11

Real-Time Embedded Operating Systems: Standards and Perspectives

11.1 Introduction .............................................. 11-1

11.2 Operating System Architecture and Functions ........ 11-3
   Overall System Architecture  •  Process and Thread Model  •  
   Processor Scheduling  •  Interprocess Synchronization and 
   Communication  •  Network Support  •  Additional Functions

11.3 The POSIX Standard .................................... 11-9
   Attribute Objects  •  Multithreading  •  Process and Thread 
   Scheduling  •  Real-Time Signals and Asynchronous Events  •  
   Interprocess Synchronization and Communication  •  
   Thread-Specific Data  •  Memory Management  •  
   Asynchronous and List Directed Input and Output  •  
   Clocks and Timers  •  Cancellation

11.4 Real-Time, Open-Source Operating Systems .......... 11-23

11.5 Virtual-Machine Operating Systems ................... 11-25

References ................................................................ 11-33

11.1 Introduction

Informally speaking, a real-time computer system is a system where a computer senses events from the outside world and reacts to them; in such an environment, the timely availability of computation results is as important as their correctness.

Instead, the exact definition of embedded system is somewhat less clear. In general, an embedded system is a special-purpose computer system built into a larger device and is usually not programmable by the end user.
The major areas of difference between general purpose and embedded computer systems are cost, performance, and power consumption: often, embedded systems are mass produced, thus reducing their unit cost is an important design goal; in addition, mobile battery-powered embedded systems, such as cellular phones, have severe power budget constraints to enhance their battery life.

Both these constraints have a profound impact on system performance, because they entail simplifying the overall hardware architecture, reducing clock speed, and keeping memory requirements to a minimum.

Moreover, embedded computer systems often lack traditional peripheral devices, such as a disk drive, and interface with application-specific hardware instead.

Another common requirement for embedded computer systems is some kind of real-time behavior; the strictness of this requirement varies with the application, but it is so common that, for example, many operating system vendors often use the two terms interchangeably, and refer to their products either as “embedded operating systems” or “real-time operating systems for embedded applications.”

In general, the term “embedded” is preferred when referring to smaller, uniprocessor computer systems, and “real-time” is generally used when referring to larger appliances, but the today’s rapid increase of available computing power and hardware features in embedded systems contributes to shade this distinction.

Recent examples of real-time systems include many kinds of widespread computer systems, from large appliances like phone switches to mass-market consumer products such as printers and digital cameras.

Therefore, a real-time operating system must not only manage system resources and offer a well-defined set of services to application programs, like any other operating system does, but must also provide guarantees about the timeliness of such services and honor them, that is, its behavior must be predictable. Thus, for example, the maximum time the operating system will take to perform any service it offers must be known in advance.

This proves to be a tight constraint, and implies that real-time does not have the same meaning as “real fast,” because it often conflicts with other operating system’s goals, such as good resource utilization and coexistence of real-time and nonreal-time jobs, and adds further complexity to the operating system duty.

Also, it is highly desirable that a real-time operating system optimize some operating system parameters, mainly context switch time and interrupt latency, which have a profound influence on the overall response time of the system to external events; moreover, in embedded systems, the operating system footprint, that is, its memory requirements, must be kept to a minimum to reduce costs.

Last, but not the least, due to the increasing importance of open system architectures in software design, the operating system services should be made available to the real-time application through a standard application programming interface. This approach promotes code reuse, interoperability, and portability, and reduces the software maintenance cost.

The chapter is organized as follows: Section 11.2 gives a brief refresher on the main design and architectural issues of operating systems and on how these concepts have been put in practice. Section 11.3 discusses the main set of international standards concerning real-time operating systems and their application programming interface; this section also includes some notes on mechanisms seldom mentioned in operating system theory but of considerable practical relevance, namely real-time signals, asynchronous I/O operations, and timers.

Next, Section 11.4 gives a short description of some widespread open-source real-time operating systems, another recent and promising source of novelty in embedded system software design. In particular, since open-source operating systems have no purchasing cost and are inherently royalty free, their adoption can easily cut down the cost of an application.

At the end of the chapter, Section 11.5 presents an overview of the operating principle and goals of a seldom-mentioned class of operating systems, namely operating systems based on virtual machines. Although they are perceived to be very difficult to implement, these operating systems look very promising for embedded applications, in which distinct sets of applications, each with its own requirements in term of real-time behavior and security, are executed on the same physical processor; hence, they are an active area of research.
11.2 Operating System Architecture and Functions

The main goal of this section is to give a brief overview on the architecture of operating systems of interest to real-time application developers, and on the functions they accomplish on behalf of the applications that run on them. See, for example, References 1 and 2 for more general information, and Reference 3 for an in-depth discussion about the internal architecture of the influential Unix operating system.

11.2.1 Overall System Architecture

An operating system is a very complex piece of software; accordingly, its internal architecture can be built around several different designs. Some designs that have been tried in practice and are in common use are:

Monolithic systems. This is the oldest design, but it is still popular for very small real-time executives intended for deeply embedded applications, and for the real-time portion of more complex systems, due to its simplicity and very low processor and memory overhead.

In monolithic systems, the operating system as a whole runs in privileged mode, and the only internal structure is usually induced by the way operating system services are invoked: applications, running in user mode, request operating system services by executing a special trapping instruction, usually known as the system call instruction. This instruction brings the processor into privileged mode and transfers control to the system call dispatcher of the operating system. The system call dispatcher then determines which service must be carried out, and transfers control to the appropriate service procedure. Service procedures share a set of utility procedures, which implement generally useful functions on their behalf.

Interrupt handling is done directly in the kernel for the most part and interrupt handlers are not full-fledged processes. As a consequence, the interrupt handling overhead is very small because there is no full task switching at interrupt arrival, but the interrupt handling code cannot invoke most system services, notably blocking synchronization primitives. Moreover, the operating system scheduler is disabled while interrupt handling is in progress, and only hardware prioritization of interrupt requests is in effect, hence the interrupt handling code is implicitly executed at a priority higher than the priority of all other tasks in the system.

To further reduce processor overhead on small systems, it is also possible to run the application as a whole in supervisor mode. In this case, the application code can be bound with the operating system at link time and system calls become regular function calls. The interface between application code and operating system becomes much faster, because no user-mode state must be saved on system call invocation and no trap handling is needed. On the other hand, the overall control that the operating system can exercise on bad application behavior is greatly reduced and debugging may become harder.

In this kind of systems, it is usually impossible to upgrade individual software components, for example, an application module, without replacing the executable image as a whole and then rebooting the system. This constraint can be of concern in applications where software complexity demands the frequent replacement of modules, and no system down time is allowed.

Layered systems. A refinement and generalization of the monolithic system design consists of organizing the operating system as a hierarchy of layers at system design time. Each layer is built upon the services offered by the one below it and, in turn, offers a well-defined and usually richer set of services to the layer above it. Operating system interface and interrupt handling are implemented like in monolithic systems; hence, the corresponding overheads are very similar.

Better structure and modularity make maintenance easier, both because the operating system code is easier to read and understand, and because the inner structure of a layer can be changed at will without interfering with other layers, provided the interlayer interface does not change.

Moreover, the modular structure of the operating system enables the fine-grained configuration of its capabilities, to tailor the operating system itself to its target platform and avoid wasting valuable memory space for operating system functions that are never used by the application. As a consequence, it is possible to enrich the operating system with many capabilities, for example, network support, without sacrificing...
its ability to run on very small platforms when these features are not needed. A number of operating systems in use today evolved into this structure, often starting from a monolithic approach, and offer sophisticated build or link-time configuration tools.

**Microkernel systems.** This design moves many operating system functions from the kernel up into operating system server processes running in user mode, leaving a minimal microkernel and reducing to an absolute minimum the amount of privileged operating system code. Applications request operating system services by sending a message to the appropriate operating system server and waiting for a reply.

The main purpose of the microkernel is to handle the communication between applications and servers, to enforce an appropriate security policy on such communication, and to perform some critical operating system functions, such as accessing I/O device registers, that would be difficult, or inefficient, to do from user-mode processes.

This kind of design makes the operating system easier to manage and maintain. Also, the message-passing interface between user processes and operating system components encourages modularity and enforces a clear and well-understood structure on operating system components.

Moreover, the reliability of the operating system is increased: since the operating system servers run in user mode, if one of them fails some operating system functions will no longer be available, but the system will not crash. Moreover, the failed component can be restarted and replaced without shutting down the whole system.

Last, the design is easily extended to *distributed systems*, where operating system functions are split across a set of distinct machines connected by a communication network. This kind of systems are very promising in terms of performance, scalability, and fault tolerance, especially for large and complex real-time applications.

By contrast, making the message-passing communication mechanism efficient can be a critical issue, especially for distributed systems, and the system call invocation mechanism induces more overhead than in monolithic and layered systems.

Interrupt requests are handled by transforming them into messages directed to the appropriate interrupt handling task as soon as possible: the interrupt handler proper runs in interrupt service mode and performs the minimum amount of work strictly required by the hardware, then synthesizes a message and sends it to an interrupt service task. In turn, the interrupt service task concludes interrupt handling running in user mode. Being an ordinary task, the interrupt service task can, at least in principle, invoke the full range of operating system services, including blocking synchronization primitives, and must not concern itself with excessive usage of the interrupt service processor mode. On the other hand, the overhead related to interrupt handling increases, because the activation of the interrupt service task requires a task switch.

**Virtual machines.** The internal architecture of operating systems based on virtual machines revolves around the basic observation that an operating system must perform two essential functions: multiprogramming and system services.

Accordingly, those operating systems fully separate these functions and implement them as two distinct operating system components: a *virtual machine monitor* that runs in privileged mode, implements multiprogramming, and provides many virtual processors identical in all respects to the real processor it runs on, and one or more *guest operating systems* that run on the virtual processors, and implement system services.

Different virtual processors can run different operating systems, and they must not necessarily be aware of being run in a virtual machine. In the oldest approach to virtual machine implementation, guest operating systems are given the illusion of running in privileged mode, but are instead constrained to operate in user mode; in this way, the virtual machine monitor is able to intercept all privileged instructions issued by the guest operating systems, check them against the security policy of the system, and then perform them on behalf of the guest operating system itself.

Interrupt handling is implemented in a similar way: the virtual machine monitor catches all interrupt requests and then redirects them to the appropriate guest operating system handler, reverting to user mode in the process; thus, the virtual machine monitor can intercept all privileged instructions issued by the guest interrupt handler, and again check and perform them as appropriate.
The full separation of roles and the presence of a relatively small, centralized arbiter of all interactions between virtual machines has the advantage of making the enforcement of security policies easier. The isolation of virtual machines from each other also enhances reliability because, even if one virtual machine fails, it does not bring down the system as a whole. In addition, it is possible to run a distinct operating system in each virtual machine thus supporting, for example, the orderly coexistence between a real-time and a general-purpose operating system.

By contrast, the perfect implementation of virtual machines requires hardware assistance, both to make it feasible and to be able to emulate privileged instructions with a reasonable degree of efficiency.

A variant of this design adopts an interpretive approach, and allows the virtual machines to be different from the physical machine. For example, Java programs are compiled into byte-code instructions suitable for execution by an abstract Java virtual machine. On the target platform an interpreter executes the byte code on the physical processor, thus implementing the virtual machine.

