Enhancing a Dependable Multiserver Operating System with Temporal Protection via Resource Reservations

Antonio Mancina, Giuseppe Lipari
Scuola Superiore Sant'Anna, Pisa, Italy
{a.mancina,lipari}@sssup.it

Jorrit N. Herder, Ben Gras, Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam, The Netherlands
{jnherder,beng,ast}@cs.vu.nl

Abstract

MINIX 3 is a microkernel-based, multiserver operating system for uniprocessors that is designed to be highly dependable. Servers are restricted according to the principle of least privilege. For example, access to resources such as system memory and device I/O is fully protected. Although MINIX 3 is a potential candidate for embedded platforms, it currently cannot safeguard processes with stringent timing requirements, such as real-time applications.

In this paper, we present the design and the implementation of a user-space resource-reservation framework (RRES) in order to augment MINIX 3's dependability infrastructure with temporal protection. In particular, we implemented the Constant Bandwidth Server (CBS), either in Soft or in Hard Reservation (CBS-HR) mode and the Idle-time Reclaiming Improved Server (IRIS) resource reservation algorithms. Important, practical applications of temporal protection include real-time computing as well as prevention of certain denial of service (DoS) attacks that monopolize the CPU. Experiments on a prototype implementation showed improved dependability in the temporal domain.

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 [34] 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 timesharing 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 [7, 18, 27]. In contrast, the work in this paper provides real-time support on MINIX 3, a novel microkernel-based, multiserver operating system. This focus allowed for a highly modular design and implementation of a reservation framework with only modest modifications to the base system.

1.1 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 contrast, a bug in a kernel module in a classic monolithic operating system could easily hang or crash the entire system due to the lack of isolation. 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 [16, 17].

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 [30] or Stack-Based Resource Protocol [10], in order to avoid priority inversion phenomena.

1.2 Resource Reservations

In order to provide temporal protection on MINIX 3, we have made several modifications to the scheduler and designed and implemented a *resource reservation* (RRES) framework. Resource reservations are a class of real-time algorithms that grant Q resource units every period P [27]. 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 [8, 9].

To the best of our knowledge, we are the first to implement resource reservations in MINIX 3. In particular, we have provided a complete implementation of CBS [7], CBS-HR and IRIS [22], which are among the first and most effective ones. The new resource reservation framework improves MINIX 3 in three important ways:

1. RRES brings soft real-time support, so that benefits can be gained in many application domains, like the ones mentioned above. Correct accounting is achieved under the assumption of MINIX 3's low-latency response times discussed above. Moreover, 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 [34]. Reliable accounting is realized by using the TSC cycle counter independent from the programmable interrupt timer (PIT), as detailed in Sec. 4.4.

1.3 Paper Outline

The remainder of the paper is organized as follows. Sec. 2 briefly surveys related work. Sec. 3 introduce the CBS, CBS-HR and IRIS resource reservation algorithms. Sec. 4 describe how we implemented the resource reservation (RRES) framework. Secs. 5 and 6 present a case study and the results of performance measurements on the introduced latency. Finally, Sec. 7 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 [6]. 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 ar-

chitecture with a monolithic kernel running on top of a microkernel layer. The most important projects are RTAI [2], RT-Linux [3] and Xenomai [5]. 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) [24, 23, 32]. RT-Mach enforced the concept of resource reservation using a fixed-priority schemes like RM [21], or, at most, a dynamic-priority scheme based on old algorithms like TBS [31], which cannot achieve full CPU utilization. In contrast, MINIX 3 implements the newer CBS, CBS-HR and IRIS algorithms. Furthermore, RT-Mach seems to have fixed the scheduling policy in the kernel, whereas we promote a minimally invasive, modular design.

