On Integrating POSIX Signals into a Real-Time Operating System[†]

Arnoldo Díaz, Ismael Ripoll and Alfons Crespo
Universidad Politécnica de Valencia
Camino de Vera s/n, Valencia, Spain
ardiara@doctor.upv.es, {iripoll,alfons}@disca.upv.es

Abstract

POSIX is a set of international standards whose main goal is to support applications portability at the source code level. It defines an operating system interface and environment based on the UNIX operating system. A series of POSIX Real-Time standards have been defined to support real-time applications portability. However, portability is often sacrificed in Real-Time Operating Systems (RTOS) since they must provide predictability and low overhead. In this paper we discuss the POSIX Real-Time Signals Extension and we propose an approach to implement it into a RTOS in order to improve the performance of systems where the execution entities are threads instead of processes, making a compromise between standard conformance and efficiency. The notion of Signal Owner is presented to avoid ambiguity and assign signal priorities accordingly in priority-based systems and to minimize overhead. Also, we discuss the consequences of the defined default POSIX signal actions in real-time systems and the implementation of signal queueing in the Minimal Real-Time System Profile, and different approaches are presented to address these issues. The paper also describes briefly the implementation of the POSIX Real-Time Signals Extension in RTLinux.

1 Introduction

POSIX[1], the Portable Operating System Interface, is an evolving set of standards whose main goal is to support applications portability at the source code level. POSIX defines an operating system interface and environment based on the UNIX[2] operating system. It doesn't specify portability of application binary code or the OS kernel itself . Instead, it defines a series of Application Program Interfaces (APIs), that are written contracts between kernel writers and application writers.

Although initially was used to refer to the original IEEE Std 1003.1-1988 [3], the name POSIX more correctly refers to a family of related standards: IEEE Std 1003.n and the parts of ISO/IEC 9945, where n is a number that indicates a specific part of the standard. The term "POSIX.1" emerged to differentiate the 1003.1 standard with the "POSIX" family of standards. POSIX has evolved since it was first proposed and approved, and many small working groups have been formed to address specific areas of the standardization effort, as the base UNIX functionality (1003.1), the commands set (1003.2 or POSIX.2), the real time extension (1003.4 or POSIX.4), and so forth. Over the years, additions and amendments to the POSIX standard has been done. For instance, the shell and utilities (commands set) standard has been incorporated into subsequent revision of POSIX 1003.1 and thus a POSIX.2 standard no longer exists. Similarly, real time extension (1003.4 or POSIX.4) were incorporated into POSIX.1. Nowadays, the IEEE Computer Society's Portable Applications Standards Committee (PASC) group continues the development the POSIX family of standards. The goal of the PASC standards has been to promote application portability at the source code level. The 2004 edition of the 1003.1 standard was published on April 30th 2004 and it has four components or volumes [4, 7, 6, 5]. This standard defines a "standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level" [4].

[†]This work has been partially supported by the the European Union project IST-2001-35102 (OCERA project)
[1]POSIX is a registered trademark of the IEEE
[2]UNIX is a registered trademark of The Open Group

Applications portability at the source code level is also needed for real time systems, and for that reason the PASC Real-time System Services Working Group (SSWG-RT) has developed a series of standards for real-time systems. This group is aimed to "develop standards which are the minimum syntactic and semantics changes or additions to the POSIX standards to support portability of applications with real-time requirements" [1]. The standards relevant to the development of real time and embedded systems are listed in Table 1.

IEEE Standard	Name
1003.1b-1993	Real-time Extension
1003.1c-1995	Thread
1003.1d-1999	Additional Real-time Extensions
1003.1j-2000	Advanced Real-time Extensions
1003.1q-2000	Tracing

TABLE 1: *POSIX Real-Time Standards*

From this list of standards, the Real-Time Extension defined formerly by 1003.1b or POSIX.1b, and support for multiple threads in a process, defined by 1003.1c, are the standards most commonly implemented. As said before, these standards have been integrated into the POSIX 1003.1 standard.

The POSIX definition of real-time in operating systems is "The ability of the Operating System to provide a required level of service in a bounded response time" [4]. Trough the Real-Time Extension, therefore, is possible to add to POSIX.1 those services necessary to achieve the predictable timing behavior needed by a real-time operating system, specially in areas such as process scheduling, real-time signals, virtual memory management, clock and timers. Also, the Real-Time Extension facilitate concurrent programming for process synchronization, shared memory, asynchronous I/O and synchronized I/O, and message queues.

