A Survey of Contemporary Real-time Operating Systems

S. Baskiyar, Ph.D. and N. Meghanathan
Dept. of Computer Science and Software Engineering
Auburn University
Auburn, AL 36849, USA
baskiyar@eng.auburn.edu
http://www.eng.auburn.edu/~baskiyar

Keywords: scheduling, POSIX, kernels.

Received: June 26, 2004

A real-time operating system (RTOS) supports applications that must meet deadlines in addition to providing logically correct results. This paper reviews pre-requisites for an RTOS to be POSIX 1003.1b compliant and discusses 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 within 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.

Povzetek: Podan je pregled operacijskih sistemov v realnem času.

1 Introduction

A real-time system is one whose correctness involves both the logical correctness of outputs and their timeliness [7]. It must satisfy response-time constraints or 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 have 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.

An embedded system is a specialized real-time computer system that is part of a larger system. In the past, it was designed for specialized applications, but re-configurable and programmable embedded systems are becoming popular. Some examples of embedded systems are: the microprocessor system used to control the fuel/air mixture in the carburetor of automobiles, software embedded in airplanes, missiles, industrial machines, microwave ovens, dryers, vending machines, medical equipment, and cameras.

We observe that the choice of an operating system is important in designing a real-time system. Designing a real-time system involves choice of a proper language, task partitioning and merging, and assigning priorities to manage response times. Language synchronization primitives such as *Schedule*, *Signal* and *Wait* simplify translation of design to code and also offer portability. Depending upon scheduling objectives, parallelism and communication [3] may be balanced. Merging highly cohesive parallel tasks for sequential execution may reduce overheads of context switches and inter-task communications. The designer must determine critical tasks and assign them high *priorities*. However, care must be taken to avoid starvation, which occurs when higher priority tasks are always ready to run, resulting in insufficient processor time for lower priority tasks [9]. Non-prioritized interrupts should be avoided if there is a task that cannot be preempted without causing system failure. Ideally, the interrupt handler should save the context, create a task that will service the interrupt, and return control 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 and helps preserve the priority system of the RTOS. Furthermore, good response times may require small memory footprints in resource-impoverished systems. Clearly the choice of an RTOS in the design process is important for support of *priorities, interrupts, timers, inter-task communication, synchronization*, *multiprocessing* and *memory management*.

The organization of this paper is as follows. Section 2 outlines the basic requirements of an RTOS for POSIX 1003.1b compliance. Section 3 reviews memory management and scheduling algorithms used in RTOS. Section 4, classifies popular RTOS, compares contemporary commercial RTOSs and discusses the real-time features of two popular general-purpose operating systems. Section 5 concludes by discussing the results of this survey with a few suggestions for future research.

2 Features

The desirable features of an RTOS include the ability to schedule tasks and meet deadlines, ease of incorporating external hardware, error recovery, low task switching latency, small footprint and overheads. The kernel is the core of an OS that provides task scheduling, task dispatching and inter-task communication. In embedded systems, usually 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. In this section, basic requirements of an RTOS and POSIX compliance requirements have been addressed.

2.1 Basic requirements

The following are the basic requirements of an RTOS:

(i) *Multi-tasking and preemptable*: To support multiple tasks in real-time applications, an RTOS must be multi-tasking and preemptable. The scheduler should be able to preempt any task in the system and give the resource to the task that needs it most. An RTOS should also handle multiple levels of interrupts to handle multiple priority levels.

(ii) *Dynamic deadline identification:* In order to achieve preemption, an RTOS should be able to dynamically identify the task with the earliest deadline. To handle deadlines, deadline information may be converted to priority levels that are used for resource allocation. Although such an approach is error prone, nonetheless it is employed for lack of a better solution.

