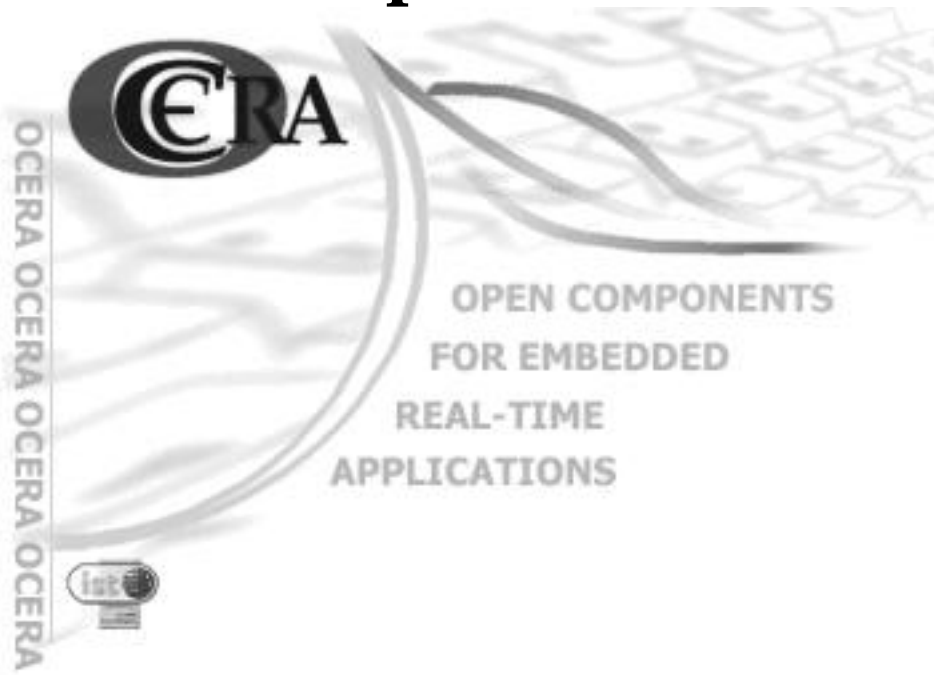
WP5 - Real-Time Scheduling Components



Deliverable D5.1 - Design of Scheduling Components

WP5 - Real-Time Scheduling Components : Deliverable D5.1 - Design of Scheduling Components

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

Published January 2002

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. Introduction

The objective of this work-package is to design and implement the basic scheduling components to build real-time systems.

This work-package is divided in two phases. The first phase is focused on implementing scheduling algorithms and providing new facilities to control timing violations.

Although Linux and RTLinux are systems that provide an API close to the POSIX standard, both still lack some important facilities needed in real-time systems. For example, RTLinux do not implement the basic UNIX signals mechanism, or the POSIX timers facility. An important effort will be done to overcome these basic requirements, and some of the OCERA components that are going to be developed are not of new research interest but are key features needed to provide a powerful and usable RTOS. Without this basic infrastructure, it will be not possible to develop new functionalities. Since the main objective of the OCERA project is to provide flexible and portable components, it is mandatory to design the new components taking as starting point a POSIX compatible RTOS.

Each component is described in a separate chapter. The information of each component is organised as follows:

Description

A detailed description of the component, and a rationale about its role and interest in a real-time operating system.

Some components implement well known facilities which are available in most RTOS but not in RTLinux or Linux. These cases do not need special justification nor description and these sections may be shorter.

Layer

As described in the "OCERA Architecture and Component Definition" document, both Linux and RTLinux can be logically divided in three sub-layers: **Low-level**, **High-level** and **Application**.

This section will present a description of the layer/s where the component is located.

API / Compatibility

The new API provided by the component if applicable. Also the compatibility with the standards.

Dependencies

Most components are not stand-alone code, but depend on some specific environment. For example, the version of the compiler, RTLinux, Linux, other components, hardware, etc.

Status

Each component is in a different development stage, some components are almost finished (that is the code has been released) while others are being designed.

