A Survey of Real-time Operating Systems

Abstract

A real-time operating system (RTOS) supports real-time applications and embedded systems. Real-time applications have the requirement to meet task deadlines in addition to the logical correctness of the results. In this report, we review the pre-requisites for an RTOS to be POSIX 1003.1b compliant and discuss memory management and scheduling in RTOS. We survey the prominent commercial and research RTOSs and outline steps in system implementation with an RTOS. We select a popular commercial RTOS for each category of real-time application and discuss its real-time features. A comparison of the commercial RTOSs is also presented. We conclude by discussing the results of the survey and suggest future research directions in the field of RTOS.

1 Introduction

A real-time system is one whose correctness involves both the logical correctness of the outputs and their timeliness [11]. A real-time system must satisfy bounded response-time constraints; otherwise risk severe consequences, including failure. Real-time systems are classified as hard, firm or soft systems. In hard real-time systems, failure to meet response-time constraints leads to system failure. Firm real-time systems are those systems with hard deadlines, but where a certain low probability of missing a deadline can be tolerated. Systems in which performance is degraded but not destroyed by failure to meet response-time constraints are called soft real-time systems. A real-time system is called an embedded system when the software system is encapsulated by the hardware it controls. The microprocessor system used to control the fuel/air mixture in the carburetor of many automobiles is an example of a real-time embedded system. An RTOS differs from common OS, in that the user when using the former has the ability to directly access the microprocessor and peripherals. Such an ability of the RTOS helps to meet deadlines.

The organization of this report is as follows. In Section 2, we discuss the basic requirements of an RTOS to be POSIX 1003.1b compliant. In Section 3, we review memory management and scheduling algorithms used in an RTOS. In Section 4, we outline and briefly explain the steps in the implementation of a project with an RTOS. In Section 5, we select a popular RTOS in each of the different categories of real-time applications and discuss the real-time features of the selected RTOS. We also compare the different contemporary commercial RTOSs. In Section 6, we conclude by discussing the results of this survey and our suggestions for future research in the field of RTOS.

2 Features

The kernel is the core of the OS that provides task scheduling, task dispatching and inter-task communication. In embedded systems, the kernel can serve as an RTOS while commercial RTOSs like those used for air-traffic control systems require all of the functionalities of a general purpose OS. The desirable features of an RTOS include the ability to schedule tasks and meet deadlines, ease of incorporating external hardware, recovery from errors, fast switching among tasks and small size and small overheads. In this section we discuss the basic requirements of an RTOS and the POSIX standards for an RTOS.

2.1 Basic requirements

The following are the basic requirements of an RTOS.

(i) Multi-threading and preemptibility

To support multiple tasks in real-time applications, an RTOS must be multi-threaded and preemptible. The scheduler should be able to preempt any thread in the system and give the resource to the thread that needs it most. An RTOS should also handle multiple levels of interrupts i.e., the RTOS should not only be preemptible at thread level, but at the interrupt level as well.

(ii) Thread priority

In order to achieve preemption, an RTOS should be able to determine which thread needs a resource the most, i.e., the thread with the earliest deadline to meet. Ideally, this should be done at run-time. However, in reality, such a deadline-driven OS does not exist. To handle deadlines, each thread is assigned a priority level. Deadline information is converted to priority levels and the OS allocates resources according to the priority levels of threads. Although the approach of resource allocation among competing threads is prone to error, in absence of another solution, the notion of priority levels is used in an RTOS.

(iii) Predictable thread synchronization mechanisms

For multiple threads to communicate among each other, in a timely fashion, predictable inter-thread communication and synchronization mechanisms are required. Also, supported should be the ability to lock/unlock resources to achieve data integrity.

(iv) Priority inheritance

When using priority scheduling, it is important that the RTOS has a sufficient number of priority levels, so that applications with stringent priority requirements can be implemented [13]. Unbounded priority inversion occurs when a higher priority task must wait on a low priority task to release a resource while the low priority task is waiting for a medium priority task. The RTOS can prevent priority inversion by giving the lower priority task the same priority as the higher priority task that is being blocked (called priority inheritance). In this case, the blocking task can finish execution without being preempted by a medium priority task. The designer must make sure that the RTOS being used prevents unbounded priority inversion [10].

(v) Predefined latencies

An OS that supports a real-time application needs to have information about the timing of its system calls. The behavior metrics to be specified are:

- **Task switching latency:** Task or context-switching latency is the time to save the context of a currently executing task and switch to another task. It is important that this latency be short.
- **Interrupt latency:** This is the time elapsed between the execution of the last instruction of the interrupted task and the first instruction in the interrupt handler, or simply the time from interrupt to task run [6]. This is a metric of system response to an external event.

- **Interrupt dispatch latency:** This is the time to go from the last instruction in the interrupt handler to the next task scheduled to run. This indicates the time needed to go from interrupt level to task level.

