Real-Time Operating Systems: An Overview

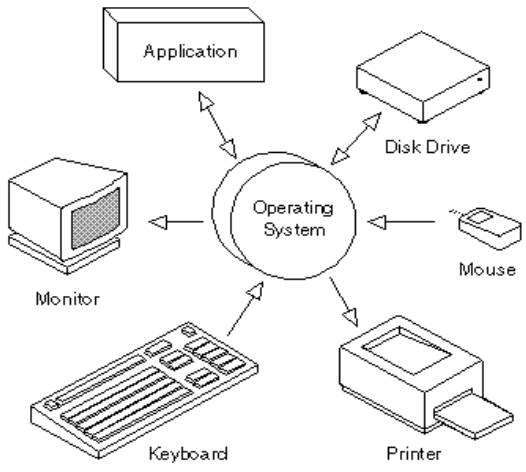
Sonali Grover,
Department of Information technology,
Dronacharya College of Engineering, Gurgaon, India

***Abstract*- A real-time operating system (RTOS) is an operating system intended to serve real-time application requests. It must be able to process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter. This paper summarizes what is a real-time operating system (RTOS), architecture and scheduling algorithms of real time operating systems and how they differ from standard general-purpose operating systems like Windows**.

***Index Terms*- Real-time operating system (RTOS), operating system, control applications, general-purpose operating systems.**

I. INTRODUCTION

An integrated operating system is a set of system software that organizes, manages and controls the resources and computing power of a computer or computer networks. In general, an operating system is responsible for managing the hardware resources of a computer and hosting applications that run on the computer.



An RTOS performs these tasks, but is also specially designed to run applications with very precise timing and a high degree of reliability. This can be especially important in measurement and automation systems where downtime is costly or a program delay could cause a safety hazard. A real-time operating system is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main objective of real-time operating systems is their quick and predictable response to events. A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is jitter. A hard real-time operating system has less jitter than a soft real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category. An RTOS that can usually or generally meet a deadline is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS. An RTOS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

Important Terminology and Concepts

Determinism: An application (or critical piece of an application) that runs on a hard real-time operating system is referred to as deterministic if its timing can be guaranteed within a certain margin of error.

Soft vs. Hard Real-Time: An OS that can absolutely guarantee a maximum time for the operations it performs is referred to as hard real-time. In contrast, an OS that can usually perform operations in a certain time is referred to as soft real-time.

Jitter: The amount of error in the timing of a task over subsequent iterations of a program or loop is referred to as jitter. Real-time operating systems are optimized to provide a low amount of jitter when programmed correctly; a task will take very close to the same amount of time to execute each time it is run. In the next sections, we examine certain important aspects of RTOSs.

This paper is organized as follows. The next section, Architecture of RTOS describes the entire architecture of RTOSs. The next section after that provides a brief description of some available RTOSs.

II. ARCHITECTURE OF RTOS

Kernel: RTOS kernel acts as an abstraction layer between the hardware and the applications. There are three broad categories of kernels

- **Monolithic kernel**

Monolithic kernels are part of Unix-like operating systems like Linux, FreeBSD etc. A monolithic kernel is one single program that contains all of the code necessary to perform every kernel related task. It runs all basic system services (i.e. process and memory management, interrupt handling and I/O communication, file system, etc) and provides powerful abstractions of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel.

- **Microkernel**

It runs only basic process communication (messaging) and I/O control. It normally provides only the minimal services such as managing memory protection, Inter process communication and the process management. The other functions such as running the hardware processes are not handled directly by microkernels. Thus, micro kernels provide a smaller set of simple hardware abstractions. It is more stable than monolithic as the kernel is unaffected even if the servers failed (i.e.File System). Microkernels are part of the operating systems like AIX, BeOS, Mach, Mac OS X, MINIX, and QNX. Etc

- **Hybrid Kernel**

Hybrid kernels are extensions of microkernels with some properties of monolithic kernels. Hybrid kernels are similar to microkernels, except that they include additional code in kernel space so that such code can run more swiftly than it would were it in user space. These are part of the operating systems such as Microsoft Windows NT, 2000 and XP. DragonFly BSD, etc

- **Exokernel**

Exokernels provides efficient control over hardware. It runs only services protecting the resources (i.e. tracking the ownership, guarding the usage, revoking access to resources, etc) by providing low-level interface for library operating systems and leaving the management to the application.