(iii) *Predictable synchronization:* For multiple threads to communicate among themselves in a timely fashion, predictable inter-task communication and synchronization mechanisms are required. Semantic integrity as well as timeliness constitutes predictability. Predictable synchronization requires compromises [14]. Ability to lock/unlock resources is one of the ways to achieve data integrity. To illustrate this point, Java methods can be declared with the keyword synchronized, e.g. synchronized *void AddOne().* Only one thread can call a synchronized method on a particular object, other threads trying to access that method on the same object wait; thus performance degradation is possible. Molesky, Shen, and Zlokapa [12] have proposed the *Deferred Bus Theorem* for binding the waiting time on a semaphore based on the number of requesters, time spent in the critical region, and the execution times of requesting and releasing a semaphore. However, they assume that the user can estimate the time each task may hold a lock, which may not be always feasible. Although deadlines may be assigned with semaphores, there is no guarantee that critical tasks have access over non-critical tasks. Another technique achieves speedup by non-blocking (lock-free) synchronization using FIFO queues [23]. The worst-case execution time of accessing a shared data object can thus be bounded.

(iv) *Sufficient Priority Levels:* When using prioritized task scheduling, the RTOS must have a sufficient number of priority levels, for effective implementation [9]. Priority inversion occurs when a higher priority task must wait on a lower priority task to release a resource and in turn the lower priority task is waiting upon a medium priority task. Two workarounds in dealing with priority inversion, namely priority inheritance and priority ceiling protocols (PCP), need sufficient priority levels.

In a priority inheritance mechanism, a task blocking a higher priority task inherits the higher priority for the duration of the blocked task. In PCP a priority is associated with each resource which is one more than the priority of its highest priority user. The scheduler makes the priority of the accessing task equal to that of the resource. After a task releases a resource, its priority is returned to its original value. However, when a task's priority is increased to access a resource it should not have been waiting on another resource.

(v) *Predefined latencies:* The timing of system calls must be defined using the following specifications:

- *Task switching latency* or the time to save the context of a currently executing task and switch to another.
- *Interrupt latency* or the time elapsed between the execution of the last instruction of the interrupted task and the first instruction of the *interrupt handler* [4].
- *Interrupt dispatch latency* or the time to switch from the last instruction in the *interrupt handler* to the next task scheduled to run.

2.2 POSIX compliance

IEEE Portable Operating System Interface for Computer Environments, POSIX 1003.1b provides the compliance criteria for RTOS services and is designed to allow application programmers write portable applications. The services required for compliance include the following:

- *Asynchronous I/O:* The ability to overlap application processing and application initiated I/O operations [5]. To support user-level I/O, an embedded RTOS should support the delivery of external interrupts from an I/O device to a process in a predictable and efficient manner.
- *Synchronous I/O:* The ability to assure return of the interface procedure when the I/O operation is completed [5].
- *Memory locking:* The ability to guarantee memory residence by storing sections of a process that were not recently referenced on secondary memory devices [20].
- *Semaphores:* The ability to synchronize resource access by multiple processes [17].
- *Shared memory:* The ability to map common physical space into independent process specific virtual space [5].
- *Execution scheduling:* The ability to schedule multiple tasks. Common scheduling methods include

round robin and priority based preemptive scheduling.

- *Timers:* Timers improve functionality and determinism of the system [9].
- *Inter-process Communication (IPC):* Common RTOS communication methods include mailboxes and queues.
- *Real-time files:* The ability to create and access files with deterministic performance.
- *Real-time threads*: Schedulable entities that have individual timeliness constraints [9].

3 Memory management and scheduling

This section addresses important issues of memory management and scheduling in an RTOS.

3.1 Memory management

An RTOS uses small memory size by including only the necessary functionality for an application while discarding the rest [22]. Below we discuss static and dynamic memory management in RTOSs.

Static memory management provides tasks with temporary data space. The system's free memory is divided into a pool of 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].

Dynamic memory management employs memory swapping, overlays, multiprogramming with a fixed number of tasks (MFT), multiprogramming with a variable number of tasks (MVT) and demand paging. Overlays allow programs larger than the available memory to be executed by partitioning the code and swapping them from disk to memory. In MFT, a fixed number of equalized code parts are in memory at the same time. As needed, the 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 reside in non-contiguous memory, unlike those in MFT and MVT [7]. In many embedded systems, the kernel and application programs execute in the same space i.e., there is no memory protection.

