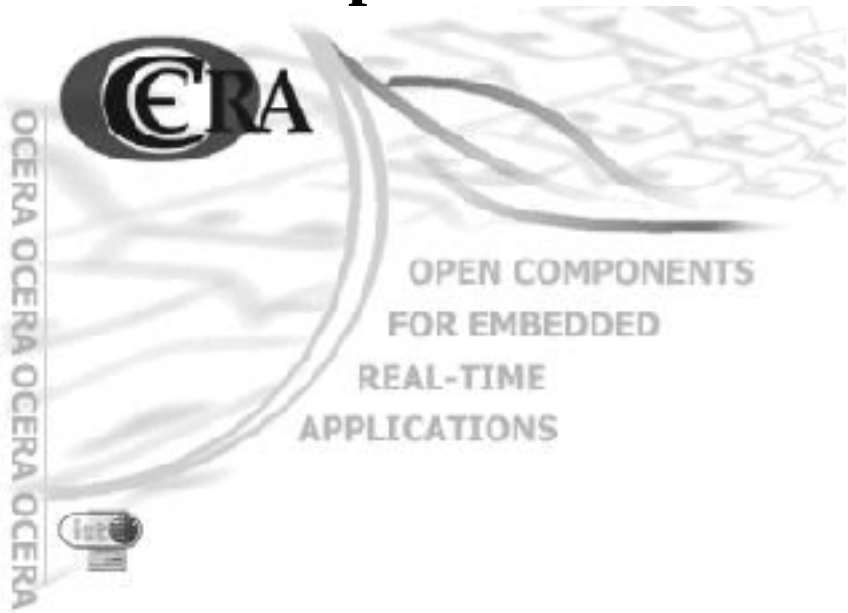
WP4 – Resource Management Components



Deliverable D4.2_rep
Resource Management Components V1

WP4 – Resource Management Components : Deliverable 4.2_rep – Resource management components V1
by Giuseppe Lipari, Luigi Palopoli, and Luca Marzario

Published April 2003

Table of Contents

Document Presentation

Project Coordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14, 46022 Valencia, Spain
Phone:	+34 963877576
Fax:	+34 963877576
Email:	alfons@disca.upv.es

Participant List

Role	Id.	Participant Name	Acronym	Country
CO	1	Universidad Politecnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA/DRT/LIST/DTSI	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	Date	Reason of change
1_0	15/04/03	First release

Chapter 1. Introduction

1.1 Summary of components

The components described in this deliverable were designed to support quality of service scheduling and management in the Linux kernel. They provide a framework for developing real-time applications with non-critical timing constraints on the Linux kernel.

A general description of the components, their structure, their objectives and the expected results can be found in Deliverable D4.1 ("*Design of Resource Management Components*"). In the first phase of the project, the following components have been developed and they are part of the first OCERA main release:

Generic Scheduler Patch. (**gensched**) It is a small patch for the Linux kernel that provides "hooks" for modules implementing generic scheduling policies. In our particular applications, it is used to implement the resource reservation scheduling module.

Integration patch. (**pcomp)** This component addresses important compatibility issues between Linux and RTLinux.

Resource Reservation Scheduling module (**qres**) It is a dynamically loadable module for the Linux kernel. It provides a resource reservation scheduler for real-time tasks running in user-space.

Quality of Service Manager (**qmgr**) It is a dynamically loadable module that provides a mechanism for identifying the temporal characteristics of a task and adjusts its scheduling parameters according to a feedback controller.

User API. (**qlib**) A library for supporting periodic threads in Linux, compatible with the RTLinux API.

These components are released for the first time (version 1.0) as part of the main OCERA release 1.0. In the next phase of the project we will improve all the components, by both performing extensive tests" and by adding new functionalities.

In the remainder of the deliverable, for component we will dedicate a chapter to describe its structure, how it was implemented and compatibility issues. We will also report the result of the tests and some performance measures related to the component, where applicable.

1.2 Reference software platform

As described in Deliverable D8.1 (Integration Plan), we decided to develop our components using Linux 2.4.18 and RTLinux 3.2-pre1.

Chapter 2. Generic Scheduler Patch

2.1.Summary

Name: **gensched**

Description: generic scheduler patch.

Author: Luca Abeni (luca@sssup.it)

Reviewer

Layer: Low level Linux components

Version: 1.0

Status: released (beta)

Dependencies: none

Release date: April 2003 (MS 2)

2.2 Description

The patch will export the following 6 hooks to the Linux scheduler:

the **fork_hook** is invoked when a new task is created;

the **cleanup_hook** is invoked when a task is destroyed;

the **block_hook** is invoked when a task blocks;

the **unblock_hook** is invoked when a task wakes up;

the **setsched_hook** and **getsched_hook** are invoked when the **sched_setsched**() or **sched_getsched**() system calls are called. If the hook function returns a negative value, the system call must fail; if it returns 0, the system call succeeds and returns a success value; whereas if the function returns a positive value the original Linux system call code is executed as a callback.

2.3 API / Compatibility

No new API is exported.

2.4 Implementation issues

The patch has been designed to be minimally invasive. Therefore, it only modifies a few points in the Linux scheduler, so as to maintain high level of compatibility and re-usability. We expect a small effort in porting the **gensched** patch to new versions of the Linux kernel.

