Real-Time Operating Systems

Chapter · July 2017

DOI: 10.1002/9781119420712.ch3

1 author:

Jiacun Wang
Monmouth University
261 PUBLICATIONS **2,300** CITATIONS

SEE PROFILE

3

Real-Time Operating Systems

The heart of many computerized embedded systems is real-time operating system (RTOS). An RTOS is an operating system that supports the construction of applications that must meet real-time constraints in addition to providing logically correct computation results. It provides mechanisms and services to carry out real-time task scheduling, resource management, and intertask communication. In this chapter, we briefly review the main functions of general-purpose operating systems, and then we discuss the characteristics of RTOS kernels. After that, we introduce some widely used RTOS products.

3.1 Main Functions of General-Purpose Operating Systems

An operating system (OS) is the software that sits between the hardware of a computer and software applications running on the computer. An OS is a resource allocator and manager. It manages the computer hardware resources and hides the details of how the hardware operates to make the computer system more convenient to use. The main hardware resources in a computer are processor, memory, I/O controllers, disks, and other devices such as terminals and networks.

An OS is a policy enforcer. It defines the rules of engagement between the applications and resources and controls the execution of applications to prevent errors and improper use of the computer.

An OS is composed of multiple software components, and the core components in the OS form its *kernel*. The kernel provides the most basic level of control over all of the computer's hardware devices. The kernel of an OS always runs in *system mode*, while other parts and all applications run in *user mode*. Kernel functions are implemented with protection mechanisms such that they could not be covertly changed through the actions of software running in user space.

Real-Time Embedded Systems, First Edition. Jiacun Wang.

The OS also provides an application programming interface (API), which defines the rules and interfaces that enable applications to use OS features and communicate with the hardware and other software applications. User processes can request kernel services through *system call* or by *message passing*. With the system call approach, the user process applies traps to the OS routine that determines which function is to be invoked, switches the processor to system mode, calls the function as a procedure, and then switches the processor back to user mode when the function completes and returns control to the user process. In the message passing approach, the user process constructs a message that describes the desired service and then uses a *send* function to pass it to an OS process. The *send* function checks the desired service specified in the message, changes the processor mode to system mode, and delivers it to the process that implements the service. Meanwhile, the user process waits for the result of the service request with a message *receive* operation. When the service is completed, the OS process sends a message back to the user process.

In the rest of this section, we briefly introduce some of the main functions of a typical general-purpose OS.

3.1.1 Process Management

A *process* is an instance of a program in execution. It is a unit of work within the system. A program is a *passive* entity, while a process is an *active entity*. A process needs resources, such as CPU, memory, I/O devices, and files, to accomplish its task. Executing an application program involves the creation of a process by the OS kernel that assigns memory space and other resources, establishes a priority for the process in multitasking systems, loads program binary code into memory, and initiates execution of the application program, which then interacts with the user and with hardware devices. When a process is terminated, any reusable resources are released and returned to the OS.

Starting a new process is a heavy job for the OS, which includes allocating memory, creating data structures, and coping code.

A thread is a path of execution within a process and the basic unit to which the OS allocates processor time. A process can be *single-threaded* or *multithreaded*. Theoretically, a thread can do anything a process can do. The essential difference between a thread and a process is the work that each one is used to accomplish. Threads are used for small tasks, whereas processes are used for more heavyweight tasks – basically the execution of applications. Therefore, a thread is often called a *lightweight* process.

Threads within the same process share the same address space, whereas different processes do not. Threads also share global and static variables, file descriptors, signal bookkeeping, code area, and heap. This allows threads to read from and write to the same data structures and variables and also facilitates communication between threads. Thus, threads use far less resources

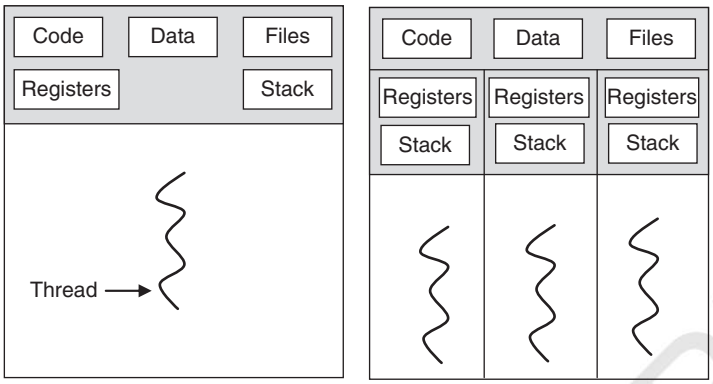

Figure 3.1 (a) Single-threaded process; (b) Multithreaded process.

compared to processes. However, each thread of the same process has its own thread status, program counter, registers, and stack, as illustrated in Figure 3.1.

A thread is the smallest unit of work managed independently by the scheduler of the OS. In RTOSs, the term *tasks* is often used for threads or single-threaded processes. For example, VxWorks and MicroC/OS-III are RTOSs that use the term tasks. In this book, processes (threads) and tasks are used interchangeably.

Processes may create other processes through appropriate system calls, such as *fork* or *spawn*. The process that does the creating is called the *parent* of the other process, which is called its *child*. Each process is assigned a unique integer identifier, call *process identifier*, or PID for short, when it is created. A process can be created with or without arguments. Generally, a parent process is in control of a child process. The parent can temporarily stop the child, cause it to be terminated, send it messages, look inside its memory, and so on. A child process may receive some amount of shared resources from its parent.

