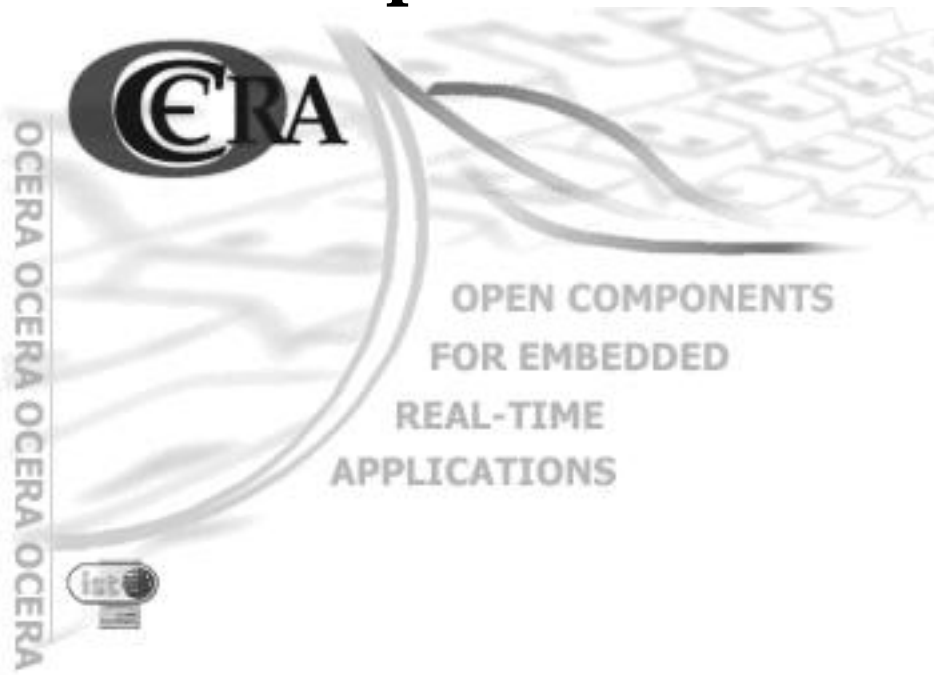
WP5 - Real-Time Scheduling Components



Deliverable D5.2_rep - Scheduling Components V1

WP5 - Real-Time Scheduling Components : Deliverable D5.2_rep - Scheduling Components V1

by Ismael Ripoll, Alfons Crespo, Patricia Balbastre, Jorge Real, Sergio Saez, Josep Vidal, and Miguel Masmano

Published January 2003

Table of Contents

List of Tables

List of Figures

Document presentation

Table 1. Project Co-ordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14. CP: 46022, Valencia, Spain
Phone:	+34 9877576
Fax:	+34 9877579
E-mail:	alfons@disca.upv.es

Table 2. Participant List

Role	Id.	Name	Acronym	Country
CO	1	Universidad Politécnica de Valencia	UPVLC	E
CR	2	Scuola Superiore S. Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA	CEA	FR
CR	5	UNICONTROLS	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	VISUAL TOOLS S.A.	VT	E

Chapter 1. Posix Barriers

1.1. Summary

Name
: Posix Barriers

Description
: A barrier is a simple and efficient synchronisation utility.

Author
: Patricia Balbastre.

Reviewer
: Ismael Ripoll

Layer
: High level RTLinux.

Version
: 0.1

Status
: Testing

Dependencies
: None. Tested for RTLinux-3.2-pre1.

Release Date
: M2

1.2. Descritpion

Barriers, are defined in the advanced real-time POSIX (IEEE Std 1003.1-2001), as part of the advanced real-time threads extensions. A barrier is a simple and efficient synchronisation utility. Threads using a barrier must wait at a specific point until all have finished before any of them can continue.

POSIX barriers are a relatively new feature and are not supported on all systems.

1.3. Layer

Posix Barriers are a high level RTLinux component, since it does not modify the RTLinux source code, but adds new features.

1.4. API / Compatibility

The API is defined by the POSIX standard. Following is the list of the functions that have been implemented.

```
/*Barrier attributes */
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *attr, int *pshared);
/* Barrier */
int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *attr, unsigned
int count);
int pthread_barrier_wait(pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

1.5. Dependencies

None.

1.6. Status

This is a testing version.

1.7. Implementation issues

Basically, the implementation will consist of two files (`rtl_barrier.c` and `rtl_barrier.h`) that implement the functions of the API. This component must not be implemented by means of a mutex, since this is an inefficient method.

To explain the implementation details of the barrier component, it will be useful to explain the steps to be made when a thread wants to use a barrier:

1. The barrier attributes are initialized.
   - This is accomplished trough the function `pthread_barrierattr_init`

2. The barrier is initialized, only once, by calling the function `pthread_barrier_init`.
   - This function sets the attributes of the barrier (specified in the previous step, or it takes a default attribute object) and the parameter *`count`*, which specifies the number of threads that are going to synchronise at the barrier. Although standard posix recommends that the value specified by *`count`* must be greater than zero, if *`count`* is 1, the barrier will not take effect, since no blocking would be produced. Therefore, in this implementation, a value of *`count`* less than or equal to 1 is not valid. Otherwise, EINVAL is returned.

3. When a thread wants to synchronise at the barrier, it calls the function `pthread_barrier_wait` .
   - When a thread calls this function, it must wait until all the rest of the threads have reached the same function call. If it is not the last thread to reach the barrier, the following steps have to be done:
     a. Suspend the thread that invokes the `pthread_barrier_wait` function.
     b. Add this thread to the list of waiting threads
        - These actions are the same that are made when a thread blocks in a semaphore, and are implemented in the function `rtl_wait_sleep`, defined in `rtl_mutex.h`. Proves have been done and the overhead of this function is minimal or, at least, is the same than implementing the same actions "by hand".
        - Another solution, it would be to make use of mutexes functions, that is, use `pthread_mutex_lock instead` instead of `rtl_wait_sleep`, but this is not recommended in the standard Posix, due to the introduced overhead.

   - If it is the last thread to reach the barrier, then all threads must be woke up. This is accomplished by means of the `rtl_wait_wakeup` function, also defined in rtl_mutex.h. This function send the signal `RTL_SIGNAL_WAKEUP` to all the threads blocked on the barrier, and resets the waiting queue of threads.

4. Finally, both the barrier and the attributes have to be destroyed (`pthreadd_barrierattr_destroy` and pthread_barrier_destroy). If there are

threads waiting on the barrier, the function `pthread_barrier_destroy` must not destroy the barrier, but it has to exit with error EBUSY.

1.8. Tests and Validation criteria

1.8.1. Validation criteria

To validate the correct implementation of this component two types of tests have been developed:

- Conformance tests. These tests check the correct behavior of the API. For example, trying to wait for a barrier that has not been initialize, must send an error.
- Overhead tests. These tests must validate the following criteria (for a Pentium III 700 MHz):
  - Barriers latency < 5 microseconds.
  - Barrier latency + Scheduler latency < 20 microseconds.

As soon as the Open Posix Test Suite releases a test set for barriers, it will be used to validate the implementation. However, for now barriers are not planned to be included in new versions of the Suite.

The target system for the tests was an AMD K6 300 MHz without APIC.

1.8.2. Tests

1.8.2.1. Conformance tests

1.8.2.1.1. Test 1

This test tries to use a barrier without being initialized. In order to do this, 4 threads are created in `init_module` function. When the threads call to the `pthread_barrier_wait` function it must return the error EINVAL.

1.8.2.1.2. Test 2

This test tries calls to `pthread_barrier_init` when there are still threads blocked on the barrier.

Four threads are created, that executes de following code:

```
void * fun(void *arg) {

    int id = (int)arg;

    if (pthread_barrier_init(&my_barrier, &barrier_attr, 5)!=0){
 rtl_printf("OK: Test passed\n");
      }
      else {
 rtl_printf("Test failed in task %d\n", id);
    }
    pthread_barrier_wait(&my_barrier);
    ...
    pthread_exit(0);
    return (void *)0;
}
```

The first thread that executes its code can successfully initialize the barrier again, but the next threads would obtain an EBUSY error.

1.8.2.1.3. Test 3

In this test a thread wants to destroy the barrier when other threads are still blocked on it. This way, when 4 threads have been already created and are waiting in the barrier to thread 5, this thread 5 destroys the barrier instead of executing `pthread_barrier_wait`. If the test is passed, then thread 5 must obtain a EBUSY error, when calling `pthread_barrier_destroy`.

1.8.2.2. Overhead test

The test implemented measures the overhead introduced by the barrier. In the test, two threads are created, and the first instruction is to wait in the barrier. The next instruction is a calling to the `gethrtime()` function to measure how much time it costs to wake up all threads in `pthread_barrier_wait` function.

This overhead (including context switch) is 18-25 ms in a AMD K6 300 MHz without APIC.

1.9. Examples

1.9.1. How to run the examples

In order to test the correct behavior of Posix Barriers, some tests will be provided with the component. To run these examples, just type make start in the same directory of the source code. To stop the examples, type make stop.

1.9.1.1. Example 1

This is na very simple example to show how a barrier works. The goal is to create two threads, that execute some delay (different for each thread) and block on the same barrier. The two threads only continue execution until the `init_module` executes the `pthread_barrier_wait` function. Thus, the barrier have to be initialized with a count of 3. The code of one of the two threads is following:

```
void *thread1 (void *not_used) {
    hrtime_t  now1;

    now1 = gethrtime();
    rtl_printf ("thread1 starting at %lld\n", now1);
    rtl_delay(20000000);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    now1 = gethrtime();
    rtl_printf ("barrier in thread1() done at %lld\n", now1);
    return (void *)0;
}
```

The same code is used for thread2 and init_module function, but with different delays.

1.9.1.2. Example 2

This example shows how threads can use a barrier more than once. There are two periodic threads that must print in the console the phrase Hello World. Thread1 must print the word "Hello" and thread2 the word "World". Obviously, thread1 must wait thread2 once it has print its word, and viceversa. This can be implemented by means of a barrier. Once a thread has execute its code, it blocks on a barrier until the other thread has reached the same point. Then, it waits for the next period. The code for both threads:

```
void *thread1 (void *not_used){
  pthread_make_periodic_np(pthread_self(), gethrtime(), 100000);
  while(1) {
    rtl_printf("Hello ");
    pthread_barrier_wait (&barrier1);
    pthread_wait_np();
```

```
  }
  pthread_exit(0);
  return (void *)0;
}

void *thread2 (void *not_used)
{
  pthread_make_periodic_np(pthread_self(), gethrtime(), 100000);
  while(1) {
    pthread_barrier_wait (&barrier1);
    rtl_printf("World\n");
    pthread_wait_np();
  }
  pthread_exit(0);
  return (void *)0;
}
```

1.9.1.3. Example 3

A barrier can be used to force periodic threads to execute its first activation at the first time. In real-time scheduling theory this is called the critical instant, that is, when all threads want to execute at the same time in its first activation. Therefore, barriers will be very useful to implement this synchronisation. The test, in this case, will consist of one barrier. All threads block on the barrier before making periodic. When the last thread arrives to the barrier, then all threads are allowed to continue execution. The code of the periodic threads is as follows:

```
void * fun(void *arg) {

    pthread_barrier_wait(&my_barrier);
    pthread_make_periodic_np(pthread_self(), now, period);

    while (1){
       //periodic code
       pthread_wait_np();
    }

    pthread_exit(0);
    return (void *)0;
}
```

1.10. Installation instructions

In order to install the component, please follow next steps:

- Edit the file 'Makefile' in pbarriers directory and set your rtlinux source directory, for example:

  "RTLINUX = /usr/src/rtlinux-3.2-pre1"

- Type: **make install**

This will install the component, copying the source code, documentation and the examples.

Chapter 2. POSIX.1 Signals

2.1. Summary

Name

POSIX.1 Signals

Description

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 and interprocess communication.

Author/s

Josep Vidal Canet (jvidal@disca.upv.es)

Reviewer

Ismael Ripoll

Layer

Low-Level component.

Version

0.2

Status

Finished. Included in RTLinux-3.2pre2 release.

Dependencies

RTLinux-3.2pre1. Also tested/available for RTLinux-3.1, RTLinux-3.2pre2.

Release Date

M2

2.2. Descritpion

This section describes the main characteristics of Posix Signals.

2.2.1. What is a signal?

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

Because a thread can send a signal to another thread, signals can be used for interprocess communication. Signals are not always the best interprocess communication mechanism; they are limited and can asynchronously interrupt a thread in ways that require clumsy coding to deal with. Signals are mostly used for other purposes, like the timer expiration and asynchronous I/O completion.

There are legitimate reasons for using signals to communicate between processes. First, signals are frequently used in UNIX systems. Another reason is that signals offer an advantage that other communication mechanisms do no support: signals are asynchronous. That is, a signal can be delivered to a thread while the thread is doing something else. The advantages of asynchrony is the immediacy of the notification and the concurrence.

2.2.2. Dealing with signals

There are three ways in which a thread can manage a signal:

- The thread can block the signal for a while.

- The thread can ignore the signal. In that case, the result is that the signal never arrives. This can be done by installing the SIG_IGN handler. This handler is, basically, a null handler.
- The thread can handle the signal, by setting up a function to be called whenever a signal with a particular number (SIGUSR1) arrives.

2.2.3. Signals handlers and ignoring signals

The `sigaction` is used to set all the details of what a process should do when a signal arrives. The struct `sigaction` encapsulates the actions to be done when receiving a particular signal. Struct `sigaction` has the following fields (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);
```

2.2.4. POSIX.1 Signals

POSIX define two types of signals: basic signals and real-time signals extensions. This component provides only the basic signal mechanisms. The real-time POSIX signals extension is far more complex (they are closer to message queues than to interrupts) and require complex data structures.

POSIX signals were designed to be used in weight-processes systems, where each process has its own signal handlers, signal mask and status. But in RTLinux, as well as most embedded RTOS, the programming model is based on lightweight processes (threads). The standard is not as clear and unambiguous as it should be. We had to extend the semantics of the signals API to threads.

RTLinux-3.1 code already had partial signal support, some system signals are supported but user signals, and the facility to define signal handlers, are not supported. This component adds the signal support to be fully UNIX compatible.

2.3. API / Compatibility

Following is the list of supported signal functions facilities:

```
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);

/* Programing actions for signals ocurrences*/
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);
```

2.4. Implementation issues

POSIX standard do not provide a clear description of how signals should be handled in a multi-threading environment. We tried our best to provide an implementation, as close as possible, to the standard. Two main issues have to be considered when implementing POSIX.1 signals:

2.4.1. Signal generation and delivery

A signal is said to be "generated" for (or sent to) a thread when the event that causes the signal occurs. Examples of such events include timer expiration, and invocations of the `pthread_kill()` function. In RTLinux, signals are generated immediately by setting 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 executes the signal handler when it is the highest priority active thread. Sending a non-blocked user signal implies to change the state of the thread to ready. When a thread sends a signal via `pthread_kill()` to another thread with higher priority, the scheduler is called immediately, and a context switch is produced to run the signal handler in the context of the high-priority thread. After returning from the signal handler, the thread will continue its execution at the point where it was interrupted. Our implementation differs from the POSIX standard in which the default handler does not kill the thread, but it is just a null handler as `SIG_IGN`. Signals are delivered sequentially, that is, when a thread is executing a signal handler no other signal is delivered until the signal handler function returns. Finally, signal handler functions are executed always with interrupts enabled.

2.4.2. When and where execute the signals handlers

In a conventional OS, the kernel is executed in its own processor mode and memory space. While the kernel is running, all the processes are stopped and the state of all the processes is stored in the PCB (Process Control Block). RTLinux does not work like a conventional OS but like a set of library functions. Concurrency is implemented as coroutines. There is no "context switch" from user code to kernel code, but it is just a function call. With this OS structure, there are two possible ways of implementing the signal delivery: The first one is by manipulating thread's stack (at the scheduler code before `rtl_switch_to()` is called) in the thread code. And the second way is to call the signal handler explicitly from the scheduler function after `rtl_switch_to()`. Both approaches have been tested, and the second one has been selected because it is cleaner, more secure, architecture independent and faster.

2.4.3. Modified source code

For now, five files of the RTLinux version 3.2pre1 have been modified. Modifications are toggled between `#ifdef CONFIG_OC_PSIGNALS`:

```
/* 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
```

2.4.4. Found bugs

During the POSIX.1 signals implementation, the following RTLinux bugs were found (and solved):

- Stack overflow when sending RTL_SIGNAL_SUSPEND signal.
- Infinite loop after calling `pthread_mutex_unlock()` (due to setting thread's structure aborts field to zero in `do_signal()`)
- long long casting in `usleep` function.

2.5. Tests and validation

A total of twenty different tests have been implemented to check functionality. The tests are located in directory examples/signals. See README file for a detailed description. Among other issues, the following features have been tested:

- POSIX.1 Signals API: Sending a signal to threads, programing actions for signal, signal handler execution, waiting for a signal, blocking signals, etc.
- Execution of signal handlers despite of being suspended, waking up thread from a signal handler (calling the scheduler from a signal handler).
- Signal handler preemption by higher priority threads.
- Finishing signal handler executions on non-periodic threads after being preempted. Programming various actions from different threads, to the same signal.
- Executing RTLinux scheduler API from signal handlers.
- Real-time requeriments in signal generation and delivery.
- The behavior of some functions (`nanosleep()`, `sem_wait()`,...) after being interrumped by a signal.

2.5.1. Validation criteria

This component increases RTLinux POSIX compatibility, since signals are required in all POSIX/UNIX systems. It also reduces the cost of porting applications to RTLinux. POSIX functionality relies strongly on the signals. For example, the POSIX timers can not be implemented on a system with no signal support.

2.5.2. Tests

2.5.2.1. Test 1

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 ..."

Tested aspects: Sending a signal to `pthread_self()`, programing actions for a signal arrivals (observe that signal handler isn't executed until next scheduling event occurs for that thread).

2.5.2.2. Test 2

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 its signal handler. After receiving a number of signals, the slave thread blocks that signal with `pthread_sigmask`. After this, no more signals will be delivered to that thread.

Tested aspects: Sending signals to other threads, blocking signals with `pthread_sigmask`, execution of signal handlers despite of being suspended, waking up threads from signal handlers (calling the scheduler from signal handlers).

2.5.2.3. Test 3

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.

Tested aspects: Scheduler behavior with ignored signals and default signal handler (it should be as it never be delivered).

2.5.2.4. Test 4

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 executing its signal handler. Special care has to be taken with the amount of computation time of the threads in the signal handler, because high computation time can hang up slow processors. A higher priority thread preempts signal handler executions periodically every 1 milisecond. Also, a higher priority signal handler preempts a lower one. A chronogram of what is happening is shown in Figure 2-1. The first task is linux. The second and the third are periodic tasks that are executing huge quantity of computatio time on their signal handlers. Last task is a high frequency periodic task. Finally, priority is increasing with the number of the task. This is, the first task the less priority one.

Tested aspects: Execution of signal handlers with interrupts enabled, receiving 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.

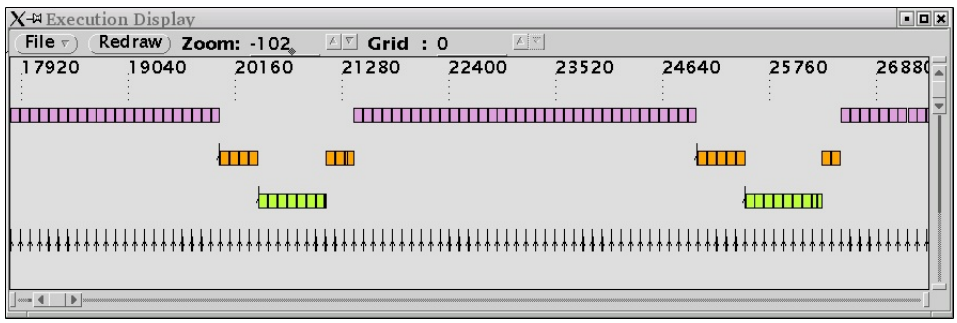


Figure 2-1. Chronogram execution for test 4

2.5.2.5. Test 5

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

Tested aspects: Finishing signal handler executions on non periodic threads after being preempted.

2.5.2.6. Test 6

In this program, almost all of the functionalities implemented are used: `pthread_sigmask`, `sigsuspend`, `pthread_kill`, terminating threads from signal handlers, blocked mask ... The test consists of a master that is sending a signal window of size three to a slave. The slave blocks all signals each time, except the one that wants to receive in 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.

Tested aspects: Programming several actions from the same signal (the last that succesfully installs the handler is the owner), `sigsuspend`, mask of blocked signals, `pthread_sigmask`, removing a thread from a signal handler.

2.5.2.7. Test 7

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 the thread while the second wakes it up. Finally, the third finishes it. The periodic thread has a handler that implements the scheduling algorithm. Each time it schedules, it sends a signal to itself. Then, the handler for that signal sends a signal to suspend the current thread, and other to wake up the next thread. Finally, when the scheduler finishes, it sends a signal to kill all the scheduled threads.

Tested aspects: Re-entrant functions. This is a function like `pthread_wakeup_np(pthread_self())`, that calls the scheduler before ending. If the signal delivered is not blocked, the receiving thread enters 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 a signal handler execution, to recover the CPU and finish the handler execution. Also, it is tested using RTLinux scheduler API from signal handlers, `sigsuspend` function, `pthread_sigmask` function.

2.5.2.8. Test 8

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 the signal generator to check that really it has taken the CPU.

Tested asptects: Real-time requeriments in signal generation and delivery.

2.5.2.9. Test 9

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.

Tested aspects: The behavior of `nanosleep` function when it is interrumped by a signal.

2.5.2.10. Test 10

In this program, various threads are blocked on a semaphore. While they are blocked, other thread generates a signal to even semaphore blocked threads. Then, even threads are interrumped and gets out the semaphore. The other threads remain blocked.

Tested aspects: 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`.

2.5.2.11. Test 11

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

Tested aspects: 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`.

2.5.2.12. Test 12

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.

2.5.2.13. Test 13

Simple program to test signal delivery order and global sigactions.

2.5.2.14. Test 14

This is a porting of some sigaction tests from the open POSIX test suite. In this directory you will find three test programs testing sigaction functionalites.

2.5.2.15. Test 15

This is a test program showing some incompatibilities when calling some functions from signals handlers. The incompatibilities come with functions performing non-active waits on signal handlers: clock_nanosleep, usleep, ...

2.5.3. Results and comments

Signals are considered to be a slow ipc (interprocess communication mechanism). Nevertheless RTLinux POSIX.1 signals implementation offers a high-performance ipc due to RTLinux characteristics and the little overhead of the implementation. One of the previous tests showed that two threads sending each other signals can send (and handle) 112 signals per millisecond (in a PIII 500MHZ with APIC).

2.6. Examples

2.6.1. How to run the examples

In order to run any example, the user must type **make test** in the example directory.

2.6.2. Description

In the directory examples, it can be found two examples showing signals basic functionality. The first one implements the client/server computing model between two threads. While in the second a thread kills other thread to avoid it to make expensive calculations.

2.7. Installation instructions

This patch has been proved to work with both 2.2.19 and 2.4.18 kernels. It is supposed to work on any kernel that is supported by RTLinux, since it only modifies scheduler related files non architecture deppendant.