Implementation issues

Key aspects of the internal design of the component will be presented in this section.

Tests

All the code produced in the OCERA project must be intensively tested and validated. All the functionality must be validated both reviewing the code by external developers (other partner); and also by building a complete test suite.

Validation criteria

A description of the expected results we try to achieve with the component. It also will be commented in this section the overhead introduced by the new code, if any.

Chapter 2. POSIX Signals

2.1. Description

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

- Exception handling.
- Task notification of asynchronous event occurrence (timer expiration,...)
- Emulation of multitasking.
- Interprocess communication.

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.

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 thread.

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

2.2. Layer

Signals are part of the core RTLinux executive, therefore they are located at the Low-Level RTLinux layer.

2.3. API / Compatibility

The following synopsis presents 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);

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
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);
int sigsuspend(const rtl_sigset_t *sigmask);
int sigpending(rtl_sigset_t *set);
```

2.4. Dependencies

RTLinux-3.1. Also tested on RTLinux 3.2-pre1.

2.5. Status

Released. All tests passed. This component is being considered to be included in the next release of RTLinux.

2.6. 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:

- **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 done to run 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 it was interrupted.

  Our implementation differs from the POSIX standard in which the default handler doesn't kill the thread, but 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 exits. Finally, signal handlers functions are executed always with interrupts enabled.

- **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 are 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 it is just a function call.

  With this OS structure, there are two possible ways to implement 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 has been tested, and the second one has been selected because it is more elegant, cleaner, secure, architecture independent and faster.

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

1. Stack overflow when sending RTL_SIGNAL_SUSPEND signal.
2. Infinite loop after calling `pthread_mutex_unlock()` (due to setting thread's structure aborts field to zero in `do_signal()`).

2.7. Tests

A total of twenty different tests have been implemented to check the functionalities. 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 handlers (calling the scheduler from signal handlers).
- 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.

2.8. Validation criteria

Increases RTLinux POSIX compatibility, since signals are required in all POSIX/UNIX systems. 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.

The overhead of the implementation is minimal. One of the previous tests showed that two threads sending each other signals can send (and handle) 209 signals per millisecond (in a PIII 500MHZ with APIC).

Chapter 3. POSIX Timers

3.1. Descritpion

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.

3.2. Layer

An efficient implementation of the timers has to be done at the lowest kernel level, therefore this component is located at the Low-Level RTLinux layer. It deeply depends on the scheduler and signal code.

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.

```
int timer_create(clockid_t clockid, struct sigevent
          *restrict evp,  timer_t *restrict timer_id);

int timer_delete(timer_t *timer_id);

int timer_settime(timer_t timer_id, int flags,
           const struct itimerspec *new_setting,
           struct itimerspec *old_setting);

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_real-time`.

3.4. Dependencies

RTLinux-3.1 and POSIX signals component.

3.5. Status

Released. All tests passed. This component is being considered to be included in the following release of RTLinux.

3.6. Implementation issues

In RTLinux timer_t type is implemented as a pointer to the `timer_struct`. 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.

Note: during the implementation two bugs were found in the RTLinux code, on functions `usleep()` and `timespec_add_ns()`.

3.7. Tests

We have used several tests sets to validate the component.

□ **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 Res Timers**: slightly modified to run on RTLinux.

3.8. 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.

Chapter 4. POSIX barriers

4.1. Description

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. A barrier ensures that all processes in a group are ready before any of them proceed.

4.2. Layer

It is a high-level RTLinux component, since it do not modifies the existing RTLinux kernel, but adds functionalities to it.

4.3. API / Compatibility

The API is defined by the POSIX standard. Here is a list of the functions that are going to be implemented.

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

4.4. Dependencies

It has no dependencies on other Ocera components.

4.5. Implementation issues

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

4.6. Status

This component is still being developed.

4.7. Tests

In order to test the correct behaviour of Posix Barriers, some tests will be provided with the component. Some debugging code will be inserted to validate the component. 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 its first activation at the first time. 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 chronogram generated must prove how all threads stop before making periodic and are active at the same time.

