11

REAL-TIME OPERATING SYSTEMS AND STANDARDS

In this chapter we present a brief overview of the state of art of real-time systems and standards. We first discuss the most common operating systems standard interfaces that play a major role for developing portable real-time applications. Then we give a brief description of the most used commercial and open source real-time kernels available today, and we finally present some research kernels, developed within the academia to experiment novel features and lead the future development.

11.1 STANDARDS FOR REAL-TIME OPERATING SYSTEMS

The role of standards in operating systems is very important as it provides portability of applications from one platform to another. In addition, standards give the possibility of having several kernel providers for a single application, so promoting competition among vendors and increasing quality. Current operating system standards mostly specify portability at the source code level, requiring the application developer to recompile the application for every different platform. There are four main operating system standards available today:

- RT-POSIX, which is the main general-purpose operating system standard, with real-time extensions;
- OSEK, for the automotive industry;
- APEX, for avionics systems;
- μITRON, for embedded systems.

11.1.1 RT-POSIX

The goal of the POSIX standard (Portable Operating System Interface based on UNIX operating systems) is the portability of applications at the source code level. Its real-time extension (RT-POSIX) is one of the most successful standards in the area of real-time operating systems, adopted by all major operating systems.

The standard specifies a set of system calls for facilitating concurrent programming. Services include mutual exclusion synchronization with priority inheritance, wait and signal synchronization via condition variables, shared memory objects for data sharing, and prioritized message queues for inter-task communication. It also specifies services for achieving predictable timing behavior, such as fixed priority preemptive scheduling, sporadic server scheduling, time management with high resolution, sleep operations, multipurpose timers, execution-time budgeting for measuring and limiting task execution times, and virtual memory management, including the ability to disconnect virtual memory for specific real-time tasks.

Since the POSIX standard is so large, subsets are defined to enable implementations for small systems. The following four real-time profiles are defined by POSIX.13 [POS03]:

- **Minimal Real-Time System profile (PSE51).** This profile is intended for small embedded systems, so most of the complexity of a general purpose operating system is eliminated. The unit of concurrency is the thread (processes are not supported). Input and output is possible through predefined device files, but there is not a complete file system. PSE51 systems can be implemented with a few thousand lines of code, and with memory footprints in the tens of kilobytes range.
- **Real-Time Controller profile (PSE52).** It is similar to the PSE51 profile, with the addition of a file system in which regular files can be created, read, or written. It is intended for systems like a robot controller, which may need support for a simplified file system.
- **Dedicated Real-Time System profile (PSE53).** It is intended for large embedded systems (e.g., avionics) and extends the PSE52 profile with the support for multiple processes that operate with protection boundaries.
- **Multi-purpose Real-Time System profile (PSE54).** It is intended for general-purpose computing systems running applications with real-time and non-real-time requirements. It requires all of the POSIX functionality for general purpose systems and, in addition, most of the real-time services.

In summary, the RT-POSIX standard enables portability of real-time applications and specifies real-time services for the development of fixed-priority real-time systems with high degree of predictability. In the future it is expected that RT-POSIX evolves towards more flexible scheduling schemes.

11.1.2 OSEK

OSEK/VDX is a joint project of many automotive industries that aims at the definition of an industrial standard for an open-ended architecture for distributed control units in vehicles [OSE03]. The term OSEK means "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" (Open systems and the corresponding interfaces for automotive electronics); the term VDX means Vehicle Distributed eXecutive.

The objective of the standard is to describe an environment which supports efficient utilization of resources for automotive control unit application software. This standard can be viewed as a set of API's for real-time operating systems integrated on a network management system (VDX), which describes the characteristics of a distributed environment that can be used for developing automotive applications.

The typical applications considered by the standard are control applications with tight real-time constraints, high criticality, and large production volumes. Therefore, to save on production costs, there is a strong push towards code optimization, by reducing the memory footprint to a minimum and enhancing the OS performance as much as possible. A typical OSEK system has the following characteristics:

- **Scalability**. The operating system is intended to be used on a wide range of hardware platforms (from 8 bit microcontrollers, to more powerful processors). To support such a wide range of systems, the standard defines four conformance classes that have increasing complexity. However, memory protection is not supported at all.
- **Software portability**. An ISO/ANSI-C interface between the application and the operating system simplifies portability of the application software. However, due to the wide variety of hardware where the kernel has to work in, the standard does not specify any interface to the I/O subsystem. This reduces portability of the application source code, since the I/O system strongly impacts on the software architecture. Extensions to support memory protection, I/O interfacing and component-based design are addressed by some projects (www.automotive-his.de) and working groups (www.autosar.org).

- **Configurability**. Appropriate configuration tools proposed by the OSEK standard help the designer in tuning the system services and the system footprint. Moreover, a language called OIL (OSEK Implementation Language) is proposed to help the definition of standardized configuration information.
- **Static allocation of software components**. All the kernel objects and the application components are statically allocated. The number of tasks, their code, the required resources and services are defined at compile time. This approach simplifies the internal structure of the kernel and makes it easier to deploy the kernel and the application code on a ROM.
- **Support for Time Triggered Architectures**. The OSEK Standard provides the specification of OSEKTime OS, a time triggered operating system that can be fully integrated in the OSEK/VDX framework.