In order to install POSIX signals in RTLinux, please follow next steps:

- Configure Makefile variables. To do this, edit Makefile and set RTLINUX variable to your RTLinux copy path.
- Type **make install**
- Compile new patched version, selecting the POSIX signal option, (by default disabled):

  make clean; make xconfig; make

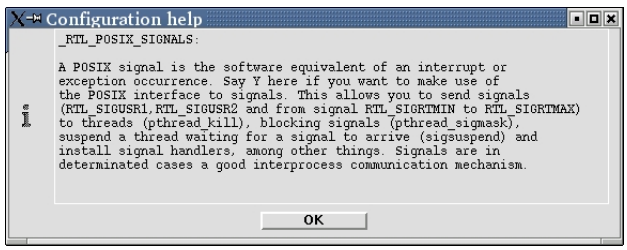


Figure 2-2. Configuration help

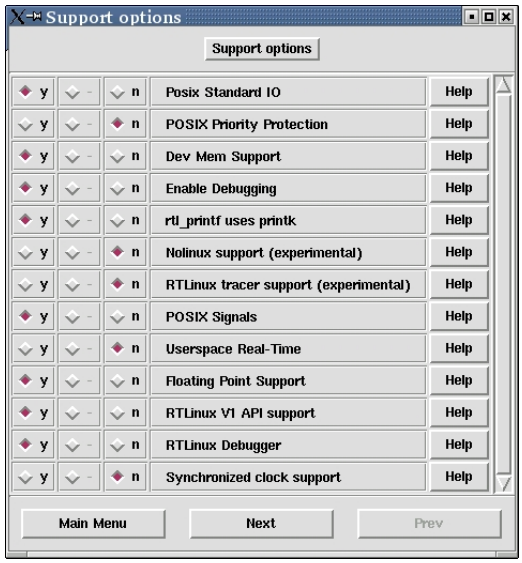


Figure 2-3. Configuration menu

Notes: Please read this carefully if you are appliying first the RT-Linux patch for linux-kernel Version 2.4.18. If that is the case, this patch modifies the file `schedulers/rtl_sched.c`. When you are applying the POSIX interface to signals patch, the patcher will ask you:

```
Reversed (or previously applied) patch detected!  Assume -R? [n] n
```

And you should answere NOT. Then, the patcher will ask for applying anyway, and you should answere YES.

```
Apply anyway? [n] y
```

Chapter 3. POSIX Timers

3.1. Summary

Name
: POSIX Timers

Description
: POSIX timers provides mechanisms to notify a thread when the time (measured by a particular clock) has reached a specified value, or when a specified amount of time has passed. Although RTLinux has good and accurate timing facilities, it do not provides general timer functionality. RTLinux defines only one timer for each thread, which is used to implement the periodic behaviour of the thread. This component implements the POSIX real-time extensions.

Author
: Josep Vidal Canet (jvidal@disca.upv.es)

Reviewer
: Ismael Ripoll Ripoll

Layer
: Low-Level component.

Version
: 0.2

Status
: Finished. Included in RTLinux-3.2pre2 release.

Dependencies
: Psignals component and RTLinux-3.2pre1. Also tested/available for RTLinux-3.1, RTLinux-3.2pre2.

Release Date
: M2

3.2. Descritpion

POSIX timers allows a mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed.

Facilities supported by POSIX timers that are desirable for real-time operating systems:

- Support for additional clocks.
- Allowance for greater time resolution (modern timers are capable of nanosecond resolution; the hardware should support it)
- Ability to use something other than SIGALARM to indicate timer expiration (in particular, a POSIX.4 real-time extended signal would be nice)

Therefore POSIX timers allows greater time resolution, implementation-defined timers, and more flexibility in signal delivery.

3.2.1. Creating a timer

Here is a simple and portable way of creating a timer:

```
#include <signal.h>
#include <time.h>
#define A_DESCRIPTIVE_NAME 13
int err;
struct sigevent signal_specification;
timer_t created_timer; /* Contains the ID of the created timer */
```

```
/* What signal should be generated when this timer expires ? */
signal_specification.sigev_signo= RTL_SIGUSR1;
signal_specification.sigev_value.sival_int = A_DESCRIPTIVE_NAME
err=timer_create(CLOCK_REALTIME, &signal_specification, &created_timer);
```

The code snipped creates a timer based upon the system clock called `CLOCK_REALTIME`. `CLOCK_REALTIME` exists on all POSIX.4-conformant systems, so it can be used. A machine may define other clocks, corresponding perhaps to extra, dedicated hardware resources on a particular target machine. The POSIX.4 conformance statement should indicate what clocks are available on a particular system.

The *evp* argument, if non-NULL, points to a sigevent structure. This structure, allocated by the application, defines the asynchronous notification to occur as specified in Signal Generation and Delivery when the timer expires. If the *evp* argument is NULL, the effect is as if the *evp* argument pointed to a `sigevent` structure with the `sigev_notify` member having the value `SIGEV_SIGNAL`, the `sigev_signo` having a default signal number, and the `sigev_value` member having the value of the timer ID. If you want to specify a particular signal to be delivered on timer expirations, use the struct `sigevent`, as defined in `signal.h`:

```
struct sigevent {
int sigev_notify; /*notification mechanism */
int sigev_signo; /*signal number */
union sigval sigev_value; /* signal data value */
}
```

This structure contains three members: `sigev_notify` is a flag value that specifies what sort of notification should be used upon timer expiration (signals, nothing, or something else). Currently, only two values are defined for `sigev_notify`: SIGEV_SIGNAL means to send the signal described by the remainder of the struct `sigevent`, and SIVEV_NONE means to send no notification at all upon timer expiration.

The third parameter is where the system stored th ID of the created timer. You will need this ID in order to use the timer.

3.2.2. Arming a timer

Once there is a timer ID, it can be set as in the following example:

```
#include <time.h>
struct itimerspec new_setting, old_setting;
new_setting.it_value.tv_sec=1;
new_setting.it_value.tv_nsec=0;
new_setting.it_interval.tv_sec=0;
new_setting.it_interval.tv_nsec=100*1000;
err=timer_settime(created_timer, 0, &new_setting, &old_setting);
```

This example sets the interval timer to expire in 1 second, and every 100.000 nanoseconds thereafter. The old timer setting is returned in the structure `old_setting`. With the second parameter, you tell the system to interpret the interval timer setting as an absolute (TIMER_ABSTIME) or as a relative setting, like in the above example.

Two timer types are required for a system to support realtime applications:

- One-shot: A one-shot timer is a timer that is armed with an initial expiration time, either relative to the current time or at an absolute time (based on some timing base, such as time in seconds and nanoseconds since the Epoch). The timer expires once and then is disarmed. With the specified facilities, this is accomplished by setting the `it_value` member of the *value* argument to the desired expiration time and the it_interval member to zero.

- Periodic: A periodic timer is a timer that is armed with an initial expiration time, again either relative or absolute, and a repetition interval. When the initial expiration occurs, the timer is reloaded with the repetition interval and continues counting. With the specified facilities, this is accomplished by setting the `it_value` member of the value argument to the desired initial expiration time and the `it_interval` member to the desired repetition interval.

For both of these types of timers, the time of the initial timer expiration can be specified in two ways:

- Relative (to the current time).
- Absolute.

3.2.3. Specification structures

Many of the timing facility functions accept or return time value specifications. A time value structure timespec specifies a single time value and includes at least the following members:

Table 3-1. Timespec structure

Member type	Member name	Description
time_t	tv_sec	Seconds
long	tv_nsec	Nanoseconds

The `tv_nsec` member is only valid if greater than or equal to zero, and less than the number of nanoseconds in a second (1000 million). The time interval described by this structure is (tv_sec * 10^9 + tv_nsec) nanoseconds.

A time value structure `itimerspec` specifies an initial timer value and a repetition interval for use by the per-process timer functions. This structure includes at least the following members:

Table 3-2. Timespec structure

Member type	Member name	Description
struct itimerspec	it_interval	Timer period
struct timespec	it_value	Timer expiration

If the value described by `it_value` is non-zero, it indicates the time to or time of the next timer expiration (for relative and absolute timer values, respectively). If the value described by it_value is zero, the timer shall be disarmed.

If the value described by `it_interval` is non-zero, it specifies an interval which shall be used in reloading the timer when it expires; that is, a periodic timer is specified. If the value described by `it_interval` is zero, the timer is disarmed after its next expiration; that is, a one-shot timer is specified.

3.3. API/Compatibility

This component provides all the functionalities described by the POSIX standard, but the one related to real-time signals, because the signals component does not provide it.

```
/* Creating a timer. Timer created is returned in timer_id location */
int timer_create(clockid_t clockid, struct sigevent *restrict evp, timer_t *restrict
timer_id);
/* Removing timer referenced by timer_id */
int timer_delete(timer_t *timer_id);
/* Setting timer referenced by location timer_id */
int  timer_settime(timer_t timer_id, int flags, const struct itimerspec *new_setting, struct
itimerspec *old_setting);
/* Getting time remainning until next expiration*/
int timer_gettime(timer_t timer_id, struct itimerspec *expires);
int timer_getoverrun(timer_t timer_id);
```

Timers implementation supports both CLOCK_MONOTONIC and CLOCK_REALTIME.

3.4. Implementation issues

In RTLinux timer_t type is implemented as a pointer to the timer structure. When a timer is created, the memory required to store the timer_struct is dynamically allocated. For this reason, timer_create() can only be called while in Linux space, that is, all timers must be created in the init_module(). For the same reason, timers can only be deleted in cleaun_module(). This implementation follows the general style of RTLinux used in mutex, semaphores, threads, etc; all data is preallocated before the threads are started. Timers are stored in a linked list sorted by thread owner priority, which speeds-up the code that finds the next timer to expire.

Right now, following files of the RTLinux version 3.2pre1 has been modified/added. Modifications are in-crusted with

```
#ifdef CONFIG_OC_PTIMERS
```

:

In schedulers directory:

```
rtl_sched.c, rtl_timer.c.
```

In include directory:

```
rtl_sched.h, rtl_timer.h, include/rtl_time.h, include/posix/time.h
```

Timers implementation supports both CLOCK_MONOTONIC and CLOCK_REALTIME.

3.5. Tests and validation

We have used several test sets to validate the component. Next a brief description of each test suit is provided:

- Self built test suite: Among other things these programs checks:
  - Timers resolution for both absolute and relative specs.
  - Timers emulation of multitasking.
  - POSIX timers API.
  - Timers effects over RTLinux scheduler API functions (pthread_make_periodic_np(), pthread_wait_np()).

- POSIX Test Suite:
  - Recently the Open POSIX Test Suite released a test suite with timers coverage. We have used these tests slightly modified to run on RTLinux. These tests are divided into four directories. Each one corresponding with the functionality to test (timer_create(),timer_delete(), timer_settime(), timer_gettime()).

- High Resolution Timers Linux implementation test suite.
  - Slightly modified to run on RTLinux.

3.5.1. Validation criteria

All tests have been passed (internal tests and independent external ones).

As in the case of signals component, increases RTLinux POSIX compatibility and reduces the cost of porting applications to Rtlinux. Allows to implement watchdog timers.

The timers overhead is negligible when no timer is armed. When several timers are armed, the overhead introduced is O(n) where n is the number of armed timers. Due to the flexibility and changing scenarios (priority inheritance, scheduler operational modes, different scheduling policies, etc.) it is not possible to use advanced data structures to achieve better worst case overhead. It is possible to use some heuristics to improve the response time in some cases, but the worst case remains the same.

3.5.2. Tests

3.5.2.1. Self built tests

3.5.2.1.1. Test 1

This test measures timer accuracy for both relative and absolute timer specifications. To do it, follows next procedure: Create a thread and program a periodic timer with some period (relative spec) or a one-shot timer for absolute specification. Then let expire the timer n times. If it is absolute, reprogram each time the timer by adding the period to the absolute time specification. After n expirations, calculate the teoric time transcurred (n*period) and the real (end_time -start_time). Finally, print the error as difference between the teoric time transcurred and the real one.

3.5.2.1.2. Test 2

This test measures POSIX.1 Signals bandwidth, following next procedure: Create two threads, a parent and a child. During an interval of time send signals between them. Measure signals send/received per milisecond.

3.5.2.1.3. Test 3

Simple program to test timers API, with the following procedure: Create a user-defined number of tasks. Make periodic them, with a period of 2 minutes. Arm timers for each task. The firs tasks arms a one-shot timer and the others repeating timers with an interval of 1 second. Install a handler for each timer that wakes up each task every timer expiration. Check that task period changes (for all tasks but the first) from 2 minutes to 1 second due to timers expirations and handlers executions.

3.5.2.1.4. Test 4

This program test RTLinux POSIX.4 timers emulation of multitasking. Provides support for testing both CLOCK_REALTIME & CLOCK_MONOTONIC. Also for system clock on mode ONESHOT or PERIODIC deppending on the defines.

Test follows next procedure: Create a user-defined number of tasks. Set the system clock on mode ONESHOT or PERIODIC. Schedule them, using RTLinux API or timers & signals. Analyse test chronograms. NOTE: You will observe that the scheduler result using the RTLinux API and timers + signals, is basically the same. Also you will observe that test chronogram for timers based on CLOCK_MONOTONIC and system clock on mode periodic is not synchronous. This is OK, since in RTLinux CLOCK_MONOTONIC with system clock on mode periodic is different from CLOCK_REALTIME (the one used by the scheduler and test chronogram). The monitor directory comes with a patched version of the scheduler that informs about tasks activations and executions times. Also provides a program (reader) to get kernel information. You need to place the program crono in that directory in order to see the chronograms (available from http://bernia.disca.upv.es/rtportal).

3.5.2.1.5. Test 5

Check that a timer can be programed by different threads. Test follows next procedure: Create two tasks. The first time both arm a timer. Only the last is the owner and therefore the one that will receive notification of timer expiraition. The handler for that timer wakes up the other task, allowing it to program an action for the signal of the timer and set the timer. The proces is repeated printing the time passed since last timer expiration. Each time the timer is being programmed by a different thread.

3.5.2.2. Posix Test Suite

The POSIX Test Suite is an open source test suite with the goal of performing conformance, functional, and stress testing of the IEEE 1003.1-2001 System Interfaces specification in a manner that is agnostic to any given implementation.

Among other POSIX functionalities, these suite supports timers testing. We have used these tests slightly modified to run on RTLinux. Timers tests are divided into four directories. Each one corresponding with the functionality to test (timer_create(),timer_delete(), timer_settime(), timer_gettime()).

Every test is well documented and ends with a report of assertions passed and failed.

3.5.2.3. High resolution timers test suite

This is the timers test suite provided by high resolution timers Linux implementation. The tests consists in two programs that test in a hard way timers functionality and implementation stability. More than 40 assertions are make against timers implementation.

3.5.3. Results and comments

We try to do our best on achieving good timers accuracy and emmulation of multitasking. One of the previous commented test timers resolution measures timers accuracy for both absolute and relative specifications. Test results vary depending on available hardware. In a PIII 500 MHZ with APIC, the resolution available is 10 microseconds for absolutes timers and 20 microseconds for relative timers. On a K6 II 3D NOW 300 MHZ without APIC the error is doubled (22 us absolute timers & 39 us relative timers). This results have been taken with CLOCK_REALTIME and system clock on mode ONE-SHOT (default mode).

Using timers for multitasking emmulation was checked in one of the previous tests. From the test results, showed in next two chronograms it can observed that there is no difference between using RTLinux API (pthread_make_periodic_np, pthread_wait_np) and usign timers and signals to run periodic tasks.

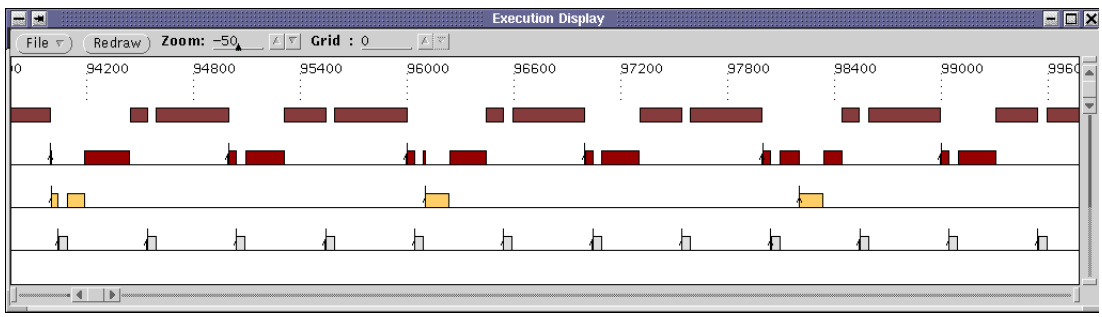


Figure 3-1. Chronogram execution using RTLinux API

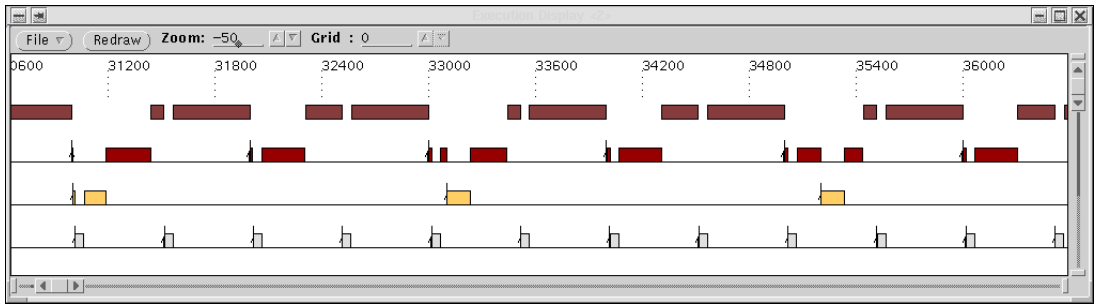


Figure 3-2. Chronogram execution using timers & signals

3.6. Examples

3.6.1. How to run the examples

In order to run any example, t he user must type make test in the example directory.

3.6.2. Description

In the examples directory, two simple programs showing basic timers functionality are provided. The first one, implements periodic threads using signals and timers, while the second uses a timers to avoid machine hang up due to huge quantity of recursive calculations.

3.7. Installation instructions

In order to install POSIX timers in RTLinux, please follow next steps:

- Install POSIX signals component.
- Install POSIX timers component. To do this, edit Makefile, set RTLINUX variable to your RTLinux copy path and type **make install**.
- Change to the RTLinux directory and type make xconfig. Next enable both posix signals and timers and type **make clean ; make**.
- Note: You can't select POSIX timers, if POSIX signals isn't selected. This is OK, since POSIX timers depends on POSIX signals implementation.

Chapter 4. POSIX Trace

4.1. Summary

Name
: POSIX Trace (**PTRACE**)

Description
: This component adds (most of) the tracing support defined in the POSIX Trace standard to RTLinux. The POSIX Trace standard defines a set of portable interfaces for tracing applications.

Author/s
: Andres Terrasa, Ana Garcia-Fornes, Agustin Espinosa.

Reviewer
:

Layer
: Low level RTLinux and Linux

Version
: 1.0

Status
: Stable

Dependencies
: RTLinux 3.2-pre1

Release Date
: M2

4.2. Description

As realtime applications become bigger and more complex, the availability of event tracing mechanisms becomes more important in order to perform debugging and run-time monitoring. Recently, IEEE has incorporated tracing to the facilities defined by the POSIX® standard. The result is called the POSIX Trace standard.

Tracing can be defined as the combination of two activities: the generation of tracing information by a running process, and the collection of this information in order to be analysed. The tracing facility plays an important role in the OCERA architecture. Besides its primary use as a debugging and tuning tool, the tracing component jointly with the application-defined scheduler component constitute the key tools for building fault-tolerance mechanisms.

The POSIX trace standard was firstly approved as the amendment 1003.1q of the POSIX 1003.1-1996 standard, and then integrated in the most recent version of POSIX, called 1003.1-2001. Considering that the Trace standard is quite recent, the reader may not be familiar with its concepts and terminology. The rest of this section is devoted to detail the main definitions of this standard, separating them in two subsections. The first subsection describes the main data structures by which applications can trace events (the trace event and the trace stream). The second subsection explains the three roles in which the standard separates the procedures to be performed in a tracing activity: the trace controller process (the process that controls the tracing), the traced or target process (the process being traced) and the trace analyzer process (the process retrieving and analysing the events).

4.2.1. Main Data Types Defined in the POSIX Trace Standard

When a program needs to be traced, it has to generate some information each time it reaches a significant step (certain instruction in the program's source code). In the POSIX Trace standard terminology, this step is called a trace point, and the tracing information which is generated at that point is called a trace event. A program containing one or more of this trace points is named instrumented application.

A trace event is thus defined as a data object representing an action which is executed by either a running process or by the operating system. In this sense, there are two classes of trace events: user trace events, which are explicitly generated by an instrumented application, and system trace events, which are generated by the operating system (which can be related to an action executed by the operating system in the process' behalf or due to an internal action not related to any particular process).

Any trace event, being either system or user, belongs to a certain trace event type (an internal identifier) and it is associated with a trace event name (a human-readable string). For system events, the definition of event types and the mapping between these types and their corresponding names is hard-coded in the implementation of the trace system. Therefore, this event types are common for all the instrumented applications and never change (they are always traced). The trace standard predefines some event types, which are related to the trace system itself, and permits the operating system designer to add some others which may be interesting to that system. The definition of user event types is very different. When an instrumented application wants to generate trace event of a particular type, it has first to create this type. This is done by invoking a particular function (`posix_trace_open()`) that, given a new trace event name, returns a new trace event type; then, events of this type can be generated from that moment on. If the event name was already registered for that application, then the previously associated identifier is returned. The mapping between user event types and their names is private to each instrumented program and lasts while the program is running.

The generation of a trace event is done internally by the trace system for a system event and explicitly (by the application when invoking `posix_trace_event()`) for a user trace event. In both cases, the standard defines that the trace system has to store some information for each trace event being generated, including, at least, the following:

1. the trace event type identifier,
2. a timestamp,
3. the process identifier of the traced process (if the event is process-dependent),
4. the thread identifier (of the thread related to the event), if the event is process-dependent and the O.S. supports threads,
5. the program address at which the event was generated,
6. any extra data that the system or the instrumented application wants to associate with the event, along with the data.

When the system or an application trace an event, all the information related to it has to be stored somewhere before it can be retrieved, in order to be analyzed. This place is called a trace stream. Formally speaking, a trace stream is defined as a non-persistent, internal (opaque) data object containing a sequence of trace events plus some internal information to interpret those trace events. The standard does not define a stream as a persisten object and thus it is assumed to be volatile, that is, to reside in main memory.

The standard establishes that, before any event can be stored for a process, a trace stream has to be explicitly created to trace that particular 'target' process (the process' pid is one of the arguments of the stream creation function). In the most general case, the relationship between streams and processes is many to many. On the one hand, many processes can be traced in a single stream; in particular, this happens if the target process forks after a stream has been created for the (parent) process. On the other hand, the standard permits that many streams are created to trace the same target process; if

so, each event generated by the process (or by the operating system) has to be registered in all these streams.

Streams also support filtering. The application can define and apply a filter to a trace stream. Basically, the filter establishes which event types the stream is accepting (and hence storing) and which are not. Therefore, trace events corresponding to types which are now filtered out from a certain stream will not be stored in the stream when traced. Each stream in the system (even if associated with the same process) can potentially be applied a different filter. This filter can be applied, modified or removed at any time.

The standard defines two classes of trace streams: active and pre-recorded, which are now described:

1. Active trace stream. This is a stream that has been created for tracing events and has not yet been shut down. This means that it is now accepting events to store. An active trace stream can be of two different types, depending on whether it has been created with or without a log.

In a trace stream with log, the stream is created along with a log. A log is a persistent object (that is, a file) in which the events stored in the stream are saved each time the stream is flushed (either by the trace system or by the application). In either case, the flushing then frees the resources previously occupied by the events just written to the log, making these resources available for new events to be stored. This is shown in Figure 4-1 below. In streams with a log, events are never directly retrieved from the stream but from the log (see 'Pre-recorded trace stream' below), once the stream has been shut down. That is, the log is not available for retrieving the events until the tracing of events is over; this leads to an off-line analysis of events.

In a trace stream without log, trace events are never written to any persistent media, but instead they remain in the stream (in memory) until they are explicitly retrieved. Thus, the stream is accessed concurrently for storing and retrieving events. These accesses can only be performed while the stream is active (that is, before it is shut down) since, after that, all the stream resources are freed. Therefore, an active trace stream without a log is used for on-line analysis of events, as shown in Figure 1.

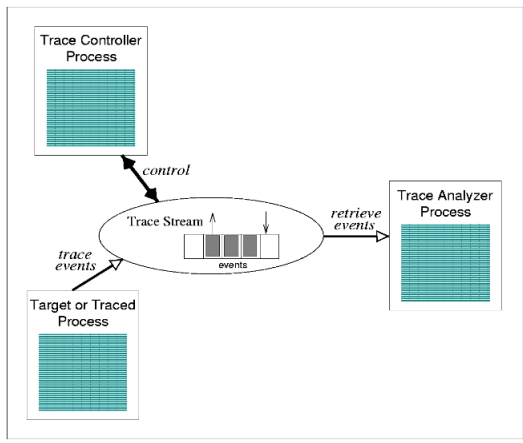


Figure 4-1. On-line tracing of events

2. Pre-recorded trace stream. A stream of this class is used for retrieving trace events which were previously stored in a log. In particular, the log file is opened into a (pre-recorded) stream from which events are then retrieved. Thus, off-line analysis of events is performed in two steps: first, events are traced into an active stream with log; second, after this stream is shut down, the log can be opened into a pre-recorded stream from which the events are retrieved. This tracing procedure is shown in Figure 2.