Processes may request their own termination by making the `exit()` system call. Processes may also be terminated by the system for a variety of reasons, such as the inability of the system to deliver necessary system resources, or in response to a *kill* command or other unhandled process interrupt. When a process terminates, all of its system resources are freed up, and open files are flushed and closed.

A process can be *suspended* for a variety of reasons. It can be *swapping* – the OS needs to release sufficient main memory to bring in a process that is ready to execute, or timing – a process that is executed periodically may be suspended while waiting for the next time interval. A parent process may also wish to suspend the execution of a child to examine or modify it or to coordinate the activity of various children.

Sometimes, processes need to communicate with each other while they are running. This is called *interprocess communication* (IPC). The OS provides mechanisms to support IPC. Files, sockets, message queues, pipes, named pipes, semaphores, shared memory, and message passing are typical IPC mechanisms.

3.1.2 Memory Management

Main memory is the most critical resource in a computer system in terms of speed at which programs run. The kernel of an OS is responsible for all system memory that is currently in use by programs. Entities in memory are data and instructions.

Each memory location has a *physical address*. In most computer architectures, memory is byte-addressable, meaning that data can be accessed 8 bits at a time, irrespective of the width of the data and address buses. Memory addresses are fixed-length sequences of digits. In general, only system software such as Basic Input/Output System (BIOS) and OS can address physical memory.

Most application programs do not have knowledge of physical addresses. Instead, they use logic addresses. A *logical address* is the address at which a memory location appears to reside from the perspective of an executing application program. A logical address may be different from the physical address due to the operation of an address translator or mapping function.

In a computer supporting *virtual memory*, the term physical address is used mostly to differentiate from a *virtual address*. In particular, in computers utilizing a *memory management unit* (MMU) to translate memory addresses, the virtual and physical addresses refer to an address before and after translation performed by the MMU, respectively. There are several reasons to use virtual memory. Among them is memory protection. If two or more processes are running at the same time and use direct addresses, a memory error in one process (e.g., reading a bad pointer) could destroy the memory being used by the other process, taking down multiple programs due to a single crash. The virtual memory technique, on the other hand, can ensure that each process is running in its own dedicated address space.

Before a program is loaded into memory by the *loader*, part of the OS, it must be converted into a *load module* and stored on disk. To create a load module, the source code is compiled by the compiler. The compiler produces an *object module*. An object module contains a header that records the size of each of the sections that follow, a machine code section that contains the executable instructions compiled by the compiler, an initialized data section that contains all data used by the program that require initialization, and the symbol table section that contains all external symbols used in the program. Some external symbols are defined in this object module and will be referred to by other object modules, and some are used in this object module and

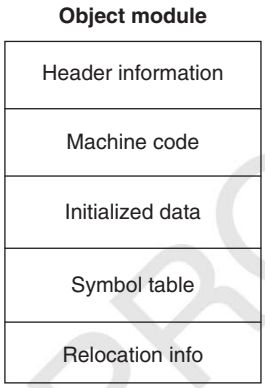

Figure 3.2 Structure of object module.

defined in other object modules. The relocation information is used by the *linker* to combine several object modules into a load module. Figure 3.2 shows the structure of object modules.

When a process is started, the OS allocates memory to the process and then loads the load module from disk into the memory allocated to the process. During the loading, the executable code and initialized data are copied into the process' memory from the load module. In addition, memory is also allocated for uninitialized data and runtime stack that is used to keep information about each procedure call. The loader has a default initial size for the stack. When the stack is filled up at runtime, extra space will be allocated to it, as long as the predefined maximum size is not exceeded.

Many programming languages support memory allocation while a program is running. It is done by the call of new in C++ and Java or malloc in C, for example. This memory comes from a large pool of memory called the *heap* or *free store*. At any given time, some parts of the heap are in use, while some are free and thus available for future allocation. Figure 3.3 illustrates the memory areas of a running process, in which the heap area is the memory allocated by the process at runtime.

To avoid loading big executable files into memory, modern OS provides two services: *dynamic loading* and *dynamic linking*. In dynamic loading, a routine (library or other binary module) of a program is not loaded until it is called by the program. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. Other routine methods or modules are loaded on request. Dynamic loading makes better memory space utilization, and unused routines are never loaded. Dynamic loading is useful when a large amount of code is needed to handle infrequently occurring cases.

In dynamic linking, libraries are linked at execution time. This is compared to *static linking*, in which libraries are linked at compile time, and thus, the resultant executable code is big. Dynamic linking refers to resolving

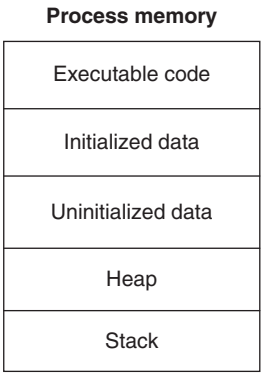

Figure 3.3 Memory areas of process.

symbols – associating their names with addresses or offsets – after compile time. It is particularly useful for libraries.

The simplest memory management technique is *single contiguous allocation*. That is, except the area reserved for the OS, all the memory is made available to a single application. This is the technique used in the MS-DOS system.