However, including all the features of POSIX in an operating system may not be appropriate. When POSIX was defined in the first place the system's execution units were process or *heavyweight process*, but now many systems use threads or *lightweight process* instead, and therefore implementing all POSIX features may be inefficient nor necessary. There are systems where only a subset of the interfaces provided by POSIX are sufficient, like in small embedded real-time systems that have space and resources limitations. The POSIX.13 profile standard [8] was defined to address this problem, providing the adequate subsets of features of the base standard that are required for a particular application environment. This standard defines four real-time application environment profiles:

- *PSE51* or *Minimal Real-Time System Profile*: Intended for small embedded systems with no Memory Management Unit (MMU), no disk (no file system) and no terminal. Only one process allowed with multiple threads running concurrently.
- *PSE52* or *Real-Time Controller*: Targeted to special purpose controller, with no MMU, but with a disk containing a simplified file system.
- *PSE53* or *Dedicated Real-Time System*: Correspond to large embedded system with no disk, but with an MMU. Applications may benefit using memory protection and network communications in this profile.
- *PSE54* or *Multi-Purpose Real-Time System*: This profile is intended for large real-time system with all the features, including a development environment, network communications, file system on disk, graphical user interfaces, and so on.

Table 2 summarizes the characteristics of the real-time environments profiles.

Profile	Multiple Processes	Threads	File System
PSE51	NO	YES	NO
PSE52	NO	YES	YES
PSE53	YES	YES	NO
PSE54	YES	Optional	NO

TABLE 2: *Real-Time Environment Profiles*

In this paper we discuss the real-time signal extension of POSIX.1 and how this extension can be improved in systems where the execution entities are threads instead of processes, as in Real-Time Operating Systems conforming the POSIX PSE51 or Minimal Real-Time System Profile.

The paper is organized as follows: Section 2 gives an overview of POSIX Real-Time Signals Extension. Section 3 discuses the Real-Time Signals Extension and how this extension can be improved in systems where the execution entities are threads instead of processes. Section 4 describes briefly the implementation of POSIX Real-Time Signals Extension in RTLinux. Finally, Sections 5 is for conclusions and future work.

2 Real-Time Signals

2.1 Signal Concepts

A signal is a software equivalent to a hardware device interrupt. Signals are used in POSIX as a notifying mechanism when an important event has occurred in the system. Examples of such events are exception handling or asynchronous interrupts. In fact, signals is perhaps the most used mechanism for asynchrony in UNIX-like systems. Next we review some concepts that are important when dealing with signals.

2.1.1 Signal definition

In the POSIX context, a signal is defined as "a mechanism by which a process or thread may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term signal is also used to refer to the event itself" [4].

2.1.2 Classification of signals

Signals can be classified depending on the way they are generated. By this criteria, signals can be classified as generated synchronously, generated asynchronously, or as sent explicitly to a process or thread.

When a process or thread, or the system itself, needs to inform to the system that something important has occurred, a signal can be generated to the corresponding process. Signals can be generated when an exception occurs, like dividing by zero or a page fault exception. In this case, the signal is *generated synchronously* since it is sent immediately in response to something the process itself has done. Also, signals can be *generated asynchronously* to the process's execution, as when a timer expires or to inform an asynchronous I/O completion. Furthermore, a process can *send explicitly a signal to a process or thread*, using the *kill()*, *sigqueue()* or *pthread_kill()* functions when used as an interprocess mechanism.

2.1.3 Signal generation

POSIX.1 specifies that a signal is said to be *generated* for, or *sent to* a process or thread when the event that causes the signal first occurs. Determining if a signal has been generated for a process or for a specific thread within the process is done when the signal is generated. There are situations where the signal is generated for an action attributable to a particular thread, as when performing an erroneous arithmetic operation. In these cases, the signal shall be generated for the thread that caused the action. In other situations, signals are generated in association with a process ID or process group ID or an asynchronous event, such as terminal activity, and then the signal shall be generated for the process.

2.1.4 Signal delivery and acceptance

A signal is said to be *delivered* to a process when the appropriate action for the process and signal is taken. We discuss the possible actions to be taken later. On the other hand, a thread can wait for a signal to arrive using any of the *sigwait()* family of functions (*sigwait()*, *sigwaitinfo()*, *sigtimedwait()*) and the action defined for the signal is then not taken. In these cases the signal is no delivered but accepted by the thread. A signal is said to be *accepted* by a process when the signal is selected and returned by one of the *sigwait()* functions.