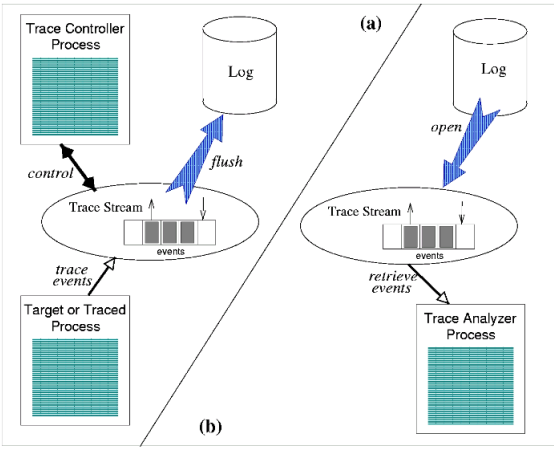


Figure 4-2. Off-line tracing of events

4.2.2. Roles Defined by the POSIX Trace Standard

Each tracing activity can only start when the trace controller process sets the tracing system up in order to trace a (target) process, which can be the same process or a different one. In particular, the trace controller process is in charge of, at least, the following actions:

1. Creating a trace stream with its particular attributes (e.g, if the stream is with or without a log, the stream full policy, etc.). This is done in two steps: first, a stream attribute object has to be created and then modified with the desired attributes for the stream. Then, a stream can be created with these attributes. As one of the parameters of the stream creation function is the process identifier (PID) of the target process, this implies that this target process has to exist in advance of the stream creation.
2. Starting and stopping tracing when necessary. Each active stream can be in two different states: running or suspended. These two states determine whether or not the stream is accepting events to be stored. The trace controller process can start and stop the stream as many times as it wants. If the stream full policy is to trace until the stream becomes full, then the trace system will automatically stop the stream when full and will start it again when some (or all) of its stored events have been retrieved.
3. Filtering the types of events to be traced. Each stream is initially created with an empty filter (that is, without filtering any event type). If this is not the required behavior, the trace controller process can build a set of event types, include the appropriate event types in it, and then apply it as a filter to the stream. While this filter is not removed or changed, the stream will not store any event whose type is in the filter set.
4. Shutting the stream down, when the tracing is over. The standard requires that shutting a stream down must free all the stream resources. That is, the stream is destroyed and no more operations can be done on it.

Optionally, the trace controller process can also perform other actions on the stream, once the stream has been created:

1. Clearing the stream. This erases all the events that are now in the stream, but leaves its behavior (attributes) intact. Clearing the stream makes it exactly in the same state that it was just after being created.
2. Flushing the stream (only for streams with a log). This produces an automatic flushing of all the events which are now in the stream to the log.
3. Querying the stream attributes and the stream current status.
4. Retrieving the list of event types defined for the stream.

5. Mapping event names to event types. This is normally performed by the target process in order to create its own user event types. However, the trace controller process can use the mapping function in the opposite way: given a well-known user trace event name, the mapping function will return the event type identifier; then, the trace controller process can use that identifier to set up a stream filter, for example.

The traced or target process is the process that is being traced, that is, is the process for which a trace stream has been created and set up. According to the standard, only two functions can actually be called from a target process:

1. A function to register a new user event type for this process. The input argument of this function is the (new) event type name. If this name has already being registered for that target, then the previously mapped event type identifier is returned. If not, then a new identifier is internally associated with this name and returned. If successful, this registration is valid for all the streams that have been created, or will be created, to trace the target process (even if no stream has still been created for that target).

From the user viewpoint, therefore, the identification of user event types is done in a per-name basis (instead of using integer values, for example). This allows for a name space wide enough to avoid collisions when independent pieces of instrumented code are linked together into a single application. This include, for example, the case of linking an instrumented third-party library to our code, even when we do not have the library's source code.

2. A function to trace an event. This function has three input arguments: the event type, which must have been previously registered (see above), a pointer to any extra data that has to be stored along with the event, and the size of this data. The event is stored in all the streams created for that particular target which are currently running and which do not have the event's type being filtered out.

It is important to point out that neither of these functions accepts a stream identifier as a parameter. That is, according to the standard philosophy, the target is programmed to invoke these functions without being aware (and independently) of actually being traced or not.

Finally, the trace analyzer process is in charge of retrieving the stored events in order to analyze them. The standard defines three alternative retrieval functions to be used by the trace analyzer process:

1. Blocking function. This function retrieves one event from the stream whose identifier is provided as a parameter. If no event is immediately available, the function blocks the invoking process (or thread) until an event is available.
2. Timeout function. This function works in a similar fashion than the previous one, but, when no event is immediately available, it blocks the process until either an event is available or an absolute timeout is reached (whatever of both happens first). If the timeout is produced first, the invoking process gets the corresponding error code.
3. Non-blocking function. This function never blocks the invoking process: when invoked, it immediately returns either a retrieved event or an error code, if no event is available at the moment.

If successful, any of these functions retrieve the oldest event stored in the stream which has not still been reported. The age of each event is calculated according to the automatic timestamp performed by the trace system when the event was recorded.

As explained above, the events can be only be retrieved from two different places: (1) from an active stream without log; (2) from the log of a (previously destroyed) stream with log, once this log has been opened into a (pre-recorded) trace stream. This defines the two kinds of analysis that the standard supports:

1. On-line analysis. In this kind of analysis, the trace analyzer process retrieves the events from an active trace stream (without log). As stated above, the retrieval func-

tion (any of them) needs to provide the stream's identifier; however, according to the standard, this identifier can only be used within the process that created the stream. This forces that, in an on-line analysis, the trace analyzer process and the trace controller process have to be the same one.

2. Off-line analysis. As explained above, this analysis is done in two steps: in the first step, events are recorded into an active trace stream with log that, automatically or under request of the trace controller process, flushes these events to the log (file). Once this step is over, the trace analyzer process opens the log into a private, pre-recorded stream, from which it can start retrieving the events. Only the first of the three retrieval functions mentioned above can actually be used in a pre-recorded stream. Obviously, in this case, this function will never make the trace analyzer process to block, since all the events are already stored in the stream.

In addition, the trace analyzer process can also retrieve other information of the stream (either active or pre-recorded), including the list of registered event types and its names, the stream attribute object (and then each of its individual attributes), the stream current status (for an active stream), etc. All this information is intended to make the trace analyzer process able to correctly interpret the trace events which it is retrieving.

4.3. API / Compatibility

The POSIX Trace standard splits the tracing support to be provided by the operating system in four different subsets or levels of support (named implementation options). Each of these levels can be optionally implemented by the system depending on the particular trace functionality that the system wants to support. Normally, each implementation option makes the trace system to support a certain subset of the tracing data types, constants, functions, etc.; occasionally, it also changes the semantics of some other functions supported by other options. In particular, the standard establishes four different implementation options, which are now briefly discussed:

1. Trace option. This option correspond to the most basic level of trace support. It is not really optional, since system conforming with trace standard must provide, at least, all the system facilities included in this option. In particular, this option requires only tracing into streams without a log with no filtering of events, and it only permits one process to be traced per trace stream. In other words, this option only supports on-line tracing of single target processes without event filtering.
2. Trace Log option. This option adds the possibility of creating trace streams with log to the Trace option, hence allowing for off-line analysis.
3. Trace Inheritance option. This option adds the possibility of tracing multiple target processes in one stream to the Trace option. Having that this option is supported, the tracing of multiple target processes in the same stream can only occur in one scenario: if a target process forks into one or more child processes and the stream on which the (parent) target process is being traced was created to support inheritance. If so, all these processes will concurrently being traced in the same stream. Otherwise, children of a target process are not traced.
4. Trace Event Filter option. This option adds the possibility of filtering events to the Trace option. As explained above, the trace controller process can apply a set of event types as a filter on a particular stream, making any event whose type is in the set to be discarded of that stream when traced. Filtering of events is specially useful in on-line analysis, in order to permit the trace analyzer process to retrieve all the important events without loss. In general, filtering of not relevant events is always a good idea in order to prevent the tracing system to process an overwhelming number of events.

The implementation described in this document, that is, the POSIX Trace (ptrace) component, supports the Trace and the Trace Event Filter implementation options as defined above, subject to some minor changes and limitations. Overall, these options allows

the programmer of a realtime application in RTLinux to perform filtered on-line tracing of events at run time. Since RTLinux concurrency is limited to lightweight processes, this implementation cannot support the Trace Inheritance option. Also, the Trace Log option has not been implemented due to the lack of permanent storage subsystem.

As explained in Section 4.2.2, *Roles Defined by the POSIX Trace Standard*, the POSIX Trace standard separates each tracing activity in three roles: the trace controller process (TCP), the target process (TP), and the trace analyzer process (TAP). Accordingly, the tracing API proposed in the standard is divided in three groups, corresponding to the these roles. The subset of functions in each group that is currently supported by the implementation is now described.

In this implementation, the TCP role is always played by the RTLinux application, normally at the initialization stage. The following is the full list of the functions corresponding to the TCP which are supported:

```
int posix_trace_attr_destroy(trace_attr_t *);
int posix_trace_attr_getclockres(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getcreatetime(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getgenversion(const trace_attr_t *, char *);
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict, size_t *restrict);
int posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *restrict, size_t *restrict);
int posix_trace_attr_getmaxusereventsize(const trace_attr_t *restrict, size_t, size_t *restrict);
int posix_trace_attr_getname(const trace_attr_t *, char *);
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict, int *restrict);
int posix_trace_attr_getstreamsize(const trace_attr_t *restrict, size_t *restrict);
int posix_trace_attr_init(trace_attr_t *);
int posix_trace_attr_setmaxdatasize(trace_attr_t *, size_t);
int posix_trace_attr_setname(trace_attr_t *, const char *);
int posix_trace_attr_setstreamsize(trace_attr_t *, size_t);
int posix_trace_attr_setstreamfullpolicy(trace_attr_t *, int);
int posix_trace_clear(trace_id_t);
int posix_trace_create(pid_t, const trace_attr_t *restrict, trace_id_t *restrict);
int posix_trace_eventid_equal(trace_id_t, trace_event_id_t, trace_event_id_t);
int posix_trace_eventid_get_name(trace_id_t, trace_event_id_t, char *);
int posix_trace_eventset_add(trace_event_id_t, trace_event_set_t *);
int posix_trace_eventset_del(trace_event_id_t, trace_event_set_t *);
int posix_trace_eventset_empty(trace_event_set_t *);
int posix_trace_eventset_fill(trace_event_set_t *, int);
int posix_trace_eventset_ismember(trace_event_id_t, const trace_event_set_t *restrict, int *restrict);
int posix_trace_eventtypelist_getnext_id(trace_id_t, trace_event_id_t *restrict, int *restrict);
int posix_trace_eventtypelist_rewind(trace_id_t);
int posix_trace_get_attr(trace_id_t, trace_attr_t *);
int posix_trace_get_filter(trace_id_t, trace_event_set_t *);
int posix_trace_get_status(trace_id_t, struct posix_trace_status_info *);
int posix_trace_set_filter(trace_id_t, const trace_event_set_t *, int);
int posix_trace_shutdown(trace_id_t);
int posix_trace_start(trace_id_t);
int posix_trace_stop(trace_id_t);
int posix_trace_trid_eventid_open(trace_id_t, const char *restrict, trace_event_id_t *restrict);
```

In this implementation, the target or traced process (TP) is always composed by a set of real-time tasks executed by the RT-Linux scheduler and, optionally, some Linux user processes. The ptrace component provides both levels with the two functions which the standard defines to this role. These functions are:

```
int posix_trace_eventid_open(const char *restrict, trace_event_id_t *restrict);
void posix_trace_event(trace_event_id_t, const void *restrict, size_t);
```

The ptrace component supports the TAP role to be played either by some real-time tasks inside the RTLinux application or by a Linux user process. In either case, the full list of functions available is:

```
int posix_trace_attr_getclockres(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getcreatetime(const trace_attr_t *, struct timespec *);
int posix_trace_attr_getgenversion(const trace_attr_t *, char *);
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict, size_t *restrict);
int posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *restrict, size_t *restrict);
int posix_trace_attr_getmaxusereventsize(const trace_attr_t *restrict, size_t, size_t *restrict);
int posix_trace_attr_getname(const trace_attr_t *, char *);
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict, int *restrict);
int posix_trace_attr_getstreamsize(const trace_attr_t *restrict, size_t *restrict);
int posix_trace_eventid_equal(trace_id_t, trace_event_id_t, trace_event_id_t);
int posix_trace_eventid_get_name(trace_id_t, trace_event_id_t, char *);
int posix_trace_eventtypelist_getnext_id(trace_id_t, trace_event_id_t *restrict, int *restrict);
```

```
int posix_trace_eventtypelist_rewind(trace_id_t);
int posix_trace_get_attr(trace_id_t, trace_attr_t *);
int posix_trace_get_status(trace_id_t, struct posix_trace_status_info *);
int posix_trace_getnext_event(trace_id_t, struct posix_trace_event_info *restrict ,
                        void *restrict, size_t, size_t *restrict, int *restrict);
int posix_trace_timedgetnext_event(trace_id_t, struct posix_trace_event_info *restrict,
                                    void *restrict, size_t, size_t *restrict, int *restrict,
                                    const struct timespec *restrict);
int posix_trace_trygetnext_event(trace_id_t, struct posix_trace_event_info *restrict,
                                  void *restrict, size_t, size_t *restrict, int *restrict);
```

4.4. Implementation issues

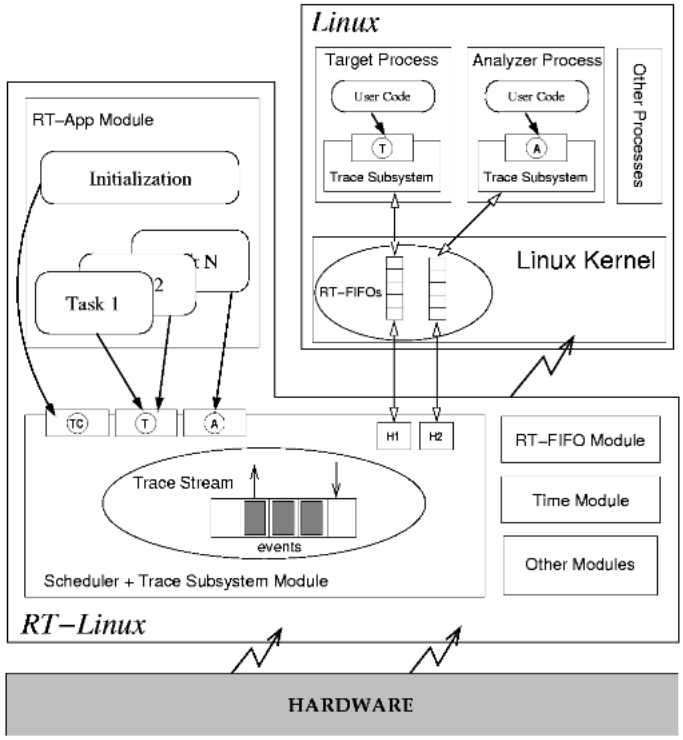
In a typical RTLinux application, the programmer usually splits the application code in a set of realtime tasks (executed by the RTLinux executive) plus one or more Linux user processes (executed by the Linux kernel). These two parts will be hereafter referred to as the application's RTLinux side and Linux side. If some trace support has to be given to the two sides, the ptrace system has to be present at both. Therefore, we have implemented ptrace as two cooperating trace subsystems, one at each level:

1. RTLinux trace subsystem. The ptrace support at the RTLinux level has been fully implemented in a new C file called `rtl_posixtrace.c`, which is compiled and then linked to the RT-Linux scheduler (module `rtl_scheduler.o`). The trace support is then always available to the RTLinux application, whether or not the application wants to use it. Nevertheless, its overhead in the case of not using it is practically null. In addition to this file and the corresponding header files (basically, `trace.h`), two minor changes have been made to other existing RTLinux source files: (1) the initialization and cleanup functions of the rtl_posixtrace module are invoked from the corresponding functions inside the `rtl_scheduler.o`, and (2) three source code files have been modified in order to generate system events which may be useful to trace (such as mutexes and condition variable manipulation, context switches between real-time tasks, etc.). The three source code files are: `rtl_sched.c`, `rtl_mutex.c` and `rtl_posix.c`. All other parts of RTLinux have been left intact.

On the other hand, all the data structures necessary to keep the entire tracing status are created and managed inside the rtl_posixtrace module. The two main data structures are: an array of trace_info_t structures, each one devoted to manage each of the eight possible trace streams available to the application, and an array of `event_type_info_t` structures, with each one representing an event type, either defined by the operating system (that is, a system event type) or by the application (that is, a user event type).

2. Linux trace subsystem. Maybe the most natural way to support POSIX Trace at the Linux level could have been to modify the Linux kernel by adding the required facilities as new Linux system calls. However, we chose not to do this for two reasons: firstly, because then the support would have been completely coupled to a particular version of the Linux kernel, and secondly because the actual functionality to be supported did not actually require such ambitious implementation. As a result, the decision was to implement this subsystem as a library (`libposix_trace.a`) to be linked with any Linux process that requires trace support. This library is made available to Linux programs when the ptrace component is compiled. Internally, this library communicates with the RTLinux scheduler (where the RTLinux trace subsystem is) in order to make both systems work in a synchronised manner. This communication is done by several dedicated RT-FIFOs.

The following figure presents an overview of the ptrace component:



.POSIX Trace component overview.

Figure 4-3. DIDMA data structure

In this figure, the subset of the API available to the TCP, TP and TAP is represented, respectively, by a circle with a 'TC', a 'T' , and a 'A' inside. 'H1' and 'H2' represent the RT-FIFO handlers used to communicate both tracing subsystems.

4.5. Tests and validation

4.5.1. Validation criteria

At the time of writing these lines, and as far as we know, this is the first implementation in C language of the POSIX trace standard for any POSIX operating system (that is, not only including real-time operating systems). This component thus adds a functionality that was not present in RTLinux before. In this sense, the main validation criterion was to use it exhaustively and to make the source code easy to understand by others (e.g., this component was successfully used during the development of the RTLGnat component).

Therefore, efficiency and speed, though considered throughout the implementation, was not the most important design issue. However, future versions of this component will probably incorporate major changes in this sense, making the code much faster and smaller.

Nevertheless, some tests were developed in order to establish the order of magnitude of the only four sensitive functions to be used by RTLinux applications, which are the function for tracing an event and the three alternative functions for retrieving an event (blocking, non-blocking and timeout). It can safely be considered that all other functions will only be invoked at the initialization stage or from the Linux side; in either case, their effect to the performance of the realtime application is not relevant.

4.5.2. Test 1

The objective of this test is to measure the execution time of the functions to trace and to retrieve an event. In order to perform this measurement, a buffer of memory was made

available to both the realtime scheduler and a Linux process, in order to minimize the overhead of the own measurement procedure. This procedure was straightforward: get the system time before and after invoking the function to be measured (and sometimes in the middle, as explained below). These times were stored in the buffer and, after the realtime application was finished, the Linux process could retrieve all of them in order to perform some simple statistical analysis.

All the tests were made on a PC computer with a 700 Mhz. Pentium-III processor, 512 kilobytes of cache memory and 128 megabytes of RAM.

4.5.3. Results and comments

The obtained results are summarized in the table below (all values in nanoseconds):

Measured Value	**Minimum**	**Maximum**	**Average**	**Variance**
Trace Event	352.0000	3094.0000	469.9319	47666.0583
Get Next Event (1)	96.0000	576.0000	137.3904	1468.1739
Get Next Event (2)	192.0000	3172.0000	235.2899	5378.6793

Although there are four functions to be measured (the function for tracing an event and the tree functions for retrieving an event at run-time), the table has grouped the results in only three values. The first value directly corresponds to the cost of the function for tracing an event. The other two values represent the costs of the two parts in which any of the three retrieval functions can be divided into: an initial part (labelled Get Next Event (1)) in which the function checks whether or not the stream is empty, and a final part (labelled Get Next Event (2)) in which an event is removed from the stream and returned to the caller. The final part may be executed right after the initial one (if there is at least one event in the stream when the function is called) or else it may be executed after a temporary suspension (blocking) of the calling task. This way of measuring is more useful than a single overhead value per function, since we can perform a finer analysis in which the overhead is included when it is actually produced. The measurements of each part in the table correspond to the three retrieval functions, since we found no significant differences among them.

The results presented in the table clearly show that the overhead of the tracing services actually used at run time is fairly low: the average values range from 100 to 500 nanoseconds approximately and the maximum measured values never exceed 4 microseconds. Moreover, the average and variance values also inform that these maximum values rarely occur.

Chapter 5. Application-defined Scheduler

5.1. Summary

Name

Application-defined Scheduler

Description

Application-defined scheduling is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way.

Author/s

Josep Vidal Canet (jvidal@disca.upv.es)

Reviewer

Ismael Ripoll

Layer

Low-Level component.

Version

0.2

Status

Finished.

Dependencies

Psignals and ptimers component. RTLinux-3.2pre1. Also tested/available for RTLinux-3.1.

Release Date

M2

5.2. Description

5.2.1. Introduction

Although research in scheduling theory has been one of the most active and fertile areas in real-time, most of the results had never been implemented in commercial RTOS. A notorious example of this situation is the EDF (Earliest Deadline First) scheduling policy; it is well known, with good theory background and in most cases provides better performance than fixed priority algorithms. The EDF is not implemented in any commercial RTOS. The commercial real-time market is very conservative, and new theory is only accepted when it has been tested and widely validated.

Previous to the Linux and RTLinux development, it was almost impossible to modify the core kernel of a commercial RTOS since it is one of the best guarded industrial secrets. RTLinux enjoys a special position in the RTOS area: it is accepted as a strong and valid RTOS that can be used to build carrier grade applications, and also it can be easily used by the academia researchers to include new functionality.

The application defined scheduler in RTLinux is a key facility which will help in the adoption of the already available scheduling theory. The ADS allows RTLinux to implement, in a very portable way, new scheduling algorithms that can be ported immediately to other RTOS.

Application-defined scheduler was designed by Mario Aldea Rivas and Michael González Harbour [Aldea02], and implemented in the MaRTE OS.

5.2.2. Application-defined scheduling overview

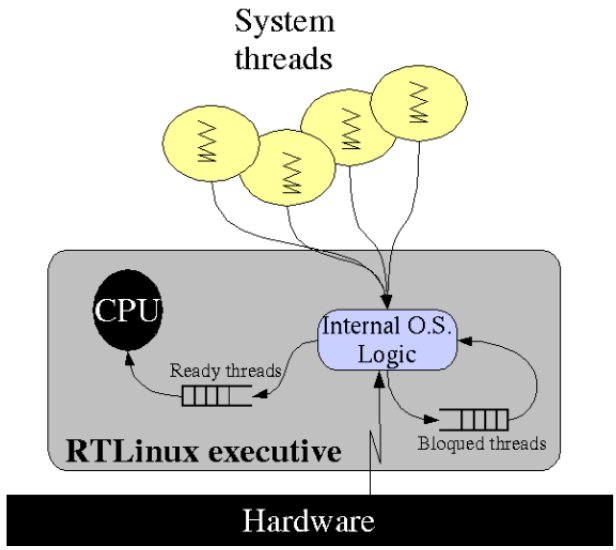


Figure 5-1. Conventional O.S. operation

Figure 5-1 shows a simplified schema of how the scheduling subsystem of a conventional operating system is internally designed. The internal scheduling algorithm (labeled as "Internal Logic") receives all the events that may change the state of the threads, and according to that information threads are dispatched to the processor or marked as blocked. The scheduling policy is the part of the scheduling subsystem that select among active thread the one that will be dispatched to the processor (CPU).

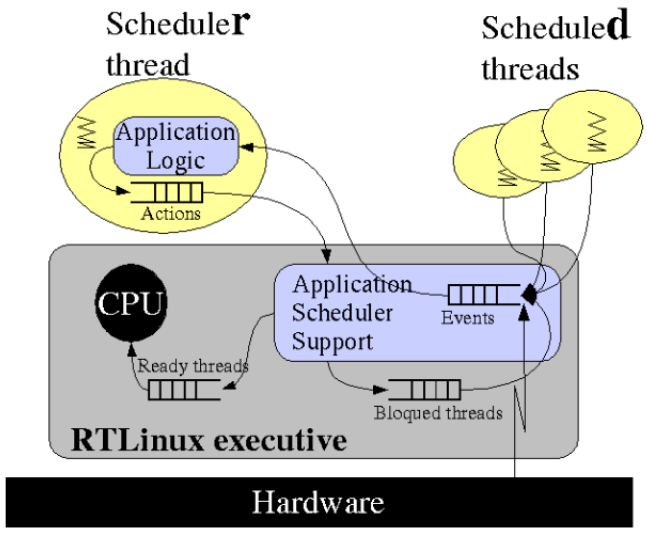


Figure 5-2. Application-defined scheduler O.S. operation

In a ADS system, the events related to scheduling decisions are not processed by the operating system but delivered to the **scheduler thread**. The scheduler thread then analyses the received information and sends back a list of actions to be performed by the operating system.

