REAL-TIME OPERATING SYSTEMS

Venkata Kowsik T

Adhokshaja V Madhwaraj

160010035, 160010032

TABLE OF CONTENTS

MOTIVATION BEHIND REAL-TIME OPERATING SYSTEMS

Embedded Computing Applications exist in a spectacular range of size and complexity for areas such as home automation to cell phones, automobiles, and industrial controllers. Most of these applications demand such functionality, performance, and reliability from the software that simple and direct assembly language programming of the processors is clearly ruled out. Moreover, as a distinguishing feature from general purpose computing, a large part of the computation is “real-time” or time constrained and also reactive or “external event-driven” since such systems generally interface strongly with the external environment through a variety of devices. Thus, an operating system is generally used. An operating system facilitates the development of an application program by making available a number of services, which, otherwise would have to be coded by the application program. The application programs “interface” with the hardware through the operating system services and functions. It is therefore important to understand the basic features of such operating systems, and this is what we will do in the coming sections.

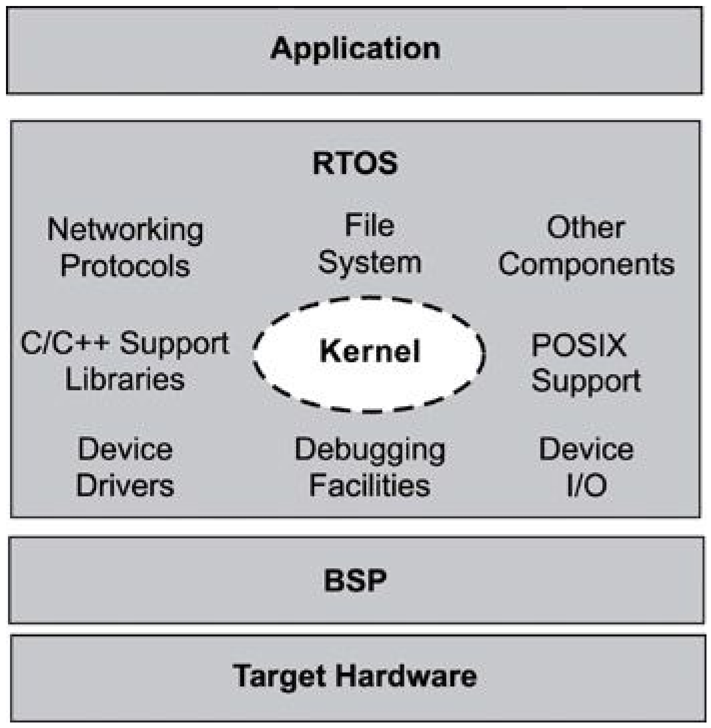
REAL-TIME OPERATING SYSTEMS

WHAT ARE OPERATING SYSTEMS?

An Operating System is a collection of programs that provide an interface between application programs and the computer system (hardware). Its primary function is to provide application programmers with an abstraction of the system resources, such as memory, input-output, and processor, which enhances the convenience, efficiency, and correctness of their use. These programs or functions within the OS provide various kinds of services to the application programs. The application programs, in turn, call these programs to avail of such services. Thus the application programs can view the computer resources as abstract entities, (for example, a block of memory can be used as a named sequential file with the abstract Open, Close, Read, Write operations) without need for knowing the low level hardware details (such as the addresses of the memory blocks).

