Real-Time Systems, 28, 237–253, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Real-Time Operating Systems

JOHN A. STANKOVIC
University of Virginia

R. RAJKUMAR
Carnegie Mellon University

1. Introduction

Real-time operating systems (RTOSs) provide basic support for scheduling, resource management, synchronization, communication, precise timing, and I/O. RTOSs have evolved from single-use specialized systems to a wide variety of more general-purpose operating systems (such as real-time variants of Linux). We have also seen an evolution from RTOSs which are completely predictable and support safety-critical applications to those which support soft real-time applications. Such support includes the concept of quality of service (QoS) for open real-time systems, often applied to multimedia applications as well as large, complex distributed real-time systems. Researchers in real-time operating system have developed new ideas and paradigms that enhance traditional operating systems to be more efficient and predictable. Some of these ideas are now found in traditional operating systems and many other ideas are found in the wide variety of RTOS on the market today. The RTOS market includes many proprietary kernels, composition-based kernels, and real-time versions of popular OSs such as Linux and Windows-NT. Many industry standards have been influenced by RTOS research including POSIX real-time extensions, Real-Time Specification for Java, OSEK (automotive RTOS standard), Ada83 and Ada95. This paper provides an overview of the architectures, principles, paradigms, and new ideas developed in RTOS research over the past 20 years. The paper concentrates on research done within the context of complete RTOSs. Note that much more research on RTOSs has been accomplished and published as specific aspects on RTOS. For example, real-time synchronization and memory management research has many exciting results. Also, many ideas found in the companion paper on real-time scheduling can be found in various RTOSs as well.

2. RTOS Taxonomy and Architectures

Real-time operating systems emphasize predictability, efficiency, and include features to support timing constraints. Several general categories of real-time operating systems exist: small, proprietary kernels (commercially available as well as homegrown kernels), real-time extensions to commercial timesharing operating systems such as Unix and

Linux, component-based kernels, QoS-based kernels, and (largely) University-based research kernels.

2.1. Small, Fast, Proprietary Kernels

The small, fast, proprietary kernels come in two varieties: homegrown[1] and commercial offerings.[2] Both varieties are often used for small embedded systems when very fast and highly predictable execution must be guaranteed. The homegrown kernels are usually highly specialized to the application. The cost of uniquely developing and maintaining a homegrown kernel, as well as the increasing quality of the commercial offerings is significantly reducing the practice of generating homegrown kernels. In addition, component-based OSs (see Section 2.3) are also reducing the need for homegrown kernels. For both varieties of proprietary kernels, to achieve speed and predictability, the kernels are stripped down and optimized versions of timesharing operating systems. To reduce the run-time overheads incurred by the kernel and to make the system fast, the kernel:

- has a fast context switch,
- has a small size (with its associated minimal functionality),
- responds to external interrupts quickly (sometimes with a guaranteed maximum latency to post an event but, generally, no guarantee is given as to when processing of the event will be completed; this later guarantee can sometimes be computed if priorities are assigned correctly),
- minimizes intervals during which interrupts are disabled,
- provides fixed or variable sized partitions for memory management (i.e., no virtual memory) as well as the ability to lock code and data in memory,
- provides special sequential (often memory-based) files that can accumulate data at a fast rate.

To deal with timing requirements, the kernel

- supports multi-tasking,
- provides a priority-based preemptive scheduling mechanism,
- provides bounded execution time for most primitives,
- maintains a high-resolution real-time clock,

- provides for special alarms and timeouts,
- supports real-time queuing disciplines such as earliest deadline first and primitives for jamming a message into the front of a queue,
- provides primitives to delay processing by a fixed amount of time and to suspend/ resume execution.

In general, the kernels also perform multi-tasking and inter-task communication and synchronization via standard primitives such as mailboxes (message queues), events, signals, mutexes, and semaphores. While all these latter features are designed to be fast, ‘‘fast’’ is a relative term and not sufficient when dealing with real-time constraints. Nevertheless, many real-time system designers use these features as a basis upon which to build real-time systems. This has been effective in small embedded applications such as instrumentation, communication front-ends, intelligent peripherals and many areas of process control. Since these applications are simple, it is relatively easy to show that all timing constraints are met. Consequently, the kernels provide exactly the minimal functionality that is needed. However, as applications become more complex, it becomes more and more difficult to craft a solution based on priority-driven scheduling where all timing, computation time, resource, precedence, and value requirements are mapped to a single priority for each task. In these situations, demonstrating predictability can be rather difficult.

2.2. Real-Time Extensions to Commercial Operating Systems

A second approach to real-time operating systems is the extension of commercial products, for example, extending Unix to RT-Unix (Furht et al., 1991), Linux to RT-Linux (FSLLabs; Niehaus, KURT; RedIce Linux), or POSIX to RT-POSIX, or MACH to RT-MACH (Tokuda et al., 1990), or CHORUS to a real-time version (CHORUS system). The real-time version of commercial operating systems are generally slower and less predictable than the proprietary kernels, but have greater functionality and better software development environments—very important considerations in many large or complex applications. Another significant advantage is that they are based on a set of familiar interfaces (standards) that facilitate portability. For Unix, since many variations of Unix have evolved, an IEEE standardization effort, called POSIX, has defined a common set of user-level interfaces for operating systems. The effort has focussed on eleven important real-time related functions: timers, priority scheduling, shared memory, real-time files, semaphores, interprocess communication, asynchronous event notification, process memory locking, asynchronous I/O, synchronous I/O, and threads.

