Unit I – Real Time Operating Systems

- Brief History of OS
- Defining RTOS
- The Scheduler, Objects, Services
- Characteristics of RTOS
- Defining a Task, Tasks States and Scheduling
- Task Operations, and Structure
- Multiprocessing and Multi-Tasking
- Cooperative Multi-Tasking
- Non Preemptive Multitasking
- Preemptive Multi-Tasking
- Synchronization, Communication and Concurrency

Operating System (OS) :

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.

An operating system (OS) is the program that, after being initially loaded into the computer by a boot program, manages all the other programs in a computer. The other programs are called *applications* or application programs. The application programs make use of the operating system by making requests for services through a defined application program interface (API). In addition, users can interact directly with the operating system through a user interface such as a command line or a graphical user interface (GUI).

- Some popular Operating Systems include Linux, Windows, OS X, VMS, OS/400, AIX, z/OS, etc.
- Mobile operating systems include Apple iOS, Google Android, BlackBerry OS and Windows 10 Mobile.

Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

All major computer platforms (hardware and software) require and sometimes include an operating system, and operating systems must be developed with different features to meet the specific needs of various applications.

An **embedded operating system** is specialized for use in the computers built into larger systems, such as cars, traffic lights, digital televisions, ATMs, airplane controls, point of sale (POS) terminals, digital cameras, GPS navigation systems, elevators, digital media receivers and smart meters.

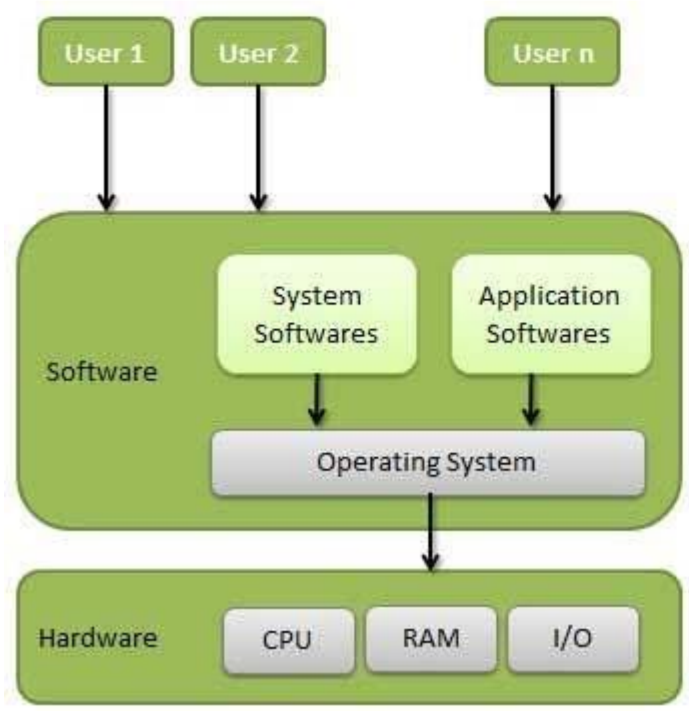


Fig. 1 Operating System Architecture

Embedded system (ES):

An embedded system is a combination of 3 things:

a. Hardware
b. Software
c. Mechanical Components

And it is supposed to do one specific task only.

The hardware & mechanical components will consist all the physically visible things that are used for input, output, etc.

An embedded system will always have a chip (either microprocessor or microcontroller) that has the code or software which drives the system.

Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, scientific experiments, medical image systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

Hard real-time systems

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

Soft real-time systems

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects likes undersea exploration and planetary rovers, etc.

A real-time operating system (RTOS) is key to many embedded systems today and, provides a software platform upon which to build applications. Not all embedded systems, however, are designed with an RTOS. Some embedded systems with relatively simple hardware or a small amount of software application code might not require an RTOS. Many embedded systems, however, with moderate-to-large software applications require some form of scheduling, and these systems require an RTOS.

A Brief History of Operating Systems

In the early days of computing, developers created software applications that included low-level machine code to initialize and interact with the system's hardware directly. This tight integration between the software and hardware resulted in non-portable applications. A small change in the hardware might result in rewriting much of the application itself. Obviously, these systems were difficult and costly to maintain.

As the software industry progressed, operating systems that provided the basic software foundation for computing systems evolved and facilitated the abstraction of the underlying hardware from the application code. In addition, the evolution of operating systems helped shift the design of software applications from large, monolithic applications to more modular, interconnected applications that could run on top of the operating system environment.

Over the years, many versions of operating systems evolved. These ranged from general-purpose operating systems (GPOS), such as UNIX and Microsoft Windows, to smaller and more compact real-time operating systems, such as VxWorks.

In the 60s and 70s, when mid-sized and mainframe computing was in its prime, UNIX was developed to facilitate multi-user access to expensive, limited-availability computing systems. UNIX allowed many users performing a variety of tasks to share these large and costly computers. multi-user access was very efficient: one user could print files, for example, while another wrote programs. Eventually, UNIX was ported to all types of machines, from microcomputers to supercomputers.

In the 80s, Microsoft introduced the Windows operating system, which emphasized the personal computing environment. Targeted for residential and business users interacting with PCs through a graphical user interface, the Microsoft Windows operating system helped drive the personal-computing era.

Later in the decade, momentum started building for the next generation of computing: the post-PC, embedded-computing era. To meet the needs of embedded computing, commercial RTOSes, such as VxWorks, were developed. Although some functional similarities exist between RTOSes and GPOSes, many important differences occur as well. These differences help explain why RTOSes are better suited for real-time embedded systems.

Some core functional similarities between a typical RTOS and GPOS include:
- some level of multitasking,
- software and hardware resource management,
- provision of underlying OS services to applications, and
- abstracting the hardware from the software application.

On the other hand, some key functional differences that set RTOSes apart from GPOSes include:
- better reliability in embedded application contexts,

- the ability to scale up or down to meet application needs,
- faster performance,
- reduced memory requirements,
- scheduling policies tailored for real-time embedded systems,
- support for diskless embedded systems by allowing executables to boot and run from ROM or RAM, and
- better portability to different hardware platforms.

Today, GPOSes target general-purpose computing and run predominantly on systems such as personal computers, workstations, and mainframes. In some cases, GPOSes run on embedded devices that have ample memory and very soft real-time requirements. GPOSes typically require a lot more memory, however, and are not well suited to real-time embedded devices with limited memory and high performance requirements.

RTOSes, on the other hand, can meet these requirements. They are reliable, compact, and scalable, and they perform well in real-time embedded systems. In addition, RTOSes can be easily tailored to use only those components required for a particular application.

Defining an RTOS

A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code. Application code designed on an RTOS can be quite diverse, ranging from a simple application for a digital stopwatch to a much more complex application for aircraft navigation. Good RTOSes, therefore, are scalable in order to meet different sets of requirements for different applications.

For example, in some applications, an RTOS comprises only a kernel, which is the core supervisory software that provides minimal logic, scheduling, and resource-management algorithms. Every RTOS has a kernel. On the other hand, an RTOS can be a combination of various modules, including the kernel, a file system, networking protocol stacks, and other components required for a particular application, as illustrated at a high level in Figure 2.

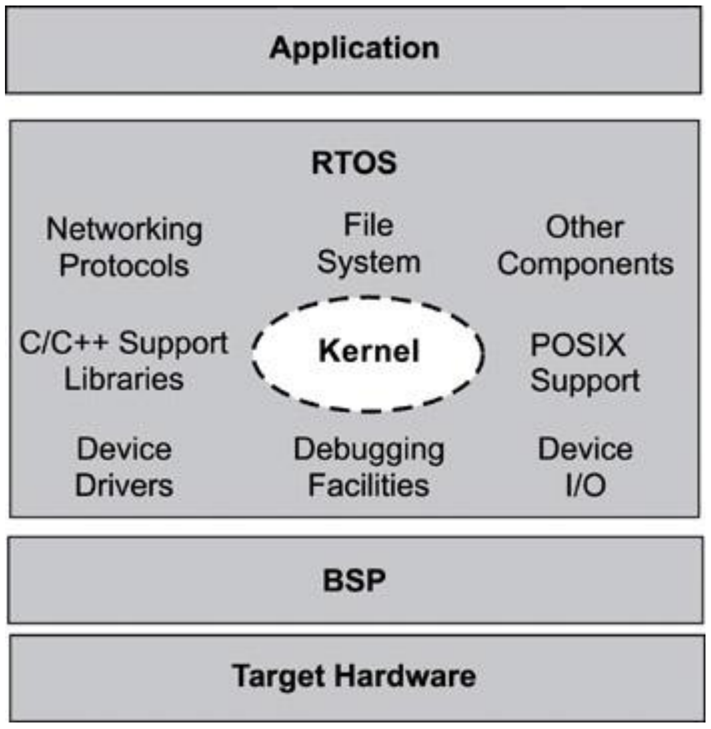


Figure 2: High-level view of an RTOS, its kernel, and other components found in embedded systems.

Most RTOS kernels contain the following components:

•**Scheduler**-is contained within each kernel and follows a set of algorithms that determines which task
executes when. Some common examples of scheduling algorithms include round-robin and

preemptive scheduling.

•**Objects**-are special kernel constructs that help developers create applications for real-time embedded
systems. Common kernel objects include tasks, semaphores, and message queues.

•**Services**-are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management.

