ABSTRACT

General-Purpose Operating Systems (GPOSes) are being used more and more extensively to support interactive, real-time and distributed applications, such as found in the multimedia domain. In fact, the wide availability of supported multimedia devices and protocols, together with the wide availability of libraries and tools for handling multimedia contents, make them an almost ideal platform for the development of this kind of complex applications. However, contrarily to Real-Time Operating Systems, General-Purpose ones used to lack some important functionality that are needed for providing proper scheduling guarantees to application processes. Recently, the increasing use of GPOSes for multimedia applications is gradually pushing OS developers towards enriching the kernel of a GPOS so as to provide more and more real-time functionality, thus enhance the performance and responsiveness of hosted time-sensitive applications. In this chapter, an overview is performed over the efforts done in the direction of enriching GPOSes with real-time capabilities, with a particular focus on the Linux OS. Due to its open-source nature and wide diffusion and availability, Linux is one of the most widely used OSes for such experimentations.

INTRODUCTION

New generation interactive distributed applications have significant demands on processing, storage and networking capabilities, as well as stringent timing requirements, such as various multimedia, virtual collaboration and e-learning applications. These are, among others, types of applications that have been undergoing in the last few years a very steep demand on the side of end-to-end interactivity level, response time, and throughput. This is also due to the growing availability of affordable broadband Internet connections. This allowed people to realize that most of the activities they used to carry on through off-line workflows (especially when relatively high data volumes were involved) now could be done within on-line collaborative systems, in a much more interactive way.

For a long time, these applications have been made available to end-users in the form of costly professional solutions running on dedicated hardware.
the growth in computation capabilities of computing systems has led to a growing interest in providing such applications in the form of applications that may be run on General-Purpose Operating Systems (GPOSes). This allows for sharing a set of physical resources among different applications or application instances, thus scaling down the costs needed to deploy and run them.

Unfortunately, GPOSes are not designed to provide the run-time support that is necessary for meeting timing requirements of individual applications. In fact, in the traditional hard real-time domain, a Real-Time Operating System (RTOS) is used for such purpose. A RTOS usually provides those features that allow for a well-known, predictable and analysable timing behaviour of hosted applications: all kernel segments are characterized through well-known worst-case durations, scheduling latencies and interrupt latencies which may be controlled through the appropriate tuning of the interrupt and process schedulers. A set of real-time scheduling policies is available for the system designer and programmers. Time may be measured (and timers may fire) with a high precision (typically sub-millisecond) while there are tools available for Worst-Case Execution Time (WCET) estimation and (off-line) analysis, as well as for schedulability analysis. Unfortunately, such OSes are designed for embedded control applications, thus they imply serious constraints on the supported hardware and available high-level software infrastructures.

Therefore, in recent years, various efforts have been done towards the integration of real-time technologies within GPOSes, particularly within Linux, for the availability of its kernel as open-source and its worldwide diffusion.

Various drawbacks make the Linux kernel, as well as most of the GPOSes, particularly unsuitable for running real-time applications: the monolithic structure of the kernel and the wide variety of drivers that may be loaded within; the impossibility to keep under control all the non-preemptable sections possibly added by such drivers; the general structure of the interrupt management core framework that privileges portability with respect to latencies; and others.

As a result, the latency experienced by time-sensitive activities can be as large as tens or hundreds of milliseconds. This makes GPOSes non-suitable for hard real-time applications with tight timing constraints. Though, soft real-time applications may run quite well within such an environment, especially when the original kernel is modified so as to integrate the necessary real-time capabilities.

Objective of this chapter is to provide an overview of the approaches appeared in the research literature aiming to integrate real-time capabilities within General-Purpose Operating Systems, with a particular focus on Linux.

**BACKGROUND**

Although not being a real-time system, the 2.6 Linux kernel includes a set of features making it particularly suitable for soft real-time applications. First, it is a fully preemptable kernel, like most of the existing real-time operating systems, and a lot of effort has been spent on reducing the length of non-preemptable sections (the major source of kernel latencies). It is noteworthy that the 2.6
kernel series introduced a new scheduler with a bounded execution time, resulting in a highly decreased scheduling latency. Also, in the latest kernel series, a modular framework has been introduced that allows for an easier integration of other scheduling policies. Second, although being a general-purpose time-sharing kernel, it includes such real-time extensions to the POSIX standard (POSIX 2004) as: the SCHED_FIFO and SCHED_RR process scheduling policies, the priority inheritance protocol for avoiding the well-known priority inversion problem, the real-time POSIX extensions related to signals and timers. Also, there seems to be a growing interest in implementing the SCHED_SPORADIC real-time scheduling class as well. These features may result useful for developing real-time systems. Third, the recently introduced support in the kernel mainstream of the support for high-resolution timers is of paramount importance for the realization of high-precision customized scheduling mechanisms, and for the general performance of soft real-time applications.