Figure 5-2 outlines the internal structure and the data flow. It is important to note that the ADS do not remove the original scheduling facility. In fact, there will be two dif-

ferent thread types depending on how are scheduled: system scheduled and application scheduled threads.

At a first glance, it could seem that moving the scheduling policy out of the RTOS's kernel, would result in a high overhead. Nevertheless, the clever design and cleanness of the interface proposed by M. González and M. Aldea, allows to implement almost any scheduling algorithm in an easy and efficient way.

5.2.3. Application-defined scheduling model

Figure 5-3 shows the model proposed for application-defined scheduling. In this model three different kinds of threads can be categorized:

- System-scheduled: Those threads scheduled by the RTLinux standard scheduler.
- Application-scheduled: Those scheduled by an application scheduler.
- Application-schedulers: Special kind of thread, that is responsible of scheduling a set of threads that have been attached to it.

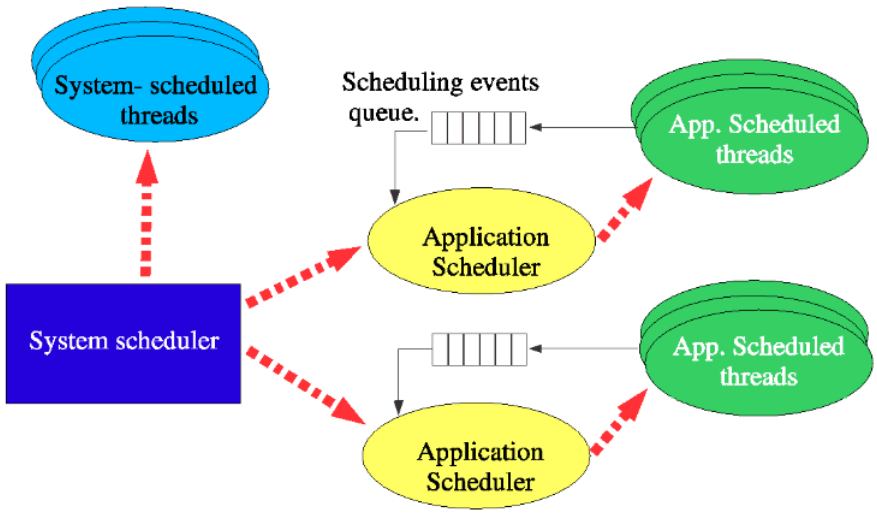


Figure 5-3. Application-defined scheduling model

5.2.4. Application-defined scheduling in practice

To show how application-scheduling works, an example of how a scheduled thread is managed by its scheduler thread will be detailed. To do this, we will show two common scheduling situations in real time schedulers. The first one, focuses in the moment in which a thread has finished its work for current activation and calls its application-scheduler to voluntary relinquish the CPU (Figure 5-4) . While the second, focuses in the moment in which an application-scheduled thread becomes ready, reaching its activation time. Before the application-scheduled thread begins its execution it must be activated by its application-scheduler (Figure 5-5) .

- Suppose that the scheduler thread is blocked at the `posix_appched_execute_actions()` waiting for the next scheduling event produced by any of its scheduled threads.
- A scheduled thread completes its current activation and suspends itself waiting for the next period. This it is done by calling `posix_appsched_invoke_scheduler()`.
- The RTOS kernel generate the associate event `POSIX_APPSCHED_EXPLICIT_CALL`, and queue this event in the scheduler thread events queue.

- The kernel wakes up the scheduler thread, which receives the event. Depending on the event type the scheduler thread adds one or more scheduling actions to the action queue. In this case it queues an action to suspend the thread that triggered the event.
- Once all the actions are programmed (inserted into the actions queue), the scheduler thread loops to the `posix_appched_execute_actions()` function to send the programmed actions and blocks to wait the next event.
- The RTOS receives and executes the requested scheduling actions. After this, the application-scheduled thread will remain blocked.

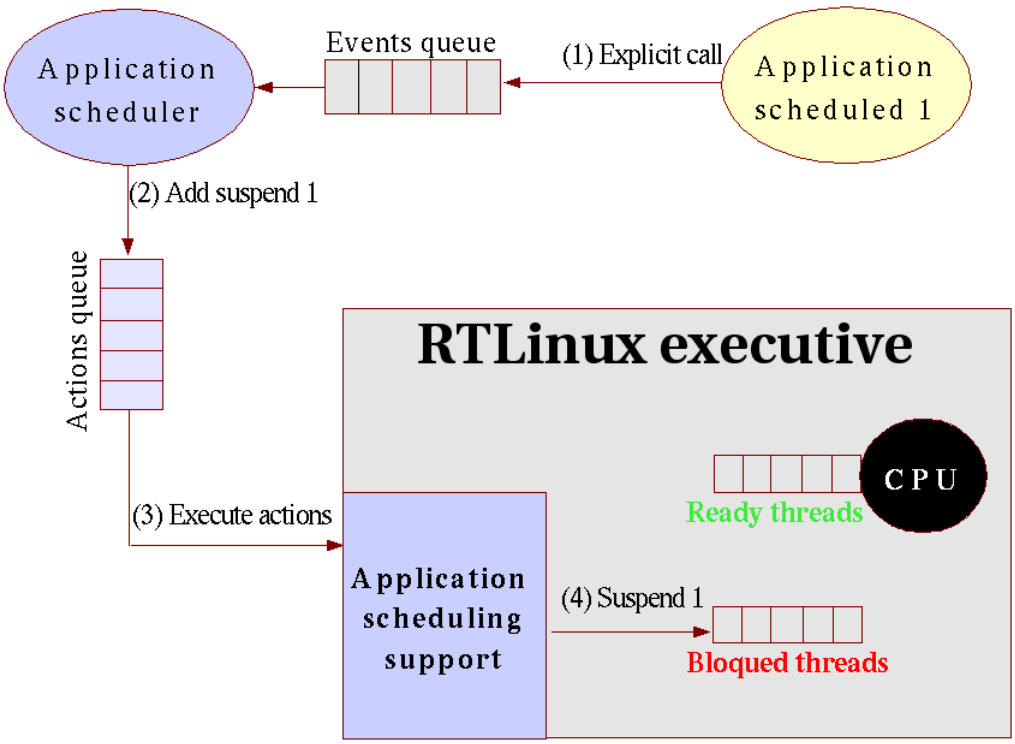


Figure 5-4. Application-defined scheduler operation: suspending a thread

- At this point the scheduler thread is blocked waiting for the next scheduling event produced by any of its scheduled threads.
- The application-scheduler timer expires and the RTOS generates the associate event `POSIX_APPSCHED_TIMEOUT`.
- The kernel wakes up the scheduler thread, which receives the event. Next the application-scheduler algorithm selects the application-scheduled thread who will be activated. Next it adds an action to activate the scheduled thread.
- Once all the action is programmed (inserted into the actions queue), the scheduler thread loops to the `posix_appched_execute_actions()` function to send the programmed actions and blocks to wait the next event.
- The RTOS receives and executes the requested scheduling action. After this, the application-scheduled thread will be ready for execution.

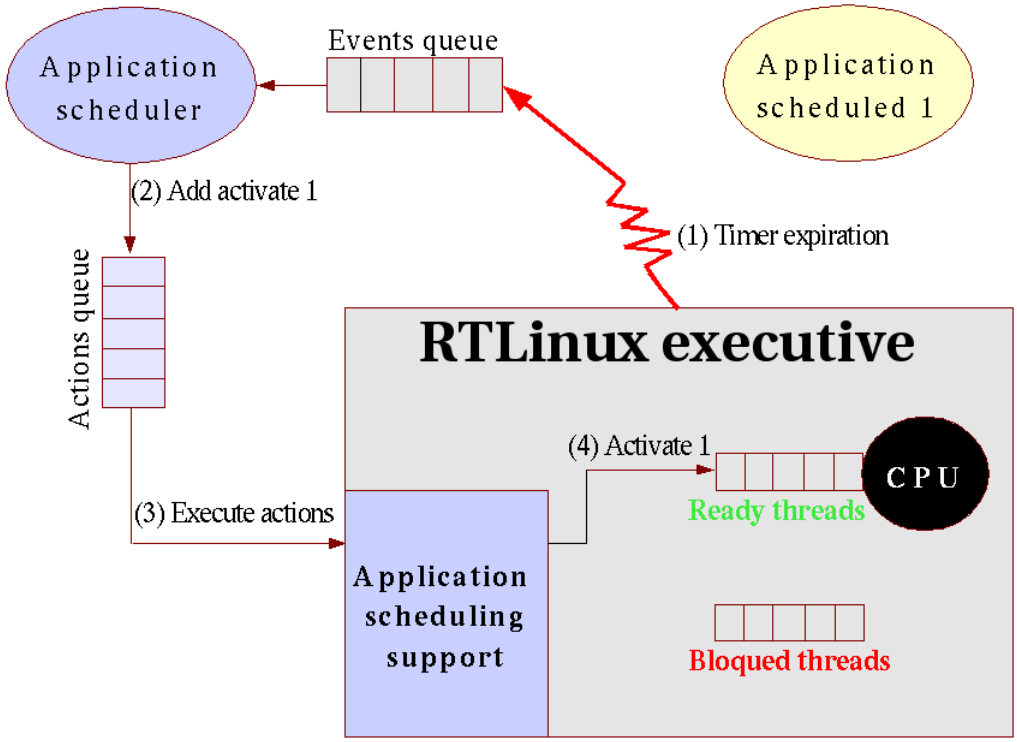


Figure 5-5. Application-defined scheduler operation: activating a thread

5.2.5. Application-defined scheduling events

In this model, the RTOS kernel sends scheduling events of the application-scheduled threads to its application-scheduler, instead of processing it. An scheduling event is everything (a blocking system call, a timer expiration, etc.) that can trigger a change in the thread's state.

Table 5-1. ADS events

Event Code	Description
POSIX_APPSCHED_NEW	A thread has been created
POSIX_APPSCHED_TERMINATE	A thread has been terminated
POSIX_APPSCHED_READY	A thread has become unblocked by the system
POSIX_APPSCHED_BLOCKED	A thread has been blocked
POSIX_APPSCHED_YIELD	A thread yields the CPU
POSIX_APPSCHED_SIGNAL	A signal belonging to the requested set has been accepted by the scheduler thread
POSIX_APPSCHED_CHANGE_SCHED_PARAM	A thread has changed its scheduling parameters
POSIX_APPSCHED_EXPLICIT_CALL	A thread has explicitly invoked the scheduler
POSIX_APPSCHED_TIMEOUT	A timeout has expired

Event Code	Description
POSIX_APPSCHED_PRIORITY_INHERIT	A thread has inherited a new system priority due to the use of system mutexes
POSIX_APPSCHED_PRIORITY_UNINHERIT	A thread has finished the inheritance of a system priority
POSIX_APPSCHED_INIT_MUTEX	A thread has initialized an application-scheduled mutex
POSIX_APPSCHED_DESTROY_MUTEX	A thread has destroyed an application-scheduled mutex

The application-defined scheduler will receive these events. Depending on the event type, it will program a scheduling action (activate, suspend or lock a thread) to be executed by the RTOS kernel. All this data flow (from RTOS to the application-scheduler) and instructions flow (from the application-scheduler to RTOS) could seem complex and few efficient. Nevertheless, it is solved thorough a single system call: `posix_appched_execute_actions()`.

5.2.5.1. Application-defined scheduling actions

The `posix_appched_execute_actions()` function is the main operation of the interface. It allows the application scheduler to execute a list of scheduling actions and then it suspends waiting for the next scheduling event to be reported by the system.

5.2.6. Application-defined mutexes

The POSIX-Compatible Application-defined scheduling API allows creating mutexes whose synchronization protocol is defined by the application scheduler. To allow this functionality a set of functions similar to those used with regular threads is available. These special mutexes are created like any other POSIX mutex but setting the value `PTHREAD_APPSCHED_PROTOCOL` for their protocol attribute. For this kind of mutexes two new attributes has been added: the appscheduler attribute and the appschedparam attribute. The appscheduler attribute identifies the scheduler thread the mutex is oto-line attached to. The optional appschedparam attribute can be used for passing application-defined mutex scheduling attributes to the scheduler.

As for the application-scheduled threads, it is also important for the scheduler to have simple mechanism to attach and retrieve the scheduling specific data associated with an application-scheduled mutex. With this purpose the interface introduces a new functionality not defined in POSIX: the mutex.

5.3. API / Compatibility

The application defined scheduler facility API is a little more complex than "normal" operating systems services like file management since the ADS has to provide two different API's. One API for the application scheduler thread and another API for the application scheduled thread.

Following is the list of function that can use the application scheduler:

```
/* program scheduling actions ( suspending or activating threads) */
int posix_appsched_actions_addactivate(posix_appsched_actions_t *sched_actions,
pthread_t thread);
int posix_appsched_actions_addsuspend(posix_appsched_actions_t *sched_actions,
pthread_t thread);
int posix_appsched_actions_addlock(posix_appsched_actions_t *sched_actions, pthread_t thread,
const pthread_mutex_t *mutex);
/* Execute Scheduling Actions */
int posix_appsched_execute_actions(const posix_appsched_actions_t *sched_actions, const
sigset_t *set , const struct timespec *timeout, struct timespec *current_time, struct
posix_appsched_event *event);
/* Getting and setting app. Scheduled thread's data */
```

```
int pthread_remote_setspecific(pthread_key_t key, pthread_t th, void *value);
void *pthread_remote_getspecific(pthread_key_t key, pthread_t th);
/* Set and get mutex-specific data */
int posix_appsched_mutex_setspecific(pthread_mutex_t *mutex, void *value);
int posix_appsched_mutex_getspecific(const pthread_mutex_t *mutex, void **data);
/* Scheduling events sets manipulation */
int posix_appsched_emptyset(posix_appsched_eventset_t *set);
int posix_appsched_fillset(posix_appsched_eventset_t *set);
int posix_appsched_addset(posix_appsched_eventset_t *set, int appsched_event);
int posix_appsched_delset(posix_appsched_eventset_t *set, int appsched_event);
int posix_appsched_ismember(const posix_appsched_eventset_t *set, int appsched_event);
int posix_appsched_seteventmask(const posix_appsched_eventset_t *set);
int posix_appsched_geteventmask(posix_appsched_eventset_t *set);
```

While in the application scheduled thread's side the API is:

```
/*Explicit scheduler invocation*/
int posix_appsched_invoke_scheduler(void *msg, size_t msg_size);
/*Manipulate application scheduled threads attributes*/
int pthread_attr_setthread_type(pthread_attr_t *attr, int type);
int pthread_attr_setappscheduler(pthread_attr_t *attr, pthread_t sched);
int pthread_attr_setappsched_param(pthread_attr_t *attr, void *param, int size);
int pthread_attr_getappscheduler(pthread_attr_t *attr, pthread_t sched);
int pthread_getappsched_param(pthread_attr_t *attr, pthread_t *sched, void *param, int *
size);
/*Application-defined Mutex Protocol*/
int pthread_mutexattr_setappscheduler(pthread_mutexattr_t *attr, struct rtl_thread_struct
*appscheduler);
int pthread_mutexattr_getappscheduler(const pthread_mutexattr_t *attr, struct
rtl_thread_struct *appscheduler);
int pthread_mutexattr_setappschedparam(pthread_mutexattr_t *attr, const struct
pthread_mutex_schedparam *sched_param);
int pthread_mutexattr_getappschedparam(const pthread_mutexattr_t *attr, struct
pthread_mutex_schedparam *sched_param);
int pthread_mutex_setappschedparam(pthread_mutex_t *mutex, const struct
pthread_mutex_schedparam *sched_param);
int pthread_mutex_getappschedparam(const pthread_mutex_t *mutex, struct
pthread_mutex_schedparam *sched_param);
```

5.4. Implementation issues

5.4.1. Optimizations

Due to the special characteristic of RTLinux, where all the threads as well as the RTLinux executive share the same memory space, system calls are implemented as simple functions calls. Even in some cases, the API is implemented as `inline` functions, and data can be shared (not copied) between RTLinux and user threads. It is important to note that these optimizations do not jeopardize the standard API.

5.4.2. Data types

In order to manage the lists of scheduling actions and events we have implemented circular queues. This data type allows to have constant access (O(1)) when producing/consuming events/actions. Special care, we have had to guarantee mutual exclusion when producing/consuming events/actions. The low level synchronization primitives used to guarantee mutual exclusion includes disabling interrupts and spin locks, depending on the situation.

5.4.3. API improvements

In the initial proposal of the POSIX-Compatible Application-defined Scheduling some changes to the `sched_param` structure were proposed. Changing this structure will require the modification, among other things, of the standard "C" library. The new version

of the "C" library will not be backward compatible. Therefore, we decided not to modify the `sched_param` structure with new member variables but add new functions to the API to send that parameters to the kernel. ADS authors (Mario Aldea and Michael González) where alerted about the compatibility problem and they have included the suggested changes in a new API review.

5.5. Tests and validation

We have developed several tests to validate the component. This tests were following next goals:

- ADS functionality validation.
- Measuring ADS overhead.
- Checking compatibility/portability.

In the following sections, each group of tests is described in detail.

5.5.1. ADS functionality validation.

The goal of these tests was to check that it was possible to implement most of the well-known scheduling policies using ADS API. With this in mind, the following policies and resource management protocols has been implemented:

- Fixed priority preemptive scheduler.
- EDF (Earliest Deadline First scheduler) [Liu73].
- CBS (Constant bandwidth Server) [Abeni98].
- PCP (Priority Ceiling Protocol) [Sha90].
- SRP (Shared Resource Protocol) [Baker91].

5.5.1.1. Fixed priority preemptive application-defined scheduler test

5.5.1.1.1. Description

This is an implementation of RTLinux scheduler using the application-defined scheduling API. The implementation offers the possibility of scheduling a set of tasks both using timer in mode one-shot and periodic, as the RTLinux scheduler does. RTLinux scheduler allows thread code to run at specific times. RTLinux has a fixed priority preemptive scheduler. The scheduling policy is as follow: each thread has assigned a priority; when several threads are ready, the thread with the highest priority is executed; a thread with high priority always preempt a minor priority thread whenever it becomes ready; each thread is supposed to relinquish the CPU voluntarily.

5.5.1.1.2. Goal

The goal of this test is to check that RTLinux scheduler can be easily implemented as an application-defined scheduler.

5.5.1.1.3. Procedure

To check the correctness of the fixed priority application-defined scheduler what it is done is to schedule a set of threads using the RTLinux system scheduler. Then the same set of threads are scheduled using the fixed priority preemptive application-defined scheduler. To do this uncomment the #define __FIXED_PRIORITY_APP_SCHED__ in the `fixed.h` file. Finally it is checked that the scheduling result is the same by comparing the two chronograms provided by the test.

5.5.1.1.4. Results

The following figures represent an instance of the test. In this instance four tasks are scheduled using RTLinux system scheduler (Figure 5-6) and its implementation using the ADS API (Figure 5-7)

In Figure 5-7, first thread corresponds with Linux while the next thread that appears corresponds to the application-defined scheduler. Observe that the application-defined scheduler, is executed before any of its scheduled tasks execution and after any of its scheduled tasks ends is execution for current activation.

As you it can be observed from both chronograms, the scheduling result is the same regardless using RTLinux scheduler or its implementation using the ADS API.

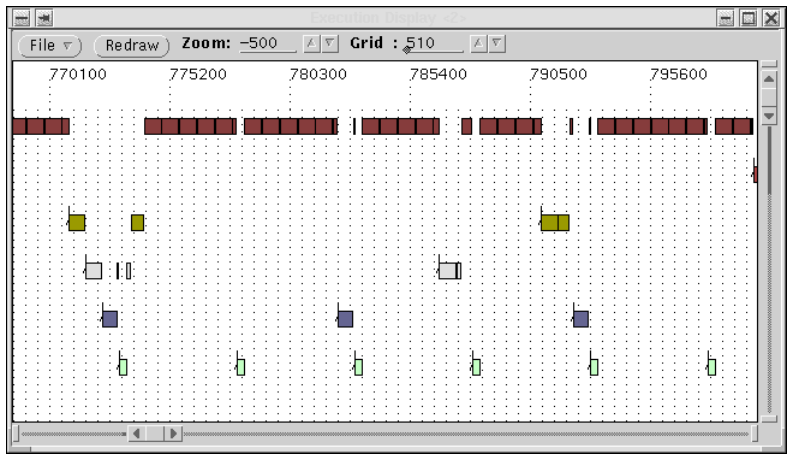


Figure 5-6. System-scheduled threads chronogram

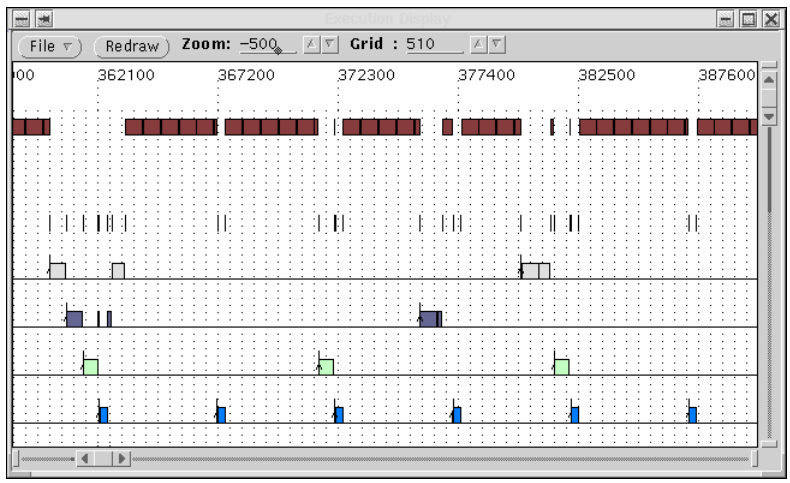


Figure 5-7. Application-scheduled threads chronogram

5.5.1.2. Earliest Deadline First scheduler with Stack Resource Protocol

5.5.1.2.1. Description

This is an implementation of the Earliest Deadline First scheduling policy with the Stack resource protocol, using the application-defined scheduling API. The implementation has followed the guidelines and tests given in [Balbastre] to provide both fixed and dynamic priority scheduling: Threads are ordered by priority, and among the same static priority threads with closer deadline are executed first. It is a Rate Monotonic policy and an EDF policy at each priority level.

Despite of not being implemented yet in any commercial RTOS, dynamic scheduling policies offers many advantages: they are particularly appropriate to soft systems; they

could help in a missed hard deadline, and a 100% processor utilization can be accomplished.

When jobs are allowed to access shared resources these accesses need to be controlled, as in any concurrent system, through the use of appropriate protocols to ensure the integrity of the resources despite potential concurrent access and minimizing priority inversion [Stankovic]. In [Baker91] a general resource access protocol is proposed that provides a solution to the priority inversion problem, the Stack Resource Policy (SRP). The SRP scheduling rules state that a task is not allowed to start executing until its priority is the highest among the active tasks and its preemption level is greater than the system ceiling. The name SRP comes from the fact that it can be easily implemented using a single stack, i.e., all threads can use the same stack to store function call parameters and return addresses (but threads can not suspend, usleep..., while active).

5.5.1.2.2. Goal

The goal of this test is to check that RTLinux EDF scheduler can be easily implemented as an application-defined scheduler.

5.5.1.2.3. Procedure

To check the correctness of the implementation, the tests provided in [Balbastre] has been run with the application-defined scheduler implementation.

5.5.1.2.4. Results

Figure 5-8 shows the result of the test for the EDF+SRP application-defined scheduler. The up arrow over the colored box indicates that the task has locked a mutex, while the down arrow means that the mutex has been unlocked.