2.2 POSIX compliance

IEEE Portable Operating System Interface for Computer Environments, POSIX 1003.1b (formerly 1003.4) provides the standard compliance criteria for RTOS services and is designed to allow application programmers to write applications that can easily be ported across OSs. The basic RTOS services covered by POSIX 1003.1b include:

- **Asynchronous I/O:** The ability to overlap application processing and application initiated I/O operations [8].
- **Synchronous I/O: T**he ability to assure return of the interface procedure when the I/O operation is completed [8].
- **Memory locking:** The ability to guarantee memory residence by storing sections of a process that were not recently referenced on secondary memory devices [24].
- **Semaphores:** The ability to synchronize resource access by multiple processes [19].
- **Shared memory:** The ability to map common physical space into independent process specific virtual space [8].
- **Execution scheduling:** Ability to schedule multiple tasks. Common scheduling methods include round robin and priority-based preemptive scheduling.
- **Timers:** Timers improve the functionality and determinism of the system. A system should have at least one clock device (system clock) to provide good real-time services. The system clock is called CLOCK_REALTIME when the system supports Real-time POSIX [13].
- **Inter-process Communication (IPC):** IPC is a mechanism by which tasks share information needed for a particular application. Common RTOS communication methods include mailboxes and queues.
- **Real-time files:** The ability to create and access files with deterministic performance.
- **Real-time threads:** Real-time threads are schedulable entities of a real-time application that have individual timeliness constraints and may have collective timeliness constraints when belonging to a runnable set of threads [13].

3 Memory management and scheduling

In this section, we discuss the various memory management and scheduling schemes adopted in RTOSs.

3.1 Memory management

Commonly an RTOS achieves small memory footprint by including only the functionality needed for the user's applications and discarding the rest [26]. There are two types of memory management in RTOSs.

The first type is used to provide tasks with temporary data space. The system's free memory is divided into fixed sized memory blocks, which can be requested by tasks. When a task finishes using a memory block it must return it to the pool. Another way to provide temporary space for tasks is via priorities. A pool of memory is dedicated to high priority tasks and another to low priority tasks. The high-priority pool is sized to have the worst-case memory demand of the system. The low priority pool is given the remaining free memory. If the low priority tasks exhaust the low priority memory pool, they must wait for memory to be returned to the pool before further execution [1].

The second type of memory management is used to dynamically swap code in and out of main memory. Specific techniques are memory swapping, overlays, multiprogramming with a fixed number of tasks (MFT), multiprogramming with a variable number of tasks (MVT) and demand paging. The memory swapping method keeps the OS and one task in memory at the same time. When another task needs to run, it replaces the first task in main memory, after the first task and its context have been saved to secondary memory. When using overlays, the code is partitioned into smaller pieces, which are swapped from disk to memory. In this way, programs larger than the available memory can be executed. In MFT, a fixed number of equalized code parts are in memory at the same time. As needed, these parts are overlaid from disk. MVT is similar to MFT except that the size of the partition depends on the needs of the program in MVT. Demand paging systems have fixed-size "pages" that are given to programs as needed in non-continuous memory. Demand paging differs from MFT and MVT because the latter two can be put only in continuous memory blocks [11].

In many embedded systems, the kernel and user execute in the same space i.e., there is no memory protection. Hence, a system and a procedure or function call within an application are indistinguishable.

3.2 Scheduling algorithms of RTOS

For small or static real-time systems, data and task dependencies are limited and therefore the task execution time can be estimated prior to execution and the resulting task schedules can be determined off-line. Periodic tasks typically arise from sensor data and control loops, however sporadic tasks can arise from unexpected events caused by the environment or by operator actions. A scheduling algorithm in RTOS must schedule all periodic and sporadic tasks such that their timing requirements are met.

The most commonly used static scheduling algorithm is the Rate Monotonic (RM) scheduling algorithm of Liu and Layland [12]. The RM algorithm assigns different priorities proportional to the frequency of tasks. RM can schedule a set of tasks to meet deadlines if total resource utilization is less than 69.3%. If a successful schedule cannot be found using RM, no other fixed priority scheduling system will avail. But the RM algorithm provides no support for dynamically changing task periods and/or priorities and tasks that may experience priority inversion. Priority inversion occurs in an RM system where in order to enforce rate-monotonicity, a non-critical task with a high frequency of execution is assigned a higher priority than a critical task with lower frequency of execution. A priority ceiling protocol (PCP) can be used to counter priority inversion, wherein a task blocking a higher priority task inherits the higher priority for the duration of the blocked task.

Earliest deadline first (EDF) scheduling can be used for both static and dynamic real-time scheduling. Its complexity is $O(n^2)$, where n is the number of tasks, and the upper bound of process utilization is 100% [11]. An extension of EDF is the time-driven scheduler. This scheduler aborts new tasks if the system is already overloaded and removes low-priority tasks from the queue. A variant of EDF is Minimum Laxity First (MLF) scheduling where a laxity is assigned to each task in the system and minimum laxity tasks are executed first. MLF considers the execution time of a task, which EDF does not. Another variant of EDF is the Maximum Urgency First (MUF) algorithm, where each task is given an explicit description of urgency.

The cyclic executive is used in many large-scale dynamic real-time systems [3]. Here, tasks are assigned to a set of harmonic periods. Within each period, tasks are dispatched according to a table that lists the order to execute tasks. No start times need be specified, but a prior knowledge of the maximum requirements of tasks in each cycle must be known.