Another technique is *partitioned allocation*. It divides memory into multiple memory partitions; each partition can have only one process. The memory manager selects a free partition and assigns it to a process when it starts and deallocates it when it is finished. Some systems allow a partition to be *swapped* out to secondary storage to free additional memory. It is brought back into memory for continued execution later.

Paged allocation divides memory into fixed-size units called page frames and the program's *virtual address space* into pages of the same size. The hardware MMU maps pages to frames. The physical memory can be allocated on a page basis while the address space appears to be contiguous. Paging does not distinguish and protect programs and data separately.

Segmented memory allows programs and data to be broken up into logically independent address spaces and to aid sharing and protection. Segments are areas of memory that usually correspond to a logical grouping of information such as a code procedure or a data array. Segments require hardware support in the form of a segment table, which usually contains the physical address of the segment in memory, its size, and other data such as access protection bits and status (swapped in, swapped out, etc.).

As processes are loaded and removed from memory, the long, contiguous free memory space becomes fragmented into smaller and smaller contiguous pieces. Eventually, it may become impossible for the program to obtain large contiguous chunks of memory. The problem is called *fragmentation*. In general, smaller page size reduces fragmentation. The negative side is that it increases the page table size.

3.1.3 Interrupts Management

An *interrupt* is a signal from a device attached to a computer or from a running process within the computer, indicating an event that needs immediate attention. The processor responds by suspending its current activity, saving its state, and executing a function called an *interrupt handler* (also called *interrupt service routine*, ISR) to deal with the event.

Modern OSs are *interrupt-driven*. Virtually, all activities are initiated by the arrival of interrupts. Interrupt transfers control to the ISR, through the *interrupt vector*, which contains the addresses of all the service routines. Interrupt architecture must save the address of the interrupted instruction. Incoming interrupts are disabled while another interrupt is being processed. A system call is a software-generated interrupt caused either by an error or by a user request.

3.1.4 Multitasking

Real-world events may occur simultaneously. Multitasking refers to the capability of an OS that supports multiple independent programs running on the same computer. It is mainly achieved through time-sharing, which means that each program uses a share of the computer's time to execute. How to share processors' time among multiple tasks is addressed by *schedulers*, which follow scheduling algorithms to decide when to execute which task on which processor.

Each task has a *context*, which is the data indicating its execution state and stored in the *task control block* (TCB), a data structure that contains all the information that is pertinent to the execution of the task. When a scheduler switches a task out of the CPU, its context has to be stored; when the task is selected to run again, the task's context is restored so that the task can be executed from the last interrupted point. The process of storing and restoring the context of a task during a task switch is called a *context switch*, which is illustrated in Figure 3.4.

Context switches are the overhead of multitasking. They are usually computationally intensive. Context switch optimization is one of the tasks of OS design. This is particularly the case in RTOS design.

3.1.5 File System Management

Files are the fundamental abstraction of secondary storage devices. Each file is a named collection of data stored in a device. An important component of an OS is the *file system*, which provides capabilities of file management, auxiliary storage management, file access control, and integrity assurance.

File management is concerned with providing the mechanisms for files to be stored, referenced, shared, and secured. When a file is created, the file system

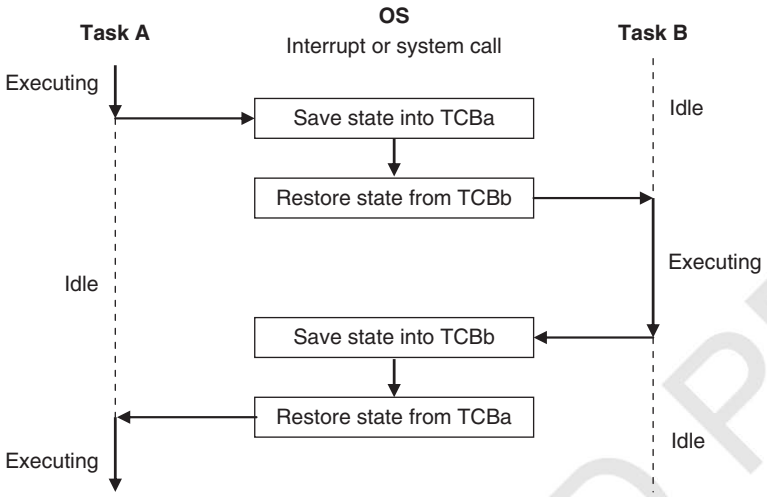

Figure 3.4 Context switch between tasks A and B.

