CLEOPATRE: A R&D project for providing new real-time functionalities to Linux/RTAI

Thibault GARCIA, Audrey MARCHAND and Maryline SILLY-CHETTO
IRIN (Institut de Recherche en Informatique de Nantes)
La Chantrerie BP 50609 Nantes Cedex 03 France
{maryline.chetto,audrey.marchand,thibault.garcia}@iut-nantes.univ-nantes.fr

Abstract

This paper is concerned with free software components dedicated to embedded real-time systems. We are providing a set of components including dynamic schedulers based on the Earliest Deadline algorithm and Fault-tolerance mechanisms based on time redundancy. All these components are designed to be integrated in the open source real-time operating system, Linux/RTAI.

1 Introduction

In this paper, we describe a work[1] which is about enhancing Linux/RTAI (Real-Time Application Interface) by adding new functionalities, such as dynamic priority schedulers, resource allocation protocols and aperiodic task servers to the original kernel. In particular, a version of the Earliest Deadline First scheduler, the Priority Inheritance protocol and aperiodic task servers have been implemented. These functionalities are selectable in a library of modules which can be loaded depending of the targeted application and the acquisitiveness of the user. Furthermore, tools have been developed in order to debug and test the implementation by providing information in a textual form[1].

2 Cleopatre project

This work is part of a French national project, Cleopatre (Software Open Components on the Shelf for Embedded Real-Time Applications)[2]. The objective of this project is first to create a library of free software modules for the design of real-time operating systems and second, to participate in the evolution of an opened community standard, Linux. The key objective is as well to demonstrate the applicability and the interoperability of these software components by tests on a mobile robotic platform (an Automated Guided Vehicle). Work is done thanks to the collaboration of final users and research teams that have skills in real-time technology including scheduling and fault-tolerance, communication, real-time vision and control. Innovations in all these aspects have a direct impact on both the performance and the reliability of every embedded system.

3 RTAI Internals Presentation

3.1 RTAI design basics

RTAI is an open-source real-time operating system based on the standard Linux kernel. It basically consists in an interrupt dispatcher: RTAI captures the interrupts from devices and if necessary, redirects them to Linux[3]. In other words, RTAI uses the RTHAL (Real-Time Hardware Abstraction Layer) concept described on the figure below:

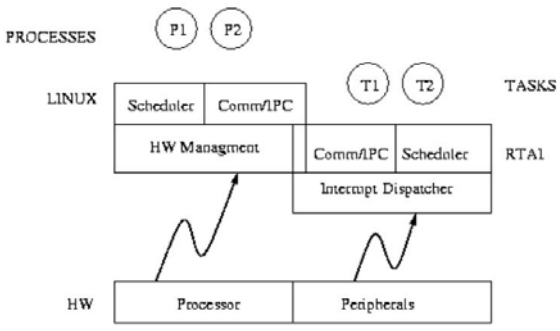

FIGURE 1: *RTAI architecture*

[1] work supported by the French research office, grant number 01 K 0742.

RTAI is a patch for the Linux kernel which provides the ability to make it fully pre-emptable (tasks may be suspended and later resumed). RTAI considers Linux as a background task which is carried out when no real-time activity is present.

3.2 RTAI modular facilities

RTAI is composed of different modules. The main module which has to be loaded for any application, comprises basic functions. For more complex developments, RTAI offers scheduling, FIFOs and shared memory facilities. RTAI includes three different priority based, pre-emptive real-time schedulers. We point out here that adding new schedulers based on dynamic priority is one of the investigating fields of the Cleopatre project.
In terms of performance, RTAI is very competitive, offering typical context switch times of 4μs, 20μs interrupt response, 100KHz periodic tasks, and 30KHz one-shot task rates.

4 Architecture of the operating system

4.1 Global description

The Cleopatre schedulers are modular and accept unusual research functionalities. The basic Linux command 'insmod' loads the modules in memory, places them into the Linux kernel and builds dependencies between the modules, using the names of all the functions involved in the application. The different modules are completely independent but need a low-level layer interface. This is a modular and flexible middleware. The application software can directly access to the low-level interface and use any intermediary module it requires. Every module can be replaced by another one which has the same interface.

