply basing on this kind of protection only.

One of the most active real-time research areas concerns adding temporal protection mechanisms to general purpose operating systems. By making use of such mechanisms, these systems become suitable for being adopted in the context of time-sensitive domains. Microkernel-based systems have always been thought of as a kind of platform not suited to real-time contexts, due to the high latencies introduced by the message passing technique as the only inter-process communication (IPC) facility within the system. With computer performances growing at a fairly high rate, this overhead becomes negligible with respect to the typical real-time processing times.

In the last years, many algorithms belonging to the class of the so-called Resource Reservations (RRES) have been devised in order to provide the systems with the needed temporal isolation. By introducing a RRES-aware scheduler in the context of a microkernel-based system, we may enrich it with the temporal benefits it needs in order to be deployed within domains with real-time requirements.

In this paper we propose a generic way to implement these mechanisms, dependent for a very small part on the underlying OS mechanisms. In order to show the

A. Mancina (✉) · D. Faggioli · G. Lipari
Scuola Superiore Sant'Anna, via Moruzzi 1, 56100, Pisa, Italy
e-mail: a.mancina@sssup.it

J.N. Herder · B. Gras · A.S. Tanenbaum
Vrije Universiteit, De Boelelaan 1081A, 1081 HV, Amsterdam, The Netherlands

generality of our RRES framework we implemented it in the context of MINIX 3, a highly dependable microkernel-based OS with an impressive users base.

Keywords Operating systems · Real-time systems · Resource reservations · Micro-kernel · Dependability

1 Introduction

Modern computer users are increasingly concerned about system dependability. While end-user requirements used to represent a trade-off between performance and costs, developers nowadays have to meet the demand for hard safety guarantees. This includes security and privacy, robustness against failures, timeliness of operation, quality of service, and so on. The dependability axis we explore in this work concerns the temporal domain. One particularly important problem in this domain is how to prevent applications from using excessive CPU time, thereby disrupting timeliness of operation and degrading quality of service.

A recent study (Tsafrir et al. 2007) showed that common process scheduling mechanisms can be subverted in a practical manner without superuser privileges in order to monopolize the CPU. This is a threat to not only time-sharing systems, but also embedded systems such as cell phones, PDAs, etc. The cheating process effectively gains the maximum priority, performing a denial of service (DoS) attack on other tasks. It was shown that almost all current operating systems, including MINIX 3 according to our analysis, are affected by this problem.

Furthermore, timeliness of operation is important in many application domains, including multimedia, VOIP, peer-to-peer services, interactive computer games and so on. Each of these domains has its own peculiarities, but all of them share an equal need of a minimum guaranteed service level. A best-effort service based on heuristic algorithms is usually adopted in order to improve the end-user perception of the overall quality, but this approach fails to provide minimum service guarantees. For example, in an attempt to keep the highest possible throughput, the performance of certain critical services may be heavily degraded under high-load conditions, which may lead to a low quality of service as perceived by the end user.

In order to improve the every-day user experience for such time-sensitive applications, a real-time operating system (RTOS) adapts the computational resources granted to each application based on its quality-of-service requirements. To date much development has focused on adding real-time features to commodity, monolithic, PC operating systems, such as Linux (Abeni and Buttazzo 1998; Kaneko et al. 1996; Rajkumar et al. 1998; Faggioli et al. 2008b).

As a matter of fact, monolithic kernels-based systems are not the best choice as far as dependability issues are concerned: a bug at kernel level may possibly lead to a complete system hang.

In contrast, the microkernel approach aims at separating at memory space level every running component, trying to confine problems in the context of the components themselves, eventually restarting the faulty process without jeopardizing the system as a whole. By acting so, a microkernel-based approach might eventually be able to run forever.

By introducing real-time features in the context of such systems, they could grant temporal protection next to the spatial one to every system component, thus preventing misbehaving processes from affecting other time-sensitive applications. The most important class of real-time algorithms, providing temporal protection capabilities is that of Resource Reservations (RRES).

1.1 Resource reservations

Resource reservations are a class of real-time algorithms that grant Q resource units every period P (Rajkumar et al. 1998). In principle, the resource can be any system facility, including CPU, memory, network and storage devices. However, we are interested in CPU reservations, which have proven to be an effective technique to serve time-sensitive applications on general-purpose operating systems (Abeni and Lipari 2002; Abeni et al. 2005).

Recently, a plethora of different RRES algorithms have been proposed and this led to several different mechanisms offering very similar performances.

Taking cognizance of this uncontrolled growth, we decided to rework the whole range of possible algorithms into a uniform logical scheme through the introduction of a taxonomy, aiming to ease the description and implementation of resource reservation algorithms.

Finally, we needed to choose an existing microkernel-based system, in order to prove the features of our proposed taxonomy, under the point of view of both simplicity of implementation and generality of description. We ended up choosing MINIX 3 both for its well-established users base and because of its widely recognized clean implementation and lightweight system organization which we deemed suitable for the introduction of these mechanisms.

1.2 MINIX 3

MINIX 3 is a microkernel-based multiserver operating system for uniprocessors that is designed to be extremely fault-tolerant. All system services run as highly restricted user-mode processes in order to isolate faults occurring in one component and prevent the damage from spreading, so that the rest of the system can continue to function normally. In addition, the extension manager can detect certain error conditions, including failures relating to CPU or MMU exceptions, internal panics or infinite loops, and restart faulty processes. These features greatly improve the system's dependability (Herder et al. 2006, 2007).

In addition to dependability, MINIX 3's highly modular structure makes it a good candidate as a real-time operating system for embedded platforms. Its code base is several orders of magnitude smaller than Linux, it is easy to remove unwanted components in order to get a minimal configuration, and the simple structure results in a small memory footprint. Moreover, MINIX 3 already has good response times due to the following design choices:

– the user-mode operating system servers and drivers have short servicing times and are fully preemptible by higher-priority processes,

– the kernel has very short interrupt latencies because its generic interrupt handler only masks the IRQ line and sends a notification message, whereas the actual interrupt handling is done by a user-mode driver, and
– finally, the kernel has short atomic kernel calls, which results in low stuck-in-kernel latencies.

However, MINIX 3 did not yet explicitly address other real-time application requirements. Realizing real-time behavior is not straightforward, since standard MINIX 3 versions lack important real-time properties, including:

– a way to describe a task's real-time constraints and schedule it accordingly,
– a temporal profile of each component in the system in order to achieve a complete system predictability, and
– typical resource access protocols, such as Priority Inheritance (Sha et al. 1990) or Stack-Based Resource Protocol (Baker 1990), in order to avoid priority inversion phenomena.

To the best of our knowledge, we are the first to implement resource reservations in MINIX 3. The new resource reservation framework improves MINIX 3 in three important ways:

1. RRES brings soft real-time support at least, so that benefits can be gained in many application domains, like the ones mentioned above. Infrequent deadline misses are tolerable due to the nature of soft real-time applications; the end user will perceive a missed deadline as a quality-of-service degradation rather than a fatal error.
2. Although our primary focus is *soft* real-time support, the RRES framework also provides limited *hard* real-time support for applications that do not rely on the standard system servers and drivers, such as sensoring applications using memory-mapped I/O. The only critical code is the kernel's generic interrupt handler, which has a short, strictly bounded execution time.
3. Our work improves dependability by enabling temporally isolated execution in order to prevent denial of service attacks (Tsafrir et al. 2007). Reliable accounting is realized by using the TSC cycle counter independent from the programmable interrupt timer (PIT), as detailed in Sect. 5.4.

1.3 Paper outline

The remainder of the paper is organized as follows. Section 2 briefly surveys related work. Section 3 introduce the CBS, CBS-HR and IRIS resource reservation algorithms. In Sect. 4 we introduce and describe the global resource reservation framework (GRRF), while Sect. 5 describes how we implemented it. Sections 6 and 7 present a case study and the results of performance measurements on the introduced latency. Finally, Sect. 8 describes the current framework status and our planned future work.

2 Related work

We distinguish different operating system structures, since each structure leads to different real-time properties.

2.1 Monolithic operating system structure

In spite of significant research efforts, introducing real-time support in monolithic systems, such as Linux, is still considered an open problem. Real-time scheduling turned out to be difficult, mainly due to the presence of many other highly unpredictable system activities, such as interrupt handling, paging and process management.

Two approaches have been adopted in order to minimize latencies and improve response times. First, shortening non-preemptible kernel code sections. This changes local code sections, but keeps the same monolithic kernel structure. As an example, Red Hat staff has contributed a series of kernel *low-latency patches* to the Linux community.[1] The patches have proven to be effective and are a substantial step towards a real-time Linux.

Second, introducing an additional real-time layer between the operating system and the real hardware in order to actively handle real hardware interrupts and mask them to the operating system when needed. This results in an hybrid architecture with a monolithic kernel running on top of a microkernel layer. The most important projects are RTAI,[2] RT-Linux[3] and Xenomai.[4] All these projects adopt a similar approach to the problem: a new interrupt dispatcher is added below the standard kernel which traps the peripheral interrupts and reroutes them to Linux whenever it is necessary. However, this approach means that real-time tasks cannot directly access standard Linux services and existing device drivers due to potentially high and unpredictable delays. For this reason, developers often have to (re)write their own real-time drivers.

2.2 Multiserver operating system structure

Real-time work also has been done in the context of multiserver systems. Here, low interrupt latencies and good response times are easier to achieve than in a monolithic system, since all services are already scheduled independently. Below, we discuss related work in three systems.

Resource reservations and temporal protection have been tested before on Real-Time Mach (RT-Mach) (Mercer et al. 1993, 1994; Tokuda et al. 1990). RT-Mach enforced the concept of resource reservation using a fixed-priority scheme like RM (Liu and Layland 1973), which cannot achieve full CPU utilization or, at most, a dynamic-priority one based on old algorithms like TBS (Spuri and Buttazzo 1994). MINIX 3 implements the newer CBS, CBS-HR and IRIS algorithms, which, in contrast to RT-Mach TBS, are able to correctly cope with aperiodic activities whose Worst-Case Execution Time (WCET) is not known a-priori. Furthermore, RT-Mach seems to have the scheduling policy hard-coded at kernel-level, whereas we promote a minimally invasive, modular design.

[1] Ingo Molnar's RT Tree. http://www.kernel.org/pub/linux/kernel/projects/rt/.

[2] RTAI home page. https://www.rtai.org/.