More sophisticated approaches to virtual machine implementation are also possible, the most common one being on-the-fly code generation, also known as just-in-time compilation.

11.2.2 Process and Thread Model

A convenient and easy to understand way to design real-time software applications is to organize them as a set of cooperating sequential processes. A process is an activity, namely the activity of executing a program, and encompasses the program being executed, the data it operates on, and the current processor state, including the program counter and registers. In particular, each process has its own address space.

In addition, it is often convenient to support multiple threads of control within the same process, sharing the same address space. Threads can be implemented for the most part in user mode, without the operating system’s kernel intervention; moreover, when the processor is switched between threads, the address space remains the same and must not be switched. Both these facts make processor switching between threads very fast with respect to switching between processes. On the other hand, since all threads within a process share the same address space, there can be only a very limited amount of protection among them with respect to memory access; hence, for example, a thread is allowed to pollute by mistake another thread’s data and the operating system has no way to detect errors of this kind.

As a consequence, many small operating systems for embedded applications only support threads to keep overheads and hardware requirements to a minimum, while larger operating systems for more complex real-time applications offer the user a choice between a single or multiple process model to enhance the reliability of complex systems.

Another important operating system design issue is the choice between static and dynamic creation of processes and threads: some operating systems, usually oriented toward relatively simple embedded applications, only support static tasks, that is, all tasks in the system are known in advance and it is not possible to create and destroy tasks while the system is running; thus, the total number of tasks in the system stays constant for all its life. Other operating systems allow us to create and destroy tasks at runtime, by means of a system call.

Dynamic task creation has the obvious advantage of making the application more flexible, but it increases the complexity of the operating system, because many operating system data structures, first of all the process table, must be allocated dynamically and their exact size cannot be known in advance. In addition, the application code requires a more sophisticated error-handling strategy, with its associated overheads, because it must be prepared to cope with the inability of the operating system to create a new task, due to lack of resources.

11.2.3 Processor Scheduling

The scheduler is one of the most important components of a real-time operating system, as it is responsible for deciding to which runnable threads the available processors must be assigned, and for how long. Among dynamic schedulers — that is, schedulers that perform scheduling computations at runtime — while the
application is running, several algorithms are in use and offer different tradeoffs between real-time predictability, implementation complexity, and overhead.

Since the optimum compromise often depends on the application's characteristics, most real-time operating systems support multiple scheduling policies simultaneously and the responsibility of a correct choice falls on the application programmer. The most common scheduling algorithms supported by real-time operating systems and specified by international standards are:

**First in, first out with priority classes.** Under this algorithm, also known as fixed priority scheduling, there is a list of runnable threads for each priority level. When a processor is idle, the scheduler takes the runnable thread at the head of the highest-priority, nonempty thread list and runs it.

When the scheduler preempts a running thread, because a higher-priority task has become runnable, the preempted thread becomes the head of the thread list for its priority; when a blocked thread becomes runnable again, it becomes the tail of the thread list for its priority.

The first in, first out scheduler never changes thread priorities at runtime; hence, the priority assignment is fully static; a well-known approach to static priority assignment for periodic tasks is the **rate monotonic** policy, in which task priorities are inversely proportional to their periods.

In order to ensure that none of the threads can monopolize the processor, when multiple, runnable threads share the same priority level, the basic algorithm is often enhanced with the additional constraint that, when a running thread has been executing for more than a maximum amount of time, the **quantum**, that thread is forcibly returned to the tail of its thread list and a new thread is selected for execution; this approach is known as **round-robin** scheduling.

**Earliest deadline first.** The earliest deadline first scheduler assigns thread priorities **dynamically**. In particular, this scheduler always executes the thread with the nearest deadline. It can be shown that this algorithm is optimal for uniprocessor systems, and supports full processor utilization in all situations.

However, its performance under overload can be poor and dynamically updating thread priorities on the base of their deadlines may be computationally expensive, especially when this scheduler is layered on a fixed-priority lower-level scheduler.

**Sporadic server.** This scheduling algorithm was first introduced in Reference 4, where a thorough description of the algorithm can be found.

The sporadic server algorithm is suitable for aperiodic event handling where, for timeliness, events must be handled at a certain, usually high, priority level, but lower-priority threads with real-time requirements could suffer from excessive preemption if that priority level were maintained indefinitely. It acts on the base of two main scheduling parameters associated with each thread: the **execution capacity** and the **replenishment period**.

Informally, the execution capacity of a thread represents the maximum amount of processor time that the thread is allowed to consume at high priority in a replenishment period.

The execution capacity of a thread is preserved until an aperiodic request for that task occurs, thus making it runnable; then, thread execution depletes its execution capacity.

The sporadic server algorithm replenishes the thread's execution capacity after some or all of its capacity is consumed by thread execution; the schedule for replenishing the execution capacity is based on the thread’s replenishment period.

Should the thread reach its processor usage upper limit, its execution capacity becomes zero and it is demoted to a lower-priority level, thus avoiding excessive preemption against other threads. When replenishments have restored the execution capacity of the thread above a certain threshold level, the scheduler promotes the thread to its original priority again.

### 11.2.4 Interprocess Synchronization and Communication

An essential function of a multiprogrammed operating system is to allow processes to synchronize and exchange information; as a whole, these functions are known as InterProcess Communication (IPC). Many interprocess synchronization and communication mechanisms have been proposed and were objects of
extensive theoretical study in the scientific literature. Among them, we recall:

**Semaphores.** A semaphore, first introduced by Dijkstra in 1965, is a synchronization device with an integer value, and on which the following two primitive, atomic operations are defined:

- The P operation, often called DOWN or WAIT, checks if the current value of the semaphore is greater than zero. If so, it decrements the value and returns to the caller; otherwise, the invoking process goes into the blocked state until another process performs a V on the same semaphore.
- The V operation, also called UP, POST, or SIGNAL, checks whether there is any process currently blocked on the semaphore. In this case, it wakes exactly one of them up, allowing it to complete its P; otherwise, it increments the value of the semaphore. The V operation never blocks.

Semaphores are a very low-level IPC mechanism; therefore, they have the obvious advantage of being simple to implement, at least on uniprocessor systems, and of having a very low overhead. By contrast, they are difficult to use, especially in complex applications.

Also related to mutual exclusion with semaphores is the problem of priority inversion. Priority inversion occurs when a high-priority process is forced to wait for a lower-priority process to exit a critical region, a situation in contrast with the concept of relative task priorities. Most real-time operating systems take this kind of blocking into account and implement several protocols to bound it; among these we recall the priority inheritance and the priority ceiling protocols.

**Monitors.** To overcome the difficulties of semaphores, in 1974/1975, Hoare and Brinch Hansen introduced a higher-level synchronization mechanism, the monitor.

A monitor is a set of data structures and procedures that operate on them; data structures are shared among all processes that can use the monitor, and it effectively hides the data structures it contains; hence, the only way to access the data associated with the monitor is through the procedures in the monitor itself.

In addition, all procedures in the same monitor are implicitly executed in mutual exclusion. Unlike semaphores, the responsibility of ensuring mutual exclusion falls on the compiler, not on the programmer, because monitors are a programming language construct, and the language compiler knows about them.

Inside a monitor, condition variables can be used to wait for events. Two atomic operations are defined on a condition variable:

- The WAIT operation releases the monitor and blocks the invoking process until another process performs a SIGNAL on the same condition variable.
- The SIGNAL operation unblocks exactly one process waiting on the condition variable. Then, to ensure that mutual exclusion is preserved, the invoking process is either forced to leave the monitor immediately (Brinch Hansen’s approach), or is blocked until the monitor becomes free again (Hoare’s approach).

**Message passing.** Unlike all other IPC mechanisms described so far, message passing supports explicit data transfer between processes; hence, it does not require a shared memory and lends itself well to be extended to distributed systems. This IPC method provides for two primitives:

- The SEND primitive sends a message to a given destination.
- Symmetrically, RECEIVE receives a message from a given source.

Many variants are possible on the exact semantics of these primitives; they mainly differ in the way messages are addressed and buffered.

A commonplace addressing scheme is to give to each process/thread in the system a unique address, and to send messages directly to processes. Otherwise, a message can be addressed to a mailbox, a message container whose maximum capacity is usually specified upon creation; in this case, message source and destination addresses are mailbox addresses.

When mailboxes are used, they also provide some amount of message buffering, that is, they hold messages that have been sent but have not been received yet. Moreover, a single task can own multiple mailboxes and use them to classify messages depending on their source or priority. A somewhat contrary
approach to message buffering, simpler to implement but less flexible, is the rendezvous strategy: the
system performs no buffering. Hence, when using this scheme the sender and the receiver are forced to
run in lockstep because the SEND does not complete until another process executes a matching RECEIVE
and, conversely, the RECEIVE waits until a matching SEND is executed.

11.2.5 Network Support

There are two basic approaches to implement network support in a real-time operating system and to
offer it to applications:

- The POSIX standard (Portable Operating System Interface for Computing Environments) [5]
specifies the socket paradigm for uniform access to any kind of network support that many real-
time operating systems provide them. Sockets, fully described in Reference 3, were first introduced
in the “Berkeley Unix” operating system and are now available on virtually all general-purpose
operating systems; as a consequence, most programmers are likely to be proficient with them.

  The main advantage of sockets is that they support in a uniform way any kind of communication
network, protocol, naming conventions, hardware, and so on. Semantics of communication and
naming are captured by communication domains and socket types, both specified upon socket
creation. For example, communication domains are used to distinguish between IPv4 and X25
network environments, whereas the socket type determines whether communication will be stream-
based or datagram-based and also implicitly selects which network protocol a socket will use.

  Additional socket characteristics can be setup after creation through abstract socket options;
for example, socket options provide a uniform, implementation-independent way to set the amount
of receive buffer space associated with a socket.

- Some operating systems, mostly focused on a specific class of embedded applications, offer network
support through a less general, but more rich and efficient, application programming interface.
For example, Reference 6 is an operating system specification oriented to automotive applica-
tions; it specifies a communication environment (OSEK/VDX COM) less general than sockets and
oriented to real-time message-passing networks, such as the Controller Area Network (CAN).

  In this case, for example, the application programming interface allows applications to easily set
message filters and perform out-of-order receives, thus enhancing their timing behavior; both these
functions are supported with difficulty by sockets, because they do not fit well with the general
socket paradigm.

In both cases, network device drivers are usually supplied by third-party hardware vendors and conform
to a well-defined interface defined by the operating system vendor.

  The network software itself, although it is often bundled with the operating system and provided by the
same vendor, can be obtained from third-party software houses, too. Often, these products are designed
to run on a wide variety of hardware platforms and operating systems, and come with source code; hence,
it is also possible to port them to custom operating systems developed in-house, and they can be extended
and enhanced by the end user.

11.2.6 Additional Functions

Even if real-time operating systems sometimes do not implement several major functions that are now
commonplace in general-purpose operating systems, such as demand paging, swapping, and filesystem
access, they must be concerned with other, less well-known functions that ensure or enhance system
predictability, for example:

- Asynchronous, real-time signals and cancellation requests, to deal with unexpected events, such
as software and hardware failures, and to gracefully degrade the system's performance should a
processor overload occur.
• High-resolution clocks and timers, to give real-time processes an accurate notion of elapsed time.
• Asynchronous I/O operations, to decouple real-time processes from the inherent unpredictability of many I/O devices.