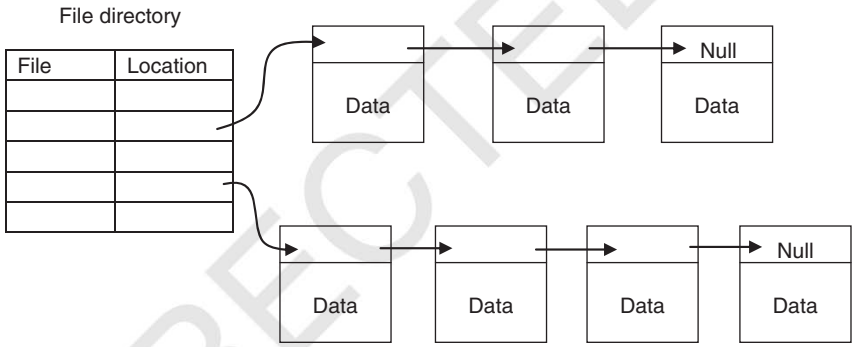

Figure 3.5 The block chaining scheme in file storage.

allocates an initial space for the data. Subsequent incremental allocations follow as the file grows. When a file is deleted or its size is shrunk, the space that is freed up is considered available for use by other files. This creates alternating used and unused areas of various sizes. When a file is created and there is not an area of contiguous space available for its initial allocation, the space must be assigned in *fragments*. Because files do tend to grow or shrink over time, and because users rarely know in advance how large their files will be, it makes sense to adopt noncontiguous storage allocation schemes. Figure 3.5 illustrates the block chaining scheme. The initial address of storage of a file is identified by its file name.

Typically, files on a computer are organized into directories, which constitute a hierarchical system of tree structure.

A file system typically stores necessary bookkeeping information for each file, including the size of data contained in the file, the time the file was last modified, its owner user ID and group ID, and its access permissions.

The file system also provides a spectrum of commands to read and write the contents of a file, to set the file read/write position, to set and use the protection mechanism, to change the ownership, to list files in a directory, and to remove a file.

File access control can be realized using a two-dimensional matrix that lists all users and all files in the system. The entry at index (i, j) of the matrix specifies if user i is allowed to access file j. In a system that has a large number of users and contains a large number of files, this matrix would be very large and very sparse.

A scheme that requires much less space is to control access to various user classes. *Role-based access control* (RBAC) is an access control method where access to data is performed by authorized users. RBAC assigns users to specific roles, and permissions are granted to each role based on the user's job requirements. Users can be assigned a number of roles in order to conduct day-to-day tasks. For example, a user may need to have a developer role as well as an analyst role. Each role would define the permissions that are needed to access different objects.

3.1.6 I/O Management

Modern computers interact with a wide range of I/O devices. Keyboards, mice, printers, disk drives, USB drives, monitors, networking adapters, and audio systems are among the most common ones. One purpose of an OS is to hide peculiarities of hardware I/O devices from the user.

In *memory-mapped I/O*, each I/O device occupies some locations in the I/O address space. Communication between the I/O device and the processor is enabled through physical memory locations in the I/O address space. By reading from or writing to those addresses, the processor gets information from or sends commands to I/O devices.

Most systems use *device controllers*. A device controller is primarily an interface unit. The OS communicates with the I/O device through the device controller. Nearly all device controllers have *direct memory access* (DMA) capability, meaning that they can directly access the memory in the system, without the intervention by the processor. This frees up the processor of the burden of data transfer from and to I/O devices.

Interrupts allow devices to notify the processor when they have data to transfer or when an operation is complete, allowing the processor to perform other duties when no I/O transfers need its immediate attention. The processor has the interrupt request line that it senses after executing every instruction. When a device controller raises an interrupt by asserting a signal on the

interrupt request line, the processor catches it and saves the state and then transfers control to the interrupt handler. The interrupt handler determines the cause of the interrupt, performs necessary processing, and executes a *return from interrupt* instruction to return control to the processor.

I/O operations often have high latencies. Most of this latency is due to the slow speed of peripheral devices. For example, information cannot be read from or written to a hard disk until the spinning of the disk brings the target sectors directly under the read/write head. The latency can be alleviated by having one or more input and output *buffers* associated with each device.

3.2 Characteristics of RTOS Kernels

Although a general-purpose OS provides a rich set of services that are also needed by real-time systems, it takes too much space and contains too many functions that may not be necessary for a specific real-time application. Moreover, it is not configurable, and its inherent timing uncertainty offers no guarantee to system response time. Therefore, a general-purpose OS is not suitable for real-time embedded systems.

There are three key requirements of RTOS design. Firstly, the timing behavior of the OS must be predictable. For all services provided by the OS, the upper bound on the execution time must be known. Some of these services include OS calls and interrupt handling. Secondly, the OS must manage the timing and scheduling, and the scheduler must be aware of task deadlines. Thirdly, the OS must be fast. For example, the overhead of context switch should be short. A fast RTOS helps take care of soft real-time constraints of a system as well as guarantees hard deadlines.

As illustrated in Figure 3.6, an RTOS generally contains a *real-time kernel* and other higher-level services such as file management, protocol stacks, a Graphical User Interface (GUI), and other components. Most additional services revolve around I/O devices. A real-time kernel is software that manages the time and resources of a microprocessor or microcontroller and provides indispensable services such as task scheduling and interrupt handling to applications. Figure 3.7 shows a general structure of a microkernel. In embedded systems, a small amount of code called *board support package* (BSP) is implemented for a given board that conforms to a given OS. It is commonly built with a bootloader that contains the minimal device support to load the OS and device drivers for all the devices on the board.

In the rest of this section, we introduce some most important real-time services that are specified in POSIX 1.b for RTOS kernels.

3.2.1 Clocks and Timers

Most embedded systems must keep track of the passage of time. The length of time is represented by the number of system ticks in most RTOS kernels.