Various problems exist when attempting to convert a non real-time operating system to a real-time version. These problems can exist both at the system interface as well as in the implementation. For example, in Unix, interface problems exist in process scheduling due to the nice and setpriority primitives and its round-robin scheduling policy. In addition, the timer facilities are too coarse, memory management (of some versions) contains no method for locking pages into memory, and interprocess communication facilities do not support fast and predictable communication often resulting in different forms of priority inversion (Sha et al., 1990). The implementation problems include intolerable overhead, excessive latency in responding to interrupts, partly but very importantly, due to the non-preemptability of the kernel, and internal FIFO queues. These and other problems can and have been solved to result in a real-time operating system that is used for both real-time and non real-time processing. However, because the underlying paradigm of timesharing systems still exists, application developers must be careful not to use certain non real-time features that might insidiously impact the real-time tasks.

Real-time capabilities can be added to operating systems in multiple ways. It is illustrative to study how many real-time versions of Linux have been created and commercialized in recent years. These versions can be grouped into the following categories.

- *Compliant kernels*: In this approach, an existing real-time operating system is modified such that Linux binaries can be run without any modification. Essentially, the functionality and semantics of Linux system calls need to be appropriately emulated under the native operating system. For example, LynxOS from LynuxWorks adopts this approach.

- *Dual kernels*: In this approach, a hard but thin real-time kernel sits below the native operating system (such as Linux or FreeBSD), and traps all accesses to and interrupts from the underlying hardware. The thin kernel schedules several hard real-time tasks co-located with it, and runs the native OS as its lowest priority task. As a result, native applications can be run without change, while hard real-time tasks can get excellent performance and predictability. A means of (non-real-time) communication is also provided between the thin real-time kernel and the native non-real-time kernel for data exchange purposes. The downside of this approach is that there is no memory protection between the real-time tasks and the native/thin kernels. As a result, the failure of any real-time task can lead to a complete system crash. The thin real-time kernel also needs to have its own set of device drivers for real-time functionality. RT-Linux (FSLLabs) is an example of this approach.

- *Core kernel modifications*: In this approach, changes are made to the core of a non-real-time kernel in order to make it predictable and deterministic enough so as to behave as a real-time OS. Using fixed-priority scheduling with a O(1) scheduler, employing high-resolution timers, making the kernel preemptive (so that a lower priority process in the kernel space due to an ongoing system call can be preempted by a higher priority process that becomes eligible to run), support for priority inheritance protocols to minimize priority inversion, making interrupt handlers schedulable using kernel threads, the use of periodic processes, replacing FIFO queues with priority queues and optimizing long paths through the kernel are typical means of accomplishing this goal. TimeSys Linux (based on CMU's Linux/RK (Oikawa and Rajkumar, 1999) discussed in Section 2.5.5) and to a smaller extent MontaVista Linux fall under this category.

- *The Resource kernel approach*: In this approach, the kernel is extended to provide support for resource reservations in addition to the traditional fixed-priority preemptive scheduling approach. The latter approach can run into problems when a relatively high-priority process overruns its expected execution time or even goes into an infinite loop. Resource kernels support and enforce resource reservation, such that no misbehaving task can directly impact the timing behavior of another task. CMU's Linux/RK and its commercial cousin, TimeSys Linux, and fall into this category.

2.3. Component-Based Kernels

A number of systems such as OS-Kit (Ford et al., 1997), Coyote (Bhatti et al., 1999), PURE (Beuche et al., 1999), 2K (Kon et al., 1998), MMLite (Helander and Forin, 1998), and Pebble (Gabber et al., 1999) have a common intent to deal with operating system construction through composition. They define OS components that can be selectively included to compose an RTOS that can be tailored to the application(s) at hand.

OS-Kit provides a set of operating system components that can be combined to configure an operating system. However, it does not supply any rules to help build an operating system. Coyote is focussed on communication protocols, and its ability for re-configuration might be adopted for operating system and embedded application areas. PURE is explicitly concerned with providing operating system components for configuration and composition of operating systems for embedded applications. PURE uses an object-oriented methodology to provide different components for configuration and customization of operating systems for embedded applications. 2K emphasizes adaptability issues to allow applications to be as customizable as possible. 2K is also concerned with component-based software for small mobile devices, or personal digital assistants (PDAs).

To explore the concepts of a component-based RTOS, consider two component-based RTOSs in more detail. MMLite is an object-based, modular system architecture that provides a menu of components for use at compile-time, link-time, or runtime to construct a wide range of applications. A component in MMLite consists of one or more objects. Multiple objects can reside in a single namespace. When an object needs to send a message to an object in another namespace for the first time, a proxy object is created in the sending object's namespace that transparently handles the marshaling of parameters.

