Testing real-time Linux. What to test and how.

Sripathi Kodi
IBM Linux Technology Center
sripathik@in.ibm.com

Chirag H Jog
IBM Linux Technology Center
chirag@linux.vnet.ibm.com

Abstract

This paper describes testing of the real-time (CONFIG_PREEMPT_RT) Linux kernel. It explains how testing the real-time kernel is different from testing the mainline Linux kernel and provides some tips and guidelines about writing test cases for the real-time kernel. It illustrates real-time tests in the Linux Test Project (LTP) suite using examples. It also briefly covers real-time tests that are not part of LTP.

1 Real-Time Operating Systems

This section explains what a Real-Time (RT) Operating System (OS) is, the need for real-time response in enterprise applications and the real-time approach adopted by CONFIG_PREEMPT_RT patchset.

1.1 What is a Real-Time Operating System?

A general purpose OS tries to get the maximum amount of work done using the resources available as effectively as possible. The focus here is on *throughput*. Real time OSes on the other hand focus on *predictability* and *determinism*. Real-time tasks should be scheduled on the processor in a predictable time frame, possibly preempting other non-realtime tasks, even though it compromises overall system throughput. Depending on the implementation and environment, real-time OSes may provide a certain level of latency guarantees to the applications. Depending on this, real-time systems are categorized into hard and soft real-time.

1.2 Need for real-time response in enterprise systems

Real-time systems have traditionally been used in niche environments, often with custom designed hardware. The hardware is very small, typically with just one cpu, sometimes even an embedded device with no support for virtual memory, external storage or TCP/IP networking. The OS is often custom designed for the purpose. They generally do not support standard APIs like POSIX and often need a specialized programming model.

On the other hand, enterprise systems have been multi-processor systems built using commodity hardware. They support virtual memory, various types of storage, good networking options and ability to add new hardware as and when required. The OSes on them support standards like POSIX and they have good support for various programming languages and generic applications. Throughput is the most important criterion in these systems.

Currently a new type of requirement is emerging. Some of the applications that run on enterprise systems have started demanding certain soft real-time characteristics even from enterprise systems because

they need to take advantage of reduced latencies and predictable performance. Some of such domains are defense [1] and financial applications. Advancements in hardware technology like increased processor capabilities, advent of multi-core processors and so on has made it possible to design such systems.

1.3 CONFIG_PREEMPT_RT approach

CONFIG_PREEMPT_RT patches [6], referred to as RT patches henceforth in this paper, were initiated by Ingo Molnar. These patches convert the standard Linux kernel into a soft real-time kernel. Since RT patches apply on top of the standard Linux kernel, all the functionality provided by Linux kernel are also provided by the real-time kernel. As a result, applications designed for the mainline Linux kernel work as-is on real-time kernel.

The RT patches cannot make all areas of the Linux kernel provide real time response. They only concentrate on some areas of the kernel like process scheduling, signal delivery, context switching and so on. On the other hand, external storage devices like disks cannot provide deterministic timing characteristics. Hence any task that performs disk IO cannot expect real-time responses even on real-time kernel. Another such example is network operations, because TCP/IP networking stack doesn't provide real-time response.

Let us try to understand how RT patches provide real-time characteristics to the Linux kernel. To understand this, let us look at some of the important changes introduced by RT patches [2]. Over a period of time, some of the changes introduced by RT patches were found to be useful even on the mainline kernel, hence they were included in the mainline kernel.

High Resolution Timers : This patch provides support for clocks that provide higher resolution. This helps in improving the granularity of timing critical operations. This patch has already been integrated into the mainline kernel.

Threaded interrupts: In the Linux Kernel, interrupt handlers (IRQs) can preempt any other process while they cannot be preempted themselves. Hence interrupt handlers can induce high latencies even in high priority time-critical processes. RT patches convert most of the interrupt handlers into kernel threads. This makes interrupt handlers schedulable entities, just like a regular process on the system. With careful design, this reduces latencies in high priority processes. However, there are some interrupts that cannot be run in threads and hence their functionality is retained in the original form. Timer interrupt is an example of this.

Threaded softirqs: Similar to interrupt handlers, softirqs run as kernel threads in the real-time kernel.

Kernel preemption: The kernel code is made as preemptible as possible by avoiding sections that disable preemption, because non-preemptible portions contribute to high latencies.