Furthermore, recent patches proposed by the group of Ingo Molnar to the interrupt-management core framework, aimed at encapsulating device drivers within kernel threads, are particularly relevant as such approaches would highly increase predictability of the kernel behaviour.

Concerning the process scheduler, recently it has undergone two major changes: from the 2.4 series to the 2.6 series, the scheduler was reengineered so as to cut its computational complexity, from linear in the number of tasks to the so advertised $O(1)$ scheduler. From the 2.6.23, a new modular framework has been introduced by Ingo Molnar, along with a complete rewrite of the default scheduling policy (the SCHED_OTHER POSIX class), now called Completely Fair Scheduler (CFS). It basically aims to exhibit fairness properties among running processes, resembling a Generalized Proportional Fair (GPF) system (in its use of a timestamping mechanism), but unfortunately it does not correspond to any algorithm with well-known properties in scheduling literature. Tasks are characterized by a configurable weight, and a run-time quantity called vruntime. Each time a task is executed for a time $t$, its vruntime value is incremented by $t \times \text{task_weight/total_weight}$, where total_weight is the total weight of the currently running tasks. When a task is reactivated, it is assigned a vruntime that varies according to the configuration of the system and to various heuristics, which tend to give a small vruntime to interactive processes. Finally, tasks are scheduled in increasing vruntime order. As of now, no formal guarantees have been derived for the service provided to individual tasks scheduled using CFS.

Also, the current Linux kernel has a partial support for hierarchical scheduling, a feature typical of real-time OSes. Both real-time and SCHED_OTHER tasks can be organized in groups and for each group it is possible to limit the CPU time consumed by its real-time tasks, as well as configure the CPU share used by its SCHED_OTHER tasks.

Due to the internal architecture of the scheduler (that is not fully hierarchical in a traditional sense) a combination of the two effects cannot be obtained. In other words, it is not possible to limit the CPU time of a whole group (real-time + SCHED_OTHER tasks) and it is not possible to limit the CPU share assigned to a whole group. The intention of enhancing the real-time limiting (throttling)
mechanism has been expressed on public discussions, but no code has been written in that direction, as of now.

Concluding, the mainline Linux kernel (a.k.a., Vanilla), despite recent improvements in the scope of real-time support, currently lacks a CPU scheduler able to fulfil the timing requirements of increasingly needed distributed real-time applications.

In what follows, related work in the area of real-time support for GPOSes is presented. Various modifications of GPOSes have appeared in the literature for supporting real-time scheduling policies at the kernel-level, for various types of resources. First, related work in the area of real-time scheduling for the CPU is surveyed. Then, approaches in which scheduling of multiple heterogeneous, possibly distributed, resources is integrated into a common framework, are described. Then, the most relevant and recent approaches, among the identified ones, are described in somewhat more detail, for the sake of completeness.

Approaches, Implementations and Comparisons

Real-time CPU scheduling in GPOSes
Various GPOS kernels exist that are compliant with the POSIX real-time extensions (POSIX 2004). However, the main drawback of such extensions is that there are various optional parts, and most of the implementations limit themselves to Fixed-Priority scheduling, sometimes with the addition of high-resolution timers and the Priority Inheritance protocol for avoiding Priority Inversion (Sha 1990). Also, one key feature which is usually not implemented is the temporal isolation property (Buttazzo 2005), such as provided by the Sporadic Server (POSIX 2004) scheduling policy. Without such a mechanism, a higher priority task runs undisturbed until it blocks, independently of the computation time that may have been considered at system analysis/design time. This results in the potential disruption of the guarantees offered to lower priority tasks. Therefore, such approaches are suitable for the traditional hard real-time settings where everything running into the system has been thoroughly checked, if not formally proved.