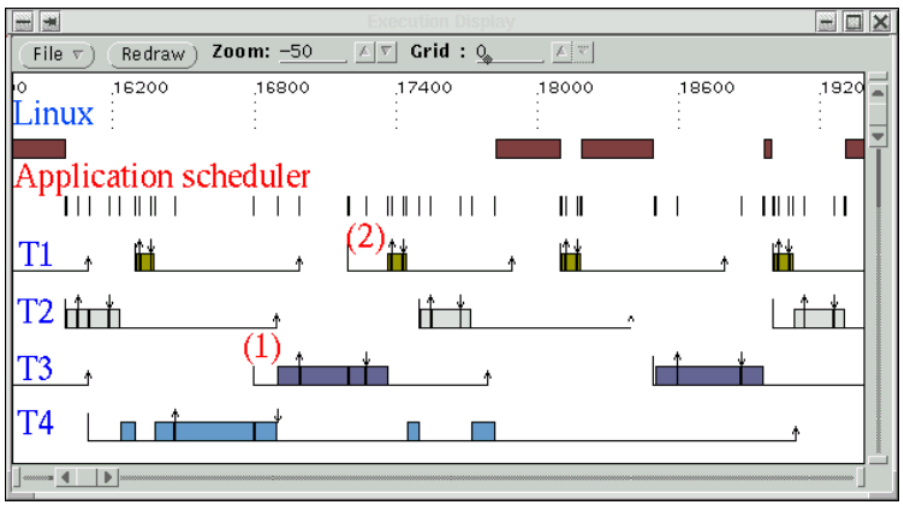


Figure 5-8. EDF+SRP application scheduler chronogram

Figure 5-9 shows the result of the test for native implementation of the EDF+SRP scheduler.

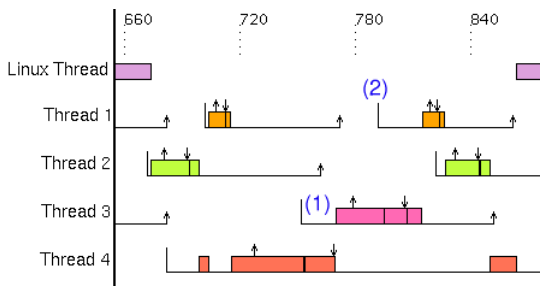


Figure 5-9. Native implementation of EDF+SRP chronogram

As you can see the scheduling result is the same for both chronograms (little differences on task CPU consumption since they aren't executed in the same machine).

In the above chronograms, thread 4 at time point (1) holds mutex 1 and prevents thread 3 (which at that time has more priority) from start its execution due to the SRP policy. It is also possible to see another characteristic of the SRP, when a thread start running all the resources it requires are granted. That is, a lock operation do not blocks the thread}, just raises the system preemption ceiling.

Also note that at time (2), thread 3 has a shorted absolute deadline and so higher dynamic priority than thread 1, therefore thread 3 is executed first. If threads were scheduled SCHED_FIFO thread 1 would always be executed first since it has always more priority.

5.5.1.3. Constant Bandwidth Server

5.5.1.3.1. Description

This are some functionalities added to EDF application-scheduler in order to provide CBS scheduling.

The Constant Bandwidth Server (CBS) [Abeni98] was developed to efficiently handle soft real-time requests with a variable or unknown execution behavior under EDF scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task is assigned a fraction of the CPU and it is scheduled in such a way that it will never demand more than its reserved bandwidth, independently of its actual requests. This is achieved by assigning each soft task a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded. As a consequence, overruns occurring on a served task will only delay that task, without compromising the bandwidth assigned to other tasks. By isolating the effects of task overloads, hard tasks can be guaranteed using classical schedulability analysis.

5.5.1.3.2. Goal

The goal of this test is to implement the Constant Bandwidth Server using the application-defined scheduling API.

5.5.1.3.3. Procedure

The CBS application-defined scheduler has been validated running several task loads and checking the correctness of the scheduling result.

5.5.1.3.4. Results

Figure 5-10 illustrates an example in which a hard periodic task, t1 is scheduled together with a soft task, t2, served by a CBS having a budget Qs = 2 and period Ts =7. Figure 5-11 shows a real execution of this example using the application-scheduler ported to RTLinux. In Figure 5-10 t1 with a period of 30 ms is colored in green while t2 with a period of 70 ms is colored in grey; jobs arrivals are generated by last thread in chronogram. The first job t2 arrives at time r1=2, when the server is idle. Being cs >= (ds,0 - r1)Us, the job is assigned the deadline ds,1 = r1 + Ts = 9 and cs is recharged at Qs = 2. At time t1 = 6, the budget is exhausted, so a new deadline ds,2 = ds,1 + Ts = 16 is generated and cs is replenished. At time r2, the second job arrives when the server is active, so the request is enqueued. When the first job finishes, the second job is served with the actual server deadline (ds,2 = 16). At time t2 = 12, the server budget is exhausted so a new server deadline ds,3 = ds,2 + Ts = 23 is generated and cs is replenished to Qs. The third job arrives at time r3 = 17, when the server is idle and cs = 1 < (ds,3 - r3)Us = 1.71, so it is scheduled with the actual server deadline ds,3 without changing the budget.

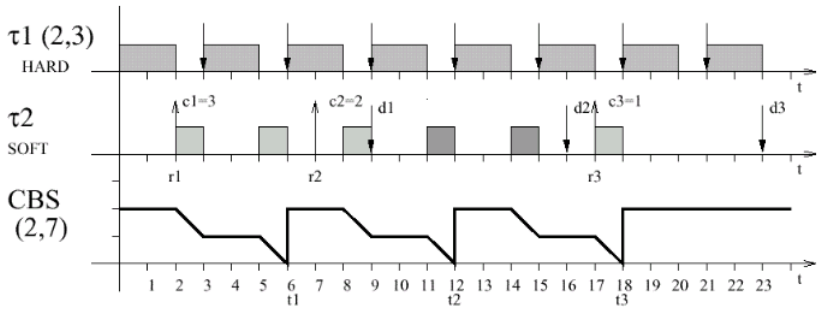


Figure 5-10. CBS example

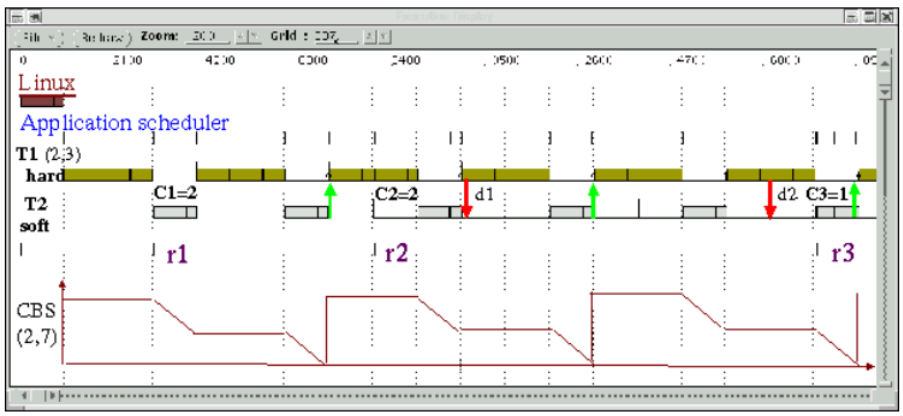


Figure 5-11. CBS application-scheduler chronogram

5.5.2. Overhead tests

The goal of these tests is to measure the overhead introduced by RTLinux application-defined scheduling support. To do this, three tests are provided:

- Scheduling a non time consuming thread.
- Measuring overhead per number of scheduled thread
- Baker's utilization test

5.5.2.1. Scheduling no time consuming threads

5.5.2.1.1. Description

The tests provided consists of the implementation of the RTLinux scheduler (fixed priority preemptive) using the application-defined scheduling API and some functionalities added to the RTLinux scheduler (allocated in directory monitor) in order to measure threads CPU usage.

5.5.2.1.2. Goal

The goal of this test is to compare the overhead needed to schedule a system-scheduled thread with the overhead needed to schedule an application-scheduled thread, using the same scheduling policy (fixed priority preemptive).

5.5.2.1.3. Procedure

What it is done is to compare the CPU usage of a periodic system-scheduled thread which only wastes time while calling the system-scheduler API with the CPU usage of

an application-scheduled thread which only wastes time while calling the application-defined scheduling API. To do so, a thread whose code is showed bellow, is scheduled in case 1 by RTLinux system scheduler Figure 5-12. In case 2, the same thread is scheduled by the fixed priority application-defined scheduler Figure 5-13.

```
      ....
      while(1){
#ifdef __FIXED_PRIORITY_APP_SCHED__
      posix_appsched_explicit_call();
#else
      pthread_wait_np();
#endif
      }
      ....
```

In both cases, the scheduled thread period is equal to one millisecond, i.e. thread code executes 1000 times in a second. As it can be observed from above code, in both cases the scheduled thread only wastes time while calling to its scheduler API. To measure the scheduling API overhead, every second, thread's CPU usage is registered. To switch from measuring RTLinux overhead to measure ADS overhead you should uncomment the #define __FIXED_PRIORITY_APP_SCHED__ located in the `fixed.h` file and type make test).

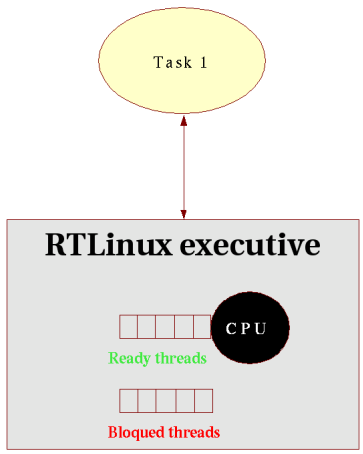


Figure 5-12. Case 1: System-scheduled thread

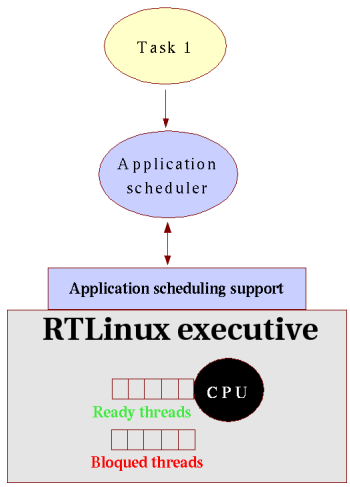


Figure 5-13. Case 2: Application scheduled thread

5.5.2.1.4. Results

As you will see when running the tests the scheduling result from both proves is the same but overhead introduced per thread activation when scheduling application-scheduled threads is 4 times overhead introduced when scheduling a system scheduled thread.

If the scheduled threads overhead is compared from case 1 and 2 (thread ids 1 and 2 respectively) you will see that they are very similar. If you divide CPU usage by the number of times that the thread is activated in a second (1000), you will find that the overhead per thread activation is similar to a context switch.

The main overhead is introduced by the application-scheduler thread (three times the cost of scheduling a thread). This thread will introduce at least an additional switch context per application scheduled thread activation (depending on the scheduling policy) plus the cost of implementing the scheduling policy (find the most urgent, arm timers,....). In the case of the fixed priority preemptive scheduler (the RTLinux default scheduling policy) the overhead introduced is three times the overhead needed to schedule a system thread per scheduled thread activation (introduces two context switches per scheduled thread activation plus the cost of implementing the policy).

To see this in more detail, take a detained look to the thread's utilization log stored in the file /tmp/ulog.txt. On an AMD k6 3D now 300MHZ for thread periods of 1 millisecond taking CPU usage samples every 1 second, this file looks like

Table 5-2. System scheduled overhead test

Second	thread_id	CPU usage (in ns)
	....	
10	0 (Linux)	94360198
10	1 (rt-task)	5639801
11	0 (Linux)	994376003
11	1 (rt-task)	5623996
	....	

Table 5-3. Application-scheduled overhead test

Second	thread_id	CPU usage (in ns)
	....	
10	0 (Linux)	978098983
10	1 (app. scheduler)	16407346
10	2 (app. scheduled)	5493670
11	0 (Linux)	978266118
11	1 (app. scheduler)	16339931
11	2 (app. scheduled)	5393950
	....	

NOTES:

- The CPU usage of each thread is measured every 1 second as the nanoseconds a thread holds the CPU.
- In RTLinux doesn't exists kernel stack, so system calls are implemented as normal functions calls.
- The thread id 0 allways corresponds to Linux (idle time) and the others threads id correspond to the order the threads appear on the chronogram showed. In the case of the application scheduled thread the id 0 corresponds to Linux, the id 1 to the application scheduler and the id 2 to the application-scheduled thread.

5.5.2.2. Measuring overhead per number of scheduled threads

5.5.2.2.1. Description

This test is the similar to the previous one. The main difference is that instead of measuring the overhead for one single thread, the overhead introduced when scheduling different number of threads (1,2,4,...) is measured.

5.5.2.2.2. Goal

The goal of this tests is to check the lineality of ADS API overhead, i.e to measure ADS overhead per number of scheduled threads.

5.5.2.2.3. Procedure

In this test the overhead introduced by RTLinux runtime and application-defined scheduling is measured depending on the number of scheduled threads. To do so a set of no time consuming tasks is scheduled using both RTLinux API and the fixed priority application-defined scheduler. Several proves are made to measure introduced overhead for both RTLinux runtime and application-defined scheduling support, when scheduling 1,2,4,8,16,32 and 64 threads with periods of 10 milliseconds each one.

5.5.2.2.4. Results

As it can be seen from Figure 5-14, the overhead introduced by application-defined scheduling runtime (in green) increases lineally with the number of threads. The application scheduling runtime overhead has been calculated as the difference between the overhead introduced (in red) when scheduling n threads using RTLinux API and the overhead introduced when scheduling the same n tasks using the application-defined scheduling API (in blue).

Also, it can be observed that application-defined runtime overhead tends to be twice RTLinux runtime overhead, when the number of scheduled threads increases. A possible explanation could be that fixed priority application-defined scheduler orders threads by priority. So there is no need of running completely threads list when finding next preemptor. This is not possible with the RTLinux scheduler. This optimization doesn't improves runtime worst case execution time. Nevertheless, mean runtime execution time is decreased considerably.

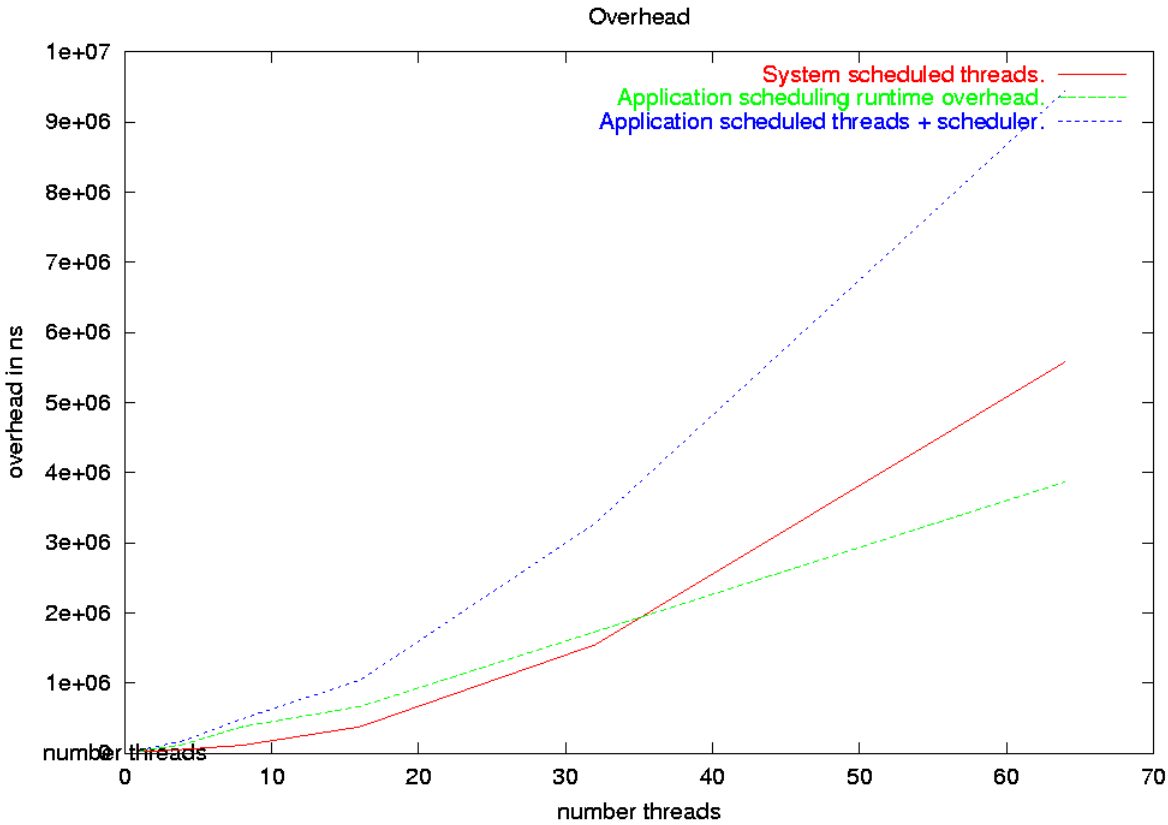


Figure 5-14. RTLinux executive and application scheduling overhead per number of scheduled threads

5.5.2.3. Baker's utilization test

5.5.2.3.1. Description

This is an implementation of Baker's utilization test, to measure runtime's overhead.

5.5.2.3.2. Goal

The goal of this test is to measure the limit of the CPU ujtilization when a set of tasks increases its computation time.

5.5.2.3.3. Procedure

This test allows to measure runtime's overhead scheduling six harmonic tasks. Harmonic tasks have the following periods: 1/320HZ, 2/320HZ, 4/320HZ, 8/320HZ, 16/320HZ and 32/320HZ= 100 milliseconds respectively. In each test iteration (every second) tasks load (CPU consumption) is increased by an amount. The test finishes when a task losses its deadline. At this point, the Utilization is calculated taking tasks load from previous iteration. A chronogram of this test for the application-scheduling runtime is showed in Figure 5-15. At point 1, task, 6 meets its deadline and still there is idle time (Linux task enters). In next iteration tasks load is increased by an amount. At point 2, tasks 6 looses its deadline and there is no idle time (Linux tasks doesn't enters) and test is finished. At that point utilization is calculated.

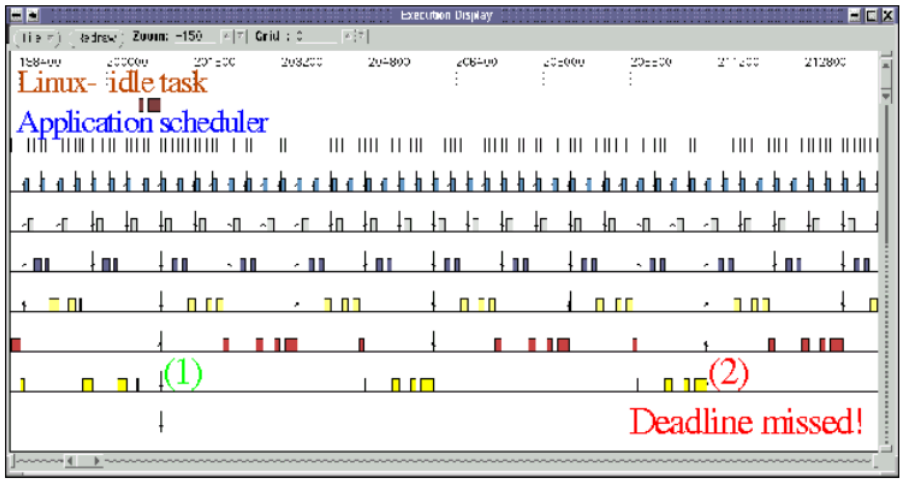


Figure 5-15. Baker utilization test chronogram

5.5.2.3.4. Results

The maximum utilization reached is equal to 97.68 % (2.32 % of overhead) for the application-scheduling runtime. In the case of the RTLinux runtime only, utilization reaches 98.94 % (1.06% of overhead). This results are in consonancy with last test results which stated that Application scheduling runtime overhead tends to twice times RTLinux runtime overhead, approximately.

5.5.2.4. MARTE OS policies & protocols porting

5.5.2.4.1. Description

This is a porting of MaRTE OS application-defined schedulers in order to run them under RTLinux.

The following policies and protocols EDF(for deadlines equal to periods), CBS and PCP have been ported from the MARTE OS application-scheduling implementation, in order to check compatibility. Major differences between MARTE OS and RTLinux are:

- No dynamic memory in RTLinux
- Lack of real time signals in RTLinux.
- Use of dynamic linkable modules to load programs.

5.5.2.4.2. Goal

The goal of this porting is to check compatibility/portability between the MaRTE OS implementation and the RTLinux one.

5.5.2.4.3. Simple EDF

This is a porting of the simple Earliest Deadline First implementation of MARTE OS with some debug code to show executions chronograms. The scheduled threads are periodic with deadline equal to period. For each scheduled thread a periodic timer is programmed which spires each time the activation time is reached.

In Figure 5-16 a chronogram of two EDF application-scheduled thread is showed. The threads have periods equal to deadline of 20 and 50 millisecond, respectively. Thread 1 (purple) corresponds to Linux and the second thread to the application scheduler.

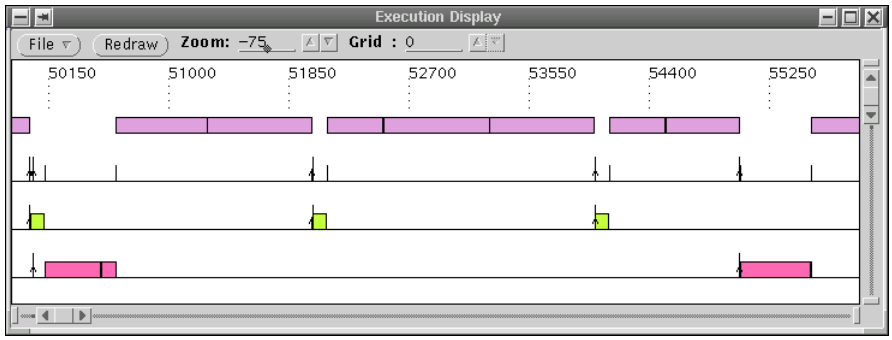


Figure 5-16. Trivial EDF chronogram (periods=deadlines)

5.5.2.4.4. Priority Ceiling Protocol

This is a porting of the Priority Ceiling Protocol implementation of MARTE OS with some debug code to show executions chronograms. PCP is a resource management protocol that solves the problem of priority inversion. Priority Ceiling means that while a thread owns the mutex it runs at a priority higher than any other thread that may acquire the mutex. Its main characteristics can be stated in:

- Each shared mutex us initialized to a priority ceiling.
- A thread can lock a mutex only if it's priority is higher than the ceilings of all other locked mutexes.
-
- Whenever a thread locks this mutex, the priority of the thread is raised to the priority ceiling.

5.5.2.4.5. Priority Ceiling Protocol

This works as long as the priority ceiling is greater than the priorities of any thread that may lock the mutex (hence its name).

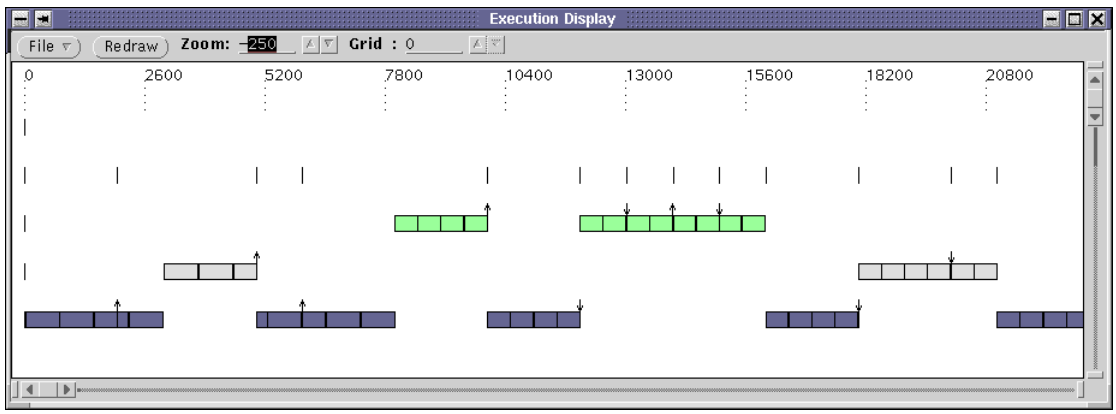


Figure 5-17. PCP chronogram

To understand Figure 5-17 name th0 to the application-scheduler and th1, th2, th3 (with colors green, grey and blue respectively) to the application scheduled threads. There are two mutexes, name them m1 and m2. The sequence of locks/unlocks showed in figure Figure 5-17 is as follows:

First th3 locks m2. Then th2 tries to lock m2 and gets blocked. Next th3 locks m1. Then th1 tries to lock m1 and gets blocked. Next th3 unlocks m1 and th1 gets ready. Next th1 unlocks m1 and finishes. Then th3 unlocks m2 and th2 gets ready.