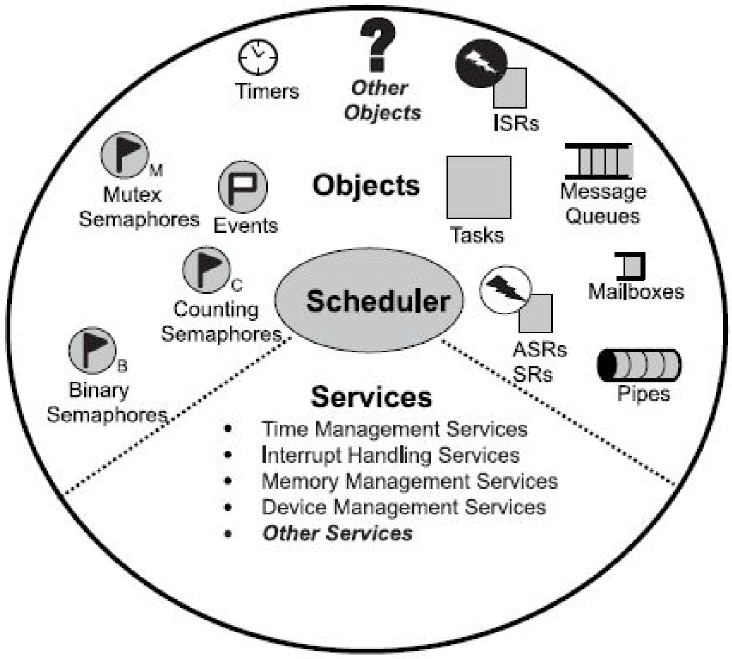
REAL-TIME OPERATING SYSTEMS

A Real-Time OS (RTOS) is an OS with special features that make it suitable for building real-time computing applications also referred to as Real-Time Systems (RTS). Real-time operating systems (RTOSs) provide basic support for scheduling, resource management, synchronization, communication, and I/O like general purpose OSs (GPOSs) but also have additional features for precise timing. RTOSs have evolved from being completely predictable and support safety-critical applications to those which support soft real-time applications. Soft and hard applications are dealt with later. Researchers have developed new paradigms and ideas that enhance the GPOSs to be more efficient and predictable. The current RTOS market includes many proprietary kernels for embedded systems like Atmega, composition-based kernels, real-time versions of popular OSs like Linux (Linux RT) and Windows NT (Windows CE).



RTOSs are mainly developed for real-time systems (RTS). In an RTS, the correctness not only depends on the correctness of the logical result but also the result delivery time. An RTS is expected to respond in a timely, predictable way to unpredictable external stimuli.

COMPONENTS OF AN RTOS KERNEL



The most common element at the heart of all RTOSs is the kernel. Most RTOS kernels contain the following components:

- The Scheduler is contained within each kernel and follows a set of algorithms that determine 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, general operations such as timing, interrupt handling, and resource management.

The components that are listed here will be explained in further detail in the coming sections.

HARD vs SOFT RTOSs

Hard real-time system

This type of system can never miss its deadline. Missing the deadline may have disastrous consequences. The usefulness of result produced by a hard real-time system decreases abruptly and may become negative if tardiness increases. (Tardiness means how late a real-time system completes its task with respect to its deadline.)

Example: Flight controller systems.

Soft real-time system

This type of system can miss its deadline occasionally with some acceptably low probability. Missing the deadline have no disastrous consequences. The usefulness of result produced by a soft real-time system decreases gradually with increase in tardiness.

Example: Telephone switches.

In general, the smaller and more deterministic kernels provide support for hard systems. Here all the inputs and system details are known and careful design and analysis result in meeting hard deadline requirements. The larger and more stochastic kernels provide support for soft RTSs. Here the Quality of Service (QoS) guarantees is defined and shown to be met in a probabilistic sense.

INSIDE AN RTOS KERNEL

THE SCHEDULER

The scheduler is at the heart of every kernel. It provides the algorithms needed to determine which task executes when. To understand how exactly scheduling works, we need to look at the following things.

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. Most 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. Note that message queues and semaphores are not schedulable entities. These items are inter-task communication objects used for synchronization and communication.

Multitasking

Multitasking is the ability of the operating system to handle multiple activities **within set deadlines** (this is the most important point of an RTOS). A real-time kernel might have multiple tasks that it has to schedule to run.