However, there is an evident drawback, since it is difficult to implement complex scheduling policies by only using the **gensched** patch hooks. In particular, it is impossible to distinguish between the different blocking situations that a process may undergo. Indeed, the block_hook invoked by the Linux scheduler does not provide any information on the reason for the blocking. Therefore, we are not currently able to understand why the process has been blocked. We plan to improve this behaviour in a future version of the patch.

2.5 Tests and validation

2.5.1 Validation criteria

This patch has to be minimally invasive in order to limit the overhead and to minimize any dependency of the particular Linux version.

2.5.2 Test

All the basic scripts of the LTP package have been run, and all tests have been successfully passed.

Chapter 3. Integration Patch

3.1.Summary

Name **pcomp**

Description Compatibility between the preemption patch and the RTLinux patch

Author: Carlo Andrea Orrico (orrico@ocera.sourceforge.net)

Reviewer

Layer: Low Level Linux component

Version 1.0

Status: released (alpha)

Dependencies: it requires Linux 2.4.18 and RTLinux 3.2-pre1

Release date April 2003 (MS2)

3.2 Description

The aim of this component is to make two different patches to the Linux kernel, the Preemption Patch [PRK] and the RTLinux patch, work together. The RTLinux patch is necessary to make RTLinux work. The Preemption Patch, based upon Robert Love's Preemptible Kernel work, reduces the latency of Linux by allowing preemption to occur inside Linux. By using the Preemption Patch and the Low Latency Patch [LLP], the latency of many Linux operations is significantly reduced. Therefore, both patches are very important for soft real-time scheduling at the Linux user level. In the mixed OCERA configuration, both hard real-time with RTLinux and soft real-time with Linux are required at the same time. Therefore, we need to apply all three patches.

However, while the Low Latency Patch does not interfere with the RTLinux patch, both the Preemption Patch and the RTLinux patch modify the interrupt handling code. If the two patches are applied at the same time, Linux crashes.

Thus this component is yet another patch that combines the two patches, RTLinux patch and Preemption Patch, while retaining -to a reasonable extent- the advantages of both.

A summary of the patch description is offered next.

Preemption patch

The basic idea behind the preemption patches is to create opportunities for the scheduler to be run more often and minimize the time between the occurrence of an event and the running of the **schedule()** kernel function.

The preemption patch modifies the spin-lock macros and the interrupt return code so that if it is safe to preempt the current process and a rescheduling request is pending, the scheduler is called.

Originally, the Linux kernel code assumed that upon entry to the kernel, from a trap or from an interrupt, the current process would not be changed until the kernel decides that it is safe to reschedule. This was a simplifying assumption that allowed the kernel to modify its data structures without requiring the use of mutual exclusion primitives (such as spin-locks). Over time, the amount of code that modified kernel data structures without protecting them has been reduced, to the point where preemption patches assume that if the code being executed is not an interrupt handler and no spin-locks are being held, then it is safe to context switch away from the current thread context.

The preemption patches add a field to the task structure (the structure that maintains state for each thread) named **preempt_count**. The **preempt_count** field is modified by the macros **preempt_disable(), preempt_enable()** and **preempt_enable_no_resched()**.

The **preempt_disable()** macro increments the **preempt_count** variable, while the **preempt_enable()** macros decrement it. The **preempt_enable()** macro checks for a reschedule request by testing the value of **need_resched** and if it is true and the **preempt_count** variable is zero, calls the function **preempt_schedule()**. This function

notifies the occurrence of a preemption schedule by adding a large value (0x4000000) to the **preempt_count** variable,

calls the kernel **schedule()** function,

subtracts the value from **preempt_count**.

The scheduler has been modified to check **preempt_count** for this active flag, thus reducing the quantity of code executed by the preemption routine.

The macro **spin_lock()** was modified to first call **preempt_disable()**, then actually manipulate the spin-lock variable. The macro **spin_unlock()** was modified to manipulate the lock variable and then call **preempt_enable(),** and the macro **spin_trylock()** was modified to first call **preempt_disable()** and then call **preempt_enable()** if the lock was not acquired.

In addition to checking for preemption opportunities when releasing a spin-lock, the preemption patches also modify the interrupt return path code. This is assembly language code in the architecture specific directory of the kernel source (e.g. arch/arm or arch/mips) that makes the same test done by **preempt_enable()** and calls the **preempt_schedule()** routine if conditions are right.

The effect of the preemption patch modifications is to reduce the amount of time between when an wakeup occurs and sets the **need_resched** flag and when the scheduler may be run. Each time a spin-lock is released or an interrupt routine returns, there is an opportunity to reschedule.

Compatibility

Our patch contains a set of variations of the preemption patch by Robert Love that makes it suitable for RTLinux. Namely, we identified three different compatibility problems. The first problem is related to the mechanism used to access **preempt_count**. The second is related to the enable/disable interrupt functions. The last one is related to the interrupt handler. Finally, we modified RTLinux code in order to take advantage of the introduction of **preempt_count**. A more detailed description of the three variations is reported below.

