REAL-TIME POSIX: AN OVERVIEW

Michael González Harbour[1]
Departamento de Electrónica
Universidad de Cantabria
Avda. los Castros s/n
39005 - Santander
SPAIN
E-mail: mgh@ccucvx.unican.es

Abstract

The POSIX standard defines a portable interface for UNIX-based operating systems. The goal of this increasingly important standard is source-level portability of applications. In this paper we discuss the real-time extensions to POSIX and how these extensions address the needs of applications with real-time requirements.

I. INTRODUCTION

POSIX is the acronym for Portable Operating System Interface. It is a proposed operating system interface standard based on the popular UNIX[2] operating system; its main goal is to support application portability at the source-code level. It is being standardized by the Computer Society of IEEE as the IEEE standard P1003, and also by ISO/IEC, as the international standard ISO/IEC-9945.

POSIX is an evolving group of standards, each of which covers different aspects of the operating systems. Some of these standards have already been approved, while others are currently being developed. They can be grouped in three categories:

1) *Base Standards*: They define system interfaces related to different aspects of the operating system. The standard specifies the syntax and semantics of system interfaces so that application programs can directly invoke the operating system services. The standard does not specify how these services are to be implemented, just their semantics; system implementors can choose their implementation as long as they follow the specification of the interface. Initially, the base standards were developed for the C language, but now they are being specified as language-independent interfaces. Table I and Table II list the base standards that are currently being standardized under POSIX.

Table I. List of POSIX Base Standards

POSIX.1	System Interface (basic reference standard)[a,b]
POSIX.2	Shell and Utilities[a]
POSIX.3	Methods for Testing Conformance to POSIX[a]
POSIX.4	Real-time Extensions
POSIX.4a	Threads Extensions
POSIX.4b	Additional Real-time Extensions
POSIX.6	Security Extensions
POSIX.7	System Administration
POSIX.8	Transparent File Access
POSIX.12	Protocol Independent Network Interfaces
POSIX.15	Batch Queuing Extensions
POSIX.17	Directory Services

[a] Approved IEEE standards
[b] Approved ISO/IEC standard

Table II. Additional POSIX Base Standards

P1224	Message Handling Services (X.400)
P1224.1	X.400 Application Portability Interface
P1238	Common OSI Application Portability Interface
P1238.1	FTAM OSI Application Portability Interface
P1201.1	Windowing Application Portability Interface
P1201.2	Recommended Practice on Driveability

[1] This work was supported in part by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government, under grants ROB91-0288 and ROB91-1553-E.

[2] UNIX is a trademark of AT&T

2) *Language bindings*: These standards provide the actual interfaces for different programming languages. The languages that are currently being used are C, Ada, Fortran 77, and Fortran 90. Table III lists the POSIX language bindings that are currently under development.

Table III. List of POSIX Language Bindings

POSIX.5	Ada Bindings[a]
POSIX.9	Fortran 77 Bindings[a]
POSIX.16	C Language Bindings
POSIX.19	Fortran 90 Bindings
POSIX.20	Ada Bindings to Real-time Extensions
[a] Approved IEEE standards	

3) *Open Systems Environment*. These standards include a guide to the POSIX environment, and application profiles. An application profile is a list of the POSIX standards that are required for a certain application environment, along with the options and parameters of these standards whose support is required for that application environment. Application profiles are a very important means of achieving a small number of well-defined types of operating system implementations appropriate for particular application environments. Table IV shows the list of standards that are being developed in this group.

Table IV. List of POSIX Application Environment Standards

POSIX.0	Guide to POSIX Open System Environment
POSIX.10	Supercomputing Application Environment Profile
POSIX.11	Transaction Processing Application Environment Profile
POSIX.13	Real-time Application Environment Profiles
POSIX.14	Multiprocessing Application Environment Profile
POSIX.18	POSIX Platform Application Environment Profile

The POSIX standard is necessary because, although UNIX is a de-facto standard, there are enough differences among the different implementations to make applications not be completely portable. But, while a UNIX application may need some changes to be ported to a different platform, portability of real-time applications is far more difficult, since there exist a large variety of real-time operating systems. UNIX is not a real-time operating system, and there is no de-facto standard for these applications.