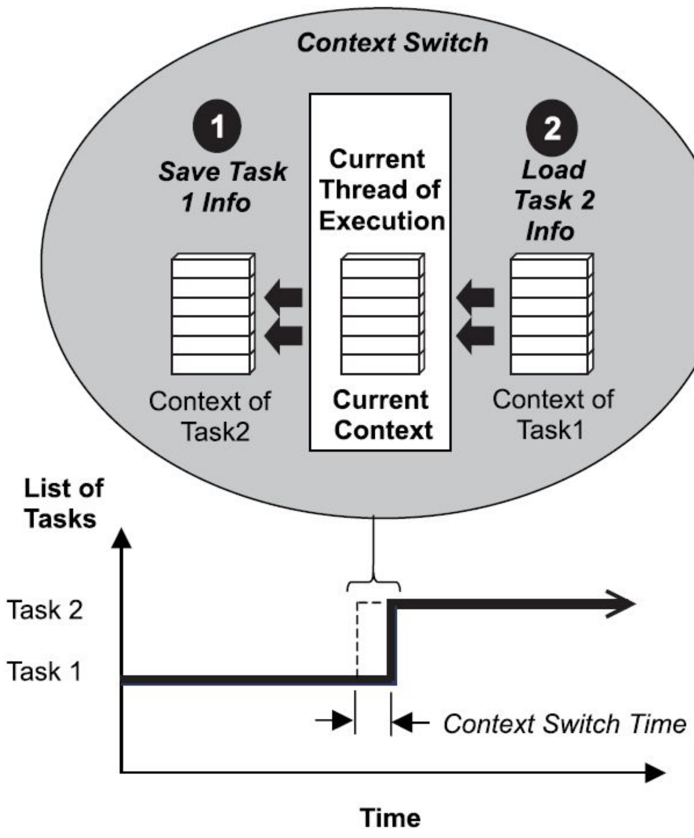
In this scenario, the kernel multitasks in such a way that many threads of execution appear to be running concurrently; however, the kernel is actually interleaving executions sequentially, based on a preset scheduling algorithm. The scheduler must ensure that the appropriate task runs at the right time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

An important point to note here is that the tasks follow the kernel's 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 context switching.

The Context Switch

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



The kernel creates and maintains an associated Task Control Block (TCB). TCBs are pretty much like PCBs of a GPOS. The context of a running process is highly dynamic. This information is stored in the TCB. The adjoining figure shows a typical context switch along with a small graph to show the same.

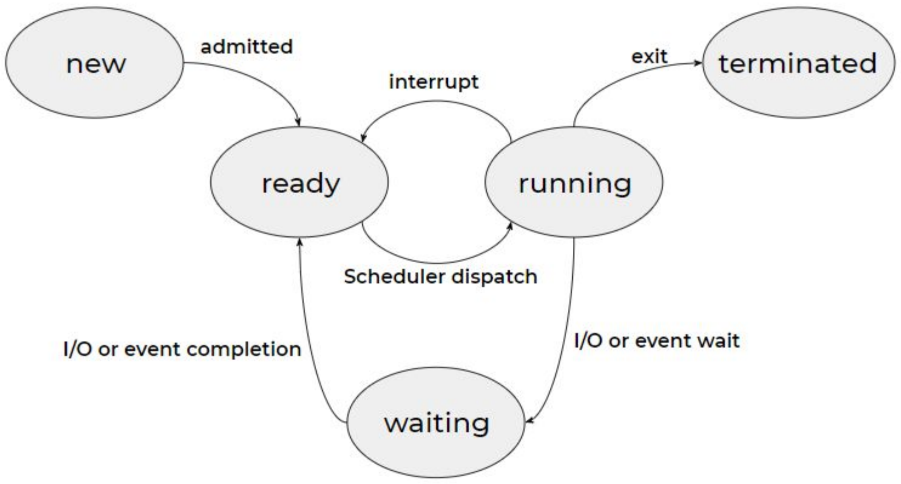
As shown in the adjacent figure, when the kernel's scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps:

1. The kernel saves task 1's context information in its TCB.
2. It loads task 2's 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. If an application's design includes frequent context switching, however, the application can incur unnecessary performance overhead.