3.2 Scheduling

In this section, we very briefly outline scheduling algorithms employed in real-time operating systems. We note that predictability requires bounded operating system primitives. A feasibility analysis of the schedule may be possible in some instances. The scheduling literature is vast and the reader is referred to [15] for a detailed discussion.

Task scheduling can be either performed preemptively or non-preemptively and either statically or dynamically. For small applications, task execution times can be estimated prior to execution and the preliminary task schedules statically determined. Two common constraints in scheduling are the resource requirements and the precedence of execution of the tasks. Typical parameters associated with tasks are:

- Average execution time
- Worst case execution time
- Dispatch costs
- Arrival time
- Period (for periodic tasks).

The objective of scheduling is to minimize or maximize certain objectives. Typical objectives minimized are: schedule-length, average tardiness or laxity. Alternatively, maximizing average earliness and number of arrivals that meet deadlines can be objectives. In [15] scheduling approaches have been classified into: static table driven approach, static priority driven preemptive approach, dynamic planning based approach, dynamic best effort approach, scheduling with fault tolerance and resource reclaiming. We briefly discuss the approaches below.

(i) *Static table driven:* The feasibility and schedule are determined statically. A common example is the cyclic executive, which is also used in many large-scale dynamic real-time systems [2]. It assigns tasks to periodic time slots. Within each period, tasks are dispatched according to a table that lists the order to execute tasks. For periodic tasks, there exists a feasible schedule if and only if there is a feasible schedule within the least common multiple of the periods. A disadvantage of this approach is that a-priori knowledge of the maximum requirements of tasks in each cycle is necessary.

(ii) *Static priority driven preemptive:* The feasibility analysis is conducted statically. Tasks are dispatched dynamically based upon priorities. The most commonly used static priority driven preemptive scheduling algorithm for periodic tasks is the Rate Monotonic (RM) scheduling algorithm [8]. A periodic system must respond with an output before the next input. Therefore, the system's *response time* should be shorter than the minimum time between successive inputs. RM assigns priorities proportional to the frequency of tasks. It can schedule any set of tasks to meet deadlines if the total resource utilization less than *ln 2*. If it cannot find a schedule, no other fixed-priority scheduling scheme will. But it provides no support for dynamically changing task periods/priorities and priority inversion. Also, priority-inversion may occur when to enforce rate-monotonicity, a non-critical task of higher frequency of execution is assigned a higher priority than a critical task of lower frequency of execution.

(iii) *Dynamic planning based:* The feasibility analysis is conducted dynamically—an arriving task is accepted for execution only when feasible. The feasibility analysis is also a source for schedules. The execution of a task is guaranteed by knowing its worst-case execution time and faults in the system. Tasks are dispatched to sites by brokering resources in a centralized fashion or via bids. A technique using both centralized and bidding-approach performs marginally better than any one of them but is more complex [15].

(iv) *Dynamic best effort approach***:** Here no feasibility check is performed. A best effort is made to meet deadlines and tasks may be aborted. However, the approaches of Earliest Deadline First (EDF) and Minimum Laxity First (MLF) are often optimal when there are no overloads. Research into overloaded conditions is still in its infancy. Earliest deadline first (EDF) scheduling can schedule both static and dynamic real-time systems. Feasibility analysis for EDF can be performed in $O(n^2)$ time, where n is the number of tasks [7]. Unlike EDF, MLF accounts for task execution times.

(v) *Scheduling with fault tolerance:* A primary schedule will run by the deadline if there is no failure and a secondary schedule will run by the deadline on failure. Such a technique allows graceful degradation but incurs cost of running another schedule. In hard real-time systems, worst-case blocking must be minimized for fault tolerance.