Preemption Patch modifications

The **preempt_count,** introduced by the preemption patch, is modified by the three macros described above (see linux/include/linux/spinlock.h), by the irq controller related code (e.g. see linux/include/asm-i386/hw_irq.h) and by the interrupt return path code (e. g. see linux/arch/i386/kernel/entry.S).

Each time an IRQ is issued this counter is incremented to prevent preemption when running an interrupt handler. The counter is referred through the **GET_CURRENT** macro that relies on the value of the ESP processor register to get the address of the task structure of the current running Linux thread (e.g. see linux/include/asm-i386/hw_irq.h).

The first issue is that RTLinux real time tasks can modify the content of ESP, so when a real time task is running and an IRQ is issued the **GET_CURRENT** macro returns a wrong value and we can end up writing in some random location in memory.

Therefore, we have modified the **GET_CURRENT** code so when a real time task is running it always returns the correct value. To this and, we rely upon the **get_linux**_current function, which is a function defined in the RTLinux module rtl_sched.c (see rtlinux/schedulers/rtl_sched.c).

We use this function through a pointer defined only when the rtl_sched module is inserted. Therefore, in absence of real time modules, the call is perfomed on the old **GET_CURRENT** macro (see include/asm-i386/hw_irq.h).

Another problem was in the use of the interrupt disabling functions. Generally RTLinux does not allow Linux to make **cli** or **sti** hardware instructions except in a few places. One of this is inside the interrupt controller related code (see arch/i386/kernel/i8259.c), where RTLinux patch redefines the **spin_lock_irqsave** and the **spin_unlock_irqrestore** macros and allows Linux to disable/enable hardware interrupts.

Now those macros also calls **spin_lock/unlock** macros that, as stated above, can disable/ enable preemption and check if the scheduler has to be called. We have to inhibit calls to the scheduler while interrupts are disabled interrupts, since they could introduce long latencies for interrupts handling and real- time tasks execution. Therefore, we modified **spin_lock_irqsave** and **spin_unlock_irqrestore** and make them call **_raw_spin_lock** and **_raw_spin_unlock** macros that avoid preemptions and calls to the scheduler.

Finally the preemption patch modifies the interrupt return path code, where it puts a couple of **cli**, **sti** hardware instruction. We replaced them with two calls to software emulating functions pointed by the two fields of the **irq_control** structure (see linux/ arch/i386/kernel/rtlinux.c) that is dynamically patched by the RTLinux module. Before the insertion of RTLinux modules those calls are hardware calls, after the insertion became soft ones (see '**patch_kernel**' function in rtlinux/main/i386/arch.h).

RTLinux code modifications

Modifications to the RTLinux code are very simple. Inside RTLinux code we cannot call the Linux scheduler safely nor can we access to the **preempt_count** variable in the **current** task structure simply relying on the value of ESP.

When RTLinux code is running we simply do not want any preemption. We have modified the **rtl_spin_lock** and **rtl_spin_unlock** macros so they call **_raw_spin_lock** and **_raw_spin_unlock**, two other macros defined by the preemption patch that make no

calls to the scheduler and do not alter the **preempt_count** value. (see rtlinux/include/rtl_sync.h) For the same reason, we replaced all the calls to the **spin_lock/unlock** macros in the RTLinux code with the **rtl_spin_lock/unlock** ones.

The **preempt_count** variable in the **current** task structure is always incremented when an IRQ is issued but when the RTLinux module is inserted we can have a different return path from interrupt so the value of the counter should be adjusted properly, in particular we have to decrement that counter when returning from interrupt without passing through the standard Linux return path code. Moreover, when altering the counter, we have to refer the correct value of **current**; once again we rely on the **get_linux_current** function.

We also modified the interrupt handling code so, when RTLinux returns the control to Linux, pending interrupts can be served as soon as possible without waiting for another interrupt or a calls to the soft sti. (All code for interrupt handling in RTLinux is in rtlinux/main/rtl_core.c)

Finally, when pending interrupts are served, preemptions should be first disabled and then re-enabled. This happens every time Linux calls a soft sti (see rtl_soft_sti in rtlinux/main/rtl_core.c).

3.3 API / Compatibility

This component does not provide any API.

3.4 Tests and validation

3.4.1 Validation criteria

The validation criteria, as reported in deliverable D4.1, is an on/off, meaning that either the patch is correct and Linux does not crash, or it is not, and Linux crashes. In order to verify compliance with these criteria, we ran both the LTP [LTP] testing suite and the rt-Linux tests at the same time. We also ran some benchmarks (taken from LMBench [LMBench]) that stress the system to compute some performance index. During the test, Linux has never crashed and all tests have been successfully passed.

3.5 Installation instructions

1) Download an updated kernel version from the ocera distribution. It should be a Linux 2.4.18 with 'BigPhysArea' and 'LowLatency' patches already applied. The RTLinux version should be a 3.2pre1. Suppose you put these kernels in ocera/kernel/

2) Apply the RTLinux patch found in RTLinux /patches directory:

```
cd ocera/kernel/linux

patch -p1 < ../rtlinux/patches/kernel_patch-2.4.18-rtl3.2-pre1
```

3) Patch Linux kernel with patch-linux-rtl3.2-preemption

```
patch -p1 < ../../components/qos/pcomp/patch-linux-rtl3.2-preemption
```

4) Patch RT-Linux with patch-rtlinux-rtl3.2-preemption

```
cd ../rtlinux

patch -p1 < ../../components/qos/patch-rtlinux-rtl3.2-preemption
```

This version of RTLinux preemptable kernel is tested to work with RTLinux 3.2pre1 and with Linux kernel 2.4.18. It has been tested with 'Preemption', 'Bigphysarea' and 'LowLatency' patches enabled but on '386' uniprocessor only with no 'Symmetric multi-processing support' and no 'Local APIC support on uniprocessors'.

