Soft Real-Time Linux Overview

Georg Schiesser and Nicholas Mc Guire
Distributed and Embedded Systems Lab
Lanzhou University (SISE), P.R.China, 730000
0307201@student.tuwien.ac.at
mcguire@lzu.edu.cn

Abstract

As real-time capabilities are becoming relevant in main-stream desk-top Linux more and more, and new variants and ideas are popping up all over the Open-Source world, a notoriously incomplete, overview of the existing variants of soft-real time Linux is given in the article. Many of the variants are for special purposes, some are more general approaches that target every-day soft-rt demands, like audio/video streaming, finding your way through the many approaches and selecting the one that best fits your problem is not always easy, this article hopes to clarify a few basic terms and technological concepts, in the hope that it will simplify this selection process.

1 Introduction

In recent years computer capabilities have increased sharply. Many operating systems are multiuser multitasking systems. Although the performance of computers is steadily increasing the timing demands cannot always be satisfied because there is a overwhelming increase of complexity, resulting in a special class of tasks, namely those with real-time demands, being degraded.

The specifics of recent development have shown that soft-realtime is not only a demand in industrial computing but can be a demand in every day computing problems. Not so much at the pure functional level, but rather at the quality level. Most PCs can emit sound, but that is not sufficient to call it an audio system. Playing videos is not a specialized task, but playing them jitter free while other activities are going on on the system is a quality level that most systems can't offer.

This paper covers basic aspects of open source soft-real-time technologies available, in the hope to give the reader a sufficiently precise picture of available solution so that the direction of further investigation for a particular problem domain becomes clear.

2 Basic concepts

Before available solutions can be introduced we must be aware of the basic concepts. So I introduce some basic definitions and technologies.

2.1 Definitions

There are many different definitions of these terms floating around on the web so we present the definitions I use here in this paper, though we in no way claim to be the authoritative source of such a definition.

- non-realtime: In non-realtime applications average or typical times are more important. There are no critical deadlines. The worst case timing depends on the system load.
- soft-realtime: Systems that are realtime, but where missing a deadline now and then is acceptable are called soft-realtime systems. Missing a deadline degrades the quality but is not a failure.
- firm-realtime: There are hard deadlines, but low probability of missing a deadline.
- hard-realtime: Applications that have real, serious, non-negotiable deadlines. For these systems worst case timing is critical. Missing a deadline is a failure.

In this paper we are only concerned about the three basic concepts non-, soft- and hard-realtime.

2.2 Examples

Realtime covers a wide variety of applications, not only high-tech problems can mandate realtime capabilities but even fairly every-day situations.

If you have ever launched the CD burner under Windows NT you will know that you may not move the mouse while burning or the CD burners buffer will not get filled on time.

Did you ever try audio recording with your laptop and were unsatisfied with the results? It almost works.

There are several application fields which are based on soft-realtime, even if some are built on non-real-time OS they expect some form of soft-real-time behavior:

- Audio, Video: If loosing frames while recording or playing doesn't cause a critical failure you won't have to use a hard-realtime OS. The more frames you loose the worse the quality of the data, so this is a very good example for soft-realtime demands.

- Telecommunication: Transmitting voice data with a mobile phone is an example for soft-realtime too. You can understand the other person even if a view samples are lost, but if it gets too bad people will no longer be willing to pay for this service quality.

- Networking: Lost packets in a ATM network must be kept to an absolute minimum as the path reconstruction times have a very serious impact on the affected connection.

2.3 Why soft-realtime

Soft-realtime is used when you have a well defined class of important events. If missing an event only reduces the quality of your data but would not results in a failure of the application soft realtime may do. In contrast to non-realtime soft-realtime improves the quality. Furthermore soft real-time adds scheduling classes so that complex systems can be structured better than with pure fair scheduling algorithms. At the same time soft real-time can operate in an unrestricted environment, utilizing most optimization strategies available, and not mandating specific programming practice. So as a quick summary - it improves quality without requiring specialized know-how. Naturally the quality improvement is not all to high if one switches to a soft real-time system without any real-time know how at all, but the transition is painless compared to hard real-time programming.