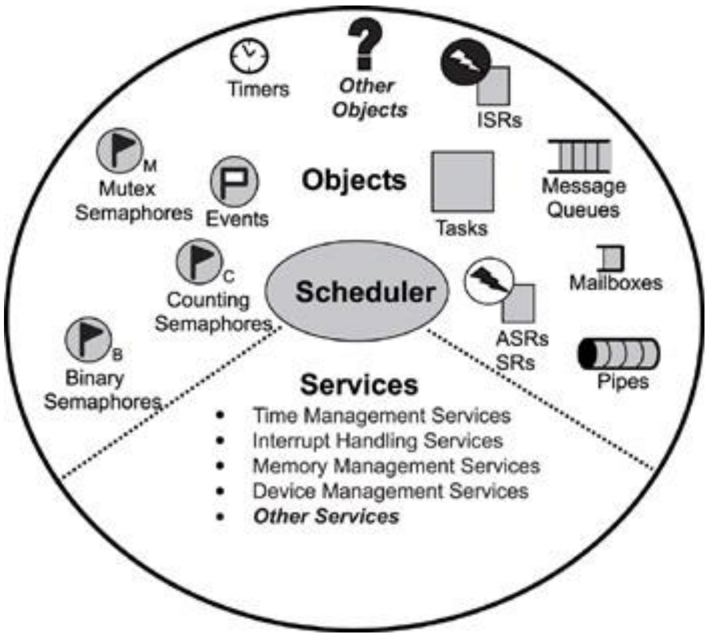


Figure 3: Common components in an RTOS kernel that including objects, the scheduler, and some services.

The Scheduler

The scheduler is at the heart of every kernel. A scheduler provides the algorithms needed to determine which task executes when. To understand how scheduling works, this section describes the following topics:

- schedulable entities
- multitasking
- context switching
- dispatcher
- scheduling algorithms.

Schedulable Entities

A schedulable entity is a kernel object that can compete for execution time on a system, based on a predefined scheduling algorithm. Tasks and processes are all examples of schedulable entities found in most kernels.

A task is an independent thread of execution that contains a sequence of independently schedulable instructions. Some kernels provide another type of a schedulable object called a process. Processes are similar to tasks in that they can independently compete for CPU execution time. Processes differ from tasks in that they provide better memory protection features, at the expense of performance and memory overhead.

Multitasking

Multitasking is the ability of the operating system to handle multiple activities within set deadlines. A real-time kernel might have multiple tasks that it has to schedule to run. One such multitasking scenario is illustrated in Figure 4.

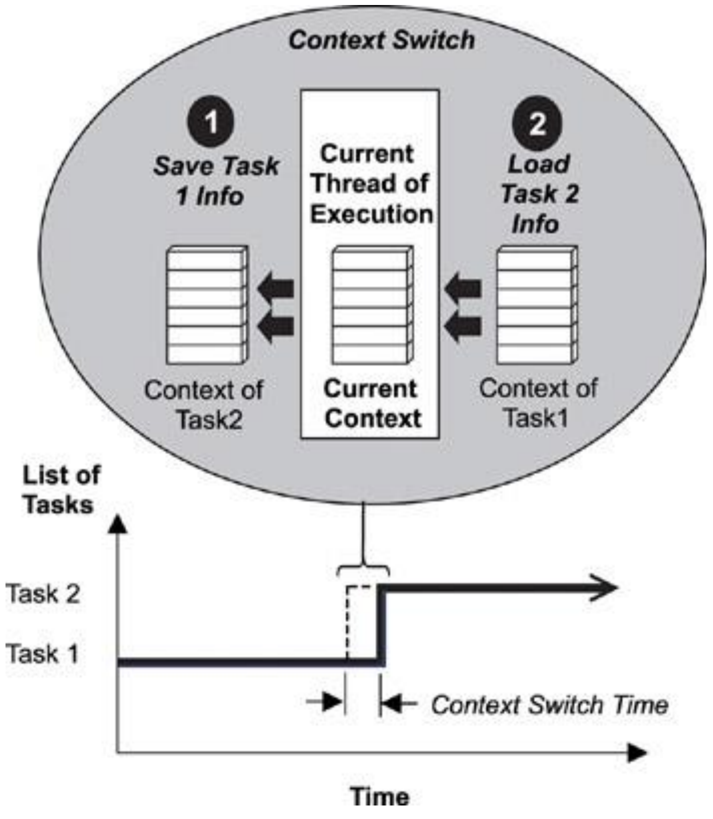


Figure 4: Multitasking using a context switch.

In this scenario, the kernel multitasks in such a way that many threads of execution appear to be running concurrently; however, the kernel is actually interleaving executions sequentially, based on a preset scheduling algorithm The scheduler must ensure that the appropriate task runs at the right time.

An important point to note here is that the tasks follow the kernels scheduling algorithm, while interrupt service routines (ISR) are triggered to run because of hardware interrupts and their established priorities. As the number of tasks to schedule increases, so do CPU performance requirements. This fact is due to increased switching between the contexts of the different threads of execution.

The Context Switch

Each task has its own context, which is the state of the CPU registers required each time it is scheduled to run. A context switch occurs when the scheduler switches from one task to another.

Every time a new task is created, the kernel also creates and maintains an associated task control block (TCB).TCBs are system data structures that the kernel uses to maintain task-specific information. TCBs contain everything a kernel needs to know about a particular task. When a task is running, its context is highly dynamic. This dynamic context is maintained in the TCB. When the task is not running, its context is frozen within the TCB, to be restored the next time the task runs. A typical context switch scenario is illustrated in Figure 4.

As shown in Figure 4, when the kernels scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps:

1.The kernel saves task 1 s context information in its TCB.

2.It loads task 2s context information from its TCB, which becomes the current thread of execution.

3.The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch.

The time it takes for the scheduler to switch from one task to another is the context switch time. It is relatively insignificant compared to most operations that a task performs. If an applications design includes frequent context switching, however, the application can incur unnecessary performance overhead. Therefore, design applications in a way that does not involve excess context switching.

Every time an application makes a system call, the scheduler has an opportunity to determine if it needs to switch contexts. When the scheduler determines a context switch is necessary, it relies on an associated module, called the dispatcher, to make that switch happen.

The Dispatcher

The dispatcher is the part of the scheduler that performs context switching and changes the flow of execution. At any time an RTOS is running, the flow of execution, also known as flow of control, is passing through one of three areas: through an application task, through an ISR, or through the kernel.

When a task or ISR makes a system call, the flow of control passes to the kernel to execute one of the system routines provided by the kernel. When it is time to leave the kernel, the dispatcher is responsible for passing control to one of the tasks in the user s application. It will not necessarily be the same task that made the system call. It is the scheduling algorithms of the scheduler that determines which task executes next. It is the dispatcher that does the actual work of context switching and passing execution control.

Depending on how the kernel is first entered, dispatching can happen differently. When a task makes system calls, the dispatcher is used to exit the kernel after every system call completes. In this case, the dispatcher is used on a call-by-call basis so that it can coordinate task-state transitions that any of the system calls might have caused. (One or more tasks may have become ready to run, for example.)

On the other hand, if an ISR makes system calls, the dispatcher is bypassed until the ISR fully completes its execution. This process is true even if some resources have been freed that would normally trigger a context switch between tasks. These context switches do not take place because the ISR must complete without being interrupted by tasks. After the ISR completes execution, the kernel exits through the dispatcher so that it can then dispatch the correct task.

Scheduling Algorithms

As mentioned earlier, the scheduler determines which task runs by following a scheduling algorithm (also known as scheduling policy). Most kernels today support two common scheduling algorithms:

- preemptive priority-based scheduling, and
- round-robin scheduling.

Preemptive Priority-Based Scheduling

Of the two scheduling algorithms introduced here, most real-time kernels use preemptive priority-based scheduling by default. As shown in Figure 5.with this type of scheduling, the task that gets to run at any point is the task with the highest priority among all other tasks ready to run in the system.

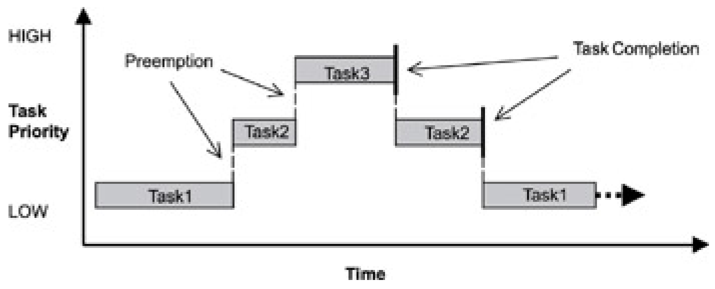


Figure 5: Preemptive priority-based scheduling.

Real-time kernels generally support 256 priority levels, in which 0 is the highest and 255 the lowest. Some kernels appoint the priorities in reverse order, where 255 is the highest and 0 the lowest.

With a preemptive priority-based scheduler, each task has a priority, and the highest-priority task runs first. If a task with a priority higher than the current task becomes ready to run, the kernel immediately saves the current tasks context in its TCB and switches to the higher-priority task. As shown in Figure 5.task 1 is preempted by higher-priority task 2, which is then preempted by task 3. When task 3 completes, task 2 resumes; likewise, when task 2 completes, task 1 resumes.

Although tasks are assigned a priority when they are created, a tasks priority can be changed dynamically using kernel-provided calls. The ability to change task priorities dynamically allows an embedded application the flexibility to adjust to external events as they occur, creating a true real-time, responsive system. Note, however, that misuse of this capability can lead to priority inversions, deadlock, and eventual system failure.

Round-Robin Scheduling

Round-robin scheduling provides each task an equal share of the CPU execution time. Pure round-robin scheduling cannot satisfy real-time system requirements because in real-time systems, tasks perform work of varying degrees of importance. Instead, preemptive, priority-based scheduling can be augmented with round-robin scheduling which uses time slicing to achieve equal allocation of the CPU for tasks of the same priority as shown in Figure 6.

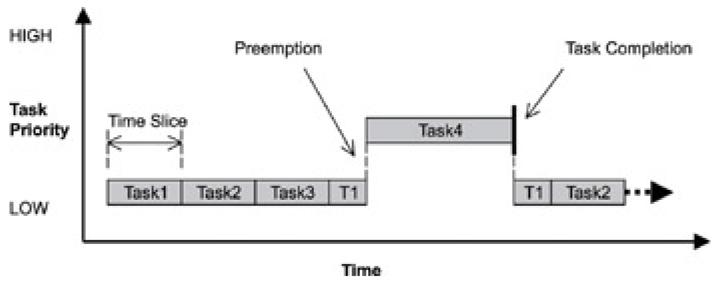


Figure 6: Round-robin and preemptive scheduling.