(vi) *Scheduling with resource reclaiming*: The actual task execution time may be shorter than the one determined a-priori because of conditionals or worst-case execution assumptions. The task dispatcher may try to reclaim such slacks, to the benefit of non real-time tasks or improved timeliness guarantees.

4 Commercial RTOSs

In this section, we select a prominent commercial RTOS for each class of real-time application and discuss its features. For small memory devices *Windows CE* has been discussed, for hard real-time systems, *LynxOS*, for embedded applications *VxWorks*, *Jbed* for the Java platform and *pSOS* for an object-oriented operating system. But first, we list the common capabilities of these operating systems.

- *Efficiency:* Most RTOSs are micro-kernels with low overhead. In some, almost no context switch overhead is incurred in sending a message to the system service provider.
- *Non-preemptable system calls*: Certain portions of system calls are non-preemptable to support mutual exclusion. These parts are optimized, made as deterministic as possible.
- *Prioritized scheduling*: For POSIX compliance, all RTOSs offer at least 32 priority levels. The number of priority levels range from 32-512.
- *Priority inversion control*: A means of handling priority inversion.
- *Memory management support:* Support for virtual memory management exists but not necessarily paging. The users are offered choices among multiple levels of memory protection.

4.1 Windows CE

Windows CE [13] is a modular, portable real-time embedded OS for small memory, mobile 32-bit devices. *Windows CE* slices CPU time among threads and provides 256 priority levels. To optimize performance, all threads are enabled to run in kernel mode. *Windows CE* kernel has the following features:

- 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-preemptable portions of the kernel, are broken into small sections reducing the duration of non-preemptable code.
- All kernel objects (such as processes, threads, critical sections, mutexes, events and semaphores) are dynamically allocated in virtual memory.
- For portability, an equipment adaptation layer isolates device dependent routines. The equipment manufacturer can specify trusted modules and processes to prevent unauthorized applications from accessing system application programming interfaces.

4.2 LynxOS

LynxOS [10] is a POSIX compatible, multithreaded OS designed for complex real-time applications that require fast, deterministic response. It is scalable from large switching systems down to small-embedded products. The micro-kernel can schedule, dispatch interrupts, and synchronize tasks. Other services offered by the kernel lightweight service modules, are TCP/IP streams, I/O and file systems, sockets, etc. In response to an interrupt, the kernel dispatches a kernel thread, which can be prioritized and scheduled similar to other threads. The priority of the interrupt handling kernel thread is the priority of the user thread that handles the interrupting device. This mechanism ensures predictable response even in the presence of heavy I/O. The OS depends upon hardware memory management units for memory protection, but does offer optional demand paging. It uses scheduling policies such as prioritized FIFO, dynamic deadline monotonic scheduling, time-slicing etc. It has 512-priority levels and supports remote operation.

4.3 VxWorks

VxWorks [21] is a widely adopted RTOS in the embedded industry with a visual development environment. It is scalable with over 1800 APIs and is available on popular CPU platforms. It offers network support, file system and I/O management. The micro-kernel supports 256 priority levels, multitasking, deterministic context switching and preemptive and round robin scheduling, semaphores and mutual exclusion with inheritance. TCP, UDP, sockets and standard Berkeley network services can all be scaled in or out of the networking stack as necessary. It can be set up so that each task has a private virtual memory upon request. For portability a Board Support Package interfaces with the hardware-dependent layer.

4.4 Jbed

Jbed [6] is a real-time operating system for embedded systems. It supports applications and device drivers written in Java. Instead of interpreting byte-codes, *Jbed* translates byte-codes to machine code prior to class loading. Its modular architecture allows dynamic code loading and scaling from small to high performance devices. It supports real-time memory allocation, exception handling and automatic object destruction. Hard real-time applications are supported via specific class libraries. It supports ten thread priority levels and EDF scheduling.