2.4 Why not soft-realtime

Obviously soft-realtime cannot be used in every situation. It is important to focus on the temporal specifics of problem and choose the right strategy. When not to use soft-realtime is best described by specifying the non-realtime and the hard-realtime domain.

There are two distinct directions:

- non-realtime: Using non-realtime increases the performance (system throughput) of your system but also increases latencies. It is the right choice if you do not care about loosing events and worst case jitter. With non-realtime you get maximum average throughput, but you get NO guarantees that an event will actually ever be processed.
  Application integration is trivial, if it compiles and runs its fine - no need to consider system-scope (at least not in general).

- hard-realtime: Loosing events can cause a failure in specific applications. Event determinism can only be guaranteed by using a hard-realtime system. Hard real-time systems are less efficient with respect to resource usage and to system throughput. They are more complex and the available programming resources are limited.
  Software integration is non-trivial, each real-time task added mandates the re-evaluation of the system at a system scope.

3 Technologies

This chapter covers the most important technologies of existing free-software implementation concepts and their problems concerning soft-realtime. In general all problems are due to high latencies which are not acceptable. The approaches differ in the way they try to tackle the high latency problem. In many cases combinations of these strategies are in use.

3.1 Timer resolution

Each time a timer interrupt occurs a counter is increased. The frequency of the timer depends on the system architecture. Systems like x86 use a 32bit counter. Traditionally the timer frequency is set to 100Hz. So the timer interrupt occurs every 10milliseconds. Traditionally the time is reported by this

hardware independent time base. An accuracy of 10ms is not acceptable for many soft-realtime systems. The solution are high resolution timers, which can access the hardware directly.

3.2 Scheduling frequency

The scheduler is called periodically by the timer interrupt. Standard desktop machines don't need a higher scheduling frequency than 100Hz because it would not be recognizable by humans. Furthermore the scheduler is invoked by other tasks as well - so the effective average scheduling frequency is somewhat higher than the 100Hz. The more interactivity the more often the scheduler is called Especially in embedded (non-interactive) systems 100Hz scheduling frequency is too low as there are no non-periodic (interactive) tasks which invoke the scheduler Furthermore a worst case jitter of 10ms is not acceptable for soft-realtime Increasing the scheduling frequency will improve this, but can lead to different problems. First the performance is decreasing because the scheduler is called more often, thus spending more time in the scheduler proper. As an example, a 486 75MHz will spend 100

3.3 Scheduling strategies

Many different scheduling strategies have been developed over time to optimize the utilization of the CPU, both for real-time and for non-real-time systems. Traditionally for GPOS some form of dynamic priorities are used as they offer the possibility that every task can run even if it's priority is very low. These fair scheduling strategies cannot be used in soft-realtime systems because realtime tasks must not be interrupted by other non-realtime tasks.

Normal non-rt systems provide the scheduling policy SCHED_OTHER which is generally some form of fair scheduling / dynamic priority, though the actual behavior is not specified by standards and can vary considerably between implementations. Soft real-time systems add SCHED_RR and SCHED_FIFO as there real-time scheduling policies. In general this will mean that SCHED_FIFO has highest priority, SCHED_RR follows in priority and SCHED_OTHER processes have the lowest priorities in the system. SCHED_RR is a scheduling policy which is specific to soft-realtime, it allows to group tasks together and make the aggregation of these tasks performing some computation to exhibit a statistically well determined behavior - the individual task has no strict timings though. Although the system naturally stays a preemptive multitasking system independent of the scheduling strategy, one can better think of SCHED_FIFO tasks being cooperative-multitasking, in the sense that they are actually able to monopolize the CPU - thus Linux mandates root-privileges for any task that wishes to utilize the soft-realtime scheduling policies.

It should be noted that in Linux a task with SCHED_FIFO or SCHED_RR has access to all user-space resources, which does mandate a certain know-how in using these scheduling policies (i.e. a while(1); in your code would not 'crash' the system but slow it down to an unusable state if run as SCHED_FIFO task).