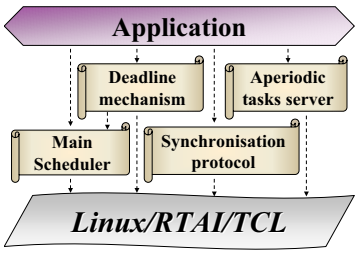

FIGURE 2: *Global architecture*

One important objective of the project is to build components as generic as possible. That's why we isolated the links with the operating system, concerning task management (creation and destruction), context switches and interruptions. Indeed, these points cannot be generic because they directly depend on both the choices made during the operating system implementation and the hardware architecture. RTAI requires, for each task, the stack size, the heap size and a function that signals task preemption. In general, operating systems do not need these data because they dynamically manage the stack and do not use signal functions.
The application software is the only program that can fill these specific data, but intermediary modules must be able to create tasks by themselves. The question is: how to build an intermediary module which is independent of the operating system? The specificities of the operating system are gathered in a special structure called TCLCreateType.

4.2 The Task Control Layer (TCL)

TCL updates the system clock, elects a task according to the dynamic priority parameter and provides a basic interface to the scheduling components of the system. TCL is at the root of the system architecture.

Primitives	Description
TCL_CREATE	create a new task
TCL_DESTROY	destroy a task
TCL_KILL	reset a task
TCL_READY	set a task ready
TCL_BLOCK	block a task
TCL_SCHEDULE	make preemption

TABLE 1: *Interface of TCL*

Combined with the TCLCreateType structure, TCL interfaces Linux/RTAI, so that the upper components are completely independent of the operating system. To adapt the Cleopatre scheduling system to other real-time operating system such as RT-Linux or others versions of Linux/RTAI, TCL has just to be exported.
TCL is the keystone of the portability because it was built to make the others components independent from the operating system, so it is the only component that should be modified to adapt the Cleopatre architecture to another real-time operating system.

4.3 Low-level mechanisms

To succeed in the integration process, the task control layer (TCL) considers the native RTAI scheduler as a background scheduler. So the native RTAI scheduler manages its own tasks, but these ones are scheduled when no Cleopatre activity is present. So, the scheduling system is compatible with the native RTAI applications.
A flexible architecture was built to use different and optional scheduling components on the same system. Components can register to access a ready task list. A static criticity parameter is associated to every ready task list. Tasks placed in the most critical list are always scheduled first. This allows us to manage different kinds of tasks, such as background tasks, normal tasks or/and safe tasks.
The TCL module manages ready tasks lists for all scheduling components and manages task election too. However, scheduling components choose the dynamic priority of the tasks according to their own protocols.
TCL manages a time handler sharing mechanism. The scheduling modules can register their own handler which is executed at every clock signal.

5 The scheduling components

Simulation was implemented to prepare the integration phase. A simulator tests the different scheduling algorithms in the same way but its interface depends on the given scheduling algorithm. A layer built simulator has been chosen. The intermediary layer establishes the link between the generic simulator interface and the different generic algorithms interfaces. Algorithms with the same interface can be simulated with the same intermediary layer. The low-level layer used for each simulation, generates random events that algorithms (the upper layer) should manage.

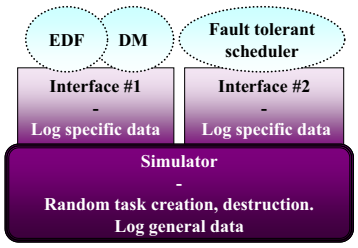

FIGURE 3: *Layer built simulator*

The simulator gives us a data report, so we can manually check all the lists each time an event occurs. This became really tedious, so the medium level was provided with a new function that checks the integrity of the actual sequence. It monitors many parameters such as the number of scheduled tasks, the number of task creation and destruction calls, etc.

6 Others components

6.1 Resource access control components

We developed two classes of resource management protocols respectively using mutex semaphores and generic semaphores. Each class is composed of several resources management components which use the same interface. So a component can easily replace another one of the same class. We created two versions of each protocol according to the order tasks get a resource after being blocked. This order depends on either dynamic task priorities or FIFO order.

Protocols	FIFO order	Priority order
None	FIFO	priority
Priority inheritance	F_PIP	P_PIP
Super priority	S_SPP	S_SPP

TABLE 2: *Resource access components*

Interfaces of the two classes of components are very similar to the 1003.1 POSIX standard.