With time slicing, each task executes for a defined interval, or time slice, in an ongoing cycle, which is the round robin. A run-time counter tracks the time slice for each task, incrementing on every clock tick. When one tasks time slice completes, the counter is cleared, and the task is placed at the end of the cycle. Newly added tasks of the same priority are placed at the end of the cycle, with their run-time counters initialized to 0.

If a task in a round-robin cycle is preempted by a higher-priority task, its run-time count is saved and then restored when the interrupted task is again eligible for execution. This idea is illustrated in Figure 6, in which task 1 is preempted by a higher-priority task 4 but resumes where it left off when task 4 completes.

Objects

Kernel objects are special constructs that are the building blocks for application development for real-time embedded systems. The most common RTOS kernel objects are

•**Tasks** are concurrent and independent threads of execution that can compete for CPU execution time.

•**Semaphores** are token-like objects that can be incremented or decremented by tasks for synchronization or mutual exclusion.

•**Message Queues**are buffer-like data structures that can be used for synchronization, mutual exclusion, and data exchange by passing messages between tasks.

Developers creating real-time embedded applications can combine these basic kernel objects (as well as others not mentioned here) to solve common real-time design problems, such as concurrency, activity synchronization, and data communication.

Services

Along with objects, most kernels provide services that help developers create applications for real-time embedded systems.

Sets of API (applicationprogrammer's interface) **calls that can be used to**
– Perform operations on kernel objects
– Facilitate
• Timer management
• Interrupt handling
• Device I/O
• Memory management

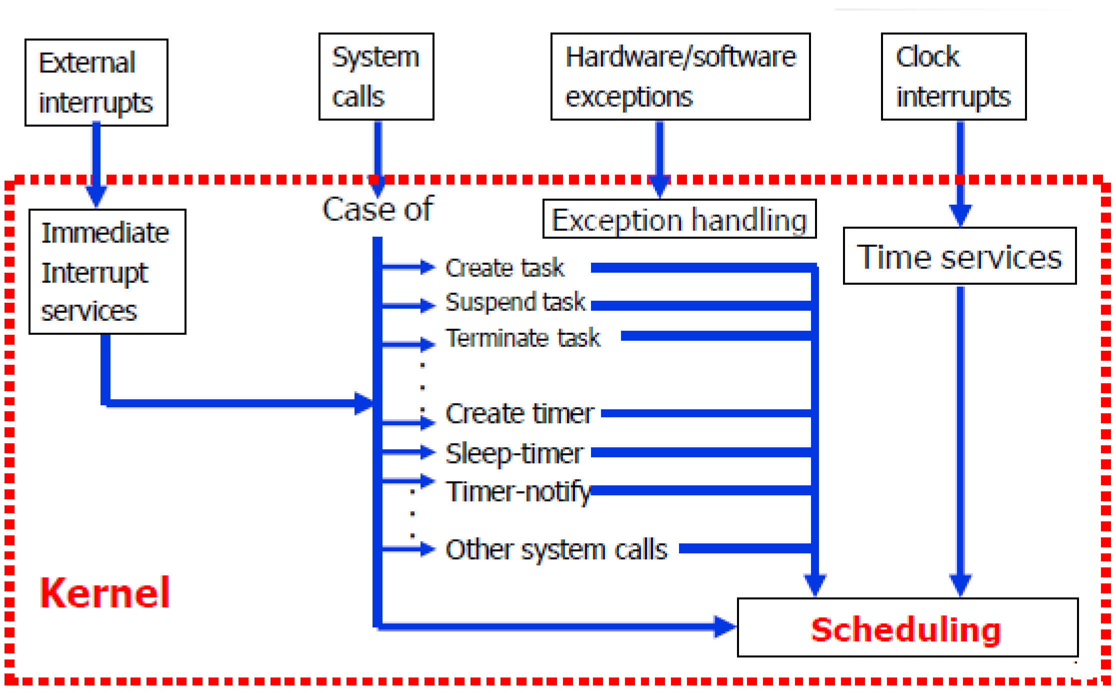


Figure 7: Basic Services Provided by a Real-Time Operating System Kernel

Key Characteristics of an RTOS

An application's requirements define the requirements of its underlying RTOS. Some of the more common attributes are

- reliability,
- predictability,
- performance,
- compactness, and
- scalability.

These attributes are discussed next; however, the RTOS attribute an application needs depends on the type of application being built.

Reliability

Embedded systems must be reliable. Depending on the application, the system might need to operate for long periods without human intervention.

Different degrees of reliability may be required. For example, a digital solar-powered calculator might reset itself if it does not get enough light, yet the calculator might still be considered acceptable. On the other hand, a telecom switch cannot reset during operation without incurring high associated costs for down time. The RTOSes in these applications require different degrees of reliability.

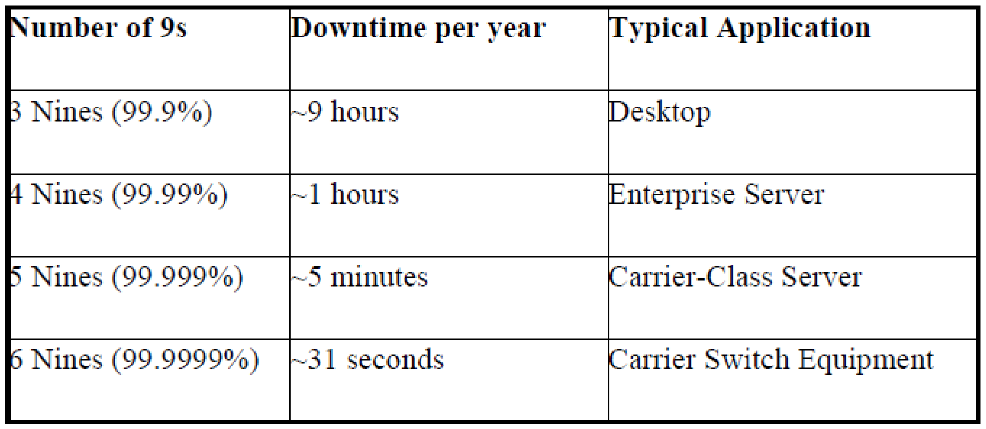
Although different degrees of reliability might be acceptable, in general, a reliable system is one that is available (continues to provide service) and does not fail.

A common way that developers categorize highly reliable systems is by quantifying their downtime per year, as shown in **Table 1**. The percentages under the 'Number of 9s' column indicate the percent of the total time that a system must be available.

While RTOSes must be reliable, note that the RTOS by itself is not what is measured to determine system reliability. It is the combination of all system elements-including the hardware, BSP, RTOS, and application-that determines the reliability of a system.

Table 1: Categorizing highly available systems by allowable downtime.

Number of 9s Downtime per year Typical application



Number of 9s	Downtime per year	Typical Application
3 Nines (99.9%)	~9 hours	Desktop
4 Nines (99.99%)	~1 hours	Enterprise Server
5 Nines (99.999%)	~5 minutes	Carrier-Class Server
6 Nines (99.9999%)	~31 seconds	Carrier Switch Equipment

Predictability

Because many embedded systems are also real-time systems, meeting time requirements is key to ensuring proper operation. The RTOS used in this case needs to be predictable to a certain degree. The term deterministic describes RTOSes with predictable behavior, in which the completion of operating system calls occurs within known timeframes.

Developers can write simple benchmark programs to validate the determinism of an RTOS. The result is based on timed responses to specific RTOS calls. In a good deterministic RTOS, the variance of the response times for each type of system call is very small.

Performance

This requirement dictates that an embedded system must perform fast enough to fulfill its timing requirements. Typically, the more deadlines to be met-and the shorter the time between them-the faster the system's CPU must be. Although underlying hardware can dictate a system's processing power, its software can also contribute to system performance. Typically, the processor's performance is expressed in million instructions per second (MIPS).

Throughput also measures the overall performance of a system, with hardware and software combined. One definition of throughput is the rate at which a system can generate output based on the inputs coming in. Throughput also means the amount of data transferred divided by the time taken to transfer it. Data transfer throughput is typically measured in multiples of bits per second (bps).

Sometimes developers measure RTOS performance on a call-by-call basis. Benchmarks are written by producing timestamps when a system call starts and when it completes. Although this step can be helpful in the analysis stages of design, true performance testing is achieved only when the system performance is measured as a whole.

Compactness

Application design constraints and cost constraints help determine how compact an embedded system can be. For example, a cell phone clearly must be small, portable, and low cost. These design requirements limit system memory, which in turn limits the size of the application and operating system.

In such embedded systems, where hardware real estate is limited due to size and costs, the RTOS clearly must be small and efficient. In these cases, the RTOS memory footprint can be an important factor. To meet total system requirements, designers must understand both the static and dynamic memory consumption of the RTOS and the application that will run on it.

Scalability

Because RTOSes can be used in a wide variety of embedded systems, they must be able to scale up or down to meet application-specific requirements. Depending on how much functionality is required, an RTOS should be capable of adding or deleting modular components, including file systems and protocol stacks.

If an RTOS does not scale up well, development teams might have to buy or build the missing pieces. Suppose that a development team wants to use an RTOS for the design of a cellular phone project and a base station project. If an RTOS scales well, the same RTOS can be used in both projects, instead of two different RTOSes, which saves considerable time and money.

Tasks: Introduction

Simple software applications are typically designed to run sequentially , one instruction at a time, in pre-determined chain of instructions. However, this scheme is inappropriate for real-time embedded applications, which generally handle multiple inputs and outputs within tight time constraints. Real-time embedded software applications must be designed for concurrency.

Concurrent design requires developers to decompose an application into small, schedulable, and sequential program units. When done correctly, concurrent design allows system multitasking to meet performance and timing requirements for a real-time system. Most RTOS kernels provide task objects and task management services to facilitate designing concurrency within an application.

Defining a Task

A *task* is an independent thread of execution that can compete with other concurrent tasks for processor execution time. Developers decompose applications into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints.