Six types of common services are listed below and explained in subsequent sections

- Task management and scheduling
- Interrupt and event Handling
- Memory management
- Task synchronization
- Device I/O management
- Time management

Task management and Scheduling: In typical designs, a task has three states:

1. Running (executing on the CPU);
2. Ready (ready to be executed);
3. Blocked (waiting for an event, I/O for example).

Most tasks are blocked or ready most of the time because generally only one task can run at a time per CPU. The number of items in the ready queue can vary greatly, depending on the number of tasks the system needs to perform and the type of scheduler that the system uses. On simpler non-preemptive but still multitasking systems, a task has to give up its time on the CPU to other tasks, which can cause the ready queue to have a greater number of overall tasks in the ready to be executed state (resource starvation).

Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

If there are no more than a few tasks on the ready list, then a doubly linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority. That way, finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted.

Care must be taken not to inhibit preemption during this search. Longer critical sections should be divided into small pieces. If an interrupt occurs that makes a high priority task ready during the insertion of a low priority task, that high priority task can be inserted and run immediately before the low priority task is inserted.

The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task to running. In a well-designed RTOS, readying a new task will take 3 to 20 instructions per ready-queue entry, and restoration of the highest-priority ready task will take 5 to 30 instructions.

In more advanced systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

Algorithms

Some commonly used RTOS scheduling algorithms are:

- Cooperative scheduling
- Preemptive scheduling
  - Rate-monotonic scheduling
  - Round-robin scheduling
  - Fixed priority pre-emptive scheduling, an implementation of preemptive time slicing
  - Fixed-Priority Scheduling with Deferred Preemption
  - Fixed-Priority Non-preemptive Scheduling
  - Critical section preemptive scheduling
  - Static time scheduling
- Earliest Deadline First approach
- Stochastic digraphs with multi-threaded graph traversal

Interrupt handling and scheduling: Since an interrupt handler blocks the highest priority task from running, and since real time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. The interrupt handler defers all interaction with the hardware if possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it won't occur again when the interrupt handler returns) and notify a task that work needs to be done. This can be done by unblocking a driver task through releasing a semaphore, setting a flag or sending a message. A scheduler often provides the ability to unblock a task from interrupt handler context.

An OS maintains catalogues of objects it manages such as threads, mutexes, memory, and so on. Updates to this catalogue must be strictly controlled. For this reason it can be problematic when an interrupt handler calls an OS function while the application is in the act of also doing so. The OS function called from an interrupt handler could find the object database to be in an inconsistent state because of the application's update. There are two major approaches to deal with this problem: the unified architecture and the segmented architecture. RTOSs implementing the unified architecture solve the problem by simply disabling interrupts while the internal catalogue is updated. The downside of this is that interrupt latency increases' potentially losing interrupts. The segmented architecture does not make direct OS calls but delegates the OS related work to a separate handler. This handler runs at a higher priority than any thread but lower than the interrupt handlers. The advantage of this architecture is that it adds very few cycles to interrupt latency. As a result, OSes which implement the segmented architecture are more predictable and can deal with higher interrupt rates compared to the unified architecture.

Memory management: Two types of memory managements are provided in RTOS – Stack and Heap. Stack management is used during context switching for TCBs. Memory other than memory used for program code, program data and system stack is called heap memory and it is used for dynamic allocation of data space for tasks. Management of this memory is called heap management.

Task synchronization: Synchronization is essential for tasks to share mutually exclusive resources (devices, buffers, etc.) and/or allow multiple concurrent tasks to be executed (e.g. Task A needs a result from task B, so task A can only run till task B produces it). Task synchronization is achieved using two types of mechanisms:

- Event Objects: Event objects are used when task synchronization is required without resource sharing. They allow one or more tasks to keep waiting for a specified event to occur. Event object can exist either in triggered or non-triggered state. Triggered state indicates resumption of the task.
- Semaphores: A semaphore has an associated resource count and a wait queue. The resource count indicates availability of resource. The wait queue manages the tasks waiting for resources from the semaphore. A semaphore functions like a key that define whether a task has the access to the resource. A task gets an access to the resource when it acquires the semaphore.

Device I/O management: RTOS generally provides large number of APIs to support diverse hardware device drivers.

Time management: Tasks need to be performed after scheduled durations. To keep track of the delays, timers- relative and absolute- are provided in RTOS.