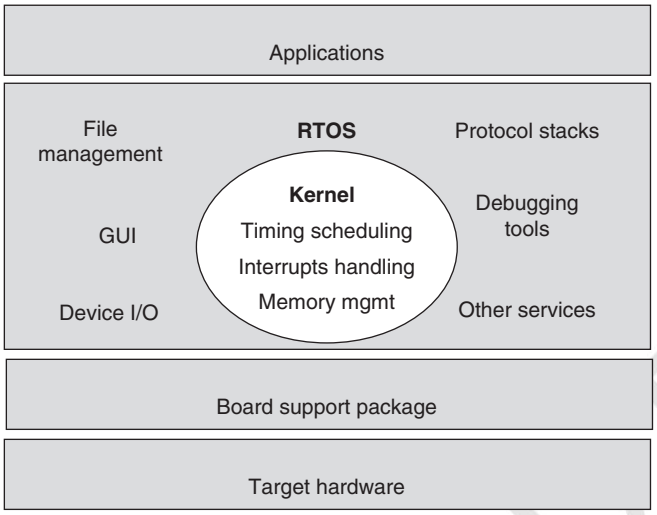

Figure 3.6 A high-level view of RTOS.

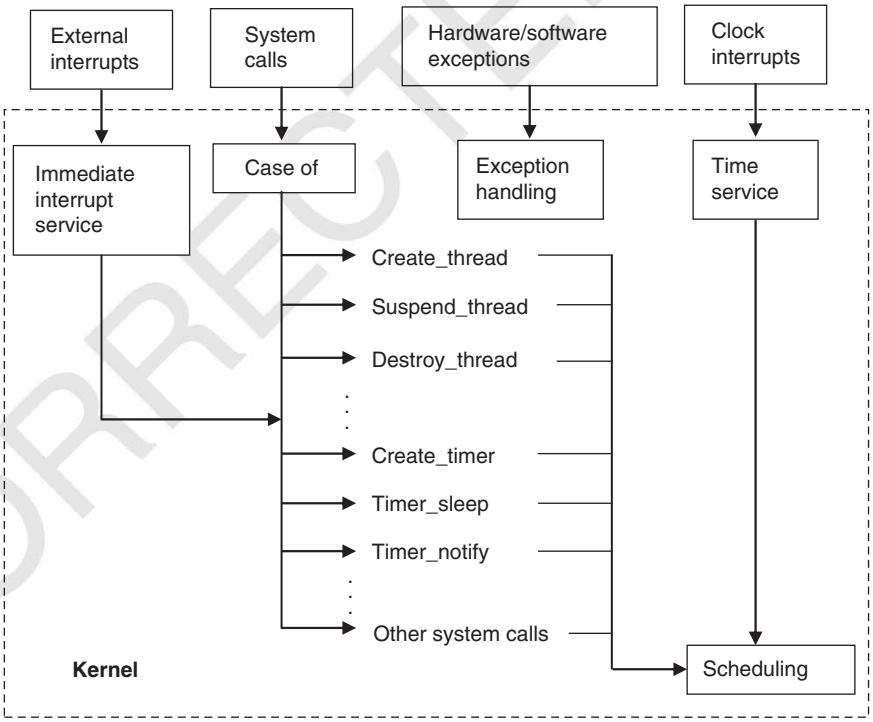

Figure 3.7 Structure of a microkernel.

The RTOS works by setting up a hardware timer to interrupt periodically, say, every millisecond, and bases all timings on the interrupts. For example, if in a task you call the *taskDelay* function in VxWorks with a parameter of 20, then the task will block until the timer interrupts for 20 times. In the POSIX standard, each tick is equal to 10 milliseconds, and in each second, there are 100 ticks. Some RTOS kernels, such as VxWorks, define routines that allow the user to set and get the value of system tick. The timer is often called a *heartbeat timer*, and the interrupt is also called *clock interrupt*.

At each clock interrupt, an ISR increments tick count, checks to see if it is now time to unblock or wake up a task. If this is the case, it calls the scheduler to do the scheduling again.

Based on the system tick, an RTOS kernel allows you to call functions of your choice after a given number of system ticks, such as the *taskDelay* function in VxWorks. Depending upon the RTOS, your function may be directly called from the timer ISR. There are also other timing services. For example, most RTOS kernels allow developers to limit how long a task will wait for a message from a queue or a mailbox and how long a task will wait for a semaphore, and so on.

Timers improve the determinism of real-time applications. Timers allow applications to set up events at predefined intervals or time. POSIX specified several timer-related routines, including the following:

- `timer_create`() – allocate a timer using the specified clock for a timing base.
- `timer_delete`() – remove a previously created timer.
- `timer_gettime`() – get the remaining time before expiration and the reload value.
- `timer_getoverrun`() – return the timer expiration overrun.
- `timer_settime`() – set the time until the next expiration and arm timer.
- `nanosleep`() – suspend the current task until the time interval elapses.