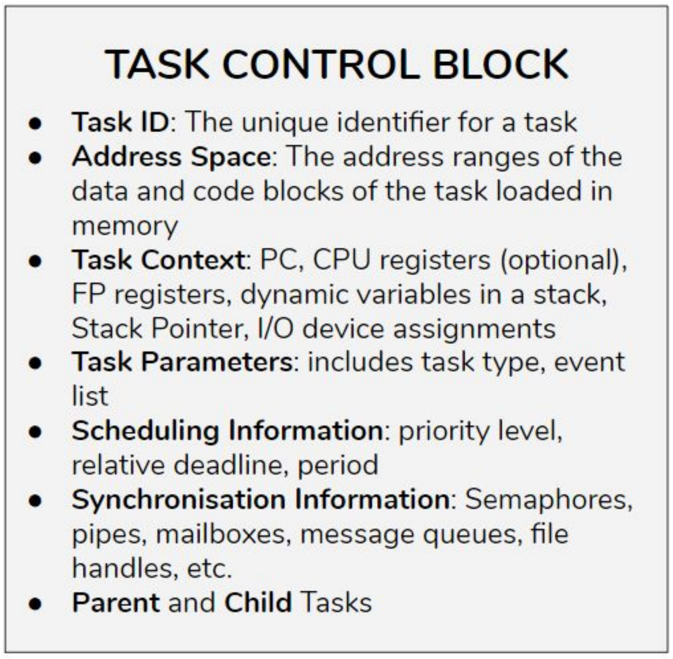
When the scheduler determines a context switch is necessary, it relies on an associated module, called the dispatcher, to make that switch happen.

A typical RTOS provides certain Task Control Functions to spawn, initialize and activate new tasks. They provide functions to gather information on existing tasks in the system, for task naming, checking of the state of a given task, setting options for task execution



such as the use of co-processor, specific memory models, as well as task deletion. Deletion often requires special precautions, especially with respect to semaphores, for shared memory tasks.

The above diagram shows the different task states that we are already familiar with.



TASK CONTROL BLOCK

- **Task ID**: The unique identifier for a task
- **Address Space**: The address ranges of the data and code blocks of the task loaded in memory
- **Task Context**: PC, CPU registers (optional), FP registers, dynamic variables in a stack, Stack Pointer, I/O device assignments
- **Task Parameters**: includes task type, event list
- **Scheduling Information**: priority level, relative deadline, period
- **Synchronisation Information**: Semaphores, pipes, mailboxes, message queues, file handles, etc.
- **Parent** and **Child** Tasks

The Dispatcher

This is the part of the schedule that actually performs context switching and changes the flow of execution. At any time the RTOS is running, the flow of control is flowing through one of three areas, namely - Application Task, Interrupt Service Routine(ISR) or 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 (need not to be the same as the system call).

The dispatcher can be used on a call-by-call basis so that it can coordinate task-state transitions that any of the system calls might have caused (more than one process may have come into the Ready Queue).

Here is where there is the catch of RTOSs. If an ISR makes system calls, the dispatcher is bypassed until the ISR fully completes its execution. This is equivalent to the ISR having high priority. 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 happen since the ISR should complete without interrupting their tasks. After the ISR completes its work, the kernel exits through the dispatcher so that the next correct task is scheduled.

Scheduling Algorithms

Most kernels today support two common scheduling algorithms:

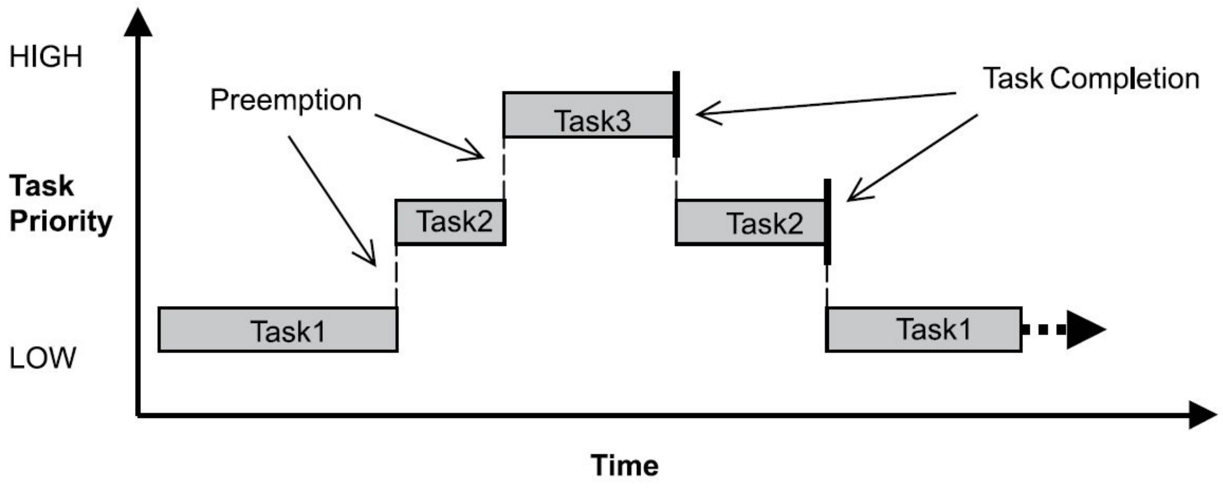
- preemptive priority-based scheduling, and
- round-robin scheduling.