Chapter 4. Resource Reservation Scheduling module

4.1.Summary

Name: **qres**

Description: It is a dynamically loadable module for the Linux kernel, that provides a resource reservation scheduler for soft real-time tasks in user space.

Authors: Luca Abeni (luca@sssup.it), Luca Marzario (lukesky@gandalf.sssup.it), Claudio Scordino (scordino@gandalf.sssup.it)

Reviewer

Layer High Level Linux Component

Version 1.0

Status Beta

Dependencies Depends on the Generic Scheduler Patch (gen_sched: see Chapter 2). It may use the High Resolution Timer Patch. It may use the Preemption Compatibility Patch (pcomp: see Chapter 4)

Release date: Aprile 2003 (MS2)

4.2 Description

This component implements a resource reservation scheduler. The algorithm is based on the Constant Bandwidth Server (CBS) algorithm [Abe98]. However, we modified and extended the original algorithm to take into account several practical issues. In our implementation, a server can handle more than one task; an automatic reclamation mechanism [Lip00] can be optionally activated; bandwidths can be tightly bounded through a self suspension mechanism. All these features are currently available on a single software module and they can be enabled/disabled through conditional compilation (see Section 4.7 for more details on how to compile and install the component). In the next version (second phase of this workpackage), we will provide different custom modules available separately.

The complete algorithm is described in the following.

Common definitions and assumptions

A CBS S_i is described by: *the server budget* Q_i and the *server period* P_i. The *server bandwidth* $U_i = Q_i / P_i$ is the fraction of the CPU bandwidth reserved to S_i. To avoid inconsistencies and overload situations, the following condition must hold at all times:

$$\sum_{i}^{n} U_i \leq 1$$

Therefore, a variation in the parameters (like for example the budget) is allowed only at replenishment time, and only if the above condition is respected.

Dynamically, each server updates two variables (q_i, d_i) . Variable q_i is the *current budget* and keeps track of the consumed bandwidth. Variable d_i is the server's *scheduling deadline*. A server is *active* if any of its tasks has a pending instance and the current budget is greater than 0. If there a is pending instance and the current budget is 0, the server is *suspended* until the current budget is recharged. If at time *t* there is no pending task's instance and the current budget is greater than 0, the server is still *active* if $t < d_i - q_i P_i / Q_i$, otherwise it is *inactive.*

The system consists of *n* servers and a global scheduler based on the Earliest Deadline First (EDF) priority assignment. At each instant, the active server with the earliest scheduling deadline is selected and the corresponding task is dispatched to execute.

A task corresponds to a Linux thread. A task instance (or job) is activated (or *arrives*) just after it has been created with a fork() or with a pthread_create(), and when it is *unblocked*. A task instance finishes when the thread is *blocked*. In this version of the scheduler we do not distinguish between different blocking situations; every time a thread is blocked, the scheduler interprets it as the finishing of the job.

Basic CBS Algorithm

The algorithm's rules are the following.

1. Initially, $q_i = 0$, $d_i = 0$ and the server is *inactive*.

2. When a task is activated at time *t*, if the server is *inactive*, then $q_i = Q_i$ and $d_i = t + P_i$, and the server becomes *active*. If the server is already active, then q_i and d_i remain unchanged.

3. At any time *t*, the global scheduling algorithm selects the active server with the earliest deadline d_i . When the server is selected, it executes the first task in its ready queue (which may be order according to any policy).

4. If some of its tasks is executing for *x* units of time, the server's current budget q_i is decremented by the same amount *x*.

5. The global scheduler can *preempt* the server for executing another server: in this case, the current budget q_i is no longer decremented.

6. If $q_i = 0$ and some of the server tasks has not yet finished, then the server is *suspended* until time d_i ; at time d_i , q_i is recharged to Q_i , d_i is set to $d_i = d_i + P_i$ and the server can execute again.

7. When, at time *t*, the last task has finished executing and there is no other pending task in the server, the server yields to another server. Moreover, if $t \leqslant d_i - q_i P_i / Q_i$, the server becomes *inactive*; otherwise it remains *active*, and it will become *inactive*

at time $d_i - q_i Q_i / P_i$, unless another task is activated before.

Reclamation

An optional reclamation mechanism has been implemented. We choose Algorithm GRUB [Lip00], which does a greedy reclamation of the unused bandwidth, i.e. it gives all free bandwidth to the currently executing task. For this reason GRUB is not a fair algorithm. However, it can be implemented with little additional overhead, and it can also be used as a voltage scheduling algorithm, allowing energy saving.

