Implementing Resource Reservations in Linux

Luca Abeni and Giuseppe Lipari
ReTiS Lab, Scuola Superiore Sant'Anna,
Piazza S. Caterina, Pisa, Italy
{luca, lipari}@sssup.it

Abstract

With the recent development of kernel patches for implementing kernel preemptability and high-resolution timers and the acceptance of the first patch in the 2.5 development branch, it is now possible to support real-time applications in Linux user-space. However, real-time applications must be ran using the POSIX fixed priority scheduler, and we believe that this solution presents a number of problems. In our opinion, a workstation OS must support temporal protection among applications, so that users can access real-time scheduling facilities without the danger to starve the system. The Resource Reservation framework is a class of techniques that can be used for providing temporal protection among different real-time and non-real-time applications. Algorithms based on the Resource Reservation framework have been implemented in a number of Linux variants (the most notable is Linux/RK). In this work, we propose a novel implementation that differs from the previous ones in some fundamental points. Thanks to the particular reservation mechanism (the Constant Bandwidth Server - CBS) and to a careful design of our scheduling infrastructure, our scheduler can correctly cope with aperiodic task arrivals.

1 Introduction

In recent years, there has been a great interest in running new kind of applications characterised by soft real-time constraints on desktop and general-purpose Operating Systems (OSs). Typical examples are video conference and media streaming applications, software audio mixers, CD burners, etc. These applications are characterised by *implicit temporal constraints* that must be satisfied to provide the desired Quality of Service (QoS). We refer to these as *time-sensitive* applications.

To support time-sensitive applications, a general-purpose OS must respect the application's temporal constraints and hence a predictable schedule is needed. This implies that low kernel latencies and high-resolution timers are needed [6]. Nowadays, effective patches for implementing high-resolution timers [8] on Linux and for decreasing the Linux kernel latency have been proposed. Thanks to these patches, it is now possible to schedule real-time applications in user-space, guaranteeing their timing constraints. However, the only scheduling support for real-time activities provided by the Linux kernel is given by the POSIX API, which provides fixed priority scheduling.

Although fixed priority scheduling is an excellent solution for implementing real-time activities in embedded systems, it is not well-suited for soft-real-time scheduling in general purpose OSs. Important problems are the fairness and the security of such schedulers. In fact, if a regular user (that has not superuser privileges) is enabled to access the fixed priority scheduler, a simple malicious denial of service attack could be to activate a task at the highest priority, which immediately goes into an infinite loop. On the other hand, if only superusers are allowed to access the real-time scheduling facilities, it is very difficult to provide soft real-time guarantees to non-privileged users. Moreover, even trusted users could starve the system during debugging.

For these reasons, a real-time workstation OS should support a scheduler that provides *temporal protection*: the temporal behaviour of a task should not be affected by the other tasks in the system, enabling all the users to access the kernel's real-time facilities without being able to starve the system. In our opinion, temporal protection is as important as memory protection, which is provided by all the most common OS kernels.

Resource Reservations [15] have been proven to be an effective way for providing temporal protection. The concept of Resource Reservation is not new, and scheduling algorithms based on it have been imple-

mented in a number of Linux variants (the most notable is Linux/RK [18, 17, 16]). However, we believe that most of the previous implementations suffer some problems, going from scheduling anomalies caused by aperiodic activations to the lack of security policies. These problems are not due to intrinsic deficiencies of the resource reservation abstraction, but they depend on the scheduling algorithm used for implementing the reservation.
In this paper, we propose a novel implementation of a reservation scheme in the Linux kernel, based on the CBS algorithm [3]; our implementation fundamentally differs from the previous ones in the following aspects:

- it is non-intrusive,
- it permits to implement advanced security policies,
- it correctly copes with aperiodic activations/deactivations,
- and it takes particular care in maintaining compatibility with standard Linux.

The remaining of the paper is organised as follows: in Section 2 the addressed problems are described; in Section 3, we explain our design goals; in Section 4 we describe the implementation of our scheduler; in Section 5 the overhead and the performance of the proposed implementation are evaluated; finally in Section 6 we state our conclusions and we give some insight about our future work.

