RT PREEMPT HOWTO

From RTwiki

Intro

Authors

Luotao Fu? (l.fu AT pengutronix DOT de), Pengutronix e.K., Kernel Development Group
Robert Schwebel (r.schwebel AT pengutronix DOT de), Pengutronix e.K., Kernel Development Group

Abstract

This document describes the procedure of installing and using the Realtime Preemption patch for the Linux kernel and discusses the first steps towards writing hard realtime programs. It focusses on x86, as this is currently the most mature architecture.

About the RT-Preempt Patch

The standard Linux kernel only meets soft real-time requirements: it provides basic POSIX operations for userspace time handling but has no guarantees for hard timing deadlines. With Ingo Molnar's Realtime Preemption patch (referenced to as RT-Preempt in this document) and Thomas Gleixner's generic clock event layer with high resolution support, the kernel gains hard realtime capabilities.

The RT-Preempt patch has raised quite some interest throughout the industry. Its clean design and consequent aim towards mainline integration makes it an interesting option for hard and firm realtime applications, reaching from professional audio to industrial control.

As the patch becomes more and more usable and significant parts are leaking into the Linux kernel, we see the urgent need for more documentation. This paper tries to fill this gap and provide a condensed overview about the RT-Preempt kernel and its usage.

The RT-Preempt patch converts Linux into a fully preemptible kernel. The magic is done with:

- Making in-kernel locking-primitives (using spinlocks) preemptible though reimplementation with rtmutexes:
- Critical sections protected by i.e. spinlock_t and rwlock_t are now preemptible. The creation of non-preemptible sections (in kernel) is still possible with raw_spinlock_t (same APIs like spinlock_t)
- Implementing priority inheritance for in-kernel spinlocks and semaphores. For more information on priority inversion and priority inheritance please consult Introduction to Priority Inversion (http://www.embedded.com/story/OEG20020321S0023)
- Converting interrupt handlers into preemptible kernel threads: The RT-Preempt patch treats soft interrupt handlers in kernel thread context, which is represented by a task_struct like a common userspace process. However it is also possible to register an IRQ in kernel context.
- Converting the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts, leading to userspace POSIX timers with high resolution.

Installation

Getting the Sources

Before you can start the fun you will first have to get a copy of the vanilla kernel source. Use your favorite kernel.org mirror server to download the kernel archive. If unsure, http://kernel.org is always a good start. To make sure that the patch runs smoothly later, you should better get a major kernel version without extra patches, i.e. 2.6.15, 2.6.16 etc. instead of 2.6.15.6 or 2.6.16.3.

You can obtain the realtime preemption patch from http://www.kernel.org/pub/linux/kernel/projects/rt/. Make sure that the RT-Preempt version fits to the kernel version you intend to use (patch-2.6.23.1-rt11 at the time of writing). If you are looking for patches for older kernel versions, try http://www.kernel.org/pub/linux/kernel/projects/rt/older/, which contains the archive of outdated patches.

Figure 1. Getting the Sources

```
# wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.1.tar.bz2
# wget http://www.kernel.org/pub/linux/kernel/projects/rt/patch-2.6.23.1-rt11.bz2
```

Patching the Kernel

After downloading, unpack the kernel tarball and change into the kernel source directory. Patch the kernel with patch level p1:

Figure 2. Patching the Sources

```
# tar xfj linux-2.6.23.1.tar.bz2
# cd linux-2.6.23.1
# bzcat ../patch-2.6.23.1-rt11.bz2 | patch -p1
```

The realtime preemption patch is currently under heavy development, so new versions appear quite frequently. If you intend to keep pace with the development, a patch management system like quilt (http://savannah.nongnu.org/projects/quilt) might be recommended.

Alternate Method to Download and Patch the Kernel

It may also be easier to just use Ketchup

Configuration and Compilation

If you are unfamiliar with the procedure of building a custom linux kernel from the source, you might want to consult some related howtos, for example the Kernel Rebuild Guide (http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html) , before you continue reading this document

Before you can compile your freshly patched kernel you will have to configure it first. If you already have a .config file fitting your hardware requirements, copy it to the kernel source directory and rename it to .config. The following screen shot shows the main configuration menu for a x86 kernel.

