POSIX Signals implementation in RTLinux

Authors: J. Vidal, F. Gonzálvez, I. Ripoll.

Signals

Signals are an integral part of multitasking in the UNIX/POSIX environment. Signals are used for many purposes, including:

- Exception handling (bad pointer accesses, divide by zero, etc.)
- Process notification of asynchronous event occurrence (timer expiration, I/O completion, etc.)
- Emulation of multitasking.
- Interprocess communication.

What is a Signal?

A POSIX signal is the software equivalent of an interrupt or exception occurrence. When a process ``gets" a signal, it means that something has happened which requires the process's attention.

Because a thread can send a signal to another thread, signals seems can be used for interprocess communication. Signals are not always the best interprocess communication mechanism; they're slow, limited and they asynchronously interrupt your thread in ways that require clumsy coding to deal with.

Signals are mostly used for other reasons, like the timer expiration and asynchronous I/O completion.

There are legitimate reasons for using signals to communicate between processes. First, signals are something of a lowest common denominator in UNIX systems--every-ons got signals, even if they don't have anything else! Another reason is that signals offer an advantage that other, higher-performance communication mechanisms do no support: signals are asynchronous. That is, a signal can be delivered to you while you are doing something else. The advantages of asynchrony are the immediacy of the notification and the concurrence.

POSIX interface to signals

The following synopsis shows the POSIX-conformant signal function:

The following are all required in all POSIX-conformant systems. I will list the name they have in RTLinux.
Also exists a real time extension that won't be treated here.

```
#include <unistd.h>
#include <signal.h>
```

```
rtl_sigaddset(sigset_t *set, sig);
rtl_sigdelset(sigset_t *set, sig);
rtl_sigismember(sigset_t *set, sig);
rtl_sigemptyset(sigset_t *set);
rtl_sigfillset(sigset_t *set);

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
/* Set the process's signal blockage mask */
int sigprocmask(int how, const rtl_sigset_t *set, rtl_sigset_t *oset);
int pthread_sigmask(int how, const rtl_sigset_t *set, rtl_sigset_t *oset);
/* Wait for a signal to arrive, setting the given mask */
int sigsuspend(const rtl_sigset_t *sigmask);
int sigpending(rtl_sigset_t *set);
/* Send a signall to a thread */
int pthread_kill(pthread_t thread, int sig);
```

Dealing with signals

There are three ways in which you can deal with signal:

- The thread can block the signal for a while.
- The thread can ignore the signal, in which case it is as if the signal never arrives.

  To ignore a signal (the behavior is similar to as it never would receive it) by installing the SIG_IGN handler. This handler , is essentially a null handler, a handler unction that just returns, having done nothing at all.
- The thread can handle the signal, by setting up a function to be called whenever a signal with a particular number (SIGUSR1) arrives.

Signal handlers and ignoring signals

The sigaction is used to set all the details of what your process should do when a signal arrives. The struct sigaction encapsulates the action to be taken on receipt of a particular signal. struct sigaction has the following form (element order may vary and additional members could be added):

```
struct rtl_sigaction {
  union {
        void (*_sa_handler)(int);
        void (*_sa_sigaction)(int, struct rtl_siginfo *, void *);
       } _u;
  int sa_flags;
  unsigned long sa_focus;
  rtl_sigset_t sa_mask;
};
```

The most important member is sa_handler, which takes a pointer to a function. This

function will be invoked whenever the process gets a particular POSIX.1 signal. The signal handler function is declared like this:

void handler_for_SIGUSR1(int signum);

Design guidelines

Unfortunately, POSIX doesn't says much about signals and threads. When POSIX talks about signals it has in mind the concept of process. So, there are a lot of things for which the designer is responsible.

Signal names and numbers.

Signal's names and numbers has been very influenced by the implementation. In RTLinux each thread has an integer (32 bits in i386 arch) for pending signals and one for blocked signals. RTLinux scheduler uses 6 of this bits for thread management with signals (RTL_SIGNAL_SUSPEND, RTL_SIGNAL_READY, RTL_SIGNAL_TIMER, ..). Also includes the Linux header for POSIX signals (#include <linux/signal.h>).