A task is *schedulable.* **T**he task is able to compete for execution time on a system, based on a predefined scheduling algorithm. A task is defined by its distinct set of parameters and supporting data structures.Specifically, upon creation, each task has an associated name, a unique ID, a priority (if part of a preemptive scheduling plan), a task control block (TCB), a stack, and a task routine, as shown in Figure 8. Together, these components make up what is known as the *task object.*

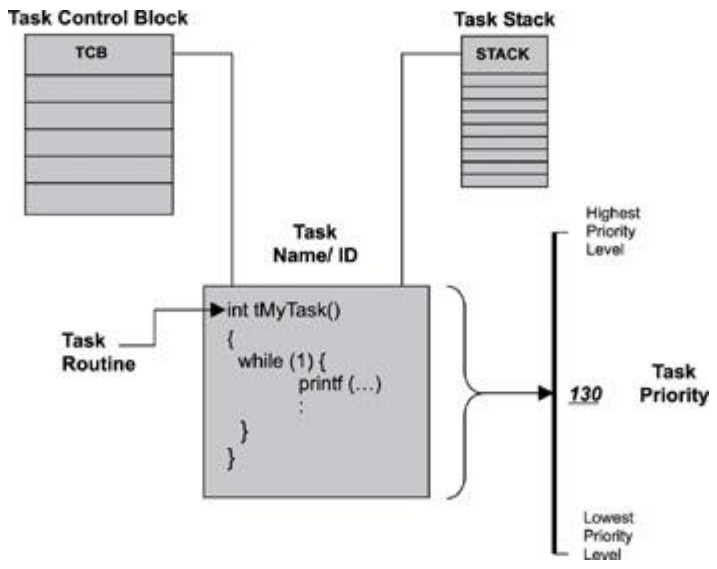


Figure 8: A task, its associated parameters, and supporting data structures.

Task Control Block

• Task ID: the unique identifier for a task

• Address Space : the address ranges of the data and codeblocks of the task loaded in memory including statically and dynamically allocated blocks

• Task Context: includes the task's program counter(PC) , the CPU registers and (optionally) floating-point registers, a stack for dynamic variables and function calls, the stack pointer (SP), I/O device assignments, a delay timer, a time-slice timer and kernel control structures

• Task Parameters : includes task type, event list

• Scheduling Information : priority level, relative deadline, period, state

• Synchronization Information : semaphores, pipes, mailboxes, message queues, file handles etc.

• Parent and Child Tasks

When the kernel first starts, it creates its own set of *system tasks* and allocates the appropriate priority for each from a set of *reserved priority levels.* The reserved priority levels refer to the priorities used internally by the RTOS for its system tasks. An application should avoid using these priority levels for its tasks because running application tasks at such level may affect the overall system performance or behavior. For most RTOSes, these reserved priorities are not enforced. The kernel needs its system tasks and their reserved priority levels to operate. These priorities should not be modified.

Examples of system tasks include:
• **initialization or startup task** initializes the system and creates and starts system tasks,
• **idle task** uses up processor idle cycles when no other activity is present,
• **logging task** logs system messages,
• **exception-handling task** handles exceptions, and
• **debug agent task** allows debugging with a host debugger.

Note that other system tasks might be created during initialization, depending on what other components are included with the kernel.

The idle task, which is created at kernel start up, is one system task that bears mention and should not be ignored. The idle task is set to the lowest priority, typically executes in an endless loop, and runs when either no other task can run or when no other tasks exist, for the sole purpose of using idle processor cycles.

In some cases, however, the kernel might allow a user-configured routine to run instead of the idle task in order to implement special requirements for a particular application. One example of a special requirement is power conservation. When no other tasks can run, the kernel can switch control to the user-supplied routine instead of to the idle task. In this case, the user-supplied routine acts like the idle task but instead initiates power conservation code, such as system suspension, after a period of idle time.

After the kernel has initialized and created all of the required tasks, the kernel jumps to a predefined entry point (such as a predefined function) that serves, in effect, as the beginning of the application. From the entry point, the developer can initialize and create other application tasks, as well as other kernel objects, which the application design might require.

Task States and Scheduling

Whether it's a system task or an application task, at any time each task exists in one of a small number of states, including ready, running, or blocked. As the real-time embedded system runs, each task moves from one state to another, according to the logic of a simple finite state machine (FSM). Figure 9 illustrates a typical FSM for task execution states, with brief descriptions of state transitions.

Although kernels can define task-state groupings differently, generally three main states are used in most typical preemptive-scheduling kernels, including:

• **ready state**-the task is ready to run but cannot because a higher priority task is executing.

• **blocked state**-the task has requested a resource that is not available, has requested to wait until some event occurs, or has delayed itself for some duration.

• **running state**-the task is the highest priority task and is running.

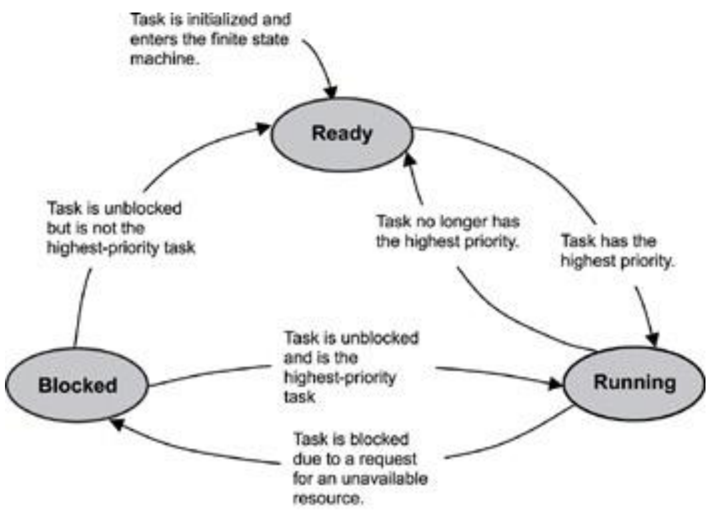

Task is initialized and enters the finite state machine.
Ready
Task is unblocked but is not the highest-priority task
Task no longer has the highest priority.
Task has the highest priority.
Task is unblocked and is the highest-priority task
Blocked
Running
Task is blocked due to a request for an unavailable resource.

Figure 9: A typical finite state machine for task execution states.

Note some commercial kernels, such as the VxWorks kernel, define other, more granular states, such as suspended, pended, and delayed. In this case, pended and delayed are actually sub-states of the blocked state. A pended task is waiting for a resource that it needs to be freed; a delayed task is waiting for a timing delay to end. The suspended state exists for debugging purposes.

Regardless of how a kernel implements a task's FSM, it must maintain the current state of all tasks in a running system. As calls are made into the kernel by executing tasks, the kernel's scheduler first determines which tasks need to change states and then makes those changes. In some cases, the kernel changes the states of some tasks, but no context switching occurs because the state of the highest priority task is unaffected. In other cases, however, these state changes result in a context switch because the former highest priority task either gets blocked or is no longer the highest priority task. When this process happens, the former running task is put into the blocked or ready state, and the new highest priority task starts to execute.

Ready State

When a task is first created and made ready to run, the kernel puts it into the ready state. In this state, the task actively competes with all other ready tasks for the processor's execution time. As Figure 9shows, tasks in the ready state cannot move directly to the blocked state. A task first needs to run so it can make a *blocking call*, which is a call to a function that cannot immediately run to completion, thus putting the task in the blocked state.

Ready tasks, therefore, can only move to the running state. Because many tasks might be in the ready state, the kernel's scheduler uses the priority of each task to determine which task to move to the running state.

For a kernel that supports only one task per priority level, the scheduling algorithm is straightforward-the highest priority task that is ready runs next. In this implementation, the kernel limits the number of tasks in an application to the number of priority levels. However, most kernels support more than one task per priority level, allowing many more tasks in an application. In this case, the scheduling algorithm is more complicated and involves maintaining a *task-ready list.* Some kernels maintain a separate task-ready list for each priority level; others have one combined list.

Figure 10 illustrates, in a five-step scenario, how a kernel scheduler might use a task-ready list to move tasks from the ready state to the running state. This example assumes a single-processor system and a priority-based preemptive scheduling algorithm in which 255 is the lowest priority and 0 is the highest.

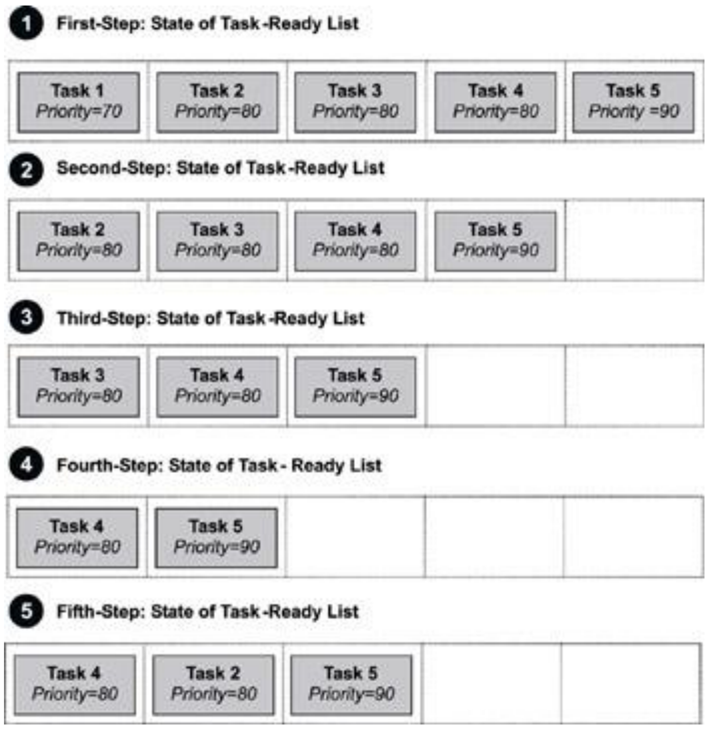


Figure 10: Five steps showing the way a task-ready list works.