Because of the need to achieve application portability for real-time systems, a real-time working group was established in POSIX. This group is developing standards to add POSIX (or UNIX) the OS services that are needed by real-time applications. The charter of the POSIX Real-time Working Group is to "develop standards which are the minimum syntactic and semantic changes or additions to the POSIX standards to support portability of applications with real-time requirements."

Many real-time applications, such as small embedded systems, have special physical constraints that demand for operating systems with a reduced set of functionality. For example, many systems exist which cannot have a disk drive, do not have a hardware memory management unit, and have a small amount of memory. For these systems it is necessary that the standard allows implementations to only support a particular subset of the POSIX functions. The subsets necessary for real-time applications are also being addressed by the Real-time Working Group, which has specified four real-time application environment profiles: for small embedded systems, real-time controllers, large embedded systems, and large systems with real-time requirements.

According to these requirements, the Real-time Working Group is currently developing four standards, that we will review in this paper:

POSIX.4:
Real-time extensions. Defines interfaces to support the portability of applications with real-time requirements.

POSIX.4a
Threads extension. Defines the interfaces to support multiple threads of control inside each POSIX process.

POSIX.4b
Additional real-time extensions. Defines interfaces to support additional real-time services.

POSIX.13
Real-time application environment profiles. Each profile lists the services that are necessary for a particular application environment.

The following sections discuss the most relevant aspects of each of these standards. The discussion is based on the status of these standards at the time this paper is being written. This status corresponds to Draft 13 of POSIX.4 [10], Draft 6 of POSIX.4a [11], Draft 6 of POSIX.4b [12], and Draft 5 of POSIX.13 [13]. Since all these standards are still being developed, changes made to them could affect the discussions in this paper. However, we believe that the spirit of most of what is discussed here will apply to the final standards.

II. REAL-TIME EXTENSIONS

This section discusses some of the most important features of POSIX.4 [10], which is the part of POSIX that defines system interfaces to support applications with real-time requirements. POSIX.4 is very near to its approval as a standard.

A. Real-time Process Scheduling

The base POSIX.1 standard [9] defines a model of concurrent activities called processes, but does not specify any scheduling policy nor any concept of priority. For real-time applications to be portable it is necessary to specify some scheduling policy suitable for real-time. POSIX.4 specifies three scheduling policies. Each process has a scheduling attribute that can be set to any of the three policies:

- SCHED_FIFO: This is a fixed-priority preemptive scheduling policy, in which processes with the same priority are treated in first-in-first-out (FIFO) order. At least 32 priority levels must be available for this policy.
- SCHED_RR: This policy is similar to SCHED_FIFO, but uses a time-sliced (round robin) method to schedule processes with the same priorities. It also has 32 priority levels.
- SCHED_OTHER: It is an implementation-defined scheduling policy.

Fixed-priority scheduling is a popular scheduling policy for real-time systems. It is very simple, and high utilization levels can be achieved by using rate-monotonic [4] or deadline-monotonic [3] priority assignments. With these scheduling policies and with the functions used to set and get the policies and priorities of each process, real-time applications can be scheduled in POSIX operating systems. A good introduction to the design and analysis of this kind of real-time systems using recent results for fixed-priority scheduling is provided in [7].

B. Virtual Memory Locking

Although virtual memory is not required by POSIX.1, it is common UNIX practice to provide this mechanism that has great benefits for non real-time software, but introduces large amounts of unpredictability in the timing response. In order to bound memory access times, functions are defined in POSIX.4 to lock into physical memory either the whole address space of a process, or selected ranges of this address space. These functions should be used for time-critical activities, and also for those activities with which they may synchronize. In this way, their response times can be made predictable.

