WP4 – Development of Resource Management Components

D4.1 Design of Resource Management Components

Design of Resource Management Components: Deliverable D4.1
by Giuseppe Lipari, Luca Abeni, Luca Marzario, and Luigi Palopoli

Draft 0.1 Edition
Published February 2003

Table of Contents

List of Tables

List of Figures

Document presentation

Table 1. Project Co-ordinator

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

Table 2. Participant List

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

Chapter 1. Introduction

1.1. Motivation

The recent evolution of computer technology made personal computers powerful enough to perform such time sensitive applications as multimedia streaming, video-conference and so on. However, traditional real-time techniques are unsuitable to the design of this family of applications.

Indeed, these applications are characterised by highly varying computational requirements that make it impossible to compute a worst case computational workload. For example, a MPEG stream is composed by a sequence of encoded video frames, each one with a different size and a different computational requirement. Since a MPEG player is made for playing any kind of stream, it is impossible to know a-priori how much execution time the player will take to decode and display a frame.

Furthermore, personal computers are very different from dedicated computing platforms, like the ones utilised in embedded critical real-time systems. A personal computer may execute very different kinds of applications, from simple batch programs, to interactive programs, to multimedia programs. Moreover, a personal computer is assembled with very diverse hardware boards. All of this makes it impossible to perform an a-priori analysis of a real-time program.

Another important point to take into account is the "criticality" of such applications. Multimedia applications are considered "soft real-time": while a hard real-time task must be guaranteed to meet all its deadlines, a soft real-time task has less strict temporal requirements and some deadlines can be missed without compromising the functionality of the application. However, it is important to "keep under control" how many deadlines are missed, and "how late" a soft task is going to be. A typical example of such QoS requirement is the *deadline miss probability*; another possible requirement is a bound on the *tardiness* of a task (i.e. how much it is late in proportion to its period). In this framework, standard real-time tasks are a special case for which the probability of a deadline miss has to be 0 over any interval of time.

1.2. Objectives

The aim of the Resource management components is to assign the system resources to tasks so that predictable Quality of Service (QoS) guarantees can be provided to real-time tasks. The Resource Management components are especially useful for soft real-time tasks, like for example multimedia applications. These applications are usually implemented as normal Linux processes. In fact, Linux offers a large number of libraries, drivers and tools to support this kind of applications, like sound and video drivers, a standard network protocol stack, etc. These services cannot be found in RT-Linux and it is unfeasible to port them.

Therefore, in the OCERA Architecture, the Resource Management components are provided at the Linux layer as additional services of the Linux kernel. They can be used both in the soft real-time configuration (if no hard real-time service is needed), or in the mixed hard/soft configuration (when both hard real-time and soft real-time tasks coexists in the system). The use of the QoS manager in the mixed configuration can be useful for those applications composed of a hard real-time part and a soft real-time part:

- the critical (hard real-time) part can be executed as a set of threads in the RT-Linux executive, and must be "trusted", because a fault in a thread can compromise the functionality of the whole system;

- less critical parts can be implemented as Linux processes, and can use memory protection, provided by the Linux kernel, and temporal protection, provided by the additional component for QoS management.

However, in the mixed hard-soft configuration, some kind of guarantee must be provided at the hard real-time level to Linux activities. In fact, Linux is currently scheduled as background activity in the RT-Linux executive, i.e. only when there is no active hard real-time tasks. If Linux is scheduled in background, no QoS guarantee is possible at the soft real-time level, because the amount of execution time allotted to Linux is not distributed evenly and Linux can be delayed by real-time tasks for large time intervals.

As a consequence, if we want to provide QoS guarantees to soft real-time tasks in the mixed hard-soft configuration, we have to reserve a fraction of the CPU bandwidth to Linux, and to schedule it according to a reservation algorithm. This can be seen as a hierarchical scheduling problem: at the hard real-time level, a scheduler selects which tasks has to be executed next; if Linux is selected, its scheduler selects which process has to be executed next.

This hierarchical scheduling problem will be addressed together with the University of Valencia (UPVLC). In WP5, a resource reservation scheduler will be implemented as RT-Linux scheduler. Moreover, SSSA will collaborate with UPVLC to devise a suitable schedulability analysis of such hierarchical systems.

1.3. Summary of Resource Management Components

In this section, we summarise the components that will be developed as part of Workpackage 4, "Resource Management Components". For each components we give a brief description, who is responsible for the development of the components, and who is responsible for its validation. The same information can be found on the "OCERA Architecture and Component Definition" deliverable. Detailed information on each component will be provided in 4, *Resource Management Components*.