The RTOS manufacturer typically pre-defines these algorithms; however, in some cases, developers can create and define their own scheduling algorithms. Here we go through these algorithms in some more detail.

Preemptive Priority-Based Scheduling

Of the two scheduling algorithms introduced here, most real-time kernels use preemptive priority-based scheduling by default. The number of priority levels is usually 256, where 0 is the highest priority and 255 is the lowest priority. If a task with a priority higher than that of the current task enters the "Ready" state, then the current task's context is captured in the TCB and the kernel switches to the higher-priority task.

An example can be seen here.



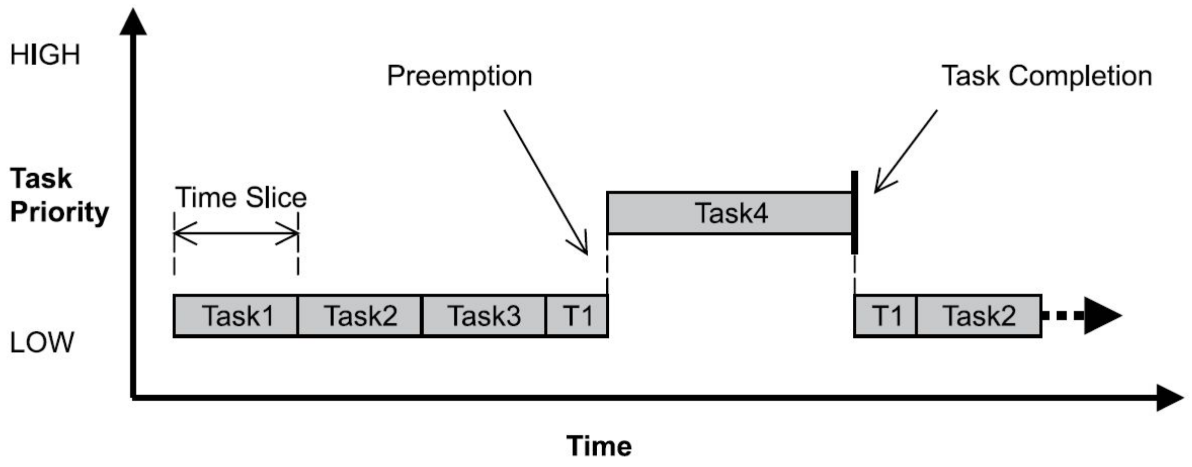
Preemptive Priority-Based Scheduling

Although tasks are assigned a priority when they are created, a task's 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

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 an equal allocation of the CPU for tasks of the same priority.

A run-time counter tracks the time slice for each task, incrementing on every clock tick. When one task's 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.



Priority Based Round Robin Scheduling

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.

MULTITASKING MODEL FOR RTOSs

What is discussed below is a complex model for real-time multi-tasking. The major features that distinguish it from the other prevalent models are what is mentioned below:

1. The explicit implementation of a scheduling policy in the form of a scheduler module. The schedule is itself a task which executes every time an internal or external interrupt occurs and computes the decision on making state transitions for every application task in the system that has been spawned and has not yet been terminated. It computes this decision based on the current priority level of the tasks, the availability of the various resources of the system, etc. The scheduler also computes the current priority levels of the tasks based on various factors such as deadlines, computational dependencies, waiting times, etc.
2. Based on the decisions of the scheduler, the dispatcher actually affects the state transition of the tasks by
   A. saving the computational state or context of the currently executing task from the hardware environment.
   B. enabling the next task to run by loading the process context into the hardware environment. It is also the responsibility of the dispatcher to make the short-term decisions in response to, e.g., interrupts from an input/output device or from the real-time clock.