[3] RTLinux home page. http://www.rtlinux.org.

[4] XENOMAI home page. http://www.xenomai.org.

Real-time support in L4 (Liedtke 1996) is based on the statistical approaches Quality-Assuring Scheduling (QAS) (Hamann et al. 2006) and Quality-Rate-Monotonic Scheduling (QRMS) (Hamann et al. 2007). By extracting task properties, the system can guarantee that the deadlines of the mandatory part are met, while deadline misses in the optional part are tolerated. However, in order to enforce the mandatory-optional splitting principle, DROPS' real-time applications require modifications at source code level, whereas our framework can directly serve any existing applications in a real-time fashion. Furthermore, QAS and QRMS can provide guarantees for only periodic tasks, whereas CBS, CBS-HR and IRIS also support aperiodic tasks with real-time requirements. We also believe that our implementation can be simpler, since no complexity is introduced at admission and reservation level, whereas QAS performs these tasks using the distribution of execution times.

Finally, two projects based on earlier versions of MINIX should be mentioned. First, Minix4RT (Pessolani 2006) aims to mimic the low-latency RT-Linux architecture in MINIX 2. Second, RT-Minix (Rogina and Wainer 1999, 2001) consists of a set of system calls added to MINIX 2 in order to explicitly invoke real-time services provided by the kernel level. The former project has been made obsolete by MINIX 3, since its generic interrupt handler achieves low interrupt latencies in a much simpler way. Furthermore, these approaches are too invasive with respect to the base system and cannot be easily ported to MINIX 3. Our work provides the first-ever implementation of resource reservations and temporal protection based on CBS, CBS-HR and IRIS in the context of MINIX 3.

3 Resource reservations

Resource reservations are a powerful concept providing *temporal protection* for time-sensitive applications. The underlying idea is to reserve a fraction of the CPU in order to ensure isolated execution. Resource reservations are typically used to run both periodic and aperiodic tasks, since they allow the scheduler to enforce classical *Earliest Deadline First* (EDF) (Liu and Layland 1973) scheduling decisions, even in presence of misbehaving tasks that execute longer than expected or unexpectedly introduced tasks that impose a temporary increase on the global utilization.

Before we continue, we briefly introduce EDF, which is the most widely adopted uniprocessor real-time scheduling algorithms in the dynamic priorities field. The EDF algorithm states: "for each time t, the task with the earliest absolute deadline is executed." Despite a higher computational complexity than *Rate Monotonic* (RM) (Liu and Layland 1973), which is the industrial standard for the fixed priorities field, EDF can always achieve full CPU utilization without any deadline misses, which is an important goal in our work (RM can reach a full utilization only through an accurate and specific choice of tasks' periods).

3.1 Achieving temporal protection

Temporal protection refers to the scheduler's ability to prevent one task from affecting the execution of other tasks by executing longer than expected due to, for example,

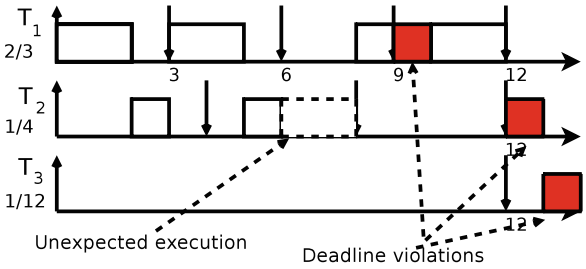


Fig. 1 Produced schedule with misbehaving task T_2. The tasks' computation times and deadlines are shown at the *left*

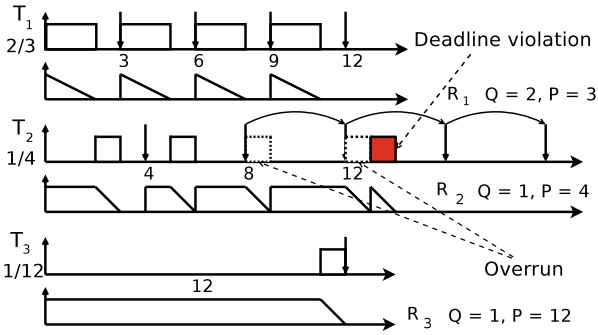


Fig. 2 Same schedule with temporal protection. The VRES' timeline is shown below the task's timeline

a programming bug. Traditional real-time operating systems do not protect against such schedule overruns, as depicted in Fig. 1. Here, the overrunning task T_2 that is scheduled according to the EDF algorithm causes the other tasks to violate their deadlines in a so-called *domino effect*.

In contrast, the use of resource reservations provides temporally isolated execution environments in which all tasks can complete within their deadlines despite of the overrun of the faulty task. The mechanism that enforces the reservation is referred to as a *Virtual Resource* (VRES) R that grants a CPU budget Q for each period P. An overrunning task can request additional CPU budget from its VRES, but this causes its deadline to be postponed by one period—so that other tasks with an earlier deadline are scheduled first. Figure 2 shows how this happens three times for task T_2. In the end, T_2 misses a deadline, but T_1 and T_3 run unaffected. We describe such an environment as compartmentalized from the scheduling point of view.

3.2 Resource reservation algorithms

In this section we are going to focus on some algorithms in particular, since the most of them share many common features and it would not make any sense to threat them separately. In contrast, we try to point out the differences among three sample algorithms which, in our opinion, well represent the variety of different reactions to typical RRES events.

We chose to describe in detail the CBS (Abeni and Buttazzo 1998), for its outstanding innovation in the RRES domain, being the first to be able to serve aperiodical requests with no a-priori knowledge about their WCET. Then, we chose its most direct derivative, CBS-HR, because of its sharp difference when serving cpu-intensive

processes. We, finally, picked up IRIS (Marzario et al. 2004), for its reclaiming capabilities, that make it different from the other two.

Other algorithms have been implemented in the context of this work, such as GRUB (Lipari and Baruah 2000) and CASH (Caccamo et al. 2000), but we chose not to include them in this description because we did not think they would have add any specific value to the presented results.

3.2.1 CBS

The *Constant Bandwidth Server* (CBS) (Abeni and Buttazzo 1998) is a resource reservations algorithm with dynamic priorities that uses the EDF algorithm at the lowest level. CBS can achieve full CPU utilization and solves many classic real-time scheduling problems, such as managing unpredictable instances of aperiodic tasks. We briefly recall the algorithm here:

1. each virtual resource (VRES) is assigned a maximum budget Q, a period P, a current budget c and a current deadline d;
2. a virtual resource is *active* if its task is active, *inactive* otherwise; initially, all virtual resources are inactive, and $c = 0$ and $d = 0$;
3. when a task is activated:
   - if $d \leq t$ or $c > (d - t)\frac{Q}{P}$, then $c = Q$ and $d = t + P$,
   - else, the current scheduling parameters are used
4. at each time t, the active virtual resource with the earliest current deadline d is chosen, and its task gets executed;
5. as long as T runs, the budget c of the virtual resource decreases at a rate $\delta c = -\delta t$;
6. whenever the virtual resource budget is exhausted ($c = 0$), it is immediately recharged ($c = Q$) and its deadline is postponed ($d = d + P$); as a consequence, rule 4 is applied and another virtual resource might be scheduled.

Since the CBS algorithm is based on EDF, and the virtual resources can be approximated as sporadic tasks with a worst-case execution time Q and minimum interarrival time P, it is possible to allocate 100% of the processor bandwidth. In addition, the CBS postponing scheme provides temporal protection against overrunning tasks. CBS rule 6 ensures that the priority of a misbehaving task is decreased by postponing the deadline of its virtual resource. The task is kept in the ready queue, but cannot execute if other tasks with an earlier deadline exist, as shown in Fig. 2.

3.2.2 CBS-HR and IRIS

Due to its simple reclamation scheme, the CBS algorithm suffers from a problem called *deadline aging* (Marzario et al. 2004). If a CPU-bound, non-real-time task T_1 (e.g. a compilation with *gcc*) is the only active task in the system, CBS' deadline-postponement rule is continuously triggered for R_1. Under the assumption that T_1 was granted only a fraction of the CPU, its deadline will be somewhere in the far future after consuming several budgets Q. If another task T_2 (e.g. *bunzip2*) starts executing, it will have the highest EDF priority for a long time, during which T_1 cannot execute. Hence, the end user will perceive T_1 as a non-responsive task.

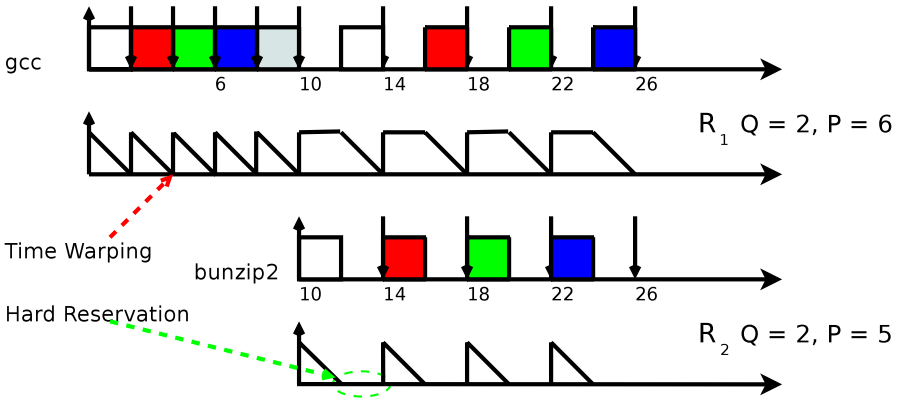


Fig. 3 IRIS solves the deadline aging problem by having VRESes wait until the next deadline before replenishing their budget. The replenishment can be instantaneous due to time warping

The problem of deadline aging has been addressed by CBS-HR through a concept known as *hard-reservation mode*. If the virtual resource's budget is exhausted, replenishment only happens at the beginning of the next period. This ensures that the virtual resource's deadline is not repeatedly postponed and stays synchronized with respect to task execution. However, CPU cycles may be wasted while recharging, which led to the notion of *time warping* in IRIS (Marzario et al. 2004). CBS-HR extends the standard CBS policy with rule 7, whereas IRIS extends CBS-HR with rule 8:

7. when budget c is exhausted, the task is suspended and the virtual resource moved to the *recharging state* until the current deadline d, when the budget is replenished to $c = Q$ and the deadline postponed to $d = d + P$;
8. if all virtual resources are in recharging state at time t and no virtual resource is currently active, they can be all recharged and their deadlines updated to $d = t + P$.