Low level Linux components

- Generic Scheduler Patch
  - It is a small patch for the Linux kernel that provides useful hooks to the Linux scheduler. These hooks will then be used by our scheduling module for implementing sophisticated real-time scheduling policies.
  - Responsible: SSSA
  - Validator: UPVLC

- Integration patch
  - This patch will take into account the introduction of the "preemption patch" and "high resolution timers" in the Linux kernel.
  - Responsible: SSSA
  - Validator: UPVLC

High level Linux components

- Resource Reservation Scheduling module

- It will be a dynamically loadable module for the Linux kernel, that will provide a resource reservation scheduler for soft real-time tasks running in user space.
- Responsible: SSSA
- Validator: UPVLC, CEA

- Quality of service Manager
  - This component will provide a mechanism for identifying the temporal characteristics of a task and to adjust its scheduling parameters so to maximise its quality of service.
  - Responsible: SSSA
  - Validator: VT, UPVLC

Linux applications

- User API
  - This component is a set of one or more libraries that will provide a convenient API to the user to access the Resource Management services.
  - Responsible: SSSA
  - Validator: UPVLC

1.4. Organisation of the report

The report is organised as follows. 2, *Resource Reservation in Linux* describes the scheduling technique (Resource Reservation) that we propose to adopt as a foundation for the Resource Management component. In particular, after introducing the resource reservation concepts in general, we present the Constant Bandwidth Server and its implementation in the Linux kernel. The chapter is concluded by a short overview of some extensions that we are currently investigating, and by a description of the user-level support that will be provided.

3, *Resource Management* describes the dynamic QoS management techniques that will be implemented over the reservation scheduler, and their implementation in the OCERA framework. After a short introduction of the resource management problem, it will be shown that some kind of *adaptive* management of system resources is needed, and a theoretically sound solution (feedback scheduling) will be presented. Finally, a short description of the implementation of a feedback scheduling mechanism based on the OCERA CBS scheduler, and deeply integrated with the OCERA real-time library, will be presented.

Finally, in 4, *Resource Management Components* we list the components that will be developed: for each component we will present the basic structure of the implementation. Moreover, we will propose measurable objectives to be checked at the end of the development phase to prove the effectiveness of our implementation. Finally, we shortly describe the relationship with the other components of the OCERA project.

Chapter 2. Resource Reservation in Linux

2.1. Background on Resource Reservation

Real-time applications are characterised by temporal constraints. A real-time task T_i is usually modelled as a sequence of jobs $J_{i,j}$, each one characterised by an arrival time, a computation requirement and an absolute deadline by which it should complete. For an hard real-time tasks, the deadlines are critical constraints and if missed the system can have a catastrophic failure. A soft real-time tasks, instead, is characterised by less critical requirements, and if some deadline is missed nothing catastrophic happens. However, the QoS experienced by the task depends on the number of deadline misses.

Many scheduling algorithms have been proposed to service real-time tasks (fixed priority, earliest deadline first, etc.). For each scheduling algorithm, it is usually possible to perform a schedulability analysis that given the periods and the worst case computation times of all tasks, checks if the task set is schedulable, i.e if all deadlines can be respected.

However, such a schedulability analysis generally depends on the estimation of the worst case execution time of each task, and it is not robust to variation in this parameter. Indeed, if a task does not respect its estimated worst case execution time, deadline misses can occur on other tasks in the system. According to an increasingly popular terminology, this anomaly is referred to as a lack of *temporal isolation* between tasks. Temporal isolation is not a primary concern in safety critical systems, for a deadline miss is regarded as a sever design problem whatever the task that undergoes it. On the contrary, in a desktop system, tasks are dynamically activated during the system lifetime, and cannot be accounted for a priori (i.e., at design time). As a consequence, a misbehaving real-time task can jeopardise the whole system schedulability, potentially affecting all the other tasks. If this behaviour is malicious, this is a security failure: a user can affect the QoS perceived by other users and, even worse, can starve all the system's tasks (denial of service). These arguments emphasise the importance of temporal protection in soft real-time systems.

With the term *Resource Reservation* we indicate a class of scheduling algorithms that allocate a resource to a set of competing tasks, ensuring the *temporal protection property*. In this chapter, we first recall some basic information on the generalities of a reservation mechanism. Then we propose an advanced reservation mechanism for the CPU.

Informally, we can give the following definition of temporal protection:

Temporal Protection This property holds if and only if the timing behaviour of a task is decoupled from that of other tasks in the system.

Temporal protection is simply enforced by ensuring that each task will receive, in a given period of time, *at least* an assigned fraction of the resource (bandwidth). Therefore, even when a task requires more computation time than its reserved bandwidth allows, it is slowed down without restricting the bandwidth allocated to other tasks. For the sake of efficiency, scheduling algorithms adhering to this paradigm can also be endowed with an additional feature that "reclaims" the unused computation bandwidth and allocates it to the executing tasks according to some policy (i.e a fair rule, greedy rule, etc.).

Temporal isolation permits to provide different kind of guarantees to different tasks: for example, it is possible to perform a hard guarantee on a critical task (i.e. it is possible to guarantee that a hard real-time task does not miss any deadline), whereas it is possible a probabilistic guarantee on soft tasks (i.e., given the distribution of computation times and arrival times, we can compute the deadline miss probability distribution).

A way to provide temporal isolation is to guarantee that every task will be assigned the resource only for the reserved amount of time in every time interval. If exec(T_i, t_1, t_2) is

the amount of time executed by task T_i in the interval (t_1, t_2), temporal protection can be ensured by enforcing that:

for all t_1, t_2, $exec(T_i, t_1, t_2) / (t_2 - t_1) > F_i$

where F_i is a real number in [0,1] that represents the share of resource assigned to the task.

However, this requirement (that leads to the *Generalised Processor Sharing* - GPS - model used by proportional share schedulers) cannot be implemented in practise. In fact, the previous equation must hold for all t_1, t_2, and hence also for infinitesimal intervals. This implies an infinite amount of context switches. A more realistic requirement is to enforce that the previous equation holds only *over well specified intervals*, for example between the arrival time and the deadline of a real-time task.

