signaltest: Using the RT priorities

Arnaldo Carvalho de Melo
Red Hat Inc.
acme@redhat.com

Abstract

This howto will describe a tool used to measure the latency of a simple message passing mechanism, pthread signals. Step by step concepts of real time scheduling will be introduced. Tools for changing and showing the priorities of running threads will also be presented.

1 Introduction

The `signaltest` utility was written by Thomas Gleixner to measure the latencies involved in a simple message passing mechanism, pthread signals. We will study its main characteristics, introducing aspects of real-time scheduling priorities. Tools for changing the priorities of running threads and to show its priorities will also be presented.

2 Scheduling Policies

In the Linux kernel, there are several scheduling policies. The default time-sharing policy is called `SCHED_OTHER` and is used by most processes. Real-time processes usually use the `SCHED_FIFO` policy, which runs processes until they voluntarily schedule or a higher priority process preempts them. This HOWTO will focus on these two policies, however the techniques described are applicable to other available scheduling policies, such as `SCHED_RR` and `SCHED_BATCH`.

Threads start with the scheduling policy and priority of its parent. Within a program, a thread may change it's priority and scheduling policy with the `sched_setscheduler` syscall.

Programs can have their priority and scheduling policy changed at run time with a tool called `chrt`. `chrt` can change the priority and policy of a running program, as well as to start a program off with a set priority and policy.

The following example effectively grabs one CPU by running the `yes` tool with the `SCHED_FIFO` scheduling policy with real-time priority 99, the highest priority available. **CAUTION:** The program `yes` runs an *infinite loop*, so only run this on a system with more than one CPU, otherwise, it will starve out the only CPU you have, and effectively lock up the system.

```
# chrt 99 yes > /dev/null
```

Leave this process running so that we can introduce another tool, `ps`, that is familiar to most Linux system administrators. This tool can be used to show the real-time priority of the running processes if used as shown here:

```
# ps -C yes -To pid,rtprio,cmd
```

```
 3860     99 yes
#
```

If we run `yes` directly from the shell, not using `chrt` the results are:

```
# ps -C yes -To pid,rtprio,cmd
 3901      - yes
#
```

The dash means that the process is not using a real-time scheduling policy.

To change the priority of a running process with pid 3901 to real-time, with a value of 85:

```
# chrt -p 85 3901
#
```

that results in:

```
# ps -p 3901 -To pid,rtprio,cmd
 3901     85 yes
#
```

These are the basics of priorities, scheduling policies, and how these knobs can be changed, programmatically and from the command line.

3 signaltest

The `signaltest` utility has several command line options. In this HOWTO we will use the defaults most of the time, introducing some of the command line options only when needed.

By default two threads will be created. These threads will continuously wait for a signal from the other thread, immediately sending a signal back and restarting the loop. At each loop a measurement of the signal delivery latency is done and statistics about the measurements are printed.

Before entering the signal sending loop the threads will change the scheduler policy they use. By default the scheduling policy used will be `SCHED_OTHER` and the priority will be zero. This can be changed using the `--prio` option, that will make the threads use the `SCHED_FIFO` real-time scheduling policy.

To understand how the scheduling policy and priority affects thread behavior, we must set up a test environment where there is competition for the system resources. Some sort of test load needs to be run to be a *determinism disturbance generator* or `DDG`. A `DDG` commonly used by kernel developers is to start a build of the kernel using the parallel make feature to run many simultaneous compiles. To do this we need the kernel sources in some directory. They may be obtained with the following command:

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.1.tar.bz2
```

and then set up with:

```
$ tar xf linux-2.6.21.1.tar.bz2
$ cd linux-2.6.21.1
$
```

This is not needed if you already have some version of the sources in your test machine. Once you are in the kernel sources directory create a separate build directory:

```
$ mkdir /tmp/kbuild
$
```

then, in the next examples, when asked to turn on the `DDG`, run the following commands:

```
$ make O=/tmp/kbuild mrproper
$ make O=/tmp/kbuild allmodconfig
$ make O=/tmp/kbuild -j64
```

Change 64 for a reasonable value in your machine, as the load can make it difficult for you to invoke commands on a non real-time shell.

Now run `signaltest` without any command line parameters:

```
# signaltest
0.43 0.10 0.37 2/175 8003

T: 0 (8000) P: 0 C: 20080 Min: 5
Act: 6 Avg: 6 Max: 77
```

The first line represents the load of the machine, it is the content of the /proc/loadavg file. The number after **Max:** is the maximum signal sending latency experienced so far, 77 microseconds.

In the above example the `DDG` was not running, lets do it now and see how a few minutes of activity will affect the machine load average (the first line) and the maximum latency (the number after **Max:**):

```
# signaltest
198.74 78.00 31.75 88/1389 1094

T: 0 (19557) P: 0 C: 373280 Min: 5
Act: 6 Avg: 7 Max: 89557
```

The `DDG` made the load go to 198, and because of that, the `SCHED_OTHER` scheduling can not provide good determinism to the `signaltest` threads, making them experience signal sending latencies as high as 89 milliseconds.

Looking at the priorities of the `signaltest` threads:

```
# ps -C signaltest -To
pid,tid,rtprio,cmd
 23039 23039 - signaltest
 23039 23097 - signaltest
 23039 23098 - signaltest
#
```

we can see that the third column, `rtprio`, has a dash for all the three threads, meaning that they are not using a real-time scheduling policy.

Now using `chrt` to set `signaltest` priority:

```
# chrt 99 signaltest
176.52 151.10 100.69 318/1459
29348

T: 0 (27421) P: 0 C: 20336 Min: 5
Act: 6 Avg: 22 Max: 60002
```

At first glance it does not seem to help, as the maximum latency is too high at 60ms.

Looking at the priorities for the `signaltest` threads we see the problem:

```
# ps -C signaltest -To
pid,tid,rtprio,cmd
 6147  6147 99 signaltest
 6147  6178  - signaltest
 6147  6179  - signaltest
#
```

The main thread, initiated by `chrt` is indeed real-time and has the specified priority, 99, but the signal sending threads are not real-time. This is so because `signaltest` sets the priority according to the `--prio` command line option.

It is also possible to change the scheduling policy and priority of individual threads. Using the above threads as an example we can do this:

```
# chrt -p 98 6178
```

changing the priority of the first thread, that has a thread id (tid) equal to 6178, to 98:

```
# ps -C signaltest -To
pid,tid,rtprio,cmd
 6147  6147 99 signaltest
 6147  6178 98 signaltest
 6147  6179  - signaltest
```

But to get the maximum latency when using `signaltest` with a real-time scheduling policy we must restart it using the `--prio` (or `-p`) command line option:

```
# signaltest --prio 99
26.05 95.89 130.80 1/1355 26979

T: 0 (26963) P:99 C: 20352 Min: 5
Act: 5 Avg: 9 Max: 43
```

looking at the priorities now shows a different picture:

```
# ps -C signaltest -To
pid,tid,rtprio,cmd
995 995  - signaltest --prio 99
995 996 99 signaltest --prio 99
995 997 99 signaltest --prio 99
#
```

Only the signal sending threads now use a real-time scheduling policy.

Using a real-time scheduling policy and the maximum priority, 99, even with the `DDG` using most of the machine resources we get:

```
# signaltest --prio 99
262.72 181.79 157.44 1/1197 27589

T: 0 (26996) P:99 C: 348208 Min: 5
Act: 13 Avg: 9 Max: 59
```

the worst latency was 59 microseconds, even with a load of 262!