It also can be implemented some tests to use barriers with client-server systems. For example, as a simple test one thread can write on a fifo whenever other thread wants. This can be implemented by means of a barrier. When the client enters the barrier, then the other thread can follow its execution that consists of a rt_fifo writing operation. The results can be observed in the console.

4.8. Validation criteria

The proposed implementation will relay on special processor instructions to achieve low overhead, instead of using mutexes to synchronise processes.

Chapter 5. POSIX Trace

5.1. Description

As realtime applications become more complex, the availability of event tracing mechanisms becomes more important in order to perform run-time monitoring. Recently, IEEE has introduced 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 tool, the tracing component jointly with the application-defined scheduler component are the key tools to build fault-tolerance mechanisms.

5.2. Layer

Low-level RTLinux. It is distributed as a patch.

5.3. API / Compatibility

The ptrace component supports all the Trace and the Trace Event Filter options defined in the standard, subject to some minor changes and limitations. Overall, this 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 can not support the Trace Inheritance option. Also, the Trace Log option has not been implemented due to the lack of permanent storage subsystem.

The following is the full list of the functions corresponding to the TCP which are supported (this particular subset of the trace functions is represented as (TC) in Figure 5-1):

```
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);
```

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 RTL-PTtrace system 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 full list of functions available for the (TAP) (see Figure 5-1) at either of the application levels 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);
```

5.4. Dependencies

RTLinux-3.1.

5.5. Status

First version (1.0) released. It is being considered to be included in the RTLinux-3.2 release.

5.6. 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 applications s RTLinux side and Linux side. If some trace support has to be given to the two sides, the RTL-PT system has to be present at both. Therefore, we have implemented RTL-PT as two cooperating trace subsystems, one at each level:

a. **RTLinux trace subsystem**. The RTL-PT support at the RTLinux level has been integrated into the RT-Linux scheduler (module `rtl_scheduler.o`). The trace support is then always available to the RTLinux application, whether or not the ap-