More formally, a reservation (Q, P) guarantees that at least an amount Q of a resource will be available to the reserved task every period P.

The period P is a measure of the *granularity* of the reservation. The smaller is P, the more precise is the allocation of the resource, but the higher is the overhead (e.g. context switches). On the other hand, the larger is P, the less the overhead, but a coarser allocation of the resource.

In a first phase, we will consider a single computational resource (CPU) that must be allocated among different real-time tasks. Moreover, we consider that all tasks are independent and preemptive. This is a very simple model, but it is useful to introduce the concept of resource reservation, and it will be easier to implement in a complex operating system like Linux.

In a second phase we will take into account more complex models, considering the possible interactions between tasks (client/server relationships, exclusive accesses to shared resources, and so on). Our preliminary experience seems to suggest that *ceiling* and *inheritance* approaches can be successfully extended to reservation systems, and can help to provide guarantees in realistic situations.

Moreover, the effects of *hard* and *soft* reservation behaviours on hierarchical guarantees should be investigated.

Finally, a reclaiming mechanism will be added to the basic reservation algorithm, to make the scheduler more effective when applied to tasks characterised by irregular execution patterns. Again, we have some preliminary and informal results that seem to indicate that a combination of hard reservation techniques and resource reclaiming can give the best results.

2.2. The Constant Bandwidth Server

In the first phase of the workpackage we will implement the Constant Bandwidth Server (CBS) [Abe98] algorithm in Linux. The CBS algorithm has been originally proposed for servicing aperiodic tasks in a hard real-time environment. However, it can be used very well as a basis for a resource reservation based system.

In this section, a brief overview of the CBS algorithm is given. A fully detailed description can be found in [Abe98]. A CBS S_i is described by two parameters: *the server maximum 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. Dynamically, each server updates two variables $(q_i, d_i$ where q_i is the *current budget* and keeps track of the consumed bandwidth, and d_i is the server's *scheduling deadline*. Initially, q_i is set to the maximum budget Q_i and d_i is set to 0. A server is *active* if the corresponding server task has a pending instance.

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 d_i is selected and the corresponding task is dispatched to execute. It is worth noting that the scheduling deadline of a server can be dynamically changed. In this case the EDF queue has to be reordered and a preemption may occur.

The CBS algorithm updates its variables as follows:

1. When job $J_{i,j}$ of task T_i arrives in the system at time t, the server checks if the current pair (q,d) can be used. If $t <= d - q (P / Q)$ then the server simply accepts the task and continues to use the current pair (q,d). Otherwise, it computes q = Q and d = t + P.
2. At each instant, the server with the earliest scheduling deadline d is selected to execute (EDF rule over the servers).
3. If server Sexecutes for E units of time, then $q = q - E$.
4. Server S is allowed to execute until q is greater than 0. If q becomes 0 and the task has not yet finished executing, the scheduling deadline is postponed to $d = d + P$ and the EDF queue is reorder. As a consequence, the server may not be the earliest deadline server and a preemption may occur.

It has been proved that algorithm CBS provides temporal protection.

2.3. Implementing CBS in Linux

2.3.1. Task Model

When implementing a scheduling algorithm in a complex operating system like Linux, we have to account for many practical issues. In particular, the model of independent periodic task as a sequence of jobs that only block at the end of each execution is not very realistic. Many other kinds of blocking situations can occur in the system. A task can be blocked:

- by a read operation (for example, reading from a disk)
- waiting for external data (for example, data coming from the network);
- by another task that is using an mutually exclusive resource (i.e. on a mutex semaphore)
- by all the forms of blocking IPC in general.

All of this blocking operations are not taken into account by the traditional real-time task model, and must be carefully handled to avoid anomalies in the CBS guarantee. In particular, if a task blocks without notifying the CBS scheduler, the algorithm can become inconsistent.

To avoid problems due to tasks dynamically blocking and unblocking, we decided to consider a job starting when a task unblocks, and finishing when the task blocks again. Therefore, the first job of a task is activated explicitly by another task (for example, by using a fork() syscall); the subsequent jobs are activated when the task unblocks.

```
int main(void)
{
  <initialisation>
  while(1) {
    <body>;
    <wait for next instance>;
  }
}
```

Figure 2-1. Example of task structure

In Figure 2-1, the typical structure of a periodic task is shown. If the task is never blocked during the execution of the operations in <body> then each job finishes at the end of the while loop, when the task blocks waiting for the next instance. However, if the task is blocked inside the body for whatever reason, then each while loop is actually composed by *more than one job*.

Note that this approach permits to easily preserve the temporal protection between tasks, even when a task is characterised by an irregular execution pattern (obviously, it is impossible to achieve protection between two tasks when they are interacting). However, if no other special caution is taken it is quite difficult to guarantee the performance of interacting tasks. In this case, some kind of ceiling or inheritance approach must be adopted.

For example, the Bandwidth Inheritance Protocol (BWI) [BWI] permits to "inherit" the server bandwidth when a task is blocked by another task on a shared resource. However, such protocol is complex. We will evaluate its use in this project and implement it in the second phase of the workpackage.