The OSEK standard comprises also an agreement on interfaces and protocols for in-vehicle communication, called OSEK COM. The basic idea is to simplify porting of applications by providing a standardized API for software communication (between nodes and within a node) that is independent from the particular communication media.

The OSEK standard also covers a standardization of basic and non-competitive infrastructure between the various embedded systems that can be present in a vehicle. In fact, very often electronic control units made by different manufacturers are networked within vehicles by serial data communication links. For that reason, the standard proposes a Network Management system (OSEK NM) that provides standardized features which ensure the functionality of inter-networking by standardized interfaces. The essential task of NM is to ensure the safety and the reliability of a communication network. This is obtained implementing access restrictions to each node, keeping the whole network tolerant to faults, and implementing diagnostic features capable of monitoring the status of the network.

11.1.3 APEX (ARINC)

APEX is a standard for an operating system interface for avionics systems. Its goal is to allow analyzable safety critical real-time applications to be implemented, certified and executed. Several critical real-time systems have been successfully built and certified using APEX, including some critical components for the Boeing 777 aircraft.

Traditionally, avionics computer systems have followed a federated approach, where separate functions are allocated to dedicated (often physically disjoint) computing

"black-boxes". In recent years, there has been a considerable effort by ARINC to define standards for Integrated Modular Avionics (IMA) [ARI91] that allow saving physical resources. IMA defines a standard operating system interface for distributed multiprocessor applications with shared memory and network communications, called the Avionics Application Software Standard Interface [ARI96]. The standard provides some indication about the kernel services expressed as pseudo-code.

Physical memory is subdivided into partitions, and software sub-systems occupy distinct partitions at run-time. An off-line cyclic schedule is used to schedule partitions. Each partition is temporally isolated from the others and cannot consume more processing time than that allocated to it in the cyclic schedule. Each partition contains one or more application processes, having attributes such as period, time capacity, priority, and running state. Processes within a partition are scheduled on a fixed priority basis. Under APEX, a missed deadline is detected when a re-scheduling operation occurs, thus deadlines expiring outside the partition time-slice are only recognized at the start of the next time-slice for that partition.

Communication between processes in different partitions occurs via message passing over logical ports and physical channels. Currently, APEX restricts such messages to be from a single sender to a single receiver. Physical channels are established at initialization time, and many ports maybe mapped to a single channel. Two types of messages are supported: *Sampling Messages*, with overriding and non consumable semantics, and *Queuing Messages*, where messages are enqueued in a FIFO order and read operation is destructive. Using sampling messages is always non blocking, whereas using queuing messages a sender blocks when the buffer is full and a receiver blocks when the buffer is empty. Processes within a partition can communicate using a variety of facilities, including conventional buffers, semaphores and events, but none of these mechanisms are visible outside the partition.

11.1.4 MICRO-ITRON

The ITRON (Industrial TRON - The Real-time Operating system Nucleus) project started in 1984, in Japan. ITRON is an architecture for real-time operating systems used to build embedded systems. The ITRON project has developed a series of de-facto standards for real-time kernels, the previous of which was the Micro-ITRON 3.0 specification [Sak98], released in 1993. It included connection functions that allow a single embedded system to be implemented over a network. There are approximately 50 ITRON real-time kernel products for 35 processors registered with the TRON association, almost exclusively in Japan.

The ITRON standards primarily aim at small systems (8-16 and 32 bits). ITRON specification kernels have been applied over a large range of embedded application domains: audio/visual equipment (TVs, VCRs, digital cameras, STBs, audio components), home appliances (microwave ovens, rice cookers, air-conditioners, washing machines), personal information appliances (PDAs, personal organizers, car navigation systems), entertainment (games, electronic musical instruments), PC peripherals (printers, scanners, disk drives, CD-ROM drives), office equipment (copiers, FAX machines, word processors), communication equipment (phone answering machines, ISDN telephones, cellular phones, PCS terminals, ATM switches, broadcasting equipment, wireless systems, satellites), transportation (automobiles), industrial control (plant control, industrial robots) and others (elevators, vending machines, medical equipment, data terminals).

The Micro-ITRON 4.0 specification [Tak02] combines the loose standardization that is typical for ITRON standards with a Standard Profile that supports the strict standardization needed for portability. In defining the Standard Profile, an effort has been made to maximize software portability while maintaining scalability. As an example, a mechanism has been introduced for improving the portability of interrupt handlers while keeping overhead small. The Standard Profile assumes the following system image: high-end 16-32 bit processor, kernel size from 10 to 20 KB, whole system linked in one module, kernel object statically generated. There is no protection mechanism. The Standard Profile supports task priorities, semaphores, message queues, and mutual exclusion primitives with priority inheritance and priority ceiling protocols.

