Finding Realtime Linux Kernel Latencies

(for RT linux kernels >= 2.6.33)

---------------------------------------

Unexplained delays while processing events are one of the biggest

problems in the Realtime Linux kernel (or any realtime system for that

matter). This document describes one mechanism for locating delays in

the kernel code. It presumes some moderate knowledge of how to

configure and build Linux kernels and kernel terminology.

1. What is "latency"?

------------------------------------------------------------------------------

The term latency, when used in the context of the RT Kernel, is the

time interval between the occurance of an event and the time when that

event is "handled" (typically "handled" means running some thread as a

result of the event). Latencys that are of interest to kernel

programmers (and application programmers) are:

- the time between when an interrupt occurs and the thread

waiting for that interrupt is run

- the time between a timer expiration and the thread waiting for

that timer to run

- The time between the receipt of a network packet and when the

thread waiting for that packet runs

Yes, the timer and network example above are usually examples of the

more general interrupt case (most timers signal expiration with an

interrupt and most network interface cards signal packet arrival with

an interrupt as well), but the main idea is that an "event" occurs and

there is some elapsed time interval which concludes with the kernel

successfully handling the event.

So, latency in and of itself is not a bad thing; there is always a

delay between occurance and completion of an event. What is bad is

when latency becomes excessive, meaning that the delay exceeds some

arbitrary threshold. What is this threshold? That's for each

application to define. A threshold or "deadline" is what defines a

real time application: meeting deadlines means success, missing

deadlines (exceeding the threshold) means failing to be real time.

2. Tools for measuring latency

------------------------------------------------------------------------------

How do you measure event latency in the Linux kernel? Since there are

many types of events that occur in the kernel, there's no one way to

measure event latency. Usually a program is written to measure a

specific type of latency. One such tool is called "cyclictest".

Cyclictest was written by Thomas Gleixner and is now maintained by

Clark Williams (with assistance from John Kacur, Carsten Emde, Uwe

Klein-Koeig and the rest of the gang on the linux-rt-users mailing

list) as part of the rt-tests package. This group of test programs may

be found at the git repository:

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

or as a tarfile at:

http://www.kernel.org/pub/linux//kernel/people/clrkwllms/rt-tests/

Cyclictest measures the amount of time that passes between when a

timer expires and when the thread which set the timer actually

runs. It does this by taking a time snapshot just prior to waiting for

a specific time interval (t1), then taking another time snapshot after

the timer finishes (t2), then comparing the theoretical wakeup time

with the actual wakeup time (t2 -(t1 + sleep\_time)). This value is the

latency for that timer wakeup.

Here is some sample output from a typical cyclictest run:

$ cyclictest --smp -p95 -m

policy: fifo: loadavg: 0.04 0.01 0.00 1/338 31976

T: 0 (31974) P:95 I:1000 C: 4990 Min: 9 Act: 37 Avg: 31 Max: 59

T: 1 (31975) P:95 I:1500 C: 3322 Min: 10 Act: 68 Avg: 57 Max: 90

^C

This shows cyclictest being run on a system with two cpu cores, with

one measurement thread per core, each running as SCHED\_FIFO priority

95, and with all memory locked. In the above ouput, cpu0 has had a max

latency of 59 microseconds with an average latency of 31 microseconds,

while cpu1 has a max of 90 microseconds with an average of 57

microseconds.

Cyclictest has many options for adjusting how measurements are made,

such as how many measurement threads are run, scheduling policies for

measurement threads, etc., which we're going to use in the next

section to try and pinpoint excessive latency sources within the

kernel.

3. Tracing

------------------------------------------------------------------------------

The RT kernel contains a subsystem that is useful for locating timing

problems, called the "ftrace" system. You can find more information on

ftrace at the rt wiki site:

http://rt.wiki.kernel.org/index.php/Ftrace

There is also an interesting article on the Linux Weekly News (LWN)

site:

http://lwn.net/Articles/322666/

For now, rather than try to explain the ins and outs of ftrace, we'll

just use it.

To make use of frace, it must be configured into your kernel. The

following configuration options will get you the features we'll use in

the rest of this howto:

CONFIG\_FTRACE=y

CONFIG\_FUNCTION\_TRACER=y

CONFIG\_FUNCTION\_GRAPH\_TRACER=y

CONFIG\_SCHED\_TRACER=y

CONFIG\_FTRACE\_SYSCALLS=y

CONFIG\_STACK\_TRACER=y

CONFIG\_DYNAMIC\_FTRACE=y

CONFIG\_FUNCTION\_PROFILER=y

CONFIG\_DEBUG\_FS=y

Once you have an RT kernel with ftace enable and that kernel is

booted, you must mount the DebugFS filesystem. This is done by

convention at /sys/kernel/debug, like so:

# mount -t debugfs debugfs /sys/kernel/debug

Nothing prevents you from mounting it somwhere with a shorter path,

such as /debug. Be aware that not all tools are smart enough to

discover where the DebugFS has been mounted if it's not at it's

traditional place (cyclictest is though).

Once the DebugFS has been mounted, you should see a 'tracing'

subdirectory in /sys/kernel/debug. This directory is where all the

interface files to frace reside.

# ls /sys/kernel/debug/tracing

available\_events ksym\_profile sysprof\_sample\_period

available\_filter\_functions ksym\_trace\_filter trace

available\_tracers options trace\_clock

buffer\_size\_kb per\_cpu trace\_marker

current\_tracer printk\_formats trace\_options

dyn\_ftrace\_total\_info README trace\_pipe

events saved\_cmdlines trace\_stat

failures set\_event tracing\_cpumask

function\_profile\_enabled set\_ftrace\_filter tracing\_enabled

kprobe\_events set\_ftrace\_notrace tracing\_on

kprobe\_profile set\_ftrace\_pid

Now you're ready to run cyclictest with some tracing options. The

easiest way to do that is like this:

# cyclictest --smp -p95 -f -b 200

That tells cyclictest to run with one measurement thread per core,

each at realtime priority (SCHED\_FIFO) 95 and to run the function

tracer, with a 'break threshold' of 200 microseconds. What this

means is that cyclictest will setup for tracing by writing values

into the tracing control files:

tracing/current\_tracer <- "function"

tracing/tracing\_thres <- "200"

tracing/tracing\_enabled <- "1"

It will then start its normal measurement run, displaying the latency

values as shown above, but when a latency value exceeds the break

threshold, cyclictest will stop the trace (by writing a "0" into

tracing/tracing\_enabled) and then stop measuring, reporting the id of

the thread that first exceeded the threshold.

# cyclictest --smp -p95 -f -b 200

could not set latency\_hist/wakeup\_latency/reset to 1

policy: fifo: loadavg: 0.16 0.23 0.18 3/336 4961

T: 0 ( 4960) P:95 I:1000 C: 0 Min:1000000 Act: 0 Avg: 0 Max:-1000000

T: 1 ( 4961) P:95 I:1500 C: 0 Min:1000000 Act: 0 Avg: 0 Max:-1000000

# Thread Ids: 04960 04961

# Break thread: 4960

The above run shows that the specified threshold (200us) was hit by

thread 4960 even before the display code could update the screen.

The next step is to retrieve the trace data, which is done by

reading the contents of tracing/trace:

# cat /sys/kernel/debug/tracing/trace >tracedata.txt

This output could be many megabytes in size, so make sure you send

the output to a filesystem with plenty of space.