One disadvantage of dynamic real-time scheduling algorithms is that even though deadline failures can be easily detected, a critical task set cannot be specified and hence there is no way to specify tasks that are allowed to fail during a transient overload.

3.2.1 Scheduling for hard real-time systems

An increasing proportion of all computers do not sit in air-conditioned computer centers or even on desktops; they are embedded in automobiles, lathes, microwave ovens, cloth dryers, aluminum rolling mills and airplane cockpits. The software system in these computer systems must meet hard real-time deadlines, e.g., a flight control surface must be adjusted several times each second to keep some new aircraft stable.

For the class of hard real-time systems, mechanisms and policies that ensure consistency and minimize worst-case blocking without incurring any unbounded or excessive run-time overheads are desired. Since most recent work in maintaining integrity of shared data has been carried out in the context of database systems, one can consider adapting database concurrency control techniques to the domain of hard real-time systems. But since virtually all database concurrency control approaches have been designed to optimize average-case performance rather than worst-case latency, these techniques must be adapted and extended for hard real-time systems. The techniques adapted must employ semantic information that is necessarily available at design time to guarantee optimum scheduling.

4 System implementation with RTOS

Implementation of a system using an RTOS requires calculation and planning. The designer has to consider all the timing aspects of the system. Based on timing calculations, and the task partitioning used, the designer can decide if the desired RTOS can provide the needed capabilities. In addition, the designer must consider task prioritization, use of interrupts, multiprocessor support, if applicable, as well as language support by the chosen RTOS.

4.1 Response time

The system should respond with an output before the next input. Therefore, the system’s response time should be shorter than the minimum time between successive inputs.

4.2 Task partitioning

After determining the required response time, the designer continues by dividing the project into tasks. The designer must balance the amount of parallelism and communication [5]. Task cohesion criteria are used to optimize partitioning by combining parallel tasks to execute sequentially. Combining tasks minimizes overhead by reducing the context switches and inter-task communications.

4.3 RTOS considerations

After a designer has completed task partitioning, it must be determined if an RTOS is capable of handling the set of tasks. The key considerations are the available timer period, inter-task communication methods, contention resolution, and memory protection.

4.4 Task priority

The priority assigned to each task is essential for proper operation of an application. Starvation occurs when higher priority tasks are always ready to run, resulting in insufficient processor time for lower priority tasks [13]. The designer must determine which tasks are critical in meeting the deadlines and give them the highest priorities. However, when execution time is at a premium, tasks, which do not contribute in meeting real-time deadlines of the system, may not get a "fair" amount of execution time compared to time-critical tasks.

4.5 Interrupts

When designing a system that uses non-prioritized interrupts, the designer must ensure that interrupt-handling time is minimized. If possible, the interrupt handler should save the context, create a task that will handle the interrupt service, and return control back to the operating system. Using a task to perform bulk of the interrupt service allows the service to be performed based on a priority chosen by the designer. It helps preserve the priority system of the RTOS. Without preservation of priorities, a lower priority event can cause an interrupt during execution of a high priority task causing missed deadlines. Non-prioritized interrupts should not be used if there is a task that cannot be preempted without causing system failure.

In systems where interrupts are used, the designer must also consider the input of the RTOS detecting the interrupts. Typically, when an RTOS is performing system operations, such as determining which task should execute next, it will turn the interrupts off. The time period during which interrupts are turned off is called the "interrupt latency" of the RTOS. During the interrupt latency time, interrupts can be delayed or even lost. It is preferable that a RTOS with a small interrupt latency be used in a system where delaying or missing an interrupt is not acceptable.

4.6 Multiprocessor RTOS

Embedded multiprocessor systems typically have a processor controlling each device in the system. Most RTOSs that are multiprocessor-capable use a separate instance of the kernel on each processor. The multiprocessor ability comes from the kernel's ability to send and receive information between processors. In many RTOSs that support multiprocessing, there is no difference between the single processor case and the multiprocessor case from the task's point of view. The RTOS uses a table in each local kernel that contains the location of each task in the system. When one task sends a message to another task the local kernel looks up the location of the destination task and routes the message appropriately. From the task's point of view all tasks are executing on the same processor [6].

4.7 Language support

The RTOS should reduce the programmer's coding burden by handling resource management. A language that directly supports synchronization primitives such as *SCHEDULE*, *SIGNAL* and *WAIT*, etc. greatly simplifies the translation from design to code. The *SCHEDULE* command schedules a process based on time or an event; *SIGNAL* and *WAIT* commands manipulate a *semaphore* that enables concurrent tasks to be synchronized.

5 Categories of RTOS

In this section, we select a prominent commercial RTOS for each category of real-time applications and discuss its features. But first, we list the common capabilities of these operating systems:

5.1 Commonalities of commercial Real-time Operating Systems

- **Speed and efficiency:** Most RTOSs are microkernels that have low overhead. In some, no context switch overhead is incurred in sending a message to the system service provider.
- **System calls:** Certain portions of system calls are non-preemptable for mutual exclusion. These parts are highly optimized, made as short and deterministic as possible.
- **Scheduling:** For POSIX compliance, all RTOSs offer at least 32 priority levels. Many offer 128 or 256 while others offer even 512 priority levels.
- **Priority inversion control:** Many operating systems support resource access control schemes that do not need priority inheritance. This avoids the overhead of priority inheritance.
- **Memory management:** Support for virtual memory management exists but not necessarily paging. The users are offered choices among multiple levels of memory protection.