2 The Problem

Real-time applications are characterised by temporal constraints, which can be better described by considering a task [1] τ_i as a stream of *jobs* (or *instances*) $J_{i,j}$. We say that a job arrives when the task unblocks (i.e., when it becomes ready for execution), and terminates when the task blocks (because it must wait some particular event before being ready for execution again); $J_{i,j}$ arrival time is denoted by $r_{i,j}$, and its finishing time is $f_{i,j}$. The required computation time of a job between its arrival and its finishing time is denoted by $c_{i,j}$. The worst case computation time of a task is denoted by $C_i = \max_j\{c_{i,j}\}$. Each job $J_{i,j}$ is characterised by a *deadline* $d_{i,j}$; we say that the deadline is respected iff $f_{i,j} \leq d_{i,j}$.
If the tasks are periodic (i.e., if $r_{i,j+1} - r_{i,j} = T_i$, where T_i is the task period) and their worst case execution time is known, the Rate Monotonic (RM) [13] or the Deadline Monotonic (DM) [9] scheduling algorithms can be used to guarantee that every task will respect its deadlines if an admission test is passed. For example, RM is guaranteed to generate a schedule that respects all tasks deadlines if $\sum_i \frac{C_i}{T_i} \leq 0.69$[2]. RM and DM scheduling are supported in Linux through the POSIX fixed priority scheduler.
However, the fact that the real-time guarantee depends on the estimation of the worst case execution time of each task makes the system fragile respect to errors in this estimate. If a task does not respect its estimated worst case execution time, a different task can miss its deadline; in other words, *there is no temporal protection between the tasks in the system.* Of course, this is not a problem in a safety critical dedicated system, because all tasks are accounted for at design time and if a worst case execution time is wrong, then the whole system design may be wrong, and must be corrected. On the other hand, in a desktop system, tasks are dynamically activated and we cannot account for them at design time. A misbehaving real-time task can jeopardise the schedulability of other tasks (even worse, a user can affect the QoS perceived by other users), and can starve all the system's tasks (denial of service).

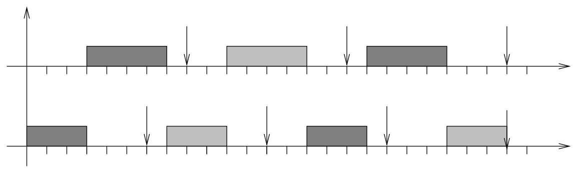

FIGURE 1: *Example of Reservation Scheduling*

A solution to this problem is given by Resource Reservations, which provide temporal protection by reserving each real-time task τ_i an amount of time Q_i every period T_i. Resource reservations are generally implemented by using an entity RSV_i called reservation to schedule a task τ_i. Each reservation RSV_i is associated a budget q_i, and is characterised by the two parameters Q_i and T_i. The budget q_i of reservation RSV_i is decreased when τ_i executes, and when it arrives to 0 RSV_i is said to be depleted. At the beginning of the next reservation period, q_i is replenished to Q_i, and RSV_i is said to be eligible. When RSV_i is in the depleted state, τ_i cannot be scheduled, whereas when it is in the eligible state τ_i is scheduled based on some real-time priority assignment, such as RM, DM, or Earliest Deadline First (EDF) [13].
As an example of reservation scheduling, consider Figure 1, representing the schedule generated by

[1] In this paper, the term "task" is used to identify either a process or a thread.
[2] This is just a sufficient admission test. More complex tests can be used to provide less pessimistic (or even a necessary and sufficient) guarantee.

two reservations RSV_1 and RSV_2 with parameters $Q_1 = 4$, $T_1 = 8$, $Q_2 = 3$, and $T_2 = 6$ when τ_1 and τ_2 are always ready to be executed. The underlying scheduling algorithm is EDF. At time 0 both tasks are ready for execution, and both the reservations have a budget $q_i > 0$; τ_2 is scheduled because RSV_2 has a shortest deadline than RSV_1 (remember that the underlying scheduling algorithm is EDF). At time 3, q_2 arrives to 0, hence RSV_2 is depleted and τ_2 cannot execute until time 6; as a result, τ_1 is scheduled. At time 6, q_2 is recharged, RSV_2 becomes eligible, and τ_2 can be scheduled, but it does not preempt τ_1 because RSV_1's deadline (8) is shortest than RSV_2's one (12). At time 7, $q_1 = 0$, hence τ_1 is depleted, τ_2 is scheduled, and so on. The schedule shows that both the tasks always receive the reserved amount of CPU time.