11.2 COMMERCIAL REAL-TIME SYSTEMS

At the present time, there are more than a hundred commercial products that can be categorized as real-time operating systems, from very small kernels with a memory footprint of a few kilobytes, to large multipurpose systems for complex real-time applications. Most of them provide support for concurrency through processes and/or threads. Processes usually provide protection through separate address spaces, while threads can cooperate more efficiently by sharing the same address space, but with no protection. Scheduling is typically preemptive, because it leads to smaller latencies and a higher degree of resource utilization, and it is based on fixed priorities. At the moment, there are only a few systems providing deadline-driven priority scheduling. The most advanced kernels implement some form of priority inheritance to prevent priority inversion while accessing mutually exclusive resources. Note that this also requires the use of priority queues instead of regular FIFO queues.

Many commercial operating systems also provide a set of tools for facilitating the development of real-time applications. Besides the general programming tools, such as editors, compilers, and debuggers, there are a number of tools specifically made for real-time systems. Advanced tools include memory analyzers, performance profilers, real-time monitors (to view variables while the program is running), and execution tracers (to monitor and display kernel events in a graphical form). Another useful tool for real-time systems is the schedulability analyzer, which enables designers to verify the feasibility of the task set against various design scenarios. There are also code analyzers to determine worst-case execution times of tasks on specific architectures.

Some major players in this field are VxWorks (Wind River), OSE (OSE Systems), Windows CE (Microsoft), QNX, and Integrity (Green Hills). In this section there is not space for describing them all, so only three major systems are described: VxWorks, QNX, and OSE.

11.2.1 VXWORKS

This real-time operating system is produced by Wind River Systems [VxW95] and it is marketed as the run-time component of the Tornado development platform. The kernel uses priority-based preemptive scheduling as a default algorithm, but round-robin scheduling can be also selected as well. It provides 256 priority levels, and a task can change its priority while executing.

Different mechanisms are supplied for intertask communication, including shared memory, semaphores for basic mutual exclusion and synchronization, message queues and pipes for message passing within a CPU, sockets and remote procedure calls for network-transparent communication, and signals for exception handling. Priority inheritance can be enabled on mutual exclusion semaphores to prevent priority inversion. The kernel can be scaled, so that additional features can be include during development to speed the work (such as the networking facilities), and then excluded to save resources in the final version.

A performance evaluation tool kit is available, which includes an execution timer for measuring the duration of a routine or group of routines, and some utilities to show the CPU utilization percentage by tasks. An integrated simulator, VxSim, simulates a VxWorks target for use as a prototyping and testing environment.

VxWorks 5.x conforms to the real-time POSIX 1003.1b standard. Graphics, multiprocessing support, memory management unit, connectivity, Java support, and file systems are available as separate services. All major CPU platforms for embedded systems are supported. Another version, called VxWorks AE, conforms to POSIX and APEX standards. The key new concept in AE is the "protection domain", which corresponds to the partition in ARINC. All memory-based resources, such as tasks, queues, and semaphores are local to the protected domain, which also provides the basis for automated resource reclamation. An optional Arinc-653 compatible protection domain scheduler (Arinc scheduler for short) extends the protection to the temporal domain. Such a two-level scheduler provides a guaranteed CPU time window for a protection domain in which tasks are able to run with temporal isolation. Priority-based preemptive scheduling is used within a protection domain, not between protection domains. VxWorks 5.x applications can run in an AE protected domain without modifications. VxWorks AE is available for a limited set of CPUs.

11.2.2 QNX NEUTRINO

QNX Neutrino [Hil92] is a real-time operating systems used for mission-critical applications, from medical instruments and Internet routers to telematics devices, process control applications, and air traffic control systems.

The QNX Neutrino microkernel implements only the most fundamental services in the kernel, such as signals, timers, and scheduling. All other components — file systems, drivers, protocol stacks, applications — run outside the kernel, in a memory-protected user space. As a result, a faulty component can be automatically restarted without affecting other components or the kernel.

Some of the real-time features include distributed priority inheritance to eliminate priority inversion and nested interrupts to allow priority driven interrupt handling. All components communicate via message passing, which forms a virtual "software bus" that allows the user to dynamically plug in, or plug out, any component on the fly. It provides support for transparent distributed processing using standard messages to access hardware and software resources on remote nodes.

QNX complies with the POSIX 1003.1-2001 standard, providing realtime extensions and threads. It includes a power management framework to enable control over the power consumption of system components. This allows the application developer to determine power management policies.

Finally, QNX Neutrino also provides advanced graphics features, using layering techniques, to create visual applications for markets such as automotive, medical, industrial automation and interactive gaming. A support for accelerated 3D graphics rendering (based on the Mesa implementation of the OpenGL standard) allows to create sophisticated displays with little impact on CPU performance.

11.2.3 OSE

OSE is a real-time operating system produced by ENEA [OSE04]. It comes in three flavors: OSE, OSEck and Epsilon. OSE is the portable kernel written mostly in C, OSEck is the compact kernel version aimed at digital signal processors (DSPs), and Epsilon is a set of highly optimized assembly kernels. The different kernels implement the OSE API in different levels, from A to D: A is the smallest set of features that is guaranteed to exist on all OSE supported platforms, while D is the full set of features including virtual memory, memory protection and concept of users. OSE operating systems are widely used in the automotive industry and the communications industry.