The GRUB algorithm keeps track of a global variable $U_{act}(t)$ called *Total Bandwidth*, which is the sum of the bandwidth of all active servers at time *t*. Therefore, when a server becomes active (rule 2) its bandwidth is added to $U_{act}(t)$; when a server becomes inactive (rule 7), its bandwidth is decremented from $U_{act}(t)$. Finally, rule 4 is modified to take into account the reclamation:

4'. If If some of its tasks is executing for *x* units of time, the server's current budget q_i is decremented by $x U_{act}$.

A proof of the correctness of the GRUB algorithm can be found in [Lip00].

Energy saving (voltage scheduling)

The same algorithm can be used as a voltage scheduling algorithm. Suppose that the processor supports two different voltages, which results in two different processor speeds, the nominal speed s_{nom} and the low speed s_{low} , and suppose that the processor normally works at speed s_{nom} . We define a minimum bandwidth and two thresholds:

$$U_{low} = \frac{s_{low}}{s_{nom}} \qquad U_{th1} < U_{th2} < U_{low}$$

Then, we add two additional rules:

8. if U_{act} is below the first threshold U_{th1} , the speed of the processor is set to s_{low} ;

9. if U_{act} is greater than the second threshold U_{th2} , the speed of the processor is set to s_{nom} .

These additional rules are currently under implementation and test on a version of Linux that runs on a Intel PXA250 processor. Therefore, this feature it is only available as an alpha release.

4.3 API / Compatibility

The provided module does not introduce any new primitive. However, it modifies an existing primitive, the **sched_setscheduler**() standard Linux primitive, by allowing a new kind of scheduler to be selected by the user.

The standard interface of the **sched_setscheduler** primitive is the following:

```
int sched_setscheduler(pit_t pid, int policy,
                       const struct sched_param *p);
```

where pid is the id of the process for which the new scheduling strategy can be set; policy can be one of **SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER**; **p** is a pointer to a structure **sched_parms** that contains the scheduling parameters. The CBS module provides a new scheduling policy, called **SCHED_CBS**.

Thanks to the use the **gensched** patch (see Chapter 2.), **sched_parm** has an additional field **sched_p**, of type **(void *)**, which can point to a dedicated structure that holds our scheduling parameters for the CBS. The size of this structure must be put into another field of **sched_parms**, called **sched_size**.

The CBS dedicated structure is listed below:

```
struct cbs_param {
 unsigned long int signature;
 int period;
 int max_budget;
};
```

To set the CBS scheduler for a certain process, it is necessary to write the following lines of code:

```
struct sched_param sp;
struct cbs_param cs;

sp.sched_size = sizeof(struct cbs_param);
sp.sched_p = &cs;
cs.signature = CBS_SIGNATURE;
cs.max_budget = Q;
cs.period = T;

res = sched_setscheduler(getpid(), SCHED_CBS, &sp);
```

Notice that only the superuser can call the **sched_setscheduler()** primitive.

PROC interface

Also, an interface through the standard **/proc** file system is available. Currently, the interface is only meant for debugging purposes. It is possible to analyse the status of the scheduler by reading all important scheduling parameters for each task. The CBS module can optionally provide the following file in the standard Linux virtual file system:

```
/proc/cbs/scheduler
```

which can be read as a normal text file and contains one line for each task served by a CBS, in which all sensitive informations are listed. We are evaluating the overhead of using such interface to see if it would be possible to send basic commands to the scheduler through such interface.

4.4 Implementation issues

The **qres** module uses the generic scheduler patch to provide its services. In particular, when the module is loaded (see function **init_module** in **src/init.c**), the hook function pointers defined in the generic scheduler patch are assigned particular functions provided by the **qres** module. All the rules presented in Section 4.2 are implemented in such functions.

The CBS implements the EDF ready queue internally. Every task descriptor has a pointer to a structure (called **cbs_struct**, in file src/include/cbs.h) where the CBS puts its own task's related data. Among these data is the current budget, the period, the absolute deadline, and so on.

If the **MULTI** option is defined (see Section 4.7), each CBS can handle more than one task. How these tasks are scheduled by Linux is up to the standard Linux scheduler.

4.5 Tests

The **qres** module is currently distributed together with a test program that can be used to verify the correct behavior of the module trough a graphical trace of the scheduling process.

The test program is made up of two executable file: *Schedtest* and *Filter*.

The *Schedtest* program run a time consuming CBS task with a specified budget and period. The syntax is:

```
schedtest Q T seconds > tracefile
```

where *Q* is the budget of the server (in micro-sec), *T* is the period of the server (in micro-sec), *seconds* is the execution time of the test and *tracefile* is the name of the file for saving the trace. You can run more than one *Schedtest* program at the same time with different budget and period to see the scheduling execution of CBS tasks.

Filter extracts some information from the output of *Schedtest*, the *tracefile*, and creates a graphic trace file that can be visualized using the program *xfig*. The syntax is:

```
filter tracefile_1 ... tracefile_n > trace.fig
```

where *tracefile_1 ... tracefile_n* are the file previous created by the *Schedtest* program.

For example, to view the scheduling trace of two CBS tasks, the first with a budget of 20000 and a period of 60000 micro-sec, the second with a budget of 40000 micro-sec and a period of 600000 micro-sec, for 30 seconds, just type:

```
./schedtest 200000 600000 30 > trace1 & ./schedtest
400000 600000 30 > trace2
```

This command runs two tests and creates two output files, called trace1 and trace2.