Taking these things in count, user's signals comes from bit 7 to the number of bits used by an architecture to represent an integer. In i386 twenty five user's signals has been added, coming from the 7 (RTL_SIGRTMIN) to the 31 (RTL_SIGRTMAX). POSIX standard names has been changed by adding the prefix RTL_ so it doesn't conflicts with Linux signals names.

The following signals names has been defined:

```
#define RTL_SIGUSR1  (RTL_SIGNAL_READY + 1)
#define RTL_SIGUSR2  (RTL_SIGUSR1 + 1)
#define RTL_SIGRTMIN (RTL_SIGUSR1 +1)
#define RTL_SIGRTMAX  RTL_MAX_SIGNAL
```

Signal generation and delivery.

A signal is said to be ``generated'' for (or sent to) a thread when the event that causes the signal first occurs. Examples of such events include timer expiration, as well as invocations of the pthread_kill(). So in RTLinux signals are generated immediately, by marking the corresponding bit of the thread's pending mask. A signal is said to be ``delivered'' when the appropriate action for the thread and signal is taken. A thread doesn't gets the signal until it takes the CPU.

Sending a signal implies to change the state of the thread to ready if that signal wasn't blocked. So, if a thread sends a signal via rtl_pthread_kill() to another thread with higher priority, the scheduler is called. At this moment the scheduling algorithm chooses from the set of threads with any pending signal that is not blocked, the one with highest priority. Next the target thread, takes the CPU if it is the highest priority thread. Then it executes the signal handler if it installed one different from the default handler. After returning from the signal handler the thread will resume its execution at the point it was interrumped. The RTLinux pthread_kill implementation doesn't calls the scheduler. It only marks that signal to the target thread as pending.

In our implementation the default handler doesn't kill the thread, it is just a null handler as SIG_IGN.

Well at this point you will want to know, what happens when a signal is generated when a thread it's executing a signal handler. As a design decision when a thread it's executing a handler this signal it's blocked. After finishing the handler that signal is unblocked. So, signal handlers are not re-entrant. During the execution of a signal handler following rules apply:

- Deliveration order is as follows: first greater signal numbers. If at a determinate moment, there are n pending signals that are not blocked, signal handlers will be executed serially.
- Signal handlers are non-reentrant, i.e. if while a thread is executing a signal handler, another signal arrives, first the scheduler terminates the execution of current handler and then the other pending signals are delivered.
- If during the delivering of a signal n1, another signal n2 arrives, following rules apply:
  - if n1 > n2, it is delivered after finishing current delivery for signal n1.
  - If n1 <= n2, it is delivered next time the thread will take the CPU.
- The signal being delivered, is blocked by the scheduler just before executing the signal handler, if there is one and its different from default or SIG_IGN. After executing the signal handler the signal is unblocked.
- Finally, signal handlers are executed always with interrupt enabled.

Implementation issues

The implementation of POSIX signals in RTLinux. Modified files:

For now, five files of the RTLinux version 3.2pre2 has been modified. Modifications are in-crusted with #ifdef __POSIX_SIGNALS__ :

/* Functions prototypes & constants definitions */
include/posix/signal.h
include/rtl_signal.h
include/rtl_sched.h
/* Signal generation, delivery & management. */
schedulers/rtl_sched.c
schedulers/rtl_sema.c
/* Functions implementations */
scheduler/signal.c

When & where execute the signals handlers

It is necessary to find an appropriate environment in which to call the signal handler Two approaches seem suitable for that purpose. The first by manipulating thread's stack (at the scheduler before rtl_switch_to is called) in the thread code. While the second consist in executing the signal handler in the scheduler after rtl_switch_to . The two approaches has been tested, being the second seemed more elegant, clean, secure and

architecture independent. The problem of executing signal handler in the scheduler is that there interrupts are disabled. One of the first things that the scheduler does is to disable interrupts. Finally, just before ending it restores them. Remember that we want to execute user signal handlers with the interrupts state of the receiver thread. The scheduler is called mainly from the timer interrupt handler with interrupts disabled after restore them. What it is done is to enable interrupts only to execute the signal handler and then disable them. When all user signals have been checked, interrupt state is restored. At that point, if current thread isn't ready (that is, has taken the CPU only to receive a user signal) the code jumps to the begin of the scheduler, to find a ready thread. So a thread remains in the state if was, just before receiving a user signal.

