Open Source Automation Development Lab (OSADL) eG

OSADL Add-on Patches for the Linux Real-time Kernel

www.osadl.org

OSADL

Open Source Automation Development Lab eG

Introduction

In addition to the so-called *PREEMPT_RT* basic patch set that is used to merely equip a vanilla Linux kernel with real-time capabilities, there are patches that additionally provide diagnostic and other functionality to the basic patch set. These patches are useful during software development; however, they are not needed when simply running the kernel. Some of the additional patches originally were part of the *PREEMPT_RT* patches, but were removed in order not to jeopardize the mainlining process of the patches. Some other patches have been developed idependently by OSADL.

OSADL makes the currently five patches available for every real-time patchable kernel and later on for every supported kernel starting from version 4, patchlevel 16. The patches altogether are named "OSADL Linux Add-on Patcches" and are offered for download from the OSADL Web site at the URL https://www.osadl.org/?id=2943. If the Linux add-on patches apply without offset for a number of kernels, only a single patch will be provided that fits all of them, but will be labeled accordingly.

Imported patches from the PREEMPT_RT patch and extended

Ping SysRq: *net-ipv4-icmp-ping-sysrq.patch*
Latency histograms with culprit/victim info: *latency-histograms.patch*

Developed by OSADL among other for its QA Farm

NMI SysRq: *add-nmi-callback-and-raw-parport-driver.patch*
Built-in kernel patchset: *save-current-patchset-in-kernel.patch*
Precise load measurement: *sched-add-per-cpu-load-measurement.patch*

To avoid errors, the patches must be applied in the following order:

add-nmi-callback-and-raw-parport-driver.patch
latency-histograms.patch
net-ipv4-icmp-ping-sysrq.patch
save-current-patchset-in-kernel.patch
sched-add-per-cpu-load-measurement.patch

Ping SysRq

What is *SysRq*?

The *SysRq* functionality is provided to allow sending diagnostic or recovery commands to the Linux kernel in a situation when normal I/O is no longer possible because of, for example, a system crash. *SysRq* usually is triggered from the keyboard by simultaneously pressing the *RightAlt* and the *SysRq* (*PrntScrn*) key. However, the system may be impaired in such a way that even the keyboard interrupt stopped working. If the network is still up and running and able to receive ICMP packets, this *SysRq* function via network *ping* may help. In order to work, the target system must have defined a particular pattern – preferably at boot time – that will be used as a key to authorize the *SysRq* action from remote.

Example prerequisite on target to authorize a particular pattern

```
echo 0x01020304 >/proc/sys/net/ipv4/icmp_echo_sysrq
```

Ping command on remote system (use the registered pattern)

```
ping -c1 -s57 -p01020304<ASCII hex command> target
```

Example: Gracefully reboot system

SysRq S (sync block devices) = ASCII 0x73:

```
ping -c1 -s57 -p0102030473 target
```

SysRq U (unmount block devices) = ASCII 0x75:

```
ping -c1 -s57 -p0102030475 target
```

SysRq B (reboot) = ASCII 0x62:

```
ping -c1 -s57 -p0102030462 target
```

Latency histograms

What are latency histograms?
Linux kernel latency histograms are used to determine the duration of certain pathways that contribute to the preemption latency. As a side effect, the data stored along with the histogram data may allow to discover the source of an unusual long latency, since these data include the name and characteristic data of the previous and the next task of a delayed scheduling action. The original latency histograms were made available in the RT patch set in 2005; a number of additions were made since then:

- Separate histograms of shared vs. non-shared priority tasks
- Separate recording of “victim” and “culprit” of the highest latency
- Additional histogram to record latency of missed timers
- Additional histogram to record the sum of timer and wakeup latency
- Additional histogram to record duration of context switch
- Additional histogram to record the sum of timer, wakeup and context switch latency

Thus, the current version of the patch covers histograms to investigate the entire time span from the wake up trigger of an RT task to its execution continuation in user space.