A unique aspect of MMLite is its focus on support for transparently replacing components while these components are in use (mutation). MMLite uses COM interfaces, which in turn support dynamic reconfigurability on a per-object and per-component basis. However, COM does not provide protection between the components. The base menu of the MMLite system contains components for heap management, dynamic on-demand loading of new components, machine initialization, timer and interrupt drivers, scheduler, threads and synchronization, namespaces, file system, network, and virtual memory. These components are typically very small (500–3000 bytes on the **×86** architecture), although the network component is much larger (84,832 bytes on **×86**). The resulting MMLite system can be quite small: the base system is 26 Kbytes on **×86**, and 20 Kbytes on the ARM architecture. It is not clear to what extent MMLite provides users with the ability to easily select components that the MMLite developers write, and to what extent users themselves define and utilize their own new components. Although there has been an apparent emphasis on developing minimal-sized components (in number of bytes), analysis tools regarding the runtime performance of components due to namespace resolution and the creation and loading of proxy objects is lacking.

Pebble is a new operating system designed to be an efficient application-specific operating system and to support component-based applications. It also supports complex embedded applications. As an operating system, it adopts a microkernel architecture with a minimal privileged-mode nucleus that is only responsible for switching between protection domains. The OS functionality is provided by user-level components (servers), which can be replaced, augmented, or layered. The programming model is client/server; client components (applications) request services from system components (servers). Examples of system components are the interrupt dispatcher, scheduler, portal manager, device driver, file system, virtual memory, and so on. The Pebble kernel and its essential components (interrupt dispatcher, scheduler, portal manager, real-time clock, console driver, and idle task) need approximately 560 Kbytes of memory. Components are like processes, where each one executes in its own protection domain (PD).

In Pebble, a PD includes a page table and a set of portals. Portals provide communication between PDs. For example, if there is a portal from PD1 to PD2, then a thread executing in PD1 can invoke a specific service (entry point) of PD2. Therefore, components communicate through transferring threads from one PD to another using portals.

The PD concept together with the portal concept can be understood as a component infrastructure. While Pebble PDs provide the means to isolate the components, portals provide the means for components to communicate with each other. Instantiation and management of portals are performed by an operating system component, Portal Manager. For instance, the instantiation process involves the registration of a server (any system or application component) in a portal and the request of a client for that portal. In Pebble, it is possible to dynamically load and to replace system components to fulfill applications requirements.

2.4. QoS-Based Kernels

QoS research has been extensive, first as applied to networking then to general distributed computing. More recently, QoS has been applied to soft real-time systems. In these systems, a guarantee is given that a certain amount of resources is assigned to a task or application. In other cases, there are differentiated guarantees meaning that certain classes of tasks are guaranteed resources compared to another class of tasks. For example, tasks dealing with the control of the plant may be required to obtain twice the resources than tasks reporting the results to a command center. The resources being controlled may just be the CPU or a set of resources. Many research results exist for developing algorithms to control the guarantees. Sometimes, these algorithms are implemented as monitors on top of an RTOS. In other cases, the algorithms may be implemented as middleware (Brandt et al., 1998). The algorithms differ in their approach and utilize many different techniques such as fair-share scheduling (Jeffay et al., 1998), proportional scheduling (Stoica et al., 1996), rate-based scheduling (Jeffay, 2001), reservations, and feedback control.

In this paper, we are more interested in RTOS that incorporate QoS support such as RT Mach (Tokuda et al., 1990) and Rialto (Jones et al., 1996, 1997). Both of these RTOSs allow users to negotiate with the RTOS for a certain amount of resources. RT-Mach employs reservations to support QoS. RT-Mach supports multimedia applications and both real-time and non-real-time tasks. Rialto allows for multiple, independent applications to co-exist. A system-wide planner reasons about the resource allocations between applications. This is similar to the reservation and admission control type work discussed above, but here, independent applications are supported on a single platform. Rialto also has support for overload and for re-negotiation of guarantees.

2.5. Research Kernels

Many past and current University-based research-oriented real-time operating systems have been developed. These projects addressed many of the following research issues including:

- identifying the need for new approaches which challenge the basic assumptions made by timesharing operating systems and developing those new paradigms;
- developing real-time process models:
    - some systems use the standard process model both to program with and at execution time,
    - some systems use the process model to program with but translate into a different run-time model to help support predictability and on-line guarantees,
    - some systems use real-time threads;
- developing real-time synchronization primitives such as those that support priority inheritance and priority ceiling protocols;
- developing solutions that facilitate timing analysis of both the initial system and upon modifications (the real-time scheduling algorithms play a large role here);
- strongly emphasizing predictability not only of the kernel but also providing good support for application-level predictability;
- retaining significant amounts of application semantics at run time;
- developing support for fault tolerance;

- investigating object-oriented approaches;
- providing support for multiprocessor and distributed real-time systems including end-to-end timing constraints;
- developing support for QoS;
- attempting to define a real-time micro-kernel;
- providing support for real-time programming languages such as the Real-Time Specification of Java (JSR-00001).