Semaphores	Mutex	POSIX
SEM_CREATE	MTX_CREATE	SEM_CREATE
SEM_DESTROY	MTX_DESTROY	SEM_DELETE
SEM_TAKE	MTX_TAKE	SEM_WAIT
SEM_TAKE_IF	MTX_TAKE_IF	SEM_TRYWAIT
SEM_RELEASE	SEM_GIVE	SEM_POST

TABLE 3: *Interface of resource components*

6.2 Aperiodic task server components

Aperiodic task servers are very important for hard real-time systems, because they check if aperiodic tasks can be scheduled without involving timing faults. Three aperiodic task server components have been implemented.
The Background server schedules aperiodic tasks as background tasks and guarantees that aperiodic tasks will not affect periodic tasks.
The Total Bandwidth[4] server gives a virtual deadline to aperiodic tasks according to the processor load. Aperiodic tasks are then scheduled with periodic tasks. TBS was proved to be an optimal server.
When there is no aperiodic activity, the Earliest Deadline Late server[5][6] schedules periodic tasks according to the Earliest Deadline First protocol. Whenever an aperiodic task is released, it schedules periodic tasks as late as possible so as to recover maximum processor idle time to execute aperiodic tasks as soon as possible.

Servers	Components
BACKGROUND TASK SERVER	BGS
TOTAL BANDWIDTH SERVER	TBS
EARLIEST DEADLINE LATE	EDL

TABLE 4: *Aperiodic task server components*

Primitives	Actions
ATS_CREATE	Create tasks
ATS_BEGIN	Wake up all periodic tasks
ATS_DESTROY	Destroy tasks
ATS_WAKEUP	Wake up an aperiodic task

TABLE 5: *Aperiodic task server interface*

6.3 Fault-tolerance components

We implemented the Deadline Mechanism[7] in a specific component. This mechanism improves reliability by incorporating redundancy: each task is composed of two versions, a primary and a back-up version.

Primitives	Description
FT_CREATE	Create a new task
FT_DESTROY	Destroy all the tasks
FT_BEGIN	Wake up all the tasks together

TABLE 6: *Deadline Mechanism interface*

7 Tests and examples

7.1 Evaluation process

Cleopatre components were tested separately and together. A debugging mechanism is designed for developers of new components. It reports any modification of the ready tasks lists and can be precisely activated. This mechanism is very useful to validate components.
An independent laboratory is currently in charge of the performance measurement to value timing overheads.

7.2 Example

Let us assume two periodic tasks sharing a common resource. Period of task 1 is 0.5s and period of task 2 is 1s. Task 1 locks the resource, waits for its next occurrence and then, unlocks it. Task 2 tries to lock the semaphore during its execution and may be blocked. Tasks 3 and 4 are aperiodic and are released by an external interrupt signal. In order to simplify the example, the duration of every task is 4ms.

Task	Critical Delay	Period	WCET
1	16 ms	0.5 sec	4 ms
2	512 ms	1 sec	4 ms
3	8 ms	none	4 ms
4	12 ms	none	4 ms

TABLE 7: *Numerical values*

We used this application with two components combinations. First, we tested the application with the Deadline Monotonic[8] main scheduler (DM) and the Background Aperiodic task server (BGS).

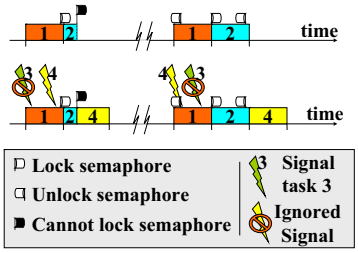

FIGURE 4: *Example: DM + BGS*

BGS never changes the periodic tasks sequence given by the main scheduler.
Then, we loaded the Earliest Deadline First[9] main scheduler (EDF) and the Total Bandwidth Aperiodic task server and we obtained better results.