5.5.2.4.6. CBS

This is a porting of the CBS implementation of MARTE OS with some debug code to show executions chronograms.

5.5.2.4.7. Results

There has been no problems to make run ported application-defined schedulers in RTLinux.

5.6. Validation criteria

The main problem of POSIX-Compatible application defined scheduling is the overhead introduced. This overhead can be divided in two parts:

Application scheduled thread overhead:

There is no significative difference between the overhead introduced when scheduling a system scheduled thread or an application scheduled thread.

Application-defined scheduler thread overhead:

This thread will introduce at least an additional switch context per application scheduled thread activation (depending on the scheduling policy) plus the cost of implementing the scheduling policy (find the most urgent, arm timers,....). In the case of the fixed priority preemptive scheduler (the RTLinux default scheduling policy) the overhead introduced is three times the overhead needed to schedule a system thread per scheduled thread activation (introduces two context switches per scheduled thread plus the cost of implementing the policy).

5.7. Installation instructions

In order to install POSIX-Compatible application-defined scheduling in RTLinux, please follow next steps:

- Get a fresh copy of the RTLinux 3.2pre1 version.
- Install the POSIX signals and timers components.
- Install the Application-defined scheduling component. Components installation is very easy, the only thing you have to do is to edit the Makefile and set the variable KERNEL to the directory where you have decompressed the RTLinux 3.2pre1 version. Next type make install and the patch and component related files will be applied.
- Select POSIX signals, timers and application scheduling from configuration menu (**make clean ;make xconfig**)

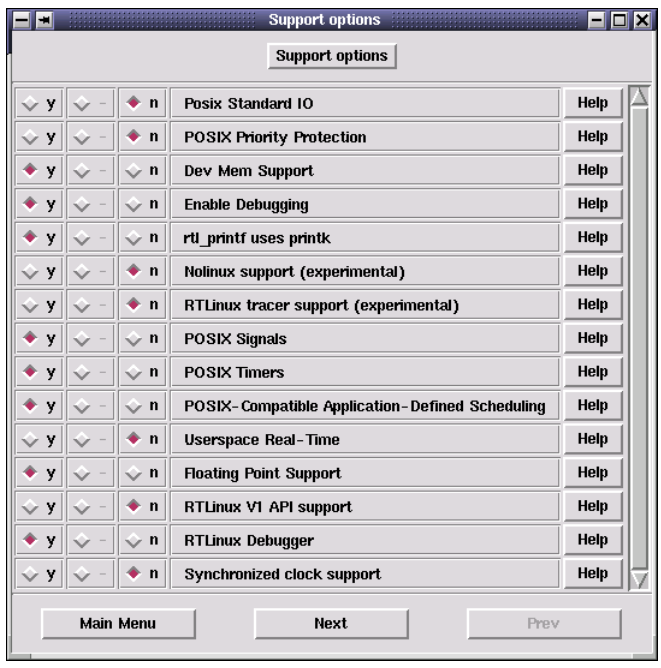


Figure 5-18. Support options

Chapter 6. POSIX Message Queues

6.1. Summary

Name
: POSIX Message Queues (**PMQUEUE**)

Description
: This component implement The POSIX message queues facility between RTLinux threads.

Author
: Sergio Saez

Reviewer
: Alejandro Lucero

Layer
: High level RTLinux.

Version
: 0.2

Status
: Testing

Dependencies
: PSIGNALS component.

Release Date
: M2

6.2. Description

UNIX systems offers several possibilities for interprocess communication: signals, pipes and FIFO queues, shared memory, sockets, etc. In RTLinux, the most flexible one is shared memory, but the programmer has to use alternative synchronisation mechanism to build a safe communication mechanism between process or threads. On the other hand, signals and pipes lack certain flexibility to establish communication channels between process.

In order to cover some of these weaknesses, POSIX standard proposes a message passing facility that offers:

- **Protected and synchronised access to the message queue**. Access to data stored in the message queue is properly protected against concurrent operations.
- **Prioritised messages**. Processes can build several flows over the same queue, and it is ensured that the receiver will pick up the oldest message from the most urgent flow.
- **Asynchronous and temporised operation**. Processes have not to wait for operation to be finish, i.e., they can send a message without having to wait for someone to read that message. They also can wait an specified amount of time or nothing at all, if the message queue is full or empty.
- **Asynchronous notification of message arrivals**. A receiver process can configure the message queue to be notified on message arrivals. So such a process can be working on something else until the expected message arrives.

6.3. Layer

POSIX Message Queues is a message passing facility that relies only on services that are already available or that are going to be incorporated by other components to the RTLinux core. As they do not require any modification of the RTLinux, they can be located at the High-Level RTLinux layer.

6.4. API / Compatibility

This components follows the POSIX API specification for message passing facility defined in IEEE Std 1003.1-2001. This API also belongs to the Open Group Base Specifications Issue 6. The following synopsis presents the list of supported message queue functions:

```
int     mq_close (mqd_t);
int     mq_getattr (mqd_t, struct mq_attr *);
int     mq_notify (mqd_t, const struct sigevent *);
mqd_t   mq_open (const char *, int, ...);
ssize_t mq_receive (mqd_t, char *, size_t, unsigned *);
int     mq_send (mqd_t, const char *, size_t, unsigned );
int     mq_setattr (mqd_t, const struct mq_attr *, struct mq_attr *);
ssize_t mq_timedreceive (mqd_t, char *, size_t, unsigned *, const struct timespec *);
int     mq_timedsend (mqd_t, const char *, size_t, unsigned, const struct timespec *);
int     mq_unlink (const char *)
```

6.5. Dependencies

This component has been developed for the RTLinux executive version 3.2 pre-release 1. It depends on several already available RTLinux services, as POSIX Timeouts and POSIX Semaphores (including the function `sem_timedwait()`), and new POSIX services developed in this project, as POSIX Signals, that also improve the behaviour of some already present services.

Although POSIX Timers are not required, if they are available, timeout parameters in functions `mq_timedsend()` and `mq_timedreceive()` have to be based on the CLOCK_REALTIME clock, as specified in IEEE Std 1003.1-2001.

6.6. Status

This component is in the testing stage. Already passed tests are described bellow, in Section 6.8, *Tests*.

6.7. Implementation issues

POSIX Message Queues implementation does not require to modify the core RTLinux executive. So implementation issues are only intended for internal structures and algorithms.

Several issues have been considered when implementing POSIX Message Queues. They are related with memory allocation at queue creation instant, synchronisation issues and management of message priorities. Next section presents these issues.

6.7.1. Queue creation

When a message queue is created, using the `mq_open()` function, the required information about queue and messages maximum size becomes available. Then, the maximum memory requirements for operation of the message queue are known and the resource reservation can be performed. This was the original intention described in the rational

of the standard: resource reservation can be performed only at one point, i.e., at queue creation.

As the original RTLinux executive had no dynamic memory support (Note that this component was developed in parallel with the DYNMEM component), message queues creation can be performed only when the module is loaded into the kernel. In this instant, Linux kernel `kmalloc()` is available for dynamic memory reservation and then `mq_open()` function can be implemented without problems. This characteristic introduces an additional restriction in the original POSIX API, that is already present in other RTLinux POSIX functions implementation (e.g. `pthread_create()`).

This restriction simplifies a lot the message queue creation process since it eliminates synchronisation requirements. The Linux module loading process guarantees the atomicity when calling `mq_open()` function, and therefore no synchronisation mechanism is needed to achieve mutual exclusion when the internal structures of message queue system are modified.

A second version of this component can rely on the dynamic memory component, providing a less restrictive implementation of the `mq_open()` function. Although this option will provide a more flexible API, the new version will require to follow the POSIX standard requirements about atomicity of message queue opening process. These requirements will introduce an additional overhead when opening and creating a new message queue.

6.7.2. Synchronisation issues

Sending and receiving messages requires to use synchronisation mechanism to achieve mutual exclusion when the internal structures of message queue system are modified. RTLinux executive provides several synchronisation services: low-level synchronisation, POSIX Pthread Mutex and Condition Variables, and POSIX Semaphores. Each of these options are analysed next

Low-level synchronisation mechanism

This mechanism is based on using spin-locks and disabling external interrupts to allow system- wide mutual exclusion regions. Using this mechanism requires to re-implement a semaphore-like service and a timeout service for timed send/receive functions. Additionally, blocking times when accessing shared data of message queues can influence in the scheduling of every thread in the system and not only in the threads that can access to a given message queue. However, this kind of custom implementations could obtain a lower overhead than a generic solution.

POSIX Mutexes and Condition Variables

This mechanism is the most elaborated one, and it is specially design for the kind of shared access that is perform in POSIX Message Queues. Only a very important restriction disallows the use this mechanism: threads waiting in a message queue must exit if they receive a signal, and threads waiting in a POSIX Mutex remains blocked after the execution of the corresponding signal handler. Additionally, there are no timed wait in a POSIX Mutex, and therefore, an extra mechanism should be design to implement timed send and received functions.

POSIX Semaphores

POSIX Semaphores is an intermediate solution between low- level mechanism and POSIX Mutexes. However, they have two additional advantages over POSIX Mutexes: first, a thread blocked in a POSIX Semaphore can be extracted from the waiting queue when a signal arrives; and second, a POSIX Semaphore allows to block a

thread during an specified amount of time. Both functionalities perfectly match the requirements that POSIX standard imposes on the behaviour of POSIX Message Queues implementations.

As it can be derived from the analysis showed above, the selected mechanism to implement mutual exclusion access to shared data in a message queue was POSIX Semaphores. This service was already available in the RTLinux executive, but it has been improved when implementing POSIX Signals and POSIX Timers components. Now, the improved RTLinux Semaphores conforms the POSIX standard providing the services and the behaviour required for implementing POSIX Message Queues.

6.7.3. Message sorting

POSIX Message Queues standard requires that the receive operation always returns the oldest message with the highest priority. That requires the message queue performs some kind of sorting that allows to extract the messages in priority order, and within each priority in FIFO order.

Different options have been analysed for the implementation of this component. The different options that have been considered are:

- Use a sorted queue that allocates all the pending messages sorted by priority and within each priority in FIFO order. This structure can be, e.g., a heap of pointers to messages.

  This solution has a low memory requirement, but each insertion and extraction from the message queue will have a logarithmic computational cost respect to the number of messages in the queue.

- Use a sorted priority queue that allocates only one token per priority, sorting the queue only by the priority value. Each priority has its own FIFO queue that stores the pending messages on that priority. When a receive operation is performed, the highest priority token is used to select the FIFO queue to extract the message. When a FIFO queue becomes empty, the corresponding priority token is removed from the top of the priority queue. When the FIFO queue of a new message is empty, the priority token have to be inserted in the priority queue. In that way, the priority queue only has tokens that correspond with no empty FIFO queues.

  This implementation provides a constant computational cost for insertions and extractions, when the FIFO queue of a given priority is not empty, i.e., this solution optimises the FIFO access to the queue. In the worst case, when a new message arrives and its FIFO queue is empty, the computational cost of inserting the priority token is logarithmic respect to the number of active priorities in the queue. This value is always equal or lower than the number of messages in the queue, and therefore, this approach has a better computational cost than the previous solution.

  On the other hand, the memory requirements of this solution are proportional to number of available priorities.

- Use a bitmap to store the priorities used by the pending messages and low-level processor-specific instructions to find out the highest priority stored in the bitmap. The rest of the implementation could remain as the previous solution (FIFO queues within each priority).

  This approach should obtain the best trade-off between computational costs and memory requirements, but it is also the less portable solution.

At this moment, the second approach has been selected as the basic sorting mechanism. A future version of this component will probably offer all these solutions as configuration options.

6.8. Tests

Basic conformance tests have been done based on the Open POSIX Test Suite from Sourceforge GPL Open Source Project. The current release only works on functionality level, being other kind of tests like definition or stress tests still not covered.

To test at definition level means to test that types defined by POSIX Message Queues are correctly defined into include files provided by the implementation. POSIX standard only requires two types to be defined: `struct mq_attr` and `mqd_t`. These types have not been tested explicitly, but the tests are implicitly performed when testing implementation at functionality level, since both types are used into API functions.

Other POSIX required types, as `struct sigevent` or `struct timespec`, depend on new components or other parts of RTLinux, and therefore its correct definition is out of the scope of this component. However, its definition has been implicitly tested when tests for the corresponding functions `mq_notify`, `mq_timedreceive` and `mq_timedsend` have been performed at functionality level.

Stress test tries to check what is the behaviour of the system when resources are massively demanded, or what is the behaviour when a high number of threads are sending and receiving messages simultaneously. The first case corresponds when a lot of message queues are created and activated at the same time. POSIX Message Queues define some limits that are fixed at compilation time, so these kind of stress tests are well delimited. Stress tests with several threads using one or more message queues structures at the same time have been performed. Since Message Queues use RTLinux semaphores for synchronisation and POSIX Signals for notification (mainly), the behaviour will be dependent of these components.

Functionality tests for every API function have been done, testing different possibilities of behaviour: how messages are inserted into queues depending on what priority they have, and how are these messages delivered. POSIX standard fixes error values returned by functions, so different conformance tests where these different error values must be returned have been done. Finally, although most of these test can be done only with one single thread, we have preferred to use distinct threads when possible to have a more realistic behaviour, instead of how Test Suite from Sourceforge does it, using one single process for send and receive messages.

6.9. Validation criteria

This component provides a new message passing facility that was not present in the current version of the RTLinux executive. This new feature allows prioritised communication between different hard real time tasks with bounded overheads. This facility was highly demanded in the Real Time Linux community. An extension that allows the same kind of communication between Linux processes and RTLinux threads is already planned.

Performance issues in POSIX Message Queues strongly rely on the performance of synchronisation mechanisms and the system memory bandwidth. This implementation uses semaphores as a synchronisation mechanism in order to achieve mutual exclusion when accessing to shared data and for blocking sending/receiving threads when a given queue is full/empty.

On the other hand, POSIX standard requires a two copies implementation of message send/receive process when coping data to/from message queues and this implies an extra overhead not required when uses lightweight threads. One of the best ways to improve the Message Queues performance could be to extend the POSIX standard API with sev-

eral function that allows a zero-copies version of send/receive process. However, the two-copies implementation simplifies the design of a future extension of Message Queues for communicating Linux processes and RTLinux threads. This issue will be studied in the second phase of the components development.

Chapter 7. Dynamic Memory Allocator

7.1. Summary

Name
: Dynamic Memory Allocator (**DYNMEM**)

Description
: This component provides standard dynamic memory allocation, `malloc()` and `free()` functions, with real-time performance.

Author/s
: Miguel Masmano.

Reviewer
: Ismael Ripoll

Layer
: High level RTLinux.

Version
: 0.70

Status
: Stable

Dependencies
: None. If available, it can make use of the BigPhysArea facility.

Release Date
: M2

7.2. Descritpion

RTLinux do not provide any kind of memory management support, neither virtual memory (by mean of the processor MMU page table or memory segments) nor simple memory allocation, as the one provided by the standard "C" library. RTLinux applications has to preallocate all the required memory in the `init_module()` function before the threads are created. Once the RTLinux threads are created, the only memory that can be used is the stack of each thread.

The main reason behind the lack of memory management support in RTLinux is the idea that both virtual memory and dynamic storage allocator (DSA) algorithms introduce a unbounded overhead, making the system response unpredictable and non-realtime.

This component provides dynamic memory allocation with real time characteristics. The allocator implemented in this component is a new DSA algorithm specially designed with the aim of obtaining bounded response time. The new algorithm is called DIDMA (Doubly Indexed Dynamic Memory Allocator). It is based on the use of a pure segregated strategy. See [Paul 95] for an exhaustive taxonomy and review of existing DSA algorithms.

The DIDMA algorithm was designed with the following main ideas in mind:

- Bounded response time for both `malloc()` and `free()` functions. For this reason, DIDMA can not be based on a simple list of free blocks like first fit or best fit strategies.
- Low response time. It should provide better worst case response time than conventional algorithms.
- Low and bounded internal fragmentation. One important difference between a non-realtime application and a realtime one, besides the timing requirements, is that a realtime application is usually executed during long periods of time. It is common that the realtime application runs as long as the system is up. On the other hand,

non-realtime applications (like for example a compiler or an accounting program) are executed only a few hours.

The longer a program is executed, the more memory fragmentation can cause. Some algorithms -like buddy systems- define some restrictions on the way free blocks are merged to buildup a bigger free block in order to get faster response time. This is not a valid strategy for a realtime system.

- Minimum splitting threshold. Allocating very small block of memory (for example smaller than 8 or 16 bytes) is not a common programming practice. Also it will permit to store in the free blocks management information. In the final implementation of DIDMA, the minimum blocks size for DIDMA is 32 bytes.
- Immediate coalescing. Some algorithms delay the coalescing of free blocks as an heuristic attempt to avoid future block splitting. Since usually programs use (allocate and free) blocks of the same size, some DSA algorithms try to give better response time to block sizes already requested.

  The two main drawbacks of delayed coalescing are:

  1. It increment the amount of external fragmentation.
  2. Increment the worst case response time when several free blocks has to be merged in order to complete a request of bigger size.

Some of the features implemented on the DIDMA algorithm was also present in the algorithm **Half-Fit** proposed by [Ogasawara 95]. The main improvement of DIDMA over Halt-Fit was the addition of a second level directory to reduce the external fragmentation.

7.3. Layer

Originally it was designed as a high-level RTLinux component, but it can be easily ported to be used in the application Linux layer to replace the non-realtime `glibc` dynamic memory implementation. Also, it can be ported to RTAI with almost no code changes.

In order to simplify the porting to other systems, the code is not portable (specific processor instructions, or RTLinux system calls) are located a special file or defined as preprocessor macros.

7.4. API / Compatibility

In order to avoid naming conflicts, the API provided by DIDMA is non POSIX, it looks like the API given by the ANSI "C" standard adding a `rt_` prefix.

```
void *rt_malloc(size_t *size);
void rt_free(void *ptr);
void *rt_calloc(size_t nsize, size_t elem_size);
void *rt_realloc(void *p, size_t new_len);
```

And it also provides several macros which are equal than ANSI-C functions interface for dynamic memory allocation.

```
void *malloc(size_t *size);
void free(void *ptr);
void *calloc(size_t nsize, size_t elem_size);
void *realloc(void *p, size_t new_len);
```

7.5. Dependencies

None. The source code contains optional code that makes use of the BigPhysArea facility automatically if available. The user do not have to configure the component.

7.6. Status

The current version of DIDMA is testing 0.70.

7.7. Implementation issues

DIDMA is based on an indexed strategy with a fixed size preallocated data structure. All DSA algorithms have a similar operation schema. Initially, there is a large pool of continuos free memory; when a block of memory is requested, `malloc()`, it is allocated from the initial memory pool; when a block is freed then if there are free blocks adjacent they are merged to form a bigger free block; once coalesced, if the free block is in the middle of allocated blocks then it is linked in the *free block data structure*. The next time the application requests block, the ADS has to find a chunk of memory big enough to fit the requested size, now the ADS has two places to look for free memory: in the free block data structure or in the initial memory pool. If the chunk of free memory found is bigger than the requested, it is splited and the remaining piece of free memory is inserted (linked) in the free block data structure.

As can be seen, the heard of any DSA algorithm is the data structure used to store freed blocks. DIDMA data structure is implemented as two arrays. Originally it was implemented as a two dimensional array, but due to efficiency issues it was split into a first level directory array and a second level that actually contains the lists of free blocks.

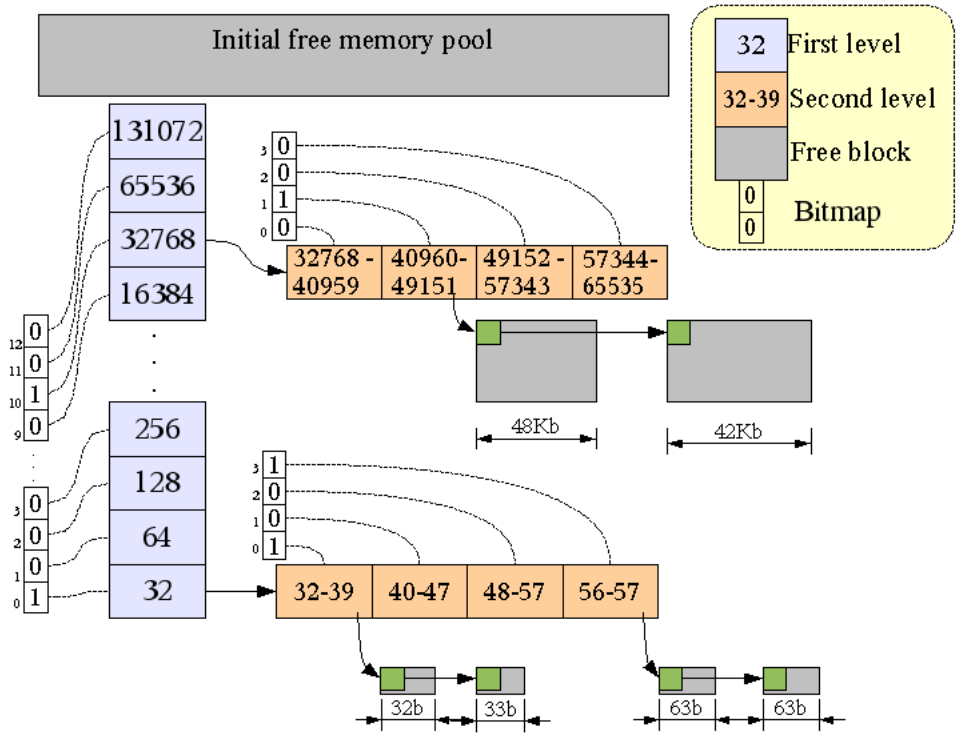


Figure 7-1. DIDMA data structure

Free blocks in the range 2^i to 2^{i+1}-1 are stored in the (i-5) element of the first array, remember that the smallest block of memory is 32=2^5. The first element holds information related to blocks of size 32, the second element in this array is for blocks of size 64, and so on. The lenght of this first array, First_Level_Size, will determine the maximum block size that the DIDMA algorithm can manage.

Each element of the first array points to a second array which divides the first block range into Second_Level_Size blocks of smaller size. The number of elements in the second array, Second_Level_Size, is one of the configuration parameters of DIDMA. The bigger is Second_Level_Size the less fragmentation, but at the same time the worst case response time gets longer. This second level divides the range 2^i to 2^{i+1}-1 lineally. In this second array are located the head of the lists of free blocks.

The pointers to link all the free block of similar size are located inside the free blocks.

In order to locate the head of the free blocks of a given size, we use two mapping functions. These mapping functions are optimised to use simple bit and arithmetic operations functions (shifts, adds, ffs, etc).

The size of each of these two arrays is not fixed and can be customised to fulfil the specific application requirements. These two parameters determine the response time of the `malloc()` and `free()` functions and also the maximum fragmentation.

In order to make the search implementations on DIDMA structure and DIDMA structure more efficient several strategies have been used:

- Since most processors has logical instructions to find the first bit set in an word, in fact, the glic library provides the function `int ffs(int)` (find first bit set in a word). Both arrays levels have an associated bitmap. Bit is set if the referenced entry in the array has at least one free block. That way it is possible to the suitable free block in contatnt time, no search, just compute a function.
- DIDMA structure first level is created in the algorithm initialization but the second level is empty until it is really needed. For the second level is reserved a memory block (of user defined size), when the second level needs a memory, it take a block from this second reserved memory and later, when the block is not necessary, DIDMA returns it. This dummy malloc used by the second level is very efficient because it only allocates and deallocates fix size blocks with constant response time.

Each free block inserted into DIDMA structure has a header which contains the following information:

- The block size.
- Pointers to the next and the previous contiguous free physical blocks, these pointers ara needed to merge contiguous free blocks inmediately, with constant time.
- Pointers to next and previous free block of the same size.

7.8. Tests

The tests will be used to both, validate that the proposed algorithm works correctly and compare its performance with other algorithms.

Test 1: The first test calls 1000 times the `malloc()` function with a value of 1 byte and later it calls 1000 times the `free()` function.