Jbed light is a smaller version for fast and precompiled applications. It contains the basic components such as the core virtual machine, a small set of standard Java libraries, and the *Jbed* libraries required to directly access peripherals. The Java virtual machine calls are implemented in the kernel. This avoids the need for a slow Java Native Interface, otherwise needed to make system calls. Current versions support ARM7, 68k and the PowerPC architectures.

4.5 pSOS

The objects, in object-oriented pSOS, include tasks, memory regions, message queues, and semaphores. It schedules tasks in preemptive priority-driven or EDF and handles priority inversion by both priority inheritance and priority-ceiling protocol. The application developer has complete control over interrupt handling. User tasks may also run in supervisory mode. Device drivers may be dynamically loaded. 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. As other objects, a memory region may be local or global.

4.6 General purpose operating systems

In this section, we outline real-time features of two popular general-purpose operating systems: Windows NT and Unix, Table 1 shows a comparison[1].

Real-time feature	Windows NT	Native Unix
Preemptive, priority-based multitasking	Yes	Yes
Deferred interrupt threads	Yes	No
Non-degrading priorities	Yes	No
Memory locks	Yes	Yes

Table 1. Real-time features of Windows NT and Unix

- *Preemption:* Although Windows NT kernel is non-preemptable there are points within the kernel where preemption is allowed. Real-time Unix also allows preemption points within system calls.
- *Deferred Procedure Calls (DPCs):* DPCs are queued calls to kernel mode functions to be executed later. They are used by drivers to schedule I/O operations that do not necessarily have to take place in an interrupt service routine at a high interrupt level and can be safely postponed until the level has been lowered. Such a mechanism allows servicing of interrupts within interrupts, if the processor disables future interrupts when an interrupt is being serviced.
- *Non-degrading priorities:* To ensure fairness, the system continuously manipulates thread priorities in Unix and Windows NT. However, Windows NT provides a band of interrupt priorities that cannot be altered. Accordingly, there exist two types 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, but not beyond the set of priorities of its class.

Although Windows NT provides fast response times, it is not as deterministic as a hard RTOS [11] because of deferred procedure calls. Since user threads have lower priority than DPCs or ISRs, mouse and keyboard handlers may preempt urgent processes. Also, DPCs are not preempted by other DPCs/threads. Furthermore, the developer has no control over third party drivers.

Since Windows NT kernel does not support priority inheritance, deadlocks may occur. It does not support prioritized queuing for inter-thread communication. In other words, if multiple threads are blocked waiting on a resource, they will be granted access in FIFO rather than priority order unlike an RTOS.

[1] Although Windows NT was not intended to be an RTOS it has been used as one in some instances.

4.7 Other commercial RTOS

Table 2 lists other common commercial RTOSs and their main features with respect to the basic requirements of an RTOS discussed in Section 2. All of the products below use a prioritized FIFO scheme for scheduling.

4.8 Research kernels

We now discuss three real-time kernels, Extensible Micro-kernel for Embedded ReAL-time Distributed Systems (EMERALDS), 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 (from Carnegie Mellon University), Harmony (from National Research Council of Canada) and Maruti (from University of Maryland).

EMERALDS is designed for small to medium sized embedded systems [24]. It maps the kernel into every user space. Therefore a system call does not need any context switch. User level communication protocol stacks and device drivers may be added without modifying the kernel. It uses preemptive fixed priority and dynamic scheduling. A user can choose the priority of a thread based on rate-monotonic, deadline-monotonic or other fixed priority scheme. It supports 32-bit non-unique thread priorities—by setting a thread's priority to its deadline, EDF scheduling can be accomplished. The priority can be dynamically modified via a system call to support dynamic EDF scheduling. The IPC mechanisms are shared memory and message passing via mailboxes. A 32-bit priority is assigned to each message that can be used to sort them to retrieve the highest-priority message first.

Arx [16] employs user level threads for scheduling, communication and multithreading. It 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 are 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.

