Abstract

This WiP describes some preliminary results obtained when experimenting with the priorities of IRQ threads in a real-time version of the Linux kernel. IRQ threads allow to schedule interrupt handlers so that their interference on real-time activities can be controlled. However, the experiments presented in this paper indicate that fixed priority scheduling does not provide enough flexibility for finding a trade-off between real-time performance and throughput, and we argue that reservation-based scheduling is needed.

1 Introduction

Real-time scheduling theory has traditionally dealt with the problem of scheduling the CPU so that the execution of a set of concurrent tasks can meet some timing constraints. The kind of real-time constraints considered range from hard real-time constraints (requiring strict and deterministic execution guarantees) to soft real-time constraints (for which occasional violations can be tolerated and probabilistic performance guarantees are required). Moreover, different kinds of tasks (ranging from tasks characterised by fixed execution and inter-activation times to tasks described by stochastic descriptions) have been analysed, and scheduling algorithms have been modified to address high variabilities in the task sets.

However, the CPU is not the only type of resource that needs to be shared between the various applications running in a system. Very often, real-time tasks need to interact with IO devices (e.g., acquiring data from sensors, or sending computation results to actuators) or with other nodes through a network link. The need for a real-time I/O creates problems of challenging complexity, which cannot be mitigated by simply using suitable real-time scheduling algorithms for the CPU.

Some recent pieces of work have shed some light on a largely underestimated problem: a real operating system kernel needs some CPU time to exchange data with hardware devices [3, 6]. For instance, it is completely useless to precisely schedule a device (e.g., a disk) if the kernel is not able to find enough CPU time to manage the incoming data. And, the CPU time spent by the kernel for handling the device must not be accounted to real-time tasks that do not use such a device (causing deadline misses) This means that a really coordinated strategy for the scheduling of different resources [9] is needed.

In this work in progress, we investigate how some recent patches for the Linux kernel permit to make IO activities schedulable, and we experiment with different priority assignments verifying that fixed priority scheduling does not provide enough flexibility for controlling both the real-time performance and the throughput of real-time and non-real-time applications coexisting in the same system.

2 Kernel Structure

As explained in the introduction, to schedule other resources than the CPU the OS kernel needs to consume CPU time in handling hardware interrupts coming from the various devices providing the resources. To understand why the time spent serving interrupts can be a problem for real-time applications, consider the structure of a traditional kernel, in which hardware interrupts are generally served in two phases:

- a short Interrupt Service Routine (ISR) is invoked as soon as an interrupt fires and is responsible for acknowledging the hardware interrupt mechanism, postponing the real data transfer and processing to a longer routine, to be executed later;
- a longer routine (soft interrupt, or bottom half) is executed later to correctly manage the data coming from the hardware device.

ISRs generally execute with interrupts disabled, while soft interrupts always execute with interrupts enabled and are served when switching from kernel space (where ISRs run)
to user space (where user programs are executed). Therefore, soft interrupts can be preempted by ISRs.

Both ISRs and soft interrupts have a higher priority than user tasks, and can "steal" execution time from them. Such “stolen time” can be accounted in real-time guarantees by modelling it as a blocking time, and/or by modelling ISRs and soft interrupts as high priority tasks\(^1\). This implies that a low-priority task can make a task set unschedulable by causing the generation of a large number of hardware interrupts.

This problem is generally solved in real-time kernels by scheduling the interrupt handlers: for example, the Real-Time Preemption patch (RT-preempt) [8] introduces real-time features in the Linux kernel and transforms ISRs and soft interrupts in kernel threads (the hard IRQ thread and the soft IRQ thread), that are schedulable entities handled by the kernel scheduler in the same way as user tasks (so, IRQ threads can have lower priorities than real-time tasks, and can be preempted by them). A real-time application that does not need to interact with a specific device can schedule its tasks in foreground respect to the device’s interrupt handlers, so that real-time tasks are not disturbed by the device’s interrupts.

This solution can present a slightly higher overhead, and requires a more careful synchronisation, but also has the advantage of permitting to correctly account the handler code in a real-time system (that is, the CPU time required to execute the handler can be correctly accounted in order not to break the system’s guarantees).

The possibility to schedule interrupt handlers (provided by IRQ threads) permits to give user-space real-time tasks higher priorities than interrupts, reducing the interference from hardware devices. However, it is still not clear how to assign priorities so that real-time and QoS guarantees are respected: although real-time theory provides tools for assigning priorities to real-time tasks (for example, by using the Rate-Monotonic - RM - priority assignment), there still are no reliable algorithms to properly assign priorities to IRQ threads.