plication wants to use it. Nevertheless, its overhead in the case of not using it is practically null.

All the data structures necessary to keep the entire tracing status are created and managed inside this module. These data structures include among other: control information, the set of event types registered for the target and all the active streams with its currently stored events.

b. **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 (see below) did not actually require such ambitious implementation. As a result, the decision was to implement this subsystem as a library to be linked with any Linux process that required trace support (called libposix trace.a). This library is made available in Linux when RTLinux (with the RTL-PT system) 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.

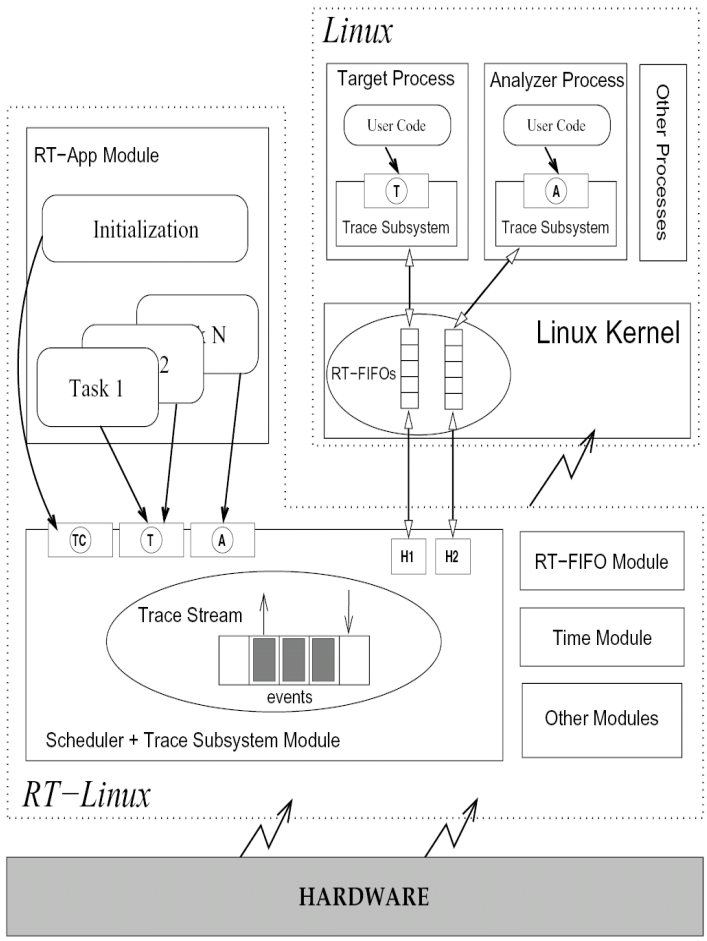


Figure 5-1. POSIX trace systems overview.

5.7. Tests

Although several simple functionality tests has been developed, the best way to test this component is by using it. The POSIX trace was successfully used during the development of the RTLGnat component.

5.8. Validation criteria

The main overhead introduced by the trace system is due to the amount of main memory used to store the logged data. There is little margin to improve the implementation. Some preliminary results show that the overhead of tracing an event is in the range from 100 to 500 nanoseconds.

Chapter 6. Application-defined Scheduler

6.1. Descritpion

Application-defined scheduling (ADS for short) 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.

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, 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 enabled 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.

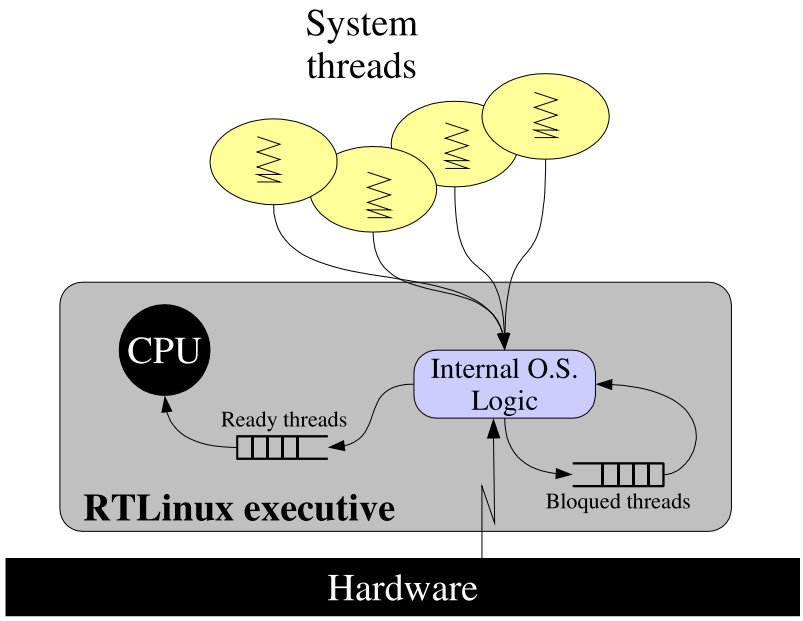


Figure 6-1. Conventional O.S. operation

Figure 6-1 shows a simplified schema of how the scheduling subsystem of a conventional operating system is internally designed. The internal scheduling algorithm (labelled 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).

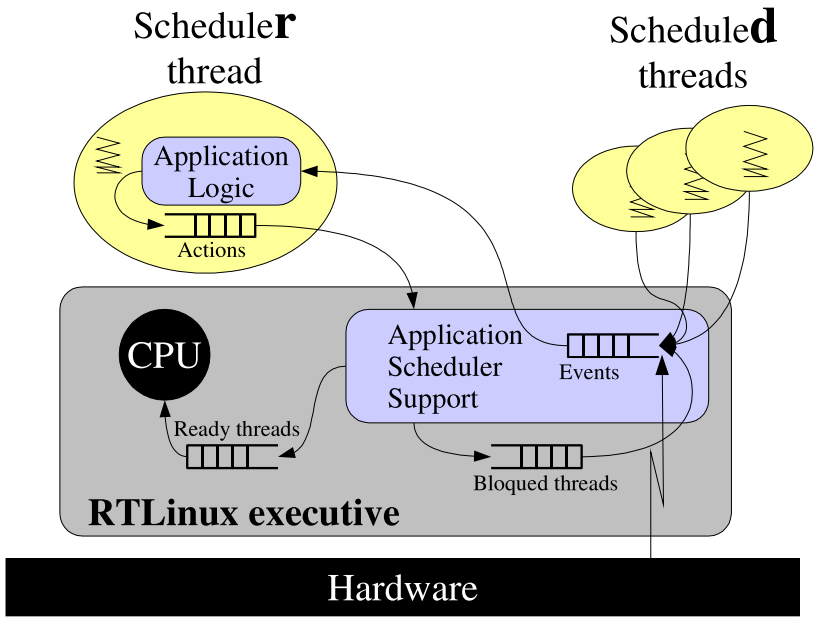


Figure 6-2. Application-defined scheduler O.S. operation

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 6-2 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 different 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.

6.2. Layer

This component is located at the low level RTLinux layer. The core scheduler was modified to intercept the required events.

6.3. 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_appschedattr_seteventmask (const posix_appsched_eventset_t *set);
int posix_appschedattr_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_t thread, 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);
```