Preemptible spin locks: In the mainline kernel, critical sections are protected by spinlocks. When a thread holds a spin lock, it won't be preempted. Large critical sections can induce high latencies in the kernel. RT patches convert spin locks into sleepable mutexes. That is, a thread can be preempted even while it is holding a spin lock. There are a few areas of the kernel that still need old style spin lock implementation that disables preemption. RT kernel provides this capability through what are called *raw* spin locks.

Priority Inheritance: One of the classic problems of designing real-time systems is of priority inversion. A higher priority process can be kept waiting indefinitely for a lock that is held by a lower priority process, because another process of medium priority hogs the cpu. One of the most popular methods of solving this problem is by priority inheritance (PI). During the time a low priority process holds a lock for which a high priority process is waiting, the low priority process *inherits* the priority of the high priority process. This priority boosting prevents medium priority processes from hogging the cpu. PI support has already been integrated into the mainline kernel.

2 Testing the real-time kernel

Most of the testing done on mainline kernel is relevant on the RT kernel as well. However, the focus of the RT kernel is different from that of the mainline kernel. Hence, the importance of some of these tests may be different on RT kernel compared to the mainline kernel.

2.1 Categories of tests

There are 3 broad categories of tests for the real-time kernel.

Functionality: The real-time kernel is expected to provide same functional characteristics as the mainline kernel. That is, all the APIs are expected to behave exactly as they do on mainline kernel. Hence standard functional tests in LTP [3] and Open POSIX test suite [4] are relevant and as important on real-time kernel as they are on the mainline kernel.

Throughput/performance: Performance tests can be run on the real-time kernel just like they are run on the mainline kernel. However, the throughput is likely to be less than that on the mainline kernel. This is because some of the modifications in real-time kernel sacrifice performance in favor of predictability. However, one of the aims of real-time kernel is to be as close to mainline kernel in performance as possible. Hence standard performance benchmarks like SPEC benchmarks, TPC benchmark are relevant on real-time kernel as well. It is important to compare performance of real-time kernel with mainline kernel and find out where performance bottlenecks are. Note, some of these bottlenecks can't be easily fixed if they arise because of the design of the real-time kernel.

Latency: Naturally, this is the most important category of tests on the real-time kernel. These tests concentrate on certain operations of the kernel. They measure the amount of time these operations take (latency) and more importantly the variation in this latency over time and under various circumstances. Some examples of such tests are signal delivery latency, scheduling jitter, scheduling delays, context switch duration and so on. It is also useful to observe the effect of background non-real-time load on the latency tests.

2.2 Setting up the real-time system

It is important to choose a suitable system and configure it for running real-time tests. The following are some of the considerations for this:

Hardware: Important considerations while choosing the hardware are as follows:

- The real-time kernel has been widely used on x86 and x86_64 architectures. It has been tested on PowerPC architecture as well. It may not be possible to run the real-time kernel on some architectures because the patches may not have been ported to that architecture.
- The real-time kernel works best when run directly on the hardware, rather than on a virtualization layer, because the virtualization layer may cause unpredictable latencies.
- Make sure the system has enough memory (RAM) to run the test case. Any swapping of memory pages used by real-time applications will induce very large, unacceptable latencies.
- Some of the hardware may support System Management Interrupts (SMIs) [5], which are controlled by the firmware, that can produce high latencies in the kernel.
- There could be other factors to consider, like NUMA characteristics of the system. Accessing memory from local NUMA node is faster than accessing memory from a remote node. Location of the memory used by the testcase can have an impact on the latencies seen.

Kernel: Download the latest latest RT patches from kernel.org [6] and apply them on the appropriate mainline kernel. The procedure to compile and boot the real-time kernel is same as the mainline kernel [7].

Kernel configuration: Some of the kernel configuration options may have an impact on real-time latencies. These options include Power Management, especially CPU frequency scaling, value of HZ, NUMA configuration and so on. Additionally, most debug options (Kernel hacking section) lead to increased latencies.