C. Process Synchronization

POSIX.4 defines functions to manage process synchronization with counting semaphores. These semaphores are identified by a name that belongs to an implementation-defined name space. This name space may or may not coincide with the file name space. The counting semaphore is a common synchronization mechanism that allows mutually exclusive access to shared resources, signaling and waiting among processes, and other synchronization requirements. One of the most common uses of semaphores is to share data among processes, and this can be accomplished in POSIX.4 by using shared memory objects (see Section II.D) together with semaphores.

Unfortunately, the counting semaphores specified in POSIX.4 do not prevent unbounded priority inversion [6]. Priority inversion occurs when a high priority process has to wait for a lower priority process to complete some action. Using appropriate protocols, priority inversion can be bounded by the duration of critical sections, that is, sections of code during which the process reserves a particular resource for exclusive use. However, with conventional semaphores unbounded priority inversion may occur; this means that the delay experienced by high priority tasks is not bounded by the duration of critical sections, but depends on the total execution time of lower priority tasks. This situation can occur when a high priority task is waiting for a low priority task to release a semaphore that controls access to a shared resource, and the low priority task is preempted by an intermediate priority task. Figure 1 shows an example of this situation. These long delays are usually unacceptable for tasks with hard real-time requirements. If appropriate protocols are used [6] the amount of priority inversion can be bounded by the duration of critical sections, which is usually very small. In Section III.C we will discuss a different synchronization mechanism —the mutex— that prevents unbounded priority inversion and can optionally be used across processes.

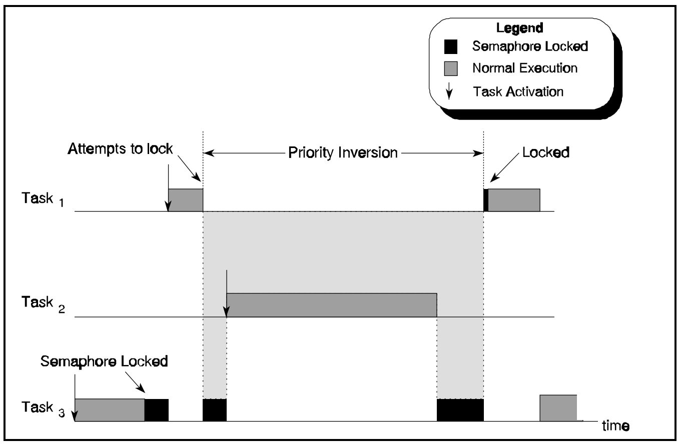

Figure 1. Example of Unbounded Priority Inversion

D. Shared Memory

POSIX.1 processes have independent address spaces, but many real-time (and non real-time) applications require sharing large amounts of data with very little overhead. This can be accomplished if processes are allowed to share portions of physical memory. POSIX.4 defines shared-memory objects, which are regions of memory that can be mapped into the address space of a process. When two or more processes map the same memory object they share the associated region of memory. As with semaphores, shared memory objects are identified by a name belonging to an implementation-defined name space. If data objects allocated in shared memory require mutually exclusive access, semaphores can be used to control these accesses. Files can also be mapped into the address space of a process, and can be shared among processes.

E. Real-Time Signals

The signal mechanism defined in POSIX.1 allows to notify events occurring in the system, but is not completely satisfactory for real-time applications. The signals are not queued, and thus some events may be lost. Signals are not prioritized, and this implies longer response times for urgent events. Also, events of the same kind produce signals with the same number, which are indistinguishable. Since many real-time systems are heavily based on the rapid exchange of events in the system, POSIX.4 has extended the signals interface to achieve the following features:

- Real-time signals are queued, so events are not lost
- Pending real-time signals are dequeued in priority order, using the signal number as the priority. This allows designing applications with faster response times to urgent events.
- Real-time signals contain an additional data field that may be used by the application to exchange data between the signal generator and the signal handler. For example, this data field may be used to identify the source of the signal.
- The range of signals available to the application is expanded.