A signal is *pending* during the time interval between the generation of the signal and its delivery or acceptance. This interval of time generally cannot be detected by an application. A process or thread can *block* the delivery or acceptance of a signal, and in that case the blocked signal will remain blocked until it is unblocked and delivered, accepted by a call to the *sigwait()* function, or the action associated with it is set to ignore the signal.

POSIX.1 specifies that if a signal is generated for a process and one or more threads are active within the process, the signal shall be delivered to exactly one of those threads within the process which is in a call to a *sigwait()* function selecting that signal or has not blocked delivery of the signal. Nevertheless, determining which thread within the process is the receiving thread is not clearly specified. If no thread makes a call to a *sigsuspend()* function or any of the *sigwait()* functions selecting that signal and all threads within the process block delivery of the signal, the signal shall remain pending until a thread a calls any of these functions selecting that signal, or a thread unblocks delivery of the signal.

2.1.5 Signal mask

In order to define the set of signals currently blocked from delivery to a thread, a thread's *signal mask* is used. POSIX.1 specifies that the signal mask for a thread shall be initialized from that of its parent or creating thread, or from the corresponding thread in the parent process if the thread was created as the result of a call to *fork()*. The *pthread_sigmask()*, *sigaction()*, *sigprocmask()*, and *sigsuspend()* functions control the manipulation of the signal mask.

2.1.6 Signal actions

Each process has an action to be taken in response to each signal defined by the system. A process can ignore the signals, take the default action for the signal, or can execute the application-defined signal handler. The definition of the signal action for a particular signal can be done using the *signal()* or *sigaction()* functions.

Next we present an overview of the POSIX.1 standard, which is mandatory for all POSIX systems, and the Real-Time Signals Extension.

2.2 POSIX.1 Signals (UNIX Signals)

Signals are a very important part of an operating system. Signal services are a basic mechanism within POSIX-based systems and are required for error and event handling. POSIX.1 defines a set of functions that must be present in all POSIX systems, and some of them are described in Table 3 [7].

Function	Description
sigemptyset(sigset_t *set)	Initializes the signal set, excluding all signals.
sigfillset(sigset_t *set)	Initializes the signal set, including all signals.
sigaddset(sigset_t *set, int signo)	Adds an individual signal to the signal set.
sigdelset(sigset_t *set, int signo)	Deletes an individual signal from the signal set.
sigpending(sigset_t *set)	Stores the set of signals that are blocked from delivery.
sigismember(const sigset_t *set, int signo)	Tests if a signal is a member of the set.
sigprocmask(int how, sigset_t *set, sigset_t *oset)	Examine or change the calling process signal mask.
sigaction(int sig, struct sigaction *sa, struct sigaction *osa)	Allows to specify the action associated with a specific signal.
pthread_sigmask(int how, sigset_t *set, sigset_t *oset)	Examine or change the calling thread signal mask.
sigsuspend(sigset_t *sigmask)	Suspends a thread until delivery of a signal in sigmask.
sigwait(sigset_t *set, int *sig)	Similar to sigsuspend, but no signal handler is executed.
kill(pid_t pid, int sig)	Sends a signal to a process.
pthread_kill(pthread_t thread, int sig)	Sends a signal to a thread.

TABLE 3: *POSIX.1 Signal Functions*

As mentioned before, each process has an action to be taken in response to each signal defined by the system. To indicate the action to be taken POSIX.1 defines several mechanisms. One of them is trough the use of the *sigaction()* function. These functions use an argument of type *struct sigaction* whose members are shown in Table 4. The first one is a pointer to a function and it is used to define a POSIX.1 signal handler that will be invoked when a signal is accepted by the process. If this pointer has the value of the SIG_IGN, then the action to be taken is to ignore the signal. If, on the other hand, the value is the macro SIG_DFL, then the action is the default one, as defined in *signal.h* [4]. The default action for the majority of signals is to abnormal termination of the process, or stopping the process. The fourth member is also a pointer to a function and it will be discussed in the next section.

Member	Use
void (*sa_handler)(int)	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t sa_mask	Set of signals to be blocked during execution of the signal handling function.
int sa_flags	Special flags.
void (*sa_sigaction) (int, siginfo_t *, void *)	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL

TABLE 4: *Struct Sigaction Members*

POSIX.1 defines several different signals. Some of them are used primarily by the operating system, while two are defined for application use: *SIGUSR1* and *SIGUSR2*. This means that an application can make use of this two signal the way it wants.

