Real-Time Kernel For Audio and Visual Applications

John Kacur
Red Hat
19243, Wittenburg
Germany
jkacur@gmail.com

Abstract

Abstract: Many of the Linux Distributions that are dedicated to audio and video make use of the Linux real-time kernel. This paper explores some of the advantages and disadvantages of using real-time. It explains how the real-time kernel achieves low-latency and shows how user-space can take advantage of real-time capabilities. This talk is presented by one of the real-time kernel programmers, and gives an overview of the real-time kernel for audio and video.

Keywords

real-time, linux kernel

1 Introduction

Linux distributions dedicated to audio and video were some of the first distributions to ship the Linux real-time kernel to a general audience. Software such as Jack and Ardour among many other programs are designed to take advantage of real-time capabilities. The standard Linux Kernel has the facillities to do priority based scheduling, but for optimum low latency, the PREEMPT_RT kernel is required. Indeed many features of the PREEMPT_RT kernel have already been integrated into the standard kernel as an option, most notibaly soft and hard threaded irqs. Debugging features such as ftrace and lockdep also were originally designed for the real-time kernel. This paper will explore how the real-time Linux Kernel achieves such low latency, how to configure a system to take advantage of the capabilities it offers, and an introdution to the programming interface from user-space.

2 What is real-time?

When we talk about real-time systems, the most important feature is predictability or determinism. Often people who are new to real-time think it is about raw speed, but this isn't so. Real-time actually sacrifices some throughput performance in-order to achieve predictability and low latency where it is deemed important. A standard kernel will achieve better throughput on average, but a process may run very quickly some of the time but be delayed the rest of the time.. In contrast a real-time system will be less “bursty”, but have predictable performance. The measurement of the degree that time deviates from the average is called jitter, and it is this jitter that is greatly reduced with real-time.

The PREEMPT-RT kernel achieves predictability by ensuring that no operation takes more than 100 microseconds. In some cases the latency is as low as 50 microseconds which is close to the capabilities of the hardware. The low latency is important for high priority processes to accomplish what is required of them on time. By this, we mean the process must do what is required neither too slowly, nor too fast. If you are playing a note in a song, it is just as wrong for it to play too late as it is for it to play too early.

Although some throughput is sacrificed for determinism and low-latency, kernel programmers are working to lower the throughput gap as well.

2.1 Hard real-time vs soft real-time

Surprisingly no single definition exists of hard-real time. Hard real-time can be thought of as a system that never misses it's deadlines, and a soft-real-time system is one that sometimes can be allowed to miss it's deadlines.

The reality is more like a spectrum between these two poles. Since no hardware (or software for that matter) is perfect, it is doubtful that the ideal of hard real-time actually exists. So, the interesting question is, what should the system do

if it misses a deadline? For example, if a deadline is missed, should an event be cancelled or delivered late? If we are talking about video, it is often okay to drop some frames without it being noticeable, so that would be a case where we could cancel meeting a deadline. In cases where we are controlling a machine, it might actually be useless to deliver an event that controls, say a motor after it is too late.

Sometimes a softer version of hard real-time is acceptable. An audio application may have a desired latency of 5 milliseconds, but can occasionally be allowed to miss this deadline, but only if it isn't more than 10 milliseconds. [1}

2.1.1 What is PREEMPT_RT?

PREEMPT_RT is a real-time solution for Linux in which almost everything in the kernel is preemptable. The standard Linux kernel has an option called Preemtible Kernel (Low-Latency Desktop) PREEMPT_DESKTOP, in which all kernel code that is not in a critical section, that means protected by locks, is involuntarily preemtible. The PREEMPT_RT option extends this to make even most critical sections involuntarily preemtible. It does this by replacing kernel spinlocks with rt-mutexes. In addition rt-mutexes are supported by priority inheritance – more on that latter

PREEMPT_RT includes threaded soft and hard irq interrupts – this is now an option in the standard kernel. By making interrupts threaded, they are also now preemtible, they can be scheduled just like any other process, and can be given a priority, which can undergo priority inheritance if necessary.

Another critical component of a real-time system developed for PREEMPT_RT is high resolution timers.

Finally, many tracing and verification features such as ftrace and lockdep were originally developed for PREEMPT_RT, but are now part of the standard kernel.

2.1.2 How does PREEMPT_RT work?

First some background: Voluntary preemption is when kernel code can choose at preselected points in the code to be preempted by higher priority processes. Involuntary preemption is when the scheduler can force a process to stop running in order to allow a higher priority process to run.