OSE processes can either be static or dynamic, that is created at compile-time or at run-time. Five different types of processes are supported: interrupt process, timer interrupt process, prioritized process, background process, and phantom process. There are different scheduling principles for different processes: priority-based, cyclic, and round-robin. The interrupt processes and the prioritized processes are scheduled according to their priority, while timer interrupt processes are triggered cyclically. The background processes are scheduled in a round-robin fashion. The phantom processes are not scheduled at all and are used as signal redirectors. The processes can be grouped into blocks, and each block may be treated as a single process; e.g., one can start and stop a whole block at once. Moreover, one can associate each block a memory pool that specifies the amount of memory available for that block. Pools can be grouped into segments that can feature hardware memory protection if available. There is a special pool for the system.

OSE processes use messages (called signals) as their primary communication method. Signals are sent from one process to another and do not use the concept of mailboxes. Each process has a single input queue from which it can read signals. It is possible to use filters to read only specific types of messages. A process may also have a redirection table that forwards certain types of messages to other processes. By the use of "link handlers" signals may be sent between OSE systems over various communication channels (network, serial lines, etc). There is an API functionality to get information about processes on other OSE systems so the right receiver can be determined. Sending

signals to a higher-priority process transfers the execution to the receiver. Sending to lower processes does not. Signals going between processes inside the same memory area do not undergo copying. Only when it is necessary from memory point of view, the signal buffer is copied. Semaphores also exist in more than one flavor, but the use of those is discouraged due to priority inversion problems.

An application can be written across several CPUs by using signal IPC and link handlers. One can mix any kernel type with any other, links are monitored for hardware failures, and alternate routes are automatically attempted to be established upon a link failure. Processes are notified upon link failure events.

Errors in system calls are not indicated by a traditional return code, but as a call to an error handler. The error handlers exist on several levels: process, block, and system level. If an error handler on one level cannot handle the error, it is propagated to the next level, until it reaches the system level.

11.3 LINUX RELATED REAL-TIME KERNELS

The main drawback of Linux as a real-time system is that it uses a monolithic kernel where most of the parts are non preemptive. As a result, the latency experienced by real-time activities can be as large as hundreds of milliseconds. This makes common Linux distributions not suitable for hard real-time applications with tight timing constraints. On the other hand, making Linux a real-time operating system would allow using the full-power of a real operating system for real-time applications, including a broad range of open source drivers and development tools.

For this reason, a considerable amount of work has been done during the last years for providing Linux with real-time features. Two different approaches have been followed. The first approach is to use a small real-time executive as a base, with full control of interrupts and processor key features, which executes Linux as a thread. To achieve real-time behavior, the whole Linux kernel is treated by the real-time scheduler as the idle task, so Linux only executes when there are no real time tasks to run, and the real time kernel is inactive. The Linux task can never block interrupts or prevent itself from being preempted. The second approach is to directly modify the Linux internals to insert real-time features. RTLinux and RTAI are examples of the first approach, whereas Linux/RK is an example of the second approach.

11.3.1 RTLINUX

RTLinux was developed by Victor Yodaiken at the University of New Mexico and it is distributed by Finite State Machine Labs, Inc. (http://www.fsmlabs.com). It works as a small executive with a real-time scheduler that runs Linux as its lowest priority thread. The Linux thread is made completely preemptable so that realtime threads and interrupt handlers are never delayed by non-real-time operations. Real-time threads in RT-Linux can communicate with Linux processes via shared memory or a file-like interface.

The executive modifies the standard Linux interrupt handler routine and the interrupt enabling and disabling macros. When an interrupt is raised, the micro-kernel interrupt routine is executed. If the interrupt is related to a real-time activity, a real-time thread is notified and the micro-kernel executes its own scheduler. If the interrupt is not related to a real-time activity, then it is "flagged". When no real-time thread is active, Linux is resumed and executes its own code. Moreover, any pending interrupt related to Linux is served.

In this way, Linux is executed as a background activity in the real-time executive. This approach has the advantage of separating as much as possible the interactions between the Linux kernel, which is very complex and difficult to modify, and the real-time executive. It also allows obtaining a very low latency for real-time activities and, at the same time, the full power of Linux on the same machine. However, there are several drawbacks. Real-Time tasks execute in the same address space as the Linux kernel; therefore, a fault in a user task may crash the kernel. When working with the real-time threads it is not possible to use the standard Linux device driver mechanism; as a result it is often necessary to re-write the device drivers for the real-time application. For example, for using the network in real-time, it is necessary to use another device driver expressly designed for RTLinux. The real-time scheduler is a simple fixed priority scheduler, which is POSIX compliant. There is no direct support for resource management facilities. Some Linux drivers directly disable interrupts during some portions of their execution. During this time, no real-time activities can be executed, and thus the latency is increased.

11.3.2 RTAI

RTAI (Real Time Application Interface) is a modification of the Linux kernel made by Paolo Mantegazza, from the Dipartimento di Ingegneria Aerospaziale at Politecnico di Milano (http://www.aero.polimi.it/ rtai/). RTAI is a living open-source project that builds on the original idea of RTLinux, but has been considerably enhanced.