F. Interprocess Communications

A simple message queue mechanism is specified for interprocess communications. Message queues are identified by a name belonging to an implementation-defined name space. Messages have an associated priority field and are extracted in priority order. This helps in reducing unbounded priority inversion in the system. Transmission and reception of messages can be blocking or non-blocking; transmission and reception are not synchronized, that is, the sender does not wait until the receiver has actually retrieved the message from the queue. The maximum sizes of messages and queues are user definable, and the resources needed by the queue may be preallocated at creation time; this allows to increase the predictability of message queue operations.

G. Clocks and Timers

A real-time clock that measures wall-time is defined. This clock must at least provide a resolution of 20 ms. Time is represented with nanosecond resolution, so implementations can take advantage of high-precision hardware clocks. Timers can be created to count time intervals, using the real-time clock or other implementation-defined clocks as the timing basis. When the specified time interval has elapsed, these timers generate a signal directed to the process that created the timer. Several options such as periodic signaling, single shot, etc. exist, allowing for example an easy implementation of periodic event generation. A relative sleep function is defined (*nanosleep*) to suspend the calling process for a specified time interval.

H. Asynchronous Input/Output

POSIX.4 defines functions that provide the ability to overlap application processing and I/O operations initiated by the application. Asynchronous I/O operations are similar to the normal I/O operations, except that, after an asynchronous operation has been initiated by a process, that process proceeds executing in parallel with the I/O operation. When the operation completes, a signal can be delivered to the application.

I. Other Functions

POSIX.4 defines other functions such as synchronized input/output, real-time files, etc. For a description of these functions the reader is referred to the documentation of the standard [10].

III. THREADS EXTENSION

The POSIX.1 process model is not adequate for some of the systems that require high efficiency, because processes have high context switch times, the time needed to create or destroy them is very high, special hardware is needed (memory management units) to provide each process with an independent address space, and the model is not adequate for shared memory multiprocessors. In most of the real-time kernels that are commercially available for small embedded systems the concurrency model is based on tasks that share the same address space and have an associated state that is small, compared to POSIX processes. The Real-time Working Group considered all these issues, and decided to develop the threads extension.

POSIX.4a defines interfaces to support multiple concurrent activities, called threads, inside each POSIX process. The threads defined in POSIX.4a have an associated state that is smaller than the state of processes. All threads inside the same process share the same address space. They can be implemented with context switch times and creation/destruction times lower than those of processes. POSIX.4a has been specifically developed to also address the needs of shared memory multiprocessors. With these characteristics, the thread model is much closer to the concurrency model of commercial real-time kernels than the process model. But threads are not only intended for real-time applications; they can also be applied for non real-time systems requiring efficient context switch times and creation/destruction times, such as windowing applications, multiprocessor software, etc.

Threads can use all the process functions defined in POSIX.4 and POSIX.1, in addition to the functions defined specifically for threads in POSIX.4a. The most relevant of these functions are discussed next:

A. Thread Management

These functions allow to manage thread creation and termination, and related operations. Functions are defined to create a thread, wait for thread termination, terminate a thread normally, detach a thread —that is, indicate to the implementation that the storage associated with a thread may be reclaimed when the thread terminates—, or create a particular thread only if it has not been created already. Other functions allow to handle thread identifiers. Also, functions are defined to manage thread creation attributes such as stack size, whether the thread storage is detachable from creation time, etc.

B. Thread Scheduling

The scheduling policies defined for threads are the same as those defined for processes in POSIX.4 (priority preemptive, with either FIFO or round robin treatment of equal priority threads). Since two schedulers may coexist in the system —the process and thread schedulers—, the concept of *contention scope* is defined. The contention scope of a thread defines the set of threads with which it must compete for the use of the CPU. Three main kinds of implementations with different contention scopes can arise:

- *Global Scheduling*: All threads have global contention scope, are therefore every thread is scheduled against all other threads in the system, no matter which process they belong to. The scheduler works only at the thread level, and the process scheduling parameters are ignored.