3.2.2 Priority Scheduling

Because real-time tasks have deadlines, being soft or hard, all tasks are not equal in terms of the urgency of getting executed. Tasks with shorter deadlines should be scheduled for execution sooner than those with longer deadlines. Therefore, tasks are typically prioritized in an RTOS. Moreover, if a higher priority task is released while the processor is serving a lower priority task, the RTOS should temporarily suspend the lower priority task and immediately schedule the higher priority on the processor for execution, to ensure that the higher priority task is executed before its deadline. This process is called *preemption*. Task scheduling for real-time applications is typically priority-based and preemptive. Examples are earliest deadline first (EDF) scheduling and rate monotonic (RM) scheduling. Scheduling algorithms that do not take task

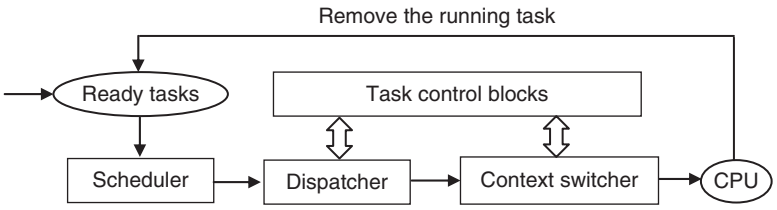

Figure 3.8 Priority scheduling.

priorities into account, such as first-in-first-service and round-robin, are not suitable for real-time systems.

In priority-driven preemptive scheduling, the preemptive scheduler has a clock interrupt task that provides the scheduler with options to switch after the task has had a given period to execute – the time slice. This scheduling system has the advantage of making sure that no task hogs the processor for any time longer than the time slice.

As shown in Figure 3.8, an important component involved in scheduling is the *dispatcher*, which is the module that gives control of the CPU to the task selected by the scheduler. It receives control in kernel mode as the result of an interrupt or system call. It is responsible for performing context switches. The dispatcher should be as fast as possible, since it is invoked during every task switch. During the context switches, the processor is virtually idle for a fraction of time; thus, unnecessary context switches should be avoided.

The key to the performance of priority-driven scheduling is in choosing priorities for tasks. Priority-driven scheduling may cause low-priority tasks to starve and miss their deadlines. In the next two chapters, several well-known scheduling algorithms and resource access control protocols will be discussed.

3.2.3 Intertask Communication and Resource Sharing

In an RTOS, a task cannot call another task. Instead, tasks exchange information through message passing or memory sharing and coordinate their execution and access to shared data using real-time signals, mutex, or semaphore objects.

3.2.3.1 Real-Time Signals

Signals are similar to software interrupts. In an RTOS, a signal is automatically sent to the parent when a child process terminates. Signals are also used for many other synchronous and asynchronous notifications, such as waking a process when a wait call is performed and informing a process that it has issued a memory violation.

POSIX extended the signal generation and delivery to improve the real-time capabilities. Signals take an important role in real-time systems as the

way to inform the processes of the occurrence of asynchronous events such as high-resolution timer expiration, fast interprocess message arrival, asynchronous I/O completion, and explicit signal delivery.

3.2.3.2 Semaphores

Semaphores are counters used for controlling access to resources shared among processes or threads. The value of a semaphore is the number of units of the resource that are currently available. There are two basic operations on semaphores. One is to atomically increment the counter. The other is to wait until the counter is nonnull and atomically decrement it. A semaphore tracks only how many resources are free; it does not keep track of which of the resources are free.

A binary semaphore acts similarly to a *mutex*, which is used in the case when a resource can only be used by at most one task at any time.

3.2.3.3 Message Passing

In addition to signals and semaphores, tasks can share data by sending messages in an organized *message passing* scheme. Message passing is much more useful for information transfer. It can also be used just for synchronization. Message passing often coexists with shared memory communication. Message contents can be anything that is mutually comprehensible between the two parties in communication. Two basic operations are *send* and *receive*.

Message passing can be *direct* or *indirect*. In direct message passing, each process wanting to communicate must explicitly name the recipient or sender of the communication. In indirect message passing, messages are sent to and received from *mailboxes* or *ports*. Two processes can communicate in this way only if they have a shared mailbox.

Message passing can also be *synchronous* or *asynchronous*. In synchronous message passing, the sender process is blocked until the message primitive has been performed. In asynchronous message passing, the sender process immediately gets control back.

3.2.3.4 Shared Memory

Shared memory is a method that an RTOS uses to map common physical space into independent process-specific virtual space. Shared memory is commonly used to share information (resources) between different processes or threads. Shared memory must be accessed exclusively. Therefore, it is necessary to use mutex or semaphores to protect the memory area. The code segment in a task that accesses the shared data is called a *critical section*. Figure 3.9 shows that two tasks share a memory region.

A side effect in using shared memory is that it may cause *priority inversion*, a situation that a low-priority task is running while a high-priority task is waiting. More details of priority inversion will be discussed in Chapter 5.

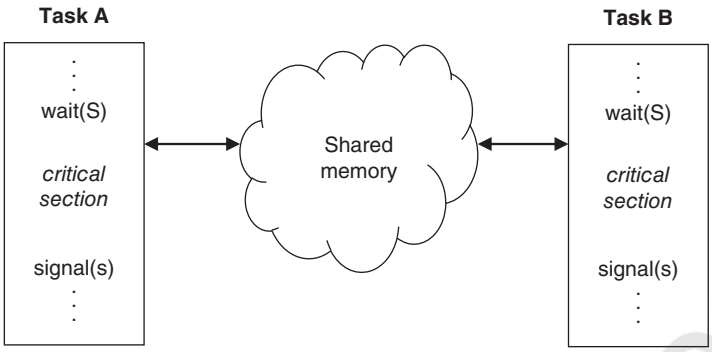

Figure 3.9 Shared memory and critical section.