Signals have been used as a basic mechanism for asynchronous communication between heavyweight processes or processes. However, the use of lightweight processes or threads have provided better mechanism for inter threads communication since threads share the same process's address space. Furthermore, signals presents some drawbacks to be used in real-time applications: too few number of signals for application use, no priority on signal delivery, limited information content associated to a signal, signal lost due to not queueing, among others problems. In order to overcome these limitations, the real-time signals extension has been proposed.

2.3 POSIX Real-Time Signals Extension

The Real-time Signals Extension is "a determinism improvement facility to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signal functions" [4].

Signals have been used as a synchronous communication mechanism for a long time, but nowadays the use of threads are a good and alternate mechanism for synchronous communication other than signals. Nevertheless, real-time applications often need signals for asynchronous notification for events such as timeout, message arrival, and hardware interrupt, and expect high performance and low latency in signal's notification. Real-time Signals provide the reliable high-performance mechanism to support such notification.

The Real-time Signals Extension improves POSIX.1 signals in several ways. For instance, there are more signals available for application use. The new signal numbers go from SIGRTMIN TO SIGRTMAX, inclusive. There must be at least RTSIG_MAX real-time signals, and the minimum number of real-time signals that the implementation is required to support has been increased from the number specified in POSIX.1, form 8 to 16. The rationale for this increase is that there are many applications that have more than 8 different kinds of events.

To define the action to be taken in response to a signal, the *sigaction()* function can be used. This function uses a parameter of the type *struct sigaction* to specify the action to be taken, but also uses this structure for other purposes in the case of real-time signals (See Table 4).

One of them is the *sa_flags* field. If the bit called SA_SIGINFO of the *sa_flags* field is set , it indicates that the corresponding signal should be queued to the process, and that the signal will carry more information. It also indicates that the signal-catching function, if defined, will follow the prototype indicated next:

```
void (*sa_sigaction)(int signo,
                     siginfo_t *info,
                     void *);
```

The POSIX.1 signal-catching function receives only one argument, but the real-time signal one receives three. The second argument of this function is of the type *siginfo_t*, and its structure is shown next:

```
typedef struct {

   int si_signo;  // Signal number

   int si_code;   // Signal code

   int si_errno;  // An errno value
                  // associated
                  // with this signal

   pid_t si_pid;  // Sending process ID

   uid_t si_uid;  // Real user ID of
                  // sending process

   void *si_addr; // Address of faulting
                  // instruction

   int si_status; // Exit value or signal

   long si_band;  // Band event for SIGPOLL

   union sigval si_value; // Signal value

} siginfo_t;
```

The *union sigval* is used to pass a value along with the signal. The value could be either an integer or a pointer data value. This extra information associated to a signal can be used by an application to identify the source of the signal for events of the same kind that use the same signal number.

Since POSIX.1 signals are not queued some events may be lost. To overcome this limitation, real-time signals are queued. If the SA_SIGINFO bit of of the *sa_flags* field is set for a given signal, multiple occurrences of the signal are queued to the process. When the process is able to receive its signals, they will be dequeued and delivered to the process without signal lost. There's a limit of the number of signals that can be queued to a process. This limit is defined in SIGQUEUE_MAX (*limits.h*). The minimum acceptable is 32.

Another improvement is that real-time signals must be delivered in order. These signals are dequeued and delivered lowest-numbered signal first. This feature allows applications to use the number as a signal priority.

There are new signal-related functions in the Real-time Signals Extension, and are shown in Table 5.

Function	Description
sigqueue(pid_t pid, int signo, union sigval value)	Sends a queued signal to a process.
sigwaitinfo(sigset_t *set, siginfo_t *info)	Similar to *sigwait()*, with additional info.
sigtimedwait(sigset_t *set, siginfo_t *info, struct timespec *timeout)	Similar to *sigwaitinfo()*, but waits *timeout* units.

TABLE 5: *Real-Time Signal Functions*

The *sigqueue()* function is used to send queued signals to a process. The Real-time Signals Extension defines that if signal is generated for the sending process (as will be the case in a single-process system) using the *sigqueue()* function, and if the signal is not blocked for the calling thread and if no other thread has the signal unblocked, it shall be queued and sent to the calling thread. This means that if a signal is sent to one process and within the process are multiple threads, to define the calling (or receiving) thread must be necessary block the signal in all but the calling thread.