BUGS 12-11-02

Stack overflow when sending RTL_SIGNAL_SUSPEND signal:

The RTLinux scheduler for version 3.2pre2 is re-entrant when it wants to suspend a thread. The procedure is the following. When it wants to suspend a thread different from pthread_self() (with pthread_suspend_np()) it sends a signal to that thread (RTL_SIGNAL_SUSPEND). Then the scheduler decides to give the CPU for that thread so it has a pending signal. But the handler for that signal marks the thread as suspended and calls the scheduler again. If the process is repeated, as in the following example, what we are doing is to push calls to the scheduler in the threads stack. If we are suspended and nobody wakes up us (in a mutex, calling pthread_suspend_np,etc ..) the stack becomes exhausted in a finite time. In version 3.0 the scheduler wasn't reentrant and this error doesn't occur. A good test is to change the stack size and observe that then iterates proportionaly to the stack increment.We have the same problem with user signals handlers. The solution was to return to the scheduler policy of version 3.0, being non re-entrant.

```
/*
 * POSIX.1 Signals
 *
 * Written by J. Vidal
 * Copyright (C) Dec, 2002 OCERA Consortium.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2.
 *
 * Program showing stack overflow when sending RTL_SIGNAL_SUSPEND
 *
 */
```

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

#define NTASKS 2
#define MAXINT 0x7fffffff
pthread_t thread[NTASKS];

void * start_routine(void *arg)
{
        struct sched_param p;
        int ret=0,i=0,param=(unsigned) arg;
        long long period=10*1000*1000;

        p . sched_priority = NTASKS -param;
        pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

        if (param==0){
          pthread_make_periodic_np (pthread_self(), gethrtime(),period );

          while (i++<MAXINT) {
            pthread_wait_np ();
            pthread_suspend_np(thread[1]);
            // Test that thread 1 is alive.
            ret=pthread_kill(thread[1],0);
            if (ret==ESRCH){
            rtl_printf("STACK EXHAUSTED in iter:%dn",i);
        pthread_suspend_np(pthread_self());
             }
              else
              rtl_printf("iter:%dn",i);
          }else {
          pthread_make_periodic_np (pthread_self(), gethrtime(),period*2);
          while (1){
            pthread_wait_np();
          }

        rtl_printf("thread %d has exited from its loop\n",param);

        return 0;
}

int init_module(void) {
  int i,err=0;
  for (i=0;i<NTASKS;i++)
    err=pthread_create (&thread[i], NULL, start_routine, (int *) i);