### 11.3 The POSIX Standard

The original version of the Portable Operating System Interface for Computing Environments, better known as “the POSIX standard,” was first published between 1988 and 1990, and defines a standard way for applications to interface with the operating system. The set now includes over 30 individual standards, and covers a wide range of topics, from the definition of basic operating system services, such as process management, to specifications for testing the conformance of an operating system to the standard itself.

Among these, of particular interest is the System Interfaces (XSH) Volume of IEEE Std 1003.1-2001 [5], which defines a standard operating system interface and environment, including real-time extensions. The standard contains definitions for system service functions and subroutines, language-specific system services for the C programming language, and notes on portability, error handling, and error recovery.

This standard has been constantly evolving since it was first published in 1988; the latest developments have been crafted by a joint working group of members of the IEEE Portable Applications Standards Committee, members of The Open Group, and members of ISO/IEC Joint Technical Committee 1. The joint working group is known as the Austin Group, after the location of the inaugural meeting held at the IBM facility in Austin, Texas in September 1998.


For real-time systems, the latest version of IEEE Std 1003.1 [5] incorporates the real-time extensions listed in Table 11.1. Since embedded systems can have strong resource limitations, the IEEE Std 1003.13-1998 [7] profile standard groups functions from the standards mentioned above into units of functionality. Implementations can then choose the profile most suited to their needs and to the computing resources of their target platforms.

Operating systems invariably experience a delay between the adoption of a standard and its implementation, hence functions defined earlier in time are usually supported across a wider number of operating systems. For this reason, in this section we concentrate only on the functions that are both related to real-time software development and are actually available on most real-time operating systems at the date of writing, including multithreading support. In addition, we assume that the set of functions common to both the POSIX and the ISO C [8] standards is well known to readers, hence we will not describe it.

Table 11.2 summarizes the functional groups of IEEE Std 1003.1-2001 that will be discussed next.

#### 11.3.1 Attribute Objects

Attribute objects are a mechanism devised to support future standardization and portable extension of some entities specified by the POSIX standard, such as threads, mutual exclusion devices, and condition variables, without requiring that the functions operating on them be changed.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003.1b</td>
<td>Basic real-time extensions; first published in 1993</td>
</tr>
<tr>
<td>1003.1c</td>
<td>Threads extensions; published in 1995</td>
</tr>
<tr>
<td>1003.1d</td>
<td>Additional real-time extensions; published in 1999</td>
</tr>
<tr>
<td>1003.1j</td>
<td>Advanced real-time extensions; published in 2000</td>
</tr>
</tbody>
</table>
In addition, they provide for a clean isolation of the configurable aspects of said entities. For example, the stack address (i.e., the location in memory of the storage to be used for the thread's stack) is an important attribute of a thread, but it cannot be expressed portably and must be adjusted when the program is ported to a different architecture.

The use of attribute objects allows the programmer to specify thread's attributes in a single place, rather than spread them across every instance of thread creation; moreover, the same set of attributes can be shared for multiple objects of the same kind so as, for example, to set up classes of threads with similar attributes.

Figure 11.1 shows how attribute objects are created, manipulated, used to configure objects when creating them, and finally destroyed. As an example, the function and attribute names given in the figure are those used for threads, but the same general architecture is also used for the attributes of mutual exclusion devices (Section 11.3.5.3) and condition variables (Section 11.3.5.4).

In order to be used, an attribute object must be first initialized by means of the `pthread_attr_init` function; this function also fills the attribute object with a default value for all attributes defined by the

---

**TABLE 11.2 Basic Functional Groups of IEEE Std 1003.1-2001**

<table>
<thead>
<tr>
<th>Functional group</th>
<th>Main functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple threads</td>
<td>pthread_create pthread_exit pthread_join pthread_detach</td>
</tr>
<tr>
<td></td>
<td>pthread_equal pthread_self</td>
</tr>
<tr>
<td>Process and thread scheduling</td>
<td>sched_setscheduler sched_getscheduler sched_setparam</td>
</tr>
<tr>
<td></td>
<td>sched_getparam pthread_setschedparam</td>
</tr>
<tr>
<td></td>
<td>pthread_getsetschedparam pthread_setschedprio</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setschedpolicy pthread_attr_getschedpolicy</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setschedparam pthread_attr_getschedparam</td>
</tr>
<tr>
<td></td>
<td>sched_yield sched_get_priority_max</td>
</tr>
<tr>
<td></td>
<td>sched_get_priority_min sched_rr_get_interval</td>
</tr>
<tr>
<td>Real-time signals</td>
<td>sigqueue pthread_kill sigaction sigaltstack sigemptyset</td>
</tr>
<tr>
<td></td>
<td>sigfillset sigaddset sigdelset sigismember sigwait</td>
</tr>
<tr>
<td></td>
<td>sigwaitinfo sigtimedwait</td>
</tr>
<tr>
<td>Interprocess and communication</td>
<td>mq_open mq_close mq_unlink mq_send mq_receive</td>
</tr>
<tr>
<td></td>
<td>mq_timedsend mq_timedreceive mq_notify mq_getattr</td>
</tr>
<tr>
<td></td>
<td>mq_setattr</td>
</tr>
<tr>
<td></td>
<td>sem_init sem.destroy sem.open sem.close sem.unlink</td>
</tr>
<tr>
<td></td>
<td>sem.wait sem.trywait sem.timedwait sem.post sem.getvalue</td>
</tr>
<tr>
<td></td>
<td>pthread_mutex_destroy pthread_mutex_init</td>
</tr>
<tr>
<td></td>
<td>pthread_mutex_lock pthread_mutex_trylock</td>
</tr>
<tr>
<td></td>
<td>pthread_mutex_timedlock pthread_mutex_unlock</td>
</tr>
<tr>
<td></td>
<td>pthread_cond_init pthread_cond_destroy pthread_cond_wait</td>
</tr>
<tr>
<td></td>
<td>pthread_cond_timedwait pthread_cond_signal</td>
</tr>
<tr>
<td></td>
<td>pthread_cond_broadcast</td>
</tr>
<tr>
<td>Thread-specific data</td>
<td>shm_open close shm_unlink mmap munmap</td>
</tr>
<tr>
<td>Mem. management</td>
<td>pthread_key_create pthread_getspecific</td>
</tr>
<tr>
<td></td>
<td>pthread_setspecific pthread_key_delete</td>
</tr>
<tr>
<td>Asynchronous and list directed I/O</td>
<td>aio_read aio_write lio_listio</td>
</tr>
<tr>
<td></td>
<td>aio_error aio_return aio_fsync aio_suspend aio_cancel</td>
</tr>
<tr>
<td>Clocks and timers</td>
<td>clock_gettime clock_settime clock_getres</td>
</tr>
<tr>
<td></td>
<td>timer_create timer_delete timer_getoverrun</td>
</tr>
<tr>
<td></td>
<td>timer_gettime timer_settime</td>
</tr>
<tr>
<td>Cancellation</td>
<td>pthread_cancel pthread_setcancelstate</td>
</tr>
<tr>
<td></td>
<td>pthread_setcanceltype pthread_testcancel</td>
</tr>
<tr>
<td></td>
<td>pthread_cleanup_push pthread_cleanup_pop</td>
</tr>
</tbody>
</table>
An attribute object holds one or more named attributes; for each attribute, the standard specifies a pair of functions that the user can call to get and set the value of that attribute. For example, the `pthread_attr_getstackaddr` and `pthread_attr_setstackaddr` functions get and set the stack address attribute, respectively.

After setting the individual attributes of an attribute object to the desired value, the attribute object can be used to configure one or more entities specified by the standard. Hence, for example, the thread attribute object being described can be used as an argument to the `pthread_create` function which, in turn, creates a thread with the given set of attributes.

Last, it should be noted that the attribute objects are defined as opaque types, so they shall be accessed only by the functions just presented, and not by manipulating their representation directly, even if it is known, because doing this is not guaranteed to be portable across different implementations of the standard.

### 11.3.2 Multithreading

The multithreading capability specified by the POSIX standard includes functions to populate a process with new threads. In particular, the `pthread_create` function creates a new thread within a process and sets up a thread identifier for it, to be used to operate on the thread in the future.

After creation, the thread immediately starts executing a function passed to `pthread_create` as an argument; moreover, it is also possible to pass an argument to the function in order, for example, to share the same function among multiple threads and nevertheless be able to distinguish them.

The `pthread_create` function also takes an optional reference to an attribute object as argument. The attributes of a thread determine, for example, the size and location of the thread's stack and its scheduling parameters; the latter will be described further in Section 11.3.3.

A thread may terminate in three different ways:

- By returning from its main function
- By explicitly calling the `pthread_exit` function
- By accepting a cancellation request (Section 11.3.10)

In any case, the `pthread_join` function allows the calling thread to synchronize with, that is wait for, the termination of another thread. When the thread finally terminates, this function also returns to the caller a summary information about the reason of the termination. For example, if the target thread terminated itself by means of the `pthread_exit` function, `pthread_join` returns the status code passed to `pthread_exit` in the first place.
If this information is not desired, it is possible to save system resources by detaching a thread, either dynamically by means of the pthread_detach function, or statically by means of a thread’s attribute. In this way, the storage associated to that thread can be immediately reclaimed when the thread terminates.

Additional utility functions are provided to operate on thread identifiers; hence, for example, the pthread_equal function checks whether two thread identifiers are equal (i.e., whether they refer to the same thread), and the pthread_self function returns the identifier of the calling thread.

### 11.3.3 Process and Thread Scheduling

Functions in this group allow the application to select a specific policy that the operating system must follow to schedule a particular process or thread, and to get and set the scheduling parameters associated with that process or thread.

In particular, the sched_setscheduler function sets both the scheduling policy and parameters associated to a process, and sched_getscheduler reads them back for examination. The sched_setparam and sched_getparam functions are somewhat more limited, because they set and get the scheduling parameters but not the policy. All functions take a process identifier as argument, to uniquely identify a process.

For threads, the pthread_setschedparam and pthread_getschedparam functions set and get the scheduling policy and parameters associated with a thread; pthread_setschedprio directly sets the scheduling priority of the given thread. All these functions take a thread identifier as argument and perform a dynamic access to the thread scheduling parameters; in other words, they can be used when the thread already exists in the system.

On the other hand, the functions pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, and pthread_attr_getschedparam store and retrieve the scheduling policy and parameters of a thread into and from an attribute object, respectively; in turn, the attribute object can subsequently be used to create one or more threads. The general mechanism of attribute objects in the POSIX standard has been discussed in Section 11.3.1.

An interesting and useful side effect of pthread_setschedprio is that, when the effect of the function is to lower the priority of the target thread, the thread is inserted at the head of the thread list of the new priority, instead of the tail. Hence, this function provides a way for an application to temporarily raise its priority and then lower it again to its original value, without having the undesired side effect of yielding to other threads of the same priority. This is necessary, for example, if the application is to implement its own strategies for bounding priority inversion.

Last, the sched_yield function allows the invoking thread to voluntarily relinquish the CPU in favor of other threads of the same priority; the invoking thread is linked to the end of the list of ready processes for its priority.

In order to support the orderly coexistence of multiple scheduling policies, the conceptual scheduling model defined by the standard and depicted in Figure 11.2 assigns a global priority to all threads in the system and contains one ordered thread list for each priority; any runnable thread will be on the thread list for that thread’s priority.