Priorities of IRQs and softirqs: As mentioned in the section 2.3 above, the real-time kernel runs most interrupts and softirqs in a thread. Hence 'ps' command can show all of these threads and their priorities as shown below:

```
#  ps -eo comm,pid,class,rtprio | grep -i irq

    sirq-high/0          5 FF      30
    sirq-timer/0         6 FF      30
    sirq-net-tx/0        7 FF      90
    sirq-net-rx/0        8 FF      90
    sirq-block/0         9 FF      30
    sirq-tasklet/0      10 FF      30
    sirq-sched/0        11 FF      30
    sirq-hrtimer/0      12 FF      92
    sirq-rcu/0          13 FF      30
    sirq-high/1         18 FF      30
    ...
    ...
    IRQ-11             119 FF      95
    IRQ-12             408 FF      95
    IRQ-1              409 FF      95
    IRQ-8              420 FF      95
    IRQ-19             438 FF      95
    IRQ-26             489 FF      95
    IRQ-6             1079 FF      95
    IRQ-24            8230 FF      95
    IRQ-4            10449 FF      95
    IRQ-3            10451 FF      95
```

Priorities of IRQ and softirq threads can be changed using 'chrt' command. Special care should be taken while deciding the priority values for these kernel threads, as incorrectly configured priorities could potentially hang the system. Linux distributors that ship real-time kernel typically provide a user-level application which makes it easy to control the priorities of these threads. For example, Red Hat's MRG [8] ships a tool called **rtctl**.

3 Tips for writing test cases

While any code written for mainline kernel will run fine on real-time kernel, programs that need to take advantage of real-time kernel's characteristics need to be carefully designed. The following are the main precautions to be taken to write good test cases for the real-time kernel.

- It is preferable to run tests that measure real-time characteristics as real-time processes (SCHED_FIFO or SCHED_RR). SCHED_FIFO is preferable because it makes the tests more predictable than SCHED_RR. Care needs to be taken while choosing priorities for threads created by the test case. If enough SCHED_FIFO threads with very high priorities are run, they can hog the system completely, with no time left even to service interrupts. This will cause the system to hang.
- Real-time applications cannot tolerate delays caused by swapping out of their pages. Hence, ensure that the memory used by the test case is limited to available RAM.
- Page faults can cause large latencies in applications. A number of page faults happen when the application starts up. These cannot be prevented. However, the test should try to avoid page faults when the real-time criterion is being tested. The following precautions could be taken to prevent page faults during the core of the test when latencies are being measured:
  - The test case should lock all it's memory using mlockall() call.
  - It should not create new threads during this time. Threads could be created beforehand and be kept ready.
  - Dynamic memory allocation (malloc) should be avoided.
  - Calls that do disk IO, network IO or file handling should be avoided.
- Real-time test cases should take into account the number of cpus on the system. For certain types of tests, it may be necessary to restrict the test case to a single cpu on the system. This can be done either using *sched_setaffinity* call in the test case or *taskset* command while starting the test. On some occasions, it may be necessary to run dummy busy threads on all but one of the cpus on the system.
- Sometimes the tests may be sensitive enough to be affected by interrupts being serviced on the cpus where the tests are running. While it is possible to run the tests at priorities above that of interrupt threads, this may not be the best solution always because it can starve the interrupts. In such cases, it may be necessary to shield the cpus on which real-time tests are run from receiving any interrupts. This can be done by writing appropriate values to /proc/irq/<n>/smp_affinity file, where <n> is the IRQ number.
- A real-time test case needs to record time at various places. It is important to use the right calls to record time by understanding the resolution they support. Using clock_gettime() is the preferred way because it supports nanosecond resolution. Note, CLOCK_REALTIME is the wall clock, hence the time value it reads can be affected by time adjustments made by NTP. On the other hand, CLOCK_MONOTONIC is a continuously increasing time value. Hence it is preferable to use CLOCK_MONOTONIC to record time.
- One needs to understand clearly the working of PI mechanism. Without this, the test may report incorrect values. For example, consider the following scenario, where a low priority thread T1 is holding a PI lock L1 on which a high priority thread T2 is waiting. Due to PI, T1 is boosted to the priority of T2 until the time P1 releases L1. T1 now wants to release the lock and record the time at which it releases the lock.

```
T1 (Holding L1)                          T2
                                         T2: Wait for L1
T1: Gets boosted due to PI
T1: Unlock (L1). Loses prio boost
                                         T2: lock(L1)
                                         T2: Do some work
                                         T2: unlock(L1)
                                         T2: schedule()
T1: Time=clock_gettime(...)
```

In the above pseudo code, T1 first releases the lock and then calls clock_gettime() to record the time. However, T1's priority is de-boosted as soon as it releases the lock. Hence it won't be able to call clock_gettime() until the high priority thread T2 gives up the cpu. Hence the time recorded in T1 is wrong. This could happen even if the lock is non-PI. However, PI mechanism makes this problem come vividly alive, because of priority de-boosting when the lock is released. To rectify this, T1 needs to record the time before unlocking the lock.