To obtain a real-time behavior, the Linux kernel was modified in the interrupt handling and scheduling policies. In this way, it is possible to have a real-time platform with low latency and high predicatbility requirements, within a full non real-time Linux environment (access to TCP/IP, graphical display and windowing systems, file and data base systems, etc.).

RTAI offers the same services of the Linux kernel core, adding the features of an industrial real-time operating system. RTAI basically consists of an interrupt dispatcher that traps the peripherals interrupts and, if necessary, reroutes them to Linux. It is not an intrusive modification of the kernel; it uses the concept of hardware abstraction layer (HAL) to get information from Linux and trap some fundamental functions.

RTAI allows to uniformly mix hard and soft real time activities by symmetrically integrating the scheduling of RTAI proper kernel tasks, Linux kernel threads and user space processes/tasks. By using Linux schedulable objects, RTAI benefits from threads protection at the price of a slight increase in latencies. RTAI offers also a native, dynamically extensible, light middleware layer based on the remote procedure call concept, that allows using all of its APIs in a distributed way. Current releases also run on top of the Adeos nano-kernel, which makes it easier to plug in additional features, such as debuggers, analyzers, and standard open middleware layers, serving all operating systems running on top of it almost without any intrusion.

11.3.3 LINUX/RK

In Linux/RK, the Linux kernel has been directly modified [Raj98, Raj00] to introduce real-time features. The kernel is supported by TimeSys Inc. RK stands for "Resource Kernel", because the kernel provides resource reservations directly to user processes. The use of this mechanism is transparent, thus it is possible to assign a reservation to a legacy Linux application. Moreover, it is possible to access a specific API to take advantage of the reservations and of the quality of service management.

A reserve represents a share of a single computing resource, which can be CPU time, physical memory pages, a network bandwidth, or a disk bandwidth. The kernel keeps track of the use of a reserve and enforces its utilization, when necessary. A reserve can be time multiplexed or dedicated. Temporal resources like CPU cycles, network bandwidth and disk bandwidth are time multiplexed, whereas spatial resources, like memory pages, are dedicated. A time multiplexed resource is characterized by three parameters: C, D, T, where T represents a recurrence period, C represents the processing time required within T, and D is the deadline within which the C units of processing time must be available within T.

Within Linux/RK an application can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is available to the application. Such a guarantee of resource allocation gives an application the knowledge of the amount of its currently available resources. A QoS manager or an application itself can then optimize the system behavior by computing the best QoS obtained from the available resources.

To simplify portability, the modifications to the original Linux kernel were limited as much as possible, and the RK layer has been developed as an independent module with several callback hooks.

11.3.4 OTHER APPROACHES

Two other approaches have been used to reduce the size of Linux non-preemptable sections: the one used by the Low-Latency patches (by Ingo Molnar and Andrew Morton), and the one used by the kernel preemptability patch (by MontaVista and TimeSys).

The Low-Latency Linux "corrects" the monolithic structure of the kernel by inserting explicit rescheduling points inside the kernel code. In this way, the size of non preemptable sections is reduced, thus decreasing the latency. The consistency of kernel data is enforced by using cooperative scheduling (instead of non-preemptive scheduling) when the execution flow enters the kernel. Using this method, the maximum latency decreases to the maximum time between two rescheduling points.

The preemptable approach (used in most real-time systems) removes the constraint of a single execution flow inside the kernel. To support full kernel preemptability, however, kernel data structures must be explicitly protected against simultaneous accesses. The Linux preemptable kernel patch, sponsored by MontaVista, uses spinlocks to protect kernel data, whereas the approach followed by TimeSys uses mutexes with the priority inheritance protocol. While the MontaVista patch disables preemption when a spinlock is acquired, the TimeSys one is based on blocking synchronization. In a preemptable kernel, the maximum latency is determined by the maximum amount of time for which a spinlock is held inside the kernel.

11.4 RESEARCH KERNELS

The main characteristics that distinguish this new generation of operating systems include

- The ability to treat tasks with explicit timing constraints, such periods and deadlines;
- The presence of guarantee mechanisms that allow to verify in advance whether the application constraints can be met during execution;
- The possibility to characterize tasks with additional parameters, which are used to analyze the dynamic performance of the system;
- The use of specific resource access protocols that avoid priority inversion and limit the blocking time on mutually exclusive resources.

Expressive examples of operating systems that have been developed according to these principles are CHAOS [SGB87], MARS [KDK+89], Spring [SR91], ARTS [TM89], RK [LKP88], TIMIX [LK88], MARUTI [LTCA89], HARTOS [KKS89], YARTOS [JSP92], HARTIK [But93], MARTE [RH01], SHARK [GAGB01], and ERIKA [GLA+00]. Most of these kernels do not represent a commercial product but are the result of considerable efforts carried out in universities and research centers.

The main differences among the kernels mentioned above concern the supporting architecture on which they have been developed, the static or dynamic approach adopted for scheduling shared resources, the types of tasks handled by the kernel, the scheduling algorithm, the type of analysis performed for verifying the schedulability of tasks, and the presence of fault-tolerance techniques.

In the rest of this chapter, only a few systems are illustrated just to provide a more complete view of the techniques and methodologies that can be adopted to develop a new generation of real-time operating systems with highly predictable behavior.