5.2 RTOS for small footprint, mobile and connected devices

In this section, we outline the real-time features of ***Windows CE 3.0*** [15], a highly modular real-time embedded OS for small footprint, mobile 32-bit intelligent connected devices. Windows CE 3.0 can work on 12 different processor architectures. It can be customized to meet specific product requirements with a minimum footprint of 400KB.

Windows CE 3.0 provides quantum-level thread control (the OS divides CPU time among the threads in timeslices or quantum) and 256 priority levels thus facilitating control over the scheduling and behavior of embedded systems. To optimize performance all threads are enabled to run in kernel mode.

Windows CE 3.0 supports system calls within interrupt service threads (ISTs). Nested interrupts are supported. Fast high-priority thread response helps to know when thread transitions occur.

Windows CE 3.0 kernel has the following features:

- Timer accuracy is 1 ms for Sleep and Wait related APIs.
- While executing non-preemptive code in the kernel, translation look-aside buffer (TLB) misses are avoided by moving all kernel data into physical memory.
- *Kcalls*, all non-preemptible but interruptible portions of the kernel, are broken down into small non-preemptible sections. Although complexity is increased by increased number of sections, preemption is turned off for short periods.
- All kernel objects (such as processes, threads, critical sections, mutexes, events and semaphores) are allocated in virtual memory and thus the memory for these objects is allocated on demand.

It uses a memory management unit (MMU) for virtual memory management. The use of multiple execute-in-place (XIP) regions eliminates boot time, avoids double footprint and reduces hardware requirements.

The use of an OEM[*] Adaptation Layer (OAL) isolates device dependent routines to increase OS portability. Hardware-assisted debugging enables the debugging of the OAL before

[*] Original Equipment Manufacturer

the kernel starts running. The OEM can specify the modules and processes that are trusted on a particular platform. This model prevents unauthorized applications from accessing system Application Programming Interfaces (APIs) and potentially damaging the platform.

5.3 RTOS for complex, hard real-time applications

In this section, we discuss LynxOS [14], a POSIX-compatible, multiprocess, multithreaded OS designed for complex real-time applications that require fast and deterministic response. LynxOS is scalable RTOS from large and complex switching systems down to small-embedded products.

LynxOS 3.0 has moved from the monolithic architecture to a microkernel design. The microkernel is 28 KB in size and provides essential services like scheduling, interrupt dispatch and synchronization. Other services are offered by the kernel lightweight service modules, called the Kernel Plug-Ins (KPIs). With the addition of KPIs to the microkernel, the system can be configured to suport TCP/IP streams, I/O and file systems, sockets, etc. The KPIs are multithreaded and there is no context switch when sending a message to a KPI, and inter-KPI communication takes only a few instructions [2].

LynxOS provides common code base across different microprocessor families. In response to an interrupt, LynxOS kernel dispatches a kernel thread, which can be prioritized and scheduled as any other thread in the system. Thus the priority of the kernel thread that executes a scheduled interrupt handling routine is the priority of the user thread that handles the interrupting device [2]. Kernel threads allow interrupt routines to be short and fast. In other words, kernel threads ensure predictable response even in the presence of heavy I/O. LynxOS provides memory protection through hardware memory management units (MMUs) but also offers optional demand paging.

LynxOS uses numerous scheduling policies such as prioritized FIFO, dynamic deadline monotonic scheduling, prioritized round robin and time-slicing etc. LynxOS offers 512 thread priority levels with typical thread switch latency between 4µs to 19µs.

Linux applications need to be recompiled in order to run on RTOSs such as QNX. With LynxOS' Application Binary Interface (ABI) compatibility [22] a Linux program's binary image can be run directly on LynxOS. LynxOS includes the AT&T System V.3 and 4.3 BSD system call interfaces and libraries, which provide a high degree of source-level compatibility for applications written in either flavor of UNIX.

Unlike many embedded RTOS, LynxOS supports memory protection. LynxOS also provides support for diskless remote operation as well as boot capability.

5.4 General purpose RTOS in the embedded industry

In this section, we discuss VxWorks [25], the most widely adopted RTOS in the embedded industry. VxWorks is the fundamental run-time component of Tornado II, a visual, automated and integrated development environment for embedded systems. VxWorks is a flexible, scalable RTOS with over 1800 APIs and is available on all popular CPU platforms.

VxWorks comprises the core capabilities of network support, file system, I/O management, and other standard run-time support. The microkernel supports a full-range of real-time features including 256 priority levels, multitasking, deterministic context switching and preemptive and round robin scheduling. Binary and counting semaphores and mutual exclusion with inheritance are used for controlling critical system resources.