These enhancements result in a more responsive system and a better reclamation policy, respectively. As an example, Fig. 3 shows how IRIS prevents deadline aging for the above scenario of two aperiodic, CPU-bound tasks.

4 Generic resource reservation framework

By now, it should be clear that the real-time system designer has many alternatives which he may choose among when selecting what types of VRESes should be set up to serve the real-time taskset. This complexity is seldom an insurmountable obstacle, since many different algorithms may successfully perform even though with very different tasksets. The main differences between equally suited algorithms to the active context reside in the amount of possible resource wasting, in the typical evaluation parameters like the mean tardiness (which directly affects the perceived quality as by the end user) and in the implementation complexity of the chosen algorithm.

In order to greatly simplify the latter point and the design of new RRES algorithms, we conceived a new framework for resource reservation algorithms. In the authors' opinion, this framework greatly improves the abstraction capabilities as far as design, conception and implementation of RRES algorithms are concerned, since it exports a common interface to the system designer and algorithm developer.

In this sense we are speaking more of a taxonomy than a real programming framework, meaning that we are giving a generic way to describe different algorithms in the same domain.

4.1 State diagram

We decided to base the framework upon the minimum number of VRES states we deem capable to describe every possible algorithm, by computing the necessary mathematical and logical operations.

By making use of its properties, the system designer is able to describe every possible running condition, regardless of the way the actual algorithms enforce it.

As an example, consider the way in which different algorithms put their reclaiming properties in action: some of them let CPU idle time be assigned to currently running VRESes, some others borrow unused budget from currently inactive VRESes. We would aim at using just our diagram with its states, events and transitions to describe this particular phase, confining the conditions to start reclaiming at a lower level, along with the specific implementation which enforces this reclaiming.

The state diagram of Fig. 4 expresses all the possible states of a VRES. Besides explaining each of them, we will analyze the set of events and consequent transitions towards the considered state.

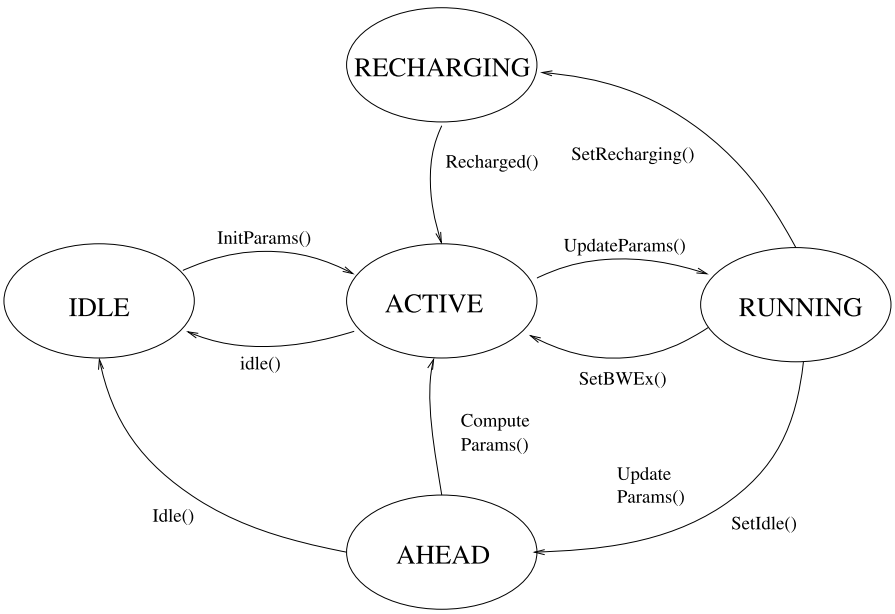


Fig. 4 Generic framework state diagram

– *IDLE*, to describe every existing VRES currently not backlogged, that is which has no ready process to serve (every VRES just created goes into this state);
– *ACTIVE*, to describe the condition of a VRES not at the highest priority in the system, with a ready-to-run process in it;
– *RUNNING*, to describe a VRES currently serving its task, thus decreasing its budget, unless an end condition occurs;
– *RECHARGING*, to describe a backlogged VRES with no budget, waiting for a budget recharging event to occur in order to start serving its task over;
– *AHEAD*, which is used in case a VRES is not currently backlogged but has consumed more budget than its "fluid" equivalent (the sources of this budget are of very different nature, as we will see). It is useful to describe several atypical conditions, like budget reclaiming, stealing and other temporary non-standard activities.

To further refining our framework description, it is necessary to speak of *events* and *transitions*: events determine an action which ends up triggering a transition between an old state and a new one.

A list of possible events follows:

`onTaskBirth` at the end of the creation phase of a new task in the system;
`onTaskReady` when the new task is ready to run (it has backlogged jobs) and the corresponding VRES has been created;
`onBudgetExhausted` when the current task instance (or job) consumes all the budget reserved for the current VRES period;
`onBudgetRecharged` in case VRESes do not get an immediate recharge of their budgets, this happens when the budget is completely recharged;
`onTaskBlock` when the current job experiences a block due to shared resources or explicit signals;
`onTaskUnblock` when, after having been blocked, a job restarts for the blocking condition does not hold any more;
`onTaskDeath` when an application completes its execution or for an abnormal terminating condition (a signal or an exception).

Depending on the events and on the current VRES state, one of the following transitions may take place:

idle2active occurs on `onTaskBirth` events;
active2running occurs on `onTaskReady` events;
running2active occurs when a higher priority task preempts a lower priority one;
running2ahead occurs on `onTaskBlock` events;
running2recharging occurs on `onBudgetExhausted` events;
gen2idle depending on the current VRES state, it occurs when a VRES is not backlogged (it has no ready jobs);
recharging2active occurs on `onBudgetRecharged` events;
ahead2idle occurs when the current VRES parameters expire, meaning that it is nonsense to save them for a later use (we will see what it means later on);
ahead2active occurs when a new task instance arrives while the VRES is in the ahead state and the current parameters may be somewhat maintained or updated;

Finally, there are globally shared operations which must be taken into account in every algorithm, along with specific steps not considered here:

InitParams() used to assign the correct values to the VRES parameters during the creation phase;
UpdateParams() used to compute the new VRES parameters following important algorithms events;
ComputeParams() as above, but with the additional computation of a test in order to decide whether UpdateParams() has to be invoked or current parameters may be exploited;
Idle() used when the VRES is not backlogged any more, that is no more jobs are ready to start;
SetIdle() is used when the current job instance stops, following a special blocking condition like a busy shared resource, an explicit blocking signal or a voluntary sleep;
SetBWExt() used when the current job gets preempted by an higher priority job to save the current parameters after an execution time frame;
SetRecharging() used in case a specific event must be waited for (whether this event is actually of recharging or more general type is left to the specific algorithm implementation);
Recharged(), invoked when the time specified through the SetRecharging() interface has been reached.

We will see, from time to time, how these operations are carried out in the context of the specific algorithms implementations.

4.1.1 Mappings in GRRF

In this section we are going to analyze the way in which the algorithms described in Sect. 3.2, get mapped on the state diagram just analyzed.

GRRF: CBS Here the mapping is quite simple:

- the IDLE state maps directly on the CBS IDLE state;
- the ACTIVE state maps directly on the CBS ACTIVE state;
- the RECHARGING state is not used;
- the RUNNING state maps directly on the CBS RUNNING state;
- the AHEAD state is used when a task instance ends and its virtual resource has still some budget. In particular, let q be the current residual budget and U the VRES utilization, if $(\frac{q}{U} \leq d - t)$, then the VRES is put and stays in the AHEAD state until $(\frac{q}{U} = d - t)$, time at which the VRES goes into the IDLE one. As long as it stays in the AHEAD state, if a new task instance is ready to run, it can directly go to the ACTIVE state.

In Fig. 5 the state diagram equivalent for the CBS case is depicted. Being the concept of budget recharging of no utility, the RECHARGING state has been removed and a circular arrow starts from and ends onto the RUNNING state, through a simple `updateparameters()` operation.

Let us consider the example of Fig. 6 and explicitly analyze the distinct phases the framework passes through.

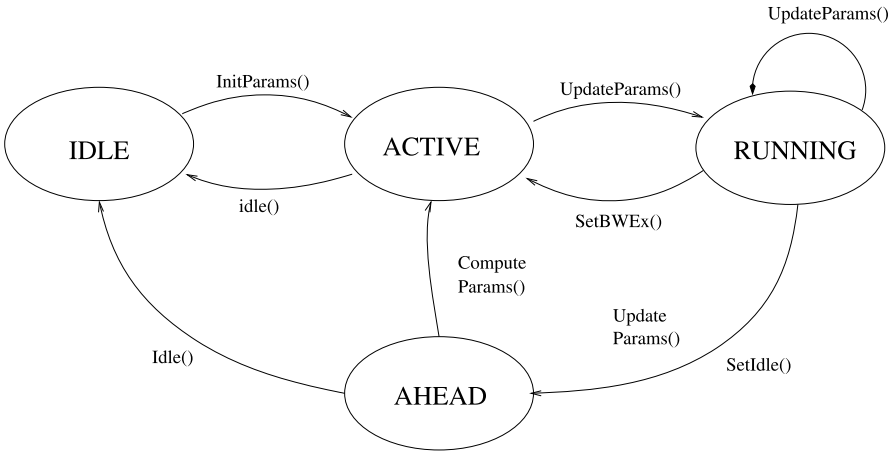


Fig. 5 The state diagram for the CBS algorithm

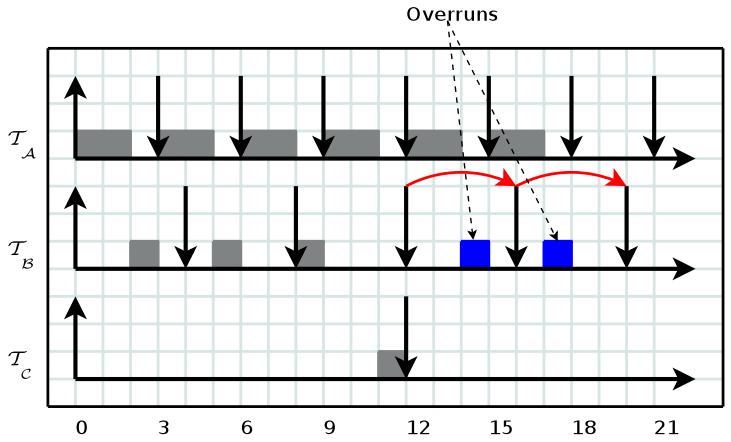


Fig. 6 A sample CBS schedule to show the GRRF in action

At time $t = 0$, three new task are born, so that three onTaskBirth events are fired up. The framework behaves invoking the corresponding generic part of this event handler which, among other activities, takes care of setting up the initial parameters (InitParameters()). It also invokes the corresponding idle2active () which, in this case, is translated into the cbsactive () function call.