The dispatcher/scheduler has two entry conditions:

1. The real-time clock interrupt and any interrupt which signals the completion of an input/output request
2. A task suspension due to a task delaying, completing or requesting an input/output transfer.

In response to the first condition, the scheduler searches for work starting with the highest priority task and checking each task in priority order. Thus if tasks with a high repetition rate are given a high priority they will be treated as if they were clock-level tasks, i.e., they will be run first during each system clock period.

In response to the second condition, a search for work is started at the task with the next lowest priority to the task which has just been running. There cannot be another higher priority task ready to run since a higher priority task becoming ready always preempts a lower priority running task.

PRIORITY LEVELS IN A TYPICAL RTOS

To ensure that response to every event is generated by executing tasks within specific deadlines which is crucial for an RTOS, it is essential to allocate the CPU and other computational resources to various tasks based on their priorities. The priority may be assigned statically based on some statistics or can be done dynamically, which has been seen to be more efficient.

The priority is assigned based on how quickly the task will have to respond to a particular event - a particular task or the elapsing of a particular amount of time. Tasks can be broadly categorized into three broad levels of priority:

Interrupt Level

Tasks at this level require very fast response measured in milliseconds and occur very frequently. There is no schedule at this level since immediate execution follows an interrupt. Obviously, to meet the deadlines of the other tasks in the system, the context switching and processing time requirements for these tasks are to be kept at the bare minimum level and must be highly predictable to make the whole system behavior predictable. To ensure the former, often, all Interrupt Service Routines (ISR) run in special common and fixed contexts, such as common stacks. The latter is ensured by a more complex mechanism, not in current scope. The **system clock** and **watchdog timers** associated with them are tasks that execute at interrupt level.

Hard Real-Time Level

At this level are the tasks which are periodic, such as the sampling and control tasks, and tasks which require accurate timing. The scheduling of these tasks is carried out based on the real-time system clock (Interrupt Level Task). A virtual Software Clock is also maintained based on interrupts generated by the system clock. Also every few clock cycles a new task gets dispatched according to the scheduling policy adopted. The lowest priority task at this level is the base level scheduler. Thus if at a clock level interrupt, the clock level scheduler finds no request for higher priority clock level tasks pending, the base level scheduler is dispatched.

Soft/Non-Real-Time Level

Tasks at this level are of soft or non-real-time in that they either have no deadlines to meet or are allowed a wide margin of error in their timing. These are therefore taken to be of low priority and executed only when no request for a higher priority task (Hard Real-Time Level Task) is pending. Tasks at this level may be allocated priorities or may all run at a single priority level - that of the base level scheduler in a round robin fashion. These tasks are typically initiated on demand rather than at some predetermined time interval. The demand may be a user input from a keypad, reading/writing to a file, etc.

KEY CHARACTERISTICS OF AN RTOS

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

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. The degree of reliability that is acceptable will depend on the application it is used on. A common way that developers categorize highly reliable systems is by quantifying their uptime/downtime per year. The percentages under the "Number of 9s" column indicate the percent of the total time that a system must be available.

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

Table: Categorizing highly available systems by allowable downtime

PREDICTABILITY

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

One definition of throughput is the rate at which a system can generate the output based on the inputs coming in. Throughput also means the amount of data transferred divided by the time taken to transfer it. The former can be quantified by Million Instructions per Second (MIPS) whereas the latter can be quantified by Bits per Second (bps).

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. 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 RTOSs can be used in a wide variety of embedded systems, they must be able to scale up or down to meet application-specific requirements.

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.

DIFFERENCE BETWEEN GPOS AND RTOS

The basic difference of using a GPOS or an RTOS lies in the nature of the system – i.e whether the system is “time critical” or not. A system can be of a single purpose or multiple purposes. We do see many embedded systems running GPOSs on them. 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.