3 Integrating Real-Time Signals into a RTOS

POSIX was formerly defined to standardize the AT&T System V and Berkeley CSGR systems interfaces, that operated in a heavyweight process environment. Subsequent modifications and additions to the POSIX standard have been done taking care of backward compatibility, which means that old applications, designed to work in a heavyweight process environment, may continue to work correctly in systems that conform to the new standard. As POSIX evolved, additions as the POSIX 1003.1c (threads) standard have made possible the creation of more efficient and flexible applications without loosing compatibility with existing API. Nevertheless, when the real-time signal extension is used in real-time systems where the execution entities are threads instead of processes, there are specific areas in which the standard is ambiguous. On the other hand, conformance with standard APIs is always desirable since it guarantees source code portability and thus flexibility. The authors believe that an extension to the real-time signals standard could be helpful in avoiding ambiguity, reduce overhead and simplify implementation, preserving portability and making a compromise between standard conformance and predictability. In this section we discuss the POSIX Real-Time Signals Extension integration into a real-time operation system conforming the POSIX Minimal Real-Time System Profile, and propose an extension to the standard semantics.

There are some issues where the extension proposed may be improve the real-time signals implementation and portability. One of them is related to selecting the target thread of a signal. In a heavyweight process environment, determining the target process of a signal is straightforward. However, in a lightweight process environment, POSIX doesn't specify it clearly. Determination whether the signal has been generated for the process or for a specific thread within the process should be made at signal's generation time. If the action that generated the signal is attributable to a specific thread, as a page fault exception, the signal is then generated for the thread that caused the action. But if the signal is generated for the whole process, it shall be delivered to **exactly one** of those threads within the process that has not blocked the signal. The operating system has to set up a loop through the thread's list to check every thread's signal mask and to find out a suitable thread to deliver the signal. This search implies both system overhead and non determinism. If there's more than one thread with the signal unblocked, is not specified which one to choose. This indeterminism can be avoided if a programmer takes care of blocking the signal in all but one thread, but this mechanism doesn't avoid overhead, and if it is missed or not set up properly the system is left in an indeterministic state, which is not acceptable for a real-time system.

Another issue is related to the signal's delivery and acceptance time. In the previous section, it was mentioned that POSIX defines that a signal is pending during the time interval between the generation of the signal and its delivery and acceptance, and that the length of this time interval generally cannot be detected by the application. When a signal must be delivered and accepted is not clearly defined in the POSIX standard, and therefore a POSIX-conformance real-time system doesn't provide predictability in this issue. This may also lead to unnecessary overhead, as we show next.

POSIX defines several scheduling algorithms [4], and among them the preemptive fixed-priority algorithm [10] is the one used for the vast majority of real-time operating systems. In such operating systems, processor is assigned to the highest priority thread, preempting a lower priority one if necessary. Once the scheduler decides which thread to execute, in some operating systems it's a common practice to program the next scheduling point (the time when a new scheduling decision must be taken) looking for the nearest highest-priority scheduling event (a new thread is created, a thread becomes ready, a timer expires, etc) and arming a timer at that time. In the case of a signal delivery, at what priority level should it be considered? Delivering signals at at highest priority level seems to be the safer choice, but this decision leads to undesirable overhead. Let's suppose that a given thread $j_i$with priority p_i becomes ready at time t_i and executes until time t_f in a priority-based system. Suppose that a timer expires at time tr ($t_i \leq tr \leq t_f$) and as a consequence of timer expiration signal s_k shall be delivered to thread j_k. Thread j_k has priority p_k ¡ p_i. If signal s_k is delivered at the highest priority level, j_i's execution is suspended at time tr and signal s_k delivered, but since the target thread j_k has lower priority than j_i, this one (j_i) continues its execution and the action associated with the signal is executed until j_i finishes its execution. This overhead could be avoided if s_k is delivered at j_k's priority level. In this case, j_i will not be unnecessarily suspended at time tr and signal s_k will be delivered when j_i finishes.

Overhead due to the issues discussed above can be reduced with the notion of *signal owner*, which is presented next. We also discuss the consequences of the defined default POSIX signal actions in real-time systems and propose a different way to address this problem, along with a discussion about signal handlers for synchronously generated signals. Finally, we discuss the implications of signal queueing in the Minimal Real-Time System Profile.

3.1 Signal Owner

We propose the *signal owner* notion to eliminate or reduce some of the problems discussed before. We'll see how this concept extends the semantics of some functions related with threads and can be used to simplify signal's implementation in a RTOS and to preserve portability among applications. The main motivation in defining the signal owner notion is to avoid ambiguity or unpredictability in determining the receiving thread when asynchronously generated signals are used. The *signal owner* is defined as the thread that installed the signal handler last, or that made a call to any of *sigwait( )*

functions last. Using the signal owner notion not also simplifies system implementation but reduces overhead. We discuss next how this can be accomplished.