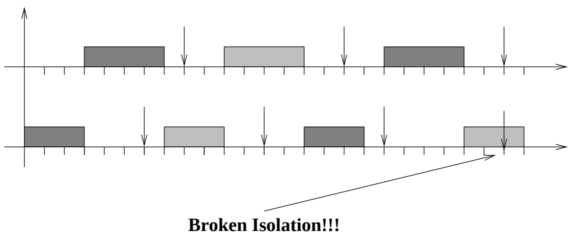

FIGURE 2: *Scheduling Anomaly in a Reservation System*

The reservation concept has been proven to be very effective and useful [14, 15, 18, 3], but the most common implementations (based on an abstraction similar to the Deferrable Server [20]) are not able to provide full temporal protection in some cases.

For example, consider Figure 2, which shows the schedule generated by the system of Figure 1 if τ_1 is not ready for execution (for example, because it is blocked by a blocking system call) at time 16, when RSV_1 becomes eligible. Since at time 17 τ_2 is depleted and τ_1 is not ready for execution, the CPU is idle between 17 and 18, when τ_1 becomes ready. Hence, τ_1 starts to execute at time 18 and continues until time 22, when its capacity arrives to 0 and τ_1 is depleted. At this point, τ_2 can execute, but it is not able to get all the 3 reserved units of CPU time before time 20. Hence, *the temporal protection is broken.* This is a very well known problem in real-time literature, and is due to the use of a Deferrable Server mechanism for implementing reservations.

Another problem with traditional implementations of resource reservation is that a task can consume all the CPU by issuing malicious requests with a particular pattern: it creates a reservation, then uses it until an instant before the depletion, destroys it, creates a new one, and so on. Again, the task could consume all CPU time starving all the other tasks.

Similar problems may occur when periodic tasks are scheduled in a reservation system. In fact, a periodic task τ_1 with worst case execution time C_i and period T_i is guaranteed to respect all its deadlines if it is scheduled by a reservation RSV_i with the same period of the task and $Q_i \geq C_i$. However, this is true if and only if τ_i and RSV_i are synchronised, that is to say if tasks arrivals happens exactly at the beginning of reservation periods. If this condition is not verified, the condition for ensuring the schedulability of τ_i is way more pessimistic than $Q_i \geq C_i$. Moreover, it may happen that τ_i is not ready for execution (because a job is finished and the next one is not arrived yet) when the budget of its reservation is replenished, and temporal isolation may be compromised. Hence, synchronising a periodic task with its reservation results to be fundamental for ensuring the correct behaviour of the system. However, such a synchronisation is not easy to achieve, and is an open issue in a lot of reservation systems.

Finally, reservation systems can be dangerous if a proper policy for controlling the amount of resources allocated to user tasks' is not used. In fact, an user could allocate almost all the CPU time to some greedy process which never releases the CPU (because of an error or because of a malicious behaviour). As a result all the other processes in the system would hardly have a chance to execute; since the correct behaviour of a Linux box depends on the execution of some daemons, starving them would be particularly dangerous. In multiuser systems, there are even more issues, because a single user would be allowed to acquire the complete control of all the system time, avoiding the execution of root's processes. Most of the existing reservation system generally perform an admission test when a reservation is created, to check that it will not compromise system schedulability. However, this admission test does not address the security issues highlighted here.

3 Design of the Scheduler

Our reservation-based scheduler was explicitly designed for solving the problems presented by traditional reservation systems presented in Section 2. In particular,

1. we use a sophisticated scheduling algorithm, based on the Constant Bandwidth Server (CBS) [3] to avoid problems with aperiodic activations
2. we designed the scheduler so that it requires minimal modifications to the kernel
3. we provide hooks in the scheduler for implementing advanced security policies.

3.1 The Constant Bandwidth Server

The Constant Bandwidth Server (CBS) is an efficient service mechanism developed for implementing CPU bandwidth reservations in a dynamic priority system (based on EDF scheduling). Each task τ_i is assigned a dynamic *scheduling deadline* d_i^s (not to be confused with the job's deadline $J_{i,j}$) by a server S_i; then, it is inserted in an EDF queue, ordered according to the scheduling deadlines. The server assigns scheduling deadlines to jobs so that each task is reserved an amount of CPU time Q_i every *server period* T_i^s.
We will now briefly describe the rules of the CBS algorithm. The interested readers can find more details in [3].