Figure 3. Configuring the Kernel (1)

http://www.pengutronix.de/images/rt-wiki/menuconfig1.PNG

Most of the realtime options can be found in the "Processor type and features" menu, as shown in the following screen shot.

Figure 4. Configuring the Kernel (2)

http://www.pengutronix.de/images/rt-wiki/menuconfig2.PNG

The most default configurations here are ok as-they-are. However you should make sure that you have

- enable CONFIG_PREEMPT_RT
- activated the High-Resolution-Timer Option (Attention, the amount of supported platforms by the HR timer is still very limited. Right now the option is only supported on x86 systems, PowerPC and ARM Support are however in queue.)
- disabled all Power Management Options like ACPI or APM (not all ACPI functions are "bad", but you will have to check very carefully to find out which function will affect your real time system. Thus it's better to simply disable them all if you don't need them. APM, however, is a no-go.) NOTE: Since rt patch 2.6.18-rt6 you will probably have to activate ACPI option to activate high resolution timer. Since the TSC timer on PC platforms, as used in the previous versions, are now marked as unsuitable for hrt mode due to many lacks of functionalities and reliabilties, you will need i.E. pm_timer as provided by ACPI to use as clock source. To activate the pm_timer, you can just activate the ACPI_SUPPORT in menuconfig and deactivate all other sub modules like "fan", "processor" or "button". If you have an old pc, which lacks ACPI support, you migh have problems using the high resolution timer.

Further interesting options can be found under the "Kernel Hacking" menu entry This menu lists options for system debugging and performance measurement. Keep in mind that the debug options may either increase the kernel size or cause higher latencies. If you do not want to debug the kernel or get some automatically produced histograms, make sure that you don't activate any unnecessary options here. If you have activated any latency critical options the kernel will warn at boot time.

Figure 5. Configuring the Kernel (3)

http://www.pengutronix.de/images/rt-wiki/menuconfig3.PNG

After configuration, the kernel can be compiled and installed as usual. The patch adds a "-rt**" suffix to the local version number of the kernel automatically, thus it might be recommended if you name your kernel image in the same way. Don't forget to create boot entries for your new kernel in the config file of your boot manager and re-run it (if you are using lilo). It is also recommended that you add "lapic" to the bootparameters of the RT-Preempt kernel.

Checking the Kernel

The first thing you have to check is that the right kernel has been booted. This can easily be done with

Figure 6. Kernel Version String

```
# uname -a
Linux krachkiste 2.6.18-rt5 #3 PREEMPT Thu Oct 06 14:28:47 CEST 2006 i686 GNU/Linux
```

The patch append its own revision number -rt* to the Kernel version string. So if you haven't changed the

kernel version string manually while doing the configuration, you will notice the -rt** appended to the kernel version after booting up the right kernel as shown above.

To assure that you have the correct kernel up and running, you can also check the dmesg output for following strings:
Real-Time Preemption Support (C) 2004-2006 Ingo Molnar

Now have a look at the process list. As mentioned earlier in this document. The IRQ handlers are treated by a patched kernel in kernel thread context. Single IRQ Handlers are transparently represented by task structs like users space tasks. Thus they can be listed or controlled by userspace tools. The following figure shows partly a list of running processes on a system with a patched kernel.

Note that, in contrast to a non-RT kernel, the interrupt handlers are kernel threads here, so they are listed [in square brackets].