- If a test runs for a very short time, it is not possible to predict whether some periodic activities like timer interrupts and SMIs interrupt the test case. This might paint an unrealistic picture of the latencies seen by the test case. Hence in such cases, it is useful to run a large number of iterations of the test case and observe both average and maximum values.
- Usage of sched_yield() is discouraged. If sched_yield() is called in a high priority SCHED_FIFO process and if all other threads are of lower priority than it, the thread immediately resumes back on the cpu. This defeats the very purpose of calling sched_yield(), which is to let some other thread run, possibly to change some data on which the high priority thread depends.
- While developing test cases, it is possible that there are bugs in it that make the test busy loop on the cpus, preventing any other normal applications from being run. As a precaution, one could run an sshd thread at a very high priority (SCHED_FIFO, priority 99) on a different port before starting such tests. In case the test case busy loops on the system which makes ssh connections unresponsive, one could still log in through the high priority sshd interface. Please note, this should be used only during test case development, not on production systems.

4 Overview of real-time tests

This section will present an overview of real-time tests, covering the tests that are in LTP as well as the ones outside LTP.

4.1 Overview of real-time tests in LTP

When we started using real-time kernel a couple of years ago, we also started writing test cases for it. We started off with tests for functionality and latencies. Later on, we wrote some stress and performance tests as well. We put these tests on kernel.org [9] and this started attracting contributions from the community. Some of these contributions include support for PowerPC architecture and a number of bug fixes and enhancements. A few months ago we integrated our tests into the LTP [3] to increase visibility of these tests and make it easy for others to use and contribute to. This has resulted in increased community activity in this area. Let's look at some of the real-time tests in LTP to understand what they test and

how. Real-time test cases are under testcases/realtime directory in LTP tree. Figure 1 shows parts of the directory tree.

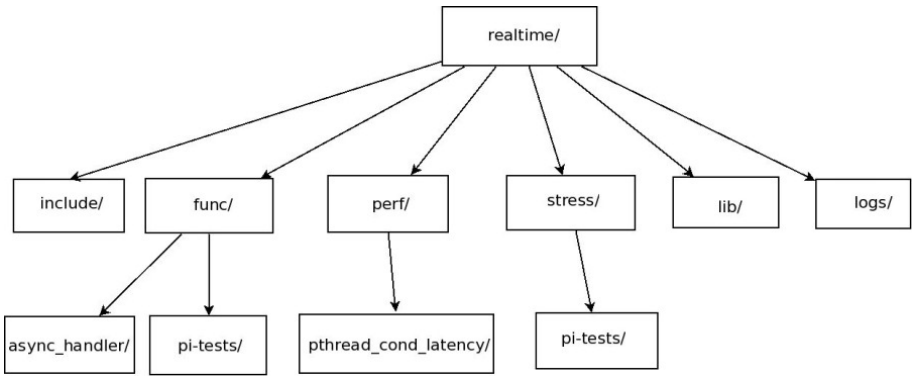

Figure 1: testcases/realtime directory in LTP

The lib/ directory provides utility functions that are used by other test cases. Figure 2 shows parts of the contents of func/ directory, which contains functional tests.

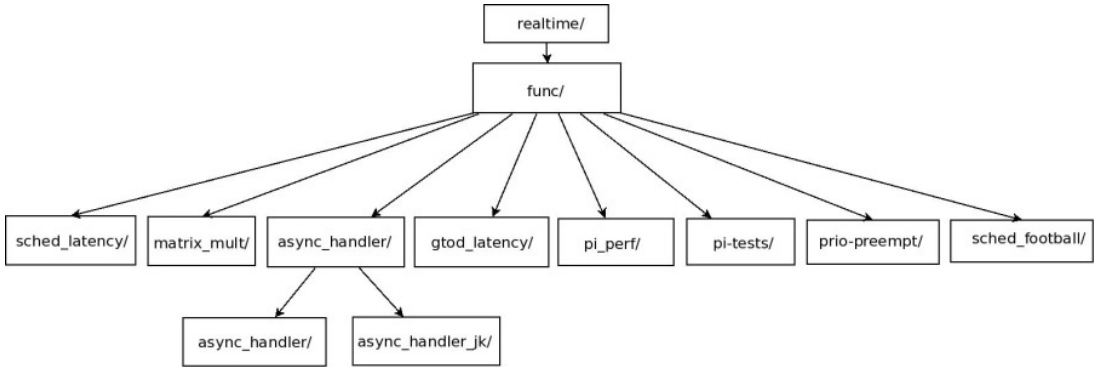

Figure 2: testcases/realtime/func directory in LTP

Each of these tests covers a specific scenario. Some of them were written to test a particular failing scenario. Though those problems may have been fixed now, these tests will serve as a regression test bucket while code changes happen.