- Each server S_i is characterised by an ordered pair (Q_i, T_i^s), where Q_i is the maximum budget and T_i^s is the server period. The ratio $B_i = Q_i/T_i^s$ is the server bandwidth. The server maintains two internal variables, the current remaining budget q_i and the dynamic scheduling deadline d_i^s. At the beginning $d_i^s = 0$.

- If τ_i is served by S_i, then each job $J_{i,j}$ is assigned a scheduling deadline equal to the server deadline d_i^s.

- the job with the earliest scheduling deadline is selected to execute, according to the EDF policy.

- Whenever a served job executes, the budget q_i is decreased by the same amount.

- When $q_i = 0$, the server budget is recharged at the maximum value Q_i and the server deadline is postponed by T_i^s: $d^s = d^s + T_i^s$. The EDF queue is update accordingly.

- When a new job $J_{i,j}$ arrives (i.e., when τ_i unblocks) at time $r_{i,j}$, if $q_i \geq (d_i^s - r_{i,j})\frac{Q_i}{T_i^s}$, then a new scheduling deadline $d_i^s = r_{i,j} + T_i^s$ is generated, and q_i is recharged to the maximum value Q_i, otherwise the job is served with the last server deadline d_i^s using the current budget.

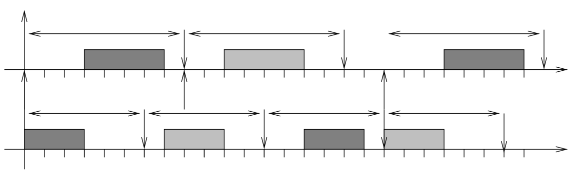

FIGURE 3: *Example of CBS Scheduling*

The CBS algorithm is work-conserving (i.e., the CPU is never idle if there is at least a task ready to execute), can correctly cope with aperiodic activations, and provides good performance, as shown in [3]. Figure 3 shows how the CBS solves the problem highlighted in Figure 2: when τ_1 arrives late at time 18, the server deadline is set to $d_1^s = 18 + 8 = 26$, so τ_1 scheduling deadline (26) is greater than τ_2 scheduling deadline (24). As a result, τ_2 is scheduled at time 18, and temporal protection is preserved.

3.2 The Scheduling Module

According to the definition of the CBS algorithm, the scheduler must intercept job arrivals (i.e., task unblockings) and job terminations (i.e., task blockings). Moreover, the scheduler must know when tasks are created and destroyed. To minimise the amount of modifications needed to the kernel, we decided to export all these relevant events to a CBS scheduler, which then decides the task to schedule and communicates to the kernel scheduler which task to dispatch.
In this way, the CBS scheduler has been implemented as a loadable module that can be inserted into the kernel at runtime, and most of the scheduler code is independent from the kernel version. Hence, it is very easy to port our 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 [8]: as a future work, we are planning to test our implementation with the kernel preemptability patch.

3.3 Security

We decided to export the CBS scheduling facilities through the standard `sched_setscheduler()` syscall, by adding a new scheduling policy `SCHED_CBS` and by extending the `sched_param` structure. This solution implies that the CBS module must "intercept" setscheduler calls (see Section 4 for more details). Since this system call must already be intercepted, we decided that this is the appropriate place for performing security checks.
In particular, 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:

1. The security module permits the creation of reservations (that is, setting the scheduling policy of a process to `SCHED_CBS`) only to a partic-

ular 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.

2. The security module implements the security policy (and the schedulability test) itself.

We decided to implement both solutions, providing two different security modules.

4 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 described in Section 3.
To intercept these events, the module must be inserted in a patched kernel, which exports 6 hooks: `fork_hook`, `cleanup_hook`, `block_hook`, `unblock_hook`, `setsched_hook`, and `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`, and when the CBS module is inserted it sets these pointers to its handlers, so that the kernel will invoke the proper handler when a 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 fallback.

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` which 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).

Based on the described patch, the CBS scheduler can be easily implemented: in fact, there is a clear relationship between the hooks provided by the patch and the description of the CBS algorithm given in Section 3.