For real-time scheduling of the CPU, hard real-time modifications to the Linux kernel have been proposed, like RT-Linux (http://www.rtlinuxfree.com), proposed by Yodaiken et al. (Yodaiken 1997) and RTAI (http://www.rtai.org), proposed by Dozio et al. (2003). In these approaches, a real-time micro-kernel layer is added between the real hardware and the Linux OS, which runs as the background/idle activity whenever there are no hard real-time tasks active in the system. This allows for respecting the very tight timing constraints (microsecond-level) typical of industrial automation and robotic applications. Main drawbacks of such approaches are due to the limitations that (hard) real-time applications exploiting real-time scheduling are subject to, i.e., they are typically required to be written as kernel modules. Therefore, these hardly constitute solutions suitable for the large class of multimedia and interactive applications, which would greatly benefit from real-time scheduling policies. However, a detailed description of these approaches follows below, due to their historical importance.
In order to overcome these limitations, other approaches targeted explicitly soft real-time applications, by adding a real-time scheduling policy as an extension to a GPOS kernel itself, typically comprising a temporal isolation mechanism. Such an approach, which is also exploited by the work in this paper, allows for the coexistence of soft real-time and best-effort applications, all within a GPOS kernel with potentially long non-preemptive sections, what leads to the impossibility to provide hard real-time guarantees. However, for soft real-time applications like multimedia ones, that approaches allowed for great enhancements achieved on the side of the predictability in the temporal behaviour exhibited by applications, which resulted in significant improvements in the QoS experienced by users. An overview of these approaches has been carried out by Gopalan (2001). Just to mention a few, remarkable works are the ones for adding resource reservations (Mercer 1993) to Microsoft Windows NT by Jones (1999) (Rialto/NT), and the ones by Rajkumar et al. in the Linux/RK project (Rajkumar 1998). The latter constitutes a modification to the Linux kernel, largely inspired by prior work of the same authors on RT-Mach. Such code has also been designed so as to be portable across multiple GPOS kernels (Oikawa 1999), but it has been implemented on Linux only, to the best of authors' knowledge. An effort on portability of a real-time scheduler across various Operating Systems (from Microsoft, UNIX and Linux families), is constituted by the DSRT scheduler (http://cairo.cs.uiuc.edu/software/DSRT-2/dsrt-2.html) by Yuan et al. (Yuan 2003). However, similarly to the GRACE (Vardhan 2005) series of architectures, the focus seems to be limited exclusively on CPU scheduling.

Also, soft real-time schedulers for Linux have been investigated and implemented in the context of various European Projects like the CBS implementation on Linux developed during the OCERA Project, and its subsequent evolution, the AQuoSA (Palopoli 2009) scheduler for Linux, developed by Tommaso Cucinotta during the FRESCOR Project. More details about OCERA, FRESCOR and AQuoSA follow below dedicated sections.

More recently, the IRMOS Project (Interactive Real-Time Multimedia Applications on Service-Oriented Infrastructures, European Project n. FP7-214777, http://www.irmosproject.eu), is also investigating on the use of real-time scheduling on high-performance Linux machines, with a strong focus on virtualized distributed real-time applications. In the context of IRMOS, the most recent real-time extensions to the Linux scheduler have been proposed: Faggioli et al. proposed a POSIX compliant implementation (Faggioli 2008) of the FP-based Sporadic Server algorithm (and variations) for enhancing the current primitive throttling mechanism of the Linux kernel with improved possibility of predictability, temporal isolation and analysability; Checconi et al. designed a novel hierarchical hybrid scheduling framework (Checconi 2009), based on a combination of partitioned EDF and global FP, designed so as to fit as much as possible (and impact as less as possible) in the current real-time scheduling class code base. A similar work is being done in the context of the ACTORS Project (Adaptivity and Control of Resources in Embedded Systems, European Project n.216586, http://www.actors-project.eu), but with a focus on single-node embedded multi-core systems, without any distributed features. Here, both
partitioned EDF (Faggioli 2009) and Global EDF (Lelli 2010) are being investigated.