The Linux Kernel is designed to run on SMP (Symmetric Multiprocessing) systems. This means that code must safely run on multiprocessors. In order for this to work properly, kernel programmers must identify critical sections. Critical sections are code with data that could be corrupted if it were suddenly interrupted, and later rerun on the same or a different processor. In order to protect this data, various types of locks are used. The most prevelant kind of lock is a spinlock. It works by simply waiting (spinning) while another process holds the lock to a critical section. When hopefully a short time later the lock is released, the waiting process can then grab the lock and continue to work. PREEMPT_DESKTOP can preempt most code, but not code that is running a critical section – that is locked code.

PREEMPT_RT works by converting most spinlocks into sleeping spinlocks, which in turn can be preempted. If you need a lock that can't sleep on -rt, then you identify it by making it a raw_spinlock. Obviously the goal is to have the bare minimum of these raw_spinlock as necessary, since they reduce the areas of code that are preemtible.

2.1.3 What is Priority Inversion and Priority Inheritance?

Priority Inversion occurs when a lower priority task runs at the expense of a higher priority task. Here is one of the simplest ways in which this can occur:

Imagine you have a high priority task A with a resource needed by an even high priority task B. Task B must block until task A frees it's resource. Then imagine a task C with a priority between A and B. Because the priority of C is higher than that of A, it preempts A and steals the processor. Thus we have process C running at the expense of the higher priority task B The solution that PREEMPT_RT provides to deal with this siutation is called priority inheritance. Priority Inheritance works by having higher priority tasks temporarily lending their prirority to lower priority tasks that hold resources that they require. So in the situation described here, task B would lend it's priority to task A, and A's prirority would rise to the same level as A's. That way task C would not be allowed to preempt A. Task A would run until it freed the resource required by B. B would then be scheduled to run, and A would return to it's original priority.

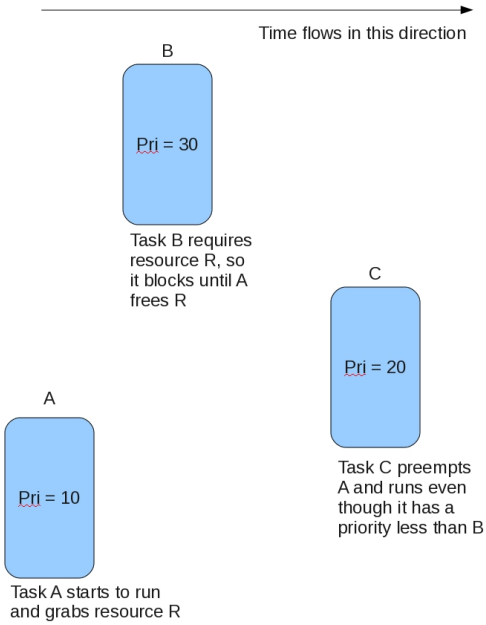

Fig.1. Without priority inversion, C preempts A and runs at the expensive of higher priority B

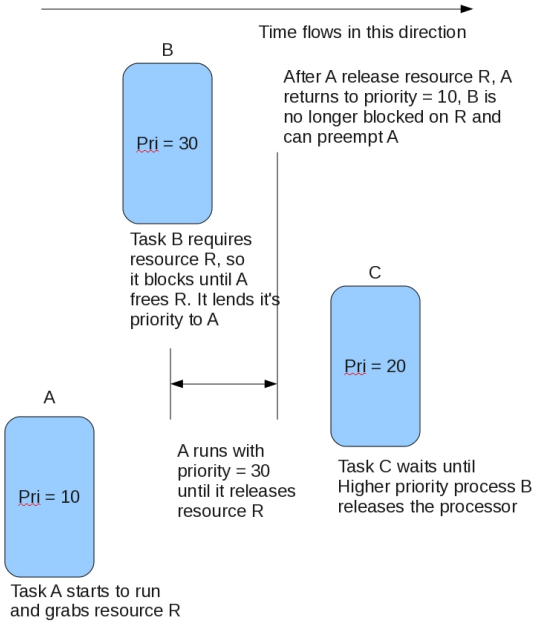

Fig.2. With priority inversion, B lends it's priority to A until A frees resource R, allowing B to run

3 Configuring real-time

3.1 Fetching, building and configuring the kernel

Some of the audio distributions are a little slower than the mainline distributions to update. So, if you need the latest hardware support, or simply want to try out the latest and greatest kernel, you may have to fetch, compile and install it by hand. Here's how to do so.