Many of these tests run a large number of iterations of a small test scenario. The absolute values from each iteration and the average of them are significant, but not as much as the maximum values. Remember, the biggest goal of the real-time kernel is to provide predictable worst case scenario. Now let us look at the design of some of these test cases.

4.2 Design of pi_perf test case

Let us look at the design of the pi_perf test case. This test case measures the amount of time spent in priority inheritance algorithm. Figure 3 helps us in understanding the test case.

The first important thing this test does is to initialize two pthread barriers. These are used to synchronize between the threads. It then initializes the main mutex used for PI tests using a utility function, init_pi_mutex(). This utility function is implemented in lib/librttest.c. This initializes the pthread_mutex_t as a PI mutex using calls pthread_mutexattr_setprotocol() and pthread_mutex_init().

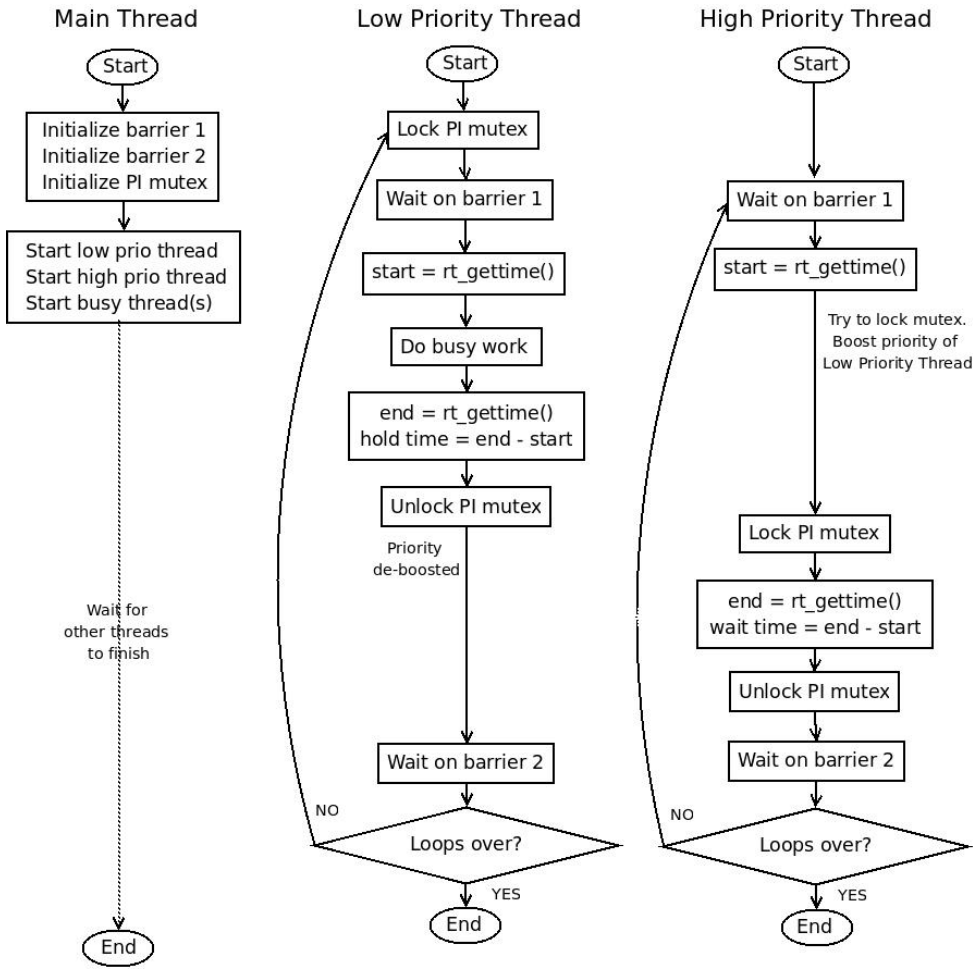

Figure 3: Design of pi_perf test case

Next, the test creates all the threads it needs using the utility function create_fifo_thread(). Again, create_fifo_thread is implemented in librttest.c. It creates a SCHED_FIFO thread pthread using appropriate attribute values. The test creates a low priority (LP) thread at priority 30, a high priority (HP) thread at priority 40 and busy threads at priority 35. Busy threads are as many as the number of cpus on the system.