11.4.1 SHARK

SHARK (Soft and HArd Real-time Kernel) is a dynamic configurable real-time operating system developed at the Scuola Superiore S. Anna of Pisa [GAGB01] to support the development and testing of new scheduling algorithms, aperiodic servers, and resource management protocols. The kernel is designed to be modular, so that applications can

be developed independently of a particular system configuration, and schedulers can be implemented independently of other kernel mechanisms.

The kernel is compliant with the POSIX 1003.13 PSE52 specifications and currently runs on Intel x86 architectures. SHARK is currently used for education in several real-time systems courses all over the world. It is available for free under GPL license at `http://shark.sssup.it`.

KERNEL ARCHITECTURE

To make scheduling algorithms independent of applications and other internal mechanisms, SHARK is based on the notion of a *Generic Kernel*, which does not implement any particular scheduling algorithm, but postpones scheduling decisions to external entities, the *scheduling modules*. In a similar fashion, the access to shared resources is coordinated by *resource modules*. A simplified scheme of the kernel architecture is depicted in Figure 11.1.

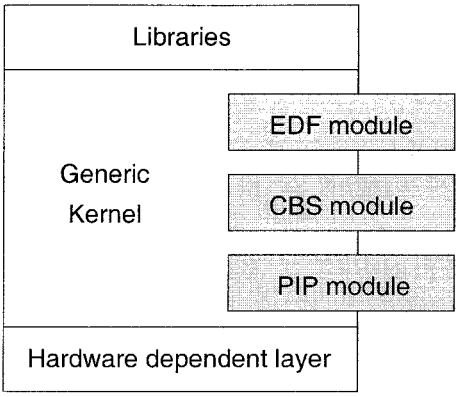

Figure 11.1 The SHARK Architecture.

The Generic Kernel provides the mechanisms used by the modules to perform scheduling and resource management, thus allowing the system to abstract from the algorithms that can be implemented. The Generic Kernel simply provides the primitives without specifying any algorithm, whose implementation resides in external modules, configured at run-time with the support of the *Model Mapper*.

Each module consists of a set of data and functions used for implementing a specific algorithm, whose implementation is independent from the other modules in the system. In this way, many different module configurations are possible. For example, a

Polling Server can either work with Rate Monotonic or EDF without any modification. Moreover, scheduling modules can be composed into priority layers, as in a multi-level scheduling approach. Several scheduling modules are already available, such as Rate Monotonic, EDF, Polling Server, Deferrable Server, Sporadic Server, Slot Shifting, Total Bandwidth Server, and Constant Bandwidth Server, as well as a number of resource protocols, such as Priority Inheritance, Priority Ceiling, and Stack Resource Policy.

Another important component of the Generic Kernel is the Job Execution Time (JET) estimator, which monitors the computation time actually consumed by each job. This mechanism is used for statistical measurements, resource accounting, and temporal protection.

The API is exported through a set of *Libraries*, which use the Generic Kernel to support some common hardware devices (i.e., keyboard, sound cards, network cards, and graphic cards). They provide a compatibility layer with the POSIX interface to simplify porting of applications developed for other POSIX compliant kernels.

Independence between applications and scheduling algorithms is achieved by introducing the notion of *task model*. Two kinds of models are provided: Task Models and Resource Models. A task model expresses the QoS requirements of a task for the CPU scheduling. A resource model is used to define the QoS parameters relative to a set of shared resources used by a task. For example, the resource model can be used to specify the semaphore protocol to be used for protecting critical sections (e.g., Priority Inheritance, Priority Ceiling, or SRP). Requirements are specified through a set of parameters that depend on the specific models. Models are used by the generic kernel to assign a task to a specific module.

Task creation works as follows (see Figure 11.2): when an application issues a request to the kernel for creating a new task, it also sends the model describing the requested QoS. A kernel component, namely the *model mapper*, passes the model to a module, selected according to an internal policy, and the module checks whether it can provide the requested QoS; if the selected module cannot serve the task, the model mapper selects a different module. When a module accepts to manage the task described by the specified model, it converts the model's QOS parameters into the appropriate scheduling parameters. Such a conversion is performed by a module component, called the *QoS Mapper*.

SCHEDULING MODULES

Scheduling Modules are used by the Generic Kernel to schedule tasks, or serve aperiodic requests using an aperiodic server. In general, the implementation of a scheduling

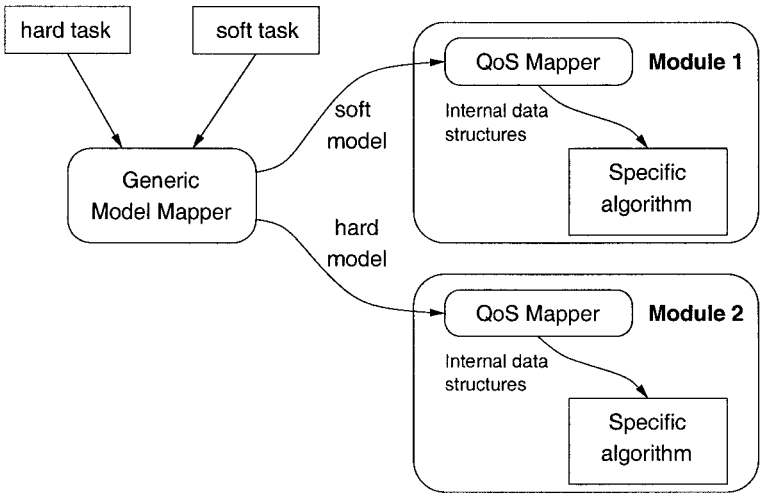

Figure 11.2 The interaction between the Model Mapper and the QOS Mapper.