In this example, tasks 1, 2, 3, 4, and 5 are ready to run, and the kernel queues them by priority in a task-ready list. Task 1 is the highest priority task (70); tasks 2, 3, and 4 are at the next-highest priority level (80); and task 5 is the lowest priority (90). The following steps explains how a kernel might use the task-ready list to move tasks to and from the ready state:

1. Tasks 1, 2, 3, 4, and 5 are ready to run and are waiting in the task-ready list.

2. Because task 1 has the highest priority (70), it is the first task ready to run. If nothing higher is running, the kernel removes task 1 from the ready list and moves it to the running state.

3. During execution, task 1 makes a blocking call. As a result, the kernel moves task 1 to the blocked state; takes task 2, which is first in the list of the next-highest priority tasks (80), off the ready list; and moves task 2 to the running state.

4. Next, task 2 makes a blocking call. The kernel moves task 2 to the blocked state; takes task 3, which is next in line of the priority 80 tasks, off the ready list; and moves task 3 to the running state.

5. As task 3 runs, frees the resource that task 2 requested. The kernel returns task 2 to the ready state

and inserts it at the end of the list of tasks ready to run at priority level 80. Task 3 continues as the currently running task.

If task 1 became unblocked at this point in the scenario, the kernel would move task 1 to the running state because its priority is higher than the currently running task (task 3). As with task 2 earlier, task 3 at this point would be moved to the ready state and inserted after task 2 (same priority of 80) and before task 5 (next priority of 90).

Running State

On a single-processor system, only one task can run at a time. In this case, when a task is moved to the running state, the processor loads its registers with this task's context. The processor can then execute the task's instructions and manipulate the associated stack.

When a task moves from the running state to the ready state, it is preempted by a higher priority task. In this case, the preempted task is put in the appropriate, priority-based location in the task-ready list, and the higher priority task is moved from the ready state to the running state.

Unlike a ready task, a running task can move to the blocked state in any of the following ways:

- by making a call that requests an unavailable resource,
- by making a call that requests to wait for an event to occur, and
- by making a call to delay the task for some duration.

Blocked State

The possibility of blocked states is extremely important in real-time systems because without blocked states lower priority tasks could not run. If higher priority tasks are not designed to block, CPU starvation can result.

CPU starvation occurs when higher priority tasks use all of the CPU execution time and lower priority tasks do not get to run.

A task can only move to the blocked state by making a blocking call, requesting that some blocking condition be met. A blocked task remains blocked until the blocking condition is met. Examples of how blocking conditions are met include the following:

- a semaphore token for which a task is waiting is released,
- a message, on which the task is waiting, arrives in a message queue, or
- a time delay imposed on the task expires.

When a task becomes unblocked, the task might move from the blocked state to the ready state if it is not the highest priority task. The task is then put into the task-ready list at the appropriate priority-based location. However, if the unblocked task is the highest priority task, the task moves directly to the running state (without going through the ready state) and preempts the currently running task. The preempted task is then moved to the ready state and put into the appropriate priority-based location in the task-ready list.

Typical Task Operations

In addition to providing a task object, kernels also provide *task-management services*. Task-management services include the actions that a kernel performs behind the scenes to support tasks, for example, creating and maintaining the TCB and task stacks.

A kernel, however, also provides an API that allows developers to manipulate tasks. Some of the more common operations that developers can perform with a task object from within the application include:

- creating and deleting tasks,
- controlling task scheduling, and
- obtaining task information.

Task Creation and Deletion

The most fundamental operations that developers must learn are creating and deleting tasks, as shown in Table 2.

Table 2: Operations for task creation and deletion.

Operation	**Description**
Create	Creates a task
Delete	Deletes a task

Developers typically create a task using one or two operations, depending on the kernels API. Some kernels allow developers first to create a task and then start it. In this case, the task is first created and put into a suspended state; then, the task is moved to the ready state when it is started (made ready to run).

Creating tasks in this manner might be useful for debugging or when special initialization needs to occur between the times that a task is created and started. However, in most cases, it is sufficient to create and start a task using one kernel call. The suspended state is similar to the blocked state, in that the suspended task is neither running nor ready to run.

Many kernels also provide *user-configurable hooks* , which are mechanisms that execute programmer-supplied functions, at the time of specific kernel events. The programmer *registers* the function with the kernel by passing a function pointer to a kernel-provided API . The kernel executes this function when the event of interest occurs. Such events can include:

- when a task is first created,
- when a task is suspended for any reason and a context switch occurs, and
- when a task is deleted.

Hooks are useful when executing special initialization code upon task creation, implementing status tracking or monitoring upon task context switches, or executing clean-up code upon task deletion.

Carefully consider how tasks are to be deleted in the embedded application. Many kernel implementations allow any task to delete any other task. During the deletion process, a kernel terminates the task and frees memory by deleting the tasks TCB and stack.

However, when tasks execute, they can acquire memory or access resources using other kernel objects. If the task is deleted incorrectly, the task might not get to release these resources. For example, assume that a task acquires a semaphore token to get exclusive access to a shared data structure. While the task is operating on this data structure, the task gets deleted. If not handled appropriately, this abrupt deletion of the operating task can result in:

- a corrupt data structure, due to an incomplete write operation,
- an unreleased semaphore, which will not be available for other tasks that might need to acquire it, and
- an inaccessible data structure, due to the unreleased semaphore.

As a result, premature deletion of a task can result in memory or resource leaks.

A *memory leak* occurs when memory is acquired but not released, which causes the system to run out of memory eventually.

A *resource leak* occurs when a resource is acquired but never released, which results in a memory leak because each resource takes up space in memory. Many kernels provide *task-deletion locks*, a pair of calls that protect a task from being prematurely deleted during a critical section of code.

At this point, however, note that any tasks to be deleted must have enough time to clean up and release resources or memory before being deleted.

Task Scheduling

From the time a task is created to the time it is deleted, the task can move through various states resulting from program execution and kernel scheduling. Although much of this state changing is automatic, many kernels provide a set of API calls that allow developers to control when a task moves to a different state, as shown in Table 3. This capability is called *manual scheduling* .

Table 3: Operations for task scheduling.

Operation	**Description**
Suspend	Suspends a task
Suspends a task	Resumes a task
Delay	Delays a task
Restart	Restarts a task
Get Priority	Gets the current tasks priority
Set Priority	Dynamically sets a task s priority
Preemption lock	Locks out higher priority tasks from preempting the current task
Preemption unlock	Unlocks a preemption lock

Using manual scheduling, developers can suspend and resume tasks from within an application. Doing so might be important for debugging purposes or, for suspending a high-priority task so that lower priority tasks can execute.

A developer might want to delay (block) a task, for example, to allow manual scheduling or to wait for an external condition that does not have an associated interrupt. Delaying a task causes it to relinquish the CPU and allow another task to execute. After the delay expires, the task is returned to the task-ready list after all other ready tasks at its priority level. A delayed task waiting for an external condition can wake up after a set time to check whether a specified condition or event has occurred, which is called *polling.*

A developer might also want to restart a task, which is not the same as resuming a suspended task. Restarting a task begins the task as if it had not been previously executing. The internal state the task possessed at the time it was suspended (for example, the CPU registers used and the resources acquired) is lost when a task is restarted. By contrast, resuming a task begins the task in the same internal state it possessed when it was suspended.

Restarting a task is useful during debugging or when reinitializing a task after a catastrophic error. During debugging, a developer can restart a task to step through its code again from start to finish. In the case of catastrophic error, the developer can restart a task and ensure that the system continues to operate without having to be completely reinitialized.

Getting and setting a task s priority during execution lets developers control task scheduling manually. This process is helpful during a *priority inversion,* in which a lower priority task has a shared resource that a higher priority task requires and is preempted by an unrelated medium-priority task. A simple fix for this problem is to free the shared resource by dynamically increasing the priority of the lower priority task to that of the higher priority task allowing the task to run and release the resource that the higher priority task requires and then decreasing the former lower priority task to its original priority.

Finally, the kernel might support *preemption locks* , a pair of calls used to disable and enable preemption in applications. This feature can be useful if a task is executing in a *critical section of code*: one in which the task must not be preempted by other tasks.

Obtaining Task Information

Kernels provide routines that allow developers to access task information within their applications, as shown in Table 4. This information is useful for debugging and monitoring.

Table 4: Task-information operations.

Operation	**Description**
Get ID	Get the current task s ID
Get TCB	Get the current task s TCB

One use to obtain a particular tasks ID, which is used to get more information about the task by getting its TCB. Obtaining a TCB, however, only takes a snapshot of the task context. If a task is not dormant (e.g., suspended), its context might be dynamic, and the snapshot information might change by the time it is used. Hence, use this functionality wisely, so that decisions aren't made in the application based on querying a constantly changing task context.

Typical Task Structure

When writing code for tasks, tasks are structured in one of two ways:

- run to completion, or
- endless loop.

Both task structures are relatively simple. Run-to-completion tasks are most useful for initialization and start-up. They typically run once, when the system first powers on. Endless-loop tasks do the majority of the work in the application by handling inputs and outputs. Typically, they run many times while the system is powered on.

Run-to-Completion Tasks

The initialization task initializes the application and creates additional services, tasks, and needed kernel objects.

Listing 1: Pseudo code for a run-to-completion task.

```
RunToCompletionTask ( )
{
        Initialize application
        Create 'endless loop tasks'
        Create kernel objects
        Delete or suspend this task—deletion
}
```

```
RunToCompletionTask ()
{
Initialize application
Create endless loop tasks'
Create kernel objects
Delete or suspend this task
```

```
RunToCompletionTask ()
{
Initialize application
Create endless loop tasks'
Create kernel objects
Delete or suspend this task
}
```

The application initialization task typically has a higher priority than the application tasks it creates so that its initialization work is not preempted. In the simplest case, the other tasks are one or more lower priority endless-loop tasks. The application initialization task is written so that it suspends or deletes itself after it completes its work so the newly created tasks can run.