When appropriate, the scheduler shall select the thread at the head of the highest-priority, nonempty thread list to become a running thread, regardless of its associated policy; this thread is then removed from its thread list. When a running thread yields the CPU, either voluntarily or by preemption, it is returned to the thread list it belongs to.

The purpose of a scheduling policy is then to determine how the operating system scheduler manages the thread lists, that is, how threads are moved between and within lists when they gain or lose access to the CPU. Associated with each scheduling policy is a priority range, which must span at least 32 distinct priority levels; all threads scheduled according to that policy must lie within that priority range, and priority ranges belonging to different policies can overlap in whole or in part. The sched_get_priority_min and...
Selection of the highest-priority thread by the scheduler.

Threads are returned to their thread list on yield or preemption.

FIGURE 11.2 Processor scheduling in POSIX.

The mapping between the multiple local priority ranges, one for each scheduling policy active in the system, and the single global priority range is usually performed by a simple relocation and is either fixed or programmable at system configuration time, depending on the operating system. In addition, operating systems may reserve some global priority levels, usually the higher ones, for interrupt handling.

The standard defines the following three scheduling policies, whose algorithms have been briefly presented in Section 11.2.3:

- First in, first out (SCHED_FIFO)
- Round robin (SCHED_RR)
- Optionally, a variant of the sporadic server scheduler (SCHED_SPORADIC)

Most operating systems set the execution time limit of the round-robin scheduling policy statically, at system configuration time; the sched_rr_get_interval function returns the execution time limit set for a given process; the standard provides no portable way to set the execution time limit dynamically.

A fourth scheduling policy, SCHED_OTHER, can be selected to denote that a thread no longer needs a specific real-time scheduling policy: general-purpose operating systems with real-time extensions usually revert to the default, non-real-time scheduler when this scheduling policy is selected.

Moreover, each implementation is free to redefine the exact meaning of the SCHED_OTHER policy and can provide additional scheduling policies besides those required by the standard, but any application using them will no longer be fully portable.

sched_get_priority_max functions return the allowable range of priority for a given scheduling policy.

The mapping between the multiple local priority ranges, one for each scheduling policy active in the system, and the single global priority range is usually performed by a simple relocation and is either fixed or programmable at system configuration time, depending on the operating system. In addition, operating systems may reserve some global priority levels, usually the higher ones, for interrupt handling.

The standard defines the following three scheduling policies, whose algorithms have been briefly presented in Section 11.2.3:

- First in, first out (SCHED_FIFO)
- Round robin (SCHED_RR)
- Optionally, a variant of the sporadic server scheduler (SCHED_SPORADIC)

Most operating systems set the execution time limit of the round-robin scheduling policy statically, at system configuration time; the sched_rr_get_interval function returns the execution time limit set for a given process; the standard provides no portable way to set the execution time limit dynamically.

A fourth scheduling policy, SCHED_OTHER, can be selected to denote that a thread no longer needs a specific real-time scheduling policy: general-purpose operating systems with real-time extensions usually revert to the default, non-real-time scheduler when this scheduling policy is selected.

Moreover, each implementation is free to redefine the exact meaning of the SCHED_OTHER policy and can provide additional scheduling policies besides those required by the standard, but any application using them will no longer be fully portable.
11.3.4 Real-Time Signals and Asynchronous Events

Signals are a facility specified by the ISO C standard and widely available on most operating systems; they provide a mechanism to convey information to a process or thread when it is not necessarily waiting for input. The IEEE Std 1003.1-2001 further extends the signal mechanism to make it suitable for real-time handling of exceptional conditions and events that may occur asynchronously with respect to the notified process like, for example:

- An error occurring during the execution of the process, for example, a memory reference through an invalid pointer.
- Various system and hardware failures, such as a power failure.
- Explicit generation of an event by another process; as a special case, a process can also trigger a signal directed to itself.
- Completion of an I/O operation started by the process in the past, and for which the process did not perform an explicit wait.
- Availability of data to be read from a message queue.

The signal mechanism has a significant historical heritage; in particular, it was first designed when multithreading was not yet in widespread use and its interface and semantics underwent many changes since their inception.

Therefore, it owes most of its complexity to the need of maintaining compatibility with the historical implementations of the mechanism made, for example, by the various flavors of the influential Unix operating systems; however, in this section the compatibility interfaces will not be discussed for the sake of clarity and conciseness.

With respect to the ISO C signal behavior, the IEEE Std 1003.1-2001 specifies two main enhancements of interest to real-time programmers:

1. In the ISO C standard, the various kinds of signals are identified by an integer number (often denoted by a symbolic constant in application code) and, when multiple signals of different kind are pending, they are serviced in an unspecified order; the IEEE Std 1003.1-2001 continues to use signal numbers but specifies that for a subset of their allowable range, between \( \text{SIGRTMIN} \) and \( \text{SIGRTMAX} \), a priority hierarchy among signals is in effect, so that the lowest-numbered signal has the highest priority of service.

2. In the ISO C standard, there is no provision for signal queues, hence when multiple signals of the same kind are raised before the target process had a chance of handling them, all signals but the first are lost; the IEEE Std 1003.1-2001 specifies that the system must be able to keep track of multiple signals with the same number by enqueuing and servicing them in order. Moreover, it also adds the capability of conveying a limited amount of information (a union signal, capable of holding either an integer or a pointer) with each signal request, so that multiple signals with the same signal number can be distinguished from each other. The queueing policy is always FIFO, and cannot be changed by the user.

As outlined above, each signal has a signal number associated to it, to identify its kind; for example, the signal associated to memory access violations has the number \( \text{SIGSEGV} \) associated to it.

Figure 11.3 depicts the life of a signal from its generation up to its delivery. Depending on their kind and source, signals may be directed to either a specific thread in the process, or to the process as a whole; in the latter case, every thread belonging to the process is a candidate for the delivery of the signal, by the rules described later.

It should also be noted that for some kinds of events, the POSIX standard specifies that the notification can also be carried out by the execution of a handling function in a separate thread, if the application so chooses; this mechanism is simpler and clearer than the signal-based notification, but requires multithreading support on the system side.
11.3.4.1 Generation of a Signal

As outlined above, most signals are generated by the system rather than by an explicit action performed by a process. For these, the POSIX standard specifies that the decision of whether the signal must be directed to the process as a whole or to a specific thread within a process must be carried out at the time of generation and must represent the source of the signal as closely as possible.

In particular, if a signal is attributable to an action carried out by a specific thread, for example, a memory access violation, the signal shall be directed to that thread and not to the process. If such an attribution is either not possible or not meaningful as it is the case, for example, of the power failure signal, the signal shall be directed to the process.

Besides various error conditions, an important source of signals generated by the system relate to asynchronous event notification and are always directed to the process; as an example, Section 11.3.8 will describe the mechanism behind the notification of completion for asynchronous I/O operations.

On the other hand, processes have the ability of synthesizing signals by means of two main interfaces, depending on the target of the signal:

- The `sigqueue` function, given a process identifier and a signal number, generates a signal directed to that process; an additional argument, a `union sigval`, allows the caller to associate a limited amount of information to the signal, provided that the SA_SIGINFO flag is set for that signal number. Additional interfaces exist to generate a signal directed to a group of processes, for example, the `killpg` function. However, they have not been extended for real time and hence they do not have the ability of associating any additional information to the signal.
- The `pthread_kill` function generates a signal directed to a specific thread within the calling process and identified by its thread identifier. It is not possible to generate a signal directed to a specific thread of another process.
11.3.4.2 Process-Level Action

For each kind of signal defined in the system, that is, for each valid signal number, processes may set up an action by means of the `sigaction` function; the action may consist of either:

- Ignore the signal completely
- A default action performed by the operating system on behalf of the process, and possibly with process-level side effects, such as the termination of the process itself
- The execution of a signal handling function specified by the programmer

In addition, the same function allows the caller to set zero or more flags associated to the signal number. Of the rather large set of flags specified by the POSIX standard, the following ones are of particular interest to real-time programmers:

- The `SA_SIGINFO` flag, when set, enables the association of a limited amount of information to each signal; this information will then be conveyed to the signaled process or thread. In addition, if the action associated with the signal is the execution of a user-specified signal handler, setting this flag extends the arguments passed to the signal handler to include additional information about the reason why the signal was generated and about the receiving thread's context that was interrupted when the signal was delivered.
- The `SA_RESTART` flag, when set, enables the automatic, transparent restart of interruptible system calls when the system call is interrupted by the signal. If this flag is clear, system calls that were interrupted by a signal fail with an error indication and must be explicitly restarted by the application, if appropriate.
- The `SA_ONSTACK` flag, when set, commands the switch of the process or thread to which the signal is delivered to an alternate stack for the execution of the signal handler; the `sigaltstack` function can be used to set the alternate stack up. If this flag is not set, the signal handler executes on the regular stack of the process or thread.

It should be noted that the setting of the action associated with each kind of signal takes place at the process level, that is, all threads within a process share the same set of actions; hence, for example, it is impossible to set two different signal handling functions (for two different threads) to be executed in response to the same kind of signal.

Immediately after generation, the system checks the process-level action associated with the signal in the target process, and immediately discards the signal if that action is set to ignore it; otherwise, it proceeds to check whether the signal can be acted on immediately.

11.3.4.3 Signal Delivery and Acceptance

Provided that the action associated to the signal at the process level does not specify to ignore the signal in the first place, a signal can be either delivered to or accepted by a thread within the process.

Unlike the action associated to each kind of signal discussed above, each thread has its own signal mask; by means of the signal mask, each thread can selectively block some kinds of signals from being delivered to it, depending on their signal number. The `pthread_sigmask` function allows the calling thread to examine or change (or both) its signal mask. A signal mask can be set up and manipulated by means of the functions:

- `sigemptyset`: initializes a signal mask so that all signals are excluded from the mask.
- `sigfillset`: initializes a signal mask so that all signals are included in the mask.
- `sigaddset`: given a signal mask and a signal number, adds the specified signal to the signal mask; it has no effect if the signal was already in the mask.
- `sigdelset`: given a signal mask and a signal number, removes the specified signal from the signal mask; it has no effect if the signal was not in the mask.
- `sigismember`: given a signal mask and a signal number, checks whether the signal belongs to the signal mask or not.

© 2006 by Taylor & Francis Group, LLC
A signal can be delivered to a thread if and only if that thread does not block the signal; when a signal is successfully delivered to a thread, that thread executes the process-level action associated with the signal.

On the other hand, a thread may perform an explicit wait for one or more kinds of signals, by means of the sigwait function; that function stops the execution of the calling thread until one of the signals passed as argument to sigwait is conveyed to the thread. When this occurs, the thread accepts the signal and continues past the sigwait function. Since the standard specifies that signals in the range from SIGRTMIN to SIGRTMAX are subject to a priority hierarchy, when multiple signals in this range are pending, the sigwait shall consume the lowest-numbered one.

It should also be noted that for this mechanism to work correctly, the thread must block the signals that it wishes to accept by means of sigwait (through its signal mask), otherwise signal delivery takes precedence.

Two, more powerful, variants of the sigwait function exist: sigwaitinfo has an additional argument used to return additional information about the signal just accepted, including the information associated with the signal when it was first generated; furthermore, sigtimedwait also allows the caller to specify the maximum amount of time that shall be spent waiting for a signal to arrive.