3.4 Kernel code

Traditional kernels are non-preemptive in kernel mode, basically any kernel path is thus to be seen as an atomic operation from the perspective of user-code. This is a source of latencies as every user-space process must switch to kernel mode when accessing hardware resources. A few of the typical problems responsible for increased latency are:

- The longest kernel code execution path determines how long the one system call can take.
- Several synchronization strategies in the kernel can lead to different problems, i.e. locking big kernel structures which are accessed by multiple tasks causes high latencies. In combination with kernel preemption it is more efficient to step through only locking small parts of the structure.
- Interrupt disabling is often used in the kernel to avoid the contention in critical sections. It is necessary because critical data regions must not be modified by interrupts while the kernel changes them, or might not be reentrant, thus not allowing interleaved interrupt processing. But there are several disadvantages: During this time interrupts can get lost, which is due to loosing events and data. In addition hardware latency is increased because the interrupt is not handled until the interrupts are enabled again.
- dynamic scheduling cause worst case delays that are proportional to the absolute number of tasks in the system.

3.5 Interrupts

Loosing interrupts is critical in many systems, long periods of disabling interrupts will cause interrupt loses as most hardware only has a single bit in hardware to indicate that an interrupt occurred, queuing is generally not implemented. A soft real-time

system must have the ability to re-enable interrupts as quickly as possible. This is achieved by delegating most of the work to handlers (bottom halves or tasklets and soft-interrupts) that can be executed after interrupts have been re-enabled and that can be queued loss-free. Preferably a soft real-time system would have the ability to assign priorities to different interrupt related service routines. Linux offers rough priorities for tasklets and for soft-interrupts. For hardware interrupts patches are available to allow priority assignment, but these patches are not merged in the mainstream kernel (and probably won't be merged).

3.6 Resource Management

This topic concerns all kinds of resources, not only available memory. CPU time is an important resource too. In non-realtime resources are managed by the kernel. For example there are two tasks p1 and p2. Both want to access the same resource. p1 gets the requested resource, because it is faster than p2. While p1 is working p2 is put asleep. After p1 has freed the resource p2 is woken up and can now do it's work. The problem is that p2 didn't recognize that it was put asleep. Every process used the same method to access the resource: request the resource, work, and free the resource. In soft-realtime you must not be waiting for a resource. A strategy called preallocated resources is often used there. All required resources are allocated when you initialize the task. So there cannot be any problems later. In cases where resource can be accessed in a reentrant way, soft-realtime systems should do so, in this respect soft-realtime tasks are not independent of system scope issues. In general it is not possible to simply share resources between non-realtime and soft-realtime tasks, if this sharing is necessary, shared resources must be included in system testing.

3.7 Optimization

General purpose operating systems have many optimization strategies which increase the performance and average throughput of non-realtime systems but can also increase the worst case jitter of realtime systems.

Here are some examples:

- Resource management, especially memory management, of non-realtime systems dynamically allocates resources when they are needed.
- When a new process is created by the fork system call some non-realtime systems use copy-on-write to allocate memory dynamically. Both processes can use the same memory until it is written to, causing large delays in that case as this writing to a shared page would trigger the actual copy.
- Caching, e.g. streaming: Often the data in the cache can be used but not in every situation. So the worst case jitter is increased in cases where unnecessary caching is performed. The O_STREAM (a patch in Linux) can be used to disable caching in soft-realtime systems.

4 Strategies

4.1 POSIX high resolution timers

POSIX high resolution timers are used to measure time more accurate because traditional timers don't meet the requirements of soft-realtime systems. The hardware timer is accessed directly to allow an accuracy in the range of nanoseconds seconds.

4.2 Advanced scheduling policies

Advanced scheduling policies like the O(1) scheduler in 2.6 kernels provide a more deterministic latency of the scheduler (though not necessarily lower latency !).

4.3 Advanced kernel strategies

Up to now kernel space has not been preemptive, only user space. There are two different solutions to reduce the latency of the longest kernel execution path.