algorithm should possibly be independent of resource access protocols, and handle only the scheduling behavior. Nevertheless, the implementation of an aperiodic server relies on the presence of another scheduling module, called the Host Module (for example, a Deferrable Server can be used if the base scheduling algorithm is RM or EDF, but not Round Robin). Such a design choice reflects the traditional approach followed in the literature, where most aperiodic servers insert their tasks directly into the scheduling queues of the base scheduling algorithm. Again, the modularity of the architecture hides this mechanism with the task models: an aperiodic server must use a task model to insert his tasks into the Host Module. In this way, the Guest Module does not rely on the implementation of the Host Module.

The Model Mapper distributes the tasks to the registered modules according to the task models the set of modules can handle. For this reason, the task descriptor includes an additional field (`task_level`), which points to the module that is handling the task.

When the Generic Kernel has to perform a scheduling decision, it asks the modules for the task to schedule, according to fixed priorities: first, it invokes a scheduling decision to the highest priority module, then (if the module does not manage any task ready to run), it asks the next high priority module, and so on. In this way, each module manages its private ready task list, and the Generic Kernel schedules the first task of the highest priority non empty module's queue.

SHARED RESOURCE ACCESS PROTOCOLS

As for scheduling, SHARK achieves modularity also in the implementation of shared resource access protocols. Resource modules are used to make resource protocols independent of the scheduling policy and the others resource protocols. Each resource module implements a specific resource access protocol and exports a common interface, similar to the one provided by POSIX for mutexes. A task may also require to use a specified protocol through a resource model.

To make the protocol independent of a particular scheduling algorithm, SHARK supports a generic priority inheritance mechanism independent from the scheduling modules. Such a mechanism is based on the concept of *shadow tasks*. A shadow task is a task that is scheduled in place on another task chosen by the scheduler. When a task is blocked by the protocol, it is kept in the ready queue, and a shadow task is bound to it; when the blocked task becomes the first task in the ready queue, its bound shadow task is scheduled instead. In this way, the shadow task "inherits" the priority of the blocked task.

To implement this solution, a new field `shadow` is added to the generic part of the task control block. This field points to the shadow task. Initially, the shadow field is equal to the task ID (no substitution). When the task blocks, the shadow field is set to the task ID of the blocking task, or to the task that must inherit the blocked task priority. In general, a graph can grow from a blocking task (see Figure 11.3). In this way, when the blocked task is scheduled, the blocking (shadow) task is scheduled, thus allowing the schedulers to abstract from the resource protocols. This approach has also the benefit of allowing a deadlock detection strategy, by simply searching for cycles in the shadow graph.

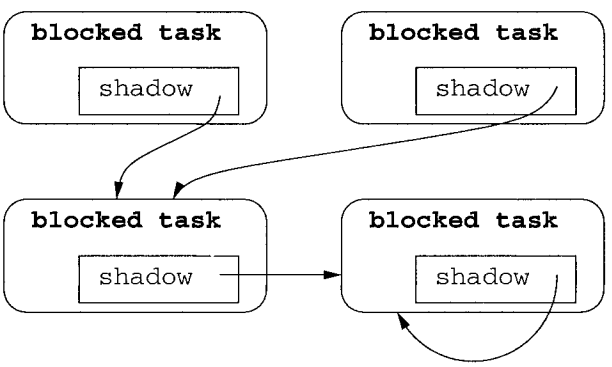

Figure 11.3 The shadow task mechanism.

DEVICE MANAGEMENT

In SHARK, device management is performed outside the kernel, so that the device manager will not steal execution time to the application code. To allow precise resource accounting, SHARK distinguishes between *device drivers* and *device managers*. Device drivers are the hardware dependent part of the device management code, implementing the routines necessary to perform low-level accesses to the devices. Drivers can easily be inherited from other free operating systems (e.g., Linux) that support most of the current PC hardware. The driver code is compiled using some glue code, which remaps the other system calls to the Generic Kernel interface.

The device manager is hardware independent and only performs device scheduling, taking device management costs into account to provide some form of guarantee on hardware accesses. For this purpose, the manager can be implemented as a dedicated thread, or as an application code. The thread implementation allows device management running with temporal protection, whereas the other solution allows a better precision in accounting the CPU time used by the device manager to the application using the hardware resource.

11.4.2 MARTE

MARTE is a "Minimal Real-Time Operating System for Embedded Applications" that follows the Minimal Real-Time POSIX.13 subset. Most of its code is written in Ada with some C and assembler parts. It is developed at the University of Cantabria and is available under the GNU General Public License (http://marte.unican.es).