Real-time support in L4 [20] is based on the statistical approaches Quality-Assuring Scheduling (QAS) [14] and Quality-Rate-Monotonic Scheduling (QRMS) [15]. 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 [26] aims to mimic the low-latency RT-Linux architecture in MINIX 2. Second, RT-Minix [29, 28] 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. Moreover, 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) [21] 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 t, the task with the earliest absolute deadline is executed." Despite a higher computational complexity than *Rate Monotonic* (RM) [21], which is the industrial standard for the fixed priorities field, EDF, in contrast to RM, can always achieve full CPU utilization without any deadline misses, which is an important goal in our work.

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, 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*.

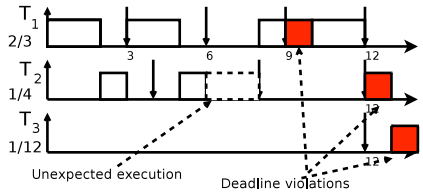

Figure 1: Produced schedule with misbehaving task T_2. The tasks' computation times and deadlines are shown at the left.

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. Fig. 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.

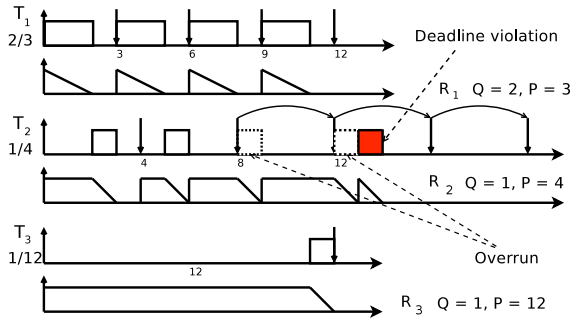

Figure 2: Same schedule with temporal protection. The VRES' timeline is shown below the task's timeline.

3.2 Resource Reservation Algorithms

We now briefly introduce the resource reservation algorithms that we implemented in MINIX 3: CBS [7], CBS-HR and IRIS [22], which are among the most widely recognized resource reservation algorithms. Other algorithms have also been proposed, e.g. [11, 12, 13], but we chose CBS, CBS-HR and IRIS because their simplicity and low overhead allowed for a clean and minimally invasive implementation.

3.2.1 CBS

The *Constant Bandwidth Server* (CBS) [7] 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 reclamation scheme provides temporal protection against overruning 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* [22]. 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.

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 [22]. 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 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 $d = t + P$.

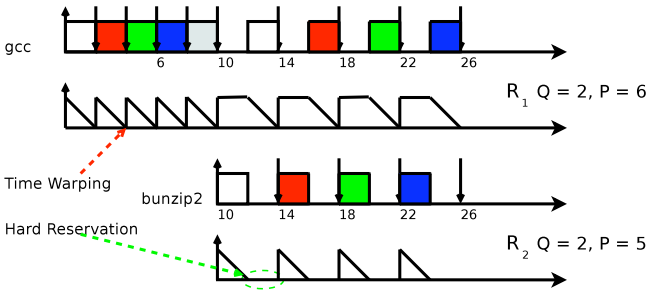

Figure 3: IRIS solves the deadline aging problem by having VRESes wait until the next deadline before replenishing their budget. The replenishment can be instantaneous due to time warping.

These extensions 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 DESIGN AND IMPLEMENTATION

This section describes how we implemented a resource reservation (RRES) framework in MINIX 3 with support for the CBS, CBS-HR and IRIS resource reservation algorithms. Three important design guidelines for the implementation of the RRES 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 RRES 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.

4.1 High-level Design Overview

Based on the above design criteria we decided to introduce a separate user-space scheduler, 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 [17]. The basic idea then is to let the kernel execute user-space scheduling requests for real-time applications 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 a separate RRES scheduler next to the MINIX 3 scheduler.

As an aside, we provided three different implementations of the RRES manager, one for each resource reservation algorithm supported: CBS, CBS-HR, and IRIS. The algorithm used is statically chosen with a compiler flag. It is currently not possible to let different VRESes serve their task using different algorithms, since the theoretical analysis to make this possible is still in progress. The reason for supporting CBS and CBS-HR next to IRIS is a matter of usability. With IRIS' time warping rule, all CPU cycles would be used for real-time task and non-real-time applications would not get a chance to execute. In such a scenario, every application should be enclosed in a reservation, resulting in a system that is harder to analyze and maintain.