You can fetch the latest rt-patch (and archived ones as well) from:
http://www.kernel.org/pub/linux/kernel/projects/rt/

The latest one when I wrote this document (March 2010) was patch-2.6.33.1-rt10.bz2

From the name we can tell that it patches linux-2.6.33 with the stable patch linux-2.6.33.1 so we will need to fetch those as well.

```
wget
http://www.kernel.org/pub/linux/ke
rnel/v2.6/linux-2.6.33.tar.bz2
wget
http://www.kernel.org/pub/linux/ke
rnel/v2.6/patch-2.6.33.1.bz2
wget
http://www.kernel.org/pub/linux/ke
rnel/projects/rt/patch-2.6.33.1-
rt10.bz2
```

Now untar and apply the patches

```
tar xjf linux-2.6.33.tar.bz2
cd linux-2.6.33
bunzip2 -c ../patch-2.6.33.1.bz2 |
patch -p1
bunzip2    -c    ../patch-2.6.33.1-
rt10.bz2 | patch -p1
```

The rt kernel is also available via git for those who know how. For more information, see the rt-users mailing list, or Rtwiki. (below)

```
linux-rt-users@vger.kernel.org
https://rt.wiki.kernel.org/
```

You can base your configuration on your distribution's configuration. Look in your /boot directory for the config that matches your current running kernel, or copy it from /proc/config.gz if available as the real-time kernel's .config file. This

will save you from having to answer all the questions when you run make menuconfig and friends. A few choices are necessary for real-time. Under “Processor type and features”, it is important that you select Complete Preemption (PREEMPT_RT) which will automatically select Thread Softirqs (PREEMPT_SOFTIRQS) and Thread Hardirqs (PREEMPT_HARDIRQS) for you. Also recommended are TREE_PREEMPT_RCU, Tickless System (Dynamic Ticks) (NO_HZ), and High Resolution Timer Support (HIGH_RES_TIMERS). You can also select FTRACE without any noticeable effect on latency when not enabled. One more note of caution, in the “General Setup” section, for “Choose SLAB allocator”, you must choose SLAB as SLUB is not currently supported.

3.2 Configuring your system

There are many possible ways this can be done, but the key element is a way to specify either which users or programs get real-time privileges. One possible scheme is to create a group called “realtime”, and make sure any user that requires real-time privileges is assigned to it. Other possible groups that are commonly used are “audio”, or “jackuser”. The following creates the group “realtime”, and adds username jkacur to it.

```
sudo groupadd realtime
sudo usermod -G realtime jkacur
```

3.2.1 /etc/security/limits.conf

Here we need to modify users or groups to give them the privileges they require to run with real-time priorities, without being superuser.

Typical values would be.

```
@realtime  soft  cpu   unlimited
@realtime  -     rtprio     100
@realtime  -     nice  -20
@realtime  - memlock    unlimited
```

These allow users that belong to the realtime group to run with unlimited cpu time, the maximum real-time priority, the maximum nice value, and to lock unlimited amounts of memory. You may have to reboot your system before these changes take effect.

3.2.2 /proc/sys/kernel/sched_rt_runtime_us /proc/sys/kernel/sched_rt_period_us

The default values here are:

```
sched_rt_period_us 1000000
sched_rt_runtime_us 950000
```

The first value is the amount of time in microseconds that represents 100% of the CPU bandwidth. [3] The second value is the amount of time in microseconds that real-time processes are allowed to run. This means that 1000000 - 950000 = 50000 microseconds or 0.05 seconds are reserved for non-realtime tasks, in other words for SCHED_OTHER tasks to run. This is a soft-real-time behaviour, designed to protect you against runaway real-time processes that could hijack your system. You should definitely leave this at the default setting for testing. However, when you are ready to go to production, and want something closer to hard-realtime behaviour, you can set sched_rt_runtime_us to -1. This will now allow real-time tasks to monopolize the processor 100% of the requested time. Once again, use caution here, because this means that a misbehaving user-space program can make your system unuseable.

To change the setting, do:

```
sudo -c 'echo -1 >
/proc/sys/kernel/sched_rt_runtime_
us'
```

3.2.3 IRQs

Now that IRQs are threads, they can be given a priority. An -rt distribution will set these values with a start-up script. However, for optimum performance, and because of differences between your particular hardware, you may want to tune these values.