Of course, it is easy to find priority assignments that provide good real-time performance in specific cases: for example, when real-time applications do not need to access a hardware device, the IRQ threads provided by RT-Preempt allow to reduce the interference caused by such a device. However, it is not easy to assign the tasks priorities when the real-time application depends on data coming from the device.

\(^1\)The schedulability of a real-time task set can be guaranteed by using an admission test, which is traditionally based on the execution times and periods of real-time tasks (utilization-based test, response time analysis, or time demand analysis). This admission test can be enhanced to account the blocking times, and/or by introducing in the admission test some high priority tasks modelling interrupt activities.

### 3 Scheduling the IRQ threads

Since there is not any theoretical model showing how IRQ threads affect devices throughputs and the performance of real-time tasks, we have assessed the effects of IRQ threads priorities through a set of experiments.

To evaluate the interactions between a set of periodic real-time tasks and a hardware device generating interrupts:

- a network card has been selected as an interrupt generating device because it is easy to generate a controlled load on it, and to measure the network throughput;
- a set of periodic periodic real-time tasks has been used to generate some time sensitive CPU load, and all the real-time tasks have been scheduled using real-time (SCHED_FIFO) priorities assigned according to RM;
- real-time performance have been quantified by measuring the latency [1] experienced by a periodic task. This latency is a good real-time performance metric, because it must be accounted in the admission test as a blocking time \(B_r\), so high latency values risk to make unschedulable task sets that would be schedulable if kernel effects were not considered.

The impact of IRQ threads’ priorities has been measured by repeating the experiments with different priority assignments. In particular, the goal of these experiments was to check how manipulating the priorities of the interrupt threads allows us to control the real-time tasks’ latency and the network throughput.

To reduce the impact of external factors, the experimental setup is composed of two computers connected by a cross network cable. The cyclictest program [5] has been used to measure the latency experienced by a real-time task with period 10\(ms\), and the netperf program [4] has been used to generate a very high network load and to measure the throughput achieved by the network card. One of the two computer generates the network traffic by using a netperf client, while the other computer runs the netperf server together with the set of real-time tasks and cyclictest. This second computer is an AMD K6-2@400Mhz\(^2\) running the 2.6.24-rc2-rt1 Linux kernel [7], and both the computers use a 100\(M\)b Realtek ethernet card.

The priorities of the cyclictest periodic task and of all other real-time tasks have been assigned according to RM, and the priorities of the IRQ threads serving the network card (the hardIRQ thread, and the softirq-net-rx thread - these two threads will be indicated as “networking threads”) have been varied from 1 (minimum priority) to 99 (maximum priority). To better

\(^2\)Note that we used a low-power computer by purpose, to better highlight the problems caused by the interrupt handlers.
<table>
<thead>
<tr>
<th>Priority</th>
<th>Maximum Latency</th>
<th>Net Throughput</th>
<th>95% confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 → 49</td>
<td>98µs</td>
<td>37Mbps</td>
<td>3Mbps</td>
</tr>
<tr>
<td>50 → 79</td>
<td>94µs</td>
<td>38.3Mbps</td>
<td>1.6Mbps</td>
</tr>
<tr>
<td>80</td>
<td>148µs</td>
<td>76.6Mbps</td>
<td>1.2Mbps</td>
</tr>
<tr>
<td>81 → 99</td>
<td>164µs</td>
<td>72.25Mbps</td>
<td>2.1Mbps</td>
</tr>
</tbody>
</table>

Table 1. Real-Time latency and network throughput experienced assigning different priorities to the interrupt threads.

![Figure 1. Latency CDF for a non real-time linux kernel running netperf.](image1)

![Figure 2. Latency CDF for a RT-Preempt linux kernel running netperf.](image2)

expose the effects of these two threads, netperf has been configured to use UDP packets composed by 600 bytes.

To have a baseline value to be used as a reference for the following results, a vanilla Linux kernel (without IRQ threads) has been used in a first set of experiments, which resulted in the latency Cumulative Distribution Function (CDF) depicted in Figure 1. Although the probability of measuring a latency < 200 is high, the distribution function has a long tail, and the maximum measured latency is 70602µs (note the logarithmic scale on the X axis in the figure). The corresponding network throughput is about 80Mbps.