3.2.4 Asynchronous I/O

There are two types of I/O synchronization: synchronous I/O and asynchronous I/O. In synchronous I/O, when a user task requests the kernel for I/O operation and the requested is granted, the system will wait until the operation is completed before it can process other tasks. Synchronous I/O is desirable when the I/O operation is fast. It is also easy to implement.

An RTOS supports the overlap of application processing and application-initiated I/O operations. This is the RTOS service of asynchronous I/O. In asynchronous I/O, after a task requests I/O operation, while this task is waiting for I/O to complete, other tasks that do not depend on the I/O results will be scheduled for execution. Meanwhile, tasks that depend on the I/O having completed are blocked. Asynchronous I/O is used to improve throughput, latency, and/or responsiveness.

Figure 3.10 illustrates the idea of synchronous I/O and asynchronous I/O.

3.2.5 Memory Locking

Memory locking is a real-time capability specified by POSIX that is intended for a process to avoid the latency of fetching a page of memory. It is achieved by locking the memory so that the page is *memory-resident*, that is, it remains in the main memory. This allows fine-grained control of which part of the application must stay in physical memory to reduce the overhead associated with transferring date between memory and disk. For example, memory locking can be used to keep in memory a thread that monitors a critical process that requires immediate attention.

When the process exits, the locked memory is automatically unlocked. Locked memory can also be unlocked explicitly. POSIX, for example, defined `mlock()` and `munlock()` functions to lock and unlock memory. The `munlock` function unlocks the specified address range regardless of the number of times the `mlock` function was called. In other words, you can lock

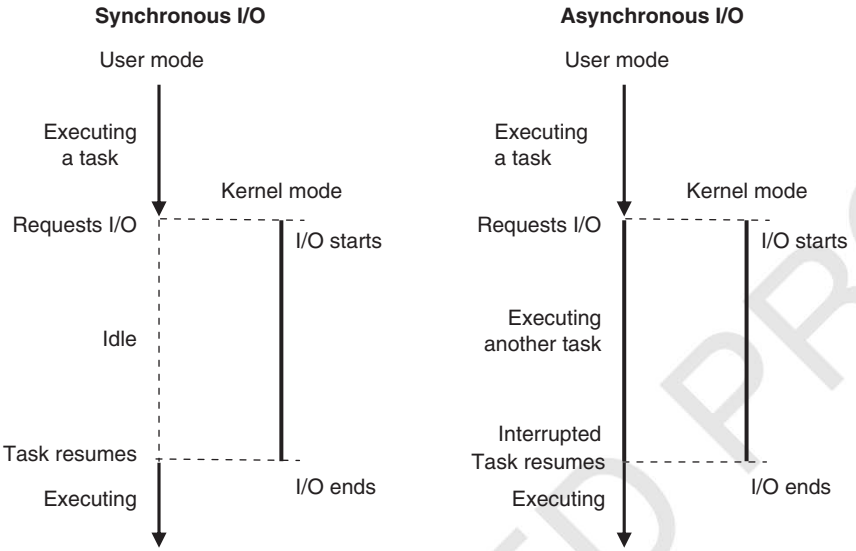

Figure 3.10 Synchronous I/O versus asynchronous I/O.

address ranges over multiple calls to the `mlock` function, but the locks can be removed with a single call to the `munlock` function. In other words, memory locks don't stack.

More than one process can lock the same or overlapping region, and in that case, the memory region remains locked until all the processes have unlocked it.

3.3 RTOS Examples

3.3.1 LynxOS

The LynxOS RTOS is a Unix-like RTOS from Lynx Software Technologies. LynxOS is a deterministic, hard RTOS that provides POSIX-conformant APIs in a small-footprint embedded kernel. It features predictable worst-case response time, preemptive scheduling, real-time priorities, ROMable kernel, and memory locking. LynxOS provides symmetric multiprocessing support to fully take advantage of multicore/multithreaded processors. LynxOS 7.0, the latest version, includes new tool chains, debuggers, and cross-development host support.

The LynxOS RTOS is designed from the ground up for conformance to open-system interfaces. It leverages existing Linux, UNIX, and POSIX programming talent for embedded real-time projects. Real-time system development time is saved, and programmers are able to be more productive using familiar methodologies as opposed to learning proprietary methods.

LynxOS is mostly used in real-time embedded systems, in applications for avionics, aerospace, the military, industrial process control, and telecommunications. LynxOS has already been used in millions of devices.

3.3.2 OSE

OSE is an acronym for the operating system embedded. It is a real-time embedded OS created by the Swedish information technology company ENEA AB. OSE uses signals in the form of messages passed to and from processes in the system. Messages are stored in a queue attached to each process. A link handler mechanism allows signals to be passed between processes on separate machines, over a variety of transports. The OSE signaling mechanism formed the basis of an open-source interprocess kernel design.

The Enea RTOS family shares a high-level programming model and an intuitive API to simplify programming. It consists of two products, each optimized for a specific class of applications:

- Enea OSE is a robust and high-performance RTOS optimized for multicore, distributed, and fault-tolerant systems.
- Enea OSEck is a compact and multicore DSP-optimized version of ENEA's full-featured OSE RTOS.