For example, the `unblock_hook`, corresponding to a job arrival, is implemented as follows: first of all, the test $q_i \geq (d_i^s - r_{i,j})\frac{Q_i}{T_i^s}$ is performed, updating d_i^s and q_i if needed (the server deadline is implicitly assigned to the task). Then, the unblocked task is put in the EDF queue, and if it results to be the first of the queue it is scheduled. When scheduling the new task, the budget of the descheduled task (if any) is updated, and a *depletion timer* is set to fire when the newly scheduled task will exhaust its budget. Figure 1 shows the code for `unblock_hook`[3]: after acquiring a spinlock for protecting the scheduler code, the time is read, and a function implementing the described algorithm is invoked. Note that `t->deadline` is d_i^s, `t->max_budget_clock` is Q_i, `t->period_clock` is T_i^s, and `t->c` is q_i.

The `cbs_schedule()` function will dispatch the scheduled task. Since we decide not to modify the original Linux scheduler, the module forces the dispatch of a task by setting its policy to `SCHED_FIFO` or `SCHED_RR`, and setting the `rt_priority` field in its task structure to the maximum Linux real-time priority + 1 (100).

Note that in our implementation the scheduling algorithm does not depend on tasks' periodicity: in fact, the scheduler directly intercepts tasks' activations/deactivations, and implementing a periodic behaviour is the tasks' responsibility. For example, the `setitimer()` syscall or POSIX timers [8] can be used for triggering periodic activations.

During the scheduler's development and debugging, we implemented a scheduling tracer, which also results to be an invaluable tool for system design. Because of the particular design of the scheduler, adapting general-purpose tracers like LTT [24] to our needs resulted to be difficult, hence we decided to develop a little tracer from scratch. The tracer provides its output through a device file, and a simple `cat` command can be used for generating a trace file, which

[3]note that all the debugging, logging, and tracing calls have been removed to simplify the code.

```
int cbs_activate(struct cbs_struct *t, unsigned long long r)
{
  /* CBS test: c > (d - r) * U ---> New deadline */
  if ((t->c > llimd(t->deadline - r, t->max_budget_clock, t->period_clock)) || (t->deadline < r)) {
    /* Generate new deadline */
    t->deadline = r + t->period_clock;
    t->c = t->max_budget_clock;
  }
  if (edf_list_add(t)) {
    /* We just modified the head of the queue... */
    if (exec) {
      if (update_used_time(exec, r)) {
        update_priorities(exec);
      }
    } else {
      last_update_time = r;
    }
    cbs_schedule(r);
  }

  return 1;
}

void generic_request(struct task_struct *t)
{
  unsigned long flags;
  unsigned long long int time;

  spin_lock_irqsave(&generic_scheduler_lock, flags);
  if (t->private_data != NULL) {
    time = sched_read_clock();
    cbs_activate(t->private_data, time);
  }
  spin_unlock_irqrestore(&generic_scheduler_lock, flags);
}
```

Figure 1: Handling job arrivals in the CBS module.

can be displayed using a portable visualisation program written in Java.

As introduced in Section 3, a task can create a reservation by using `sched_setsched()` to change the scheduling policy to `SCHED_CBS`. The CBS module intercepts this call through `setsched_hook`, hence a security module can easily implement a security policy by modifying the setsched hook to call a different handler. 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) [1, 23], which uses hooks in the kernel for implementing security policies.

We developed two different security modules, based on the two ideas introduced in Section 3. The first module tries to move the policy to user space (leaving only the mechanism in the kernel). In practice, 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[4]. 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

[4] Of course, a special user can also be created for this purpose

probably less secure, because attacking an user level daemon can be easier than breaking a policy implemented at kernel level.
Hence, we implemented 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. As an example, we implemented a very simple policy, allowing an user to create a reservation if $\sum_{\text{all tasks}} \frac{Q_i}{T_i^s} \leq 1$ and $\sum_{\text{non root tasks}} \frac{Q_i}{T_i^s} \leq B^{max}$, with $B^{max} \leq 1$ (note that since we were going to implement an admission test for security reason, we included the schedulability test in it).

5 Experimental Evaluation

In this section, we present an experimental evaluation of our reservation-based scheduler. First of all, we show that the hooks introduced in the kernel do not affect the system performance. Then we show that even when the CBS module is inserted in the kernel, the regular Linux scheduler does not degrade its performance, and we evaluate the overhead introduced by our scheduler.
After that, we show the behaviour of the CBS scheduler, by using our scheduling tracer. Finally, we evaluated the performance of our CBS implementation, and we show its effects on real applications.
All the experiments were performed on a Pentium 166 with 16MB of RAM.