To see the current values on your system, use ps with the -o which is the option to control the output. We will select, pid, cls for the scheduling class, see table below

CLS Scheduling Class reported from ps	
-	not reported
TS	SCHED_OTHER
FF	SCHED_FIFO

RR	SCHED_RR
?	unknown value

Table 1.

rtprio for the real-time priority, prio for the priority, nice, and finally the cmd. [2]

For example, on an untuned vanilla distribution running a real-time kernel, looking at the tasklets which are similar to the bottom halves of traditional interrupts: (that is, the part of the interrupt that is scheduled to run later than the part that is serviced right away)

```
$ ps -eLo
pid,cls,rtprio,pri,nice,cmd | grep
-i tasklet
10  FF  49  89 - [sirq-tasklet/0]
25  FF  49  89 - [sirq-tasklet/1]
```

These values can be changed using the ***chrt*** command. Chrt changes or retrieves the real-time scheduling attributes of a process or task. It is usually installed by default on most distributions. It is part of the util-linux-ng package.

Tasklets should run higher than most real-time processes so 82 would be a reasonable value to set them to on this machine. [4]
To change the above.

```
$ su -c "chrt -f -p 82 10"
$ su -c "chrt -f -p 82 25"
$ ps -eo
pid,cls,rtprio,pri,nice,cmd | grep
-i tasklet
10  FF  82 122 - [sirq-tasklet/0]
25  FF  82 122 - [sirq-tasklet/1]
```

In general hardirqs (hardware interrupts) should be set slightly higher than the soft-interrupts (which you recognize by “sirq”). With tasklets at 82, 85 would be a reasonable priority for hardirqs.

Certain threads have the highest real-time priority, of 99

```
ps -eo pid,cls,rtprio,pri,nice,cmd
| grep -i ff | grep 99
3  FF 99 139   - [migration/0]
14  FF 99 139   - [posixcputmr/0]
15  FF 99 139   - [watchdog/0]
17  FF 99 139   - [migration/1]
18  FF 99 139   - [posixcputmr/1]
29  FF 99 139   - [watchdog/1]
```

These are critical to the system, and it is not recommented that userspace real-time process compete with them. Some papers suggest, and some distributions and scripts set the audio and video processes to run between the highest priorities and that of the tasklets and hardirqs. For this to work well, the audio and video processes would have to be very well written and run with real-time priorities for very short periods. I would suggest that audio and video run at the highest priority just under the tasklets. Given the scheme presentented here, 80 would be a good value.

```
@audio  -  rtprio     80
@audio  -  nice       -20
```

This should give the audio / video processes high enough priorities to achieve very low latency without interfereing with any systerm critical real-time processes, which in turn could degrade the overall perforance of a system.

4 Tests, Benchmarks, measuring

4.1 Rt-tests

Rt-tests is a suite of tests to stress various parts of the real-time kernel. The core of it is cyclictest written by Thomas Gleixner. It is now being maintained by Clark Williams, and can be fetched via git here:
git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git

The idea behind cyclictest is simply to fire off a number of high resolution timers from real-time threads and measure the difference between the time the timer is supposed to conclude and the time that it actually concludes. cyclictest can be used to measure your system's minimum, average and maximum latencies, and can thus be useful for tuning. Here is a typical run, just accepting the defaults.

```
./cyclictest
defaulting realtime priority to 2
policy: fifo: loadavg: 0.13 0.06
0.01 1/319 6879
T: 0 ( 6879) P: 2 I:1000 C:   7889
Min:     16 Act:  117 Avg:  107
Max:     298
```

The above shows two fifo threads, running at priority 2. The timer is firing at intervals of 1000 nanoseconds. (1 microsecond). So far, 7889 cycles occurred.

The results were a minimum latency of 16 microseconds, and average latency of 107 microseconds, and a maximum of 298.

4.2 hwlatdetect and SMIs

Hwlatdetect is a tool supplied with rt-tests to try to detect unexplained hardware latencies. One source of hardware latencies that we have very little influence over is SMIs – System Management Interrupts. These interrupts cannot be handled by software and can last tens of microseconds. It can be very dangerous for the stability of your hardware to attempt to turn them off too, because they often are in charge of such functions as making sure that your cpu doesn't overheat. In some cases with extreme caution, if your BIOS allows you to, you can turn some of them off. In some cases the SMI routines have been poorly written and the best we can do is bring to the attention of hardware makers that an SMI on a particular piece of hardware is taking an excessive amount of time.