OSE supports many mainly 32-bit processors, such as those in ARM, PowerPC, and MIPS families.

3.3.3 QNX

The QNX Neutrino RTOS is a full-featured and robust OS is developed by QNX Software Systems Limited, a subsidiary of BlackBerry. QNX products are designed for embedded systems running on various platforms, including ARM and x86, and a host of boards implemented in virtually every type of embedded environment.

As a microkernel-based OS, QNX is based on the idea of running most of the OS kernel in the form of a number of small tasks, known as servers. This differs from the more traditional monolithic kernel, in which the OS kernel is a single, very large program composed of a huge number of components with special abilities. In the case of QNX, the use of a microkernel allows developers to turn off any functionality they do not require without having to change the OS itself; instead, those servers will simply not run.

The BlackBerry PlayBook tablet computer designed by BlackBerry uses a version of QNX as the primary OS. The BlackBerry line of devices running the BlackBerry 10 OS are also based on QNX.

3.3.4 VxWorks

VxWorks is an RTOS developed as proprietary software by Wind River, a subsidiary company of Intel providing embedded system software, which

comprises runtime software, industry-specific software solutions, simulation technology, development tools, and middleware. As other RTOS products, VxWorks is designed for use in embedded systems requiring real-time and deterministic performance.

VxWorks supports Intel, MIPS, PowerPC, SH-4, and ARM architectures. The VxWorks Core Platform consists of a set of runtime components and development tools. VxWorks core development tools are compilers such as Diab, GNU, and Intel C++ Compiler (ICC) and its build and configuration tools. The system also includes productivity tools such as its Workbench development suite and Intel tools and development support tools for asset tracking and host support. Cross-compiling is used with VxWorks. Development is achieved on a "host" system where an integrated development environment (IDE), including the editor, compiler toolchain, debugger, and emulator, can be used. Software is then compiled to run on the "target" system. This allows the developer to work with powerful development tools while targeting more limited hardware.

VxWorks is used by products over a wide range of market areas: aerospace and defenses, automotive, industrial such as robots, consumer electronics, medical area, and networking. Several notable products also use VxWorks as the onboard OS. Examples in the area of spacecraft are the Mars Reconnaissance Orbiter, the Phoenix Mars Lander, the Deep Impact Space Probe, and the Mars Pathfinder.

3.3.5 Windows Embedded Compact

Formerly known as Windows CE, Windows Embedded Compact is an OS subfamily developed by Microsoft as part of its Windows Embedded family of products. It is a small-footprint RTOS and optimized for devices that have minimal memory, such as industrial controllers, communication hubs, and consumer electronics devices such as digital cameras, GPS systems, and also automotive infotainment systems. It supports x86, SH (automotive only), and ARM.

Exercises

1 What is the kernel of an operating system? In which mode does it run?

2 What are the two approaches that a user process interacts with the operating system? Discuss the merits of each approach.

3 What is the difference between a program and a process? What is the difference between a process and a thread?

4 Should terminating a process also terminate all its children? Give an example when this is a good idea and another example when this is a bad idea.

5 Should the open files of a process be automatically closed when the process exits?

6 What is the difference between an object module and a load module?

7 Discuss the merits of each memory allocation techniques introduced in this chapter.

8 What do we mean when we say an OS is interrupt-driven? What does the processor do when an interrupt occurs?

9 What is a context switch and when does it occur?

10 Does a file have to be stored in a contiguous storage region in a disk?

11 What are the benefits of using memory-mapped I/O?

12 Why can a general-purpose OS not meet the requirements of real-time systems?

13 How does memory fragmentation form? Name one approach to control it.

14 What are the basic functions of an RTOS kernel?

15 How does an RTOS keep track of the passage of time?

16 Why is it necessary to use priority-based scheduling in a real-time application?

17 What are the general approaches that different tasks communicate with each other and perform synchronization of their actions in access to shared resources?

18 Compare synchronous I/O and asynchronous I/O and list their advantages and disadvantages.

19 How does the memory locking technique improve the performance of a real-time system?

Suggestions for Reading

Many textbooks are available that provide thorough and in-depth discussion on general-purpose OSs. Examples are [1–3]. Cooling [4] provides a great overview of the fundamentals of RTOS for embedded programming without being specific to any one vendor. POSIX-specified real-time facilities are described in Ref. [5]. More information regarding the commercial RTOS products can be found on their official websites.

References

1 Doeppner, T. (2011) *Operating Systems in Depth*, Wiley.
2 McHoes, A.M. and Flynn, I.M. (2011) *Understand Operating Systems*, 6th edn, Course Technology Cengage Learning.
3 Stallings, W. (2014) Operating Systems*:* Internals and Design Principles, 8th edn, Pearson.
4 Cooling, J. (2013) *Real-Time Operating Systems*, Kindle edn, Lindentree Associates.
5 Gallmeister, B. (1995) POSIX 4.0*:* Programming for the Real World, O'Reilly & Associates, Inc..

URLs

LynxOS, http://www.lynxos.com/
VxWorks, http://www.windriver.com/products/vxworks/
Windows CE, https://www.microsoft.com/windowsembedded/en-us/windows-embedded-compact-7.aspx
QNX, http://www.qnx.com/content/qnx/en.html
OSE, http://www.enea.com/ose