  return err;
```

```
}

void cleanup_module(void) {
  int i;
  for (i=0;i<NTASKS;i++)
    pthread_delete_np (thread[i]);
}
```

Mutex bug when setting do->abort=0 on do_signal:

Mutex BUG 4-12-02:

Well, the fact is that do_signal when receives the signal RTL_SIGNAL_SUSPENDS sets

t->abort to zero. So, when it calls do_abort(t) it has no effect.

In our example, blocked thread was blocked on a mutex. Meantime mutex owner thread

is sending signal to it (pthread_wakeup_nppthread_suspend_np, pthread_wakeup_np).

What happens is the following:

1.- The blocked gets blocked on the mutex (calling rtl_wait_sleep on the

pthread_mutex_lock loop), and it is queued in the mutex wait queue.

2.- At this point, mutex owner sends the following signals:

2.1.- RTL_SIGNAL_WAKEUP: blocked thread takes the CPU to manage it. Then it calls to do_abort and rtl_wait_abort takes it out of the mutex wait queue. Then it loops again and calls rtl_wait_sleep and it is queued on mutex wait queue. This time the behaviour is the spected.

2.2.- Next blocked thread receives the signal RTL_SIGNAL_SUSPEND which sets

t->abort to zero, marks blocked thread suspended and calls the scheduler.

2.3.- Finally, blocked thread receives the RTL_SIGNAL_WAKEUP again. But this time when do_abort is called, it has no effect (so RTL_SIGNAL_SUSPEND set its to zero). So blocked thread isn't removed from mutex wait queue. But the code follows (rtl_wait_sleep returns) looping at pthread_mutex_lock loop, calling rtl_wait_sleep again. This function queues blocked thread in mutex wait queue which was queued allready (well, not exactly since the struct queued was local to rtl_wait_sleep, but this struct reamins when the stack is decremented). At this time mutex wait queue contains the following: head -> thread1 -> head.

And here is the bug we got a double linked wait queue where the head and the tail are pointing to the same waiter (blocked thread).

3.- When thread 0 calls pthread_mutex_unlock and runs the mutex wait queue to wake

up blocked threads it runs an infinite loop and the user lost the machine control (so linux never enters).

Proposed solution:

Possibly, setting do_abort to zero when managing RTL_SIGNAL_SUSPEND is for future compa\tibility or a simple mistake. Now abort field of thread's structure is only used for mutexes an\d semaphores. One solution is to not set to zero and execute do_abort when managing the RTL_SIG\NAL_SUPEND signal.

Example program:

```
/*
 * POSIX.1 Signals
 *
 * Written by J. Vidal
 * Copyright (C) Dec, 2002 OCERA Consortium.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2.
 *
 * Simple program showing a nasty mutex bug when setting
 * t->do_abort=0 on do_signal.
 *
 */

#include <rtl.h>
#include <rtl_mutex.h>
#include <rtl_sched.h>

static pthread_t mutex_owner, blocked_thread;
static pthread_mutex_t mut;

static void *mutex_owner_routine(void *arg)
{
```

```
  struct sched_param p;
  int err=0,i=0;

  p . sched_priority =1;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

  rtl_printf("MUTEX OWNER. Just before adquiring the mutex.\n");
  pthread_mutex_lock(&mut);
  rtl_printf(" \n\n MUTEX OWNER - Mutual exclusion code.\n");
  rtl_printf("MUTEX_OWNER - Testing mutexes --> sending signals
RTL_SIGNAL_SUSPEND & RTL_SIGNAL_WAKEUP\n\n");
  // wait for the other task to get blocked.
  usleep(1000);

  err=pthread_wakeup_np(blocked_thread);
  rtl_printf("pthread_wakeup_np(thread[blocked_thread]) returns %d\n",err);
  err=pthread_suspend_np(blocked_thread);
  rtl_printf("pthread_suspend_np(blocked_thread) returns %d\n",err);
  err=pthread_wakeup_np(blocked_thread);
  rtl_printf("pthread_wakeup_np(thread[blocked_thread]) returns %d\n",err);

  rtl_printf("MUTEX OWNER. BEFORE pthread_mutex_unlock \n");
  err=pthread_mutex_unlock(&mut);
  rtl_printf("MUTEX OWNER. pthread_mutex_unlock returned %d. errno
%d\n",err,errno);

  rtl_printf("MUTEX OWNER about to end \n");
  return (void *)err;
}

static void *blocked_thread_routine(void *arg)
{
  int err=0;
  struct sched_param p;
```

```
  p . sched_priority =1;
  pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);



  rtl_printf("BLOCKED THREAD. Just before adquiring the mutex.\n");
  pthread_mutex_lock(&mut);
  rtl_printf(" \n\n BLOCKED THREAD - Mutual exclusion code.\n");
  rtl_printf("BLOCKED THREAD. BEFORE pthread_mutex_unlock \n");
  err=pthread_mutex_unlock(&mut);
  rtl_printf("BLOCKED THREAD. pthread_mutex_unlock returned %d. errno
%d\n",err,errno);


  rtl_printf("BLOCKED THREAD about to end \n");
  return (void *)err;
}

int init_module(void) {
  pthread_attr_t attr;
  pthread_mutexattr_t mutattr;