- *Local Scheduling*. Threads only compete with the other threads belonging to the same process. Scheduling is done at two levels. First, processes are scheduled against each other. Then, the threads of the selected process compete among each other for the CPU.

- *Mixed Scheduling*. Some threads have global contention scope, and other threads have local contention scope. Scheduling is done at two levels: in the first level, processes and global threads are scheduled; at the second level, local threads within the selected process are scheduled.

Both the global and mixed schedulers will provide the best results for most real-time applications, since they allow to schedule all the different concurrent objects that have strict timing requirements at the same level. Systems with mixed scheduling can also handle local scheduling for selected threads. Local scheduling is usually faster and more efficient than global scheduling. However, this feature should only be used for groups of threads whose priority is globally smaller than or larger than the priorities of other groups of threads in the system (i.e., when no other thread in the system is required to have a priority level in between the priority levels of any of the threads in the group). The reason for this is that the process priority, rather than the thread priorities, will be used to schedule the group of threads with local contention scope. The same discussion applies to systems with local scheduling.

C. Thread Synchronization

Two synchronization primitives are defined for threads: mutexes, and condition variables. Mutexes are used to synchronize threads for mutually exclusive access to shared resources, while condition variables are used to signal and wait for events among threads. Waiting for a condition variable to be signaled can be specified with a timeout. These primitives can optionally be used by threads belonging to different processes.

Mutexes are defined with three optional synchronization protocols:

- NO_PRIO_INHERIT: The priority of the thread does not depend on its ownership of mutexes (a mutex is *owned* by the thread that locked it).

- PRIO_INHERIT: The thread owning a mutex inherits the priorities of the threads waiting to acquire that mutex. This is the priority inheritance protocol [6].

- PRIO_PROTECT: When a thread locks a mutex it inherits the priority ceiling of the mutex, which is defined by the application as a mutex attribute. With the appropriate ceiling priorities, this is the priority protect protocol, also called the priority ceiling protocol emulation [2][7].

Unbounded priority inversion may be avoided by using one of the last two protocols, thus allowing to achieve a high level of utilization in systems with hard real-time requirements. The priority protect protocol with the appropriate priority ceiling definitions, can also be used to avoid a special kind of priority inversion that appears in multiprocessors, called *remote blocking*. See [5] for a discussion on remote blocking and synchronization in multiprocessors.

D. Other Functions

Other functions are defined in POSIX.4a for managing thread specific data, thread cancellation, delivery of signals to threads, and reentrant functions. For a description of these functions the reader is referred to the draft of the standard [11].

IV. ADDITIONAL REAL-TIME EXTENSIONS

POSIX.4b defines additional real-time extensions to support portability of applications with real-time requirements. The reason for the real-time extensions being divided into two standards is to facilitate a faster approval of the features that were considered essential for real-time —those specified in POSIX.4—, leaving other real-time features for a second standard.

Since POSIX.4b has started its standardization process later than POSIX.4, the features that are now included in the draft documents are more likely to change than those of POSIX.4. Here are some of the most relevant features that are being standardized in POSIX.4b:

A. Timeouts

Some operating system services defined in POSIX.1 and POSIX.4 can suspend the calling process for an indefinite period of time, until the necessary resources become available. In time-critical systems it is important to limit the maximum amount of time that a process can stay waiting for one of these services to complete. This allows to detect abnormal conditions, thus increasing the program robustness and allowing fault-tolerant implementations. POSIX.4b defines new versions of some of the blocking services with built-in timeouts. These timeouts specify the maximum amount of time that the process may be suspended while waiting for the service to complete. The services chosen are those that are most likely going to be used in time-critical code, and did not already have timeout capability:

- Wait for a semaphore to become unlocked
- Wait for the arrival of a message to a message queue
- Send a message to a queue
- Wait for a mutex to become unlocked.