**Integrated scheduling of multiple resources**

Prior works exist that integrate real-time scheduling of heterogeneous resources and an architecture for their management for real-time applications, like the one by Stankovic et al. (Zhang 2002), or Hola QoS by Valls et al. (2002). The latter is an architecture specifically tied to the needs of consumer electronics embedded multimedia systems, providing flexible resource management and adaptivity. The Eclipse/BSD (Blanquer 1999) Project integrates real-time scheduling of CPU, network and disk access, and exposes to applications a file-system based user-space interface. However, the project does not deal with distributed real-time applications. More recently, Gopalan et al. (2007) proposed MURALS, a distributed real-time architecture built upon TimeSys Linux (http://www.timesys.com), supporting real-time applications with end-to-end constraints making use of distributed heterogeneous resources, such as disks, CPUs and network links. The architecture embeds a global admission control scheme that takes into account the entire dependency graph of the application.

The above mentioned Nahrstedt research group also worked on QualMan (Nahrstedt 1998), a distributed real-time resource allocation architecture supporting also network, disk and memory allocation, with prototype implementation on the Solaris OS. However, the same authors highlight that the modularity degree of their architecture is somewhat limited and that it would benefit from a CORBA-oriented design.

In fact, the CORBA specification has been extended to address reusability in the CORBA Component Model (CCM), which also considers QoS aspects. For example, this has been implemented in the Component-Integrated ACE ORB (Schmidt 2003). TAO (Schmidt 1997) constitutes a C++ implementation of the Real-Time CORBA specification (Wolfe 1997), which exposes fundamental functionality of distributed real-time applications via the CORBA paradigm.

Also, TAO has been integrated with QuO (Krishnamurty 2001), a framework that exploits the capabilities of CORBA to reduce the impact of QoS management on the application code. The result (Schantz 2003) is a middleware for adaptive QoS control using real-time scheduling facilities at the computation and network levels. Recently, such an architecture has been used by Shankaran et al. within their HiDRA (Shankaran 2006) architecture for hierarchical management of multiple resources in distributed real-time systems. However, these works are focused on issues related to the monitoring of the run-time application behaviour, and the dynamic adaptation of resource allocations and/or application behaviour to their continuously changing needs, so they consider only marginally issues strictly related to the low-level mechanisms needed for guaranteeing the respect of timing constraints needed by real-time applications. Eide et al. (Eide 2004) also presented a CORBA-based middle-ware for the management of the CPU in distributed systems, however they didn't consider other resources.

Furthermore, it is worth to mention the architecture (Cucinotta 2009) developed in the context of the RI-MACS project for distributed real-time applications in the
factory automation domain. The architecture relies on the capabilities of service-oriented infrastructures for providing discovery of resources and their real-time capabilities, self-configuration, fault-tolerance and scheduling parameters negotiation. The RI-MACS architecture also exploits AQuoSA (Palopoli 2009) as the low-level CPU scheduler (as this work does), but it lacks both the support for so many multiple heterogeneous resources as the present paper has (cable and wireless network, CPU, disk), and the well-defined unified API enabling applications to exploit such real-time capabilities.

In 2002, Ravindran introduced a conceptual framework (Ravindran 2008) encompassing the elements necessary in a middleware for distributed real-time applications: a system description language comprising real-time and QoS requirements specification, and run-time components for management of resources, fault detection and recovery, and detection of violations of real-time constraints and recovery actions (i.e., dynamic changes in the resources allocation). However, the work is quite abstract, and even if it refers to a prototype implementation in C on the Solaris OS, it does not address some essential details like the type of scheduling that is used at the resources level, or whether or not the implementation is anyway portable. Furthermore, the Ravindran framework highlights the need for the capability of migrating applications across nodes for tolerating hardware failure at the host level, or for enhancing the performance of applications whose workload requirements is deviating from the one predicted at deployment time, but it does not give any detail on how such a mechanism could be implemented and supported by the middleware (i.e., there is no discussion about the possible use of virtualization technologies or check-pointing). Also, the framework uses TCP/IP primitives for communications at the middleware level. Therefore, it has to re-implement such features as service registration and discovery, which are standard in a modern distributed middleware like CORBA.