VxWorks is designed for scalability, which enables developers to allocate scarce memory resources to their application rather than to the OS. Portability requires a distinct separation of low-level hardware dependent code from high-level application or OS code. A Board Support

Package (BSP) represents the hardware-dependent layer. A BSP is required for any target board that executes VxWorks.

TCP, UDP, sockets and standard Berkeley network services can all be scaled in or out of the networking stack as necessary. VxWorks supports ATM, SMDS, frame relay, ISDN, IPX/SPX, AppleTalk, RMON, web-based solutions for distributed network management and CORBA for distributed computing environments.

VxWorks [25] supports virtual memory configuration. It is possible to choose to have only virtual address mapping, to have text segments and exception vector tables write protected, and to give each task a private virtual memory upon request.

RainFront [17] provides a highly available and load-balancing platform for embedded systems built using VxWorks. VxWorks is the embedded RTOS used in networking equipments running Voice over IP (VoIP) and Fax over IP (FoIP) [7]. The CompactNET multiprocessing technology [4] supports processing with VxWorks.

5.5 RTOS for the Java Platform

The Jbed RTOS package [9] is a real-time capable virtual machine developed for embedded systems and Internet applications under the Java platform. It allows an entire application including the device drivers to be written using Java. Instead of interpreting the bytecode, the Jbed RTOS translates the bytecode to fast machine code prior to downloading or class loading.

The component-based architecture allows loading of code dynamically and makes Jbed scalable from small ROM-based devices to high performance devices connected to the Internet. Jbed also facilitates real-time memory allocation, exception handling and automatic object destruction. Jbed real-time class library supports hard real-time applications.

Jbed Light is a smaller, low-cost version for fast and precompiled standalone applications. It contains the basic components including the core Jbed virtual machine, a small set of standard Java libraries, and the Jbed libraries required to directly access peripherals.

The Java virtual machine calls are directly implemented in the kernel. This avoids the need for a slow and complex Java Native Interface (JNI), which would otherwise be needed to make system calls. Also no adapter is needed to translate between the Java and native OS threads.

Jbed runs on *32*-bit microprocessors and controllers. Current versions support ARM7, 68k and the PowerPC architectures. The Jbed RTOS supports up to 10-thread priority levels. The thread switch latency and maximum interrupt latency are processor dependent. The standard Java thread API is suitable only for soft real-time parts of an application. Additional thread API is added in Jbed support hard real-time features of an application. The scheduling policy of the hard real-time threads is Earliest Deadline First, which is widely applicable for periodic, harmonic and sporadic tasks.

5.6 Objected-oriented RTOS

pSOSystem is a modular object-oriented operating system. The objects in pSOS include tasks, memory regions, message queues, and semaphores. Objects may be global or local. A global object can be accessed from any processor in the system, while a local object can be accessed only by tasks on its local processor. Node of residence is the processor on which the system call that created an object was made.

pSOS schedules a task in a preemptive priority-driven or time-driven fashion like EDF. User tasks can be chosen by the application developer to run in either user or supervisory mode. It supports both priority inheritance and priority-ceiling protocol.

The application developer is given complete control over interrupt handling. Device drivers can be loaded and removed at run-time. During an interrupt, the processor jumps directly to the interrupt service routine pointed to by the vector table.

A memory region is a physically contiguous block of memory, created in response to a call from an application. pSOS allocates memory regions to tasks. Like all objects, a memory region may be local (i.e., strictly in local memory) or global.

5.7 Real-time features of general purpose operating systems

In this section, we review the real-time features of two common general purpose operating systems viz., Windows NT and UNIX. Windows NT (in this report, we base our discussion with reference to Microsoft Windows NT operating system Version 4.0) on an Intel platform clearly delivers many of the open system promises that UNIX systems failed to: binary compatibility, market acceptance, a common development environment and ubiquitous third-party software. Table 1 shows the comparison between the real-time features of Windows NT and native UNIX.

Real-time feature	**Windows NT**	**Native UNIX**
Preemptive, priority-based multitasking	Yes	Yes
Interrupt threads (Deferred Procedure Calls in NT)	Yes	No
Non-degrading, real-time priorities	Yes	No
Processor isolation/ processor binding	Some	No
Locking virtual memory	Yes	Yes
Precision of timers	1 millisecond	10 milliseconds
Asynchronous I/O	Yes	No

Table 1. Comparison of real-time features in Windows NT and native UNIX

- Preemption: Even though the Windows NT kernel in general is non-preemptable, there exist certain points within the kernel where a process can be preempted. Native UNIX on the other hand, disables preemption any time a system call is performed or an interrupt service routine is executed.
- Deferred procedure calls (DPCs): DPCs in Windows NT permit the kernel to defer major portions of interrupt processing to a later point in time decided by its scheduling mechanisms. Since interrupt service routines (ISRs) disable other interrupts while executing, using DPCs or interrupt threads permits interrupts to be responded to at more regular intervals.
- Non-degrading real-time priorities: These are the priorities that are not dynamically altered by the operating system. The scheduler to ensure fairness to all activities of the system constantly manipulates normal thread priorities for UNIX and Windows NT. Both Windows NT provides a band of interrupt priorities that are fixed – unaltered by the kernel under any circumstances.
- Processor isolation/processor binding: This feature is advantageous in multiprocessor systems to help isolate real-time activities from non-real-time activities of the operating system. Windows NT has processor isolation and process binding capabilities but lacks the ability to eliminate or minimize interprocessor synchronization interrupts on isolated CPUs.