The *Spring* kernel [18] provides real-time support for distributed systems. It can schedule 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. It schedules safety-critical tasks using a static table. The kernel helps retain enough application semantics to improve fault-tolerance and performance on overloads. It supports both application and system level predictability. Spring supports abstraction for process groups [19], which provides a high level of granularity and a real-time group communication mechanism. Processes within a "process group" in *Spring* work towards a common goal. *Spring* supports a system description language, which allows programmers to predefine groups and impose timing and precedence constraints on them. It supports both synchronous and asynchronous multicasting groups. It achieves predictable low-level distributed communication via globally replicated memory. It provides abstractions for reservation, planning and end-to-end timing support.

A comparison of the features of *Arx*, EMERALDS and *Spring* show that all of them incorporate most of the basic recommendations of POSIX 1003.1 b. However, the feature of real-time files has not been incorporated in any of the above research kernels.

5 Conclusion

This paper 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. Designing an embedded system using an RTOS may help lower cost and the time to market. If an application 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.

6 References

[1] S.R. Ball, Embedded Microprocessor Systems: Real World Design, Third edition, Newnes, 2002.

[2] G. D. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System," CACM, v 27, no. 9, 1984.

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

[4] S.Heath, Embedded Systems Design, Second edition, Newnes, 2002.

[5] IEEE Information Technology—Portable Operating System Interface (POSIX)—Part1:

[6] System Application Program Interface, ANSI/IEEE Std 1003.1, 1996 Edition. Jbed RTOS, http://www.esmertec.com, Accessed Nov 15, 2004.

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

[8] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," Journal of the ACM, v. 20, no. 1, pp. 46-61, 1973.

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

[10] LynxOS, http://www.lynuxworks.com, Accessed Nov 15, 2004

[11] Microsoft Windows NT, http://www.microsoft.com, Accessed Nov 10, 2004

RTOS, Vendor	*Thread priority levels*	*Synchronization mechanisms*	*Priority inversion prevention provided*	*Development hosts, kernel characteristics*
AMX, *KADAK Products Ltd.*	N/A	Mailboxes; wait-wake requests	Yes	Windows, predictable memory block availability
C Executive, *JMI Software Systems, Inc.*	32	Messages, dynamic data queues	Yes	Windows, Solaris
CORTEX, *Australian Real-time Embedded Systems.*	62	Recursive locks, mutexes	Yes, uses priority ceiling	Windows/Unix, CPU-independent software interrupt manager; statically and dynamically segmented memory models
Delta OS, *CoreTek Systems, Inc.*	256	Semaphores, timers, message queues	Yes	Windows, Linux.
Ecos *RedHat, Inc.*	1-32	Semaphores, timers and counters	Yes, uses priority ceiling	Windows, Linux For soft real-time embedded applications in small devices
Emboss *SEGGER Microcontroller Systems.*	255	Mailbox, binary and counting semaphore	No	Windows, Linux, profiling to collect timing information for every task; task activation time independent of number of tasks.
ERTOS *JK Microsystems, Inc.*	256	Inter-thread messaging, queues, semaphores	No	Windows, DOS, OS/2. High-speed interrupt driven serial port routines
INTEGRITY *GreenHills Software, Inc.*	255	Semaphores, breakpoints can be placed any where in the system including ISRs.	Yes, mutex, semaphore	Used in critical embedded applications; object-oriented; supports distributed processing; task execution profiling.
IRIX 1.1.1.1.1 SGI	255	Message queues	Yes	SGI, Double-precision matrix support; Multi-pipe scalability
Nuclear Plus *Accelerated Technology, Inc.*	N/A	Mailboxes, pipes and queues	Yes	Windows.
OS-9 *Microware Systems Corporation.*	65535	Semaphore and queues	Yes	Windows.
OSE *OSE Systems.*	32	Message passing	Yes	Windows, Solaris, Linux. User-defined system clock resolution; fault-tolerant; suited for wireless applications.
RT-Linux *Finite State Machine Labs.*	1024	Shared memory or via files	Yes, lock free data structures and priority ceiling	Linux; supports hard real-time applications
ThreadX *Express Logic, Inc.*	32	Mutexes, counting semaphores and messaging	Yes, by disabling preemption over ranges of priorities and by priority inheritance	Windows.
QNX Neutrino *QNX Software Systems Ltd.*	64	Message passing	Yes, using priority inheritance	Windows, Solaris, Linux, Symmetrical multiprocessor systems. Every OS component runs in its own MMU-protected address space