We survey several research projects as representative of a much wider set of work in the field.

2.5.1. MARS

The MARS kernel (Damm et al., 1989; Kopetz et al., 1989) offers support for controlling a distributed application based entirely on the passage of time (rather than asynchronous events) from the environment. Emphasis is placed on an *a priori* static analysis to demonstrate that all the timing requirements are met. An important feature of this system is that flow control on the maximum number of events that the system handles is automatic and this fact contributes to the predictability analysis. This system is based on a paradigm, that is, the time-triggered model, that is different than what is found in timesharing systems. The scheduling approach is static and table-driven. Support for distributed real-time systems includes a hardware-based clock synchronization algorithm and a TDMA-like protocol to guarantee timely message delivery. A number of extensions to the original work have added flexibility to handle more dynamic situations. The time-triggered approach advocated in MARS has seen success in the automotive industry and in several other safety-critical application domains.

2.5.2. SPRING

The Spring kernel (Stankovic and Ramamritham, 1995; Stankovic et al., 1999) contains real-time support for multiprocessors and distributed systems. A novel aspect of the kernel is the dynamic planning-based scheduling of tasks that arrive dynamically. Such tasks are subject to admission control and dynamically acquire reservations for resources. This takes tasks' time and resource constraints into account and avoids the need to *a priori* compute worst-case blocking times. Safety-critical tasks are dealt with through static table-driven scheduling. The kernel also embodies a reflective architecture (Stankovic and Ramamritham, 1995) that retains a significant amount of application semantics at run time. This approach provides a high degree of flexibility along with support for graceful degradation. These planning and application semantic features are integrated to provide direct support for achieving both application- and system-level predictability. The kernel also uses global replicated memory to achieve predictable distributed communication. The abstractions provided by the Kernel include dynamic guarantees, reservations, planning, and end-to-end timing support. Spring, like MARS, presents a new paradigm for real-time operating systems, but unlike MARS it strives for a more flexible combination of off-line and on-line techniques. Concepts of admission control, reflection and reservations found in the Spring kernel have been used by many other systems.

2.5.3. ARTS

The ARTS kernel (Tokuda and Merger, 1989) provides a distributed real-time computing environment that works in conjunction with the static priority-driven preemptive scheduling paradigm. The kernel supports the notion of real-time objects and real-time threads. Each real-time object is time-encapsulated. This is enforced by a time fence mechanism which provides a run-time check that ensures that the slack time is greater than the worst-case execution time for an object invocation about to be performed. If it is, the operation proceeds, else it is aborted. Each real-time thread can have a value function, timing constraints, worst-case execution time, phase, and delay value associated with it. Communication (object invocation) proceeds in a request–accept–reply fashion, but does not address deadlines for messages. A real-time transport protocol has been developed. The ARTS kernel is also tied to various tools that *a priori* analyze the system-wide schedulability of the system.

2.5.4. HARTOS

The hexagonal architecture for real-time systems (HARTS) consists of multiple sites connected by a hexagonal mesh network. Each site may be a uniprocessor or multiprocessor and contains an intelligent network processor. The intelligent network processor handles much of the low-level communication functions. An experimental operating system called HARTOS (Kandlur et al., 1992) is a distributed real-time kernel running on HARTS. On each site, HARTOS runs in conjunction with the commercial uniprocessor OS, pSOS. Hence, by itself, HARTOS is not a full operating system. Rather, HARTOS focusses on interprocess communication, thereby providing some support for distributed real-time systems. In particular, HARTOS supports message send and receive, non-queued event signals, reliable streams, and message scheduling that provides a best-effort approach in delivering a message by its deadline. Support for fault-tolerant routing, clock synchronization, and for replicated processes are also planned.

2.5.5. RK

In extensions to Real-Time Mach, Mercer et al. (1994) added the notion of processor reservations based on the Liu and Layland periodic task model of each task obtaining C_i units of time every T_i units of time. Rajkumar et al. (1998) generalized this concept to the notion of a resource kernel, which is defined as one which provides guaranteed, timely and enforced access for applications to system resources. In addition, scheduling policies could be changed within the OS without affecting any guarantees. Resources that could be guaranteed access to can include CPU cycles (Rajkumar et al., 1988), disc bandwidth (Molano et al., 1997; Saewong and Rajkumar, 2003), network bandwidth (Ghosh and Rajkumar, 2002) or memory space (Easwaran and Rajkumar, 2004). Resource reservations on multiple resources could also be combined to form a resource set to which one or more applications could be bound. An application bound to a resource set essentially has access to a ''virtual machine'' that comprises a time- or space-multiplexed subset of the underlying physical resources. This virtualization also enabled the binding of binary images to be bound to arbitrarily sized reservations (w/o access to source). An interesting variation of the priority inversion problem occurs when processes bound to two (or more) different reservations need to share a resource (such as the X-windows server). Solutions to this problem are also based on variants of priority inheritance and led to reservation inheritance protocols (de Niz et al., 2001).