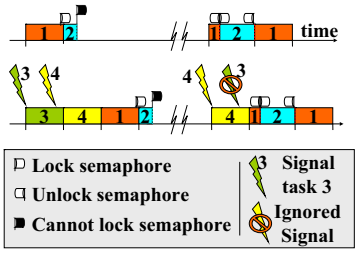

FIGURE 5: *Example: EDF + TBS*

7.3 Source code

```
struct AtsTaskType t1; /* Tasks descriptors */
struct AtsTaskType t2;
struct AtsTaskType t3;
struct AtsTaskType t4;

struct MtxType mtx;  /* Semaphore descriptor */

void fct1() {        /* -- Task 1 -- */
  simul_wait(4);     /* Simulating activity */
  mtx_take(&mtx);    /* Locking Semaphore */

  Dsch_wait();       /* Waiting next period */

  mtx_release(&mtx); /* Unlocking Semaphore */
  simul_wait(4);     /* Simulating activity */
}

void fct2() {        /* -- Task 2 -- */
  mtx_take(&mtx);    /* Locking Semaphore */
  simul_wait(4);     /* Simulating activity */
  mtx_release(&mtx); /* Unlocking Semaphore */
}

void fct3() { simul_wait(4); } /* Aperiodic */
void fct4() { simul_wait(4); } /*   tasks   */
```

```
int init_module(void) {
  struct TCLCreateType create = {0, 2000, 0, 0};

  mtx_create(&mtx);  /* Creating semaphore */

                     /* Creating tasks */
  ats_create(&t1, fct1,  16,  500, 4, create);
  ats_create(&t2, fct2, 512, 1000, 4, create);
  ats_create(&t3, fct3,   8,    0, 4, create);
  ats_create(&t4, fct4,  12,    0, 4, create);

  ats_begin(1000);   /* Releasing tasks after 1s */

  start_rt_timer(nano2count(TIMERTICKS));

  return 0;
}

void cleanup_module(void)
{
  stop_rt_timer();

  ats_destroy(&t1);  /* Destroying Tasks */
  ats_destroy(&t2);
  ats_destroy(&t3);
  ats_destroy(&t4);

  mtx_destroy(&mtx); /* Destroying Semaphore */
}
```

8 Conclusion and perspectives

We implemented a modular and flexible architecture which manages many kinds of components. This architecture is based on the Task Control Layer which interfaces the operating system. We are currently proving that the new components are independent from any operating system, thus adapting the Task Control Layer to the RT-Linux operating system without modifying any components.
At the same time, we are implementing more sophisticated resource management protocols such as Dynamic Priority Ceiling Protocol[10] and Stack Resource Policy[11]. An independent laboratory is in charge of time measurement and performance comparisons with the native Linux/RTAI.
The Cleopatre executive was integrated into a mobile robotic platform. The others partners of the project will add their own components to enhance the platform functionalities, thus implementing real-time visual recognition algorithms linked to navigation and robotic arm control.

References

[1] A. Marchand, C. Plot and M. Silly-Chetto, June 2002, *Real-time mobile robot navigation with*

LINUX/RTAI, Proceedings of Int. Conf. On Control and Automation, Xiamen, China.

[2] http://www.cleopatre-project.org/

[3] Lineo Inc., P. Mantegazza, 2000, *RTAI Programming Guide 1.0*, Lindon, Utah.

[4] M. Caccamo, G. Lipari, G. Buttazzo, December 1999, *Sharing Resources among Periodic and Aperiodic Tasks with Dynamic Deadlines*, Proceedings of the IEEE Real-Time Systems Symposium, Phoenix, Arizona.

[5] H. Chetto, M. Chetto-Silly, 1989, *Some results of earliest deadline scheduling algorithm*, IEEE Trans. on SW Eng., 18(8), pp.736-748.

[6] J.P. Lehoczky, L. Sacha, Y. Ding, 1992, *An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority pre-emptive systems*, Proceedings of the IEEE Real-time Systems Symposium, pp.110-123.

[7] A.L. Liestman, R.H. Campbell, 1986, *A fault tolerant scheduling problem*, IEEE Trans. on software engineering, vol.12 pp.1089-1095.

[8] J.Y.T. Leung, M.L. Merril, 1980, *A note on preemptive scheduling of periodic real-time tasks*, Information processing letters, vol. 20 n3 pp. 115-118.

[9] C. Liu, J.W. Layland, 1973, *Scheduling algorithms for multiprogramming in a hard real-time environment*, Journal of ACM, vol.20 n1 pp.46-61.

[10] M.I. Chen, K.J. Lin, 1990, *Dynamic Priority Ceilings: A Concurrency Control Protocols for Real-Time Systems*, Real-Time Systems Journal, 2(4), pp. 325-346.

[11] T.P. Baker, 1990, *Stack-Based Scheduling of real-time Processes*, Proceedings of IEEE Real-Time Systems Symp., pp.191-200.