More recently, Rajkumar et al. (Lakshmanan 2008) proposed Distributed Resource Kernels, an extension of Linux/RK adding support for distributed real-time applications. Linux/RK is geared towards a model of non-modifications to applications, because real-time support for legacy applications is one of its crucial features. So, applications are not necessarily aware that they are receiving service-guarantees by the OS, but rather the system is administered and configured in such a way that real-time scheduling services, for the various resources, is automatically activated each time a given application is launched in the system. Major drawbacks of Linux/RK are that, for improved efficiency, the kernel includes functionality for the distributed management and allocation of resources, by means of dedicated kernel threads which exchange messages over UDP in order to communicate to each other. Therefore, the kernel includes various components that are traditionally present at a higher level, in user-space, in other OSes, like an HTTP server, a DNS-like server, and an NTP-like server. Implementation of such services in user-space, instead, is highly beneficial and recommended for security and robustness purposes.
**RTLinuxFree and RTAI**

RTLinuxFree is the evolution of RT-Linux (see Figure 1), a modification (patch) to the Linux kernel aimed to support hard real-time applications, initially developed by Victor Yodaiken at University of New Mexico, then carried on by FSMLabs, which was acquired by WindRiver in 2007. It works as a small executive with a hard real-time scheduler that, in addition to the hard real-time tasks, executes the entire Linux OS (both the kernel and the user-space applications) as one of its low priority tasks (Setz 2007).

RTLinuxFree adds a hard real-time executive between the hardware and the Linux kernel, which takes direct control of the interrupts of the hardware, passing them to the Linux OS only when they are not relevant for the hard real-time tasks. This way, hard real-time tasks run undisturbed and without interferences from the entire Linux OS.

![Figure 1: The RT-Linux Architecture](image)

Real Time Application Interface (RTAI) is a modification of the Linux kernel made by Prof. Paolo Mantegazza from Dipartimento di Ingegneria Aerospaziale at Politecnico di Milano (DIAPM). RTAI is an open-source project that builds on the original idea of RT-Linux, but has been considerably enhanced. RTAI allows to uniformly mix hard and soft real-time by symmetrically integrating the scheduling of RTAI proper kernel tasks, Linux kernel threads and user space tasks. However, similarly to RT-Linux, in RTAI Linux tasks run in the background with respect to the hard real-time kernel. Linux only executes when there are no real-time tasks to run, and the real-time kernel is inactive. Furthermore, the Linux OS can never block interrupts or prevent itself from being preempted by the real-time kernel (Ripoll 2002).

Both approaches aim to integrate hard real-time tasks and standard Linux tasks within the same system. This allows, for example, running a set of real-time tasks
that control an industrial plant together with the high-level software infrastructure and applications that allow monitoring of the plant from remote or from complex GUI-oriented interfaces. The main drawback of these approaches is that the real-time tasks cannot really exploit the full power of the hosting GPOS, and can only access a very limited set of peripherals (e.g. serial ports). A complex multimedia streaming application, that needs at least TCP/IP networking, heavy use of the disk and access to audio and video adapters would not be suitable to run on such a system.

**OCERA and FRESCOR**

A different approach has been undertaken in the open-source Linux/OCERA variant, developed as a result of the OCERA (Open Components for Embedded Real-time Applications, European Project n.IST2001-35102, [http://www.ocera.org](http://www.ocera.org)) European project, that embeds within the Linux OS itself a reservation-based scheduling policy (Mercer 1993) typical of real-time systems. This way, standard Linux applications, making use of the full set of resources available through OS services and libraries, may at the same time exploit the benefits of predictable timing behavior and temporal isolation for those threads that are most computation intensive. Actually, OCERA provided a customized distribution of the Linux kernel suitable for both hard and soft real-time activities, that integrates: RT-Linux, a set of drivers useful in the context of industrial control (i.e., for the CAN Bus), Linux with soft real-time extensions at the scheduler level enriched with a feedback-based QoS controller useful for continuous adaptation of the scheduling parameters to the actual task requirements. A unified kernel configuration interface allows one to decide what components to compile in the OCERA kernel.

An approach similar to OCERA has also been adopted by Linux/RT, a commercial variant of Linux supported by TimeSys Inc. and based upon the original Linux Resource Kernel (Linux/RK) from Carnegie Mellon University (Rajkumar 1998). In Linux/RT the kernel has been directly modified so to provide CPU, network and disk reservations directly to user processes. This allows the provision of timing guarantees to legacy Linux applications in a transparent way. Moreover, it is possible to access a specific API to take advantage of the reservations and of quality of service management facilities.