Figure 7. Checking for Kernel Threads in the Process List

```
# ps ax
PID TTY      STAT   TIME COMMAND
1 ?        S      0:00 init [2]
2 ?        S      0:00 [softirq-high/0]
3 ?        S      0:00 [softirq-timer/0]
4 ?        S      0:00 [softirq-net-tx/]
5 ?        S      0:00 [softirq-net-rx/]
6 ?        S      0:00 [softirq-block/0]
7 ?        S      0:00 [softirq-tasklet]
8 ?        S      0:00 [softirq-hrtreal]
9 ?        S      0:00 [softirq-hrtmono]
10 ?        S<     0:00 [desched/0]
11 ?        S<     0:00 [events/0]
12 ?        S<     0:00 [khelper]
13 ?        S<     0:00 [kthread]
15 ?        S<     0:00 [kblockd/0]
58 ?        S      0:00 [pdflush]
59 ?        S      0:00 [pdflush]
61 ?        S<     0:00 [aio/0]
60 ?        S      0:00 [kswapd0]
647 ?        S<     0:00 [IRQ 7]
648 ?        S<     0:00 [kseriod]
651 ?        S<     0:00 [IRQ 12]
654 ?        S<     0:00 [IRQ 6]
675 ?        S<     0:09 [IRQ 14]
/  687 ?        S<     0:00 [kpsmoused]
689 ?        S      0:00 [kjournald]
691 ?        S<     0:00 [IRQ 1]
769 ?        S<s    0:00 udevd --daemon
871 ?        S<     0:00 [khubd]
882 ?        S<     0:00 [IRQ 10]
2433 ?        S<     0:00 [IRQ 11]
[...]
```

Now have a look at /proc/interrupts. The format of the interrupts proc entry under a patched kernel is slightly different than the one from a vanilla kernel, as shown in the following figure:

Figure 8. Checking /proc/interrupts (version earlier than 2.6.19-rt4)

```
# cat /proc/interrupts
CPU0
0:     497464  XT-PIC         [........N/  0]  pit
2:          0  XT-PIC         [........N/  0]  cascade
7:          0  XT-PIC         [........N/  0]  lpptest
10:          0  XT-PIC         [........./  0]  uhci_hcd:usb1
11:      12069  XT-PIC         [........./  0]  eth0
14:       4754  XT-PIC         [........./  0]  ide0
NMI:          0
LOC:       1701
ERR:          0
MIS:          0
```

The bit fields in the forth row provide informations on the IRQ Line:

- *I* (IRQ_INPROGRESS) - IRQ handler active
- *D* (IRQ_DISABLED) - IRQ disabled
- *P* (IRQ_PENDING) - IRQ pending
- *R* (IRQ_REPLAY) - IRQ has been replayed but not acked yet
- *A* (IRQ_AUTODETECT) - IRQ is being autodetected
- *W* (IRQ_WAITING) - IRQ not yet seen - for autodetection
- *L* (IRQ_LEVEL) - IRQ level-triggered
- *M* (IRQ_MASKED) - IRQ masked - shouldn't be seen again
- *N* (IRQ_NODELAY) - IRQ must run immediately

A "N" marks a IRQ, which is declared as hard irq and thus handled in kernel context. In our example we have IRQ0, IRQ2 and IRQ7 marked as hard IRQs, which are handled in kernel context. IRQ0 (timer) and IRQ2 (cascade controller) are set as hard IRQ by system by default. IRQ7 (lpptest) is used for benchmarking interrupt latency time. The lpptest comes with the realtime preemption patch. To mark an irq as hard irq, you will have to manipulate the irq description manually. With the irq status marked as IRQ_NODELAY and the flags of the irq action marked as IRQF_NODELAY | IRQF_DISABLED the handler of the irq will not be treated as a kernel thread. Since a such irq handler is unpreemptable, it's absolutely not recommended to mark an irq as hard irq manually while developing your own kernel modules.

NOTICE: Since patch version 2.6.19-rt4 the bitfield as shown above is removed to obtain as much compatibility with upstream kernel as possible. With a recent version of preempt-rt patch the /proc/interrupt interface may look like this. (Thx to Steven Scholz for the Screenshot, which is taken on 2.6.19-rt15)

Figure 9. Checking /proc/interrupts (version later or equal to 2.6.19-rt4)

```
CPU0
0:      15499   IO-APIC-edge      timer
1:          8   IO-APIC-edge      i8042
4:        161   IO-APIC-edge      serial
9:          0   IO-APIC-fasteoi   acpi
12:          3   IO-APIC-edge      i8042
14:       4082   IO-APIC-edge      ide0
16:         93   IO-APIC-fasteoi   eth0
```

You can change the priority of a kernel thread like i.E. a interrupt handler in the userspace during run time using chrt, which is downloadable at Schedutils Download Site (http://rlove.org/schedutils/) . With this tool you can change the internal scheduling policy and priority of processes.