At time $t = 2$, τ_A blocks (or, equivalently, its current instance completes). Thus, the framework invokes the onTaskBlock() handler which takes care of moving it to the AHEAD state (through the running2ahead()) and selecting the new VRES to be put in execution.

An analogous reasoning may be carried on at time $t = 3$, when τ_A wakes up (a new task instance is ready to run). The framework ends up calling the ahead2active () followed by a active2running () which makes τ_A's VRES start.

At time $t = 9$, a BUDGET_EXHAUSTED event occurs (the task is possibly misbehaving and asking for further execution), so that the onBudgetExhausted handler gets called. It is immediately translated into the corresponding cbs equivalent, cbsBudgetExhausted(), which works recharging immediately the VRES budget and postponing the current deadline (at time $t = 16$), according to the algorithm rules. As a matter of fact, this implies a priority drop and prevents the task from delaying other tasks execution.

GRRF: IRIS In the IRIS mapping the IDLE, ACTIVE, RUNNING and AHEAD states have exactly the same meaning as in the CBS. The RECHARGING state is directly mapped on the IRIS RECHARGING state.

An important feature of IRIS is the *Time Warping* rule taking place every time there are VRESes in the RECHARGING and IDLE state only (see Sect. 3.2.2).

To model this behavior, it is sufficient to issue a check every time a RUNNING → RECHARGING state transition occurs. If the state-changing VRES is the last in the RUNNING state and no other one is in the ACTIVE queue, then this rule is triggered and every parameter is updated accordingly.

5 Design and implementation

This section describes how we implemented the key components at kernel and user level in MINIX 3 along with the specific implementation of the three algorithms previously described in the context of the GRRF.

Three important design guidelines for the implementation of the framework were:

1. pluggable real-time support next to best effort;
2. minimizing the amount of intrusive kernel code;
3. maximizing the policy-mechanism separation.

First, we did not want to break the standard MINIX 3 distribution for reasons of acceptance and backward compatibility. Therefore, we designed the framework as an optional component that can be started at run-time to enhance the system with real-time support when needed. Second, a general dependability strategy in MINIX 3 is to move as much code as possible out of the kernel into user space. Since kernel-mode code runs with all privileges of the machine it must be fully trusted, whereas user-mode bugs may be confined to the process in which they occurred. Third, separating the scheduling policies from mechanisms leads to a flexible, easily adaptable system. Fortunately, these guidelines go hand in hand, as discussed below.

5.1 High-level design overview

Based on the above design criteria we decided to introduce a separate user-space component, called the RRES manager or *RRES* for short, which is logically located at the MINIX 3 server level. RRES can be started through the MINIX 3 extension manager at run-time like all other extensions (Herder et al. 2007). The basic idea then is to let the kernel execute user-space scheduling requests for real-time applications

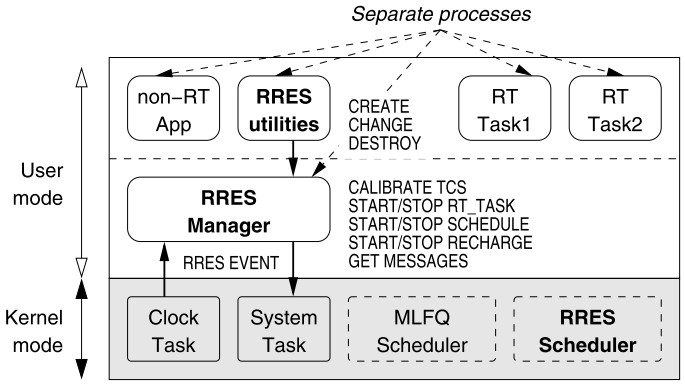


Fig. 7 High-level architecture over the resource reservation framework. Messages exchanged between the RRES helper utilities, RRES manager and kernel are shown

on behalf of RRES. In particular, the kernel's built-in best-effort scheduling policies should be temporarily suspended, so that the real-time task is not affected by the heuristics of the standard scheduler. In other words, the scheduling policy is enforced in user-space, but the kernel provides mechanisms for starting and stopping a task and for accounting its execution. Logically, this leads to an enhancements of the standard MINIX 3 scheduler to cope with RRES extensions.

The algorithm used is dynamically chosen at run-time. Although being possible to let different VRESes serve their task using different algorithms at the same time, this is not actually a viable solution, since the theoretical analysis to make this possible is still in progress.

In addition to the RRES manager, three helper utilities were created in order to manage real-time applications. First, *rres_create* can be used to start a new real-time application by passing the binary's name its period P and budget Q. Second, *rres_change* can be used to change the scheduling parameters at run-time. Third, the *rres_destroy* utility can be used to stop a running real-time task. Figure 7 gives a high-level overview of the RRES framework.

5.2 Implementation of the RRES manager

The RRES manager has the same code structure as other MINIX 3 servers. After the initialization of its data structures, RRES starts a never-ending loop in which it accepts new requests, processes them and sends back an answer.

5.2.1 RRES data structures

The main RRES data structure has five scheduling queues for the virtual resources that are uniquely associated with the real-time tasks. The queues are ordered by increasing current VRES deadline, so that RRES can quickly decide which task to schedule based on the underlying EDF policy.

– The **ACTIVE** queue keeps track of ready-to-run VRESes. The first VRES on this queue is the currently scheduled one, that is, the associated task is the running process in the system.
– The **RECHARGING** queue comprises all the VRESes which exhausted their budget and need it to be replenished. This queue is only used for CBS-HR and IRIS. With plain CBS it is always empty since hard-reservation mode is not used. Conceptually, all VRESes in this queue are recharging, but RRES only sets a single alarm for the first recharging event.
– The **BLOCKED** queue contains the VRESes that blocked during their execution, for example, because they have to wait for some event to happen.
– The **AHEAD** queue is used for special events handling within the specific RRES algorithm implemented (like budget reclaiming or donating).
– Finally, the **INACTIVE** queue which contains just created or about to die VRESes.

5.2.2 RRES interactions

As shown in Fig. 7, the RRES manager has several interactions with both the RRES help utilities and the kernel tasks. The exact messages that are exchanged are shown in Fig. 8. First, the RRES helper utilities can request RRES to CREATE, CHANGE or DESTROY virtual resources. In order to prevent random tasks from changing their scheduling policy only the system administrator is allowed to send RRES requests. RRES verifies this by asking the MINIX 3 process manager for the requester's user ID.

Second, although RRES is responsible for the scheduling policy, it relies on kernel mechanisms to perform the actual RRES scheduling. In particular, the following messages are exchanged with the kernel's system task:

– CALIBRATE_TSC: used at RRES initialization time to determine the number of CPU cycles per microsecond; the kernel programs the timer to a known frequency, reads the TSC cycle counter start value, waits 1000 timer ticks, and reads the TSC end value.
– START_RT_TASK: tell that a process now is a real-time task and needs to be treated in a special manner.
– STOP_RT_TASK: inform the kernel that a real-time task has been destroyed so that special events related to this task are no longer forwarded to RRES.
– START_SCHEDULE: tell the kernel to start scheduling a real-time task using the RRES scheduler rather than the standard scheduler.

Fig. 8 Messages exchanged within the RRES framework

Helper Utility –> RRES Manager	RRES Manager–> Kernel task	Kernel task –> RRES Manager
1. CREATE	1. CALIBRATE_TCS	1. RRES_EVENT
2. CHANGE	2. START/STOP_RT_TASK	– budget exhausted
3. DESTROY	3. START/STOP_SCHEDULE	– recharge time
	4. START/STOP_RECHARGE	– task blocked
	5. GET_MESSAGES	– task unblocked
		– task exited

– STOP_SCHEDULE: issued whenever RRES needs to stop the currently scheduled real-time task.
– START_RECHARGE: if a VRES becomes the head of the RECHARGING queue, RRES schedules an alarm to be notified when the recharging time is reached.
– STOP_RECHARGE: used to handle a time warping event in IRIS and if the scheduling parameters of a currently recharging task are changed.
– GET_MESSAGES: whenever the kernel's mechanisms encounter a special event, as shown in Fig. 8, the RRES manager is notified with an RRES_EVENT message; the RRES manager then makes a callback to find out which event triggered the notification.

While this modularity brings many benefits with respect to flexibility, the message passing interactions between RRES and the kernel introduces a small latency. Experiments on a prototype implementation have shown, however, that the incurred context-switching overhead is not at all prohibitive, as discussed in Sect. 7.

5.2.3 The rres_server structure

In order to provide the framework with the maximum degree of flexibility, the GRRF takes advantage of a group of function pointers, living inside the C structure describing a VRES:

```
int (*inactive2active)(struct rres_server*);
int (*active2running)(struct rres_server*);
int (*running2active)(struct rres_server*);
int (*running2ahead)(struct rres_server*);
int (*running2recharging)(struct rres_server*);
int (*gen2inactive)(struct rres_server*);
int (*recharging2active)(struct rres_server*);
int (*ahead2inactive)(struct rres_server*);
int (*ahead2active)(struct rres_server*);
int (*admission_test)(struct rres_server*, int);
```

Each of these functions is called whenever a state change event occurs and represents a hook function every algorithm must implement in order to take the actions corresponding to a particular event. As an example, let us analyze the way the CBS-HR algorithm initializes these hooks.

```
s->algo_type = CBSHR;
s->ahead2active = cbshr_start_job;
```

As it is immediately clear, we are filling the structure fields with functions related to the Hard Reservation mode of the CBS. Finally, let us give a look at one of these hooks implementations.

```
PRIVATE int cbshr_start_job(struct rres_server *s)
{
    /* ... */
    if (cbs_test(s) == RRES_NB)
    {
        /* New parameters must be generated */
        s->tsc_C = mul64u(usec_value, s->Q);
        s->tsc_D = add64(rres_curr_time,
            mul64u(usec_value, s->P));
    }
    /* The old ones may be used */
    /* ... */
}
```

We are reactivating a job after it reached a stop condition. In this case we have to compute the result of the CBS test in cbs_test () and, depending on its result, enqueue it in the ACTIVE queue with different parameters. Similar code paths may be identified in all the other important algorithm conditions.