The way in which the system selects a thread within a process to convey a signal depends on where the signal is directed:

- If the signal is directed toward a specific thread, only that thread is a candidate for delivery or acceptance.
- If the signal is directed to a process as a whole, any thread belonging to that process is a candidate to receive the signal; in this case, the system selects exactly one thread within the process with the appropriate signal mask (for delivery), or performing a suitable sigwait (for acceptance).

If there is no suitable thread to convey the signal when it is first generated, the signal remains pending until its delivery or acceptance becomes possible, by following the same rules outlined above, or the process-level action associated with that kind of signal is changed and set to ignore it. In the latter case, the system forgets everything about the signal, and all other signals of the same kind.

### 11.3.5 Interprocess Synchronization and Communication

The main interprocess synchronization and communication mechanisms offered by the standard are the semaphore and the message queue, both described in Section 11.2.4. The blocking synchronization primitives have a nonblocking and a timed counterpart, to enhance their real-time predictability. Moreover, multitasking support also adds support for mutual exclusion devices, condition variables, and other synchronization mechanisms. The scope of these mechanisms can be limited to threads belonging to the same process to enhance their performance.

#### 11.3.5.1 Message Queues

The `mq_open` function either creates or opens a message queue and connects it with the calling process; in the system, each message queue is uniquely identified by a name, like a file. This function returns a message queue descriptor that refers to and uniquely identifies the message queue; the descriptor must be passed to all other functions that operate on the message queue.

Conversely, `mq_close` removes the association between the message queue descriptor and its message queue. As a result, the message queue descriptor is no longer valid after successful return from this function. Last, the `mq_unlink` function removes a message queue, provided no other processes reference it; if this is not the case, the removal is postponed until the reference count drops to zero.

The number of elements that a message queue is able to buffer, and their maximum size, are constant for the lifetime of the message queue, and are set when the message queue is first created.

The `mq_send` and `mq_receive` functions send and receive a message to and from a message queue, respectively. If the message cannot be immediately stored or retrieved (e.g., when `mq_send` is executed on a full message queue) these functions block as long as appropriate, unless the message queue was opened.
with the nonblocking option set; if this is the case, these functions return immediately if they are unable to perform their job.

The `mq_timedsend` and `mq_timedreceive` functions have the same behavior, but allow the caller to place an upper bound on the amount of time they may spend waiting.

The standard allows to associate a priority to each message, and specifies that the queueing policy of message queues must obey the priority so that, for example, `mq_receive` retrieves the highest-priority message that is currently stored in the queue.

The `mq_notify` function allows the caller to arrange for the asynchronous notification of message arrival at an empty message queue, when the status of the queue transitions from empty to nonempty, according to the mechanism described in Section 11.3.4. The same function also allows the caller to remove a notification request it made previously.

At any time, only a single process may be registered for notification by a message queue. The registration is removed implicitly when a notification is sent to the registered process, or when the process owning the registration explicitly removes it; in both cases, the message queue becomes available for a new registration.

If both a notification request and an `mq_receive` call are pending on a given message queue, the latter takes precedence, that is, when a message arrives at the queue, it satisfies the `mq_receive` and no notification is sent.

Last, the `mq_getattr` and `mq_setattr` functions allow the caller to get and set, respectively, some attributes of the message queue dynamically after creation; these attributes include, for example, the nonblocking flag just described and may also include additional, implementation-specific flags.

11.3.5.2 Semaphores

Semaphores come in two flavors: unnamed and named. Unnamed semaphores are created by the `sem_init` function and must be shared among processes by means of the usual memory sharing mechanisms provided by the system. On the other hand, named semaphores created and accessed by the `sem_open` function exist as named objects in the system, like the message queues described above, and can therefore be accessed by name. Both functions, when successful, associate the calling process with the semaphore and return a descriptor for it.

Depending on the kind of semaphore, either the `sem_destroy` (for unnamed semaphores) of the `sem_close` function (for named semaphores) must be used to remove the association between the calling process and a semaphore.

For unnamed semaphores, the `sem_destroy` function also destroys the semaphore; instead, named semaphores must be removed from the system with a separate function, `sem_unlink`.

For both kinds of semaphore, a set of functions implements the classic P and V primitives, namely:

- The `sem_wait` function performs a P operation on the semaphore; the `sem_trywait` and `sem_timedwait` functions perform the same function in polling mode and with a user-specified timeout, respectively.
- The `sem_post` function performs a V operation on the semaphore.
- The `sem_getvalue` function has no counterpart in the definition of semaphore found in literature and returns the current value of a semaphore.

11.3.5.3 Mutexes

A mutex is a very specialized binary semaphore that can only be used to ensure the mutual exclusion among multiple threads; it is therefore simpler and more efficient than a full-fledged semaphore. Optionally, it is possible to associate to each mutex a protocol to deal with priority inversion.

The `pthread_mutex_init` function initializes a mutex and prepares it for use; it takes an attribute object as argument, working according to the general mechanism described in Section 11.3.1 and useful to specify the attributes of the mutex like, for example, the priority inversion protocol to be used for it.

When default mutex attributes are appropriate, a static initialization technique is also available; in particular, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize a mutex that the application has statically allocated.
In any case, the `pthread_mutex_destroy` function destroys a mutex.

The following main functions operate on the mutex after creation:

- The `pthread_mutex_lock` function locks the mutex if it is free; otherwise, it blocks until the mutex becomes available and then locks it; the `pthread_mutex_trylock` function does the same, but returns to the caller without blocking if the lock cannot be acquired immediately; the `pthread_mutex_timedlock` function allows the caller to specify a maximum amount of time to be spent waiting for the lock to become available.
- The `pthread_mutex_unlock` function unlocks a mutex.

Additional functions are defined for particular flavors of mutexes; for example, the `pthread_mutex_getprioceiling` and `pthread_mutex_setprioceiling` functions allow the caller to get and set, respectively, the priority ceiling of a mutex, and make sense only if the priority ceiling protocol has been selected for the mutex, by means of a suitable setting of its attributes.

### 11.3.5.4 Condition Variables

A set of condition variables, in concert with a mutex, can be used to implement a synchronization mechanism similar to the monitor without requiring the notion of monitor to be known at the programming language level.

A condition variable must be initialized before use by means of the `pthread_cond_init` function; this function takes an attribute object as argument, which can be used to configure the condition variable to be created, according to the general mechanism described in Section 11.3.1. When default attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` is available to initialize a condition variable that the application has statically allocated.

Then, the mutex and the condition variables can be used as follows:

- Each procedure belonging to the monitor must be explicitly bracketed with a mutex lock at the beginning, and a mutex unlock at the end.
- To block on a condition variable, a thread must call the `pthread_cond_wait` function giving both the condition variable and the mutex used to protect the procedures of the monitor as arguments. This function atomically unlocks the mutex and blocks the caller on the condition variable; the mutex will be reacquired when the thread is unblocked, and before returning from `pthread_cond_wait`. To avoid blocking for a (potentially) unbound time, the `pthread_cond_timedwait` function allows the caller to specify the maximum amount of time that may be spent waiting for the condition variable to be signaled.
- Inside a procedure belonging to the monitor, the `pthread_cond_signal` function, taking a condition variable as argument, can be called to unblock at least one of the threads that are blocked on the specified condition variable; the call has no effect if no threads are blocked on the condition variable. The rather relaxed specification of unblocking at least one thread, instead of exactly one, has been adopted by the standard to simplify the implementation of condition variables on multiprocessor systems, and to make it more efficient, mainly because condition variables are often used as the building block of higher-level synchronization primitives.
- A variant of `pthread_cond_signal`, called `pthread_cond_broadcast`, is available to unblock all threads that are currently waiting on a condition variable. As before, this function has no effect if no threads are waiting on the condition variable.

When no longer needed, condition variables shall be destroyed by means of the `pthread_cond_destroy` function, to save system resources.

### 11.3.5.5 Shared Memory

Except message queues, all IPC mechanisms described so far only provide synchronization among threads and processes, and not data sharing.
Moreover, while all threads belonging to the same process share the same address space, so that they implicitly and inherently share all their global data, the same is not true for different processes; therefore, the POSIX standard specifies an interface to explicitly set up a shared memory object among multiple processes.

The `shmem_open` function either creates or opens a new shared memory object and associates it with a file descriptor, which is then returned to the caller. In the system, each shared memory object is uniquely identified by a name, like a file. After creation, the state of a shared memory object, in particular all data it contains, persists until the shared memory object is unlinked and all active references to it are removed. Instead, the standard does not specify whether a shared memory object remains valid after a reboot of the system or not.

Conversely, `close` removes the association between a file descriptor and the corresponding shared memory object. As a result, the file descriptor is no longer valid after successful return from this function.

Last, the `shmem_unlink` function removes a shared memory object, provided no other processes reference it; if this is not the case, the removal is postponed until the reference count drops to zero.

It should be noted that the association between a shared memory object and a file descriptor belonging to the calling process, performed by `shmem_open`, does not map the shared memory into the address space of the process. In other words, merely opening a shared memory object does not make the shared data accessible to the process.

In order to perform the mapping, the `mmap` function must be called; since the exact details of the address space structure may be unknown to, and uninteresting for the programmer, the same function also provides the capability of choosing a suitable portion of the caller’s address space to place the mapping automatically. The function `munmap` removes a mapping.

### 11.3.6 Thread-Specific Data

All threads belonging to the same process implicitly share the same address space, so that they have shared access to all their global data. As a consequence, only the information allocated on the thread’s stack, such as function arguments and local variables, is private to each thread.

On the other hand, it is often useful in practice to have data structures that are private to a single thread, but can be accessed globally by the code of that thread. The POSIX standard responds to this need by defining the concept of thread-specific data, of which Figure 11.4 depicts the general usage.

The `pthread_key_create` function creates a thread-specific data key visible to, and shared by, all threads in the process. The key values provided by this function are opaque objects used to access thread-specific data.

In particular, the pair of functions `pthread_getspecific` and `pthread_setspecific` take a key as argument and allow the caller to get and set, respectively, a pointer uniquely bound with the given key.

![Thread-specific data in POSIX](image)

**Figure 11.4** Thread-specific data in POSIX.
key, and private to the calling thread. The pointer bound to the key by pthread_setspecific persists for the life of the calling thread, unless it is replaced by a subsequent call to pthread_setspecific.

An optional destructor function may be associated with each key when the key itself is created. When a thread exits, if a given key has a valid destructor, and the thread has a valid (i.e., not NULL) pointer associated with that key, the pointer is disassociated and set to NULL, and then the destructor is called with the previously associated pointer as argument.

When it is no longer needed, a thread-specific data key should be deleted by invoking the pthread_key_delete function on it. It should be noted that, unlike in the previous case, this function does not invoke the destructor function associated with the key, so that it is the responsibility of the application to perform any cleanup actions for data structures related to the key being deleted.

11.3.7 Memory Management

The standard allows processes to lock parts or all of their address space in main memory by means of the mlock and mlockall functions; in addition, mlockall also allows the caller to demand that all of the pages that will become mapped into the address space of the process in the future must be implicitly locked.

The lock operation both forces the memory residence of the virtual memory pages involved, and prevents them from being paged out in the future. This is vital in operating systems that support demand paging and must nevertheless support any real-time processing, because the paging activity could introduce undue and highly unpredictable delays when a real-time process attempts to access a page that is currently not in the main memory and must therefore be retrieved from secondary storage.