While the achieved network throughput is reasonable, a worst-case latency of more than 70ms is not acceptable for a large number of real-time applications. Running the same experiments on a RT-Preempt kernel (without any tuning of the IRQ priorities) resulted in lower worst-case latencies, as shown in Figure 2 which compares the latency CDFs for an “-rt” and a vanilla kernel. Note that the CDF for the 2.6.24-rc2-rt1 ends before 100µs (so, the worst-case latency is less than 100µs), while the CDF for the vanilla kernel is truncated (as shown in Figure 1, it reaches 1 after 70ms). However, the network throughput for Preempt-RT went down to less than 50Mbps. To avoid this decrease in networking performance (while not renouncing to low latencies), we investigated the effects of tasks priorities the latency and throughput, through a new set of experiments.

Since some first experiments seemed to confirm that assigning the same priority to the hard IRQ thread and to the soft IRQ thread gives the best results, we decided to always assign priorities in this way. The experiments’ results showed four different possibilities for priority assignment:

1. the networking threads have the lowest priorities in the system. This includes all the priorities from 1 to 49 (50 is the default priority of all the IRQ threads);
2. the networking threads have priorities between 50 (the priority of all the other IRQ threads) and the priority of the periodic real-time threads (in particular, cyclictest, whose priority is 80);
3. the networking threads have the same priority as cyclictest;
4. the networking threads have the highest priority in the system. This include all the priorities ranging from 81 to 99.
Table 1 summarises the results obtained in the most relevant cases. In particular, it is possible to see that when the networking threads have priority from 1 to 49 (case 1), the latencies experienced by cyclictest are smaller than 100\(\mu s\) but the achieved network throughput is low.

The latencies and throughput measured in case 2 (network threads priorities between 50 and the priorities of the real-time threads) are basically equivalent to the ones measured in case 1.

When the networking threads are scheduled at priority 80, which is the same priority as cyclictest (case 3), the throughput measured by netperf increases to 76.1Mbps\(^3\), but the latency experienced by real-time tasks is increased by about 50\(\mu s\).

Further increasing the networking threads priority (case 4) increases the latency, but has no positive effects on the network throughput.

Unfortunately, the increase in latency is not gradual, so it is not possible to assign tasks’ priorities to obtain a latency between 100\(\mu s\) and 140\(\mu s\); in the same way, it is not possible to have a fine-grained control on the network throughput by only playing with priorities. Note that the throughput obtained by assigning to the IRQ threads priorities smaller than the real-time tasks priorities is very bad, and these priority configurations can be hardly considered useful.

The previous experiments show that fixed priority scheduling does not provide enough flexibility to control both real-time performance and hardware device throughput in an effective way. Hence, we argue that more advanced schedulers should be used for the IRQ threads; since the load of such tasks is highly variable and unpredictable (being generated by hardware interrupts which often do not follow any controlled arrival pattern), we believe that a scheduler allowing us to reserve a fraction of CPU time to IRQ threads would be more appropriate for scheduling them.

A first candidate for scheduling IRQ threads is the Completely Fair Scheduler (CFS) that has been recently introduced in the Linux kernel (and implements a form of Proportional Share scheduling). Unfortunately, some preliminary experiments seem to indicate that CFS is not yet able to provide latencies below 200\(\mu s\) (it is not clear if this is due to the CFS algorithm itself, or to implementation issues).

4 Future Work

This Work-in-Progress only reports preliminary results (which look very interesting, because they show that IRQ threads require scheduling algorithms more advanced than the traditional fixed priority one).

We are currently working on some experiments to check if CFS can be used to enforce temporal protection between IRQ threads and real-time applications running in user space\(^4\). We also plan to run some experiments using a Sporadic Server (which is included in the POSIX standard) to implement this form of temporal protection.

The temporal protection between tasks can also be obtained by using a reservation-based scheduler such as a CBS-based one [2]. Our prototype of CBS scheduler for Linux is compatible with the RT patch, and we are starting to use it for scheduling IRQ threads. We expect that the flexibility and guarantees provided by this scheduler will allow us to find a good trade-off between latency and throughput, but we have no numbers to show yet.

Finally, we plan to confirm the obtained results by using different interrupt-generating devices (for example, the hard disk controller) and different workloads.

After collecting a large amount of data through the previously described experiments, we aim developing a mathematical model allowing us to provide real-time and QoS guarantees by scheduling IRQ threads and to use the experiments results for validating the model.

References


\(^{4}\)Some preliminary results seem to indicate that CFS can easily provide temporal protection between tasks, but it cannot provide low latencies. We are still investigating the reason for this results.