2.3.2. Structure of the implementation

The CBS scheduler has been implemented as a loadable module that, once inserted into the kernel, enables a new scheduling policy, SCHED_CBS. This scheduling module intercepts the scheduling events needed to correctly implement the CBS algorithm (in particular, process creation / deletion, and blocking / unblocking).

To intercept these events, the module must be inserted in a patched kernel, which exports 6 hooks: the fork_hook, the cleanup_hook, the block_hook, the unblock_hook, the setsched_hook, and the getsched_hook. For every hook, the patched kernel contains a function pointer that is exported to be used by loadable modules. All the pointers are initially set to NULL. When the CBS module is initialised, it sets these pointers to the appropriate handlers, so that the kernel will invoke the proper handler when one of the following specific condition happens:

- the fork_hook is invoked when a new task is created;
- the cleanup_hook is invoked when a task is destroyed, so that the CBS scheduler can deallocate the resources associated to it;
- the block_hook is invoked when a task blocks, so that the CBS module understands that the current job is finished;
- the unblock_hook is invoked when a task wakes up, so that the CBS scheduler is notified of a new job arrival;
- 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.

When the unblock_hook is invoked, we only need to set a timer to the time when the budget becomes zero. This timing event is then handled internally by the module. All the hooks but setsched_hook and getsched_hook receive the pointer to the task_struct of the interested task as a parameter (setsched_hook and getsched_hook have the same parameters of sched_setsched() and sched_getsched()). Finally, the kernel patch adds a field to the task_struct structure, representing a pointer to void that can be used by the

CBS scheduler to point to the CBS private data for each task (current budget, server parameters, deadline, and so on).

Such hooks, and the pointer to the task private data are added by the *Generic Scheduler* (gensched) patch, that we designed to be generic, simple, and minimally intrusive. Although the gensched patch has been developed to implement the OCERA CBS scheduler, it can be easily used to implement also other different scheduling algorithms in the Linux kernel (we believe that the interface exported by the gensched patch, based on the scheduler activations concept, is generic enough to implement any scheduling algorithm).

Based on the gensched patch, the CBS scheduler will be implemented as a loadable module that can be inserted into the kernel at runtime, and most of the scheduler code will be independent from the kernel version. Since the CBS module is based only on the gensched patch, it will be very easy to port the OCERA CBS implementation to new Linux versions, and to use it in combination with other kernel patches that provide useful features for real-time applications. For example, the CBS module can be used in combination with the high-res-timers patch, and with the kernel preemptability patch.

Note that in our implementation the scheduling algorithm does not depend on tasks' periodicity: in fact, the scheduler directly intercepts tasks' activations / deactivations. Implementing a periodic behaviour is the tasks' responsibility. For example, the standard setitimer() syscall or POSIX timers can be used for triggering periodic activations.

2.3.3. Reclaiming spare resource

When the sum of the reservation bandwidths in the system is less than 100%, or when a task uses less than it has been reserved, there is some spare resource that can be re-distributed among the demanding tasks. In this way, we provide each task *at least* the amount of required bandwidth. However, in the average, a task will get more than required because of the reclaiming mechanism.

Many reclamation mechanism have been proposed in the literature for resource reservation algorithms. Two are based on the CBS algorithm. the CASH (Capacity Sharing) algorithm has been proposed by Caccamo Sha and Buttazzo CASH and is very useful for periodic real-time tasks, but it is difficult to be used in sporadic and aperiodic tasks.

The GRUB (Greedy reclamation of Unused Bandwidth) algorithm has been proposed by Lipari and Baruah GRUB and does not assume any particular task model. Moreover, it has the same complexity as CBS, and can be implemented with a small additional overhead. For this reason, we selected GRUB as the algorithm to be implemented in OCERA.

In this algorithm, a server S_i is characterised by a period P_i and by a bandwidth U_i. A server can be in three different states:

- **Active Contending**, when the corresponding task is ready to execute;
- **Active non contending**, when the corresponding task has finished to execute but its bandwidth cannot be reclaimed
- **Inactive**, when the corresponding task is not executing and we can reclaim the bandwidth.

The GRUB algorithm follows the same basic rules of the CBS algorithm, with the following differences.

A server becomes **Active contending** when a new instance of the task arrives (i.e. the task is activated for the first time after a fork() or the task is unblocked).

A server becomes **Active non contending** at time t when the task is blocked and $q < (d - t) U_i$. Otherwise, the server directly becomes **Inactive**

If a server is **Active non contending**, it becomes **Inactive** at time $t = d - q / U_i$.

If a server executes for c units of time, then its budget is decreased as:

$q = q - c\ U / U_i$

where U is the sum of the bandwidths of all active (contending and not contending) tasks.

It can be proved that Algorithm GRUB provides temporal isolation and timing guarantees exactly like the CBS algorithm. In addition, in average over long interval of time Algorithm GRUB distributes the spare bandwidths to all tasks in a fair manner.

Algorithm GRUB can be implemented with little effort. The difference with the CBS is that we need one more timer for handling the transition between the **Active non contending** and the **Inactive** states.

2.3.4. Resource Reservation on Multi-processor platforms

In the previous sections, we described the scheduling problem and the solutions that we propose (the reservation approach in particular) referred to a single CPU system. The presented scheduling algorithm can be extended to multi-processor (SMP, NUMA, and distributed) systems.