Counter-intuitive as it may seem, the reservation model of guaranteeing and enforcing C_i units of time every T_i units of time is not just useful for periodic tasks. It can also act as a traffic shaper to aperiodic tasks in the exact same spirit of a deferrable (Sprunt et al., 1989) or sporadic server (Strosnider et al., 1995).

3. Paradigms

Real-time operating systems utilize various paradigms. Key concepts found in these paradigms include: hard and soft real-time guarantees, admission control, reflection, reservations, and resource kernels. Many of these key concepts work together in achieving the overarching paradigm presented by a particular kernel.

3.1. Hard and Soft Real-Time Guarantees

In general, the smaller, more deterministic kernels provide support for hard deadline systems. Here all the inputs and system details are known, and careful design and analysis can result in meeting hard deadline requirements. In performing the analysis it is also possible to carefully account for the kernel overheads. Safety-critical hard real-time systems also typically employ comfortable margins on resource utilization (such as ensuring that total utilization on a resource does not exceed 50–60%).

The larger, more dynamic, more probabilistic kernels provide support for soft real-time systems. Here quality of service guarantees are defined and shown to be met in a probabilistic sense. We sometimes find hierarchical real-time scheduling or partitioned scheduling to handle different classes of tasks.

3.2. Admission Control

Admission control is a function that decides if new work entering the system should be admitted or not. The key ingredients of admission control include a model of the state of system resources, knowledge about the incoming request, the exact algorithm to make the admission control decision, and policies for the actions to take upon admission and upon rejection.

First consider hard real-time systems. Many hard real-time systems are statically scheduled and operate in highly deterministic fashion. This facilitates the timing analysis required of these systems and there is no notion of admission control.

But, many hard real-time systems operate in dynamic environments where static scheduling is too costly or rigid. What is required is a solution that enables on-line careful timing analysis and dynamic scheduling. A solution provided in the Spring kernel (Stankovic and Ramamritham, 1989, 1995) included the synergistic combination of admission control, resource reservation, and reflection; so this concept already exists in the hard real-time domain. Here the model of the state of the system is a detailed timeline that identifies the start and finish time (based on a worst-case execution time model) for each admitted task on each resource that it requires. Significant reflective information is known about each incoming task because they are pre-analyzed for a particular real-time system; there are no general purpose on-the-fly tasks created. The reflective information known about the requested work includes the worst-case execution time, shared data required by this task, precedence constraints, importance level, which tasks communicate with this task, deadline, etc. The algorithm is a heuristic that schedules the task on the detailed time line along with all the previously admitted tasks in such a manner that if successful, all the tasks will meet their deadlines. See Zho et al. (1987) for a detailed description of the algorithm. If the task is admitted, it has been assigned a very specific time-slice (although it may actually execute early under certain conditions). If it is not admitted, then a separate policy is invoked to decide what action to take. Typical actions include: try a simpler version of the task if any exists, or if the deadline is far away try to schedule the task on another node, or if the deadline is close then just reject this task. These policies can be modified based on the importance of the task. The low-level details of the entire guaranteed schedule are available to the application. A large amount of application semantic information is pushed into the kernel (via the compiler and a special system description language). For example, the process control block (PCB) contains, in addition to the typical information, worst case execution times, deadline information, precedence requirements, a communication graph, fault-tolerance information, etc.

Work on supporting QoS for audio and video has also used admission control and reservations. In many systems, various amounts and types of reflective information are also used. The typical model of the system has been utilizations identified independently for multiple resources such as CPU, network bandwidth, disc, and memory. The precise admission control algorithm has varied from system to system, but it is usually based on some simple sum of utilizations of all previously admitted work together with the requirements of the new work. For example, if the utilizations are less then 100% then the work is admitted under the earliest deadline first scheduling policy, else some renegotiation might take place and the work admitted with less guaranteed service. Application-level information regarding peak loads, end-to-end delays, jitter requirements, etc. are pushed down into the network layers. Performance data may be pushed up to the application level if renegotiation is required.

3.3. Resource Reservation

Resource reservation is the act of actually assigning portions of resources to a task. In the early OS work for maintaining a reasonable level of multiprogramming, no resource reservation was done.

In the Spring kernel, explicit resource reservation was done very precisely on each type of resource and in a coordinated manner. For example, if TASK A has the CPU reserved from time 10 to 15 time-units, this task would also be assured to have exclusive access to any needed resources including shared data structures during that same specific time interval. In fact, this integrated resource reservation is orchestrated so carefully that semaphores are not required (i.e., resource conflicts are avoided via scheduling). Such precise resource reservation is valuable for a hard real-time system where careful timing guarantees are required. For soft real-time systems, reservations of various types are used to support QoS. As mentioned above these include Rialto, RT-Mach, and RK.

3.4. Reflection

Reflective information is comprised of meta data about the system. It can be meta data about the application and it can be meta data about the performance and properties of the OS (or microkernels). All OSs are reflective to some degree. However, if we elevate the notion of reflection to a central principle of the system, then it is possible to build more flexible systems (Stankovic and Ramamritham, 1995; Yokote, 1992), and highlight the need to choose the right reflective information for a given system. Some of this flexibility can be used to obtain better performance, for example, better performance to meet real-time constraints.