5.1 Overhead Measurements

We used the LMBench [22] test suite to compare the performance of a vanilla 2.4.18 Linux kernel with the performance of the patched kernel.
First of all, we performed the comparison with a kernel having the hooks in place but no scheduling module inserted. We repeated 50 runs per kernel, and the results showed that no sensible variations in the performance can be noticed. After that, we repeated the experiments with the CBS scheduling module inserted in the kernel, obtaining similar results. Table 1 shows the numbers.
Finally, we instrumented our scheduling module, to measure the amount of time taken by the most important hooks (that are `block_hook` and `unblock_hook`). We ran a workload composed of tasks that continuously sleep for $10ms$, each one handled by dedicated CBSs, and we found out that after 1 day, the maximum time needed for a `block_hook` execution was $8\mu s$, and the minimum was $2\mu s$. The execution time of `unblock_hook` had a maximum of $17\mu s$ and a minimum of $5\mu s$.

5.2 Trace of the schedule

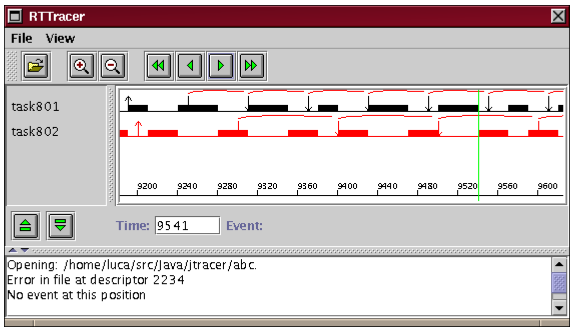

FIGURE 4: *Schedule visualisation with the tracer*

We tested the functionality of the scheduler by using our tracer program. As a first experiment, we created several time consuming tasks (that never release the CPU), scheduled through CBSs with $Q = 10ms$ and $T^s = 200ms$. By analysing the amount of time executed by each task, we verified that they shared the CPU fairly. We also varied Q_i and T_i^s, verifying that if $\sum_i \frac{Q_i}{T_i^s} \leq 1$ each task executed for a fraction $\frac{Q_i}{T_i^s}$ of the CPU time.
Finally, we performed some other runs using our tracer to check that the produced schedule matched with the expected one. For example, Figure 4 shows a snapshot of the trace produced by two time consuming tasks τ_1 and τ_2 scheduled by two servers with parameters $Q_1 = 20ms$, $T_1^s = 60ms$, $Q_2 = 30ms$, and $T_2^s = 100ms$.

5.3 Performance Evaluation

After evaluating the impact of our scheduling module on regular Linux applications and verifying the correct behaviour of the CBS scheduler, we performed some experiments to show the effectiveness of our CBS module in providing a controllable QoS to user applications.
As a test application, we selected `mplayer` [12], an advanced media player application that is able to reproduce many different audio/video formats and to perform accurate audio/video synchronisation.
First of all, we proved that the standard Linux scheduler is not adequate for managing concurrent time-sensitive applications: we ran two simultaneous instances of mplayer, each of which playing a video stream at 25 FpS (Frames per Second) and requiring about $32ms$ to decode a frame[5]. Hence, each player

[5]the frame decoding time is not constant; $32ms$ is an approximate estimation of the average time.

Kernel	NULL Call	NULL I/O	Fork Process	2p/0K ctxsw	2p/16K ctxsw	8p/16K ctxsw	Pipe Latency
Vanilla Linux	0.87	1.9040	1512.2	8.4340	81.400	98.300	29.800
Linux with hooks	0.87	1.8670	1523.8	8.5790	80.400	92.100	30.400
Linux with CBS module	0.87	1.8600	1525.7	8.7460	81.200	96	30.600

Table 1: Summary of LMBench results for vanilla Linux kernel, patched kernel without any scheduling module inserted, and patched kernel with the CBS scheduling module inserted. The table shows the mean value of 20 runs (all the confidence intervals are under 10%)

can be modelled as a periodic task with execution time $32ms$, period $1000/25 = 40ms$, and utilisation $32/40 = 0.8$ (each player requires a fraction of 0.8 of the CPU bandwidth). Since $0.8 + 0.8 = 1.6 > 1$, the system is overloaded and it is not able to satisfy each task's temporal requirements. We decided to quantify the QoS perceived by a player by measuring the difference between the time when a frame is played and the time when the previous one was player. We call this quantity the *Inter-Frame Time* (IFT). The IFT of a player should ideally be constant and equal to the player period (i.e., $1/FpS$).