Figure 10. Example of using chrt

```
# chrt -f -p $PRIO $PID_OF_THE_KTHREAD
# chrt -p $PID_OF_THE_KTHREAD
```

With first command in the example above you can change the priority of the thread with the pid $PID_OF_THE_KTHREAD to $PRIO with the policy "fifo". With the second command you can get a view of the results of your changes.

A Realtime "Hello World" Example

We described some internal mechanisms of the realtime preemption earlier in this document. A main goal of the patch is, however, realizing a real time environment without changing the given programming APIs by a common linux environment. Still there are some important points while programming real time applications. There are three things needs to be set by a task is order to provide deterministic real time behavior:

- Setting a real time scheduling policy and priority.
- Locking memory so that page faults caused by virtual memory will not undermine deterministic behavior
- Pre-faulting the stack, so that a future stack fault will not undermine deterministic behavior

The following example contains some very basic example code of a real time application with realtime preemption patch. Compile as follows:
gcc -o test_rt test_rt.c -lrt

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>

#define MY_PRIORITY (49) /* we use 49 as the PRREMPT_RT use 50
                            as the priority of kernel tasklets
                            and interrupt handler by default */

#define MAX_SAFE_STACK (8*1024) /* The maximum stack size which is
                                   guranteed safe to access without
                                   faulting */

#define NSEC_PER_SEC    (1000000000) /* The number of nsecs per sec. */

void stack_prefault(void) {

        unsigned char dummy[MAX_SAFE_STACK];

        memset(dummy, 0, MAX_SAFE_STACK);
        return;
}

int main(int argc, char* argv[])
{
        struct timespec t;
        struct sched_param param;
        int interval = 50000; /* 50us*/

        /* Declare ourself as a real time task */

        param.sched_priority = MY_PRIORITY;
        if(sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
                perror("sched_setscheduler failed");
                exit(-1);
        }

        /* Lock memory */

        if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
                perror("mlockall failed");
                exit(-2);
        }

        /* Pre-fault our stack */

        stack_prefault();

        clock_gettime(CLOCK_MONOTONIC ,&t);
        /* start after one second */
        t.tv_sec++;

        while(1) {
                /* wait until next shot */
                clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);

                /* do the stuff */

                /* calculate next shot */
                t.tv_nsec += interval;

                while (t.tv_nsec >= NSEC_PER_SEC) {
                       t.tv_nsec -= NSEC_PER_SEC;
                        t.tv_sec++;
                }
   }
}
```

An example which creates a square wave on parallelport: Squarewave-example

Benchmarking

A simple tool to determine the realtime performance of your brand new preempt-rt patched kernel is the cyclictest tool by Thomas Gleixner. This Tool acquires timerjitter by measuring accuracy of sleep

and wake operations of highly prioritized realtime threads. The tool is designed to test different timer APIs. Thus you have to start it with proper parameters. Below in the screen shot is an example of using cyclictest.

Figure 11. Benchmarking the Realtime Environment using Cyclictest