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. Fig. 4 gives a high-level overview of the RRES framework.

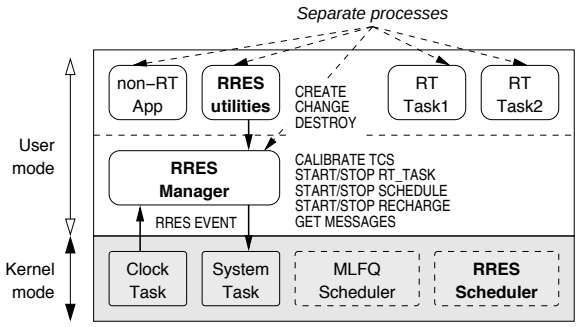

Figure 4: High-level architecture over the resource reservation framework. Messages exchanged between the RRES helper utilities, RRES manager and kernel are shown.

4.2 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.

4.2.1 RRES Data Structures

The main RRES data structure has three 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, finally, contains the VRESes that blocked during their execution, for example, because they have to wait for some event to happen.

4.2.2 RRES Interactions

As shown in Fig. 4, 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. 5. 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.

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

Figure 5: Messages exchanged within the RRES framework.

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.
- 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. 5, 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 Sec. 6.

4.3 Kernel and Scheduler Modifications

Scheduling in the standard MINIX 3 kernel is done on best-effort basis using a multilevel-feedback-queue scheduler (MLFQ) [33]. 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_SERVE request to the kernel. Second, the scheduler data structure was extended with two new scheduling queues at the highest priorities, as shown in Fig. 6.

- **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.
- **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,

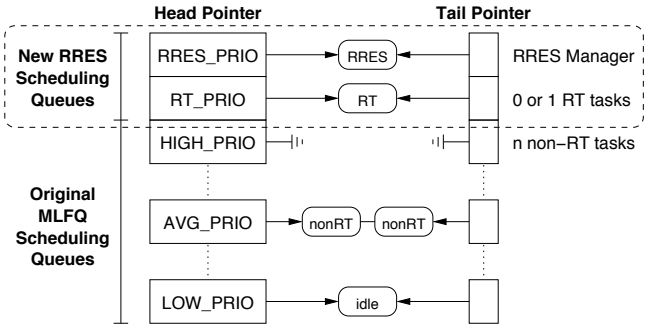

Figure 6: RRES-enhanced MINIX 3 scheduling queue data structure. Two new queues at the two highest priority levels were added for the RRES manager and the current real-time task.

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 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.

4.4 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 [25] and having two sources of timer interrupts would have complicated the kernel's code.

4.4.1 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.

4.4.2 Eliminating CPU Monopolization

An important benefit of our design is that denial of service (DoS) attacks that monopolize the CPU [34] 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 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. 7. Initially, the administrator starts the tasks using the *rres_create* utility. The command entered is

$ rres_create <budget> <period> <binary>

where the request parameters are

<budget>: CPU budget given in each period (Q) in μs;
<period>: the VRES granularity (P) in μs;
<binary>: the application to be managed by RRES.

This request has to be made for both task T_1 and T_2 with parameter $Q_1 = 3000\ \mu s$, $P_1 = 9000\ \mu s$ and $Q_2 = 2000\ \mu s$, $P_2 = 3000\ \mu s$. 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 executes the binary. RRES first checks if the user is authorized and then performs an admission test. Since the CPU utilization does not exceed 100%, 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.

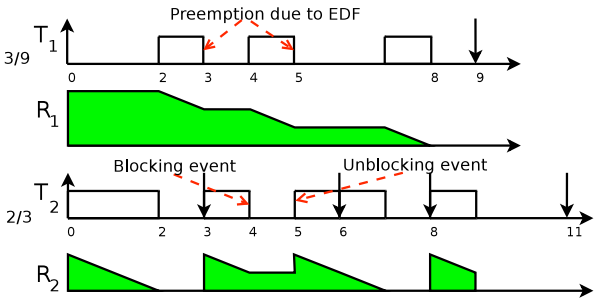

Figure 7: Schedule of the case study in milliseconds.