Now the action switches to the threads. The LP thread always gets the mutex first because of the pthread barrier synchronization used. It first records the time using rt_gettime() utility function. This function uses clock_gettime() to get the system time and converts it to nanoseconds.

The LP thread then does busy work for a fixed number of milliseconds using busy_work_ms() utility function. It then records the unlock time, calculates the hold time and releases the mutex.

The HP thread, when it is past the barrier wait, first records the time. It then tries to get the mutex using pthread_mutex_lock(), but this fails because the low priority thread already has the mutex. By this time, busy threads are already active and are trying to hog the cpu. However, because of the Priority Inheritance, the LP thread now inherits the priority of HP thread. Hence it's priority becomes more than that of busy threads and so busy threads won't be able to preempt it.

When the LP thread releases the lock, it's priority is reset to it's old low value. The HP thread now grabs the mutex and records the time. It calculates the amount of time spent while waiting to get the lock. It also calculates the time difference between the time low priority thread releases the lock to the time high priority thread gets the lock.

This activity is repeated a number of times in a loop and values from each iteration are saved in

arrays. These values are passed to utility functions to find out max, min, average, quantiles and so on. The passing condition for this test is that the difference between the wait time of HP thread and the hold time of LP thread should not be more than a defined maximum, which is 200 microseconds in the test.

4.3 Design of pthread_kill_latency test case

As a second example, let us look at the pthread_kill_latency test case. The aim of this test is to measure the latency involved in sending a signal to a thread. Figure 4 helps us in understanding the test case.

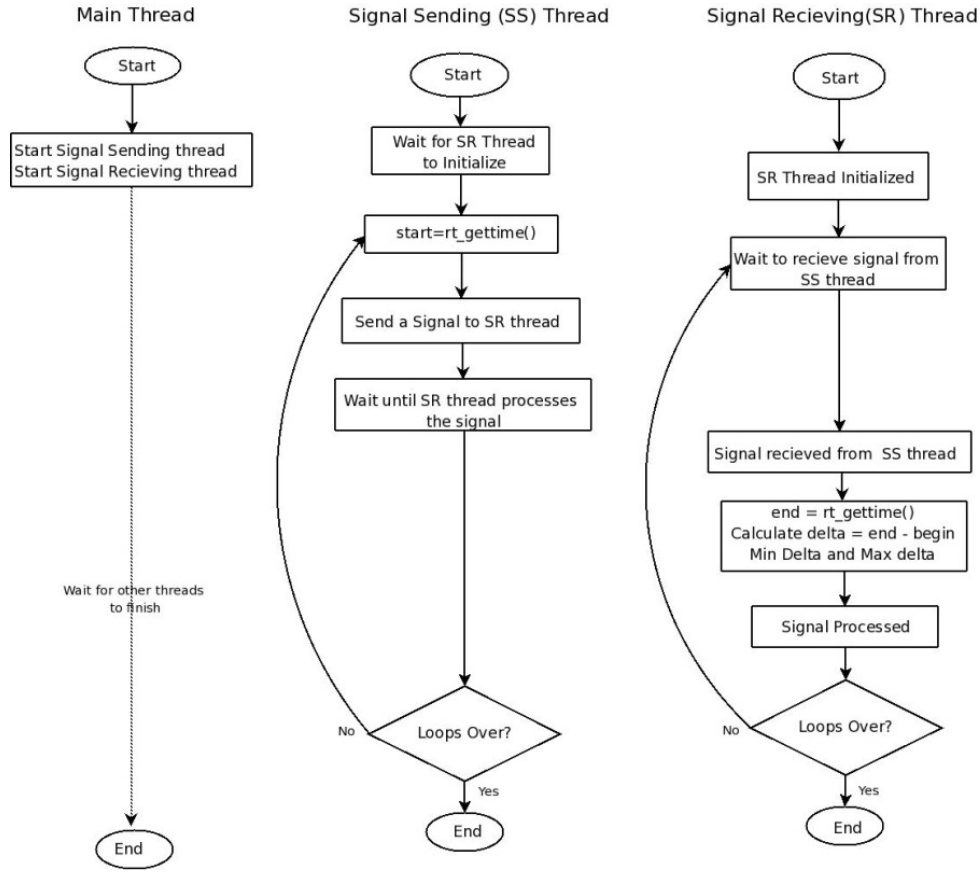

Figure 4: Design of pthread_kill_latency test case

The main thread creates two threads - the signal sending (SS) thread and the signal receiving (SR) thread. It then just waits for the threads to finish.

