Linux from a real-time perspective

Some definitions of the theoretical part are recalled form Linux point of view. ■

A system with real-time constraints aims to perform work with a guarantee of the time when the work will be finished. This time is often called a *deadline*, and the system is designed with the purpose of not missing any, or as few as possible, deadlines. A system where the consequences of missing deadlines are severe, for example with respect to danger for personnel or damage of equipment, is called a *hard real-time system*. A system where deadlines occasionally can be missed is called a *soft real-time system*.

The work done by a real-time system is often initiated by an external event, such as an interrupt. The nature of the work often requires participation of one or more concurrently executing tasks. Each task participates by doing processing, combined with interaction with other tasks. The interaction typically leads to one or more task switches.

When implementing a real-time system and using Linux as an operating system, it is important to try to characterize possible sources of non-determinism. This knowledge can then be used to configure, and perhaps also modify, Linux so that its real-time properties become more deterministic, and hence that the risk of missing deadlines is minimized, although not guaranteed.

1 Kernel preemption model

A task switch occurs when the currently running task is replaced by another task. In Linux, a task switch can be the result of two types of events:

- As a **side effect of a kernel interaction**, e.g a system call or when the kernel function `schedule()` is called. This type of task switch is referred to as **yield**. The function `schedule()` can be used by kernel threads to explicitly suggest a yield.
- As a result of an **asynchronous event**, e.g. **an interrupt**. This is referred to as preemption and occurs asynchronously from the preempted tasks point of view.

In the kernel documentation, the terms **voluntary preemption** is used instead of yield and **forced preemption** for what here is called preemption. The terms were chosen since, strictly speaking, preemption means to interrupt without the interrupted thread's cooperation. Note that the preemption model only determines when a task switch may occur. The algorithms used to implement preemption models are called **schedulers**.

A task can be preempted depending on whether it executes in **user space** or in **kernel space**. A task executes in user space if it is a thread in a user application. Otherwise it executes in kernel space, i.e. system calls, kernel threads, etc. Tasks can always be preempted in user space. In kernel space, you either allow or disallow preemption at specific places and moments.

The choice of preemption model is a balance between **responsiveness (latency)** and **scheduler overhead**. Lower latency requires more frequent opportunities for task switches which results in higher overhead. Linux offers several different models, specified at build time:

- No Forced Preemption (Server);
- Voluntary Kernel Preemption (Desktop);
- Preemptible Kernel (Low-Latency Desktop);
- Preemptible Kernel (Basic real-time);
- Fully Preemptible Kernel (Real-time).

The last two models require the `Preempt_RT` patch plus kernel configuration.

The server and desktop configurations both rely entirely on yield (voluntary preemption). The difference is mainly that with the desktop option there are more system calls that may yield.

Low-latency desktop introduces kernel preemption. This means that the code is preemptible everywhere except in parts of the kernel where preemption has been explicitly disabled, as for example in **spin locks** (active waits, cf. Wikipedia page for more details).

The preemption models real-time and basic real-time (used mainly for debugging) are not only additional preemption models as they also add a number of modifications that further improve the worst-case latency (cf. Section 4). Real-time model minimizes parts of the kernel where preemption is explicitly disabled.

2 Scheduling under Linux

Linux supports a number of different scheduling policies:

- `SCHED_FIFO` (FIFO scheduling);
- `SCHED_RR` (Round-Robin scheduling);
- `SCHED_DEADLINE` (Earliest Deadline First EDF scheduling);
- `SCHED_OTHER` (also called `SCHED_NORMAL`);
- `SCHED_BATCH`;
- `SCHED_IDLE`.

`SCHED_FIFO` and `SCHED_RR` are the two real-time scheduling policies used in Linux. Each task that is scheduled according to one of these policies has an associated **static priority** value (by default or user defined) that ranges from 1 (lowest priority) to 99 (highest priority). The scheduler keeps a list of ready-to-run tasks for each priority level. Using these lists, the scheduling principles are quite simple:

- `SCHED_FIFO` can be used only with static priorities higher than 0, which means that when a `SCHED_FIFO` threads becomes runnable, it will always immediately preempt any currently running `SCHED_OTHER`, `SCHED_BATCH`, or `SCHED_IDLE` thread. `SCHED_FIFO` is a simple scheduling algorithm without time slicing. For threads scheduled under the `SCHED_FIFO` policy, the following rules apply:
  - A thread that has been preempted by another thread of higher priority will stay at the head of the list for its priority and will resume execution as soon as all threads of higher priority are blocked again;
  - When a thread becomes runnable, it will be inserted at the end of the list for its priority;
  - Task executed at the same level of priority are executed one behind the other in a FIFO mode;
  - A thread runs until either it is blocked by an I/O request or it is preempted by a higher priority thread;
- `SCHED_RR` is a simple adaptation of `SCHED_FIFO` for lower priority (critical and urgent) activities. Everything described above for `SCHED_FIFO` also applies to `SCHED_RR`, except that each thread is allowed to run only for a maximum time quantum. If a `SCHED_RR` thread has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A `SCHED_RR` thread, that has been preempted by a higher priority thread and subsequently resumes execution as a running thread, will complete the unexpired portion of its time quantum.

In a general-purpose (voluntary) Linux, the scheduling `SCHED_FIFO` (or `SCHED_RR`) behavior is constrained by the higher priority of kernel tasks. In a real-time Linux, the `SCHED_FIFO` and `SCHED_RR` behaviors are more efficient because of the use of a fully preemptible kernel.

As long as there are real-time tasks that are ready to run, they might consume all CPU power. A mechanism called RT throttling can help the system to avoid that problem (cf. Section 3).

Under multicore architectures, tasks are statically assigned to CPUs. One reason for doing this is to increase determinism and ensure real-time, for example by making the response time to external events more predictable. Assigning a task to a CPU or a set of CPUs is referred to as setting its **affinity**.

SCHED_OTHER is the most widely used policy. These tasks do not have static priorities. Instead they have a nice value ranging from -20 (highest) to +19 (lowest). This scheduling policy is quite different from the real-time policies in that the scheduler aims at a **fair** (or **equitable**) distribution of the CPU e.g., each task shall get an average share of the execution time according to its nice value.

SCHED_BATCH is very similar to SCHED_OTHER. The difference is that SCHED_BATCH is optimized for **throughput**. The scheduler will assume that the process is CPU-intensive and treat it slightly differently. Consequently, the scheduler will apply a small scheduling penalty with respect to wakeup behavior, so that this thread is mildly disfavored in scheduling decisions.

SCHED_IDLE can be used only at static priority 0. The process nice value has no influence for this policy. It is intended for running jobs at extremely low priority (lower even than a +19 nice value with the SCHED_OTHER or SCHED_BATCH policies).

Since version 3.14, Linux provides a deadline scheduling policy SCHED_DEADLINE. This policy is currently implemented using EDF (Earliest Deadline First) in conjunction with CBS (Constant Bandwidth Server) a variant of TBS (Total Bandwidth Server) useful for multimedia applications [1]. CBS supports resource reservations: each task scheduled under such policy is associated with a budget C_s, and a period T_s, corresponding to a declaration to the kernel that C_s time units are required by that task every T_s time units, on any processor. This makes SCHED_DEADLINE particularly suitable for real-time applications, like multimedia or industrial control, where sporadic events are recurrent. T_s corresponds to the minimum arrival time between subsequent sporadic events, and C_s corresponds to the worst-case execution time needed by the task's jobs induced.

3 Real-time throttling

As long as there are real-time tasks, i.e. tasks scheduled as SCHED_FIFO or SCHED_RR, that are ready to run, would consume all CPU power if the scheduling principles were followed literally. Sometimes that is the wanted behavior, but bugs in real-time threads may completely block the system.

To prevent this from happening, there is a real-time **throttling mechanism** which makes it possible to limit the amount of CPU power that the real-time threads can consume. The mechanism is controlled by two parameters: rt_period and rt_runtime. The semantics is that the total execution time for all real-time threads may not exceed rt_runtime during each rt_period. As a special case, rt_runtime can be set to -1 to disable the real-time throttling.

The throttling mechanism allows the real-time tasks to consume rt_runtime times the number of CPUs for every rt_period of elapsed time. A consequence is that a real-time task can utilize 100% of a single CPU as long as the total utilization does not exceed the limit. The default settings assigns for rt_period a value of 1000000 ns (1 s) and for rt_runtime a value of 950000 ns (0.95 s) give a limit of 95% CPU utilization. The parameters are associated with two files /proc/sys/kernel/sched_rt_period_us and /proc/sys/kernel/sched_rt_runtime_us and can be changed by writing on them new numbers.