B. Execution-Time Clocks

An optional CPU-time clock is defined for each process and each thread. The POSIX.4 clocks&timers interface is used to manage execution-time clocks. Timers may be defined based on these clocks; they can detect the consumption of an excessive amount of execution time by a process or thread, allowing run-time detection of software errors, or errors in the estimation of the worst-case execution times. Detecting when a task exceeds the worst-case execution time assumed during the analysis phase is very important in robust time-critical systems, because if the assumptions are violated, the results of the schedulability analysis are no longer valid, and the system may miss its deadlines. Execution time clocks allow to detect when an execution time overrun occurs, and to activate the appropriate error handling actions.

C. Sporadic Server

A new scheduling policy is defined (SCHED_SPORADIC) that implements the sporadic server scheduling algorithm [8]. This policy can be used to process aperiodic events at the desired priority level, allowing to guarantee the timing requirements of lower priority tasks. The sporadic server gives fast response times and makes systems with aperiodic events predictable.

D. Interrupt Control

Many real-time systems need the ability to capture interrupts generated by special devices, and handle them at the application program. The functions proposed in the standard allow a process or thread to capture an interrupt by registering a user-written interrupt service routine, to block waiting for the arrival of an interrupt, and to protect critical sections of code from interrupt delivery. The interfaces defined will not achieve complete portability of the application programs due to the many differences in interrupt handling for the different architectures. However, application portability is enhanced by this interface, because a reference model is established and because non portable code is confined to specified modules, thus reducing the number of changes necessary to port the application.

E. Input/Output Device Control

In real-time systems it is common to interact with the environment through special devices such as digital or analog input/output devices, counters, etc. Typically, drivers for these special devices are written by the application developer, and a standardized operation for interfacing with these drivers allows the application operations calling that driver to be well defined. POSIX.4b defines a function that allows an application program to transfer control information to and from a device driver. In the same way as with the interrupt control functions, programs using the device control function may not be completely portable, since the drivers themselves are not usually portable across platforms. However, application portability is enhanced by the use of this interface that provides a reference model for interfacing device drivers.

F. Other Functions

Other interesting features are defined in POSIX.4b such as efficient process creation (spawn). For a description of these functions the reader is referred to the draft of the standard [12].

V. REAL-TIME APPLICATION ENVIRONMENT PROFILES

The POSIX.1 standard along with the real-time extensions and the threads extension constitute a powerful set of interfaces that allow to implement large operating systems capable of addressing real-time requirements. However, for smaller embedded real-time systems a subset of these interfaces would be preferable. For example, many small embedded systems have limited hardware that makes it very difficult to implement features such as a file system or independent address spaces for processes. The real-time application environment profiles (AEPs) defined in POSIX.13 provide the adequate subsets of features of the base standards that are required for a particular application environment. Four real-time AEPs are being defined in POSIX.13:

1) *Minimum System*: Corresponds to a small embedded system with no need for a memory management unit (MMU), no file system (no disk), and no I/O terminal. Only one process is allowed, but multiple threads can run concurrently.

2) *Real-time Controller*: Corresponds to a special purpose controller system. It is like the minimum real-time profile, but adding a file system and I/O terminal. Only one process but multiple threads are allowed.

3) *Dedicated System*: Corresponds to a large embedded system with no file system. It has multiple processes and threads.

4) *Multi-purpose System*: Corresponds to a large real-time system with all the features supported.

Table V summarizes the main characteristics of each of the real-time profiles.

With the real-time AEPs defined, POSIX compliant operating systems can be implemented for a variety of real-time platforms of different sizes and hardware requirements. Applications will be portable from one platform to another, provided that they comply with the same AEP, or that the new platform includes all the features of the previous one. For example, an application will be portable from a minimum system to a real-time controller, or a dedicated real-time system platform. Furthermore, the same application that runs on a very small embedded system can run on a full-featured development system for debugging purposes. With the large range of possibilities defined in the current profiles, current small commercial real-time kernels will have the possibility to provide a POSIX interface. It is forecasted that most of the real-time kernels and operating systems that will be commercially available in the next few years will comply with one of the POSIX real-time AEPs; this will bring application portability to the real-time world.