5.7.1 Interrupt handling in Windows NT

Even though Windows NT provides very fast response times, it is not as deterministic as a hard RTOS [1. This is evident from how the kernel handles interrupts. Assume that a user thread is blocked awaiting the completion of an I/O request. When an interrupt occurs to notify the system that the I/O request can be fulfilled, it is first handled by an interrupt service routine (ISR) that is part of the device driver written for the interrupting device. An ISR will simply post a request for a DPC to be queued and then relinquish the CPU. The DPC will run at a later time on behalf of the ISR and would complete the I/O request and notify the user thread that the request is complete.

All DPCs are added to a first-in, first-out (FIFO) queue of pending DPCs. Once, executing a DPC will run to completion. ISRs always run before DPCs and DPCs always run before user threads. The user thread becomes ready to run once the DPC has fulfilled the I/O request, but it is not dispatched until there are no ISRs executing and the DPC queue is empty. The major flaws in the mechanism explained above include: (a) DPCs executing in FIFO order instead of priority (b) user threads not sharing priority space with DPCs (c) DPCs not preemptable by other DPCs or threads and (d) developers having no control over third-party drivers.

In Windows NT, it is not possible for a user-level thread to execute at a higher priority than ISRs or DPCs. In other words, even low-priority ISRs, such as mouse and keyboard handlers will be able to preempt real-time processes. There are two classes of thread priorities: a real-time class and a dynamic class. Real-time class threads operate with fixed priorities that are not altered by the kernel. There are 16 priority levels in the real-time class. But any given thread is restricted only to a subset of priorities in the range of (+ or -) 2 levels of its initial priority plus the two extreme priorities of the class.

The Windows NT kernel has no support for priority inheritance, so deadlocks can occur when using real-time priorities. On heavily loaded systems, high-priority real-time processes could potentially be blocked indefinitely. Additionally, Windows NT has no support for priority queuing in its inter-thread communication mechanisms. In other words, if there are multiple threads at multiple priorities blocked waiting for a resource, the threads will be granted access to that resource in FIFO order rather than in priority order. Conversely, a RTOS queues the threads according to priority.

5.8 Other commercial RTOS

The following table lists the other widely used commercial RTOSs and their main features with respect to the five basic requirements of an RTOS as explained in Section 2.

RTOS, *Vendor*	**Scheduling**	**Thread priority levels**	**Synchronization mechanisms**	**Priority inversion prevention provided**	**Development hosts, kernel characteristics and behavior metrics**
AMX *KADAK Products Limited.*	Preemptive	N/A	Mailbox or Message exchange manager; wait-wake requests	Yes	Windows, portable. A pre-configured collection of AMX tasks and AMX compatible device drivers form the basis of the Palm OS; predictable memory block availability

C Executive
 	
JMI Software Systems, Inc.
	Prioritized FIFO, time slicing	32000	64 system calls; messages and dynamic data queues	Yes	Windows, Solaris.
					95% ANSI portable C; ROM resident kernel;
					Thread switch latency: 3µs;
					Maximum Interrupt latency: 2µs

CORTEX *Australian Real-time Embedded Systems.*	Prioritized FIFO, prioritized round-robin, time slicing	62	Recursive resource locks, mutexes, monitors and counting semaphores	Yes, uses priority ceiling	Windows, Solaris, Linux. CPU-independent software interrupt manager; statically and dynamically segmented memory models, high degree of configurability.

Delta OS
 	
CoreTek Systems, Inc.
	Prioritized round-robin	256	Semaphores, timers and message queues	Yes	Windows, Linux.
					Thread switch latency: 23µs;
					Maximum interrupt latency: 13µs; 1µs clock resolution;
					System calls made from Interrupt service routines (ISR) return to the ISR, eliminating time-consuming kernel scheduling mechanisms.

ECos *RedHat, Inc.*	Prioritized FIFO, Bitmap	1 to 32	Rich set of synchronous primitives including timers and counters	Yes, uses priority ceiling	Windows, RedHat Linux 5.x, 6.x. For soft real-time uses; Supports EL/IX Level 1, a Linux compatibility interface for embedded applications in small devices
embOS *SEGGER Microcontroller Systems.*	Prioritized round-robin	255	Mailbox, binary and counting semaphore	No	Windows, Linux. Uses profiling to collect precise timing information for every task; System analysis via UART; task activation time is independent of number of tasks.
eRTOS *JK Microsystems, Inc.*	Prioritized round-robin	256	Inter-thread messaging (messages and queues), semaphores	No	Windows, DOS, OS/2. High-speed interrupt driven serial port routines; Provides a fast, general-purpose mathematical library called eMath.