6.4. Dependencies

This component depends on RTLinux 3.1, POSIX signals and Posix timers components.

6.5. Status

It is a beta version. All the examples implemented in the MaRTE OS has been successfully compiled and run in RTLinux.

6.6. Implementation issues

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 optimisations do not jeopardise the standard API.

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 will include the suggested changes in a new API review.

6.7. Tests

The test suite provided consists of the implementation of some well-known scheduling policies using the application-defined scheduling API.

- Fixed priority preemptive scheduler: This is an implementation of RTLinux scheduler. This implementation allow us to check the correctness of implementation and measure the overhead.
- EDF (Earliest Deadline First scheduler) [Liu73].
- CBS (Constant Bandwith Server) [Abeni98].

And the following resource management protocols:

- PCP (Priority Ceiling Protocol) [Sha90].
- SRP (Shared Resource Protocol) [Baker91].

The following policies and protocols EDF , CBS & PCP have been ported from the MaRTE OS application-scheduling implementation, in order to check compatibility.

6.8. 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 (deppending 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).

Chapter 7. POSIX Message Queues

7.1. Description

UNIX systems offers several possibilities for interprocess communication: signals, pipes and fifos, 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 notify him on message arrivals. So such a process can be working on something else until the waited message arrives.

7.2. 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.