From the RRES server point of view, there is the code for events management:

```
case RRES_BUDGET_EXHAUSTED:
        result = onBudgetExhausted(rres_new);
        break;
case RRES_BUDGET_RECHARGED:
        result = onBudgetRecharged(rres_new);
        break;
case RRES_JOB_START:
        result = onJobStart(rres_new);
        break;
case RRES_JOB_END:
        result = onJobEnd(rres_new);
        break;
```

Here, whenever the kernel sends an event message to the RRES server, RRES parses the message and invokes the corresponding function. Job start and end functions represent the condition of unblocking and blocking of a process, respectively (since in a real operating system we have only a few examples of tasks with a real periodic nature).

Lastly, here is a code example for managing one of the previous events:

```
PUBLIC int onJobStart(struct rres_server *s)
{
        /* ... */
        return s->ahead2active(s);
}
```

This is the place where we invoke the specialized version of the transition function. According to the server nature, the correct function implementation gets invoked.

Similar mechanisms may be exploited to implement the whole variety of possible actions within the context of RRES algorithms. When a reclaiming property must be enforced, the RECHARGING queue and event insertion within it is the way to go.

5.3 Kernel and scheduler modifications

Scheduling in the standard MINIX 3 kernel is done on best-effort basis using a multilevel-feedback-queue scheduler (MLFQ) (Torrey et al. 2007). Processes with the same priority reside in the same queue and are scheduled round-robin. When a process is scheduled, its quantum is decreased every clock tick until it reaches zero and the scheduler gets to run again. To prevent starvation of low-priority processes, a process' priority is degraded whenever it consumes a full quantum. Since CPU-bound processes are penalized more often, interactive applications have good response times. Periodically, all process priorities are increased if not at their initial value.

As mentioned above, the kernel should bypass the standard scheduler for real-time tasks managed by RRES. Therefore, the MINIX 3 kernel and scheduler were changed in two ways. First, we added *rres_f* flag to the process structure in order to tell whether a task should be scheduled in the context of MLFQ or RRES. This flag is set when RRES sends a START_RT_TASK request to the kernel. Second, the scheduler data structure was extended with two new scheduling queues at the highest priorities, as shown in Fig. 9.

– **RRES_PRIO**: the highest priority in the system is now used for the RRES manager, so that it can always immediately react to the various kinds of events, such as budget exhaustion and budget recharged events. Depending on the kind of event RRES may schedule another real-time task. When RRES has processed the event, it returns to its main loop and blocks waiting for the next event—allowing a real-time task to run.

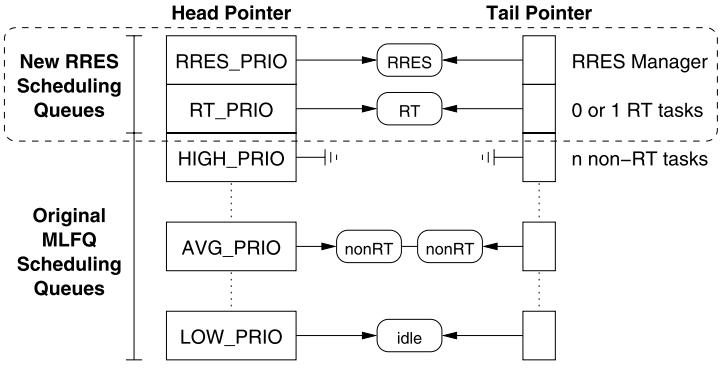


Fig. 9 RRES-enhanced MINIX 3 scheduling queue data structure. Two new queues at the two highest priority levels were added for the RRES manager and the current real-time task

– **RT_PRIO**: the second highest priority is reserved for the real-time tasks served by the RRES manager. At most a single task can be active at any given time. When there is a task to schedule, it runs uninterrupted until either its budget is exhausted or some other RRES event makes a higher-priority task ready to run. In the latter case, *preemption* occurs and RRES requests the kernel to schedule the higher-priority task.

Third, we identified the points which needed change in order to modify the default scheduler behavior. In particular, if a real-time task needs to be scheduled, that is, if a process' *rres_f* flag is set, the scheduler simply picks the queue with priority level RT_PRIO rather than its MLFQ priority. Also, a task running in the RT_PRIO queue is not affected by the heuristics of the normal MLFQ algorithm, such as decreasing the process priority of long-running processes and periodic balancing of the scheduling queues.

Finally, we changed the scheduler to cope with blocking and unblocking events. Whenever a real-time task blocks the kernel sends an event notification to RRES, so that it can schedule another task. Blocking can occur, for example, during synchronous service requests or while waiting for an I/O completion interrupt. We decided to consider a task's blocking and unblocking events as job completion and activation times (as in Abeni's work, Abeni and Lipari 2002) respectively in order to be able to provide the classic real-time properties previously described. The blocked task's VRES is put on RRES' BLOCKED queue. When the kernel notifies RRES that the task is unblocked, RRES moves the corresponding VRES to the ACTIVE queue and may schedule it depending on its current priority.

5.4 CPU time accounting

In order to serve real-time tasks the RRES framework requires a reliable source of high-precision timing. Our implementation is based on the x86's TSC cycle counter, but depending on the system architecture, other timing sources may also be available. The TSC cycle counter is convenient because it is accessible to both the user-space RRES manager and the kernel's scheduling code. However, since the TSC cycle counter is read-only and cannot interrupt when a task's budget is exhausted or needs to be replenished, an interrupt-based programmable timer is also needed. For this, we decided to modify the standard MINIX 3 system timer, which is based on the i8259 Programmable Interval Timer (PIT). Another option would have been to use the CMOS 'Real-Time Clock', but it is already in use for the MINIX 3 profiling code (Meurs 2006) and having two sources of timer interrupts would have complicated the kernel's code.

5.4.1 Working of RRES accounting

Although the PIT ticks come at a lower frequency than the TSC cycle counter, the RRES framework can do its work as follows. During initialization RRES calibrates the TSC cycle counter using the CALIBRATE_TSC in order to determine the number of cycles per microsecond. Budget exhaustion and budget replenishment events are expressed in CPU cycles rather than PIT ticks in order to prevent rounding errors in

the calculation. This number is reported to the kernel on START_SCHEDULE and START_RECHARGE, respectively, which stores the count in a global variable and compares it to the current cycle counter value on each PIT tick. If the current cycle counter value exceeds the exhaustion or recharging time, the kernel deschedules the task (in the former case only) and sends an RRES_EVENT notification to the user-space RRES manager.

One important decision was at which frequency the TSC counter should be read, that is, the PIT interrupt frequency—since a higher frequency leads to a lower worst-case accounting error. The maximum usable frequency is limited, however, since each PIT interrupt requires reprogramming the timer. After some experimentation we decided to use a PIT frequency of 4000 Hz, which limits RRES accounting error to at most 250 μs. Moreover, task overruns are taken into account by the RRES manager by reading the TSC cycle counter after the RRES_EVENT notification, comparing it with the original deadline, and reducing the task's CPU budget in its next execution frame.

Although RRES accounting works at 4000 Hz, we used a frequency of 500 Hz for the system's normal tick facility. This distinction takes place in the clock task's interrupt handler, which scales the hardware PIT frequency into lower-frequency system-wide ticks, that is, only 1 in every 8 interrupts is transformed into a system tick.

5.4.2 Eliminating CPU monopolization

An important benefit of our design is that denial of service (DoS) attacks that monopolize the CPU (Tsafrir et al. 2007) are structurally eliminated. By basing accounting on the actual number of CPU cycles used, independent of the PIT ticks, a task can no longer cause another task to be billed by suspending execution just before a PIT tick occurs. In contrast, whenever a task served by RRES stops execution, the RRES manager is informed and the current TSC cycle counter is read to decrease its remaining budget with the number of CPU cycles consumed. Processes that use MINIX 3's standard scheduling facilities are still vulnerable, but real-time tasks and, in fact, any application with stringent timing requirements can use the new RRES framework for temporal protection.

5.5 Abstracting from MINIX 3

Up to now, we spoke about a general taxonomy, meaning that by making use of it the implementation procedure of many rres algorithms gets noticeably simplified. We have just shown the implementation of some of the most employed algorithms within real systems. A question naturally arises: to what extent can this work be extended to other operating systems contexts?

At first glance the underlying scheduling system issue might seem to confine this experience to the specific implementation we just described. Actually, the GRRF makes just a few assumptions about what stands underneath: every algorithm mechanism resides at user-space level, thus resulting completely independent of the peculiar operating system which it is employed within.

In particular, the adaptation of the OS-dependent parts is comprised of a few modifications:

– let the RRES manager be the highest priority component in the system;
– base the scheduling time accounting on the TSC (this is valid for x86 architectures only) in order to have a timeline shared with the user-space level;
– provide a context-specific implementation of the different messages that the RRES manager may address to the kernel;
– provide a notification mechanism for the RRES manager to know about its former requests.

This description completely left out of consideration the specific nature of the operating system. When we speak of messages, in facts, we do not necessarily refer to the IPC mechanisms typically employed in the context of microkernel operating systems: these messages might equally be implemented by means of ordinary system calls and/or synchronous signals (if any).

6 RRES case study

To better clarify how the framework works, we now discuss an example that shows the interactions of the RRES framework, configured to use CBS with hard-reservation mode (CBS-HR). We analyze the sequence of events for two real-time tasks, T_1 and T_2, producing the schedule shown in Fig. 10. Initially, the administrator starts the tasks using the *rres_create* utility. The command entered is

$ rres_create <*budget*> <*period*> <*rresalgo*> <binary>

where the request parameters are

<*budget*>: CPU budget given in each period (Q) in µs;
<*period*>: the VRES granularity (P) in µs;
<*rresalgo*>: the RRES algorithm according to which the VRES must behave;
<*binary*>: the application to be managed by RRES.

This request has to be made for both task T_1 and T_2 with parameter $Q_1 = 3000$ µs, $P_1 = 9000$ µs and $Q_2 = 2000$ µs, $P_2 = 3000$ µs. The *rresalgo* parameter, for both tasks, was set to CBS-HR. The sum of the fractions $\frac{Q}{P}$ gives the CPU utilization and is 100% in this example.