INTEGRITY *GreenHills Software, Inc.*	Prioritized round-robin, ARINC 653	255, but configurable	Semaphores; break-points can be placed any where in the system including ISRs.	Yes, using mutex, highest locker semaphore	Used in mission critical embedded applications; object-oriented design; supports distributed processing and dynamic downloading of user applications; per task and system wide execution profiling.
IRIX *SGI*	Prioritized FIFO, round-robin	255	Message queues	Yes	SGI. Double-precision matrix support; Multi-pipe scalability; multiple root partitions on a system disk; supports scheduled transfer protocol (STP).
Nuclear Plus *Accelerated Technology, Inc.*	Prioritized FIFO	N/A	Mailboxes, pipes and queues can be created dynamically as required	Yes	Windows. Highly portable, functional, usable and configurable; written completely in ANSI C.
OS-9 *Microware Systems Corporation.*	Prioritized	65535	Uses Active Queue, Event Queue, Semaphore Queue, Wait Queue and Sleep Queue.	Yes	Windows. Provides advanced networking and graphics capabilities for embedded devices. Uses the Hawk Integrated Development Environment (IDE); dynamic and modular architecture.
OSE *OSE Systems.*	Prioritized FIFO	32	Message-based architecture	Yes	Windows, Solaris, Linux. User-defined system clock resolution; ***first certified RTOS for safety***; fault-tolerant; suited mainly for wireless telecom and wireless applications.
RT-Linux *Finite State Machine Labs.*	Prioritized FIFO, uses an extensible scheduler	1024	Realtime tasks in RT-Linux can communicate with Linux processes via either shared memory or through a file-like interface.	Yes, uses lock free data structures and priority ceiling	Linux. Runs Linux and NetBSD as the idle thread (GPOS) of the real-time kernel; supports hard real-time applications; uses a "virtual machine scheduler" to run lower priority independent tasks in the standard linux kernel.

ThreadX *Express Logic, Inc.*	Prioritized FIFO, preemption -threshold	32	Event flags, mutexes, couting semaphores and message services	Yes, using preemption -threshold (disables preemption over ranges of priorities instead of disabling preemption of the entire system) and priority inheritance	Windows. Thread switch latency: 2µs Maximum interrupt latency: 2µs. ThreadX software timers are kept in an ordered list of expiration without performing a linear search on each timer activation; boosts the performance of a higher priority thread by placing its stack in a fast memory area.
QNX Neutrino *QNX Software Systems Ltd.*	Prioritized FIFO, prioritized round-robin	64	Message-based architecture	Yes, using message-based priority inheritance	Windows, Solaris, Linux, QNX4. Symmetrical multiprocessor systems. Every OS component runs in its own MMU-protected address space; supports Execute-in-place (XIP), which allows applications to run directly out of ROM or flash. Programs the clock device to raise an interrupt at each timer expiration time.

Table 2. Commercial RTOSs and their features

5.9 Research real-time kernels

In this section, we discuss two real-time kernels, namely, the Spring and Arx to provide an overview of the scope and type of ongoing research in the field of RTOS. Other prominent research kernels include Chimera [CMU], Harmony [National Research Council of Canada], Maruti [University of Maryland], etc.

Arx [18] is a research kernel being developed at Seoul National University. *Arx* uses user-level threads in scheduling and signal handling and multithreading.

Arx consists of *virtual threads* and a *scheduling event upcall* mechanism. Virtual threads provide a kernel-level execution environment for user threads. They are passive entities that would be temporarily bound to user-level threads when necessary. The *scheduling event upcall* mechanism enables the kernel to notify user processes of kernel events such as thread blocking and timer expiration.

User-level I/O allows programmers to write flexible and efficient device drivers for proprietary devices. To support user-level I/O, embedded RTOS should support the delivery of external interrupts from an I/O device to a process in a predictable and efficient manner. An efficient user-level I/O scheme is implemented in Arx, exploiting the multithreaded architecture of the kernel such as virtual threads and scheduling event upcalls.

The Spring real-time kernel [21] provides real-time support for distributed systems. It can schedule arriving tasks dynamically based upon execution time and resource constraints. Thus the need to *a priori* compute the worst case blocking time for tasks is avoided. The Spring kernel also deals with safety-critical tasks through a static table-driven scheduling. The kernel helps to retain a significant amount of application semantics to improve fault-tolerance and performance in overload situations. The kernel supports both application and system level predictability.

Spring supports abstraction for process groups [23], which provides a high level of granularity and a real-time group communication mechanism. A process group in Spring is a collection of processes that work towards a common goal. Spring supports a system description language (SDL), which allows programmers to predefine groups and impose timing and precedence constraints on them. Spring supports synchronous and asynchronous multicasting groups.

The Spring kernel achieves predictable low-level distributed communication using global replicated memory [21]. It provides abstractions for reservation, planning and end-to-end timing support.

EMERALDS (Extensible Microkernel for Embedded ReAL-time Distributed Systems) is a real-time microkernel designed for cost-conscious, small to medium size embedded systems [27]. EMERALDS maps the kernel into each user-level address space, hence with even full memory protection, a system call may not need context switches unless a user-level server is involved. Also EMERALDS supports adding user-level communication protocol stacks and device drivers without modifying the kernel. Moreover, EMERALDS provides the flexibility of having multiple protocol stacks on the same node.