To create a grafic trace file, type:

```
./filter trace1 trace2 > trace.fig
```

This command creates a file called *c.fig*, that can be viewed using the *xfig* program.

4.6 Examples

The **qres** module is currently distributed together with two examples that can be used as utilities.

Wrapper is a program that can be used to schedule a task with CBS scheduler. The syntax is the following:

```
wrapper Q T COMMAND
```

where Q is the budget of the server (in micro-sec), T is the period of the server (in micro-sec), and COMMAND is the name of the command to start (with its absolute path). For example, to run the program *top* with a budget of 20000 and a period of 60000 micro-sec, just type

```
./wrapper 20000 60000 /usr/bin/top
```

Cbser is a program that transforms a task running on the Linux system in a CBS task; its syntax is the following:

```
cbser Q T PID
```

where *Q* is the budget of the server (in micro-sec), *T* is the period of the server (in micro-sec), and *PID* is the Process ID of the task (to know the pid of the task is possible to use the command *top*). For example, if the program *find* is running with the PID 2001, and you want to run it with a budget of 20000 and a period of 60000 micro-sec, just type:

```
./cbser 20000 60000 2001
```

4.7 Installation instructions

CBS is a loadable Kernel Module that can be inserted on a Linux platform to distribute the CPU time to the system tasks. Before using this feature, the following steps are needed:

Patching the Linux kernel with the 'Generic Scheduler Patch'

Compiling the **qres** module

Installing the **qres** module

Patching the kernel

Locate the directory of the kernel on your Linux system (usually /usr/src/linux). Be sure that the release of the kernel used on your system is 2.4.18 (to verify the kernel release run the command 'uname -r'). If there isn't any kernel directory on the system, or if the kernel release is wrong, download the kernel from *http://www.kernel.org*, and extract the content of the file on a directory of your choice. Copy the 'Generic Scheduler Patch' on the kernel directory, and type

```
patch -p1 < generic_scheduler.patch
```

Now the kernel is patched. To continue, it's necessary to compile and install the patched kernel. Compile the kernel using the following commands:

make dep

make bzImage

make modules

make modules_install

Restart your system.

Compiling the qres module

If the kernel directory isn't '/usr/src/linux', set the environment variable KERNEL_DIR typing the command 'export KERNEL_DIR=your_kernel_directory'. Go in the directory containing the CBS code (usually qres/src). This directory should contain a file called Makefile. There are many options that can be set during the compilation. The easiest way to compile the module is to type 'make'. This will create a module with the basic features. However, it's possible to specify the following options (which should be appended to the 'make' command):

DEBUG =1 The module prints some warnings during the execution. To read the warnings printed, type 'dmesg'

MULTI=1 Gives a bandwidth of 10% to all Linux tasks. In this way these tasks can execute during the execution of the tasks scheduled by CBS scheduler. In addition this option lets you assign an amount of bandwidth to a set of tasks.

HRT=1 Use this option if you patched the kernel with 'High Resolution Timers' patch. This gives to the system a good precision (below 10msec).

PRECISE_ALLOC=1 Any task scheduled by CBS scheduler executes exactly as specified by its bandwidth (even if there is only one CBS task on the system). This is a possible solution to solve some problems that the CBS algorithm encounters when used to schedule very long jobs.

LOG=... During debugging often happens that the kernel is unable to correctly log all the warning messages printed by the module. This option specifies the directory of the logger to be used, and is usually set when the option DEBUG is set too.

GRUB=1 The module uses the algorithm GRUB instead of CBS (GRUB is capable of reclaiming the unused bandwidth).

GRUB_DEBUG=1 Works like the option 'DEBUG', and should be used when the option 'GRUB' is set.

GRUB_EXTRA_BANDWIDTH=1 When a task blocks, its remaining budget is given to next running task.

GRUB_PWR=1 Adds the feature of power saving to the GRUB scheduler, and must be used only when the option 'GRUB' is set. By now, it is possible to use this feature only on Intel PXA250 processors (see the option 'PXA250').

ARM=1 Enables a cross compiling for ARM processors. It requires the 'arm-linux' utilities (i.e. Arm-linux-gcc).

PXA250=1 Enables a cross compiling for Intel PXA250 processor.

PROC=1 Enables the /proc FileSystem support for CBS scheduler.

QMGR=1 Support the qmgr module. You must enable this modules if you want to use the qos manager module.

Installing the qres module

To install the module in the system, type '**make load**' (equivalent to **insmod**). To remove the module from the system, type '**make unload**' (equivalent to **rmmod**).

Chapter 5. Quality of Service Manager

5.1.Summary

Name **qmgr**

Description: it is a dynamically loadable module for the Linux kernel that provides QoS management services.

Author : Luca Marzario (lukesky@gandalf.sssup.it)

Reviewer:

Layer: Linux High Level.

Version 1.0

Status alpha

Dependencies: the module depends on the generic scheduler patch and on the qres component. It takes advantage of the High Resolution Patch and of the Preemption Patch [PRK].

Release date (milestone) April 2003.