The Signal Sending (SS) thread waits until the Signal Receiving (SR) thread is initialized. This synchronization is handled by a global atomic variable (atomic_t flag). The SR thread adds the SIGNALNUMBER (shared by both the threads) to its signal set via the sigaddset() and modifies it's signal mask using pthread_sigmask(). It then lets the SS thread know thats it is ready.

The SS thread records the current time (begin) using rt_gettime() and sends the signal to SR thread using pthread_kill(). The SR thread, on receiving the signal, records the time (end) using rt_gettime(). Now, delta = end - begin is calculated as the latency. These steps are run in a loop of 10000 and the maximum and minimum are calculated. The maximum value of latency is used to determine whether the test passed or not. Currently, the acceptable latency is 20 us, although this value can be changed via command line.

In each iteration, after sending the signal SS thread waits until SR thread has finished processing the signal and has done the necessary calculations, before sending the next signal in the next iteration. This synchronization is achieved by the atomic global variable (flag) as mentioned above.

4.4 A comparison of pthread_kill_latency test on mainline and real-time kernels

Comparing the output of functional tests on mainline and real-time kernels can help us visualize the improved predictability provided by the real-time kernel. The charts in Figure 5 show a plot of outputs of pthread_kill_latency test case described in section 5.3.

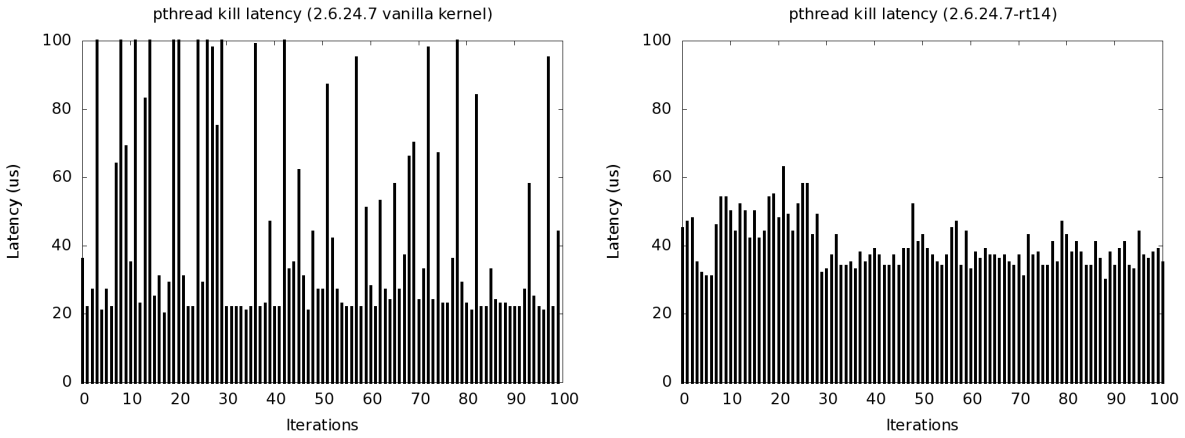

Figure 5: Comparison of mainline and real-time kernels

Figure 5 shows the plots of outputs from 100 iterations of pthread_kill_latency test case. The picture on the left shows the plot on mainline kernel version 2.6.24.7. The picture on the right is the same on real-time kernel version 2.6.24.7-rt14. Taking a close look at the values reveals that though the mainline kernel has the lowest values, the variation in the output is quite a bit. In comparison, real-time kernel has provided more consistent, predictable outputs.

4.5 Real-time Tests outside the LTP

Some of popular real-time tests are not part of the LTP. The most popular of these is the rt-tests suite maintained by Thomas Gleixner [10]. This test suite consists of three tests.

cyclictest: This test is used to determine the accuracy of the High Resolution Timers (hrtimers). Multiple threads are created. Each thread sleeps for a particular predefined amount of time using clock_nanosleep or sys_nanosleep. The actual time a thread slept is calculated. The difference between the time the thread should have slept and the actual time it slept is the latency that is reported. This test measures the minimum latency, maximum latency and average latency.

signaltest: This test measures the round trip for a signal. The test creates N threads. Each thread waits for a signal to be received. The main thread sends a signal to the 0^{th} thread. The 0^{th} thread in turn sends the signal to the 1^{st} thread and so. The N^{th} thread sends a signal to the 0^{th} thread.