Endless-Loop Tasks

As with the structure of the application initialization task, the structure of an endless loop task can also contain initialization code. The endless loop's initialization code, however, only needs to be executed when the task first runs, after which the task executes in an endless loop.

The critical part of the design of an endless-loop task is the one or more blocking calls within the body of the loop. These blocking calls can result in the blocking of this endless-loop task, allowing lower priority tasks to run.

Listing 2: Pseudo code for an endless-loop task.

```
EndlessLoopTask ( )
{
    Initialization code
    Loop Forever
    {
        Body of loop
        Make one or more blocking calls
    }
}
```

Multitasking

Multitasking is the ability of a computer to run more than one program, or *task* , at the same time. Multitasking contrasts with single-tasking, where one process must entirely finish before another can begin. MS-DOS is primarily a single-tasking environment, while Windows 3.1 and Windows NT are both multi-tasking environments.

On a single-processor multitasking system, multiple processes don’t actually run at the same time since there’s only one processor. Instead, the processor switches among the processes that are active at any given time. Because computers are so fast compared with people, however, it appears to the user as though the computer is executing all of the tasks at once. Multitasking also allows the computer to make good use of the time it would otherwise spend waiting for I/O devices and user input–that time can be used for some other task that doesn’t need I/O at the moment.

More complex systems may have many complex tasks. In a multi-tasking system, numerous tasks require CPU time, and since there is only one CPU, some form of organization and coordination is needed so each task has the CPU time it needs. In practice, each task takes a very brief amount of time, so it seems as if all the tasks are executing in parallel and simultaneously.

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity.

The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed.

To realise such a system, the following major functions are to be carried out.

A. Process Management

- interrupt handling
- task scheduling and dispatch
    - create/delete, suspend/resume task
    - manage scheduling information
        - – priority, scheduling policy, etc

B. Interprocess Communication and Synchronization

- Code, data and device sharing
- Synchronization, coordination and data exchange mechanisms
- Deadlock and Livelock detection

C. Memory Management

- dynamic memory allocation
- memory locking
- Services for file creation, deletion, reposition and protection

D. Input/Output Management

- Handles request and release functions and read, write functions for a variety of peripherals

The following are important requirements that an OS must meet to be considered an RTOS in the contemporary sense.

A. The operating system must be multithreaded and preemptive. e.g. handle multiple threads and be able to preempt tasks if necessary.

B. The OS must support priority of tasks and threads.

C. A system of priority inheritance must exist. Priority inheritance is a mechanism to ensure that lower priority tasks cannot obstruct the execution of higher priority tasks.

D. The OS must support various types of thread/task synchronization mechanisms.

E. For predictable response :

a. The time for every system function call to execute should be predictable and independent of the number of objects in the system.

b. Non preemptable portions of kernel functions necessary for interprocess synchronization and communication are highly optimized, short and deterministic

c. Non-preemptable portions of the interrupt handler routines are kept small and deterministic

d. Interrupt handlers are scheduled and executed at appropriate priority

e. The maximum time during which interrupts are masked by the OS and by device drivers must be known.

f. The maximum time that device drivers use to process an interrupt, and specific IRQ information relating to those device drivers, must be known.

g. The interrupt latency (the time from interrupt to task run) must be predictable and compatible with application requirements

F. For fast response:

a. Run-time overhead is decreased by reducing the unnecessary context switch.

b. Important timings such as context switch time, interrupt latency, semaphore get/release latency must be minimum

Type of Multitasking:

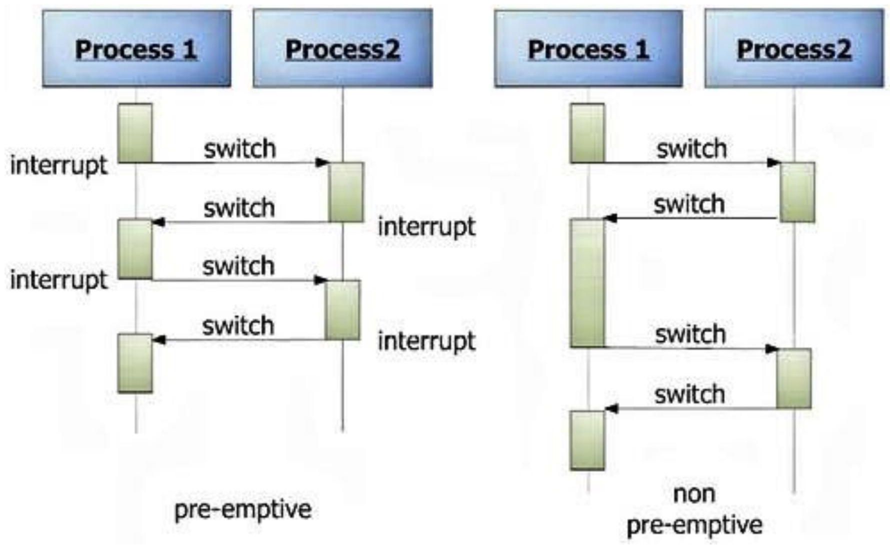
Preemptive and Non-Preemptive Multitasking

Within the category of multitasking, there are two major sub-categories: *preemptive* and *non-preemptive* (or cooperative). In *non-preemptive multitasking* , use of the processor is never taken from a task; rather, a task must voluntarily yield control of the processor before any other task can run.

*Preemptive multitasking*differs from non-preemptive multitasking in that the operating system can take control of the processor without the task’s cooperation. (A task can also give it up voluntarily, as in non-preemptive multitasking.) The process of a task having control taken from it is called preemption.

A preemptive operating system takes control of the processor from a task in two ways:

- When a task’s time quantum (or time slice) runs out. Any given task is only given control for a set amount of time before the operating system interrupts it and schedules another task to run.
- When a task that has higher priority becomes ready to run. The currently running task loses control of the processor when a task with higher priority is ready to run regardless of whether it has time left in its quantum or not.



Multiprocessor Systems

Although the concept of a multiprocessor system has been around for decades, it's only recently attained commercial viability as demand grows for scalable, high-performance, highly reliable systems. This increased demand for so-called "highly available" CPU-intensive systems (systems with 99.999% uptime) has spawned several types of multiprocessor systems, each designed for a specific application. Multiprocessor systems are widely used in applications involving 3D graphics with audio/video compression and decompression running on specialized multiprocessor chips (for example, Fuzion 150 from PixelFusion). These systems are also seen in high bandwidth network traffic switch/router designs, including special network management features on specialized multiprocessors (for example, SB1250 with SB-1 chip of Broadcom's Mercurian family of programmable network processors).

A *multiprocessing* operating system is one that can run on computer systems that contain more than one processor. S*ymmetric multiprocessing (SMP)* system, meaning that it assumes that all of the processors are equal and that they all have access to the same physical memory. Therefore, Windows NT can run any thread on any available processor regardless of what process, user or Executive, owns the thread.

There are also asymmetric multiprocessing (ASMP) systems in which processors are different from each other–they may address different physical memory spaces, or they may have other differences. These operating systems only run certain processes on certain processors–for instance, the kernel might always execute on a particular processor.

*Symmetric multiprocessing (***SMP***)*

In SMP configuration, the term *tightly coupled* means that the individual processor cores lie close to each other and are physically connected over a common high-speed bus. The processors share a global memory module (*shared memory*) and peripheral devices through a common I/O bus interface.

It's important to know the difference between such an SMP system and a distributed multiprocessor environment. In a distributed multiprocessor system, the individual processing units typically reside as separate nodes. Each of these nodes can have a different processor type and its own memory and I/O devices. Each processor runs its own operating system and synchronizes with other processors using messages or semaphores over an interconnect such as Ethernet or a more specialized high-speed interconnect such as Infiniband.

On the other hand, tightly coupled shared-memory SMP system runs a single copy of an operating system that coordinates simultaneous activity occurring in each of the identical CPUs. Since these tightly coupled CPUs access a common memory region, they synchronize with each other using a communication mechanism based on low-latency shared memory.

An SMP system, by definition, has multiple identical processors connected to a set of peripherals, such as memory and I/O devices, on a common high-speed bus. The term *symmetric* is used because each processor has the same access mechanism to the shared memory and peripheral devices.

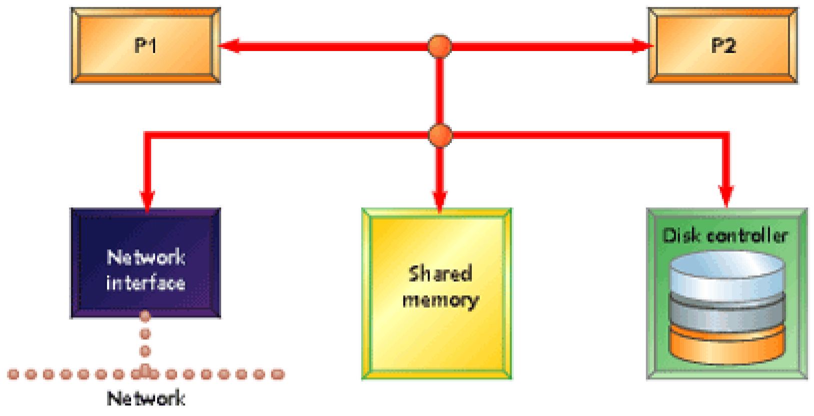


Figure 11: Shared-memory symmetric multiprocessor architecture

Figure 11 shows an SMP system with two identical processors (P1 and P2), a global memory module, and I/O devices such as disk controllers and network interfaces. The two processors share the memory device over a common high-speed bus and are typically connected to each of the peripheral devices through a high-speed bus interface. The processors have to contend for access to the bus (for example, to read/write to the global memory).

If P1 has occupied the bus for receiving data from the network interface, P2 has to wait for the operation to complete before getting access to write data to the disk. Thus, while any one device is busy accessing the memory for a read-write operation, all the other devices seeking access enter the busy-wait state.