- low latency patch: This strategy takes the longest kernel path and adds preemption points. So the code must be reentrant, and the scheduler must be called by one's own. This method is difficult and there are no clear definitions where to place preemption points.
- preemptive kernel: This means that spin locks are used as preemption points. The big advantage of this method is that you must only program the code SMP safe, you need not care about preemption points because they come with spin locks.

Synchronization latencies in the kernel can be reduced with the lock breaking patch. Fine grain locking means that big data structures are split into several parts and so small parts can be locked, and not only the whole structure.

4.4 Interrupts

It is important to split realtime and non-realtime interrupts. One way is to use interrupt priorities. Important interrupts get higher priorities than others. Soft-realtime systems often use DSRs (deferred service routines) to apply priorities on interrupts. The most important part of the code stays in the ISR (interrupt service routine). The execution time of this code must be very short. ISR no longer need interrupt priorities, because it can be guaranteed that each interrupt is caught. All other code is implemented in a soft interrupt, which is also called DSR. Soft interrupt priorities are more useful because no other interrupt is lost while executing DSR code. Furthermore more important tasks can be executed before others.

4.5 Linux-based solutions

In recent years different Linux-based solutions have been developed. Note that soft-realtime capabilities should not be limited in perspective to the cpu. This list covers some of the general purpose operating systems that have soft-realtime capabilities.

- Montavista Linux: [6] preemptive kernel, high resolution timers
- Timesys Linux (Linux/RT): [7] resource mutexes
- Kurt: [3] low latency (preemption points), high resolution timers
- LXRT user-space realtime [5]
- kpreempt patches for 2.4 Linux kernels [**?**]
- 2.6 Linux kernels: [1] kernel preemption, high resolution timers, O(1) scheduler

4.6 Mainstream Linux soft-realtime capabilities

In recent years Linux has become very popular especially for industrial purpose. Allthough mainstream Linux is not targeting realtime capabilities it in the mean time offers several features related to soft-realtime. This is basically due to a large part of the soft-realtime problems being very similar to the problems posed by scalability in multiprocessor systems. If latencies are too high SMP systems with many CPUs become inefficient. Thus the SMP scalability problem has indirectly solved some of the soft-realtime problems aswell. The main soft-realtime related features in mainstream Linux are:

- high resolution timers
- lock breaking
- advanced synchronization primitives like sequence locks, read-write locks and buffer swapping.
- kernel preemption: preemption points and scheduling check in spin locks
- interrupt priorities: soft interrupts, DSRs, tasklets, as well as patches to introduce ISR priorities.
- compile-time configurable scheduling frequency
- boot-time selectable scheduling strategies.

5 Conclusion

Developing realtime software can lead to serious problems if one does not or can not strictly split non-realtime and realtime components, stating with this split is the first step.

As there are many existing solutions floating around the web, these need to be evaluated for the particular problem before deciding which one to take.

It is important to consider that non-realtime optimization strategies can increase worst case jitter extremely, thus introducing real-time in a system also limits what the non-realtime tasks may do, this needs to be taken into consideration when designing the non-realtime task-set.

As a general rule we would recommend to stick to the variant that has the least modifications to the main stream system selected, as maintenance of, especially exotic, patches is not always that good.

Realtime extensions should only be considered if it is clearly shown that non-realtime will not due, too many system out there running realtime extensions that simply do not need them.

References

[1] [Linux Kernel] *Linux Kernel Home-Page*, `http://www.kernel.org`

[2] [Preemptive Kernel] *kpreempt Home-Page*, `http://kpreempt.sourceforge.net`

[3] [Libertos] *LibeRTOS Home-Page*, `http://www.linutronix.de`

[4] [RTLinux on the web] *RTLinux Home-Page*, `http://www.rtlinux-gpl.org`

[5] [Real Time Application Interface] *RTAI Home-Page*, `http://www.rtai.org`

[6] [Montavista Software] *Montavista Home-Page*, `http://www.mvista.com`

[7] [TimeSys] *TimeSys Home-Page*, `http://www.timesys.com`