Test 2: Initially reserves 1000 blocks and later it frees all the blocks except the last block reserved, afterwards the second test calls 1000 times `malloc()` with increasing values, and finally it calls 1000 times `free()`.

Test 3: For the initialization, the third test reserves 1000 blocks of a increasing value and then it only frees even blocks, when the initialisation has finished, its behaviour is like the first test but `malloc()` is called with random values. This test was designed to test how the allocator performs with highly fragmented memory.

Test 4: The fourth test calls randomly 1000 times `malloc()` with fixed values, these values are: 16, 330, 512, 600, 816, 1030, 3000, 4800, 8000 and later `free()` is called. This test allocates block of non related size (the chosen numbers are not multiple among them, also are not power of 2) and not completely random. This test tries to behalf like a real application.

Test 5: The fifth test calls randomly 1000 times `malloc()` or the `free()` function with random values.

7.9. Validation criteria

The worst case temporal complexity of the proposed algorithm is bounded since it do not depend on the number of free blocks, it uses a closed functions (no loop) to find a free block and to return a free block to the free block structure.

Previous tests were used to obtain empirical performance measures if DIDMA. The same test set were also executed with other memory allocator algorithms to do a comparative analysis. Following is the list of DSA algorithms implemented ad compared.

□ Binary Buddy.
□ RTAI dynamic memory allocator
□ RTEMS dynamic memory allocator
□ Best fit
□ First fit

Buddy, First fit and Best fit are classical DSA algorithms. The other two algorithms are the one used in real-time systems. It has not been possible to implement and compare more DSA algorithms used in real-time systems due to the little information provided available.

The testes were executed on an Athlon XP 2000 with 512 Mbytes of RAM. The DIDMA algorithm parameters was customised to First_Level_Size = 24 (with this value the size of the memory pool will be 16 MBytes) and Second_Level_Size = 16 (the second level index has importance on the internal fragmentation), which are reasonable default values.

An important part of the high performance of current processor relies on the memory cache subsystem. Most DSA algorithms share a common internal code characteristics that stress the cache effect: the size of the code of the DSA algorithms is small; and the code is mostly sequential code with no loops. As a result, the first time the processor has to execute the `malloc()` or `free()` functions, the timing penalty due to cache misses is several times greater (in some cases one order of magnitude) than the time we pretend to measure. In order to properly compare the performance of the algorithms, the first executions of the algorithm were used as cache "warming-up" and not taken into account.

First number is the time of the `malloc()` call and the second number is the time required to complete a `free()` operation (malloc/free). All the results are on nanoseconds:

Table 7-1. Test 1 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	62/159	160/176	58/82	79/88	109/145	109/134
Maximum	64/320	480/544	96/96	96/128	128/256	128/192
Minimum	64/96	96/128	32/64	64/64	96/128	96/128

Table 7-2. Test 2 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	193/189	Failed	Failed	81/111	114/191	137/163
Maximum	224/352	Failed	Failed	96/128	160/352	160/224
Minimum	64/96	Failed	Failed	64/64	96/128	96/128

Table 7-3. Test 3 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	187/178	175/289	1172/153	1153/93	1627/186	136/153

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Maximum	256/320	512/1323	1344/1184	1376/160	2560/512	160/224
Minimum	64/96	96/128	32/64	64/64	1504/128	96/128

Table 7-4. Test 4 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	155/161	142/164	94/107	167/85	1889/151	145/156
Maximum	224/448	448/448	224/160	1600/128	14944/352	224/256
Minimum	64/96	96/128	32/64	64/64	1632/128	96/128

Table 7-5. Test 5 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	186/154	180/264	64/56	1176/68	1626/160	134/129
Maximum	256/320	512/1152	128/96	1376/128	2560/320	160/192
Minimum	64/96	128/96	32/32	64/32	1472/96	96/96

As shown in the tables, DIDMA has passed all the tests. A test was marked as "Failed" when an algorithm was unable to allocated the requested amount of memory due to excessive external fragmentation.

In the following diagrams show the maximum response time of malloc and free functions respectively.

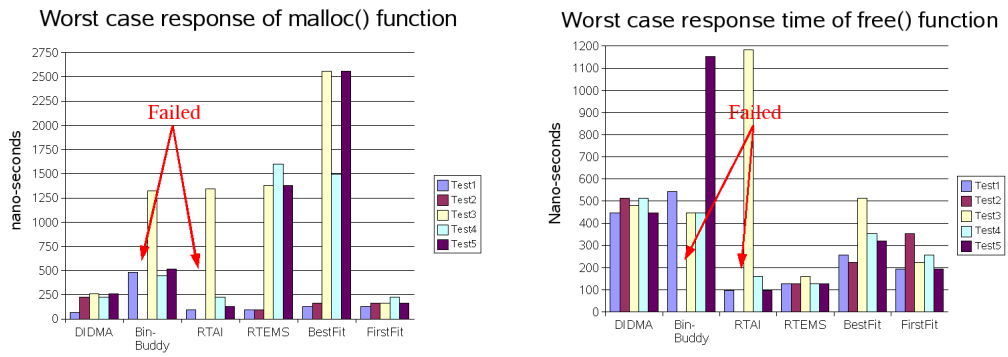


Figure 7-2. Comparison summary

As can be seen, the response time of the DIDMA algorithm is bounded and independent of the workload.

The response time of the DIDMA `free()` function is not as good as other algorithms. The main reason is that DIDMA do not delay the block merging, but do immediate coalescing. Let's remember that DIDMA has developed to work in a system with no virtual memory support, therefore, it is not possible to reorganise the memory to buildup new larger memory blocks using page tables. DIDMA was also designed to be used with long time running applications. For these two reasons, free memory must be merged immediately to avoid fragmentation and reduce worst case response time. The good performance of the RTEMS `free()` is because this algorithm do not merge free blocks since it was designed for hardware with virtual memory, which allows to enlarge the memory pool once exhausted (by calling the `brk()` function).

RTAI do immediate coalescing on free, but he previous and next adjacent free blocks are search by using a linear search in the list of free blocks. This search may produce a high

overhead if there are a large number of free blocks, this scenario is presented in test 3 with a workload the produces a high external fragmentation. DIDMA also do immediate coalescing, but previous and next free physical blocks are linked with pointers.

Although the First Fit algorithms perform the best in these tests, the theoretic worst case response time is very high (search through a lengthy linked list). For this reason the First Fit can not be used in real-time systems.

The most important conclusion from these results is that the new proposed algorithm, DIDMA, provides a good response time (low mean response time and bounded worst case response time) and also it makes an efficient use of the available memory.

Chapter 8. RTL-Gnat Porting

8.1. Summary

Name
: RTL GNAT Porting (**RTLGNAT**)

Description
: Provides support for Ada programming in RTLinux.

Author/s
: Miguel Masmano.

Reviewer
: Jorge Real.

Layer
: High level RTLinux.

Version
: 0.1

Status
: Alpha

Dependencies
: GNAT 3.14 and BigPhysArea patch applied to the Linux kernel.

Release Date
: M2

8.2. Description

RTLGnat is an Ada compiler for RTLinux. Ada is a standard programming language [Ada95] that was designed with a special emphasis on real-time and embedded systems programming, also covering other parts of modern programming such as distributed systems, systems programming, object oriented programming or information systems.

Why is Ada important to RTLinux? In RTLinux, real-time tasks are implemented as kernel modules, implemented in "C" (RTLinux also provides support for the "C++" language). Special care must be taken when writing kernel modules: an error in a single task can make the whole system to hang or crash, since these modules are executed in the kernel memory space. "C" is a language widely used in low level programming because of its high efficiency, but it is also true that it is not a programmer friendly language: simple and error-prone syntax, weakly typed, no run-time cheking, etc.

This is clearly an area where Ada can be of great help: Ada's strong typing, consistency checking, robust syntax and readability, and the availability of high quality compilers, encourage the writing of correct software and allow to catch bugs early in the implementation.

Not surprisingly, this component has caused a notorious public interest, with more than one hundred internet connections to its site in just one week. Also, a paper on the topic has been accepted ---with very positive reviews--- in the 8th International Conference on Reliable Software Technologies organised by Ada-Europe.

8.3. Layer

The modules created by RTLGnat are loaded on the RTLinux Applications layer. The porting has been done using only the facilities already available in RTLinux. The base RTLinux API has not been modified.

8.4. API / Compatibility

RTLGnat allows to load and run programs written in Ada on RTLinux. The parts of Ada that are supported by version 0.1 of RTLGnat are the following:

- Simple sequencial programs.
- Delays (both absolute and relative).
- Exceptions.
- Tasks.
- Asynchronous transfer of control.
- Dynamic priorities.
- Protected objects.
- Ceiling locking protocol.

Due to its early stage of development, RTLGnat does not provide support for general input-output of text (the Ada standard package `Ada.Text_IO` has not been implemented). Conversions between float and other types is neither possible yet. These two cases may well be covered in the next release of RTLGnat.

8.5. Dependencies

RTLGnat depends on the GNAT-3.14 compiler, GCC 2.8.1, RTLinux3.2-pre1 or above, and the BigPhysArea patch.

Although it is not a dependency, the POSIX trace (PTRACE) component has been used during the development and debugging phase.

8.6. Status

RTLGnat is in alpha status. The current version of RTLGnat is 0.1.

8.7. Implementation issues

The main goal to achieve is to be able to generate loadable kernel modules from Ada sources. First we will explain how RTLGnat does it with simple Ada programs, i.e., programs that do not use the run-time system support provided by GNAT.

There are two ways to load an Ada program as a kernel module; one way is to export the necessary symbols from the Ada program, as suggested by T. Baker in [Baker00]. The other way is to implement a C program, which exports the necessary symbols, and then link it with our Ada program; this is the method of our choice. We have found two reasons for this approach:

- Several RTLinux functions are written as macros, and it is easier to use them in C.
- The functions `init_module()` and `cleanup_module()` call several RTLinux functions (e.g., for the creation of threads). If we adopted the first approach, we should provide an interface to all these functions (Baker suggests to include them in the GNAT package `System.OS_Interface`). In our approach, we provide an object file that is incrementally linked with the application, therefore hiding RTLinux-specific symbols. This object file defines the Real-Time GNAT Layer, RTGL, that will be explained below.

An Ada program with no run-time can be easily compiled into a loadable kernel module by only providing the appropriate modifiers to the compiler command, such that the program is linked with a simple C object file before the final module is created.

Programs that make calls to the Ada run-time system require a modified run-time system in order to be executed on RTLinux. The applications compiled with GNAT are structured in several layers. The Ada program uses the services of the GNU Ada Run-time Library (GNARL), which interfaces with the underlying OS through a lower level layer called GNULLI (GNARL Lower Level Interface).

We have done a minimum number of changes to GNARL, most of them at the GNULLI level. In particular, the packages `System.OS_Primitives` and `System.OS_Interface` have been changed to use the interface of RTLinux instead of Linux and to provide adequate support for the ceiling locking operation.

In addition, several packages have been added to provide the following functionalities:

- `Rtl_IO`. This package implements a front-end for the `rtl_printf()` function, which allows to print objects of several data types on the screen. We have not implemented a `Get` equivalent yet.
- `Rtl_Fifo`. This package provides an interface to use the RTLinux FIFOs. It allows to create and destroy FIFOs, and to send to and receive from them. `Rtf_Create` and `Rtf_Destroy` make calls to the Linux Kernel. Since RTLinux modules run in the kernel memory space, special care must be taken to implement these calls. The algorithm we have implemented invokes the FIFO creation primitive by means of an interrupt that is queued to Linux, not a direct call. This avoids producing errors in the Linux kernel. This method is not bounded in time, but it is important to note that the FIFOs are created statically at startup, not during the normal program execution.
- `Rtl_Interrupts`. This package provides the functionality needed to deal with interrupts. It is basically a binding to the RTLinux functions for interrupt enabling, disabling and handling support.
- `Rtl_Pt1` This package provides a binding to the RTL Posix Tracing standard (POSIX 1003.1q), version 1.0. This package, in conjunction with a graphical viewer, allows us to graphically monitor the temporal behaviour of the Ada tasks in a program.

To allow a port of the GNAT run-time system, a new kernel module, RT-Gnat Layer (RTGL), has been implemented with all the glue code required by both RTLinux and the upper layers of the Ada run-time system. RTGL exports kernel symbols (like `init_module()`, `cleanup_module()`, `author`, `license` and `kernel_version` strings), and also the RTLinux interface and library functions.

The services offered by RTGL are:

- Library functions. The GNAT run-time system uses some standard functions (for string management and time conversions) which are not provided by RTLinux or the Linux kernel. RTGL provides them. Some of these functions are taken from the OSKIT project (Utah University, [Lepreau02] and from the source code of GNU GCC 2.8.1.
- Exported symbols. Since RTGL is implemented in C, it can import Kernel headers and it can directly export symbols like kernel version, etc. RTGL imports the `init_module` and `cleanup_module` functions.
- The DIDMA dynamic memory manager.

The resulting application architecture can be seen in Figure 8-1. The Ada application is divided into two parts, one with hard real-time requirements (the *Ada real-time application*) and another part that is scheduled by Linux (*Ada background tasks*), since it has no strict timing requirements. These two parts are two different Ada programs. The link between them is via real-time FIFOs. The real-time FIFO mechanism also allows hard real-time tasks to communicate with applications written in a different language.

The Ada real-time application uses the services provided by RTGL only at startup, for the creation of threads. The program then uses the modified run-time system to obtain the tasking services (tasks themselves, rendezvous, protected objects, etc). The Linux

kernel is scheduled in the background with respect to the Ada real-time application components.

The process of generating the module from the Ada sources consists in calling `gnatmake` with the appropriate options. A script called `rtgnatmake` has been written to automate this process.

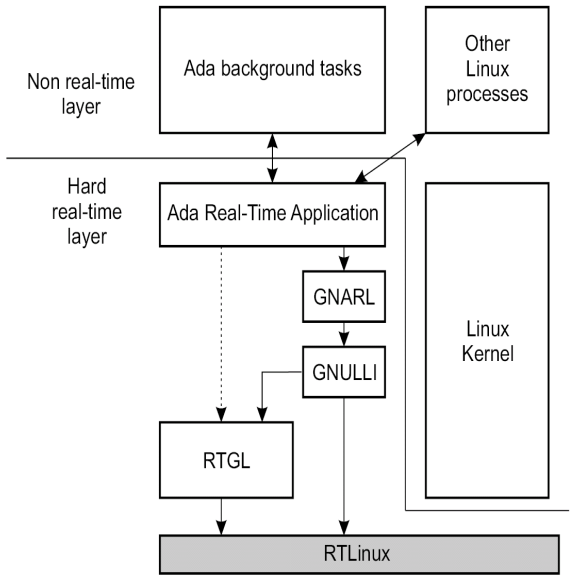


Figure 8-1. Architecture of the whole system, with an Ada application running on top of RTLinux

8.8. Tests

We have performed qualitative tests to verify the language functionalities mentioned above. The simple sequential program is the well known "hello world" test application. For the rest of features tested, we developed simple programs focusing on the particular feature to be tested. For example, the asynchronous transfer of control (ATC) was tested with a 3-level nested ATC program; for the ceiling locking protocol, two tasks performing protected actions with changing priorities; for RTFIFOs, a program that creates the FIFO and reads/writes to it; and so on. We have used the POSIX tracing tool by A. Terrasa to verify the reults of the examples.

We have also done some performance tests for RTLGnat. Up to now, we know that the maximum frequency of a single periodic task is 50 KHz on an Athlon/600 MHz. On the same hardware, MaRTE 1.0 [Aldea01] achieves 66 KHz. We did not expect to have better performance, since we have not focused on optimizations yet. For instance, the hardware clocks we are using are the Programmable Interval Timer (PIT) and Time Stamp Counter (TSC) of the ix86 architecture. It takes 3 microseconds to program the TSC+PIT timer on a Pentium III/550 MHz. We expect to achieve better results by using the APIC clock, which only requires 324 nanoseconds on the same platform. Another important source of performance differences with MaRTE is the fact that MaRTE uses a flat memory approach, with paging disabled, whilst Linux requires paging to be enabled, which also imposes an overhead.

To measure the kernel overhead, we have reproduced the tests presented by T. Baker, based on calculating the maximum achievable utilisation (in practice) of an harmonic task set. The task set is formed by 6 tasks with frequencies 320Hz, 160Hz, 80Hz, 40Hz, 20Hz and 10Hz. We gradually adjust the amount of time consumed by each task proportionally to the taks periods, i.e., 1 microsec increments for the most frequent task, 2

microsec for the second, 4 microsec for the third, 8 microsec for the fourth, and so on. The maximum theoretical utilisation in this case is 100%. The difference between the actual utilisation achieved and the theoretical maximum gives a figure of the total overhead imposed by the OS. We have compared an Ada implementation of the test program with a similar program in C, therefore we can measure both the kernel and the run-time system overhead. The measured utilisation are 98.69% for the Ada version and 99.45% for the C version.

8.9. Validation criteria

Quantitatives

- As a summary of the previous paragraph, the measured overhead of the Ada run-time system is only 0.76% with respect to the C version of the same program. This result has been obtained on a PC with an Athlon 900 processor.
- Some preliminary results show that the worst observed overhead in a task switch has been 20 microseconds.

Qualitatives

- With RTLGnat now it is possible to compile concurrent Ada programs on RTLinux and execute the program's tasks as POSIX threads. The Posix Tracer for RTLinux implemented by Andres Terrasa has been used to verify the operation of RTLGnat. All the tests have been passed.

Chapter 9. Constant Bandwith Server in RTLinux

9.1. Summary

Name
: Native CBS implementation based in RTLinux.

Description
: This component implements the CBS scheduling policy for RTLinux threads. The Constant Bandwidth Server (CBS) was developed to efficiently handle soft real-time requests with a variable or unknown execution behaviour under EDF scheduling policy.

Authors
: Pau Mendoza and Patricia Balbastre.

Reviewer
: Alfons Crespo.

Layer
: Low level RTLinux.

Version
: 0.1

Status
: Stable

Dependencies
: EDF scheduler for RTLinux.

Release Date
: M2

9.2. Descritpion

The problem of integrating flexible Quality of Service (QoS) guarantees in real-time systems has been widely studied in the last years, resulting in some interesting proposals. Probably, the most important theoretical result that emerged in this work is the idea that in order to provide a predictable QoS to different applications running on the same system, the OS kernel must provide temporal isolation between different applications or tasks. Temporal isolation (also known as temporal protection), requires that the temporal behaviour of a task is not influenced by the temporal behaviour of other tasks in the system. Based on classical real-time scheduling (EDF or RM priority assignment), it is possible to implement a reservation guarantee by simply enabling a task to execute as a real-time task (scheduled, for example, by EDF or RM) for the reserved time Q, and then blocking it (or scheduling it in background as a non real-time task) until the next reservation period. One of the most interesting resource reservation protocols is the Constant Bandwidth Server (CBS).

The Constant Bandwidth Server (CBS) [Abeni98] was developed to efficiently handle soft real-time requests with a variable or unknown execution behaviour under EDF scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to wich each soft task is assigned a fraction of the CPU and it is scheduled in such a way that it will never demand more than its reserved bandwidth, independently of its actual requests. This is achieved by assigning each soft task a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected

computation time, its deadline is postponed so that its reserved bandwidth is not exceeded. As a consequence, overruns ocurring on a served task will only delay that task, without compromising the bandwidth assigned to other tasks. By isolating the effects of task overloads, hard tasks can be guaranteed using classical schedulability analysis.

In more detail, as stated in [Abeni98] CBS can be defined as follows:

- A CBS server is characterized by a budget c_s and by the server bandwidth $U_s=Q_s/T_s$. Where Q_s is the maximum budget and T_s is the period of the server.
- Initially, the server is assigned the deadline $d_s=0$
- Each served job, is assigned the deadline of the server.
- Whenever the server task executes, the budget c_s is decreased by the same amount.
- When $c_s=0$ the server budget is recharged to the maximum value C_s and a new server deadline is generated as $d_{s,k+1}=d_{s,k}+T_s$.
- When a job $J_{i,j}$ arrives and the server is idle (there are no pending jobs), if $c_s \geq (d_{s,k}-r_{i,j})U_s$ the server generates a new deadline $d_{s,k+1}=r_{i,j}+T_s$ and c_s is recharged to the maximum value Q_s
- When a job finishes, the next pending job is served using the current budget and deadline.

In [Caccamo01] the CBS algorithm is extended to deal with shared resources, specifically, it is integrated with the Stack Resource Policy (SRP) [Baker91].

This component implemetns the CBS scheduling policy for RTLinux threads, that is, when the CBS thread is the highest priority one, it can execute Linux task. How linux processes are scheduled is not related with this component.

9.3. Layer

This is a low-level RTLinux and low-level Linux component, since it modifies both RTLinux and Linux kernel. Thus, the distribution of this component comes in the form of two patches, one for RTLinux (`rtlinux-3.2pre1-rtlcbs.patch`) and the other for Linux (`linux-2.4.18-rtlcbs.patch`).

9.4. API / Compatibility

To initialise CBS threads, new API functions have been defined. Specifically:

```
extern inline int pthread_attr_set_initbudget_np(pthread_t thread, hrtime_t initbudget);
extern inline int pthread_attr_get_initbudget_np(const pthread_t thread, hrtime_t
*initbudget);
extern inline int pthread_setinitbudget_np(pthread_t thread, hrtime_t initbudget);
extern inline int pthread_getinitbudget_np(pthread_t thread, hrtime_t *initbudget);
extern inline int pthread_initcbs_np(pthread_t thread, hrtime_t period);
void make_linux_task_cbs_server(hrtime_t start, hrtime_t deadline, hrtime_t period, int
priority);
```

9.5. Dependencies

CBS depends on EDF scheduler, since aperiodic events are scheduled with EDF scheduling policy. V1 API support has not to be selected when configuring RTLinux installation.

9.6. Status

This is a stable version. Some bugs have been already fixed from the beta version.

9.7. Implementation issues

The implementation work consists of modifications to two files: `rtl_sched.h` and `rtl_sched.c`. The modifications are mainly in the scheduler function `rtl_schedule()`, apart from the new functions and parameters added. A new scheduling policy is defined (`SCHED_CBS_NP`), for those threads working as CBS servers.

All the modifications done to the RTLinux code have been enclosed by macros (CONFIG_OC_RTLCBS) that can be toggled from the RTLinux configuration tool.

Each CBS server thread must have two parameters:

- The maximum budget Q_s. It has been added to the `rtl_sched_param` struct, because usually params belonging to this struct are initialized once, and after this, they do not change its value. The parameter has been called `sched_cbs_init_budget`:

```
struct rtl_sched_param {
  int sched_priority;
  hrtime_t sched_deadline;
#ifdef CONFIG_RTL_CBS
  hrtime_t sched_cbs_init_budget;
#define RTL_INIT_BUDGET(th) ((th)->sched_param.sched_cbs_init_budget)
#endif
};
```

- The current budget c_s. This param decreases and it is recharged as time goes by. So, it is more suitable to include it in the `rtl_thread_struct` struct:

```
struct rtl_thread_struct {
        ... fields of original RTLinux ...
#ifdef CONFIG_RTL_EDF
        hrtime_t current_deadline;
        int policy;
#ifdef CONFIG_RTL_CBS
        hrtime_t sched_cbs_current_budget;
#endif
#endif
};
```

When an aperiodic event arrives, depending on wether the CBS thread is idle or not, a new deadline for the CBS thread is calculated. This deadline is assigned to the thread associated with the aperiodic event. The implementation has been done in such a way that the user can choose between two options:

- The aperiodic events are interrupts, and the thread associated with these events is the linux task. When an interrupts arrives (for example, a network interrupt), linux executes until it schedules the idle task. In this implementation, there is no need to implement queues of pending jobs. As an example, let's suppose that two interrupts arrive at the same time. In the first interrupt arrival, if the CBS server is idle, a new deadline is calculated for the linux task. Linux task will recover its original state (as the background thread) when it schedules the idle task. So, the second interrupt will be already treated. If not, linux do not schedule the idle task until all tasks are finished.
- The user can customize CBS. Therefore, events have to be defined by the user, and the same CBS thread attends the execution of events. When the event processing finishes, the CBS thread suspends itself. In this option, the user must implement the queues of pending jobs.

From now on, the CBS thread will refer to the linux task or the CBS thread itself, depending on the option chosen.

In the first option, when a network interrupt arrives, the linux task must execute with the priority (static and dynamic) of the CBS server thread associated with the interrupt.