We believe that such an extension to multi-processors is particularly important because using multiple CPU is an easy way to satisfy the high computational demands of modern multimedia applications. However, multiprocessor machines present a number of well known scheduling anomalies when applied to real-time applications (there are some famous examples in which increasing the computational power causes increases the number of missed deadlines).

In our first multiprocessor implementation, we will consider SMP machines, because SMP is the most common multiprocessor architecture, and is easily available for development and testing. The simplest way to implement resource reservations on a SMP is to statically associate processes to CPUs, so that algorithms and guarantees for developed for a single CPU can be directly applied to each CPU. Such a *static partitioning* has the advantage of the simplicity, but does not permit to fully utilise the whole computational power of the SMP machine.

A more efficient approach would be to permit *task migration* between different CPUs. However, in this case, sever analysis problems arise and a deeper study of the multiprocessor scheduling algorithm and schedulability analysis is needed. The goal of the algorithm should be to achieve an utilisation equal to $n\ U_{lub}$ where n is the number of processors, and U_{lub} is the utilisation least upper bound for a single-CPU system. This goal can be achieved only in some particular cases (and for particular task sets), and it is still not clear which is the optimal bound in the generic case.

A first preliminary study on the use of reservation techniques in multiprocessor platforms has been presented in MCBS. The authors present the M-CBS algorithm that is based on a multi-processor version of EDF where tasks are allowed to migrate from one processor to another. The utilisation bound can be expressed as:

$U_{lub} = m - (m - 1)\ U_{max}$,

where U_{max} is the maximum server bandwidth in the system.

We will further investigate such approach and try an experimental implementation in the context of OCERA.

2.4. User Library

The functionalities provided by the CBS scheduler can be accessed by using an extension of the standard C library interface (a new policy SCHED_CBS is added to the sched_setsched() interface). However, performing real-time activities requires a lot of complex activities, and performing them explicitly in the user application can add an excessive amount of complexity to the code. For this reason, we will implement an user-level library providing a simple and effective interface for real-time applications.

This real-time library will permit, for example, to easily create periodic tasks removing the need for an explicit management of signals and periodic timers. Moreover, if the kernel provides high-resolution timers the real-time library will be able to automatically take advantage of them. Another important functionality provided by the real-time library will be the possibility to read the system time from user space, without performing system calls. This will require to read the CPU timestamp counter, or some similar kind of register, and to calibrate the timing code before at program startup. The real-time library will automatically perform this operation without adding complexity to the user code.

2.5. Mixed Hard/Soft configuration

When OCERA is configured in the mixed hard/soft configuration, we need a way to give QoS guarantees to user processes running under Linux. Linux is currently scheduled as background activity in the RT-Linux executive, i.e. only when there is no active hard real-time tasks. If Linux is scheduled in background, no QoS guarantee is possible at the soft real-time level, because the amount of execution time allotted to Linux is not distributed evenly and Linux can be delayed by real-time tasks for large time intervals.

A possible solution is to use resource reservation algorithm in the RT-Linux executive, like the CBS algorithm. SSSA is collaborating with UPVLC for this purpose, and the CBS will be provided as optional component of the RT-Linux scheduler in Workpackage 5 of the project.

However, we also need a way to analyse the schedule and being able to provide QoS guarantees at the Linux level. The analysis of such a hierarchical scheduler is still missing from the real-time system literature. Some hierarchical system based entirely on fixed priority schedulers has been analysed by Saewong et al. [Sae02]. Statically scheduled hierarchical systems have also been addressed by Feng and Mok [Fen02]. However, no methodology exists to derive the parameters of the scheduling depending on the application characteristics. Also, it is not currently possible to analyse hierarchical schedulign systems based on EDF. In the OCERA project we will explicitly address this topic.

SSSA is also involved in the FIRST IST project. One of the goal of the FIRST project is to study the composition of schedulers, and the analysis of such hierarchical systems. Therefore, the two projects, OCERA and FIRST, will collaborate to this end by exchanging ideas and results.

Finally, it is important to note that the ARTIST network of excellence has considered the study of resource reservation and of hierarchical schedulers an important issue for future real-time systems research.

Chapter 3. Resource Management

3.1. Allocating Resources

Real-time systems have traditionally been thought of as static systems, in which tasks' parameters are known in advance, and scheduling can be performed based on this a-priori knowledge. For example, when tasks' minimum inter-arrival times and maximum execution times are known a-priori, it is possible to perform an admission test in order to guarantee that all the deadlines will be respected.

In recent years, a considerable amount of work has been done to apply real-time techniques to new kinds of time-sensitive applications, which are inherently more dynamic than classical real time applications. Examples are multimedia streaming programs, video/audio players, software sound mixers, and embedded systems used in data-intensive contexts, where relatively high volumes of sensor data are flowing and must be processed and analysed in real time.