The *sigaction()* function is used as a mechanism for threads to define the action associated with a specific signal. This function supersedes the *signal()* function and POSIX suggests that it should be used in preference. One of the actions associated with a specific signal is to handle it by setting up a function to be called when a signal is delivered. This function is called the *signal handler*. As we've seen, the signal owner is defined as the thread that installed the signal handler last. When a signal is asynchronously generated, it's delivered to its owner, and if a handler is defined for that signal, it is executed at the owner's priority level. Determining the calling thread is done without ambiguity.

The *sigsuspend()* function can be used to wait for a signal, and once it is received, the defined action is executed. If the action defined for the received signal is to execute a signal handler, then *sigsuspend()* shall return after the signal handler returns if the thread in a call of the *sigsuspend()* is the signal owner.

There exist another way for a thread to wait for a signal. The *sigwait()* functions provide a synchronous mechanism for threads to wait for signals. When a thread makes a call to a *sigwait()* function, it is suspended until any of the waited signals arrive. When this happens, the signal handler or the action defined for the signal is not executed; instead, the blocked thread returns when the signal arrives, allowing a user-level signal handling operation to be executed. The use of this functions avoids the overhead associated with the execution of a kernel-level signal handler. The family of *sigwait()* functions offer an efficient method to deal with signals, and in heavyweight process environment determining the receiving process is done unambiguously. Nevertheless, determining it in lightweight process environment is not as clear. When these functions where proposed, one important question for the POSIX group was how many threads that are suspended in a call to a *sigwait( )* functions for a signal should return from the call when the signal is received [7]. Four choices were considered:

1. Return an error for multiple simultaneous calls to sigwait functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to *sigwait()* for the same signal was felt to be overly restrictive, and the consensus was that exactly one thread that was suspended in a call to a sigwait function for a signal should return when that signal occurs. However, determining which of the threads should return when the signal is sent is not specified. The concept of signal owner offers a way to address this problem, since it clearly defines the which is receiving thread. The signal owner is, in this case, the thread that made a call to any of *sigwait( )* functions last. If more than one thread made a call to any of *sigwait( )* functions, signal acceptance should be done in a *First In Last Out* basis.

If signals are used as a asynchronous notification mechanism, determining the calling thread is straightforward using the notion of signal owner. When a signal is generated, either because a timer has expired, a message has been received on a message queue, or an asynchronous I/O has been completed, the operating systems doesn't need to look to every thread's signal mask to find out a suitable thread to deliver the signal. It only needs to check the owner's signal mask and if the signal is not blocked or if is not set to be ignored, it is delivered to the signal owner. On the other hand, if it is blocked for that thread, it is marked pending until unblocked.

Deciding when to deliver the signal is also simpler and more congruent in a priority based system using the signal owner concept. The signal inherits the owner's priority and therefore signal delivery and signal handler execution is done at the corresponding priority level. This way, signal delivery and signal handler execution will be done without interfering with higher priority threads execution, giving predictability to the system.

The *pthread_kill()* function deserves special attention. In a multi-threaded environment, since threads share process address space, there are better and more efficient mechanisms than using signals for inter-thread communication, and therefore its use should not be encouraged for this purposes. With the semantics proposed in this paper, the *pthread_kill()* function should be limited to a very specific situations, as to stop, continue or terminate a given thread's execution, or to check the validity or existence of the given thread. More about this is discussed next.

As summary, the notion of signal owner is proposed in order to avoid ambiguity in determining the receiving thread when asynchronously generated signals are used. Furthermore, it gives a clear way to define the priority level at which a signal handler must be executed. This notion also reduces overhead and simplifies implementation.

3.2 Signals default action

Three types of action are associated to a signal: ignore the signal, execute a signal handler, or execute the default action. Depending on the signal, the default action can be [4]:

- T Abnormal termination of the process. The process is terminated with all the consequences of *exit()* except that the status made available to *wait()* and *waitpid()* indicates abnormal termination by the specified signal.
- A Abnormal termination of the process. Additionally, implementation-defined abnormal termination actions, such as creation of a core file, may occur.
- I Ignore the signal.
- S Stop the process.
- C Continue the process, if it is stopped; otherwise, ignore the signal.