Configure latency histograms

```
CONFIG_PREEMPT_OFF_HIST=y
CONFIG_INTERRUPT_OFF_HIST=y
CONFIG_MISSED_TIMER_OFFSETS_HIST=y
CONFIG_WAKEUP_LATENCY_HIST=y
CONFIG_SWITCHTIME_HIST=y
```

Enable latency histograms

```
enabledir=/sys/kernel/debug/latency_hist/enable
for i in wakeup missed_timer_offsets
timerandwakeup\ switchtime timerwakeupswitch
do
  enable=$enabledir/$
  if test -f $enable
  then
    echo 1 >$enable
  fi
done
```

Display latency histograms of, for example, core #0

```
cd /sys/kernel/debug/latency_hist/timerwakeupswitch
cat CPU0
```

Display the longest latency since the most recent reset of, for example, core #0 along with culprit and victim data

```
cat /sys/kernel/debug/latency_hist/
timerwakeupswitch/max_latency-CPU0
```

```
13457 99 42 (2,9) cyclictest <- 6681 5 alsa-sink-
HDMI  1164604.885757 SyS_clock_nanosleep
```

Victim

```
Process ID Priority Latency values Program name <-
```

Culprit

```
Process ID Priority Program name
```

Additional information

```
Seconds.microseconds Name of the most recent kernel
call
```

NMI SysRq

Why NMI?
A system may become non-responsive in such a way that no input can be sent any longer to the system; such situation normally is the result of disabled interrupts and a complete system misbehavior that prevents it from re-enabling them. The last resort to gain diagnostic data in order to elucidate the origin of the system misbehavior is the non-maskable interrupt (NMI) that very probably still is executed from time to time and which may be polled. For such polling an input channel is needed that can be probed without the need for interrupt processing. An appropriate input channel for this purpose is the parallel interface aka Centronics printer port. It has eight output and four input channels.

While the input channels are used to select an action to be executed when the NMI fires, the output channels can be used to signal a particular action – again without the need of interrupt processing. Some – rather oldish – computers still may be equipped with a Centronics printer port, while on newer computers a separate parallel printer port with PCI or PCIe bus interface must be installed. A simple device with a 25-pin parallel port connector, eight LEDs to display the output channels and four touch buttons to set the input channels dubbed OSADL Parport Monitor can be ordered at OSADL. Alternatively, it can be home made, since all required production material is available online. Last not least, the OSADL Parport Monitor can be purchased from OSADL.

Kernel configuration help text

It sometimes is required to directly signal a specific state at the parallel port without using a driver, e.g. in a crashed system that still has some kind of life in it. Usage:

```
echo 0 .. 255 >/dev/setparport set output byte
echo 256 .. 511 >/dev/setparport "or" output byte
echo 512 >/dev/setparport clear all output bits
echo 513 >/dev/setparport set all output bits
echo 514 >/dev/setparport invert output bits
echo 515 >/dev/setparport increment output bits
echo 516 >/dev/setparport decrement output bits
echo 517 >/dev/setparport status register to output
echo 518 >/dev/setparport jiffies>>10 to output
```

In addition, this driver is used as a callback in the NMI handler. If installed, it allows to monitor NMI activity, e.g. using LEDs connected to the parallel port. The module parameter "nmicode" is then used to define the code to be sent at every NMI call, e.g. to increment the 8-bit number at the parallel port at every NMI

```
modprobe setparport nmicode=515
```

or

```
echo 515 >/sys/module/setparport/parameters/nmicode
```

The four input lines can also be used to request specific actions; defaults are enabled, if the parameter actions=yes is given:

- S4: Show task states (SysRq-T)
- S5: Sync block devices (SysRq-S)
- S6: Unmount bloick devices (SysRq-U)
- S7: Reboot (SysRq-B)

Please note that this is a simple polling mechanism; you need to press the button at least as long as until the next NMI occurs. This was deliberately implemented this way in order to keep it functional even if the entire IRQ subsystem is no longer working. The only prerequisite is a working memory mapping of the parallel port's IO region. If you want to let the NMI execute other debug actions, they must be programmed into drivers/misc/setparport.c.