Each thread measures the round trip time of the signal i.e the amount of time taken for the i^{th} thread to receive a signal from $(i-1)^{th}$ thread after the i^{th} thread sent a signal to the $(i+1)^{th}$ thread.

This test measures the minimum time taken for any thread to receive a signal after it sent one, the maximum time taken for any thread to receive a signal after it sent one and Average times.

pi_tests: These tests test the Priority Inheritance Mutexes and their ability to avoid Priority Inversion from occurring. These tests are similar to the pi-tests tests which are part of real-time test suite in LTP.

5 How to get started

This section explains how one could get started with real-time testing and predicts the future direction of real-time testing in LTP.

5.1 Where do I begin?

The following are the ways in which someone who is interested in using and contributing to the real-time tests can get started.

- Using real-time tests in LTP is pretty straightforward. There are instructions available along with the test cases. It is important to understand the characteristics of real-time testing and how it is different from testing the regular kernel.
- Bug reporting, handling and patching in the real-time kernel works similar to the mainline kernel. Problems are reported to linux-rt-users mailing list [11] as well as Linux Kernel Mailing List (LKML)[12], copying the relevant people. Since many of the features in real-time kernel eventually make their way into the mainline kernel, kernel developers are interested in knowing about problems seen in real-time kernel as well. Also, on some occasions, real-time kernel exposes certain race conditions that are much harder to recreate on the mainline kernel.
- Debugging a latency problem can be a difficult task. It is a good idea to narrow down the problem as much as possible by testing various kernel versions or various scenarios. There are some useful tools to trace latencies, like latency tracer and logdev [13]. Sometimes, lockdep, lockstat and oprofile also prove very useful.
- Those who wish to contribute to the real-time tests in LTP should go through existing real-time tests both in and outside LTP and understand what they do. Identify any missing functionality or ways to improve the existing test cases. Finally, the work should be submitted to LTP mailing list [14].

5.2 Future

The real-time tests in LTP are still evolving. Hence they can be enhanced/improved in a number of ways. The following list is sort of a wish list of the authors for the future of real-time tests in LTP.

- Clean-up. There may be overlapping functionality in different tests. It will be useful to consolidate such functionality.
- Write new test cases: For functionality that is not covered by existing tests.
- Find out if there are functionalities covered by tests outside LTP that are not covered by tests in LTP. Either convince the authors of those tests to submit them to LTP or write new test cases that provide equivalent functionality.
- Improve stress tests.

Conclusion

Real-time Linux is an emerging area of Linux kernel. Hence there is a lot of need for testing it thoroughly. This paper has explained the existing test cases for real-time Linux and explained how to contribute in this area.

Legal Statement

References

[1] IBM's press release about using real-time Linux in defense sector http://www-03.ibm.com/press/us/en/pressrelease/21033.wss

[2] Steven Rostedt, Darren Hart. "Internals of the RT Patch in" Ottawa Linux Symposium, 2007.

[3] Linux Test Project: http://ltp.sourceforge.net/

[4] Open POSIX test suite: http://posixtest.sourceforge.net/

[5] Intel Architecture Software Developer's Manual Volume 3: System Programming: http://www.intel.com/design/pentiumii/manuals/243192.htm

[6] Real-time patches: http://www.kernel.org/pub/linux/kernel/projects/rt/

[7] Real-time kernel howto on the RT wiki http://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

[8] Red Hat's MRG: http://www.redhat.com/mrg/

[9] Our first real-time test repository: http://www.kernel.org/pub/linux/kernel/people/dvhart/realtime/tests/

[10] Thomas Gleixner's real-time tests http://www.kernel.org/pub/linux/kernel/people/tglx/rt-tests/

[11] linux-rt-users mailing list: http://vger.kernel.org/vger-lists.html#linux-rt-users

[12] Linux Kernel Mailing List: http://vger.kernel.org/vger-lists.html#linux-kernel

[13] Steven Rostedt's logdev: http://rostedt.homelinux.com/logdev/

[14] LTP mailing list: https://lists.sourceforge.net/lists/listinfo/ltp-list