In the Spring kernel, there is significant reflective information which enables on-the-fly performance guarantees in a dynamic setting. Most of the reflective information is from the application down to the OS. Some work was also undertaken to make use of reflective information from the kernel (regarding schedules) back to the application (agile manufacturing).

Supporting QoS for audio and video also makes use of reflective information about the needs of the various audio and video streams. This is usually in the form of bandwidth, peak load, end-to-end, and jitter requirements. In addition, there is no reason why the OS itself cannot make reflective information about the OS available to applications as is used in systems like the exokernel work (Engler et al., 1997) and in various thread packages

(Anderson et al., 1991). Because all this information is available to admission control and renegotiation policies, better application-specific performance is possible and reservations can be dynamically adjusted if necessary.

3.5. Resource Kernels

Resource kernels provided interfaces to create/destroy resource sets,[3] create/destroy reservations on different resources, attach/detach reservations to/from resource sets, resize reservations, bind/unbind processes to/from resource sets, and to obtain process usage information on resources. Oikawa and Rajkumar (1999) proposed the notion of a ''portable'' resource kernel, which comprised of a single code base and a uniform API using the resource kernel concept but one that could be hosted within different operating systems. Linux/RK, FreeBSD/RK and Windows NT/RK (referred to as NT/RK) were built to demonstrate the notion. A real-time version of Java that could utilize reservations was also built (de Niz and Rajkumar, 2000). Resource kernels also introduced different types of reservations. A task bound to a hard reservation could not exceed its reservation limits under any conditions. A task bound to a firm reservation could use any cycles (or bandwidth) that would go unused by any unreserved tasks. A task bound to a soft reservation could use any cycles/bandwidth not reserved by any other task. Linux/RK has also been commercially available through TimeSys Corporation. Reservations can also be (recursively) hierarchical (Saewong et al., 2002), in that reservations can be created within reservations. Deng and Liu (1997) have also studied two-level reservation schedulers extensively and refer to the approach as ''open systems'' scheduling since it enables tasks operating within one partition to be unaware of tasks running within other partitions. Admission control, scheduling subsystems, accounting and enforcement subsystems are key components of a resource kernel. An abstraction called a resource control list specifies which principal (user or process) can use a system resource, when and for how long (Miyoshi and Rajkumar, 2001). This concept, which is analogous to access control lists in (say) filesystems, is helpful in protecting real-time tasks from malicious or unintentional denial-of-service attacks.

To support audio and video processing percentages of resources are reserved independently. Some resources such as memory might be explicitly reserved, while other resources such as cpu or network bandwidth might be reserved in the form of percentages. In some sense, resource reservations need to have application information (what data structures does a task need, or how much cpu does an application require) pushed down into the OS so that application-specific performance can be attained. For renegotiation of QoS, OS-level performance data is pushed up to the application so that it may modify its requirements if possible (and obtain a new reservations).

4. Summary

The domain of real-time operating systems has been a very active area of research in recent years. The field has seen many RTOSs being built with many different ideas, principles and paradigms. Since the application domain of real-time systems is broad, different approaches are needed for different situations. RTOS research has influenced many products (with many companies being formed around a particular RTOS solution). See http://www.faqs.org/faqs/ for a list of commercial real-time operating systems. Real-time research has also influenced many standards such as POSIX real-time extensions, real-time variants of LINUX, Real-Time Specification of Java, OSEK (automotive RTOS standard), Ada83 and Ada95. For a list of Real-time versions of Linux, see http://www.realtimelinuxfoundation.org/. New challenges continue to be posed to RTOSs, for example, new solutions are needed for wireless sensor networks and other pervasive computing applications. Solutions are needed to better support large scale distributed real-time systems. The tension between RTOS predictability and efficiency and the need for standards and good development environments continues to breed new research. RTOS support for component-based real-time systems must also be studied further. Recently, significantly attention has also been paid to energy-aware RTOSs (see for example, AbouGhazaleh et al., 2003 and Saewong and Rajkumar, 2003) where the RTOS attempts to minimize energy consumption while satisfying timing constraints. However, additional practical experience with such RTOSs is desirable.

Notes

1. Examples include Alger and Lala (1986) and Holmes et al. (1987).
2. Examples of commercials kernels include QNX, PDOS, pSOS, VCOS, VRTX32, Integrity, VxWorks and run-time environments for languages such as Ada.
3. A resource set is a collection of reservations on different systems resources and can be considered to be a virtual machine whose resources are guaranteed to one or more applications bound to the resource set (see, Rajkumar et al., 1998).