  pthread_mutexattr_init(&mutattr);
  /*    pthread_mutexattr_settype(&mutattr,PTHREAD_MUTEX_SPINLOCK_NP); */

  pthread_mutex_init(&mut,&mutattr);

  pthread_attr_init(&attr);
  // Threads creation.
  pthread_create (&mutex_owner, &attr, mutex_owner_routine,(void *) 0);
  pthread_create (&blocked_thread, &attr, blocked_thread_routine,(void *) 0);

 return 0;
}
```

```
void cleanup_module(void) {
  void *retval;



    pthread_join (mutex_owner, &retval);
    rtl_printf("pthread_join on MUTEX OWNER returned %d\n", (int) retval);
    pthread_join (blocked_thread, &retval);
    rtl_printf("pthread_join on BLOCKED THREAD returned %d\n", (int) retval);



    //pthread_delete_np(thread[i]);


  pthread_mutex_destroy(&mut);

}
```

Appendix

Test programs

In the directory examples/signals, there is a nice collection of test programs. Each one testing some functionallity of signals. See the file README located for a detailed explanation of each one. To load any test program, type make test in the correspondent directory. To see, a brief description of each progream, type make help in the correspondent directory. In the following list, it can be found a brief description of each one:

hello.c:

This is a trivial program in which a periodic thread sends a signal to itself every time a condition is acomplished. The signal handler for that signal prints a message saying ``Hello world! Signal handler called for signal ..."

Things to test: Sending a signal to pthread_self(), programing an action for a signal, handler execution (observe that signal handler isn't executed until next scheduling event occurs for that thread).

signals.c:

This is a test program in which there is a master thread and a user defined number of slaves. The master thread is the only thread that is periodic. It sends a signal to each

slave in each activation. Slaves are suspended and waked up by it's signal handler. After reciving a number of signals, the slave thread blocks that signal with pthread_sigmask. After this no more signals are delivered.

Things to test: Sending signals to other threads, blocking signals with pthread_sigmask, execution of signal handlers despite of being suspended, waking up thread from a signal handlers (calling the scheduler from signal handlers).

ign_signals.c:

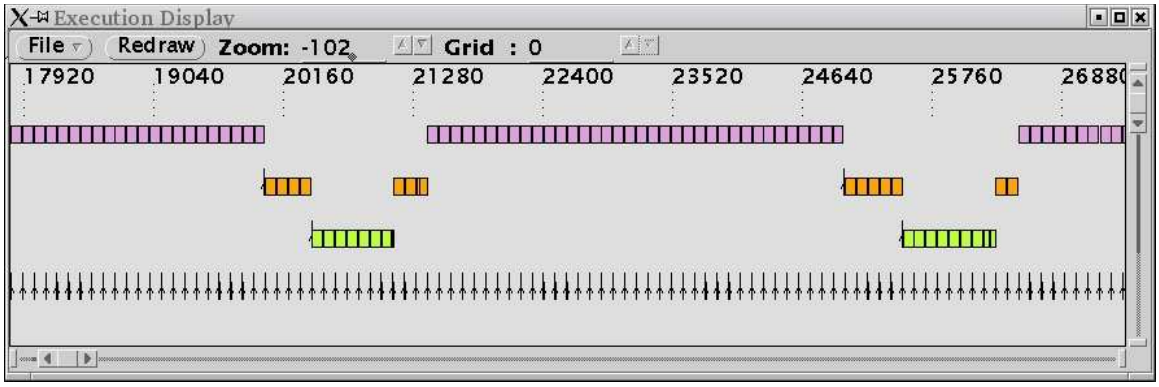
In this test a master sends signals to all its slaves. Only odd threads should get delivered generated signals. Since, the rest of threads have installed the default handler or SIG_IGN.

Things to test: Scheduler behavior with ignored signals and default handler (it should be as it never be delivered).

test.c:

This is a simple program to test that the execution of a signal handler can be interrumpted by other higher priority threads. In this test two periodic threads are wasting time when they execute its signal handler. Be careful with the amount of computing they do in its signal handlers. They can hang up slow processors. A hight priority thread expulses signal handler executions periodically every 1 milisecond. Also a higher priority signal handler expulses a lower. A chonogram of what its happening is shown in next figure. The first task is linux. The second and the third are periodic tasks that are executing huge quantity of computing on their signal handlers. Last task is a high frequency periodic task. Finally, priority is increassing with the number of the task. This is, the first task the less prioritary, the last the most.

Things to test: Execution of signal handlers with interrupts enabled, reciving interrupts inside of a signal handler (for example timer interrupt that calls the scheduler and then puts the highest priority task with pending signals), signal handler expulsion by a higher priority thread.



sig_intr.c:

This program shows what happens when a non periodic thread it's executing a handler and is interrumped by other hight priority thread.

Things to test: Finishing signal handler executions on non periodic threads after being preempted.