Table 2. Features of commercial *RTOSs*

[12] L. Molesky, C. Shen, G. Zlokapa, “Predictable Synchronization Mechanisms For Multiprocessor Real-Time Systems,” COINS, University of Massachusetts, *Technical Report 90-30*, 1990.

[13] C. Muench, The Windows CE Technology Tutorial: Windows Powered Solutions for the Developer, First edition, Addison Wesley, 2000.

[14] R. Ortega, “Timing Predictability in Real-Time Systems,” *Technical Report*, Dept. of Computer Science, University of Washington, 1994.

[15] K. Ramamritham and J. A. Stancovic, “Scheduling Algorithms and Operating Systems Support for Real-time Systems,” Proceedings of the IEEE, pp. 55-67, Jan 1994.

[16] 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.

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

[18] J.A. Stankovic and K. Ramamritham, “The Spring Kernel: A New Paradigm for Hard Real-time Operating Systems,” ACM Operating Systems Review, vol. 23, no. 3, pp. 54-71, 1989.

[19] M. Teo, “A Preliminary Look at Spring and POSIX 4,” Spring Internal Document, 1995, available at http://www-cs.umass.edu/spring/internal/spring-kernel-docs.html

[20] The Open Group, http://www.opengroup.org/, Accessed Nov 10, 2004

[21] VxWorks, http://www.windriver.com, Accessed Nov 10, 2004

[22] C. Walls, “RTOS for Microcontroller Applications,” Electronic Engineering, vol. 68, no. 831, pp. 57-61, 1996.

[23] Y. Zhang, Non-blocking Synchronization: Algorithms and Performance Evaluation, Ph.D. Thesis, Chalmers University of Technology, Sweden, 2003.

[24] K. M. Zuberi and K. G. Shin, “EMERALDS: A Small-Memory Real-Time Micro-kernel,” IEEE Trans. on Software Engineering, vol. 27, no. 10, pp. 909-928, October 2001.

7 Acknowledgements

This paper has been developed while the first author was teaching classes in Real-time and Embedded Computing over the last several years. As such several students within these classes helped generously in this work to whom the first author is very grateful. The first author also thanks Dr. James H. Cross, Dept. of Computer Science and Software Engineering, Auburn University, Auburn, AL for reviewing this paper. This work was supported in part by an Auburn University internal grant and National Science Foundation Grant numbers 0408136 and 0411540.

8 Biography

S. Baskiyar is Assistant Professor in the Department of Computer Science and Software Engineering at Auburn University, Auburn, AL. His research interests are in the areas of Task Scheduling on Clusters, Computer Architecture and Embedded Systems. He has published extensively in the area of Task Scheduling on Clusters. He received the PhD and MSEE degrees from the University of Minnesota, Minneapolis and the BE (Electronics and Communications) degree from the Indian Institute of Science, Bangalore. He received the BS degree in Physics with honors and distinction in Mathematics. He received the competitive State-merit and the Indian Institute of Science scholarships. He has taught courses in Real-time and Embedded Computing, Computer Architecture, Operating Systems, Microprocessor Programming and Interfacing and VLSI Design. His experience includes working as an Assistant Professor at the Western Michigan University, as a Senior Software Engineer in the UNISYS Corporation and as an Assistant Computer Engineer in Tata Engineering and Locomotive Company Ltd., India.

N. Meghanathan received the Master’s degree in Computer Science from Auburn University, Auburn, AL in 2002. He received the Bachelor’s degree in Chemical Engineering from Anna University, Chennai, India. He was a research assistant in the Department of Computer Science and Software Engineering at Auburn University.