5.2 Description

The use of computer based solutions for time-sensitive applications is increasingly popular. Important examples include multimedia streaming, video-conference, CD/DVD burning and so forth. Such applications are certainly time-sensitive but classical hard real-time techniques prove unsuitable to support them.

Soft real-time scheduling solutions are commonly regarded as a better solution since they provide temporal isolation, thus allowing for individual task timing guarantees. Moreover, they approximate a fluid allocation of resources, which is certainly a desirable feature to have, for instance, in multimedia applications. However, the problem of finding a correct allocation of bandwidth to the different tasks is still to be considered as a tough problem.

This problem is addressed in this software components and it essentially amounts to finding an appropriate allocation of resources to competing activities. To this regard, a static policy, in our evaluation, confronts unfavorably to a dynamic one. The main argument to support this point of view is that there is often a structure in the dynamic variations of resource requirements for tasks. For instance a typical movie alternates “"fast scenes" (which induce a heavy computation workload) to "slow" dialogues between the characters (which on the contrary require a light computation). To take advantage of this structure, we propose a dynamic adjustment of the bandwidth, in which a software module complementary to the scheduler collects QoS measures for the executing tasks and varies the assigned bandwidth accordingly.

The design of this module is largely based on concepts borrowed from control theory.

The application of control theory to scheduling problem has been explored in the recent years. In our context, the design of a feedback controller is greatly aided by the availability of a a dynamic model for CPU reservations [Abe02] combining accuracy with analytically tractability. This result has opened up an interesting research field on appropriate design of a feedback controller. A first result of this kind is shown in [Pal03] and is the theoretical foundation for the algorithms used in the package. However, the exploration of different alternatives is under way.

5.3 API / Compatibility

As in the **qres** scheduler component, the only way of setting the parameters for the feedback scheduler is through the **sched_setscheduler**() standard API. The **qmgr** module defines a new data structure:

```
struct qmgr_param {
  unsigned long int signature;
  unsigned long int cbs_period;
  unsigned long long int qmgr_period;
  unsigned long int qmgr_max_budget;
  unsigned int delta;
  unsigned long int h;
  unsigned long int H;
  unsigned long int ei;
  unsigned long int Ei;
  unsigned long int e;
  unsigned long int E;
};
```

The parameters have the following meaning:

signature identifies the structure and must be assigned the value QMGR_SIGNATURE.

cbs_period is the same as the CBS parameter (see Section 4.2).

qmgr_period is the period of the task. It may be different from the period of the CBS server. Usually, the period of the task is a multiple of the period of the server.

qmgr_max_budget is the maximum possible budget that can be assigned to the server. The CBS inizial max_budget is inizialized with this value.

delta is the maximum estimated variation of the execution time between two consecutive instances, and it is expressed in microseconds.

H and **h** are the maximum and minimum values, respectively, of each task's instance computation time, expressed in microseconds.

Ei and **ei** are the maximum and minimum values, respectively, of the desired scheduling error.

E and **e** are the maximum and minimum value, respectively, of the guaranteed scheduling error.

To specify that a task has to be served by a CBS server with a feedback scheduler, we have to invoke the **sched_setscheduler** in the following way:

```
struct qmgr_param parms;
struct sched_parms sp;

parms.signature = QMGR_SIGNATURE;
parms.max_budget = ...;
// set all parameters...

sp.sched_size = sizeof(struct qmgr_param);
sp.sched_p = &parms;

sched_setscheduler(getpid(), SCHED_QMGR, &sp);
```

5.4 Implementation issues

The **qmgr** module requires the presence of the **qres** module, as it completely relies on it for task scheduling. When the **qmgr** module is loaded, it substitutes all the generic scheduler hook with its own functions. When the **sched_setscheduler**() is called, it checks if the signature is equal to QMGR_SIGNATURE. If it is not, it invokes the original hook (i.e. the **qres** function). Otherwise, it simply sets the parameters to the internal data structures, and then invokes the original **qres** hook, passing the correct data structure.

When a task is blocked, the **qmgr** hook is called: it computes the new bandwidth to assign to the CBS task and updates the corresponding **max_budget** field.

When a task forks, the new task by default is not handled by the **qmgr** module to not compromise the execution of the parent.

When a task ends, the relative **qmgr_struct** is freed.

5.5 Tests and validation

A test suite has been devised and evaluated, but its application on the module is under way upon the release of this document.

5.6 Examples

The **qmgr** module is currently distributed together with one examples that can be used as utilities.

Wrapper_qmgr is a program that can be used to schedule a task with the Qos Manager. The syntax is

```
wrapper_qmgr Qmax Tcbs Tqmgr h H ei Ei e E COMMAND
```

where *Qmax* is the maximum budget that can be assigned to the CBS server (in microsec), *Tcbs* is the period of the CBS server (in microsec), *Tqmgr* is the period of the task, *h, H, ei ,Ei, e E* are the execution time's profile of the task (see Section 5.3), and

COMMAND is the name of the command to start (with its absoulte path). For example, for a task with a period of 32000 microsec (i.e. an mpegplayer):

```
./wrapper 10000 20000 32000 5000 90000  15000 0 20000
                    12000 /usr/bin/mpegplayer
```

5.7 Installation instructions

The **qmgr** module is based on the CBS module, so if you want to install this module, the following steps are needed:

Installing the qres module (see Section 4.7)

Compiling the **qmgr** module

Installing the **qmgr** module

Compiling the qmgr module

If the kernel directory isn't '/usr/src/linux', set the environment variable KERNEL_DIR typing the command 'export KERNEL_DIR=your_kernel_directory'. Go in the directory containing the qmgr code (usually qmgr/src) and type 'make'. This will create a module called qmgr.o.