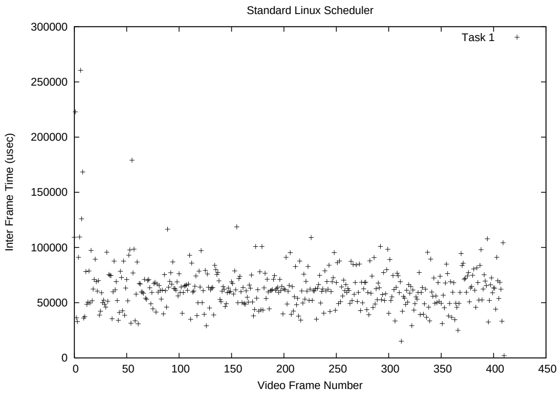

FIGURE 5: *Inter-Frame Times for the first player, when both the two players are scheduled using the standard Linux scheduler.*

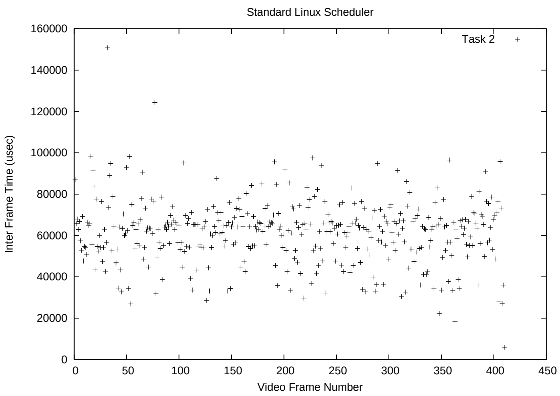

FIGURE 6: *Inter-Frame Times for the second player, when both the two players are scheduled using the standard Linux scheduler.*

Figures 5 and 6 plot the IFT for the two tasks, showing that none of the two tasks is able to get the expected QoS (in fact, both the two tasks have IFTs greater than $1/FpS = 40ms$).

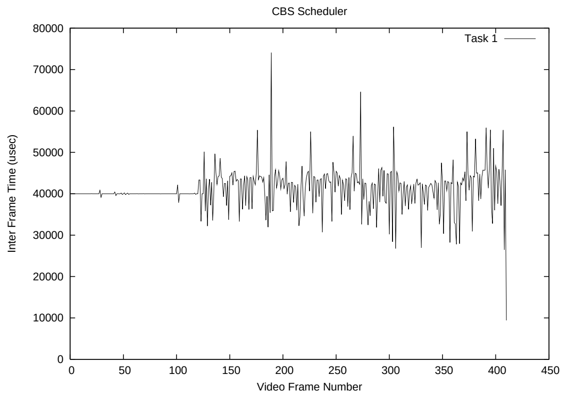

FIGURE 7: *Inter-Frame Times for the first player, scheduled by a CBS with $Q = 32ms$ and $T = 40ms$.*

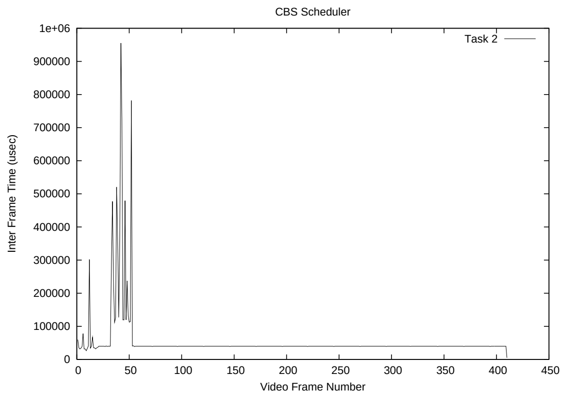

FIGURE 8: *Inter-Frame Times for the second player, scheduled by a CBS with $Q = 4ms$ and $T = 40ms$.*

The CBS scheduler can help to control the IFT of each player, by reserving a different fraction of the CPU bandwidth to each task. For example, we can schedule the two players with two servers having parameters $Q_1 = 32ms$, $T_1^s = 40ms$, $Q_2 = 4$, and

$T_2^s = 40ms$ to control the IFT of τ_1 to $40ms$. Since the players depend on the X server for displaying the video output, we scheduled the X server through a CBS with parameters $Q_x = 1ms$ and $T_x^s = 20ms$. This solution is similar to the PCP protocol [19] for managing the server (which is a shared resource); see [10] for more details.