Due to their temporal constraints, these systems are great candidates for using real-time techniques; however, they present new challenges due to the variability and unpredictability of their processing times, and to the data-dependent processing requirements that characterise them. When multiple real-time tasks of this type share the same CPU, allocating resources to them becomes difficult, and the tasks can be properly served only if the resource scheduler is able to cope with the high variance and unpredictability of their requirements. To better understand the scheduling problem in these conditions, two different situations should be considered:

1. The sum of the average amount of resources needed by all the tasks is bigger than the amount of available resource
2. The sum of the average amount of resources needed by all the tasks is less than the amount of available resources

In the first case, the system is said to be *overloaded*, and it is impossible for the scheduler to serve the applications so that all their QoS constraints are respected. To enable the scheduler to properly schedule all the system tasks, applications must implement some kind of QoS adaptation. This mechanism can be an effective policy for reducing their requirements and removing the overload condition [Abe01], otherwise the scheduler will choose which tasks to privilege based on some user-defined importance [Abe99-3].

When the amount of available resources is sufficient for serving all the applications so that they respect their QoS constraints, the scheduling problem becomes more interesting, since a properly designed scheduler could provide a considerably better QoS compared to a conventional scheduler. Since the tasks running in the system are characterised by unpredictable behaviour, it is important to provide *temporal protection* (as described in the previous chapter), so that each task is protected from the fluctuations in the resource requirements of the other ones.

It is clear that we need some way of dynamically adapting the scheduling parameters to the actual workload. We propose to do this, by using a feedback controller to monitor and adapt the reservation to the observed requirements. In other words, we believe that a combination of feedback scheduling techniques and resource reservations is a useful technique for properly serving time-sensitive applications in a modern multimedia OS. Some of the advantages of this combined use of feedback and reservation techniques are:

- Better *portability* of real-time code: using a feedback reservation-based scheduler, the performance of the application does not depend on the execution time estimation.

Thus, the application can easily run on different machines and achieve a predictable QoS level.

- A higher-level *programming interface*: the use of a feedback mechanism in a reservation-based scheduler permits the implementation of high level task models that separate the task parameters from the scheduling parameters
- *Robustness* to variations in execution times, either caused by DMA, caches, PCI bus masters, and similar mechanisms, or by the interrupt handling overhead. Adaptive schedulers can help to cope with the unpredictabilities caused by interference from other tasks which affect cache hit rate, memory contention, pipeline effects, and so on.
- Increased system *efficiency*: one of the biggest problems of reservation systems is that resource reservations are often over-dimensioned, wasting system resources. Using a feedback mechanism, each reservation can be automatically adapted to the application's real requirements, and an explicit reclaiming mechanism is not necessary anymore. Note, however, that the degree of efficiency gained using a feedback scheduler depends on the "speed and accuracy of the adaptation", or, to be more precise, on the dynamics of the closed loop system.

Clearly, the efficiency requirement contrasts with the requirement of maximising the QoS perceived by each single application. Using a feedback scheduler, it is possible to specify the dynamic behaviour of the closed-loop system, enabling trade-offs between efficiency and QoS. In other words, the problem of providing high system utilisation and high QoS to applications can be decomposed into two simpler problems: 1) designing a feedback controller that is stable and provides a specified closed-loop dynamics, defined in terms of overshoot and response time, closed-loop poles, etc. and 2) choosing the closed loop dynamic that provides the desired QoS/utilisation trade-off.

3.2. Feedback Scheduling

One of the biggest problems of CPU reservation systems is the selection of the fraction (or bandwidth) of the CPU assigned to each task. In presence of wide variations of the required computation time, one could either give the task too low or too high a bandwidth. In the former case the system experiences unacceptable degradations in the offered QoS, whereas in the latter case system resources are wasted. To cope with this problem, many authors proposed the use of feedback control mechanism inside the operating system. A first proposal of this kind for time sharing systems dates back to 1962 [Cor62]. More recently, feedback control techniques have been applied to real-time scheduling and multimedia systems. However, little theoretical analysis of such mechanisms has been provided. One of the main problems that hinders this type of analysis is the lack of a realistic and analytically tractable model for "the plant", i.e. for a real-time scheduler. In [Abe01], this gap has been filled in for CPU reservation schedulers: the authors proved that a CPU reservation scheduler for a task can realistically be modelled as a switching and parametric dynamic system (the varying parameter being the computation time of each activation of the task). The authors also addressed the problem of developing a feedback scheduler based on resource reservations. They proposed to use tools from control theory to design a feedback mechanism that imposes a closed-loop dynamic of the system.

In this workpackage, we will provide an implementation of such techniques in Linux, based on the Resource Reservation Scheduler described in Chapter 2, *Resource Reservation in Linux*. The feedback mechanism will be implemented as a Linux module, called qman, that can be an optional dynamically loaded (and hence activated in the system) at any time.

A task (process) that wants to use the qman services, will have to send a "request" to the module. The request can be accepted or refused. In case it is accepted, a CBS server is created for the task and is inserted in the queue of CBS ready tasks. The initial budget can be specified as an optional parameter of the request.

A user can also require a "Quality of Service monitoring" for a number of tasks, where the sensitive data collected by the QoS manager can be traced on a log file (requires system privileges).

There will be two classes of service: guaranteed service, and best-effort service. In the guaranteed class, a task has to specify its temporal parameters as an initial contract. This request can be accepted, if there are enough resources to guarantee a minimum amount of service, or refused in case there is not enough bandwidth. The task can optionally specify a "responsiveness parameter", which represents the level of responsiveness of the feedback controller. The use of feedback in this class is only optional, in that a task can also be assigned a fixed reservation. In this case, the task QoS performance will only depend on its own temporal behaviour.