This way of inheritance between the linux task and the CBS thread is made in the interrupt handler, through the `make_linux_task_cbs_server` function. In this function, the CBS thread parameters (initbudget, deadline, period and priority) are attached to the linux task. Then, a new deadline is calculated in the `rtl_cbs_reset_deadline` function. The code of both functions is listed below:

```
extern void make_linux_task_cbs_server(hrtime_t start, hrtime_t initbudget,
                    hrtime_t deadline, hrtime_t period, int priority){
 hrtime_t now;
 int cpu_id = rtl_getcpuid();
 pthread_t linux_task = rtl_get_linux_thread(cpu_id);
 schedule_t *sched;

 if (linux_task->policy != SCHED_CBS_NP) {
     sched = &rtl_sched[cpu_id];
     now = sched->clock->gethrtime(sched->clock);

     RTL_INIT_BUDGET(linux_task) = initbudget;
     linux_task->sched_param.sched_deadline = deadline;
     linux_task->period = period;
     linux_task->sched_param.sched_priority = priority;
     linux_task->policy = SCHED_CBS_NP;

     linux_task->sched_cbs_current_budget = 0;
     if (linux_task->current_deadline < now)
         linux_task->current_deadline = now;
     cbs_linux_idle = 1;

     rtl_cbs_reset_deadline(linux_task);

     rtl_reschedule_thread(linux_task);

 };
 return;
}


static inline void rtl_cbs_reset_deadline(pthread_t t){
        hrtime_t now;
 rtl_irqstate_t flags;
 int cpu_id = rtl_getcpuid();
 schedule_t *sched;
 rtl_no_interrupts(flags);
 sched = &rtl_sched[cpu_id];
       if ((t->policy == SCHED_CBS_NP)) {
         now = sched->clock->gethrtime(sched->clock);
  if (now + (t->sched_cbs_current_budget/ RTL_INIT_BUDGET(t)) * t->period
                      >= t->current_deadline) {
          t->current_deadline = now + t->period;
   t->sched_cbs_current_budget = RTL_INIT_BUDGET(t);
  }
  t->resume_time = now;
 }
 rtl_restore_interrupts(flags);
}
```

If the second option is chosen, there is no need to use the `make_linux_task_cbs_server` function, since the CBS thread already has the correct parameters. Simply, the `rtl_cbs_reset_deadline` function is executed when the CBS thread is woke up. This is made in the `pthread_kill` function:

```
int pthread_kill(pthread_t thread, int signal){
 ... Original RTLinux code ...
#ifdef CONFIG_RTL_CBS
 if ((unsigned) signal == RTL_SIGNAL_WAKEUP)
         rtl_cbs_reset_deadline(thread);
#endif
 ... Original RTLinux code ...
}
```

Then, the CBS thread will execute when its new deadline is the nearest of all active threads. It is important to note that if another interrupt arrives,

the `rtl_cbs_reset_deadline` function will not execute, since the thread is already active. By definition of CBS, if a job arrives and the server is active the request is enqueued, so it will execute with the current deadline.

When the CBS thread is chosen by the scheduler as the highest priority thread, two situations can stop the thread execution:

- A higher priority thread becomes active in the future, but before the CBS thread finishes its execution. This is solved in the `find_preemptor` function in the original RTLinux code.
- The budget is exhausted, so it must be recharged and a new deadline is assigned.

In the original RTLinux code, the last situation is not taken into account, so if the thread is not preempted, the timer is programmed to ten miliseconds after. Now, the timer must be programmed to the nearest time of the previous cases commented: a higher thread activation, or the actual time plus the current budget. The code is:

```
if ((sched->clock->mode == RTL_CLOCK_MODE_ONESHOT && !test_bit (RTL_SCHED_TIMER_OK,
    &sched->sched_flags)) || (new_task->policy == SCHED_CBS_NP)) {
 if ( (preemptor = find_preemptor(sched,new_task))) {
#ifdef CONFIG_OC_RTLCBS
           if (new_task->policy == SCHED_CBS_NP) {
       if (( preemptor->resume_time - now) > (new_task->sched_cbs_current_budget)){
           (sched->clock)->settimer(sched->clock, new_task->sched_cbs_current_budget);
#endif
                 (sched->clock)->settimer(sched->clock, preemptor->resume_time - now);
#ifdef CONFIG_OC_RTLCBS
 }
 } else {
         (sched->clock)->settimer(sched->clock, preemptor->resume_time - now);
 }
#endif
 } else {
#ifdef CONFIG_OC_RTLCBS
         if (new_task->policy == SCHED_CBS_NP) {
                 (sched->clock)->settimer(sched->clock, new_task->sched_cbs_current_budget);
 } else
#endif
         (sched->clock)->settimer(sched->clock, (HRTICKS_PER_SEC / HZ) / 2);
 }
 set_bit (RTL_SCHED_TIMER_OK, &sched->sched_flags);
}
```

Once the CBS thread is preempted, the scheduler must measure the execution time, in order to decrease the current budget. When the budget is exhausted, it must be recharged and a new deadline is generated, according to CBS rules. This must be checked in the `rtl_schedule()` function before choosing the new task to execute. However, if the linux threads executes as the CBS server (first option) and it schedules the idle task (which is indicated by the flag `cbs_linux_idle`) then it must recover its original state, that is, as the background thread.

```
elapsed_time = now - rtl_prev_sched_time;
rtl_prev_sched_time = now;

if (sched->rtl_current->policy == SCHED_CBS_NP) {
  if (sched->rtl_current->sched_cbs_current_budget >= elapsed_time){
 sched->rtl_current->sched_cbs_current_budget -= elapsed_time;
 } else {
         sched->rtl_current->sched_cbs_current_budget = RTL_INIT_BUDGET(sched->rtl_current);
         sched->rtl_current->current_deadline += sched->rtl_current->period;
         sched->rtl_current->resume_time = now;
   }
  if (sched->rtl_current == &sched->rtl_linux_task) {
          if (cbs_linux_idle == 0) {
           sched->rtl_linux_task.sched_param.sched_priority = -1;
#ifdef CONFIG_RTL_EDF
   sched->rtl_linux_task.sched_param.sched_deadline = 0;
   sched->rtl_linux_task.current_deadline =           0;
#endif
   sched->rtl_linux_task.sched_cbs_current_budget = 0;
   RTL_INIT_BUDGET(&sched->rtl_linux_task) = 0;
   sched->rtl_linux_task.period = HRTIME_INFINITY;
```

```
    sched->rtl_linux_task.policy = SCHED_OTHER;
    }
   }
  }
```

The only point to solve now is to detect wether the linux task schedules the idle task. To do this, the linux code must be modified, specifically the `kernel/sched.c`, which is where the linux scheduler is implemented. The flag `cbs_linux_idle` is defined as *extern* in the `include/sched.h` file. It has the value 0 whenever the idle task is scheduled, and 1 otherwise. Then, simply, before the context switch in the `schedule()` function, it is added the following code:

```
if(next == idle_task(this_cpu)){
cbs_linux_idle = 0;
}
```

9.8. Tests and Validation Criteria

9.8.1. Validation criteria

The CBS implementation in RTLinux must have the following behavior:

- If the served tasks have the same period as the CBS server, then the CBS algorithm behaves as plain EDF.
- The CBS server must reclaim any spare time caused by early completions.

Regarding context switches, the CBS server must call the scheduler at most every *budget* units of time. Therefore, no more of (*task computation*/*server budget*) context switches must be introduced by the new implementation of CBS.

Also, a low overhead have been assured, since there is no need to implement queues of pending aperiodic arrivals.

9.8.2. Tests

Four tests have been implemented to validate the correct behaviour of the system. These tests can also be used as examples for the user, since it creates CBS threads to validate the correct behavior. The target system was a Pentium 133 MHz.

For every test developed, it has been used the same workload, that is, threads scheduled under EDF are the same in all tests. The parameters of the EDF periodic threads are listed in Table 9-1, *Thread parameters (in miliseconds)*. CBS parameters will be presented when explaining the specific tests.

Table 9-1. Thread parameters (in miliseconds)

Task id	Compute	Deadline	Period	Prio level
T1	0.8	6	6	10
T2	2.4	10	10	10
T3	3	11	11	10
T4	3.5	19	19	10

The structure of the tests is made up, basically, of two files: `rt_process.c` and `CBS_app.c`. The former contains the module that creates the four threads plus the CBS threads. The latter contains the code that generates the aperiodic events. The code to implement the four periodic threads is listed below:

```
pthread_attr_t attrib;
struct {
    int id;
    int compute;
    int period;
    hrtime_t deadline;
```

```
    hrtime_t budget;
    struct sched_param params;
} sched_attrib[6];
...
for (x=0; x<NTASKS; x++) {
      pthread_attr_init(&attrib);
      pthread_attr_setschedparam(&attrib, &sched_attrib[x].params);
      pthread_attr_setdeadline_np(&attrib, sched_attrib[x].deadline);
      pthread_attr_setschedpolicy(&attrib, SCHED_EDF_NP);
      pthread_create(&(tasks[x]), &attrib, fun, (void *)x);
      pthread_attr_destroy(&attrib);
}
```

While implementing these tests, some debugging code was also included. That feature was used to obtain the execution chronogram of the threads for each test. The debugging can be made selecting the option CONFIG_RTL_EDF_DEBUG in the config menu of RTLinux install. This way, it is obtained a trace via the rtf0 where the scheduler writes the thread identifier, initial and final execution times and activations of the thread, in a text file. This file is the input data to a X window application "crono" that represents the chronogram of the execution.

9.8.2.1. Test 1. CBS thread serves aperiodic jobs

This test has five threads: the four threads shown in Table 9-1, *Thread parameters (in miliseconds)*, plus the CBS thread. Parameters for the CBS thread are presented in Table 9-2, *CBS thread parameters for test 1 (in miliseconds)*

Table 9-2. CBS thread parameters for test 1 (in miliseconds)

Task id	Compute	Deadline	Period	Prio level	Max. budget
CBS1	1.2	3	3.1	10	0.15

The code implemented to create the CBS thread is the following:

```
for (x=0; x<N_CBSs; x++) {
 pthread_attr_init(&attrib);
 pthread_attr_setschedparam(&attrib, &sched_attrib[NTASKS+x].params);
 pthread_attr_setschedpolicy(&attrib, SCHED_CBS_NP);
 pthread_attr_setinitbudget_np(&attrib, sched_attrib[NTASKS+x].budget);
 pthread_create(&cbs_task,&attrib, cbs_code,(void *)x + NTASKS);
 pthread_attr_destroy(&attrib);
}
```

This thread also executes the code of aperiodic events. These events are generated by a user application by writing on a rt_fifo. Following, is the code of the user application (CBS_app):

```
#define RTF_IN_1           "/dev/rtf1"

int main(){
  int event=1;
  int ret_out=0;
  int fd;
  if ((fd = open(RTF_IN_1, O_WRONLY)) < 0) {
    fprintf(stderr, "Error opening /dev/rtf\n");
    exit(1);
  }

  fprintf(stderr, "%d\n",event);
  ret_out = write(fd,&event,sizeof(event));
  fprintf(stderr, "%d\n",event);
  close(fd);

  return 0;
}
```

As it shows the previous code, the user application generates two aperiodic events. The first event wakes up the CBS server (by means of the rt_fifo handler), that executes a dummy loop to consume its computation time. Afterwards, the thread suspends itself until the arrival of the second event. The code that executes the CBS server is:

```
void * cbs_code(void *arg){
    int id = (int)arg;
    int compute = sched_attrib[id].compute;
    int my_loop;

    pthread_initcbs_np(pthread_self(), sched_attrib[id].period);
    while(1) {
        pthread_suspend_np(pthread_self());
 for (my_loop=0; my_loop < compute ; my_loop++) {
     rtl_delay(DELAY);
 }
    }
    ...
}
```

Note that the CBS thread is not make periodic, that is, the new function `pthread_initcbs_np` is used instead of `pthread_make_periodic_np`. If the latter function is used, the CBS thread would be woke up in every period. This behaviour is not correct for a CBS server, since it must be woke up only when an aperiodic job arrives. Following is the chronogram (Figure 9-1) of the execution:

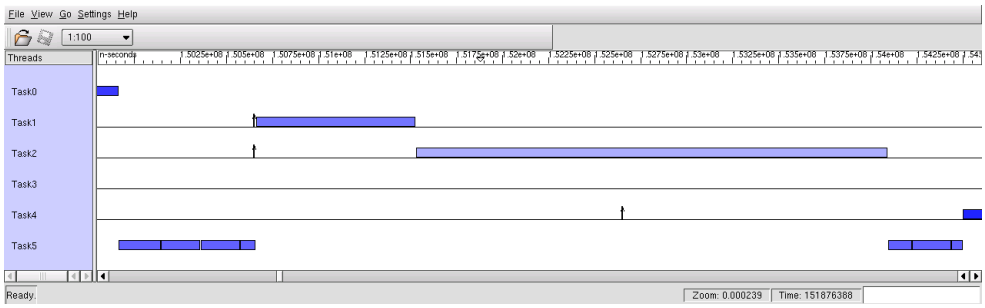


Figure 9-1. Chronogram execution for test 1

The chronogram generated by the execution shows the four periodic tasks plus the CBS server (task 5 in the chronogram) and the linux thread (task 0). In the first three replenishments CBS executes without interferences (since its deadline is shorter than others), but in the fourth it is preempted by task 1, and later by task 2. This is because in each replenishment the deadline is increased by the period.

9.8.2.2. Test 2. CBS thread serves aperiodic jobs

It is possible that when the CBS thread wakes up and executes, no other threads are active. In this situation, several replenishments can occur before other threads become active. Therefore, CBS deadline becomes greater, and at the end it behaves as the background thread. To increase the system utilization, this test is made up of the four periodic threads in Table 9-1, *Thread parameters (in miliseconds)* and two CBS threads. This way, CBS and periodic threads will alternate their executions whenever a replenishment occurs. Parameters for the CBS threads are presented in Table 9-3, *CBS thread parameters for test 2 (in miliseconds)*.

Table 9-3. CBS thread parameters for test 2 (in miliseconds)

Task id	Compute	Deadline	Period	Prio level	Max. budget
CBS1	0.6	3	3	10	0.2
CBS2	0.84	5	6	10	0.3

Regardin the implementation details, an periodic auxiliary thread is created to wake up CBS threads when the user application writes on a rt_fifo. This auxiliary thread wakes up all CBS threads, so they compete for the processor control. The code is following:

```
void *fun_aux(void *arg){
    int j;
    int end = 5;
```

```
    while (end != 0) {
 pthread_suspend_np(pthread_self());
 for (j = 0; j < N_CBSs; j++) {
     pthread_wakeup_np(cbs_tasks[j]);
 }
 end--;
    }
    pthread_exit(0);
    return (void *) 0;
}
```

The chronogram is shown in Figure 9-2. Again, tasks 1 to 4 are the periodic threads, tasks 5 and 6 are the CBS ones and task 7 is the auxiliary thread. The figure shows how the CBS thread with the shorter deadline wakes up and executes after the execution of the auxiliary thread. In its first replenishment, its deadline becomes greater than the other CBS thread, so CBS2 executes. CBS1 and CBS2 executions are alternating until periodic threads with shorter deadlines become active.

Figure 9-2. Chronogram execution for test 2

9.8.2.3. Test 3. Linux thread serves aperiodic jobs

Tests 3 and 4 shows how to associate linux task to serve aperiodic events. Is is considered that the aperiodic job has been served when linux schedules the idle task, and new events arrive when an interrupt is forwarded to linux. In test 3, aperiodic events are generated as in the previous test, that is, by means of the user aplication that writes on a fifo. The fifo handler now do not wake up the CBS server, since in this test is linux who serves events. Instead, linux executes as a CBS thread with its properties. This is made in the `make_linux_task_cbs_server` function:

```
int my_handler(unsigned int fifo){
    int err, event;

    while ((err = rtf_get(AUX_FIFO, &event, sizeof(event)))
            == sizeof(event)) {

 make_linux_task_cbs_server(gethrtime(),
    sched_attrib[4].budget,
    sched_attrib[4].deadline,
    sched_attrib[4].period,
    sched_attrib[4].params.sched_priority);
    }
    if (err != 0) {
 return -EINVAL;
    }
    return 0;
}
```

The parameters of the linux task when executing as CBS server are shown in Table 9-4, *CBS thread parameters for test 4 (in miliseconds)*.

Task id	Deadline	Period	Prio level	Max. budget

Table 9-4. CBS thread parameters for test 4 (in miliseconds)

Task id	Deadline	Period	Prio level	Max. budget
Linux CBS	3	3	10	0.2

The chronogram is presented in Figure 9-3. The chronogram generated by the execution draws the Linux task in a different position depending on wether it is scheduled as the task in background (task 0), or as a CBS thread (task 5).

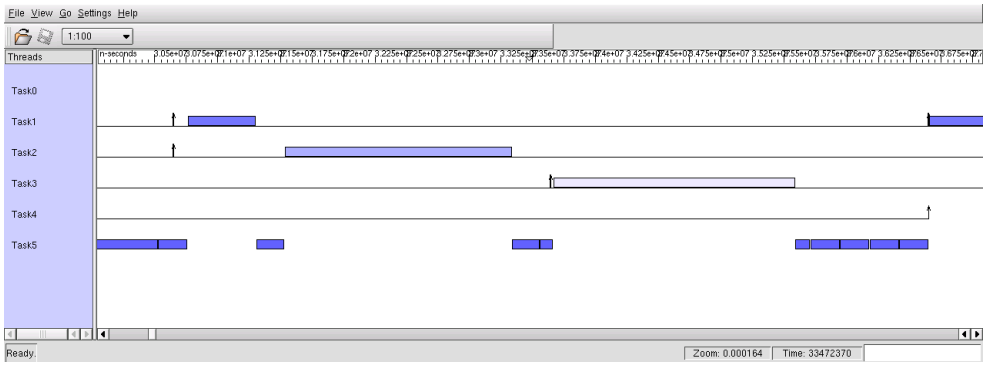


Figure 9-3. Chronogram execution for test 3

9.8.2.4. Test 4. Linux thread serves aperiodic jobs

This test is similar to the previous one, it only changes the way of detecting aperiodic events. Now, aperiodic events are parametrized interrupts. Therefore, there is no user application file (CBS_app). By default, the interrupt detected is the keyboard, but network interrupts or other kind of interrupts can be easily detected, with small modifications of the code. When an keyboard interrupt occurs, the interrupt handler attach CBS properties to the linux task by means of the `make_linux_task_cbs_server` function:

```
unsigned my_keyboard_interrupt_handler(unsigned int irq,
        struct pt_regs *regs){
    make_linux_task_cbs_server(gethrtime(),
        sched_attrib[4].budget,
        sched_attrib[4].deadline,
        sched_attrib[4].period,
        sched_attrib[4].params.sched_priority);
    rtl_global_pend_irq(KEYBOARD_INTERRUPT);
    return 0;
}
```

The parameters of the linux task when executing as CBS server are shown in Table 9-5, *CBS thread parameters for test 4 (in miliseconds)*.

Table 9-5. CBS thread parameters for test 4 (in miliseconds)

Task id	Deadline	Period	Prio level	Max. budget
Linux CBS	3	3	10	0.5

The chronogram is presented in Figure 9-4. The chronogram generated by the execution draws the Linux task in different position depending on wether it is scheduled as the task in background (task 0), or as a CBS thread (task 5).

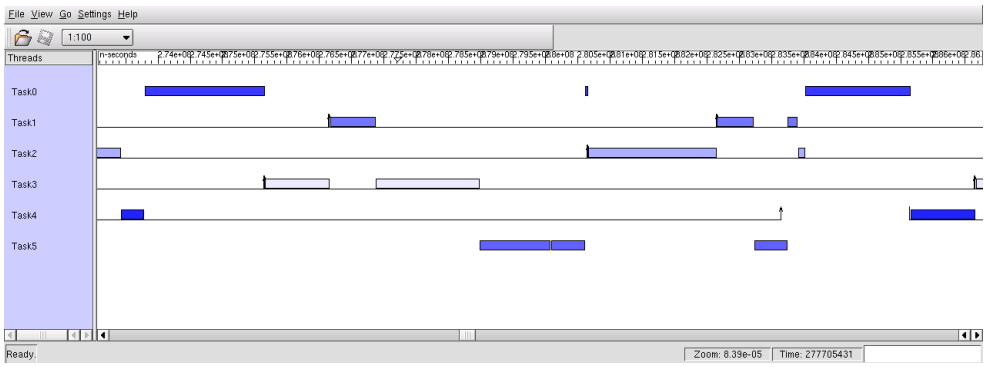


Figure 9-4. Chronogram execution for test 4

9.9. Installation instructions

To install the cbs scheduling support in rtlinux, you should:

- Install the rtlcbs ocera component in your source code:
  - Change directory to the component main one: '**cd rtlcbs_directory**'
  - Edit the file 'Makefile' and set your rtlinux and linux source directory, for example:
    "RTLINUX = /usr/src/rtlinux-3.2-pre1"
    "LINUX = /usr/src/linux-2.4.18"

  - Type **make install** in the main directory of the rtlcbs component.

  This will install the component: Copying the documentation, the examples. If necessary, (if it is not installed yet) will install the rtledf features patching your rtlinux with a retailed version. And will patch your source code to support the cbs scheduling.

- After this, you must rebuild your kernel image and your RTLinux modules as usual:
  - Rebuilding the kernel image:
    cd /usr/src/linux-2.4.18
    make clean dep
    make {your_image_type}
    Install it (i.e. with lilo)

  - Rebulding your RTLinux:
    cd /usr/src/rtlinux-3.2-pre1
    make clean
    make
    make modules_install

  - And finally, reboot your system using the new kernel.

Bibliography

[Aldea02] Mario Aldea-Rivas and Michael González-Harbour, 2002, 14 th Euromicro Conference on Real-Time Systems (ECRTS'02), *POSIX-Compatible Application-Defined Scheduling in MaRTE OS*.

[Abeni98] Luca Abeni and Giorgio Buttazzo, IEEE Real-Time Systems Symposium, Madrid, Spain, 1998, *Integrating Multimedia Applications in Hard Real-Time Systems*.

[Sha90] L. Sha, R. Rajkumar, and J.P. Lehoczky, 1990, IEEE Trans. on Computers, 39, 1175-1185, *Priority Inheritance Protocols: An Approach to Real-Time Synchronisation*.

[Baker91] T.P. Baker, 1991, The Journal of Real-Time Systems, 3, 67-100, *Stack-Based Scheduling of Realtime Processes*.

[Liu73] C.L. Liu and J.W Layland, 1973, Journal of the ACM, *Scheduling algorithms for multiprogramming in a hard real-time environment*.

[PTS] *Posix Test Suite*.

[BPA] *Bigphysarea*.

[Balbastre] P. Balbastre and I. Ripoll, 2001, Real time Workshop, *Integrated dynamic priority scheduler for rtlinux*.

[Baruah90] S. Baruah, A. Mok, and L. Rosier, 1990, IEEE Real-Time Systems Symposium, 182-190, *Preemptively scheduling hard real-time sporadic tasks on one processor*.

[Ripoll96] I. Ripoll, A. Crespo, and A. Mok, 1996, Journal of Real-Time Systems, 11, 19-40, *Improvement in feasibility testing for real-time tasks*.

[Stankovic] A. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, 1998, Kluwer Academic Publishers, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*.

[Caccamo01] Marco Caccamo and Lui Sha, IEEE Real-Time Systems Symposium, London, England, 2001, *Aperiodic Servers with Resource Constrains*.

[Paul 95] Paul R. Winson, Mark S. Johnstone, Michael Neely, and David Boles, 1995, Proc. Int. Workshop on Memory Management, *Dynamic Storage Allocation: A Survey and Critical Review*.

[Ogasawara 95] Takeshi Ogasawara, 1995, 2nd International Workshop on Real-Time Computing Systems and Applications, *An Algorithm with Constant Execution Time for Dynamic Storage Allocation*.

[Ada95] T. Taft, R. Duff, R. Brukardt, and E. Ploedereder, 2000, Springer, Lecture Notes on Computer Science, *Consolidated Ada Reference Manual*, 3-540-43038-5.

[Baker00] T. Baker, 2000, SIGAda Conference, *Ada and Embedded Real Time Linux*.

[Lepreau02] L. Lepreau and M. Flatt, University of Utah, 2002, *The Oskit Project*.

[Aldea01] Mario Aldea-Rivas and Michael González-Harbour, 2001, Ada-Europe, *MaRTE OS: An Ada Kernel for Real-Time Embedded Applications.* .