Table V. Characteristics of the Real-Time Profiles

Profile	File System	Multiple Processes	Threads
Minimum Real-Time System	NO	NO	YES
Real-Time Controller	YES	NO	YES
Dedicated Real-time System	NO	YES	YES
Multipurpose Real-Time System	NO	YES	Optional

VI. CONCLUSIONS

POSIX is an emerging operating system standard that is forecasted to be widely extended in the next few years. One important part of this standard is intended for providing portability to applications with real-time requirements. Application environment profiles are being standardized which will allow implementors to develop real-time POSIX operating systems for a variety of platforms, from small embedded kernels to large real-time operating systems. The standard defines interfaces in different programming languages. In particular, real-time interfaces are being defined for C and Ada, which are the most important standard languages used in practical real-time systems.

The functionality specified in the POSIX standard is similar to what is found in most of the current commercial real-time kernels and operating systems. The POSIX interfaces follow recent results of fixed-priority scheduling theory. Implementations based on the early drafts of POSIX.4 and POSIX.4a have already been developed [1], and show promising results. In summary, the POSIX standard will allow to build analyzable and predictable systems that meet their real-time requirements, and that can be easily portable across different platforms.

VII. ACKNOWLEDGEMENT

The author would like to thank José Angel Gregorio Monasterio for his careful review of this paper.

VIII. REFERENCES

[1] B.O. Gallmeister, and C. Lanier. "Early Experience with POSIX 1003.4 and POSIX 1003.4a". *Proceedings of the IEEE Real-Time Systems Symposium*, December 1991, pp. 190-198.

[2] B.W. Lampson, and D.D. Redell. "Experience with Processes and Monitors in Mesa". *Communications of the ACM* 23, 2, February 1980, pp. 105-107.

[3] J. Leung, and J.W. Layland. "On Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks". *Performance Evaluation 2*, 237-50, 1982.

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

[5] R. Rajkumar, L. Sha, and J.P. Lehoczky. "Real-Time Synchronization Protocols for Multiprocessors". *IEEE Real-Time Systems Symposium*, December 1988.

[6] L. Sha, R. Rajkumar, and J.P. Lehoczky. "Priority Inheritance Protocols: An approach to Real-Time Synchronization". IEEE Trans. on Computers, September 1990.

[7] L. Sha, and J.B. Goodenough. "Real-Time Scheduling Theory and Ada". IEEE Computer, Vol. 23, No. 4, April 1990.

[8] B. Sprunt, L. Sha, and J.P. Lehoczky. "Aperiodic Task Scheduling for Hard Real-Time Systems". *The Journal of Real-Time Systems*, Vol. 1, 1989, pp. 27-60.

[9] ISO/IEC Standard 9945-1:1990, and IEEE Standard 1003.1-1990, *"Information Technology —Portable Operating System Interface (POSIX)— Part 1: System Application Program Interface (API) [C Language]*". Institute of Electrical and electronic Engineers, 1990.

[10] IEEE Standards Project P1003.4, "*Draft Standard for Information Technology —Portable Operating System Interface (POSIX)— Part 1: System Application Program Interface (API) — Amendment 1: Realtime Extension [C Language]*". Draft 13. The Institute of Electrical and Electronics Engineers, September 1992.

[11] IEEE Standards Project P1003.4a, "*Threads Extension for Portable Operating Systems*". Draft 6. The Institute of Electrical and Electronics Engineers, February 1992.

[12] IEEE Standards Project P1003.4b, "*Draft Standard for Information Technology —Portable Operating System Interface (POSIX)— Part 1: Realtime System API Extension*". Draft 6. The Institute of Electrical and Electronics Engineers, February 1993.

[13] IEEE Standards Project P1003.13, "*Draft Standard for Information Technology —Standardized Application Environment Profile— POSIX Realtime Application Support (AEP)*". Draft 5. The Institute of Electrical and Electronics Engineers, February 1992.