III. EXAMPLES OF RTOS

Some of the best known, most widely deployed, real-time operating systems are:

- LynxOS
- OSE
- QNX
- RTLinux
- VxWorks
- Windows CE
- FreeRTOS

LynxOS: The LynxOS RTOS is a Unix-like real-time operating system from Lynx Software Technologies. Sometimes, known as Lynx Operating System, LynxOS features full POSIX conformance and, more recently, Linux compatibility. LynxOS is mostly used in real-time embedded systems, in applications for avionics, aerospace, the military, industrial process control and telecommunications.

RTLinux: RTLinux is a hard real time RTOS microkernel that runs the entire Linux operating system as a fully preemptive process. It is one of the hard real-time variants of Linux, among several, that makes it possible to control robots, data acquisition systems, manufacturing plants, and other time-sensitive instruments and machines.

Windows CE: Microsoft Windows CE 5.0 is an open, scalable, 32-bit operating system (OS) that integrates reliable, real time capabilities with advanced Windows technologies. Windows CE allows you to build a wide range of innovative, small footprint devices. A typical Windows CE–based device is designed for a specific use, often runs disconnected from other computers, and requires a small OS that has a deterministic response to interrupts. Examples include enterprise tools, such as industrial controllers, communications hubs, and point-of-sale terminals, and consumer products, such as cameras, Internet appliances, and interactive televisions.

FreeRTOS: FreeRTOS is a market leading real time operating system (or RTOS) from Real Time Engineers Ltd. that supports 35 architectures and receives 107000 downloads a year. It is professionally developed, strictly quality controlled, robust, supported, and free to use in commercial products without any requirement to expose your proprietary source code. It is used in every imaginable market sector from toys to aircraft navigation.

IV. How RTOS Differs From General-Purpose OS

Operating systems such as Microsoft Windows and Mac OS can provide an excellent platform for developing and running your non-critical measurement and control applications. However, these operating systems are designed for different use cases than real-time operating systems, and are not the ideal platform for running applications that require precise timing or extended up-time. This section will identify some of the major under-the-hood differences between both types of operating systems, and explain what you can expect when programming a real-time application.

Setting Priorities

When programming an application, most operating systems (of any type) allow the programmer to specify a priority for the overall application and even for different tasks within the application (threads). These priorities serve as a signal to the OS, dictating which operations the designer feels are most important. The goal is that if two or more tasks are ready to run at the same time, the OS will run the task with the higher priority.

In practice, general-purpose operating systems do not always follow these programmed priorities strictly. Because general-purpose operating systems are optimized to run a variety of applications and processes simultaneously, they typically work to make sure that all tasks receive at least some processing time. As a result, low-priority tasks may in some cases have their priority boosted above other higher priority tasks. This ensures some amount of run-time for each task, but means that the designer's wishes are not always followed.

In contrast, real-time operating systems follow the programmer's priorities much more strictly. On most real-time operating systems, if a high priority task is using 100% of the processor, no other lower priority tasks will run until the high priority task finishes. Therefore, real-time system designers must program their applications carefully with priorities in mind. In a typical real-time application, a designer will place time-critical code (e.g. event response or control code) in one section with a very high priority. Other less-important code such as logging to disk or network communication may be combined in a section with a lower priority.

Interrupt Latency

Interrupt latency is measured as the amount of time between when a device generates an interrupt and when that device is serviced. While general-purpose operating systems may take a variable amount of time to respond to a given interrupt, real-

time operating systems must guarantee that all interrupts will be serviced within a certain maximum amount of time. In other words, the interrupt latency of real-time operating systems must be bounded.

Performance

One common misconception is that real-time operating systems have better performance than other general-purpose operating systems. While real-time operating systems may provide better performance in some cases due to less multitasking between applications and services, this is not a rule. Actual application performance will depend on CPU speed, memory architecture, program characteristics, and more.

Though real-time operating systems may or may not increase the speed of execution, they can provide much more precise and predictable timing characteristics than general-purpose operating systems.

REFERENCES

[1] Wikipedia, Real time operating systems http://en.wikipedia.org/wiki/Real-time_operating_system

[2] Real time operating systems Dedicated Systems Encyclopedia.http://www.realtime-info.be/encyc/buyersguide/rtos/rtosmenu.htm.

[3] Microsoft Technical Document, "Real-Time Systems with Microsoft Windows CE", Available at, http://www.eu.microsoft.com/windows/embedded/ce/resources/howitworks/realtime.asp.

[4] OSE, "OSE Realtime Kernel", http://www.ose.com/PDF/rtk.pdf

[5] FreeRTOS, http://www.freertos.org