The results of this second experiment are shown in Figures 7 and 8: after an initial transient (until about frame 120) in which the IFT is very stable because τ_1 is the only CBS task in the system, τ_2 starts and the IFT of τ_1 is affected by the second CBS, but is still controlled around the desired value: the variation in the IFT is never bigger than the player period.

Obviously, since the system is overloaded it is not possible to control the IFT of both the task, and τ_2 does not obtain the desired QoS. In fact, when τ_1 is active (until frame 50 of the second player), the IFT of τ_2 is completely out of control. By changing the scheduling parameters of the two CBSs, it is possible to control the number of deadline misses of each player, giving more importance to the first or to the second.

It can be easily seen that assigning the correct parameters to the two CBSs is not an easy task, since it heavily depends on the system, on the tasks, and on the input data (in this case, the media to be reproduced). However, such assignment is out of the scope of this paper; we plan to use feedback techniques for performing a correct assignment.

6 Conclusions and Future Work

In this paper, we described a new approach to reservation scheduling in Linux, and we presented our implementation of the CBS algorithm (a reservation algorithm based on EDF).

Our implementation differs from the previous ones because it can cope with aperiodic activations/deactivations, it is well-integrated in Linux without requiring big modifications to the kernel, and it provides hooks for implementing security policies. The CBS scheduler is currently alpha quality software. It is released under the GNU Public License (GPL) [11], and can be dowloaded from `http://hartik.sssup.it/~luca/cbs`. Everyone is welcome to download and give us some feedback.

This work has been developed in the context of the OCERA project [2], and our final goal is to provide full support for QoS sensitive and adaptive applications on Linux. Of course, providing resource reservations is only a first step in such direction: once the kernel provides reservations, and the possibility to precisely allocate resources to user tasks, a user level QoS Manager must be implemented.

We are currently designing a QoS Manager that will use the CBS scheduling module for reserving a fraction of the CPU bandwidth to each QoS sensitive application, and will use advanced resource allocation policies for determining the correct fraction of the CPU bandwidth to be allocated to each task. We expect that adaptive and feedback scheduling techniques [4, 21, 7] will result to be very effective. A prototypal implementation of the QoS manager has already been implemented on the HARTIK real-time kernel [5], and a porting has been tested on Linux/RK [17]. We believe that porting it to Linux + our CBS scheduler will be fairly easy, and will enable further research in the field of adaptive scheduling.

Acknowledgements

This work has been supported in part by the IST programme of the Commission of the European Communities, IST-2001-35102 (OCERA project).

References

[1] Linux security modules.

[2] Open components for embedded real-time applications. http://www.ocera.org.

[3] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[4] Luca Abeni and Giorgio Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hong Kong, December 1999.

[5] Luca Abeni and Giorgio Buttazzo. Support for dynamic QoS in the HARTIK kernel. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000.

[6] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. A measurement-based analysis of the real-time performance of linux. In *Proceedings of the IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)*, San Jose, CA, September 2002.

[7] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, November 2002.

[8] George Anzinger. High resolution timers project. http://high-res-timers.sourceforge.net.

[9] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, May 1991.

[10] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, London. UK, December 2001.

[11] Free Software Foundation. Gnu general public license. http://www.gnu.org/licenses/gpl.html.

[12] Arpad Gereoffy et al. MPlayer - Movie player for linux. http://www.mplayerhq.hu.

[13] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[14] Clifford W. Mercer, Raguanathan Rajkumar, and Hideyuki Tokuda. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[15] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[16] Shui Oikawa and Raj Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.

[17] Ragunathan (Raj) Rajkumar, Luca Abeni, Dionisio de Niz, Sourav Ghosh, Akihiko Miyoshi, and Saowanee Saewong. Recent developments with linux/rk. In *Proceedings of the Second Real-Time Linux Workshop*, Orlando, Florida, november 2000.

[18] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[19] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[20] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1989.

[21] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third usenix-osdi*. pub-usenix, feb 1999.

[22] Larry Mac Voy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, San Diego, CA, Jan 1996.

[23] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, San Francisco, CA, Aug 2002.

[24] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Technical Conference*, San Diego, CA, June 2000.