Installing the qmgr module

To install the module in the system, type 'make load' (it's equivalent to the command 'insmod'). To remove the module from the system, type 'make unload' (it's equivalent to the command 'rmmod').

Chapter 6. User API

6.1.Summary

Name **qlib**

Description A library for compatibility between RTLinux and Linux

Authors: Emiliano Giovannetti (giovannetti@gandalf.sssup.it), Giuseppe Lipari (lipari@sssup.it)

Reviewer

Layer Linux user space

Version 1.0

Status alpha

Dependencies depends on the High Resolution timer.

Release date (milestone) April 2003, (MS2)

6.2 Description

This component is a library of functions for threads in user space that tries to replicate, as much as it is possible the RTLinux API. The objective of this component is to let an user to test a real-time application in user space.

The motivation is that Linux user space is a much more controlled environment, where each application has its own memory space and faulty application cannot harm the functionality of the entire system. Also, in user space it is possible to specify access permissions for each user, so that a common user cannot access critical services and compromise the functionality of the system.

LinuxThreads

The proposed library is built upon the LinuxThreads implementation of the POSIX threads. The LinuxThreads implementation is known to be non-compatible with the standard POSIX for signal handling. According to the POSIX standard, “asynchronous” signal are addressed to the whole process (i. e. the collection of all threads), which then delivers the signal to one particular thread. The thread that actually receives the signal can be any of the threads that do not currently block the signal.

In LinuxThreads, each thread is actually a kernel process with its own PID, so external signals are always directed to one particular thread. This is significantly different from the standard.

Our library should not be based on any particular implementation of the POSIX threads. However, we have to develop it on top on Linux, which currently supports only the LinuxThreads. There are proposal that will probably change such implementation in the future [Coo02]. This is also driven by the need for a faster and more responsible API.

The first release of this component is however based on the LinuxThreads implementation. In the next release we will try to provide a library that is more adherent to the standard.

6.3 API / Compatibility

The library currently supports a few functions for periodic thread support and for suspension / activation of a thread.

```
struct timeval timeval_from_ns (long long t);
```

This function transforms a time value from nanoseconds (expressed as long long) into a **timeval** structure.

```
long long timeval_to_ns (const struct timeval *ts);
```

This function transforms a time value from a **timeval** structure into nanoseconds (expressed as long long)

```
int start_rt (void);
```

This function must be called before entering into “real-time mode”. It installs some handler and initialise library's internal data structures.

```
int pthread_wait_np (void);
```

This function suspends the thread until it is awaken by calling either pthread_make_periodic_np() or pthread_wakeup_np(), or by a timer expiration. If the thread is periodic, it will be activated at the next period.

```
int pthread_suspend_np(pthread_t pth);
```

This function suspends the thread specified by the parameter until it is explicitely activated by a pthread_wakeup_np(). If the suspended thread is periodic, it will not be activated at the next period.

```
int pthread_make_periodic_np(pthread_t pth);
```

This function makes a thread specified by pth periodic with period ..., The period is expressed in ... The thread can have an initial offset specified by ..., The thread will be activated ath the following time instants, ... , unless is has been suspended by a pthread_suspend_np().

6.4 Implementation issues

To work properly, the library needs the High Resolution Times patch installed. ... describe the HRT.

6.5 Tests and validation

A complete set of test for this module is currently under development. As a preliminary

test, we have compiled and run, with little modification, two of the examples distributed with RTLinux. The results obtained are quite encouraging, as the same output was given by running the examples on RTLinux and on Linux with our library. However, a more formal test suite is currently under development.

6.6 Examples

Not available yet.

6.7 Installation instructions

See the REAME file.

Bibliography

Abe02: Luca Abeni, Luigi Palopoli, Giuseppe Lipari, Jonathan Walpole, Analysis of a Reservation-Based Feedback Scheduler, 2002

Abe98: Luca Abeni and Giorgio Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems, 1998

Coo02: Jerry Cooperstein, Linux Multithreading Advances, 2002, http://www.onlamp.com/lpt/a/2839

Lip00: Giuseppe Lipari and Sanjoy Baruah, Greedy Reclamation of Unused Bandwidth in Constant Bandwidth Servers, 2000

LLP: Ingo Molnar, Low Latency Patch, , http://www.zip.com.au/~akpm/linux/schedlat.html

LMBench: Larry McVoy and Carl Staelin, LMBench, Tools for Performance Analysis, , http://www.bitmover.com/lmbench/

LTP: various, Linux Test Project, , http://ltp.sourceforge.net

Pal03: Luigi Palopoli, Luca Abeni, Giuseppe Lipari, On the applications of hybrid control to CPU Reservations, 2003

PRK: Robert Love, Preemption Patch, , http://www.tech9.net/rml/linux/