Last not least, this driver can be used to output the LSB of the most recent syscall, hardware IRQ or software IRQ vector at the parallel port which may provide useful post-mortem information in case of a system crash.

System call:

```
modprobe setparport sysenter=1 sysexit=0
```

or

```
echo 1 >/sys/module/setparport/parameters/sysenter
echo 0 >/sys/module/setparport/parameters/sysexit
```

Hardware IRQ number:

```
modprobe setparport irqenter=1, irqexit=384
```

or

```
echo 1 >/sys/module/setparport/parameters/irqenter
echo 384 >/sys/module/setparport/parameters/irqexit
```

Software IRQ vector number:

```
modprobe setparport sirqenter=1 sirqexit=384
```

or

```
echo 1 >/sys/module/setparport/parameters/sirqenter
echo 384 >/sys/module/setparport/parameters/
sirqexit
```

If configured as a built-in kernel module, the following kernel command line parameters apply:

```
setparport=<actions>,<nmicode>
setparportirq=<irqenter>,<irqexit>
setparportsirq=<sirqenter>,<sirqexit>
setparportsyscall=<sysenter>,<sysexit>
```

Built-in kernel patchset

Why storing the patchset in the kernel?
The kernel configuration can be stored in the binary kernel using the CONFIG_IKCONFIG=y kernel configuration setting; this is a welcome feature to reproduce a particular kernel at a later date when the original configuration may got lost. However, the kernel configuration is useless, if the kernel was patched and the patch set got lost as well. Therefore, an additional mechanism to also store the quilt queue in the binary kernel was added, configuration setting CONFIG_IKPATCHSET=y. If the patchset shall be made available via /proc filesystem, the configuration setting CONFIG_IKPATCHSET_PROC=y must additionally be given (equivalent to CONFIG_IKCONFIG_PROC=y).

BTW: Making the kernel sources available through this mechanism certainly does not fulfill the disclosure obligations of the GPL (but it already is much better than making available nothing).

How to recreate the kernel source tree?

The following command sequence will create a patched kernel source tree from which an identical kernel can be rebuilt (assumes that IKPATCHSET_PROC is also configured):

```
tar zxf /proc/patchset.tar.gz baseversion
major=`cut -d. -f1 baseversion`
urldir=http://www.kernel.org/pub/linux/kernel/
v$major.x
dir=linux-`cat baseversion`
rm -f baseversion
archive=$dir.tar.xz
wget $urldir/$archive
tar Jxf $archive
cd $dir
tar zxf /proc/patchset.tar.gz
quilt push -a
zcat /proc/config.gz >.config
```

Precise load measurement

Why another load measurement?
The standard load measurement of the Linux kernel has the disadvantage that it is collecting load data at discrete points in time, but the load may be completely different some time before and some time after the data were collected. A more precise per-CPU load measurement can be obtained, if the time is recorded a particular CPU core spends in idle processing and this time is compared against the total time.

How does it work?

The related patch (configured with CONFIG_CPU_IDLERUNTIME= y) adds entries for every CPU in */proc/idleruntime/cpuX/data* in the format "<idletime> <runtime>". The counters can be reset by writing to */proc/idleruntime/cpuN/reset*. To calculate the per-core CPU usage since the most recent reset, divide the runtime by the sum of runtime plus idletime, e.g. on a 4-core processor:

```
for i in `ls -1d /proc/idleruntime/cpu* | sort
-nk1.22`
  do
  echo "$i: `awk '{ print (100.0*$2) / ($1+
$2)"%" }' <$i/data`"
  echo 1 >$i/reset
done
/proc/idleruntime/cpu0: 72.0048%
/proc/idleruntime/cpu1: 5.49522%
/proc/idleruntime/cpu2: 0.27916%
/proc/idleruntime/cpu3: 32.3493%
```

In addition, summed up data of all present CPUs are available in /proc/idleruntime/all in the same format as above. Thus, to calculate the overall CPU usage since the most recent reset, the following command may be used:

```
awk '{ print (100.0*$2) / ($1+$2)"%" }' </proc/
idleruntime/all/data
```