In the best-effort service class, a task is always accepted unless the system is in permanent overload situation. Clearly, it will recieve an acceptable service (i.e. enough bandwidth) only when the system load is below a threshold.

3.3. Security

Since we decided to export the CBS scheduling facilities through the standard sched_setscheduler() syscall, the CBS module must “intercept” setscheduler calls. In our evaluation, this is the appropriate place for performing the checks in order to avoid the security problems intrduced above. In particular, we want to avoid that an unprivileged user allocates all the CPU time for his/her tasks, starving all the other users’ tasks and even the root’s activities and the system’s daemons.

In our design, the scheduling module does not perform any admission test at all, and a different security module can intercept the proper calls and implement the acceptance policy. We believe that two alternative solutions are possible:

- The security module permits the creation of reservations only to a particular user (for example a superuser, or a privileged user). A user level daemon having this user’s rights will be responsible for the security tests and the schedulability tests.
- The security module implements the security policy (and the schedulability test) itself.

We decided to implement both solutions, and we will provide two different security modules, that intercept the setsched hook before the CBS module. When the setsched handler provided by the security module is invoked, it enforces the security policy, returning -1 if the security check fails, and invoking the setsched handler of the CBS module if the security check is passed.

This approach for implementing security modules is similar to the one used by LSM (Linux Security Modules), which uses hooks in the kernel for implementing security policies.

As said above, we will develop two different security modules. The first one tries to move the policy to user space (leaving only the mechanism in the kernel). In practise, every time that a user tries to create a reservation, the security module checks if the user has root privilege. If not, the sched_setscheduler() call fails. This is the same mechanism regularly used by Linux to protect POSIX fixed priorities. A user level task, running with root privileges, will be in charge of implementing the security policy, by receiving

users' requests (through a Unix socket, or a named pipe), deciding if they are acceptable, and eventually performing the proper system call.

The first solution has the advantage that the user level daemon can be controlled through a configuration file (living in the /etc directory) to fine-tune the implemented policy. On the other hand, it is probably less secure, because attacking an user level daemon can be easier than breaking a policy implemented at kernel level. Hence, we will develop a second module, which directly implements the security policy in kernel space. This solution is obviously less flexible (the kernel module cannot parse a configuration file), but we expect it to be more secure.

An example of very simple security policy is to allow a user to create a reservation if only if the utilisation of all the non-root reservations is less than Bmax, with Bmax < 1. Note that since we were going to implement an admission test for security reason, we can also include the schedulability test in it.

Chapter 4. Resource Management Components

In this chapter, we shortly describe the components that will be developed as part of Workpackage 4, "Resource Management". For each component, we describe the basic structure of the implementation, measurable objectives to be checked at the end of the implementation phase and relationship with other OCERA components.

4.1. Generic Scheduler Patch

Description

It is a small patch for the Linux kernel that provides useful hooks to the Linux scheduler.

Design

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, so that the CBS scheduler can deallocate the resources associated to it;
- the **block_hook** is invoked when a task blocks, so that the CBS module understands that the current job is finished;
- the **unblock_hook** is invoked when a task wakes up, so that the CBS scheduler is notified of a new job arrival;
- 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.

Objectives

This patch has to be minimally invasive to limit the overhead and to minimise the dependency of the particular Linux version. To evaluate the overhead of the proposed patch we will measure duration of the system primitives with or without the patch.

Relationship with other components

The Resource Reservation Scheduling (described next) module requires this patch.

4.2. Integration Patch

Description

This patch will integrate the "preemption patch" for the Linux kernel with the RT-Linux patch.

Design

The "preemption patch" will be analysed and compared with the RT-Linux patch. Once again, the proposed modifications have to be minimal to make both patches work together.

Objectives

This patch has to be minimally invasive. An evaluation of the work on this patch is on-off: either the proposed modifications are correct, and the system works well, or they are not correct and the system does not work!

Relationship with other components

The preemption patch improves the responsiveness of Linux application. Therefore its use is optional for the Resource Reservation Scheduler.

4.3. Resource Reservation Scheduling Module

Description

It will be a dynamically loadable module for the Linux kernel that will provide a resource reservation scheduler for soft real-time tasks in user space. It will be based on the Constant Bandwidth Server. This module is the core of Workpackage 4 and it will be provided in different versions during the course of the project.

Design

A first version of this module will provide the basic resource reservation through the CBS algorithm. A second version will provide reclamation of the unused bandwidth. A third version will implement extensions for symmetric multiprocessors platforms (SMP). In addition, we will investigate the possibility to extend such techniques to resources different from the processor, as the network, the disk and the memory. Finally, We will also investigate the possibility to provide inheritance mechanisms to deal with priority inversion. All the versions of this module will be provided as a loadable module in the Linux kernel, to exploit its modular nature. Every module will be based on the Generic Scheduler Patch. In particular, it will install new functions for each one of the provided hooks. The user will access the services of such module through the normal Linux API, the *sched_setsched()* and *sched_getsched()* primitives. This access can be regulated by some customisable access policy.

Objectives

This module will provide temporal isolation between Linux processes/threads. Therefore, first we will evaluate to which degree two Linux activities can be temporally isolated with these techniques. Experimental evaluation will be based mostly on multimedia applications that are of interest to the consortium.

In the second phase, we will evaluate the reclamation techniques that we will implement. In particular, we will make experiments to evaluate the reclamation abilities of our implementation with respect to normal Linux.