When the lock is no longer needed, the process can invoke either the munlock or the munlockall function to release it and enable demand-paging again.

Other memory management functions, such as the mmap function already described in Section 11.3.5.5, establish a mapping between the address space of the calling process and a memory object, possibly shared between multiple processes. The mapping facility is general enough; hence, it can also be used to map other kinds of objects, such as files and devices, into the address space of a process, provided both the hardware and the operating system have this capability, which is not mandated by the standard. For example, once a file is mapped, a process can access it simply by reading or writing the data at the address range to which the file was mapped.

Finally, it is possible for a process to change the access protections of portions of its address space by means of the mprotect function; in this case, it is assumed that protections will be enforced by the hardware. For example, to prevent inadvertent data corruption due to a software bug, one could protect critical data intended for read-only usage against write access.

11.3.8 Asynchronous and List Directed Input and Output

Many operating systems carry out I/O operations synchronously with respect to the process requesting them. Thus, for example, if a process invokes a file read operation, it stays blocked until the operating system has finished it, either successfully or unsuccessfully. As a side effect, any process can have at most one pending I/O operation at any given time.

While this programming model is simple, intuitive, and perfectly adequate for general-purpose systems, it shows its limits in a real-time environment, namely:

- I/O device access timings can vary widely, especially when an error occurs; hence, it is not always wise to suspend the execution of a process until the operation completes, because this would introduce a source of unpredictability in the system.
- It is often desirable, for example, to enhance system performance by exploiting I/O hardware parallelism, to start more than one I/O operation simultaneously, under the control of a single process.
To satisfy these requirements, the standard defines a set of functions to start one or more I/O requests, to be carried out in parallel with process execution, and whose completion status can be retrieved asynchronously by the requesting process.

Asynchronous and list-directed I/O functions revolve around the concept of asynchronous I/O control block, struct aiocb; this structure contains all the information needed to describe an I/O operation, and contains members to:

- Specify the operation to be performed, read or write.
- Identify the file on which the operation must be carried out, by means of a file descriptor.
- Determine what portion of the file the operation will operate upon, by means of a file offset and a transfer length.
- Locate a data buffer in memory to be used to store or retrieve the data read from, or to be written to, the file.
- Give a priority classification to the operation.
- Request the asynchronous notification of the completion of the operation, either by a signal or by the asynchronous execution of a function, as described in Section 11.3.4.

Then, the following functions are available:

- The aio_read and aio_write functions take an I/O control block as argument and schedule a read or a write operation, respectively; both return to the caller as soon as the request has been queued for execution.
- As an extension, the lio_listio function schedules a list of (possibly asynchronous) I/O requests, each described by an I/O control block, with a single function call.
- The aio_error and aio_return functions allow the caller to retrieve the error and status information associated with an I/O control block, after the corresponding I/O operation has been completed.
- The aio_fsync function asynchronously forces all I/O operations associated with the file indicated by the I/O control block passed as argument and currently queued to the synchronized I/O completion state.
- The aio_suspend function can be used to block the calling thread until at least one of the I/O operations associated with a set of I/O control blocks passed as argument completes, or up to a maximum amount of time.
- The aio_cancel function cancels an I/O operation that has not been completed yet.

### 11.3.9 Clocks and Timers

Real-time applications very often rely on timing information to operate correctly; the POSIX standard specifies support for one or more timing bases, called clocks, of known resolution and whose value can be retrieved at will. In the system, each clock has its own unique identifier. The clock_gettime and clock_settime functions get and set the value of a clock, respectively, while the clock_getres function returns the resolution of a clock. Clock resolutions are implementation-defined and cannot be set by a process; some operating systems allow the clock resolution to be set at system generation or configuration time.

In addition, applications can set one or more per-process timers, using a specified clock as a timing base, by means of the timer_create function. Each timer has a current value and, optionally, a reload value associated with it. The operating system decrements the current value of timers according to their clock and, when a timer expires, it notifies the owning process with an asynchronous notification of timer expiration; as described in Section 11.3.4, the notification can be carried out either by a signal, or by awakening a thread belonging to the process. On timer expiration, the operating system also reloads the timer with its reload value, if it has been set, thus possibly realizing a repetitive timer.
When a timer is no longer needed, it shall be removed by means of the timer_delete function, that both stops the timer and frees all resources allocated to it.

Since, due to scheduling or processor load constraints, a process could lose one or more notifications of expiration, the standard also specifies a way for applications to retrieve, by means of the timer_getoverrun function, the number of "missed" notifications, that is, the number of extra timer expirations that occurred between the time at which a given timer expired and when the notification associated with the expiration was eventually delivered to, or accepted by, the process.

At any time, it is also possible to store a new value into, or retrieve the current value of a timer by means of the timer_settime and timer_gettime functions, respectively.

11.3.10 Cancellation

Any thread may request the cancellation of another thread in the same process by means of the pthread_cancel function. Then, the target thread's cancelability state and type determines whether and when the cancellation takes effect. When the cancellation takes effect, the target thread is terminated.

Each thread can atomically get and set its own way to react to a cancellation request by means of the pthread_setcancelstate and pthread_setcanceltype functions. In particular, three different settings are possible:

- The thread can ignore cancellation requests completely.
- The thread can accept the cancellation request immediately.
- The thread can be willing to accept the cancellation requests only when its execution flow crosses a cancellation point. A cancellation point can be explicitly placed in the code by calling the pthread_testcancel function. Also, it should be remembered that many functions specified by the POSIX standard act as implicit cancellation points.

The choice of the most appropriate response to cancellation requests depends on the application and is a trade-off between the desirable feature of really being able to cancel a thread, and the necessity of avoiding the cancellation of a thread while it is executing in a critical section of code, both to keep the guarded data structures consistent and to ensure that any IPC object associated with the critical section, for example, a mutex, is released appropriately; otherwise, the critical region would stay locked forever, likely inducing a deadlock in the system.

As an aid to do this, the POSIX standard also specifies a mechanism that allows any thread to register a set of cleanup handlers on a stack to be executed, in LIFO order, when the thread either exits voluntarily, or accepts a cancellation request. The pthread_cleanup_push and pthread_cleanup_pop functions push and pop a cleanup handler into and from the handler stack; the latter function also has the ability to execute the handler it is about to remove.

11.4 Real-Time, Open-Source Operating Systems

Although in the general-purpose operating system camp a handful of products dominates the market, there are more than 30 real-time operating systems available for use today, both commercial and experimental, and new ones are still being developed. This is due both to the fact that there is much research in the latter area, and that real-time embedded applications are inherently less homogeneous than general-purpose applications, such as those found in office automation; hence, ad hoc operating system features are often needed. In addition, the computing power and the overall hardware architecture of embedded systems are much more varied than, for example, those of personal computers.

This section focuses on open-source real-time operating systems, a recent source of novelty in embedded system software design. Open-source operating systems are especially promising, for two main reasons:

1. The source code of an open-source real-time operating system can be used both to develop real-world applications, and for study and experimentation. Therefore, open-source operating
systems often implement the most advanced, state-of-the-art architectures and algorithms because researchers can play with them at will and their work can immediately be reflected into real applications.

2. Open-source operating systems have no purchasing cost and are inherently royalty free, so their adoption can cut down the costs of an application. Moreover, one of the most well-known issues of open-source operating systems, namely the lack of official technical support, has recently found its way to a solution with more and more consulting firms specializing in their support.

Among the open-source operating systems that have found their way into commercial products we recall:

- **eCos.** The development of eCos [9] is coordinated by Red Hat Inc. and is based on a modular, layered real-time kernel. The most important innovation in eCos is its extensive configuration system that operates at kernel build time with a large number of configuration points placed at source code level, and allows a very fine-grained adaptation of the kernel itself to both application needs and hardware characteristics. The output of the configuration process is an operating system library that can then be linked with the application code.

  Its application programming interface is compatible with the POSIX standard, but it does not support multiple processes with independent address spaces, even when a Memory Management Unit (MMU) is available.

- **µClinux.** It is a stripped-down version of the well-known Linux operating system. The most interesting features of µClinux [10] are the ability to run on microcontrollers that lack an MMU and its small size, compared with a standard Linux kernel. As is, µClinux does not have any real-time capability, because it inherits its standard processor scheduler from Linux; however, both RT-Linux [11] and RTAI [12] real-time extensions are available for it.

- **RT-Linux and RTAI.** These are hard real-time-capable extensions to the Linux operating system; they are similar, and their architecture was first outlined in 1997 [11–13]. The main design feature of both RT-Linux and RTAI is the clear separation between the real-time and nonreal-time domains: in RT-Linux and RTAI, a small monolithic real-time kernel runs real-time tasks, and coexists with the Linux kernel.

  As a consequence, nonreal-time tasks running on the Linux kernel have the sophisticated services of a standard time-sharing operating system at their disposal, whereas real-time tasks operate in a protected, predictable and low-latency environment.

  The real-time kernel performs first-level real-time scheduling and interrupt handling, and runs the Linux kernel as its lowest-priority task. In order to keep changes in the Linux kernel to an absolute minimum, the real-time kernel provides for an emulation of the interrupt control hardware. In particular, any interrupt disable/enable request issued by the Linux kernel is not passed to the hardware, but is emulated in the real-time kernel instead; thus, for example, when Linux disables interrupts, the hardware interrupts actually stay enabled and the real-time kernel queues and delays the delivery of any interrupt of interest to the Linux kernel. Real-time interrupts are not affected at all, and are handled as usual, without any performance penalty.

  To handle communication between real-time and nonreal-time tasks, RT-Linux and RTAI implement lock-free queues and shared memory. In this way, real-time applications can rely on Linux system services for nontime-critical operations, such as filesystem access and graphics user interface.

- **RTEMS.** The development of RTEMS [14] is coordinated by On-Line Applications Research Corporation, which also offers paid technical support. Its application development environment is based on open-source GNU tools, and has a monolithic/layered architecture, commonly found in high-performance real-time executives.

  RTEMS complies with the POSIX 1003.1b application programming interface and supports multiple threads of execution, but does not implement a multiprocess environment with independent application address spaces. It also supports networking and a filesystem.
11.5 Virtual-Machine Operating Systems

According to its most general definition, a virtual machine is an abstract system, composed of one or more virtual processors and zero or more virtual devices. The implementation of a virtual machine can be carried out in a number of different ways, such as, for example, interpretation, (partial) just-in-time compilation, and hardware-assisted instruction-level emulation. Moreover, these techniques can be, and usually are, combined to obtain the best compromise between complexity of implementation and performance for a given class of applications.

In this section, we focus on perfect virtualization, that is, the implementation of virtual machines whose processors and I/O devices are identical in all respects to their counterpart in a physical machine, that is, also the machine on which the virtualization software runs. The implementation of virtual machines is carried out by means of a peculiar kind of operating system kernel; hardware assistance keeps overheads to a minimum.

As described in Section 11.2.1, the internal architecture of an operating system based on virtual machines revolves around the basic observation that an operating system must perform two essential functions:

- Multiprogramming
- System services

Accordingly, those operating systems fully separate these functions and implement them as two distinct operating system components:

- A virtual machine monitor
- A guest operating system

11.5.1 Related Work

A particularly interesting application of the layered approach to operating system design, the virtual machines, were first introduced by Meyer and Seawright in the experimental CP-40 and, later, CP-67 system [15] on an IBM 360/67. This early system was peculiar because it provided each user with a virtual IBM 360/65 (not 67) including I/O devices. So, processor and I/O virtualization was perfect, but MMU virtualization was not attempted at all, and the virtual machines thus lacked an MMU.