For both tasks, the *rres_create* utility forks a new process, sends a CREATE message to the RRES manager to inform it about the new real-time task's parameters, and

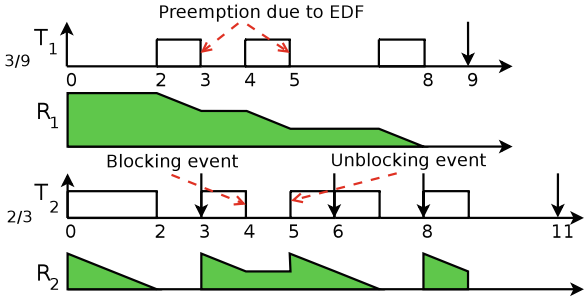


Fig. 10 Schedule of the case study in milliseconds

executes the binary. RRES first checks if the user is authorized and then performs an admission test (whose nature is dependent on the underlying scheduling algorithm and, with very minor exceptions, corresponds to its schedulability test). Since the CPU utilization does not exceed 100% (being EDF the underlying scheduling mechanism), RRES accepts the requests, creates two virtual resources R_1 and R_2 with the required parameters, and sends a START_RT_TASK message to the kernel to tell that T_1 and T_2 are real-time tasks from now on. The virtual resources, R_1 and R_2, will be enqueued in RRES' ACTIVE queue, with task T_2 at the head of the queue, since T_2's initial deadline is earlier than that of T_1.

We will now analyze the interactions between the RRES manager and kernel during the execution of tasks T_1 and T_2, which produces the schedule shown in Fig. 10. As discussed in Sect. 5.4, the RRES manager uses the TSC cycle counter for accounting. For reasons of simplicity, however, all times below are expressed in milliseconds.

At time $T = 0$, RRES issues a RRES_SCHEDULE request to the kernel specifying the task to be scheduled, in this case T_2, and the amount of CPU budget, that is, how long the task is allowed to execute, in this case 2. The kernel accepts the RRES request, sets up the time at which the budget is exhausted, and schedules the task in the queue with priority level RT_PRIO.

At time $T = 2$, the kernel notifies RRES about the budget exhaustion of T_2. RRES moves R_2 from the ACTIVE to the RECHARGING queue and, since hard-reservation mode is used, asks the kernel to recharge R_2's budget until the absolute time of R_2's deadline, $T = 3$. RRES also tells the kernel to schedule task T_1 with budget $Q = 3$.

At time $T = 3$, the kernel notifies RRES about R_2's budget being recharged, so that RRES moves it from the RECHARGING queue back into the ACTIVE one. Since R_2 has the earliest deadline, T_1 is preempted and RRES asks the kernel to schedule task T_2 with a budget of 2.

At time $T = 4$, task T_2 experiences a blocking event. The kernel notifies RRES, which in turn moves T_2's virtual resource, R_2, to the BLOCKED queue. Then RRES asks the kernel to resume execution of T_1 with a budget of 2.

At time $T = 5$, task T_2 unblocks. RRES is notified by the kernel and computes the test in CBS rule 3. Since the remaining budget $c = 1 \geq (6 - 5)\frac{2}{3} = \frac{2}{3}$ a new deadline is placed at $T = 8$ and the budget is recharged. R_2 is moved to the ACTIVE queue and task T_1 is preempted by T_2.

At time $T = 7$, R_2's budget is exhausted again. RRES is notified by the kernel, moves R_2 to the RECHARGING queue, and tells the kernel to recharge until $T = 8$. RRES also requests the kernel to resume execution of task T_1 with R_1's remaining budget of 1.

At time $T = 8$, two things happen: R_1's budget is exhausted and R_2's budget is recharged. R_1 is moved to the RECHARGING queue and the kernel is told to recharge the task until R_1's absolute deadline, $T = 9$. In addition, RRES ask the kernel to schedule task T_2 with a budget of 2.

This example shows how a user-space scheduler can do all the work using a small number of interactions with the kernel, obtaining the schedule produced in Fig. 10. In Sect. 7 we will see how these interactions impose a very limited timing overhead on the system.

7 Experimental evaluation

In addition to the case study shown in Sect. 6, we ran several experiments on a prototype implementation to evaluate the RRES framework. The results are presented below.

7.1 Timing measurements

As explained in Sect. 5.4, time accounting is done using the *TSC* cycle counter. The TSC facility is available in both kernel space and user space, allowing RRES to be kept synchronized with the kernel time line. In addition, this enabled precise timing measurements, depending on CPU speed only. The tests were conducted on a DELL desktop PC with a 2.4 GHz Intel Pentium IV CPU and 512 MB RAM. None of the tests required to access the disk.

First, we measured the latency introduced by MINIX 3's message passing subsystem, which is independent from the RRES framework. In particular, we measured the time between issuing a request in a user process (just before IPC_SEND) and the moment that the kernel starts working on it (just after IPC_RECEIVE), that is, the time purely spent on delivering the message from the user process to the SYSTEM task. We found a message delivery time of about 2 μs. This time has to be summed up to every interaction which takes place in the framework.

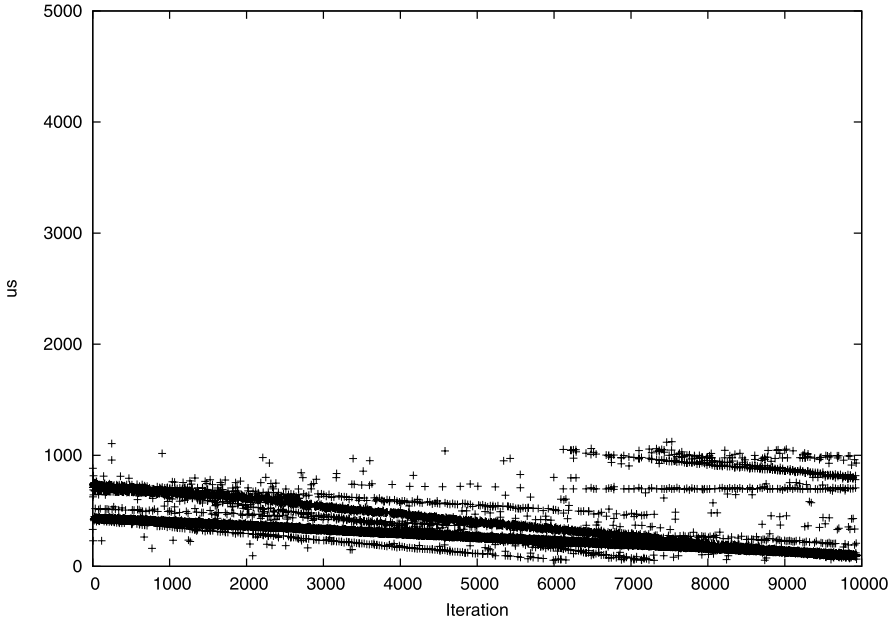
Second, we measured the latency introduced by the GRRF framework. We planned to separately measure the degree of performance hit in three different flavours:

- Time between receiving a *rres_create* or a *rres_destroy* command in the RRES framework and the moment in which the kernel schedules or stops scheduling the task respectively according to the new policy.
- Time between a RRES-related kernel event and the corresponding action which the RRES manager requires the kernel to enforce.
- Actual times spent within the kernel with respect to the actual times requested by the manager.

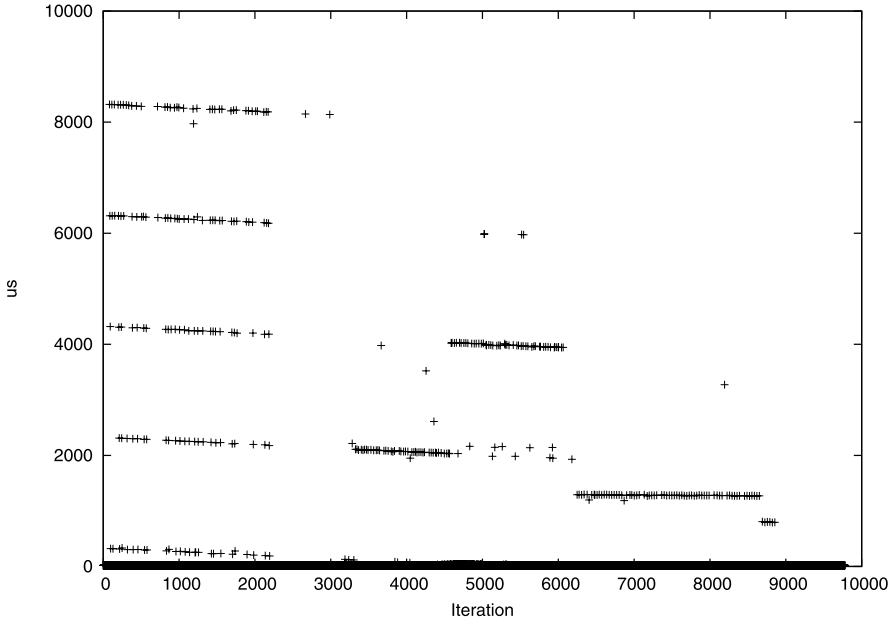
In order to measure the times needed for the creation phase, we simply created and repeatedly destroyed one VRES serving a cpu-bound task (here the nature of the process is not relevant, being the creation and destruction phases totally independent of it) for ten thousands time (see Fig. 11).

As far as running and recharging times measurements are concerned, we created a cpu-intensive task, triggering continuous budget exhaustions and recharging events with a (50 ms, 200 ms) budget/period couple. By letting it execute for a while, we took slightly less than 2000 samples shown in Figs. 12 and 13. This numbers represent the amount of actual time the system dedicates to serve a real-time task with the specified parameters, during its execution: the closer they are to the requested values, the more likely task deadlines will be met. The recharging values refer to the amount of time the task has to wait for before being scheduled over (so it has to be around 150 ms).

Finally, in order to evaluate the overhead imposed by the RRES manager when dealing with the algorithms events, we exploited the previous experiment and obtained the results shown in Fig. 14.