The OCERA soft real-time scheduler inside the Linux kernel has been almost rewritten within the on-going FRESCOR (Framework for Real-Time Embedded Systems based on Contracts, European Project n. FP6/2005/IST/5-034026, [http://www.frescor.org](http://www.frescor.org)) European project, also due to the main changes undergone by the kernel from the 2.4 to the 2.6 series, resulting in the AQuoSA framework, described below. Furthermore, the FRESCOR project focused on development of components for supporting distributed and multi-resource applications, and realizing more complex QoS control strategies that also account for power-scaling issues typical of embedded applications, and that are also suitable for distributed embedded applications. In fact, as of now, in FRESCOR there are kernel components for providing resource reservation capabilities at the CPU, disk and networking level, that may be accessed through a uniform API
designed around the POSIX well-known real-time extensions API. Also, hard real-time components are being developed for hard real-time OSes, in the project.

Generally, the main objective of the FRESCOR project is to develop the enabling technology and infrastructure required to effectively use the most advanced techniques developed for real-time applications with flexible scheduling requirements, in embedded systems design methodologies and tools, providing the necessary elements to target reconfigurable processing modules and reconfigurable distributed architectures.

The approach integrates advanced flexible scheduling techniques directly into an embedded systems design methodology, covering all the levels involved in the implementation, from the OS primitives, through the middleware, up to the application level. This is achieved through a contract model that specifies which are the application requirements with respect to the flexible use of the processing resources in the system, and also what are the resources that must be guaranteed if the component is to be installed into the system, and how the system can distribute any spare capacity that it has, to achieve the highest usage of the available resources.

The main disadvantage of FRESCOR is its focus on embedded systems, so, for example, multiprocessor capabilities of the platform are only marginally addressed, and the Linux components developed at the scheduler level do not support it. Instead, in the context of the IRMOS European Project, a real-time scheduler for Linux has been developed (Checconi 2009) with full support for multi-processor and multi-core platforms.

**Adaptive Quality of Service Architecture (AQuoSA)**

The AQuoSA framework ([http://aquosa.sourceforge.net](http://aquosa.sourceforge.net)) enhances a standard GNU/Linux system with scheduling strategies based on the Resource Reservation techniques. AQuoSA has a layered architecture as depicted in Figure 2.

At the lowest level, there is a small patch (Generic Scheduler Patch, GSP) to the Linux kernel that allows dynamically loaded modules to customize the CPU scheduler behaviour, by intercepting and reacting to scheduling-related events such as: creation and destruction of tasks, blocking and unblocking of tasks on synchronization primitives, receive by tasks of the special SIGSTOP and SIGCONT signals. A Kernel Abstraction Layer (KAL) aims at abstracting the higher layers from the very low-level details of the Linux kernel (which may change from version to version of the kernel), by providing a set of C functions and macros that abstract the needed kernel functionalities. The Resource Reservation Module implements a variant of the CBS scheduling policy on top of an internal EDF scheduler. The Interface Module allows user-space applications to request reservations of the CPU, letting all of the posted requests go through the Supervisor Module. The latter implements an appropriately designed access control model (Cucinotta 2008), by means of which AQuoSA is available to non-privileged users under a security policy that may be configured by the system administrator. Most of other real-time extensions to Linux, instead, allow only privileged users to take advantage of the available real-time functionality.
A well designed user-space library allows user-space applications to take advantage of the AQuoSA functionality. In addition to the resource reservation library (QRESLIB), used by unprivileged applications to reserve the CPU, a supervisor library (QSUPLIB) is used by a system tool in order to provide to the kernel the access-control policy configured by the system administrator. Finally, a QoS Management Library (QMGRLIB) is available implementing adaptive reservations, and providing various bandwidth controllers potentially of general interest for developers, especially in the case of periodic applications.

An interesting feature of the AQuoSA architecture is that it does not replace the default Linux scheduler, but coexists with it, giving to soft real-time tasks a higher priority than any non-real-time Linux task. Furthermore, the AQuoSA architecture follows a non-intrusive approach by keeping at the bare minimum (the GSP patch) the modifications needed to the Linux kernel. Unfortunately, in the current version, AQuoSA only supports single-processor systems.

The IRMOS Real-Time Scheduler

Here we recall the basic characteristics of the real-time scheduler developed in the context of the IRMOS project (http://www.irmosproject.eu) reminding to (Checconi 2009) for a more complete description.