7.3. API / Compatibility

This components follows the POSIX API specification. 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 *)
```

7.4. Dependencies

RTLinux 3.2. POSIX Signals are used if available.

7.5. Status

This component is in the first stages of development. An implementation of the basic functionality is currently under testing. The asynchronous notify functionality and the timed send and receive operations are still in its design stage, due to its dependencies with POSIX signals.

7.6. Implementation issues

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

Two main issues has to be considered when implementing POSIX.1 signals:

- **Queue creation**: When a message queue is created, using the `mq_open` function, the required information about queue and messages maximum size is available. Therefore, the maximum memory requirements for operation of the message queue are known and the resource reservation can be performed.

  As RTLinux has no dynamic memory support, message queues creation can be performed only when the module is loaded into the kernel. In this instant, Linux kernel `malloc` is available for dynamic memory reservation and then `mq_open` function can be implemented without problems.

  As soon as the dynamic memory component will be available, a less restrictive implementation of the `mq_open` function can be performed.

- **Sorting of the prioritised messages**: POSIX Message Queues standard requires that a receive operation always obtains as a result 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.

  Several possibilities arise when this sorting mechanism is analysed. The different options that has been analysed for the implementation of this component 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 provides low memory requirements, but each insertion and extraction from the message queue will have a logarithmic computational cost.
  - Use a sorted queue that allocates only one token per priority, sorting the queue only by the priority value. Each token should have a pointer to a FIFO queue that represents the pending messages of that priority. This implementation provides a constant computational cost for insertions and extractions, when the queue of a given priority is not empty, i.e., this solution optimises the FIFO access to the queue. The memory requirements of this solution are probably 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).

All these possibilities will be analysed carefully. Probably, all this solutions will be available as configuration options in the final component. At this moment, the second one has been selected as the basic sorting mechanism.

7.7. Tests

The basic conformance tests have been started and there are still no related results. Open POSIX Test Suite [PTS] will be used if available, now the message queue facility tests are under development.

7.8. Validation criteria

POSIX Message queue performance strongly relies on the performance of synchronisation mechanisms and the system memory bandwidth.

Chapter 8. Dynamic Memory Allocator

8.1. Descritpion

RTLinux provides support for neither dynamic memory management nor virtual memory. This component provides the basic `malloc`, `free` and `remalloc` library functions.

This component meet the realtime requirements, that is, it will have a bounded and predictable worst case response time. Other, design guidelines are:

- Low internal fragmentation.
- None external fragmentation.
- Immediate coalescing.
- Definable minimum block size.
- Definable splitting threshold.

The new proposed (and implemented) allocation algorithm is called Doubly Indexed Dynamic Memory Allocator (DIDMA).

8.2. Layer

Originally is was designed as a high-level RTLinux component, but it can be easily ported to be used in the high-level Linux layer to replace the non-realtime `glibc` dynamic memory implementation.

8.3. 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 nelem, size_t elem_size)
void *rt_realloc (void *p, size_t new_len)
```

8.4. Dependencies

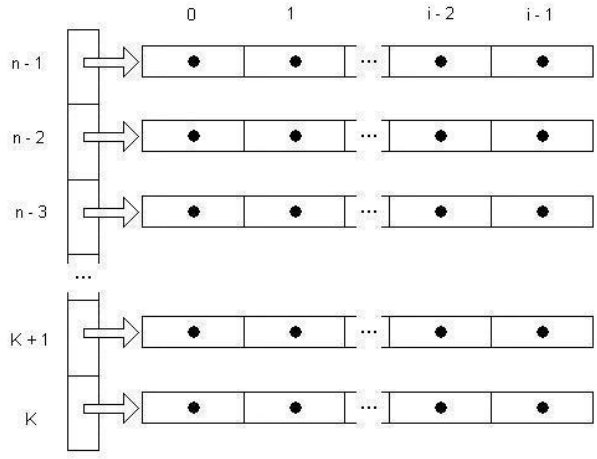
None. If the BigPhysArea[BPA] patch is installed then it can be used.

8.5. Status

The current version of DIDMA is beta 0.2.

8.6. Implementation issues

DIDMA uses an indexed strategy with a fixed size preallocated data structure. The structure used by DIDMA is implemented as two arrays indexed by two indexes.



$$(K = \text{MIN\_LOG2\_SIZE\_BLOCK}) \wedge (K \leq n < FIRST\_INDEX) \wedge (0 \leq i < SECOND\_INDEX)$$

Figure 8-1. DIDMA data structure

Mapping functions are used to quickly find the list of the required block size. The mapping functions are optimised to use fast numeric functions (shifts, adds, 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. A guide to select the best valued of these parameters will be provided with the final version of this component.

8.7. 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: For the initialization, the second test reserves 1000 blocks and later it free all the blocks except the last block reserved, afterwards the second test calls 1000 times `malloc` with increasing values, and later 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 initialization has finished, its behaviour is like the first test but `malloc` is called with random values.

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.

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

8.8. Validation criteria

The worst case temporal complexity of the proposed algorithm (DIDMA) has to be

bounded and it is data independent. The results of the tests (benchmarks) previously presented show that DIDMA has better response time than the following studied algorithms:

□ Binary Buddy

□ RTAI dynamic memory allocator

□ RTEMS dynamic memory allocator

□ Best fit

□ First fit

Here are some **preliminary results**, running on an Athlon XP 2000 with 512 Mbytes of RAM. The DIDMA algorithm parameters has been customised to First_Level_Index = 24 (with this value the size of the memory pool will be 16 MBytes) and Second_Level_Index = 16 (the second level index has importance on the internal fragmentation), which are reasonable default values.