Later on, MMU virtualization was added by A. Auroux, and CP-67 evolved into a true IBM product, VM/370. Offsprings of VM/370 are still in use today on IBM mainframes; for example, z/VM [16] runs on IBM zSeries mainframes and supports the execution of multiple z/OS and Linux operating system images, each in its own virtual machine.

An extensive, early discussion of virtual machines and their properties can be found in References 17 and 18.

More recently, microcode support for virtual machines was added to the 680x0 microprocessor family in the transition between the 68000 and the 68010 [19]; in particular, the privilege level required to execute some instructions was raised to make processor virtualization feasible.

Commercial virtualization products now available include, for example, the VMware virtualization software for the Intel Pentium architecture [20]. For example, in Reference 21 this product was used to give a prototype implementation of a virtual machine-based platform for trusted computing.

More advanced attempts at virtualization in various forms can be found in References 22 and 23. In particular, Reference 22 discusses in detail the trade-off between "perfect" virtualization and efficiency.

11.5.2 Views of Processor State

The execution of each machine instruction both depends on, and affects, the internal processor state. In addition, the semantics of an instruction depend on the processor execution mode the processor was in.
when the instruction itself was executed, since the processor mode directly determines the privilege level of the processor itself. For example:

- On the ARM V5 [24] processor, the execution of the \texttt{ADD R13, R13, #1} instruction increments by one the contents of register \texttt{R13}.
- The outcome of the \texttt{BEQ label} instruction depends on the current value of the \texttt{Z} processor state flag, and conditionally updates the contents of the program counter.

The \textit{view} that machine code has of the internal processor state depends on the mode the processor is running in. In particular, let us define two, somewhat simplified, views of the processor state:

\textit{User-mode view.} It is the portion of the processor state that is accessible through machine instructions, with either read-only or read-write access rights, when the processor is in user mode. In other words, it is the portion of the processor state that can be accessed by unprivileged machine instructions.

\textit{Privileged-mode view.} It is the portion of the processor state that is accessible through machine instructions, with either read-only or read-write access rights, when the processor is in privileged mode. It usually is a superset of the user-mode state and, if the processor supports a single privileged mode, coincides with full access to the processor state as a whole.

When the processor supports either privilege rings or multiple, independent privileged modes, the definition of privileged-mode view becomes more complicated, and involves either:

- A nested set of views when the processor supports privilege rings. In this case, the inner view, corresponding to the most privileged processor mode, encompasses the machine state as a whole with the most powerful access rights; outer, less privileged modes have more restricted views of the processor state. Above, \textit{nested} means that the outer view either has no visibility of a processor state item that is visible from the inner view, or that the outer view has less powerful access rights than the inner view on one or more processor state items.
- A collection of independent views when the processor supports multiple, independent privileged modes. In this case, it should be noted that the intersection among views can be, and usually is, not empty; for example, in the ARM V5 processor, user registers \texttt{R0} through \texttt{R7} are accessible from, and common to all unprivileged and privileged processor modes. In addition, registers \texttt{R8} through \texttt{R12} are common to all but one processor mode. It should be noted also that only the union of all views give full access to the processor state: in general, no individual view can do the same, not even the view corresponding to the “most privileged” privileged mode, even if the processor specification contains such a hierarchical classification of privileged modes.

Continuing the first example above, if the processor implements multiple, mode-dependent instances of register \texttt{R13}, the execution of the \texttt{ADD} instruction presented above in user mode will update the user-mode view of \texttt{R13}, but will not affect the view of the same register in any other mode.

As customary, we define a \textit{process} as the activity of executing a program; a process therefore encompasses both the program \textit{text} and a view of the current \textit{processor state} when the execution takes place. The latter includes the notion of execution progress that could be captured, for example, by a program counter.

### 11.5.3 Operating Principle

The most important concept behind processor virtualization is that a low-level system software component, the \textit{Virtual Machine Monitor} (or VMM for short), also historically known as the \textit{Control Program} (or CP), performs the following functions, likely with hardware assistance:

- It gives to a set of machine code programs, running either in user or privileged mode, their own, independent view of the processor state; this gives rise to a set of independent sequential processes.
- Each view is “correct” from the point of view of the corresponding program, in the sense that the view is indistinguishable from the view the program would have when run on a bare machine,
without a VMM in between. This requirement supports the illusion that each process runs on its own processor, and that processor is identical to the physical processor below the VMM.

- It is able to switch the processor among the processes mentioned above; in this way, the VMM implements multiprogramming. Switching the processor involves both a switch among possibly different program texts, and among distinct processor state views.

The key difference of the VMM approach with respect to traditional multiprogramming is that a traditional operating system confines user-written programs to run in user mode only, and to access privileged mode by means of a system call. Thus, each service request made by a user program traps to the operating system and switches the processor to privileged mode; the operating system, running in privileged mode, performs the service on behalf of the user program and then returns control to it, simultaneously bringing the processor back to user mode.

A VMM system, instead, supports virtual processors that can run either in user or in privileged mode, just like the real, physical processor they mimic; the real processor must necessarily execute the processes inside the virtual machine — also called the virtual machine guest code — in user mode, to keep control of the system as a whole.

We must therefore distinguish between:

- The real, physical processor mode, that is, the processor mode the physical processor is running in. At each instant, each physical processor in the system is characterized by its current processor mode.
- The virtual processor mode, that is, the processor mode each virtual processor is running in. At each instant, each virtual machine is characterized by its current processor mode, and it does not necessarily coincide with the physical processor mode, even when the virtual machine is being executed.

### 11.5.4 Virtualization by Instruction Emulation

The classic approach to processor virtualization is based on privileged instruction emulation; with this approach, the VMM maintains a virtual machine control block (or VMCB for short) for each virtual machine.

Among other things, the VMCB holds the full processor state, both unprivileged and privileged, of a virtual processor; therefore, it contains state information belonging to, and accessible from, distinct views of the processor state, with different levels of privilege.

When a virtual machine is being executed by a physical processor, the VMM transfers part of the VMCB into the physical processor state; when the VMM assigns the physical processor to another virtual machine, the physical processor state is transferred back into the VMCB.

It is important to notice that virtual machines are always executed with the physical processor in user mode, regardless of the virtual processor mode. Most virtual machine instructions are executed directly by the physical processor, with zero overhead; however, some instructions must be emulated by the VMM to incur a trap handling overhead. In particular:

1. Unprivileged instructions act on, and depend on, the current view of the processor state only, and are executed directly by the physical processor. Two subcases are possible, depending on the current virtual processor mode:
   - (a) Both the virtual and the physical processor are in user mode. In this case, virtual and physical instruction execution and their corresponding processor state views fully coincide, and no further manipulation of the processor state is necessary.
   - (b) The virtual processor is running in a privileged mode and the physical processor is in user mode. So, instruction execution acts on the user-mode view of the processor state, and the intended effect is to act on one of the privileged views. To compensate for this, the VMM must update the contents of the user-mode view in the physical processor from the appropriate
portion of the VMCB whenever the virtual processor changes state. Even in this case, the overhead incurred during actual instruction execution is zero.

2. Privileged instructions act on one of the privileged views of the processor state. So, when the execution of a privileged instruction is attempted in physical user mode, the physical processor takes a trap to the VMM. In turn, the VMM must emulate either the trap or the trapped instruction, depending on the current virtual privilege level, and reflect the outcome of the emulation in the virtual processor state stored in the VMCB:

(a) If the virtual processor was in user mode when the trap occurred, the VMM must emulate the trap. Actual trap handling will be performed by the privileged software inside the virtual machine, in virtual privileged mode, because the emulation of the trap, among other things, switches the virtual processor into privileged mode. The virtual machine privileged software actually receives the emulated trap.

(b) If the virtual processor was in privileged mode, and the trap was triggered by lack of the required physical processor privilege level, the VMM must emulate the privileged instruction; in this case, the VMM itself performs trap handling and the privileged software inside the virtual machine does not receive the trap at all. Instead, it sees the outcome of the emulated execution of the privileged instruction.

(c) If the virtual processor was in privileged mode, and the trap was triggered by other reasons, the VMM must emulate the trap; the actual trap handling, if any, will be performed by the privileged software inside the virtual machine, in virtual privileged mode. It should be noted in fact that, in most simple processor architectures and operating systems, a trap occurring in a privileged mode is usually considered to be a fatal condition, and triggers the immediate shutdown of the operating system itself.

In the first and third case above, the behavior of the virtual processor exactly matches the behavior of a physical processor in the same situation, except that the trap enter mechanism is emulated in software instead of being performed either in hardware or in microcode.

In the second case, the overall behavior of the virtual processor still matches the behavior of a physical processor in the same situation, but the trap is kept invisible to the virtual machine guest software because, in this case, the trap is instrumental for the VMM to properly catch and emulate the privileged instruction.

3. A third class of instructions includes unprivileged instructions whose outcome depends on a physical processor state item belonging to privileged processor state views only.

The third and last class of instructions is anomalous and problematic in nature from the point of view of processor virtualization, because these instructions allow a program to infer something about a processor state item that would not be accessible from its current view of the processor state itself.

The presence of instructions of this kind hampers the privileged instruction emulation approach to processor virtualization just discussed, because it is based on the separation between physical processor state and virtual processor state, and enforces this separation by trapping (and then emulating in the virtual processor context) all instruction that try to access privileged processor state views. Instructions of this kind are able to bypass this mechanism as a whole, because they generate no trap, so the VMM is unable to detect and emulate them; instead, they take information directly from the physical processor state.

11.5.5 Processor-Mode Change

A change in the mode of a virtual processor may occur for several reasons, of which the main ones are:

- When the execution of an instruction triggers a trap; in this case, trap handling is synchronous with respect to the code being executed.
When an interrupt request for the virtual machine comes in, and is accepted; the interrupt request is asynchronous with respect to the code being executed, but the hardware implicitly synchronizes interrupt handling with instruction boundaries.

In both cases, the VMM takes control of the physical processor and then implements the mode change by manipulating the virtual machine's VMCB and, if the virtual machine was being executed when the mode change was requested, a portion of the physical processor state.

In particular, to implement the processor-mode change the VMM must perform the following actions:

- Save the portion of the physical processor state pertaining to the processor state view of the old processor state into the VMCB; for example, if the processor was running in user mode, the user-mode registers currently loaded in the physical processor registers must be saved.
- Update the VMCB to reflect the effects of the mode change to the virtual processor state; for example, a system call instruction will likely modify the program counter of the virtual processor and load a new processor status word.
- Load the physical processor state pertaining to the state view of the new processor state from the VMCB; for example, if the processor is being switched to a privileged mode, the privileged-mode registers must be transferred from the VMCB into the user-mode registers of the physical processor.

Notice that the mode of the registers saved in the VMCB and the accessibility mode of the physical processor registers into which they are loaded do not coincide.

### 11.5.6 Privileged Instruction Emulation

To perform its duties, the VMM must be able to receive and handle traps on behalf of a virtual machine, and these traps can be triggered for a variety of reasons. When using the privileged instruction emulation approach to processor virtualization, the most common trap reason is the request to emulate a privileged instruction.