EMERALDS provides fully-preemptive fixed-priority scheduling and partial support for dynamic scheduling. A user can choose the priority for a thread based on rate-monotonic, deadline-monotonic or any fixed-priority scheme suitable for a given application. At run time, a system call can be provided to change a thread's priority to respond to changing operating conditions. This feature can be used to emulate dynamic earliest-deadline first (EDF) scheduling at the user level. EMERALDS supports 32-bit non-unique thread priorities, hence by setting a thread's priority to its deadline, EDF scheduling can be realized [27].

EMERALDS supports both direct network access and use of protocol stacks for inter-process communication (IPC). Shared memory is used for intra-processor communication and the primary IPC mechanism is message passing using mailboxes. EMERALDS allows a 32-bit priority to be assigned to each message that can be used to sort messages in a mail box so that the receiver thread retrieves the highest-priority message first.

EMERALDS supports system calls that allow a device driver to map a memory-mapped device into its address space [27]. It also provides system calls that allow device drivers to handle interrupts. Hence device drivers will be able to tell the kernel which ISR subroutine to execute when an interrupt occurs. EMERALDS supports non-device drivers also to use such system calls.

6 Conclusion

In this report, we reviewed the basic requirements of an RTOS including the POSIX 1003.1b features. The POSIX 1003.1b standard does not address support for fixed-size buffers and heterogeneous multiprocessing. RTOS use is beneficial in most real-time embedded design projects. If an applic ation has real-time requirements, an RTOS provides a deterministic framework for code development and portability. To meet the needs of commercial multimedia applications, low code size and high peripheral integration is needed. Reliability in complex real-time systems could be achieved using multilevel specifications that check the correctness of systems at compile-time and run-time. The popular Windows CE and Jbed need further development in order to be used for hard real-time applications. RTOSs should be ABI compatible in order to avoid third-party vendor applications to be recompiled. Code reuse considerations are also important. Lastly, since the use of an RTOS is important in the embedded design world, a fast time to market and minimized development costs are as important as low hardware costs.

References

[1] S.R.Ball, *Embedded Microprocessor Systems*, Second edition, Butterworth-Heinemann, 2000.

[2] M.Bunnell, "Galaxy White Paper," http://www.lynx.com/lynx_directory/galaxy/galwhite.html

[3] G.D.Carlow, "Architecture of the Space Shuttle Primary Avionics Software System," *CACM*, v 27, n 9, 1984.

[4] CompactNET, http://www.compactnet.com.

[5] H.Gomaa, *Software Design Methods for Concurrent and Real-time Systems*, First edition, Addison-Wesley, 1993.

[6] S.Heath, *Embedded Systems Design*, First edition, Butterworth-Heinemann, 1997.

[7] http://www.rtos4voip.com/index.html

[8] IEEE. *Information technology--Portable Operating System Interface* (POSIX)-Part1: System Application: Program Interface (API) [C Language], ANSI/IEEE Std 1003.1, 1996 Edition.

[9] Jbed RTOS, http://www.esmertec.com

[10] E.Klein, "RTOS Design: How is Your Application Affected?," *Embedded Systems Conference*, 2001.

[11] P.A.Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook*, Second edition, IEEE Press, 1997.

[12] C.L.Liu and J.W.Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," *Journal of the ACM*, v 20, n 1, pp. 46-61, 1973.

[13] J.W.S.Liu, *Real-time Systems*, First edition, Prentice Hall, 2000.

[14] LynxOS, http://www.lynuxworks.com

[15] Microsoft Windows NT workstation resource kit.

[16] C.Muench and R.Kath, *The Windows CE Technology Tutorial: Windows Powered Solutions for the Developer*, First edition, Addison-Wesley, 2000.

[17] Rainfinity: http://www.rainfinity.com/index.html

[18] H.Y.Seo, and J.Park. “ARX/ULTRA: A new real-time kernel architecture for supporting user-level threads,” *Technical Report SNU-EE-TR1997-3*, School of Electrical Engineering, Seoul National University, 1997.

[19] A.Silberschatz, P.B.Galvin and G.Gagne, *Operating Systems Concepts*, Sixth edition, John Wiley, 2001.

[20] W.Stallings, *Operating Systems: Internals and Design Principles*, Third edition, Prentice-Hall, 1997.

[21] J.A.Stankovic and K.Ramamritham, “The Spring Kernel: A New Paradigm for Hard Real-time Operating Systems,” *ACM Operating Systems Review*, v 23, n 3, pp. 54-71, 1989.

[22] M.Stokely and N.Clayton, *FreeBSD Handbook*, Second edition, Wind River Systems, 2001.

[23] M.Teo, “A Preliminary Look at Spring and POSIX 4,” *Spring Internal Document*, 1995.

[24] The Open Group, http://www.opengroup.org/

[25] VxWorks, http://www.windriver.com

[26] C.Walls, “RTOS for Microcontroller Applications,” *Electronic Engineering*, v 68, n 831, pp. 57-61, 1996.

[27] K.M.Zuberi and K.G.Shin, “EMERALDS: A Microkernel for Embedded Real-Time Systems,” *Proceedings of RTAS*, pp.241-249, 1996.