However, each of these devices typically has some local memory reserved exclusively for the device's own use. For example, an I/O device, such as a typical network interface controller, has some local memory to buffer and preprocess data frames that are received over the network before they're transferred (typically through DMA) to the global memory for further processing by P1 or P2.

In addition to the per-CPU interrupt controller, such a system usually has another interrupt controller (an IO-APIC from Intel, for example) that routes the interrupt request signals from the peripheral devices to one of the CPUs for servicing.

Some benefits of SMP system:

- Two processing units running in parallel complete a set of tasks more quickly than one processor. Four processors are, likewise, better than two.
- An SMP system that can be scaled by adding more processing cores and/or peripheral devices to execute more tasks in parallel offers a viable platform for scalability.
- If one processor suffers a fatal failure, another can take over its work seamlessly, reducing downtime.

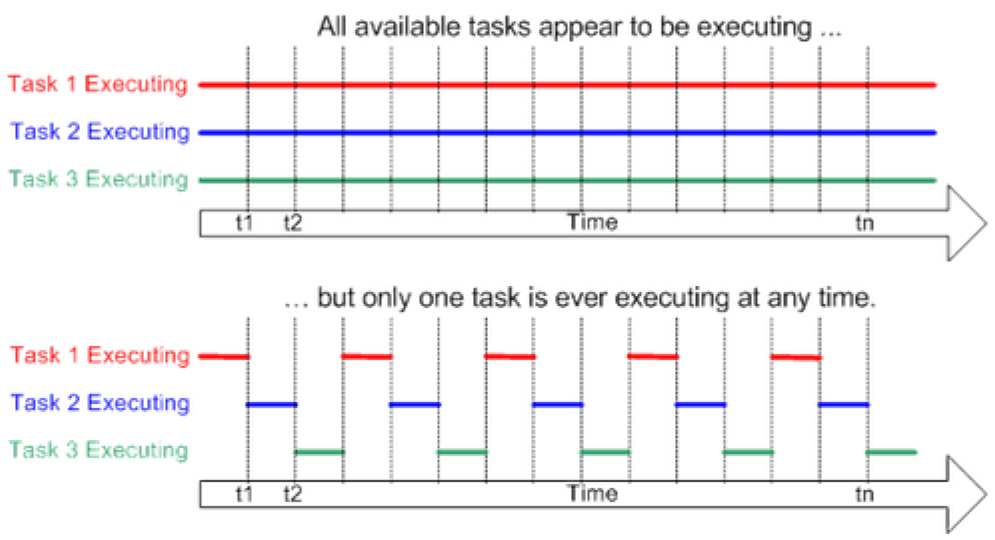
Synchronization, Communication, and Concurrency

Tasks synchronize and communicate amongst themselves by using *intertask primitives* , which are kernel objects that facilitate synchronization and communication between two or more threads of execution. Examples of such objects include semaphores, message queues, signals, and pipes, as well as other types of objects.

The task object is the fundamental construct of most kernels. Tasks, along with task-management services, allow developers to design applications for concurrency to meet multiple time constraints and to address various design problems inherent to real-time embedded applications.

Multitasking Vs Concurrency

A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it **appear** as if each task is executing concurrently. This is depicted by the diagram below which shows the execution pattern of three tasks with respect to time. The task names are color coded and written down the left hand. Time moves from left to right, with the colored lines showing which task is executing at any particular time. The upper diagram demonstrates the perceived concurrent execution pattern and the lower the actual multitasking execution pattern.



Concurrency

It is important to encapsulate concurrency within an application into manageable units. A *unit of concurrency* can be a task or a process; it can be any schedulable thread of execution that can compete for the CPU's processing time.

Although ISRs are not scheduled to run concurrently with other routines, they should also be considered in designing for concurrency because they follow a preemptive policy and are units of execution competing for CPU processing time.

The primary objective of this decomposition process is to optimize parallel execution to maximize a real-time application's performance and responsiveness. If done correctly, the result can be a system that meets all of its deadlines robustly and responsively. If done incorrectly, real-time deadlines can be compromised, and the system's design may not be acceptable.

Pseudo versus True Concurrent Execution

Concurrent tasks in a real-time application can be scheduled to run on a single processor or multiple processors. Single-processor systems can achieve pseudo concurrent execution, in which an application is decomposed into multiple tasks maximizing the use of a single CPU.

It is important to note that on a single-CPU system, only one *program counter* (also called an *instruction pointer)* is used, and, hence, only one instruction can be executed at any time. Most applications in this environment use an underlying scheduler's multitasking capabilities to interleave the execution of multiple tasks; therefore, the term *pseudo concurrent execution* is used.

In contrast, true concurrent execution can be achieved when multiple CPUs are used in the designs of real-time embedded systems. For example, if two CPUs are used in a system, two concurrent tasks can execute in parallel at one time, as shown in Figure 12. This parallelism is possible because two program counters (one for each CPU) are used, which allows for two different instructions to execute simultaneously.

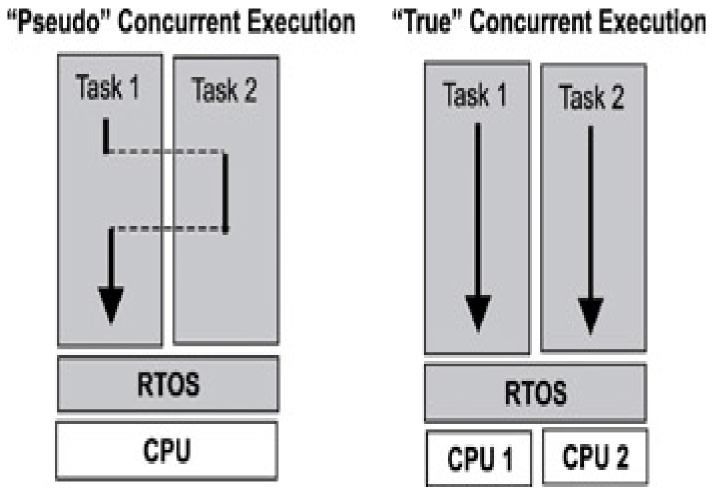


Figure 12: Pseudo and true concurrent (parallel) execution.

In the case of multiple CPU systems, the underlying RTOS typically is *distributed,* which means that various components, or copies of RTOS components, can execute on different CPUs. On such systems, multiple tasks can be assigned to run on each CPU, just as they do on single-CPU systems. In this case, even though two or more CPUs allow true concurrent execution, each CPU might actually be executing in a pseudo-concurrent fashion.

Synchronization

Software applications for real-time embedded systems use concurrency to maximize efficiency. As a result, an application's design typically involves multiple concurrent threads, tasks, or processes. Coordinating these activities requires inter-task synchronization and communication.

Synchronization is classified into two categories: *resource synchronization* and *activity synchronization.* Resource synchronization determines whether access to a shared resource is safe, and, if not, when it will be safe. Activity synchronization determines whether the execution of a multithreaded program has reached a certain state and, if it hasn't, how to wait for and be notified when this state is reached.

Resource Synchronization

Access by multiple tasks must be synchronized to maintain the integrity of a shared resource. This process is called *resource synchronization*, a term closely associated with critical sections and mutual exclusions. *Mutual exclusion* is a provision by which only one task at a time can access a shared resource. A *critical section* is the section of code from which the shared resource is accessed.

As an example, consider two tasks trying to access shared memory. One task (the sensor task) periodically receives data from a sensor and writes the data to shared memory. Meanwhile, a second task (the display task) periodically reads from shared memory and sends the data to a display. The common design pattern of using shared memory is illustrated in Figure 13.

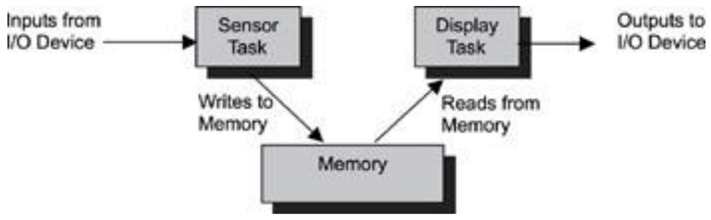


Figure 13: Multiple tasks accessing shared memory.

Problems arise if access to the shared memory is not exclusive, and multiple tasks can simultaneously access it. For example, if the sensor task has not completed writing data to the shared memory area before the display task tries to display the data, the display would contain a mixture of data extracted at different times, leading to erroneous data interpretation.

The section of code in the sensor task that writes input data to the shared memory is a critical section of the sensor task. The section of code in the display task that reads data from the shared memory is a critical section of the display task. These two critical sections are called *competing critical sections* because they access the same shared resource.
A mutual exclusion algorithm ensures that one task's execution of a critical section is not interrupted by the competing critical sections of other concurrently executing tasks.

One way to synchronize access to shared resources is to use a client-server model, in which a central entity called a *resource server* is responsible for synchronization. Access requests are made to the resource server, which must grant permission to the requestor before the requestor can access the shared resource. The resource server determines the eligibility of the requestor based on pre-assigned rules or run-time heuristics. While this model simplifies resource synchronization, the resource server is a bottleneck.
Synchronization primitives, such as semaphores and mutexes, and other methods allow developers to implement complex mutual exclusion algorithms. These algorithms in turn allow dynamic coordination among competing tasks without intervention from a third party.

Activity Synchronization

In general, a task must synchronize its activity with other tasks to execute a multithreaded program properly. *Activity synchronization* is also called *condition synchronization* or *sequence control* . Activity synchronization ensures that the correct execution order among cooperating tasks is used. Activity synchronization can be either synchronous or asynchronous.

One representative of activity synchronization methods is *barrier synchronization.* For example, in embedded control systems, a complex computation can be divided and distributed among multiple tasks. Some parts of this complex computation are I/O bound, other parts are CPU intensive, and still others are mainly floating-point operations that rely heavily on specialized floating-point coprocessor hardware. These partial results must be collected from the various tasks for the final calculation. The result determines what other partial computations each task is to perform next. The point at which the partial results are collected and the duration of the final computation is a *barrier.*

One task can finish its partial computation before other tasks complete theirs, but this task must wait for all other tasks to complete their computations before the task can continue.