```
# ./cyclictest -p 80 -t5 -n
1.58 1.61 1.62 3/68 4079

T: 0 ( 3131) P:80 I:    1000 C:16469865 Min:       8 Act:      13 Max: 62
T: 1 ( 3132) P:79 I:    1500 C: 9979903 Min:       8 Act:      18 Max: 75
T: 2 ( 3133) P:78 I:    2000 C: 7934887 Min:       9 Act:      22 Max: 83
T: 3 ( 3134) P:77 I:    2500 C: 6587910 Min:       9 Act:      25 Max: 81
T: 4 ( 3135) P:76 I:    3000 C: 5489925 Min:       9 Act:      27 Max: 86
```

With the parameter used above cyclictest is started with 5 threads (-t5 -- DO NOT USE -t 5), which has the highest priority of 80 (-p) and sleeps regularly calling nano_sleep() (-n). Further you can determine the sleep period and the step range between the threads using -i and -d. In the output you can see (from left to right) the thread number, pid, priority, sleep interval, wakeup count, minimal jitter, recent jitter, maximal jitter. All results here are given in microseconds. You can obtain cyclictest from the Cyclictest Download Site (http://www.tglx.de/projects/misc/cyclictest/) . In the example given above the program runs in interactive mode. In case you want to log the measured results and plot it, you can use the trigger -v and pipe the stdout to a file. You can stop the application by hitting "ctrl-c" or use the trigger -l to define the total loops to be run. Under Cyclic_Parser Download Site (http://www.pengutronix.de/software/cyclic_parser/download/) you can obtain a script, which parses and splits the whole output file, so that single threads can be plotted individually.

To verify the realtime behavior of your system you might want to add some additional workload to your system while measuring. A recommended tool is the Cache Calibrator (http://homepages.cwi.nl /~manegold/Calibrator/) . This tool produces heavy cache pollutions and causes thus high latency time while switching between applications. Together with a fping, which generates high interrupt load you can get reliably extreme load situations.

Troubleshooting

Besides the userspace testing tools you can also use the built-in test- and debugging mechanism coming with the RT-PREEMPT Kernel. To use these tests, you have to run a kernel, which is compiled with the test options enabled. You can find these options under the section "Kernel hacking" while configuring the kernel. With the the test option enabled you can measure latency time of waking up a task with high priority, entering and lefting a non-preemptible section or critical sections with irqs turned off can be measured and access the results, either as single value or as plottable histogram in the proc file system. To learn more about these mechanism please consult the help pages of the single options while configuring the kernel. If you are experiencing problems with i.e. unusually long latency times and desire to examine the runpath. You can turn on the built-in function tracer (option "Latency tracing"). The tracer uses the profile generating function of the compiler gcc to register the most function calls in the running system. Thus it can log all calls made in system during a complete runpath. You can access the logged results in the proc file system. You can control the test and debugging mechanism like turning on/off during run time by accessing their proc file system entries. To find out how you can exactly do it please consult the help pages. A short example can, however be found int the source code of Cyclictest (http://www.tglx.de/projects/misc/cyclictest/)

Figure 12. Example of using proc entries of the debugging mechanism

```
# echo 1 > /proc/sys/kernel/trace_all_cpus
# echo 1 > /proc/sys/kernel/trace_enabled
# echo 1 > /proc/sys/kernel/trace_freerunning
# echo 0 > /proc/sys/kernel/trace_print_at_crash
# echo 1 > /proc/sys/kernel/trace_user_triggered
# echo -1 > /proc/sys/kernel/trace_user_trigger_irq
# echo 0 > /proc/sys/kernel/trace_verbose
# echo 0 > /proc/sys/kernel/preempt_thresh
# echo 0 > /proc/sys/kernel/wakeup_timing
# echo 0 > /proc/sys/kernel/preempt_max_latency
```

Online Resource

MichaelBarr. *Introduction to Priority Inversion (http://www.embedded.com/story/OEG20020321S0023)*
KwanLowe. *Kernel Rebuild Guide (http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html)* .
ThomasGleixner. *Cyclictest Download Site (http://www.tglx.de/projects/misc/cyclictest/)* .
StefanManegold. *Cache Calibrator Download Site (http://monetdb.cwi.nl/Calibrator/)*
RobertLove. ~~*Schedutils Dowload Site (http://rlove.org/schedutils/)*~~ *now part of the util-linux package (ftp://ftp.kernel.org/pub/linux/utils/util-linux/)*
LuotaoFu. *Cyclic_Parser Download Site (http://www.pengutronix.de/software/cyclic_parser/download)*
Link to 'Realtime Testing Best Practices' on Embedded Linux elinux.org wiki (http://elinux.org/Realtime_Testing_Best_Practices)

- This page was last modified 22:06, 12 July 2011 by caleb crome?. Based on work by Erik Thiele, Jérôme Carretero?, Michael Mueller?, Jörn Nettingsmeier?, hyksos?, Robert Schwebel, Suresh?, Uwe Kleine-König?, Trevor Harmon?, Gilad Ben-Yossef?, Leon Woestenberg?, Lawrence Toal?, Steven Rostedt?, Jaswinder Singh, Peter Feuerer?, Andrew Burgess?, Luotao Fu?, Chuck Harding and Michal Schmidt? and RTwiki user Dirk?.