As can be seen from Table 6, that shows a list of signals that may be present in any POSIX conformance system, the default action for most signals is the abnormal termination of the process or stopping the process. This seems appropriate in a heavyweight process environment, but in a PSE51 system where there's only one process and many threads, the default action is too drastic. From a fault-tolerance point of view, either an abnormal termination of the process or stopping the process may not be acceptable since the PSE51 system is terminated as a whole. The authors propose that for a PSE51 system instead of terminating or stopping the whole process, the default action be cancelling the offending or calling thread in the first place, and if the thread is not cancelled, then suspending the thread. In this case, the system will still be active and perhaps a degraded level of system operation can be activated. In any case, the proposed defaults actions are always less restrictive than the current standard. With this new approach the *pthread_kill()* function could be used safely since only the receiving thread and not the process is affected via the default action.

On the other hand, if a signal is not generated synchronously, then the thread to be suspended is the signal owner. Nevertheless, for asynchronous generated signals, or when a signal is generated using the *sigqueue()* function, not always a signal owner is clearly defined. If no signal handler has been defined for the signal, or if there is no thread waiting for it with the *sigwait()* functions, the signal doesn't have an owner accordingly to our definition. If the signal is not set to be ignored, the default action takes place. Since no signal owner is defined, which thread should be suspended? In order to solve this ambiguity, the authors propose that as default every signal is set to be ignored when the system starts execution. Setting the default action for a signal must be done explicitly, and the thread that sets it becomes the signal owner. With this approach, a signal owner is always defined and ambiguity is avoided at any time.

It could be claimed that applications should install handlers for every signal used and that a change in the signal default action is not needed. However, some considerations must be taken into account. If in an environment where the execution entities are process the default action is to stop the offending execution entity (process), a natural extension of the standard to environments where the execution entities are threads should be to also stop the the offending execution entity, a thread in this case. On the other hand, signals use are not a common practice among programmers, and therefore that a signal is the cause of an abnormal termination of the whole process may not be easily detected or prevented and may lead to catastrophic results. As an example, the European Ariane 5 launcher crashed about 40 seconds after takeoff due to a floating-point error not handled by any exception-handler [9]. Perhaps a different default action could be helpful in avoiding this costly disaster.

Signal	Default Action	Description
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped, or continued.
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hang-up.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.
SIGTSTP	S	Terminal stop signal.
SIGTTIN	S	Background process attempting read.
SIGTTOU	S	Background process attempting write.
SIGUSR1	T	User-defined signal 1.
SIGUSR2	T	User-defined signal 2.
SIGPOLL	T	Pollable event.
SIGPROF	T	Profiling timer expired.
SIGSYS	A	Bad system call.
SIGTRAP	A	Trace/breakpoint trap.
SIGURG	I	High bandwidth data is available at a socket.
SIGVTALRM	T	Virtual timer expired.
SIGXCPU	A	CPU time limit exceeded.
SIGXFSZ	A	File size limit exceeded.
SIGRTMIN to SIGRTMAX	T	Real-Time user-defined signals.

TABLE 6: *Signals List*

3.3 Synchronous signals

So far we have mainly discussed asynchronously-generated signals. The notion of signal owner lets define, for a given signal, a per-thread signal handler. For synchronously-generated signals, however, a global signal handler seems to be more appropriate. Let's suppose that a thread tries to perform a zero-divide operation. The operating systems detects the invalid operation and generates a SIGFPE signal for the thread that caused the erroneous arithmetic operation. If no signal handler is defined for SIGFPE, the offending thread is cancelled or suspended and the system will go on. If a signal handler is defined, on the other hand, proper actions may take place to solve the problem or to activate a fault-tolerant action in order to leave the system in a stable state. Since all threads share the same signal handler, the *pthread_self()* function can be used inside the handler code to allow detection of the offending thread and to take the proper action. Furthermore, using this technique, the creation of a handler's library is easier and can be used to add a certain level of fault-tolerance to the system.

3.4 Queued Signals

Before the *sigqueue()* function was introduced, signals were used in a heavyweight process environment as software interrupts for specific events. When signal queueing was incorporated into the POSIX standard it made possible the use of signals as message queues for inter-process communication, which offers many advantages. However, in a single-process multi-threaded environment there are better mechanisms for inter-threads communication other than signals. Using *sigqueue()* for this purpose is not the most efficient option since latency of signals can be too high, and for the overhead and resources waste associated to signal queueing. Nevertheless, in a PSE51 system the *sigqueue()* function can be useful for generating software interrupts that will use handlers defined for other signals, such as the handler of a group of timers. Also, these group of timers can use the same signal number benefiting from the signal queueing services.