References

AbouGhazaleh, N., Mosse, D., Childers, B., Melhem, R., and Craven, M. 2003. Collaborative operating system and compiler power management for real-time applications. In *Proceedings of The Real-time Technology and Application Symposium.*

Alger, L., and Lala, J. 1986. Real-time operating system for a nuclear power plant computer. In *Proceedings of the Real-Time Systems Symposium.*

Anderson, T., Bershad, B., Lazowska, E., and Levy, H. 1991. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on OS Principles.*

Beuche, D., Geurrouat, A., Papajewski, H., Schroder-Preikschat, W., Spinczyk, O., and Spinczyk, U. 1999. The pure family of object oriented operating systems for deeply embedded systems. In *Proceeding of the 2nd International Symposium on Object Oriented Real-Time Distributed Computing.*

Bhatti, N., Hiltunen, M., Schlichting, R., and Chiu, W. 1998. Coyote: a system for constructing fine grain configurable communication services. *ACM Trans. Comput. Syst.* 16(4).

Brandt, S., Nutt, G., Berk, T., and Mankovich, J. 1998. A dynamic quality of service middleware agent for mediating application resource usage. *RTSS.*

Carlow, G. D. 1984. Architecture of the space shuttle primary avionics software system. *CACM* 27(9).

Chorus Kernel v3 r4.0 Programmer's Reference Manual. Technical Report CS/TR-91-71, Chorus system, 91/09.

Damm, A., Reisinger, J., Schnakel, W., and Kopetz, H. 1989. The real-time operating system of MARS. *Operat. Syst. Rev.* 141–157.

Deng, Z., and Liu, J. 1997. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97).* pp. 308–319.

de Niz, D., and Rajkumar, R. 2000. Chocolate: a reservation-based real-time java environment on Windows NT. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium.* Washington, D.C.

de Niz, D., Abeni, L., Saewong, S., and Rajkumar, R. 2001. Resource sharing in reservation-based systems. In *Proceedings of the IEEE Real-Time Systems Symposium.*

Eswaran, A., and Rajkumar, R. 2004. Memory reservation in resource kernels. Technical Report, Electrical and Computer Engineering, Carnegie Mellon University.

Engler, D., Kaashoek, M. F., and O'Toole, J. 1995. Exo-kernel: an operating system architecture for application level resource management. MIT TR, March 24.

Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., and Shivers, O. 1997. The flux oskit: a substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium Operating System Principles.*

FSLLabs, RT-Linux, http://www.fsmlabs.com/.

Furht, B., Grostick, D., Gluch, D., Rabbat, G., Parker, J., and Roberts, M. 1991. *Real-Time Unix Systems.* Norwell, MA: Kluwer Academic Publishers.

Gabber, E., Small, C., Bruni, J., Brustolini, J., and Silberschatz, A. 1999. The pebble component based operating system. In *Proceedings of the USENIX Annual Technical Conference.*

Ghosh, S., and Rajkumar, R. 2002. Resource management of the OS network subsystem. In *Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing.*

Gudmundsson, O., Mose, D., Ko, K., Agrawala, A., and Tripathi, S. 1992. MARUTI, An environment for hard real-time applications. In A. Agrawala, K. Gordon, and P. Hwang, *Mission Critical Operating Systems.* IOS Press.

Helander, J., and Forin, A. 1998. MMLite: a highly componentized system architecture. *SIGOPS.*

Holmes, V. P., Harris, D., Piorkowski, K., and Davidson, G. 1987. Hawk: an operating system kernel for a real-time embedded multiprocessor. Sandia National Labs Report.

Jensen, D. 1992. The kernel computational model of the alpha real-time distributed operating system. In A. Agrawala, K. Gordon and P. Hwang, *Mission Critical Operating Systems.* IOS Press.

Jones, M., Rosu, D., and Rosu, M.-C. 1997. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles.*

Jones, M., Barbera III, J., and Forin, A. 1996. An overview of the rialto real-time architecture. In *Proceedings of the 7th ACM Sigops European Workshop.*

Jeffay, K., Stone, D., and Poirier, D. 1991. YARTOS, kernel support for efficient, predictable real-time systems. *Workshop on Real-Time Operating Systems and Software.*

Jeffay, K. 2001. Rate-based allocation models for embedded systems. *EMSOFT.*

Jeffay, K., Donelson Smith, F., Moorthy, R., and Anderson, J. 1998. Proportional share scheduling of operating system services for real-time applications, *RTSS.*

JSR-00001: The Real-Time Specification for Java, Java Community Process. http://jcp.org/aboutJava/communityprocess/first/jsr001/.

Kandlur, D., Kiskis, D., and Shin, K. 1992. A real-time operating system for HARTS. In A. Agrawala, K. Gordon and P. Hwang, *Mission Critical Operating Systems.* IOS Press.

Kon, F., Singhai, A., Campbell, R., Carvalho, D., Moore, R., and Ballesteros, F. 1998. 2K: a reflective component based operating system for rapidly changing environments. *ECOOP.*

Kopetz, H., Demm, A., Koza, C., and Mulozzani, M. 1989. Distributed fault tolerant real-time systems: the MARS approach. *IEEE Micro* 25–40.

Lee, C., Yoshida, K., Mercer, C. W., and Rajkumar, R., 1996. Predictable communication protocol processing in real-time mach. In *Proceedings of the Real-time Technology and Applications Symposium.*

Locke, C. D. 1992. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems* 4(1): 37–52.