Hwlatdetect works in cooperation with the hwlat_detect.ko kernel module. This kernel module comes as a standard feature in recent rt kernels. It tries to detect SMIs by monopolizing a cpu for a long time, with interrupts disabled. Since the only thing that could interrupt such a cpu is an SMI, any gaps detected in the time that hwlatdetect calls stop_machine are thus likely due to SMIs.

4.3 Rteval

Rteval is a program for evaluating the latency and performance of your system. The idea is simply to stress your system with some standard linux benchmarks such as dbench and hackbench, to see if they have an effect on the real-time capabilities of your system as measured by cyclictest.

You can fetch it via git here:

```
git://git.kernel.org/pub/scm/linux
/kernel/git/clrkwllms/rt-tests.git
```

5 Userspace Programming, a whirlwind tour and a caution.

What does a real-time program look like in userspace? Surprisingly simple. Of course it is a real art to identify which processes need to run with real-time priorities, and in most cases it is optimal to run with a higher priority for the minimum amount of time necessary.

Here is a whirlwind tour of some of the calls.

To raise the priority of a process, you can use standard POSIX calls such as:

```
sched_setscheduler(pid_t pid, int
policy, const struct sched_param
*param)
```

This call sets both the policy (SCHED_FIFO in the example below) and the priority. If the process id is set to 0, the parameters are applied to the calling process.

For example. (error checking code omitted)

```
struct sched_param param;
struct sched_param *pparm =
&param;
pparam->sched_priority = 50;
sched_setscheduler(0, SCHED_FIFO,
pparam);
```

You can retrieve the current scheduling policy with

```
sched_getscheduler(pid_t pid)
```

To retrieve or set the priority, you can use:

```
sched_getparam(pid_t pid, struct
sched_param *param)
sched_setparam(pid_t pid, const
struct sched_param *param)
```

To determine the minimum and maximum priorities allowed for a particular scheduling policy on your system, you can use:

```
int sched_get_priority_max(int
policy);
 int sched_get_priority_min(int
policy);
```

To prevent unexpected page faults of real-time programs, it is common to lock all current and future memory.

```
mlockall(MCL_CURRENT | MCL_FUTURE)
```

However, one caution, priority inheritance is only supported by rt_mutexes in the kernel. In user space this translates to pthread_mutexes. Ordinary unix semaphores in user-space are not supported by priority inheritance. The reason for this is, that it is not possible (or easy in any case) to identify the owner (pid) that blocked around a semaphore.

If you don't want to use pthread programming but want support for priority inheritance, you can create a lock using shared memory and a pthread_mutex that the usual POSIX style process calls can access. For an example of how this is done, see pip_stress in the rt-test suite.

6 Conclusion

Linux is well suited as a low-latency real-time operating system for audio and video. With a little bit of back-ground knowledge, an end user can tune his / her system for maximum performance and real-time determinism. This paper should get you well on your way to experimenting and tuning your real-time Linux system for optimal audio and video performance.

7 Acknowledgements

Ingo Molnar, Thomas Gleixner, Paul McKenney, Steven Rostedt, Peter Zilstra and countless others for creating PREEMPT_RT.

References

[1} Paul McKenney, 2007.

SMP and Embedded Real Time, January 2007 Linux Journal, issue #153 – There are more examples here illustrating the differences between hard and soft real-time.

[2] http://subversion.ffado.org/wiki/IrqPriorities

does an excellent job and explaining how to set irq priorities for real-time audio.

[3] See Documentation/scheduler/sched-rt-group.txt in linux kernel source 2.6.33.1-rt11 or later

[4] The suggested values are based on Red Hat's MRG distribution.

Other Sources of information

“Real Time” vs “Real Fast”: How to Choose?

Paul McKenney, Eleventh Real-Time Linux Workshoop, Dresden, 2009

Programming for the Real World, POSIX.4

Bill O.Gallmeister, O'Reilly & Associates, INC. Copyright 1995

Web Resources

https://rt.wiki.kernel.org

Approaches to Real-time, Jon Corbet
http://lwn.net/Articles/106010/

Realtime preemption, part 2, Jon Corbet

http://lwn.net/Articles/107269/

A real-time preemption overview, Paul McKenney

http://lwn.net/Articles/146861/