Barrier synchronization comprises three actions:

- a task posts its arrival at the barrier,
- the task waits for other participating tasks to reach the barrier, and
- the task receives notification to proceed beyond the barrier.

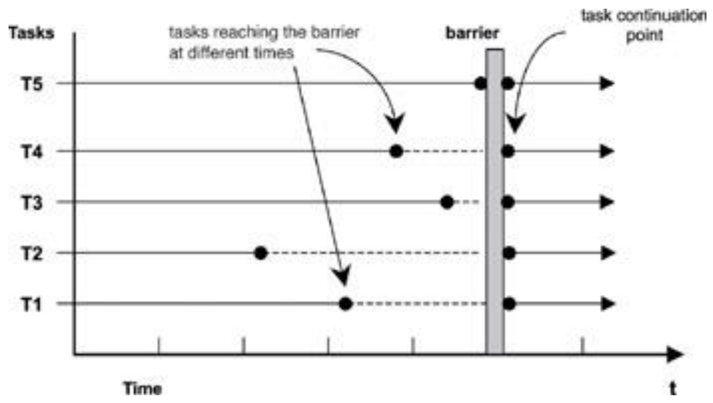


Figure 14: Visualization of barrier synchronization.

As shown in Figure 14, a group of five tasks participates in barrier synchronization. Tasks in the group complete their partial execution and reach the barrier at various times; however, each task in the group must wait at the barrier until all other tasks have reached the barrier. The last task to reach the barrier (in this example, task T5) broadcasts a notification to the other tasks. All tasks cross the barrier at the same time (conceptually in a uniprocessor environment due to task scheduling. We say 'conceptually' because in a uniprocessor environment, only one task can execute at any given time. Even though all five tasks have crossed the barrier and may continue execution, the task with the highest priority will execute next.

Another representative of activity synchronization mechanisms is *rendezvous synchronization,* which, as its name implies, is an execution point where two tasks meet. The main difference between the barrier and the rendezvous is that the barrier allows activity synchronization among two or more tasks, while rendezvous synchronization is between two tasks.

In rendezvous synchronization, a synchronization and communication point called an *entry* is constructed as a function call. One task defines its entry and makes it public. Any task with knowledge of this entry can call it as an ordinary function call. The task that defines the entry accepts the call, executes it, and returns the results to the caller. The issuer of the entry call establishes a rendezvous with the task that defined the entry.

Rendezvous synchronization is similar to synchronization using event-registers, in that both are synchronous. The issuer of the entry call is blocked if that call is not yet accepted; similarly, the task that accepts an entry call is blocked when no other task has issued the entry call. Rendezvous differs from event-register in that bidirectional data movement (input parameters and output results) is possible.

A derivative form of rendezvous synchronization, called *simple rendezvous* in this book, uses kernel primitives, such as semaphores or message queues, instead of the entry call to achieve synchronization. Two tasks can implement a simple rendezvous without data passing by using two binary semaphores, as shown in Figure 15.

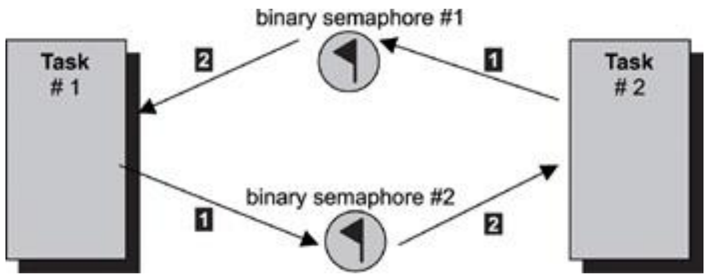


Figure 15: Simple rendezvous without data passing.

Both binary semaphores are initialized to 0 . When task #1 reaches the rendezvous, it gives semaphore #2, and then it gets on semaphore #1. When task #2 reaches the rendezvous, it gives semaphore #1, and then it gets on semaphore #2. Task #1 has to wait on semaphore #1 before task #2 arrives, and vice versa, thus achieving rendezvous synchronization.

Implementing Barriers

Barrier synchronization is used for activity synchronization. Listing 1 shows how to implement a barrier-synchronization mechanism using a mutex and a condition variable.

Listing 1: Pseudo code for barrier synchronization.

```
typedef struct {
mutex_t br_lock; /* guarding mutex */
cond_t br_cond; /* condition variable */
int br_count; /* num of tasks at the barrier */
int br_n_threads; /* num of tasks participating in the barrier
synchronization */
} barrier_t;
barrier(barrier_t *br)
{
mutex_lock(&br->br_lock);
br->br_count++;
if (br->br_count < br->br_n_threads)
cond_wait(&br->br_cond, &br->br_lock);
else
{
br->br_count = 0;
cond_broadcast(&br->br_cond);
}
mutex_unlock(&br->br_lock);
}
```

Each participating task invokes the function barrier for barrier synchronization. The guarding mutex for br_count and br_n_threads is acquired on line #2. The number of waiting tasks at the barrier is updated on line #3. Line #4 checks to see if all of the participating tasks have reached the barrier. If more tasks are to arrive, the caller waits at the barrier (the blocking wait on the condition variable at line #5).

If the caller is the last task of the group to enter the barrier, this task resets the barrier on line #6 and notifies all other tasks that the barrier synchronization is complete. Broadcasting on the condition variable on line #7 completes the barrier synchronization.

Communication

Tasks communicate with one another so that they can pass information to each other and coordinate their activities in a multithreaded embedded application. Communication can be signal-centric, data-centric, or both. In *signal-centric communication,* all necessary information is conveyed within the event signal itself. In *data-centric communication*, information is carried within the transferred data. When the two are combined, data transfer accompanies event notification.

When communication involves data flow and is unidirectional, this communication model is called *loosely coupled communication.* In this model, the data producer does not require a response from the consumer. Figure 16 illustrates an example of loosely coupled communication.

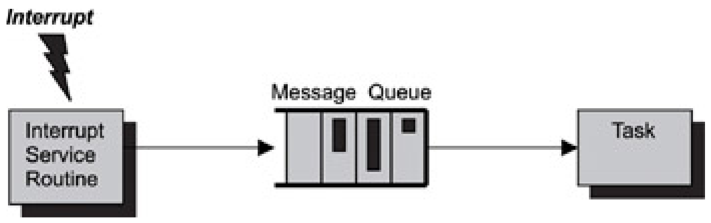


Figure 16: Loosely coupled ISR-to-task communication using message queues.

For example, an ISR for an I/O device retrieves data from a device and routes the data to a dedicated processing task. The ISR neither solicits nor requires feedback from the processing task. By contrast, in *tightly coupled communication* , the data movement is bidirectional. The data producer synchronously waits for a response to its data transfer before resuming execution, or the response is returned asynchronously while the data producer continues its function.

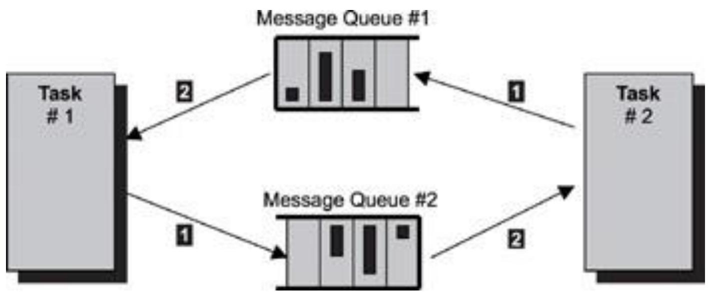


Figure 17: Tightly coupled task-to-task communication using message queues.

In tightly coupled communication, as shown in Figure 17, task #1 sends data to task #2 using message queue #2 and waits for confirmation to arrive at message queue #1. The data communication is bidirectional. It is necessary to use a message queue for confirmations because the confirmation should contain enough information in case task #1 needs to re-send the data. Task #1 can send multiple messages to task #2, i.e., task #1 can continue sending messages while waiting for confirmation to arrive on message queue #2.

Communication has several purposes, including the following:

- transferring data from one task to another,
- signaling the occurrences of events between tasks,
- allowing one task to control the execution of other tasks,
- synchronizing activities, and
- implementing custom synchronization protocols for resource sharing.

➢ The first purpose of communication is for one task to transfer data to another task. Between the tasks, there can exist data dependency, in which one task is the data producer and another task is the data consumer. For example, consider a specialized processing task that is waiting for data to arrive from message queues or pipes or from shared memory. In this case, the data producer can be either an ISR or another task. The consumer is the processing task. The data source can be an I/O device or another task.

➢ The second purpose of communication is for one task to signal the occurrences of events to another task. Either physical devices or other tasks can generate events. A task or an ISR that is responsible for an event, such as an I/O event, or a set of events can signal the occurrences of these events to other tasks. Data might or might not accompany event signals. Consider, for example, a timer chip ISR that notifies another task of the passing of a time tick.

➢ The third purpose of communication is for one task to control the execution of other tasks. Tasks can have a master/slave relationship, known as *process control* . For example, in a control system, a master task that has the full knowledge of the entire running system controls individual subordinate tasks. Each subtask is responsible for a component, such as various sensors of the control system. The master task sends commands to the subordinate tasks to enable or disable sensors. In this scenario, data flow can be either unidirectional or bidirectional if feedback is returned from the subordinate tasks.

➢ The fourth purpose of communication is to synchronize activities. The computation example given in 'Activity Synchronization' , when multiple tasks are waiting at the execution barrier, each task waits for a signal from the last task that enters the barrier, so that each task can continue its own execution. In this example, it is insufficient to notify the tasks that the final computation has completed; additional information, such as the actual computation results, must also be conveyed.

- The fifth purpose of communication is to implement additional synchronization protocols for resource sharing. The tasks of a multithreaded program can implement custom, more-complex resource synchronization protocols on top of the system-supplied synchronization primitives.