The IRMOS real-time scheduler allows to reserve a „slice“ of the processing capability of a system to a group of threads and/or processes (shortly, tasks). This
is done by specifying two scheduling parameters for each group: a budget $Q$ and a period $P$, with the meaning that the tasks in the group are entitled to run on each of the CPUs (processor, or cores when present) available to the OS, for $Q$ time units every period of $P$ time units. This constitutes a scheduling guarantee and a limitation at the same time. This is achieved by a hard-reservation variant of the EDF-based Constant Bandwidth Server (CBS) scheduler (Abeni 1998), implemented as a partitioned scheduling strategy, where each CPU has its own private task queue, and it is scheduled independently. However, when a group is entitled to run on each CPU, the IRMOS scheduler employs a POSIX priority-based real-time scheduling strategy (POSIX 2004) among its tasks, in such a way that, if there are $m$ CPUs, (at most) the $m$ tasks with the highest priority are the ones which actually run. The system performs admission control over admitted reserved groups, so that the overall system capacity may be properly partitioned among concurrently running activities in the system, without overloading it.

Also, the scheduler has a hierarchical configuration capability, by which it is possible to define groups and nested subgroups of real-time tasks with given scheduling parameters.

**Solaris 10**

Solaris 10 provides real-time capabilities (Litchfield, 2007). Its main characteristics include:

- **Fully preemptible kernel**: If a RT process becomes runnable, it will immediately be placed on a CPU if its priority is higher than the thread running on that CPU.

- Interrupts as threads: Interrupts are converted into threads with their own data structures. Interrupts can block on kernel synchronization primitives. This enables the ability to protect data structures in the Solaris kernel with synchronization primitives rather than raising and lowering interrupt priority levels.

- **Real-time scheduling**: Entities needing real-time response latencies can use the real-time scheduling class which offers two options: round-robin or FIFO scheduling.

- **Priority inheriting synchronization primitives**: The primary synchronization primitive in the Solaris kernel is the mutex. They are adaptive in that if the entity owning the mutex is not currently running then the entity attempting to gain the mutex is put to sleep as there is no chance of gaining the mutex while the current owner is sleeping. Mutexes (and other primitives) offer the possibility of “priority inversion.” Solaris mutexes implement the basic priority inheritance protocol. When the high level entity blocks, all of the entities blocking it are given the high level entity’s priority. When they cease to block the thread, their priorities revert to their previous level.

- **POSIX Compliance**: Solaris 10 has almost full support for POSIX 1003.1b and full support for POSIX 1003.1c.
Others
There is also a number of commercially available OSes that are real-time capable, comprising Linux-based ones such as SUSE Linux Enterprise Real Time 10 and Red Hat Enterprise MRG 1.0. The discussion of the features of these other OSes is outside the scope of this document.

CONCLUSION
In this chapter, an overview was done about the real-time support in GPOSes with a particular focus on Linux. As evident from the discussion, there is a growing interest in integrating real-time support at the kernel level, even in General-Purpose Operating Systems, like Linux, other UNIX-es as Solaris, or Microsoft OSes. In fact, these constitute excellent development platforms for a wide variety of soft real-time applications, such as interactive and multimedia applications, thanks to the wide availability of multimedia and compression libraries, and the support for widespread multimedia devices and supports.

Looking at the development trend within the Linux kernel, it is evident how in the last few years, despite the growing computational capabilities of the hardware on which the kernel runs, such issues as interrupt and scheduling latencies, preemptability, timer precision and scheduling policies are gaining more and more importance in the community of users and developers. The current Linux scheduler for non-real-time processes already embeds concepts borrowed from the world of real-time scheduling, even if not in a completely effective way for the purpose of real-time applications.

It is expected that, in the next years, each GPOS will integrate more real-time mechanisms than nowadays, in order to better meet the growing requirements of time sensitive applications.

KEYWORDS
Real-Time Systems, Scheduling, Multi-Processor and Multi-Core Systems, Resource Reservations, Operating Systems, Multimedia, Linux

REFERENCES


Lelli, J. (2010). Design and development of deadline based scheduling mechanisms for multiprocessor systems. Thesis presented for the partial fulfilment of the Master Degree in Computer Engineering at the University of Pisa.