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 microseconds:

Table 8-1. Test 1 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	85,6/160	161/176	59/82	80/88	109/145	110/134
Typical deviation	18/31	64/46	16/15	21/14	18/17	18/13
Variance	337/1009	4154/2140	269/253	471/201	344/293	326/177
Maximum	416/416	1120/544	416/128	544/160	416/256	384/224
Minimum	64/128	96/128	32/64	64/64	96/128	96/128

Table 8-2. Test 2 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	221/194	Failed	Failed	81/111	114/191	137/163
Typical deviation	107/20	Failed	Failed	15/19	15/42	15/16
Variance	11595/420	Failed	Failed	254/370	255/1775	240/262
Maximum	384/416	Failed	Failed	96/416	160/480	160/384
Minimum	64/160	Failed	Failed	64/64	96/128	96/128

Table 8-3. Test 3 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	343/179	175/289	1172/154	1153/93	1627/186	136/154
Typical deviation	119/21	53/114	283/256	275/17	209/40	15/18
Variance	14335/478	2843/13018	80378/65611	76002/288	43881/1630	238/354

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Maximum	544/480	512/1312	1344/1472	1376/448	2560/512	192/416
Minimum	64/128	96/128	32/64	64/64	1504/128	96/128

Table 8-4. Test 4 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	186/175	142/172	94/112	168/89	1889/159	145/164
Typical deviation	78/27	28/34	32/23	303/15	495/33	17/23
Variance	6188/734	793/1216	1058/535	92289/232	245114/1155	290/551
Maximum	416/256	448/448	224/160	1600/128	14944/352	224/256
Minimum	64/96	96/128	32/64	64/64	1632/128	96/128

Table 8-5. Test 5 results

	DIDMA	Bin Buddy	RTAI	RTEMS	Best fit	First fit
Average	341/154	180/264	64/56	1176/68	1626/160	134/130
Typical deviation	119/17	52/110	11/13	279/12	209/38	14/16
Variance	14393/322	2744/12167	125/188	78232/154	44053/1477	206/263
Maximum	544/320	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 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.

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 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 9. RTL-Gnat Porting

9.1. Descritpion

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: a bug 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: simply and error prone syntax, weak 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. RTLGnat is a porting of GNAT Ada compiler that allows to write rtlinux modules in Ada language.

9.2. Layer

The modules create 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.

9.3. API / Compatibility

RTLGnat allows to load programs compiled in Ada 95 on RTLinux.

9.4. Dependencies

RTLGnat depends on the GNAT-3.14 compiler, GCC 2.8.1 and RTLinux-3.1.

9.5. Status

RTLGnat are on alpha status.

9.6. Implementation issues

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 runtime support.

On one side, the RTGL module exports the symbols (function and variable names) that are required by the Linux module loader (like `init_module`, `cleanup_module`, author, license and kernel version strings); and on the other side RTGL provides to the GNAT runtime support the required, but not provided by RTLinux, OS API functions like for example the `malloc` and `free`.

Some of these functions of the RTGL has been taken from the OSKIT project and from the source code of GNU GCC 2.8.1. exported symbols.

9.7. Tests

Tests programs has been developed to check the following functionalities:

- Simple sequencial programs.
- Delays.
- Exceptions.
- Tasks.
- Protected objects, ceiling_locking and dynamic priorities.

9.8. Validation criteria

Quantitatives

- Currently several test are being developed to obtain maximum achievable utilisation of an harmonic task set.
- Some preliminary results show that the worst observed overhead in a task switch has been 20 microseconds.

Qualitatives

- Posix Tracer in RTLinux, implemented by Andres Terrasa, has been used to verify the operation of RTLGnat, and all the tests have been passed.

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*.