Mercer, C. W., Zelenka, J., and Rajkumar, R. 1994. On predictable operating system protocol processing. Technical Report CMU-CS-94-165, School of Computer Science, Carnegie Mellon University.

Miyoshi, A., and Rajkumar, R. 2001. Protecting resources with resource control lists. In *Proceedings of 7th IEEE Real-Time Technology and Applications Symposium.* Taipei, Taiwan.

Molano, A., Juvva, K., and Rajkumar, R. 1997. Real-time filesystems: guaranteeing timing constraints for disc accesses in RT-Mach. In *Proceedings of the IEEE Real-Time Systems Symposium.*

Niehaus, D. KURT: The KU Real-Time Linux. http://www.itc.ku.edu/kurt/old-index.html.

Oikawa, S., and Rajkumar, R. 1999. Portable RK: a portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium.* Vancouver.

Rajkumar, R., Sha, L., and Lehocklzy, J. 1988. Real-time synchronization protocols for multiprocessors. *Real-Time Systems Symposium.*

Rajkumar, R., Juvva, K., Molano, A., and Oikawa, S. 1998. Resource kernels: a resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking.*

Ramamritham, K., and Stankovic, J. A. 1991. Scheduling strategies adopted in spring: an overview. In Andre van Tilborg, and Gary Koob *Foundations of Real-Time Computing: Scheduling and Resource Management.*

Ramamritham, K., Stankovic, J. A., and Shiah, P. 1990. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 1(2): 184–194.

Ramamritham, K., Stankovic, J. A., and Zhao, W. 1989. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers* 38(8): 1110–1123.

Ready, J. 1986. VRTX: a real-time operating system for embedded microprocessor applications. *IEEE Micro* 8–17.

RedIce Linux. http://www.linuxdevices.com/news/NS6639736490.html.

Saewong, S., and Rajkumar, R. 1999. Cooperative scheduling of multiple resources. In *Proceedings of the IEEE Real-Time Systems Symposium.*

Saewong, S., Rajkumar, R., Lehoczky, J. P., and Klein, M. H. 2002. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems.*

Saewong, S., and Rajkumar, R. 2003. Practical voltage-scaling for fixed-priority RT-systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).*

Schwan, K., Geith, A., and Zhou, H. 1990. From Chaos$^{\text{base}}$ to Chaos$^{\text{arc}}$: a family of real-time kernels. In *Proceedings of the Real-Time Systems Symposium.* pp. 82–91.

Sha, L., Rajkumar, R., and Lehoczky, J. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transac. Computers* 39(3): 1175–1185.

Stankovic, J. A. 1988. Misconceptions about real-time computing. *IEEE Computer* 21(10).

Stankovic, J. A., and Ramamritham, K. 1989. The spring kernel: a new paradigm for real-time operating systems. *ACM Operating Systems Review* 23(3): 54–71.

Stankovic, J. A., and Ramamritham, K. 1991. The spring kernel: a new paradigm for hard real-time operating systems, *IEEE Software* 8(3): 62–72.

Stankovic, J., and Ramamritham, K. 1995. A reflective architecture for real-time operating systems. In *Advances in Real-Time Systems.* Prentice Hall, pp. 23–38.

Stankovic, J. A. 1991. SpringNet: a scalable architecture for high performance, predictable, distributed, real-time computing. University of Massachusetts, Technical Report, pp. 91–74.

Stankovic, J., Ramamritham, K., Niehaus, D., Humphrey, M., and Wallace, G. 1999. The spring system: integrated support for complex real-time systems. *Real-Time Systems Journal* Special Issue 16(2/3).

Stoica, I., Addel-Waheb, H., Jeffay, K., Baruah, S., Gehrke, J., and Plaxton, C. 1996. A proportional share resource allocation algorithm for real-time time-shared systems. *RTSS.*

SYSTRAN Corporation. Scramnet Network Reference Manual, Dayton, Ohio, 45432.

Tokuda, H., and Mercer, C. 1989. ARTS: a distributed real-time kernel. *ACM Operating Systems Review* 23(3).

Tokuda, H., Nakajima, T., and Rao, P. 1990. Real-time mach: towards a predictable real-time system. In *Proceedings of the Usenix Mach Workshop.*

van Scoy, R., Bamberger, J., and Firth, R. 1992. An overview of DARK. In A. Agrawala, K. Gordon, and P. Hwang, *Mission Critical Operating Systems.* IOS Press.

Sprunt, B., Sha, L., and Lehoczky, J. P. 1989. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems* 1(1): 27–60.

Strosnider, J. K., Lehoczky, J. P., and Sha, L. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers* 44(1): 73–91.

Yokote, Y. 1992. The apertos reflective operating system. In *Proceedings of Object-oriented Programming Systems, Languages, and Applications.* ACM Press.

Zhao, W., and Ramamritham, K. 1987. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *Journal of Systems and Software* 7: 195–205.

Zhao, W., Ramamritham, K., and Stankovic, J. A. 1987. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transac. Software Eng.* SE-12(5): 567–77.