(a) Creation phase



(b) Destruction phase

Fig. 11 Times spent creating and destroying new sample task VRESes

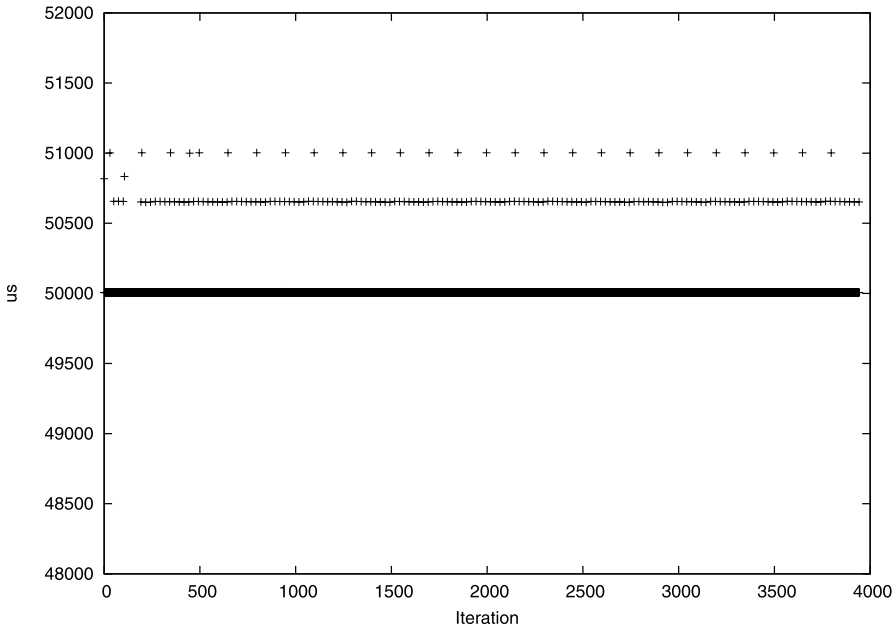


Fig. 12 Times spent in executing as long as the sample VRES budget gets exhausted

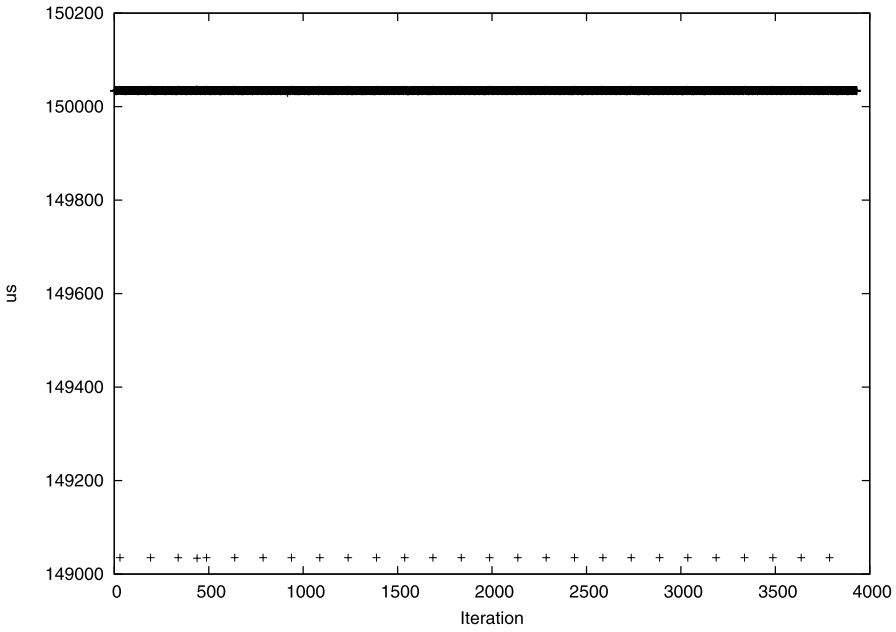


Fig. 13 Times spent in recharging the sample VRES budget

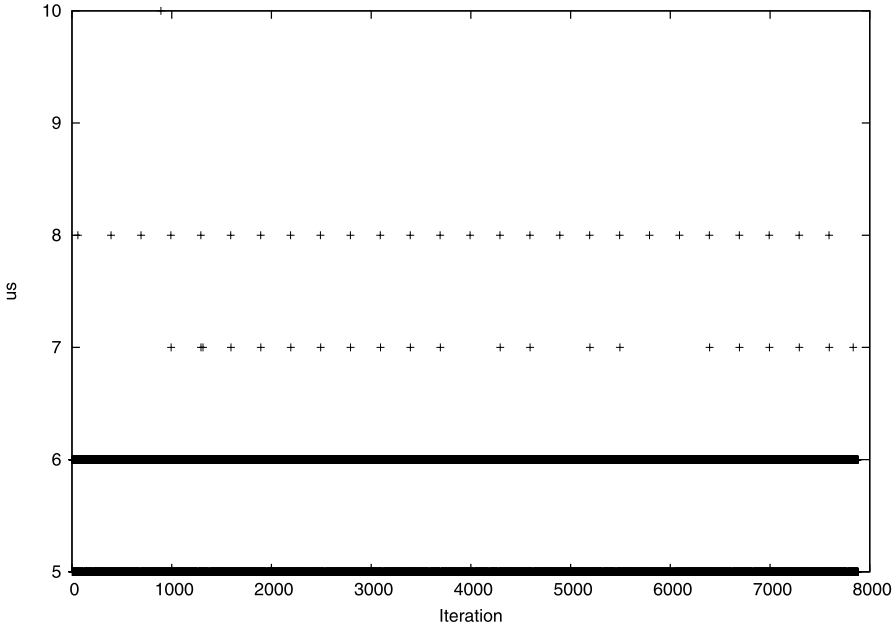


Fig. 14 Times spent within the RRES manager to manage the various kernel notifications

Table 1 Several statistics about the measurements taken. Times are expressed in µs where applicable

Measure	Avg, Var, StdDev, MaxValue
Creation	(334.74, 17088, 130.72, 1122)
Destruction	(139.8, 90.14, 9.5, 8320)
Running	(50038, 22458, 149.85, 51003)
Recharging	(150030, 32075, 179, 150037)
RRES computations	(5, 0, 0, 5)
Message passing	(2, 0, 0, 2)

Since it is evident how in all these experiments random system events may lead to some variable response times, we computed some statistics about these variations and presented them in Table 1. The most important parameters are the amount of running time a real-time task may get advantage of and the maximum delay it may be subjected to with respect to its activation period. In the first case we obtained an average standard deviation of 0.3%, with respect to the average value, and in the second case this value gets even lower, totalizing a 0.16%. This means that for each execution period, a real-time task might execute with a budget in a range equal to $\frac{[99.7,100.3]}{100} Q$ every period in a range of $\frac{[99.84,100.16]}{100} P$, with respect to (Q, P) as the nominal reservation parameters. Hence, it imposes an utilization in the following range: $\frac{[99.56,100.44]}{100} \frac{Q}{P}$. We deem this is a really negligible fluctuation for a system with soft real-time purposes.

An important observation is that, being the maximum average measured standard deviation lower than 200 µs, and being the RRES clock tick at kernel level manually set at 4000 Hz (e.g. 250 µs), the RRES framework seems not to impose a penalizing overhead to the used real-time parameters, thus not further limiting the resolution obtainable at user-space level.

As a further note, the overhead imposed by the message passing mechanism is clearly negligible if compared to the other parameters here considered.

It is important to realize that these values are not dependent on the presence of other real-time tasks, because (1) the kernel's interrupt handler always preempts running tasks and (2) messages that are exchanged upon RRES events are delivered and handled at the highest priority, as shown in Fig. 9.

The measured values have to be compared with the resolution the system is able to grant to the framework. Since time accounting is done at 4000 Hz, the minimum amount of budget and period can, in principle, be 250 µs. However, to prevent compromising the requested parameters, they should be at least an order of magnitude larger. Therefore, the budget and period should be set starting from 5–10 ms in practice.

7.2 Impact on kernel and user-space code

With the help of the Source Code Line Counter[5] tool available on the Internet we collected data on the total engineering effort required. The number of lines of source code for both the standard and modified version of the MINIX 3 kernel are shown in Table 2. Notice that the files shown here are the only that got modified: all the other ones, for a total amount of slightly more than $4k$, have not been changed. Similar statistics for the new user-space RRES manager are shown in Table 3.

These figures may be compared to the one shown in Table 4, with GRRF in place. Although there is a consistent increment in the total number of lines of code, we may notice how the generic common header and source files are decreased in length, since the complex algorithm management operations are demanded to the specific algorithm implementations to be found in the corresponding source file.

Table 2 Lines of source code (LoC) for the standard MINIX 3 kernel and the modified version with the RRES framework

File	Standard	RRES MINIX 3	Delta
proc.h	99	103	+4
proc.c	482	500	+18
clock.c	115	137	+22
system.c	314	327	+13
rres.h	–	24	+24
rres.c	–	197	+197
do_resres.c	–	131	+131
Total Changes			+339

[5]The Source Code Line Counter. http://www.cmcrossroads.com/bradapp/clearperl/sclc.html.

Table 3 Lines of source code (LoC) for the RRES server

Header Files	LoC
glo.h	42
inc.h	29
proto.h	51
rres.h	106
Header Total	228

Source Files	LoC
main.c	158
rres.c	543
rres_kernel.c	254
rres_userspace.c	251
Source Total	1206

Table 4 Lines of source code (LoC) for the RRES server with GRRF

Header Files	LoC
glo.h	47
inc.h	29
proto.h	67
rres.h	79
Header Total	222

Source Files	LoC
main.c	205
rres.c	510
rres_kernel.c	66
rres_userspace.c	227
rres_algo_CBS.c	179
rres_algo_CBSHR.c	190
rres_algo_IRIS.c	205
Source Total	1582

Furthermore, the new file organization is a lot clearer and more easily maintainable, since each algorithm implementation resides in a different file.

7.3 RRES tracer and simulations

We also created a tool written in Ruby to trace the execution of RRES real-time tasks. The tool parses a log file generated by the RRES server and produces a graphical representation of the scheduling decisions taken.

Figure 15 represents a piece of the scheduling of the task set in Table 5 which is scheduled according to CBS-HR (CBS with hard reservations); IRIS time warping is not used. The three horizontal lines represent each task's timeline. The tasks used are an infinite CPU-bound program (*cpuload*) which performs calculations in a loop and a finite I/O-bound program (*interactive*) that does some work, sleeps one second, and continues calculating. The tracer output shows three aspects:

- *cpuload* continuously triggers CBS' deadline postponement rule, as is clear in the first two task lines where arcs connect consecutive deadlines;
- since *interactive* has a large budget, it can execute whenever there is a free slot, unless it blocks on the sleep() system call;
- at that point, the hard-reservation mode becomes evident, since the two *cpuload* utilities run without time warping (the scheduling is not work-conserving).