The VMM must perform privileged instruction emulation when a virtual processor attempts to execute a legal privileged instruction while in virtual privileged mode. In this case, the physical processor (running in physical user mode) takes a “privileged instruction” trap that would have not been taken if it were in privileged mode as the virtual machine software expects it to be.

The main steps of the instruction emulation sequence are:

- Save into the VMCB all registers in the view corresponding to the current virtual processor mode. This both “freezes” the virtual machine state for subsequent instruction emulation, and frees the physical processor state for VMM use.
- Locate and decode the instruction to be emulated in the virtual processor instruction stream. This operation may involve multiple steps because, for example, on superscalar or deeply pipelined architecture, the exact value of the program counter at the time of the trap can not be easy to compute.
- Switch the physical processor into the appropriate privileged mode for instruction emulation, that is, to the processor mode of the virtual processor. The trap handling mechanism of the physical processor always switches the processor into a privileged mode, but if the processor supports multiple privileged modes then the privileged mode might not coincide with the actual privileged mode of the virtual processor.
- Emulate the instruction using the VMCB as the reference machine state for the emulation, and reflect its outcome into the VMCB itself. Notice that the execution of a privileged instruction may update both the privileged and the unprivileged portion of the virtual processor state, so the VMCB as a whole is involved. Also, the execution of a privileged instruction may change the processor mode of the virtual processor.
• Update the virtual program counter in the VMCB to the next instruction in the instruction stream of the virtual processor.
• Restore the virtual processor state from the updated VMCB and return from the trap.

In the last step above, the virtual processor state can in principle be restored either:

• From the VMCB of the virtual machine that generated the trap in the first place, if the processor scheduler of the VMM is not invoked after instruction emulation; this is the case just described.
• From the VMCB of another virtual machine, if the processor scheduler of the VMM is invoked after instruction emulation.

### 11.5.7 Exception Handling

When any synchronous exception other than a privileged instruction trap, occurs in either virtual user or virtual privileged modes, the VMM, and not the guest operating system of the virtual machine, receives the trap in the first place.

When the trap is not instrumental to the implementation of virtual machines, as it happens in most cases, the VMM must simply emulate the trap mechanism itself inside the virtual machine, and appropriately update the VMCB to reflect the trap back to the privileged virtual machine code.

Another situation in which the VMM must simply propagate the trap is the occurrence of a privileged instruction trap when the virtual processor is in virtual user mode. This occurrence usually, but not always, indicates a bug in the guest software: an easy counterexample can be found when a VMM is running inside the virtual machine.

A special case of exception is that generated by the system call instruction, whenever it is implemented. However, from the point of view of the VMM, this kind of exception is handled exactly as all others; only the interpretation given to the exception by the guest code running in the virtual machine is different.

It should also be noted that asynchronous exceptions, such as interrupt requests, must be handled in a different and more complex way, as described in the following section.

### 11.5.8 Interrupt Handling

We distinguish between three kinds of interrupt; each of them requires a different handling strategy by the VMM:

• Interrupts triggered by, and destined to, the VMM itself, for example, the VMM processor scheduler timeslice interrupt, and the VMM console interrupt; in this case, no virtual machine ever notices the interrupt.
• Interrupts destined to a single virtual machine, for example, a disk interrupt for a physical disk permanently assigned to a virtual machine.
• Interrupts synthesized by the VMM, either by itself or as a consequence of another interrupt, and destined to a virtual machine, for example, a disk interrupt for a virtual disk emulated by the VMM, or a network interrupt for a virtual communication channel between virtual machines.

In either case, the general approach to interrupt handling is the same, and the delivery of an interrupt request to a virtual machine implies at least the following steps:

• If the processor was executing in a virtual machine, save the status of the current virtual machine, if any, into the corresponding VMCB; then, switch the processor onto the VMM context and stack, and select the most privileged processor mode. Else, the processor was already executing the VMM; the processor already is in the VMM context and stack, and runs at the right privilege level. In both cases, after this phase, the current virtual machine context has been secured in its VMCB and the physical processor can freely be used by VMM code; this is also a good boundary for the transition between the portion of the VMM written in assembly code and the bulk of the VMM, written in a higher-level programming language.
• Determine the type of interrupt request and to which virtual machine it must be dispatched; then, emulate the interrupt processing normally performed by the physical processor in the corresponding VMCB. An additional complication arises if the target virtual machine is the current virtual machine and the VMM was in active execution, that is, it was emulating an instruction on behalf of the virtual machine itself, when the request arrived. In this case, the simplest approach, which also adheres most to the behavior of the physical processor, is to defer interrupt emulation to the end of the current emulation sequence. To implement the deferred handling mechanism efficiently, some features of the physical processor, such as Asynchronous System Traps (ASTs) and deferrable software interrupts, may be useful; unfortunately, they are now uncommon on RISC (Reduced Instruction Set Computer) machines.

• Return either to the VMM or the virtual machine code that was being executed when the interrupt request arrived. Notice that, at this point, no actual interrupt handling took place yet, and that some devices may require some limited intervention before returning from their interrupt handler, for example, to release their interrupt request line. In this case, it may be necessary to incorporate this low-level interrupt handling in the VMM directly, and at the same time ensure that it is idempotent when repeated by the virtual machine interrupt handler.

The opportunity and the relative advantages and disadvantages of invoking the VMM processor scheduler after each interrupt request will be discussed in Section 11.5.10.

11.5.9 Trap Redirection

A problem common to privileged instruction emulation, exception handling, and interrupt handling is that the VMM must be able to intercept any exception the processor takes while executing on behalf of a virtual machine and direct it toward its own handler.

Most modern processors use an unified trap vector or dispatch table for all kinds of trap, exception, and interrupt. Each trap type has its own code that is used as an index in the trap table to fetch the address in memory at which the corresponding trap handler starts. A slightly different approach is to execute the instruction inside the table directly (in turn, the instruction will usually be an unconditional jump instruction), but the net effect is the same. A privileged register, usually called the trap table base register, gives the starting address of the table.

In either case, all vectors actually used by the physical processor when handling a trap reside in the privileged address space, and are accessed after the physical processor has been switched into an appropriate privileged mode. The VMM must have full control on these vectors, because it relies on them to intercept traps at runtime.

On the other hand, the virtual machine guest code should be able to set its own trap table, with any vectors it desires; the latter table resides in the virtually privileged address space of the virtual machine, and must be accessible to the virtual machine guest code in read and write mode. The content of this table is not used by the physical processor, but by the VMM to compute the target address to which to redirect traps via emulation.

The simplest approach to accommodate these conflicting needs, when it is not possible to map the same virtual address to multiple, distinct physical addresses depending on the processor mode without software intervention — which is quite a common restriction on simple MMUs — is to reserve in the addressing space of each virtual machine a phantom page that is not currently in use by the guest code, grant read-only access to it only when the processor is in physical privileged mode to keep it invisible to the guest code, and store the actual trap table there and direct the processor to use it by setting its trap table base register appropriately.

The initialization of the actual trap table is made by the VMM for each virtual machine during the initial instantiation of the virtual machine itself. Since the initialization is performed by the VMM (and not by the virtual machine), the read-only access restriction described above does not apply.
The VMM must then intercept any access made by the virtual machine guest code to the trap table base register, in order to properly locate the virtual trap table and be able to compute the target address to which to redirect traps.

In other words, the availability of a trap table base register allows the guest code to set up its own, virtual trap table, and the VMM to set up the trap table obeyed by the physical processor, without resorting to virtual/physical address mapping functions that depend on the processor mode.

Two complementary approaches can be followed to determine the exact location of the phantom page in the address space of the virtual machine. In principle, the two approaches are the same; the difference between them is the compromise between ease of implementation, overheads at runtime, and flexibility:

- If the characteristics of the guest code, in particular of the guest operating systems, are well known, the location of the phantom page can be fixed in advance. This is the simpler choice, and has also the least runtime overhead.
- If one does not want to make any assumption at all about the guest code, the location of the phantom page must be computed dynamically from the current contents of the page table set up by the guest code and may change with time (e.g., when the guest code decides to map the location in which the phantom page currently resides for its own use). When the location of the phantom page changes, the VMM must update the trap table base register accordingly; moreover, it must ensure that the first-level trap handlers contain only position-independent code.

### 11.5.10 VMM Processor Scheduler

The main purpose of the VMM, with respect to processor virtualization, is to emulate the privileged instructions issued by virtual processors in virtual privileged mode. Code fragments implementing the emulation are usually short and often require atomic execution, above all with respect to interrupt requests directed to the same virtual machine.

The processor scheduling activity carried out by the VMM by means of the VMM processor scheduler is tied to the emulation of privileged instructions quite naturally, because the handling of a privileged instruction exception is a convenient scheduling point.

On the other hand, the main role of the VMM in interrupt handling is to redirect each interrupt to the virtual machine(s) interested in it; this action, too, must be completed atomically with respect to instruction execution on the same virtual machine, like the physical processor itself would do.

This suggests to disable the rescheduling of the physical processor if the processor was executing the VMM when the interrupt arrived, and to delay the rescheduling until the end of the current VMM emulation path. The advantage of this approach is twofold:

- The state of the VMM must never be saved when switching between different virtual machines.
- The VMM must not be reentrant with respect to itself, because a VMM/VMM context switch cannot occur.

By contrast, the processor allocation latency gets worse because the maximum latency, not taking higher-priority activities into account, becomes the sum of:

- The longest emulation path in the VMM.
- The longest sequence of instructions in the VMM to be executed with interrupts disabled, due to synchronization constraints.
- The scheduling time.
- The virtual machine context save and restore time.

In a naive implementation, making the VMM not preemptable seems promising, because it is conceptually simple and does impose a negligible performance penalty in the average case, if it is assumed that the occurrence of an interrupt that needs an immediate rescheduling while VMM execution is in progress is rare.
Also, some of the contributions to the processor allocation latency described earlier, mainly the length of instruction emulation paths in the VMM and the statistical distribution of the different instruction classes to be emulated in the instruction stream, will be better known only after some experimentation because they also depend on the behavior of the guest operating systems and their applications. It must also be taken into account that making the VMM preemptable will likely give additional performance penalties in the average case:

- The switch between virtual machines will become more complex, because their corresponding VMM state must be switched as well.
- The preemption of some VMM operations, for example, the propagation of an interrupt request, may be ill-defined if it occurs while a privileged instruction of the same virtual machine is being emulated.

So, at least for soft real-time applications, implementing preemption of individual instruction emulation in the VMM should be done only if it is strictly necessary to satisfy the latency requirements of the application, and after extensive experimentation.

From the point of view of the scheduling algorithms, at least in a naive implementation, a fixed-priority scheduler with a global priority level assigned to each virtual machine is deemed to be the best choice, because:

- It easily accommodates the common case in which nonreal-time tasks are confined under the control of a general-purpose operating system in a virtual machine, and real-time tasks either run under the control of a real-time operating system in another virtual machine, or have a private virtual machine each.
- The sensible selection of a more sophisticated scheduling algorithm can be accomplished only after extensive experimentation with the actual set of applications to be run, and when a detailed model of the real-time behavior of the application itself and of the devices it depends on is available.
- The choice of algorithms to be used when multiple, hierarchical schedulers are in effect in a real-time environment has not yet received extensive attention in the literature.

References


© 2006 by Taylor & Francis Group, LLC