Example of a “time-critical system” is an Automated Teller Machines (ATM). Here an ATM card user is supposed to get his money from the teller machine within 4 or 5 seconds from the moment he presses the confirmation button. The card user will not wait 5 minutes at the ATM after he pressed the confirm button. So an ATM is a time critical system. Whereas a personal computer (PC) is not a time-critical system. The purpose of a PC is multiple. A user can run many applications at the same time. After pressing the SAVE button of a finished document, there is no particular time limit that the doc should be saved within 5 seconds. It may take several minutes (in some cases) depending upon the number of tasks and processes running in parallel.

A GPOS is used for systems/applications that are not time critical. Example:- Windows, Linux, Unix, etc.

An RTOS is used for time-critical systems. Example:- VxWorks, uCos, etc.

TASK SCHEDULING

In the case of a GPOS – task scheduling is not based on “priority” always. GPOS is programmed to handle scheduling in such a way that it manages to achieve high throughput. Here throughput means the total number of processes that complete their execution per unit time. In such a case, sometimes the execution of a high priority process will get delayed in order to serve 5 or 6 low priority tasks. High throughput is achieved by serving 5 low priority tasks than by serving a single high priority one. Where in case of an RTOS, scheduling is always priority based. Most RTOS uses preemptive task scheduling method which is based on priority levels. Here a high priority process gets executed over the low priority ones. All “low priority process execution” will get paused. A high priority process execution will get override only if a request comes from an even high priority process.

HARDWARE AND ECONOMIC FACTORS

An RTOS is usually designed for a low end, a stand-alone device like an ATM, Vending machine, Kiosk, etc. RTOS is lightweight and small in size compared to a GPOS. A GPOS is made for a high end, general purpose system like a personal computer, a work station, a server system, etc. The basic difference between a low-end system and high-end system is in its hardware configuration. Nowadays a personal computer or even a smartphone comes with high-speed processors (in the range of many Gigahertz), large RAM's (in the range 2 or 3 GB's and even higher), etc. But an embedded system works on low hardware configurations usually in the speed in the range of Megahertz and RAM in the range of Megabytes. A GPOS being too heavy demands very high-end hardware configurations. It is economical to port an RTOS to an embedded system of limited expectations and functionalities (Example: An ATM is supposed to do only certain functions like Money transfer, Withdrawal, Balance check, etc). So it is more logical to use an RTOS inside the ATM with its limited hardware. It is not economical to improve the hardware of an ATM just to port a GPOS as it's the user interface.

LATENCY ISSUES

Another major issue with a GPOS is unbounded dispatch latency, which most GPOS falls into. The number of threads to schedule, latencies will get added up. An RTOS has no such issues because all the process and threads in it have got bounded latencies which means that a process/thread will get executed within a specified time limit.

PREEMPTIBLE KERNEL

The kernel of an RTOS is preemptible whereas a GPOS kernel is not preemptible. This is a major issue when it comes to serving a high priority process/threads first. If the kernel is not preemptible, then a request/call from kernel will override all other process and threads. For example:- a request from a driver or some other system service comes in, it is treated as a kernel call which will be served immediately overriding all other process and threads. In an RTOS the kernel is kept very simple and only very important service requests are kept within the kernel call. All other service requests are treated as external processes and threads. All such service requests from the kernel are associated with a bounded latency in an RTOS. This ensures a highly predictable and quick response from an RTOS.

TAKEAWAY POINTS

➔ RTOSs are best suited for real-time, application-specific embedded systems; GPOSs are typically used for general-purpose systems.
➔ RTOSs are programs that schedule execution in a timely manner, manage system resources, and provide a consistent foundation for developing application code.
➔ Kernels are the core module of every RTOS and typically contain kernel objects, services, and scheduler.
➔ Kernels can deploy different algorithms for task scheduling. The most common two algorithms are preemptive priority-based scheduling and round-robin scheduling.
➔ RTOSes for real-time embedded systems should be reliable, predictable, high performance, compact, and scalable.

REFERENCES

➔ Real-Time Concepts for Embedded Systems, Elsevier - Qing Li & Caroline Yao
➔ Industrial Embedded and Communication Systems, Real-Time Operating Systems: Introduction and Process Management, IIT Kharagpur NPTEL
➔ Real-Time Operating Systems, John A. Stankovic & R. Rajkumar
➔ https://en.wikipedia.org/wiki/Real-time_operating_system
➔ https://www.geeksforgeeks.org/operating-system-real-time-systems/