MARTE allows software cross-development of Ada and C applications using the GNU compilers Gnat and Gcc. The cross environment is formed by a PC running Linux as "Host" and a bare PC as "Target" (386 or above), with both systems connected by an Ethernet LAN (for application booting) and a serial line (for remote debugging). Once the application development is finished, the target can run as an embedded stand-alone system booting from a flash memory or some other non-volatile media.

The kernel has a low-level abstract interface for accessing the hardware that encapsulates operations for interrupt management, clock and timer management, and thread context switches. Its objective is to facilitate migration from one platform to another, being the implementation of this hardware abstraction layer the only thing that needs to be modified in that process. This migration has been already tested for a 683XX microcontroller, and for running MaRTE on top of Linux, by replacing the hardware operations with OS services that emulate them.

Peculiar features of the kernel include application-level scheduling, interrupts management, and the drivers framework.

APPLICATION-DEFINED SCHEDULING

A particular characteristic of the MARTE operating system is an application program interfaces (API) that enables Ada and C applications to use application-defined scheduling algorithms in a way compatible with the scheduling algorithms defined in POSIX and in the Ada 95 Real-Time Systems Annex. Several application-defined schedulers, implemented as special user tasks, can coexist in the system in a predictable way. Each application scheduler is responsible for scheduling a set of tasks that have previously been attached to it.

The scheduler is typically a task that executes a loop where it waits for scheduling events, and then determines the application task(s) to be activated or suspended. A scheduling event is generated every time a task requests attachment to the scheduler or terminates, gets ready or blocks, invokes a yield operation, changes its scheduling parameters, inherits or uninherits a priority, or executes any operation on an application-scheduled mutex.

The application scheduler can make use of the regular operating system services, including the high-resolution timers to program future events, and the execution-time clocks and timers to impose execution-time budgets on the different threads and to simplify the implementation of scheduling algorithms such as the Sporadic Server or the Constant Bandwidth Server.

Because mutexes may cause priority inversions, it is necessary that the scheduler task knows about the use of mutexes to establish its own protocols. Two kinds of mutexes are considered:

- System-scheduled mutexes, created with the current POSIX protocols and scheduled by the system.
- Application-scheduled mutexes, whose protocol will be defined by the application scheduler.

INTERRUPTS MANAGEMENT AT APPLICATION LEVEL

MARTE offers to Ada and C programmers an application program interface (API) that allows dealing with hardware interrupts in an easy way. Basically this API offers operations to:

- Enable and disable hardware interrupts.
- Install interrupt handler procedures (several can be installed for the same interrupt number).
- Synchronize threads with the hardware interrupts.

Semaphores can be used as an alternative synchronization mechanism between interrupt handlers and tasks.

DRIVERS FRAMEWORK

MARTE includes a drivers framework that simplifies and standardizes installation and use of drivers, allowing programmers to share their drivers with other people in a simple way. This framework also facilitates the adaptation of drivers written for open operating systems (such as Linux).

The implemented model is similar to what is used in most UNIX-like operating systems. Applications access devices through "device files" using standard file operations (open, close, write, read, ioctl). In this case the device files are stored as a set of data tables inside the kernel, with pointers to the driver primitive operations.

11.4.3 ERIKA

The last example presented in this book refers to a kernel designed to support embedded real-time applications running on small microcontrollers with limited memory and computational power, but still characterized by a real-time behavior.

ERIKA (Embedded Real tIme Kernel Architecture) is a small size, but fully functional kernel distributed by Evidence s.r.l., under the GNU GPL license. It comes in two versions: ERIKA Enterprise, for supporting new hardware platforms in automotive applications, and ERIKA Educational, for teaching and didactical purposes (`http://erika.sssup.it`).

ERIKA Educational has been designed to be an effective and attractive educational platform for real-time programming in embedded systems courses. A first possibility for students and instructors is to use ERIKA Educational as a programming environment to develop small control applications. To this purpose, the kernel fully supports the widespread Lego Mindstorms platform.

Another possibility is to use the package as a representative example of a real-time operating system, encompassing state of the art design solutions. The kernel proposes an efficient implementation of priority schedulers, clocks and IPC primitives, comparable to the OSEK BCC1 conformance class. A typical kernel configuration consumes less than 2Kb of memory, that fits perfectly on the tiny memory available on the Lego RXC.

Depending on the characteristics of the application, the system designer has the possibility to choose what to optimize. The kernel architecture consists of two main layers: the kernel layer and the hardware abstraction layer.

The kernel layer contains a set of modules that implement task management and real-time scheduling policies. At the moment, the available policies are: Fixed priority with preemption threshold and an EDF with preemption threshold kernel layer (the latter only available under ERIKA Enterprise). Both use the Stack Resource Policy as a protocol for accessing shared resources among threads and sharing the system stack among all the threads while preserving time predictability.

The Hardware Abstraction Layer (HAL) contains the hardware dependent code that manages context switches an interrupt handling. Two hardware abstraction layers are provided to supports two target architectures, one suitable for mono-stack models, and the other one for multi-stack models.