In the final phase we will evaluate the SMP scheduler. In particular, we will measure the maximum load achievable.

For all provided modules, we will also evaluate the overhead of the proposed schedulers, by comparing the duration of the normal Linux primitives with the duration of the modified primitives.

Relationship with other components

This module depends on the Generic Scheduler patch. It provides services to the Quality of Service Manager module.

Moreover, this module is useful also for the Fault Tolerant Workpackage (TBD: insert number): some fault protection mechanisms need the temporal protection property and will be based on the services provided by the resource reservation modules.

4.4. Quality of service Manager

Description

It will be a dynamically loadable module for the Linux kernel plus a process daemon that will provide QoS management services. In particular, it will provide a mechanism for identifying the temporal characteristics of a task and adjust its scheduling parameters so to maximise its quality of service.

Design

As explained in 3, *Resource Management*, the algorithm will be based on a feedback scheduling mechanism. The QoS module will accept requests from user processes for a certain QoS level. The requirements will be expressed as "type" of the required service (hard, soft or best effort), and with quantitative parameters (i.e. in the case of hard requirements, the process must specify the worst case budget and the period of the required reservation).

The QoS can decide to accept or reject such requests. Once accepted, the process will be assigned a server with proper parameters. The process QoS will be constantly measured and the server parameters adjusted in order to maximise the quality of the service experienced by the user. However, if the process does not respect the initial contract, the QoS will be bounded within a certain limit.

This module will optionally provide a customisable access policies. In particular, it will be possible to restrict the kind of requests that can be made only to authorised users: for example, only trusted users can require a hard reservation.

The component will consist of dynamically loadable kernel module (QoS module) that will measure the QoS and implement the control algorithm. In addition a daemon process will accept the requests and send commands to the Scheduling module and to the QoS module. The daemon process will also enforce the access policy and can be customised through a configuration file.

Objectives

The objectives of this module are: to maximise the qos experienced by the user processes; to maximise the resource utilisation; to protect the access to the qos services by means of a customisable access policy. To evaluate its effectiveness we will provide theoretical proofs of the properties of the control algorithm. Moreover, we will experimentally measure the effect of such policy on the quality of service experienced by the user with respect to the behaviour under normal Linux. Finally, we will measure the overhead of such techniques.

Relationship with other components

Of course, this module depends on the presence of the Resource Reservation scheduling module.

4.5. User API

Description

This component is a set of one or more libraries that will provide a convenient API to the user to access the Resource Management services.

Design

This API will be as similar as possible to the POSIX API provided by the RTLinux executive: in this way, it will be possible to move a RTLinux thread from the hard real-time level to the soft real-time level, and vice versa, with little effort. The library will also be transparent to the underline Linux configuration.

Objectives

To provide transparent and easy access to the Resource Management services. We will measure the effectiveness of our approach by measuring the amount of code to be changed to port a RT-Linux thread to execute under Linux (ideally it should be 0 lines of code).

Relationship with other components

To work effectively, this library depends on the presence of the other Resource Reservation modules.

Bibliography

[Abe98] Luca Abeni and Giorgio Buttazzo, *Integrating Multimedia Applications in Hard Real-Time Systems*, December 1998, IEEE Real-Time Systems Symposium.

[BWI] Gerardo Lamastra, Giuseppe Lipari, and Luca Abeni, *A Bandwidth Inheritance Algorithm for Real-Time Task Synchronization in Open Systems*, December 2001, IEEE Real-Time Systems Symposium.

[Abe01] Luca Abeni and Giorgio Buttazzo, *Hierarchical QoS Management for Time Sensitive Applications*, May 2001, Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS 2001).

[GRUB] Giuseppe Lipari and Sanjoy Baruah, *Greedy reclaimation of unused bandwidth in bandwidth sharing servers* , June 2000, IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems.

[CASH] Marco Caccamo, Giorgio Buttazzo, and Lui Sha, *Capacity Sharing for Overrun Control* , Dec. 2000, Proceedings of the IEEE Real-Time Systems Symposium .

[Abe99-3] Luca Abeni and Giorgio Buttazzo, *Adaptive Bandwidth Reservation for Multimedia Computing*, Dec 1999, Proceedings of the IEEE Real Time Computing Systems and Applications.

[Cor62] F. J. Corbato, M. Merwin-Dagget, and R.C. Daley, *An Experimental Time-sharing System*, May 1962, Proceedings of the AFIPS Joint Computer Conference.

[Abe01] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole, *Analysis of a Reservation-Based Feedback Scheduler*, Dec 2002, Proc. of the Real-Time Systems Symposium.

[MCBS] Sanjoy Baruah, Joel Goossens, and Giuseppe Lipari, *Implementing constant-bandwidth servers upon multiprocessor platforms* , Sept. 2002, Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium.

[Sae02] Saowanee Saewong, Ragunathan Rajkumar, John P. Lehoczky, and Mark H. Klein, *Analysis of Hierarchical Fixed-Priority Scheduling* , June 2002, Proceedings of the 14th IEEE Euromicro Conference on Real-Time Systems .

[Fen02] Xiang Feng and Al Mok, *Proceedings of the 23^{rd} IEEE Real-Time Systems Symposium* , Dec. 2002, .