Independently of the mechanism used to send a queued signal, it may fail if the system has insufficient resources to queue the signal. POSIX defines an explicit limit on the number of queued signals that a process could send, and it suggest that this limit is "per-sender", even though it does not specify that the resources be part of the state of the sender. The notion of signal owner, on the other hand, offers a different approach to address this issue. If the limit on the number of queued signals that a process could send is set "per signal owner" instead than "per sender", implementation is simplified and performance improved. When the signal owner is chosen to be executed, it dequeue any queued signals directly.

4 Implementation

A preliminary version of POSIX real-time signals including some of the proposals made in this paper has been implemented in RTLinux [12]. RTLinux is a small and fast hard real-time operating system. It uses Linux as a general-purpose operating system and the real-time components are left in a single, multi-threaded, real-time process running on a bare machine. Linux can be seen as a PSE54 or Multi-Purpose Real-Time System, while RTLinux as a PSE51 or Minimal Real-Time System Profile system. Our approach in implementing the POSIX Real-Time signals and timers was to build a PSE51 conformance system making a compromise between efficiency and standard conformance. The implementation of both POSIX.1 [12] and POSIX Real-Time signals was done using the OCERA [2] GPL kernel (based on RTLinux version 3.2pre1 and Linux kernel 2.4.18).

5 Conclusions and Future Work

POSIX defines a standard operating system interface and environment to support applications portability at the source code level. Implementing all the features of POSIX in an real-time operating system may not be appropriate. There are systems where a subset of the interfaces provided by POSIX are sufficient. Furthermore, portability is often sacrificed to provide predictability and low overhead. In this paper we discussed the POSIX Real-Time Signals and proposed an extension to the standard semantics in order to simplify implementation, improve performance and to provide portability of systems where the execution entities are threads instead of processes, making a compromise between standard conformance and efficiency. The authors proposal does not add new system calls and it is compatible with current methodology related with the use of signals. The notion of Signal Owner has been presented to improve predictability, minimize overhead and give flexibility. Also, we discussed the consequences of the defined default POSIX signals action in real-time systems and propose a different way to address this issue in order to avoid unexpected system termination. Furthermore, we discussed the implications of signal queueing in the Minimal Real-Time System Profile and a proposal was made to improve performance and simplify implementation. The POSIX Real-Time Signals Extension has been implemented in RTLinux. Future work includes implementation of the proposal in another RTLinux areas, such as Asynchronous I/O.

References

[1] M. Aldea and M. Gonzalez-Harbour, 2003, *Evaluation of New POSIX Real-Time Operating Systems Services for Small Embedded Platforms*, Proceedings of the 15th Euromicro Conference on Real-Time Systems, pp.161-168.

[2] A. Crespo and I. Ripoll, 2003, *OCERA Whitepaper*, OCERA Project and Universidad Politecnica de Valencia.

[3] IEEE 1003.1-1988, 1988, *IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std 1003.1-1988*, Institute of Electrical and Electronic Engineers.

[4] IEEE Std 1003.1, 2004 Edition, 2004, *The Open Group Technical Standard Base Specifications, Issue 6. Base Definitions*, Institute of Electrical and Electronic Engineers and The Open Group.

[5] IEEE Std 1003.1, 2004 Edition, 2004, *The Open Group Technical Standard Base Specifications, Issue 6. Ratioale (Informative)*, Institute of Electrical and Electronic Engineers and The Open Group.

[6] IEEE Std 1003.1, 2004 Edition, 2004, *The Open Group Technical Standard Base Specifications, Issue 6. Shell and Utilities*, Institute of Electrical and Electronic Engineers and The Open Group.

[7] IEEE Std 1003.1, 2004 Edition, 2004, *The Open Group Technical Standard Base Specifications, Issue 6. System Interfaces*, Institute of Electrical and Electronic Engineers and The Open Group.

[8] IEEE Std 1003.13-2003,2003, *IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Suppor*, Institute of Electrical and Electronic Engineers.

[9] J.M. Jazequel and B. Mayer, 1997, *Design by Contract: The Lessons of Ariane*, IEEE Computer, pp.129-130.

[10] C.L. Liu and J.W. Layland, 1973, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, pp.44-61

[11] J. Vidal, F. Gonzales and I. Ripoll, 2002, *POSIX Signals Implementation in RTLInux*, OCERA Project and Universidad Politecnica de Valencia.

[12] V. Yodaiken, 1999, *The RTLinux Manifesto*, Proceedings of the 5th Linux Expo.