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. 7. As discussed in Sec. 4.4, 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. 7. In the following section we will see how these interactions impose a very limited timing overhead on the system.

6 EXPERIMENTAL EVALUATION

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

6.1 Timing Measurements

As explained in Sec. 4.4, 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 Fujitsu-Siemens desktop PC with a 2.8 GHz AMD Athlon64 CPU and 1 GB 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 1.5 μs.

Second, we measured the latency introduced by the RRES framework. These tests were done in the context of the case study discussed in Sec. 5. We ran several tests and computed the mean result rounded to microsecond precision.

- Time between receiving a *rres_create* command in the RRES framework and the moment that the kernel schedules the new real-time task: $192\mu s$.
- Time between budget exhaustion in the kernel, causing an RRES_EVENT notification processed by the user-space RRES framework, and the moment that the kernel puts the VRES in the recharging state: $43\mu s$.
- Time between detecting a budget-recharged event in the kernel, notifying RRES, and the moment that the kernel reschedules the corresponding task: $41\mu s$.

These results clearly show a very limited overhead imposed by the RRES framework on the system in order to enforce the CBS, CBS-HR and IRIS algorithms. 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. 6.

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 $250\mu s$. 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.

6.2 Impact on Kernel and User-Space Code

With help of the Source Code Line Counter [4] tool available on the Internet we collected data on the total engineering effort required. The number of executable lines of code for both the standard and modified version of the MINIX 3 kernel are shown in Fig. 8. Similar statistics for the new user-space RRES manager are shown in Fig. 9

6.3 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.

Fig. 11 represents a piece of the scheduling of the task set in Fig. 10 that is scheduled according to CBS-HR (CBS with hard reservations); IRIS' time warping is not used. The tasks used are an infinite CPU-bound program (*cpuload*) that 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).

Numerous other simulations have been run to verify the behaviour of our implementation in few real cases, but we refrained from including them here due to space limitations.

7 CONCLUSIONS AND FUTURE WORK

MINIX 3 is a dependable multiserver operating system for uniprocessor systems. Its modular design makes it a likely candidate for embedded systems, but MINIX 3 currently lacks real-time support. Therefore, we have enhanced MINIX 3 with *temporal protection* via *resource reservations*. To the best of our knowledge, this had not been done before. Long latencies and slow response times caused by the message passing mechanism were a potential bottleneck, but measurements on a prototype implementation have shown that this effect is very limited and can be mitigated by carefully designing the framework interactions.

Our resource reservation framework (RRES) implements the CBS, CBS-HR and IRIS resource reservation algorithms and provides temporal protection in order to prevent ordinary users from monopolizing the CPU. Our design enables running soft real-time applications on MINIX 3. The current status is that correct time accounting happens in presence of

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**

Figure 8: Lines of executable code (LoC) for the standard MINIX 3 kernel and the modified version with the RRES framework.

Header Files	LoC
glo.h	42
inc.h	29
proto.h	51
rres.h	106
Header Total	**156**

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

Figure 9: Lines of executable code (LoC) for the RRES server.

Task	Type	Budget (ms)	Period (ms)
cpuload	CPU-bound	100	400
cpuload	CPU-bound	200	2000
interactive	I/O-bound	10000	20000

Figure 10: Task set and reservation parameters used for tracer simulation. The execution is shown in Fig. 11.

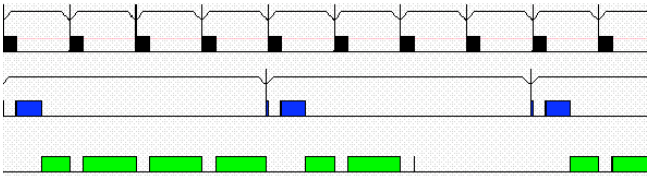

Figure 11: Actual schedule executed for the task set of Fig. 10 produced by the RRES tracer based on RRES server logs.

nonblocking 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 [34] targeting the scheduler, because time accounting uses the TSC cycle counter independent from the system tick facility.