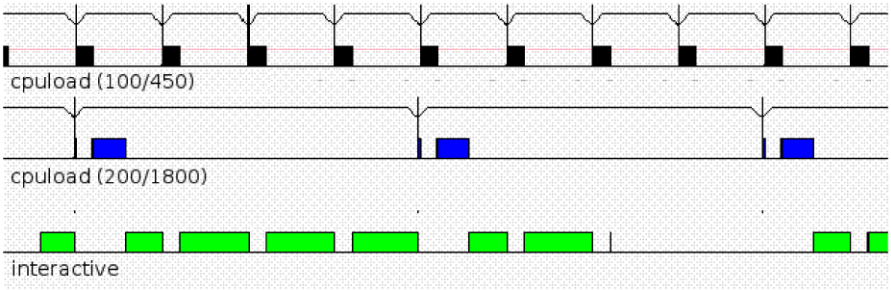


Fig. 15 Actual schedule executed for the task set of Fig. 5 produced by the RRES tracer based on RRES server logs

Table 5 Task set and reservation parameters used for tracer simulation. The execution is shown in Fig. 15

Task	Type	Budget (ms)	Period (ms)
cpuload	CPU-bound	100	450
cpuload	CPU-bound	200	1800
interactive	I/O-bound	10000	20000

Numerous other simulations have been run to verify the correctness of our implementation in few real cases, but we refrained from including them here, since they simply are comprised of enlarged task sets of very different nature whose scheduling sequences and performances are tightly dependent on the specific nature of the chosen scheduling algorithm.

8 Conclusions and future work

Resource Reservation algorithms constitute a really valid choice for providing operating systems with temporal protection capabilities. This kind of protection is definitely attractive for systems that must accommodate time-sensitive applications within them. Being the number of RRES algorithms really high, it is often not clear as to what extent an algorithm perform better than another one and why.

Basing our framework on the state diagram of Sect. 4, we are able to represent a number of very different resource reservation algorithms with an effective and unified programming interface. By refining this diagram, it is possible to represent resource sharing concepts as well, so far set aside from our analysis.

On the other hand, although representing an almost perfect solution as far as working dependability is concerned, most microkernel based systems do not offer such temporal protection capabilities. Being MINIX 3 one of the most adopted systems of this kind, we decided to experiment our framework in this context, in order to prove its pliability.

By performing many tests on the prototype implementations, we were able to determine that the old detriments about its IPC long latencies and slow response times simply do not hold any more and may be considered as negligible with respect to typical real-time tasks parameters.

Our design enables running soft real-time applications on top of MINIX 3. The current status is that correct time accounting happens in presence of non-blocking tasks. If blocking events occur, the framework operates correctly under the assumption of short server and driver execution times. Since kernel's generic interrupt handler has a short strictly bounded execution time, limited hard real-time support is provided for tasks that do not rely on the standard MINIX 3 services. In addition, the RRES framework eliminates denial of service (DoS) attacks (Tsafrir et al. 2007) targeting the scheduler, because time accounting uses the TSC cycle counter independent from the system tick facility.

Work is in progress about implementing a microkernel equivalent of *bandwidth inheritance* (Lamastra et al. 2001) algorithm (as it has been done in the context of Linux, Faggioli et al. 2008a) so that the drivers and servers working on behalf of a real-time task can use its RRES parameters during the servicing time. This gives two important benefits, namely, correct time accounting and a very simple resource-access protocol, *priority inheritance*, in order to prevent priority-inversion phenomena. In addition, we intend to analyze the possibility of reserving other resources types, such as file system and network access, through the RRES framework. Success in this area would result in a completely compartmentalized and fully protected resource environment, enabling full hard real-time support.

References

Abeni L, Buttazzo G (1998) Integrating multimedia applications in hard real-time systems. In: Proc IEEE real-time systems symposium, Madrid, Spain

Abeni L, Lipari G (2002) Implementing resource reservations in Linux. In: Real-time Linux Workshop

Abeni L, Cucinotta T, Lipari G, Marzario L, Palopoli L (2005) Qos management through adaptive reservations. Real-Time Syst J 29(2–3):131–155

Baker TP (1990) A stack-based allocation policy for realtime processes. In: Proc IEEE real time systems symposium

Caccamo M, Buttazzo G, Sha L (2000) Capacity sharing for overrun control. In: Proc 21st IEEE real-time systems symposium, pp 295–304

Faggioli D, Lipari G, Cucinotta T (2008a) An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the Linux kernel. In: Proceedings of the fourth international workshop on operating systems platforms for embedded real-time applications, pp 1–10, July 2008

Faggioli D, Mancina A, Checconi F, Lipari G (2008b) Design and implementation of a posix compliant sporadic server for the Linux kernel. In: 10th real-time Linux workshop, pp 65–80, Oct 2008

Lipari G, Baruah S (2000) Greedy reclamation of unused bandwidth in constant bandwidth servers. In: Proc 12th Euromicro conf on real-time systems

Hamann C-J, Reuther L, Wolter J, Härtig H (2006) Quality-assuring scheduling. Technical report, TU Dresden

Hamann C-J, Roitzsch M, Reuther L, Wolter J, Härtig H (2007) Probabilistic admission control to govern real-time systems under overload. In: Proc 19th Euromicro conf on real-time systems

Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS (2006) Construction of a highly dependable operating system. In: Proc 6th European dependable computing conf

Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS (2007) Failure resilience for Device Drivers. In: Proc 37th int'l conf on dependable systems and networks

Kaneko H, Stankovic JA, Sen S, Ramamritham K (1996) Integrated scheduling of multimedia and hard real-time tasks. In: Proc IEEE real-time systems symposium

Lamastra G, Lipari G, Abeni L (2001) A bandwidth inheritance algorithm for real-time task synchronization in open systems. In: Proc 22nd IEEE real-time systems symposium

Liedtke J (1996) Toward real microkernels. CACM 39(9):70–77

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J Assoc Comput Mach 20(1):46–61
Marzario L, Lipari G, Balbastre P, Crespo A (2004) Iris: A new reclaiming algorithm for server-based real-time systems. In: Proc IEEE real-time and embedded techn and app symp
Mercer CW, Savage S, Tokuda H (1993) Processor capacity reserves: an abstraction for managing processor usage. In: Proc 4th workshop on workstation operating systems
Mercer CW, Rajkumar R, Zelenka J (1994) Temporal protection in real-time operating systems. In: Proc 11th IEEE workshop on real-time operating systems and software
Meurs R (2006) Building performance measurement tools for the MINIX 3 OS. Master's thesis, 2006. Vrije Universiteit, Amsterdam
Pessolani PA (2006) MINIX4RT: A real-time operating system based on MINIX. Master's thesis. Universidad Nacional de La Plata
Rajkumar R, Juvva K, Molano A, Oikawa S (1998) Resource kernels: a resource-centric approach to real-time and multimedia systems. In: Proc conf on multimedia comp and netw
Rogina P, Wainer G (2001) Extending rt-minix with fault tolerance capabilities. In: Proc Latin-American conf on informatics
Rogina P, Wainer G (1999) New real-time extensions to the minix operating system. In: Proc of 5th int conf on information systems analysis and synthesis
Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans Comput 39(9):1175–1185
Spuri M, Buttazzo GC (1994) Efficient aperiodic service under the earliest deadline scheduling. In: Proc IEEE real-time systems symposium
Tokuda H, Nakajima T, Rao P (1990) Real-time mach: towards predictable real-time systems. In: Proc USENIX mach workshop
Torrey LA, Coleman J, Miller BP (2007) A comparison of interactivity in the Linux 2.6 scheduler and an mlfq scheduler. Softw Pract Exp 37(4):347–364
Tsafrir D, Etsion Y, Feitelson DG (2007) Secretly monopolizing the CPU without superuser privileges. In: USENIX security

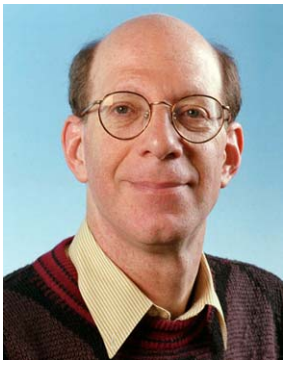
Antonio Mancina took his Ph.D. in 2009 working at the Real-Time System Laboratory, Scuola Superiore Sant'Anna. In his thesis, entitled "Operating Systems and Resource Reservations", he focused on the problem of real-time scheduling algorithms applied to multimedia domains from both a theoretical and practical point of views especially in the context of general purpose operating systems.

Dario Faggioli received his master degree (cum laude) in computer science engineering from the University of Pisa in 2007. He is now Ph.D. student in embedded systems at the Scuola Superiore Sant'Anna. His ongoing research is about scheduling and synchronization mechanisms for real-time and embedded systems, and on integrating real-time mechanisms inside general purpose operating systems.

Giuseppe Lipari graduated in Computer Engineering at the University of Pisa in 1996, and received the Ph.D. degree in Computer Engineering from Scuola Superiore Sant'Anna in 2000. He is Associate Professor of Operating Systems with Scuola Superiore Sant'Anna. His main research activities are in real-time scheduling theory and its application to real-time operating systems, soft real-time systems for multimedia applications and component-based real-time systems. He has been member of the program committes of many conferences in the field. He is currently Associate Editor of IEEE Transactions on Computers.

Jorrit N. Herder received an M.Sc. degree in computer science (cum laude) from Vrije Universiteit in Amsterdam in 2005 and is currently wrapping up his Ph.D. project there. His research focuses on operating system dependability. He is closely involved in the design and implementation of MINIX 3, and has authored various papers on this topic.

Ben Gras holds an M.Sc. degree in computer science from Vrije Universiteit in Amsterdam and has previously worked as system administrator and programmer. He is now employed by Vrije Universiteit in the Department of Computer Systems as a scientific programmer working on the MINIX 3 project.



Andrew S. Tanenbaum is a professor of computer science at Vrije Universiteit in Amsterdam. He has written 17 books and 140 papers and is a Fellow of both the ACM and the IEEE. He is also a member of the Royal Netherlands Academy of Arts and Sciences. He firmly believes that we need to radically change the structure of operating systems to make them more reliable and secure and that MINIX 3 is a small step in this direction.