wait_4_sig.c:

In this program, we prove most of implemented functionalities: pthread_sigmask, sigsuspend, pthread_kill, terminating threads from signal handlers, blocked mask ... The logic of the program consists in a master that is sending a signal window of size three to a slave. Slave blocks all signals each time except the one that they want to receive for that moment. Only the handler for the signal that is unblocked will be executed. The other signals generated don't have effect so they are blocked.

Things to test: Programming various actions from the same signal (last that succesfully installs the handler is the owner, sigsuspend, mask of blocked signals, pthread_sigmask, removing a thread from a signal handler.

sched.c:

This program implements a very simple version of the round robin scheduler using a periodic thread and user signals. Each scheduled thread has three signal handlers: the first suspends that thread while the second wake ups it. Finally the third finishes it. The periodic thread has a handler that implements the scheduling algorithm. Each time it has to schedule sends a signal to itself. Then the handler for that signal sends a signal to suspend the current thread and other to wake up next thread. Finally, when the scheduler finishes sends a signal to kill all the scheduled threads.

Things to test: Re-entrant functions. That is a function like (pthread_wakeup_np(pthread_self())) that calls the scheduler before ending. If signal being delivered won't be blocked, receiving thread will enter in a infinite loop. The reason is that the pending bit for that isn't removed until the signal handler terminates. So it will be ready, if someone interrumps signal handler execution, to recuperate the CPU and finish handler execution.

Also is tested using the RTLinux scheduler API from signal handlers, sigsuspend, pthread_sigmask.

sig_prio.c:

In this program a thread sends a signal to a higher priority thread. At this moment the higher priority thread should take the CPU. Then it suspends signal generator to prove that has taken the CPU.

Things to test: Real time requeriments in signal generation and delivery.

sig_nanosleep.c:

In this program a thread is sleeping for a while. While it is sleeping other thread generates a signal for it. Then the thread is interrumped and shows the sleep time remained.

Things to test: The behavior of nanosleep function when it's interrumped by a signal.

sig_sem.c:

In this program, various threads are blocked on a semaphore. While they are blocked other thread generates a signal to no odd semaphore blocked threads. Then no odd threads are interrumped and gets out the semaphore. The other threads remain blocked.

Things to test: The behavior of sem_wait function when it's interrumped by a signal.

Note: There is also, a test version for timed semaphores on file sig_timedsem.c

sig_mutex.c:

In this program, various threads are blocked on a mutex. While they are blocked on the mutex, the master threads sends signals to them. At this point blocked threads must execute the signal handler and remain blocked.

Things to test: The behavior of pthread_mutex_lock function when it's interrumped by a signal.

Note: There is also, a test version for timed mutexes on file sig_timedmutex.c.

sig_condvar.c:

In this program a real-time thread before becoming blocked on a condition variable sends a signal to itself. After becoming blocked it executes the signal handler for previous generated signal. After this, it resumes blocked on the condition variable.

pending.c:

Simple program to test signal delivery order and global sigactions.

Posixtestsuite:

In this directory you will find three test program testing sigaction functionality.

References.

Programming for the Real World -POSIX.4, Bill O. Gallmeister, 1995.

The Single UNIX® Specification, The Open Group, 1997

UNIX Programación Práctica. Kay A. Robbins, Steven Robbins. Prentice Hall.