Work in the context of FRESCOR [1] is in progress to implement a microkernel equivalent of *bandwidth inheritance* [19] algorithm 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

[1] FRESCOR - Framework for Real-time Embedded Systems based on COntRacts. `http://www.frescor.org`.
[2] RTAI home page. `https://www.rtai.org/`.
[3] RTLinux home page. `http://www.rtlinux.org`.
[4] Sclc.pl - the Source Code Line Counter. Available online.
[5] XENOMAI home page. `http://www.xenomai.org`.
[6] Ingo Molnar's RT Tree. Available online.
[7] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.
[8] L. Abeni and G. Lipari. Implementing resource reservations in linux. In *Real-Time Linux Workshop*, 2002.
[9] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Real-Time Systems Journal*, 29(2-3), March 2005.
[10] T. P. Baker. A stack-based allocation policy for realtime processes. In *Proc. IEEE Real Time Systems Symposium*, 1990.
[11] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. 21st IEEE Real-Time Systems Symposium*, pages 295–304, 2000.
[12] M. Caccamo, G. C. Buttazzo, and D. C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, Feb. 2005.
[13] G.Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant bandwidth servers. In *Proc. 12th Euromicro Conf. on Real-Time Systems*, 2000.
[14] C.-J. Hamann, L. Reuther, J. Wolter, and H.Härtig. Quality-Assuring Scheduling. Technical report, TU Dresden, 2006.
[15] C.-J. Hamann, M. Roitzsch, L. Reuther, J. Wolter, and H. Härtig. Probabilistic admission control to govern real-time systems under overload. In *Proc. 19th Euromicro Conf. on Real-Time Sys.*, 2007.
[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. 6th European Dependable Computing Conf.*, 2006.
[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for Device Drivers. In *Proc. 37th Int'l Conf. on Dependable Systems and Networks*, 2007.
[18] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *Proc. IEEE Real-Time Systems Symposium*, 1996.
[19] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proc. 22nd IEEE Real-Time Systems Symposium*, 2001.
[20] J. Liedtke. Toward real microkernels. *CACM*, 39(9):70–77, 1996.
[21] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.
[22] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proc. IEEE Real-Time and Embedded Techn. and App. Symp.*, 2004.
[23] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: An Abstraction for Managing Processor Usage. In *Proc. 4th Workshop on Workstation Operating Systems*, 1993.
[24] C. W. Mercer, R. Rajkumar, and J. Zelenka. Temporal Protection in Real-Time Operating Systems. In *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software*, 1994.
[25] R. Meurs. Building Performance Measurement Tools for the MINIX 3 OS. Master's thesis. Vrije Universiteit, Amsterdam, 2006.
[26] P. A. Pessolani. MINIX4RT: A Real-Time Operating System Based on MINIX. Master's thesis. Universidad Nacional de La Plata, 2006.
[27] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proc. Conf. on Multimedia Comp. and Netw.*, 1998.
[28] P. Rogina and G. Wainer. Extending rt-minix with fault tolerance capabilities. In *Proc. Latin-American Conf. on Informatics*, 2001.
[29] P. Rogina and G. Wainer. New real-time extensions to the minix operating system. In *Proc. of 5th Int. Conf. on Information Systems Analysis and Synthesis*, 1999.
[30] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9): 1175–1185, Sept. 1990.
[31] M. Spuri and G. C. Buttazzo. Efficient aperiodic service under the earliest deadline scheduling. In *Proc. IEEE Real-Time Systems Symposium*, 1994.
[32] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards Predictable Real-Time Systems. In *Proc. USENIX Mach Workshop*, 1990.
[33] L. A. Torrey, J. Coleman, and B. P. Miller. A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler. *Softw. Pract. Exper.*, 37(4):347–364, 2007.
[34] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Secretly Monopolizing the CPU Without Superuser Privileges. In *USENIX Security*, 2007.