4 The patch Preempt_RT

Preempt_RT is a set of changes to the Linux kernel source code. When applied, these modifications will make the Linux kernel more responsive to user-triggered external interrupts and more time-deterministic for tasks execution. It aims to minimize the kernel non-preemptible tasks. This is accomplished by adding and modifying functionality in the Linux kernel. The main functional changes done by Preempt_RT are:

- Converting spin locks to **sleeping locks**. This allows preemption while holding a lock;
- Running **interrupt handlers as threads**. This allows preemption while servicing an interrupt;
- Adding **priority inheritance** to different kinds of semaphores and sleeping locks. This avoids the **priority inversion phenomenon**: scenarios where a lower prioritized process hinders the progress of a higher prioritized processes, due to the lower priority process holding a lock;
- **Lazy preemption** which increases throughput for applications with SCHED_OTHER tasks.

The standard Linux kernel only meets soft real-time requirements: it provides basic POSIX operations for user space time handling but has no guarantees for hard timing deadlines. The `Preempt_RT` patch, developed by Ingo Molnar (a hungarian Linux hacker employed by Red Hat since May 2013) and generic clock event layer with high resolution support (developed by Thomas Gleixner, a german Linux hacker and founder of Linutronix), the kernel gains hard realtime capabilities. The `Preempt_RT` set of Linux kernel's patches are available in `https://www.kernel.org/pub/linux/kernel/projects/rt`.

The `Preempt_RT` functionality is activated, after the `Preempt_RT` patches have been applied and the kernel has been built, by selecting some kernel configuration menu alternative. The main menu alternative is named *Fully Preemptible Kernel* (RT). The performance of a Linux kernel with the `Preempt_RT` patches applied can then be evaluated. A common evaluation methodology involves measuring the **scheduling latency**. In the context of the scheduler, latency is the delay from the occurrence of an event until the handling of the said event. Often it is typically the delay from the time of an interrupt until the time when a task, that is activated as a result of the interrupt, begins executing in user space.

Latency of itself is natural, there is always some latency, it becomes problematic when it exceeds the deadline given by your application's restraints. If it's less than the deadline, it's a success, if it's bigger, it's a failure. There can be many varied causes for high scheduling latencies, some worth mentioning are: a misconfigured system, bad hardware, badly programmed kernel modules, CPU power management, faulty hardware timers, etc.

When trying to determine the system's maximum scheduling latency, the system needs to be put under load. A busy system will in general experience bigger latencies than an idle one. It would be recommendable to run tests for a long time and under different natural and artificial load conditions. It would also be recommendable to stress all sub systems that would be in use on the production system, like disk and network I/O, usb, the graphics processing, etc.

5 Latency testing utilities

A commonly used tools for the check of kernel scheduling latencies are those of the package `rt-tests` [4]:

- `cyclictest` is is a high resolution test program included in `rt-tests`. It is used to verify the maximum scheduling latency for tracking down the causes of latency spikes. `cyclictest` works by measuring the time between the expiration of a timer of a set of threads and when the thread starts running again. Here is the result of a typical test run:

```
# cyclictest -t -n -a -m -p98
/dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 239.09 220.49 134.53 142/1304 23799

T: 0 (23124) P:98 I:1000 C: 645663 Min:      2 Act:    4 Avg:    4 Max:      23
T: 1 (23125) P:98 I:1500 C: 430429 Min:      2 Act:    5 Avg:    3 Max:      23
T: 2 (23126) P:98 I:2000 C: 322819 Min:      2 Act:    4 Avg:    3 Max:      15
T: 3 (23127) P:98 I:2500 C: 258247 Min:      2 Act:    5 Avg:    4 Max:      32
```

  It shows a four CPU core system running one thread (`SCHED_FIFO`) per core at priority 98, with memory locked and `clock_nanosleep` activated (it allows the calling thread to sleep for an interval specified with nanosecond precision). The system is also under a high load due to running `hackbench` in a separate terminal. What is most interesting is the max schedling latency detected, in this case 32 microseconds on core 3 (cf. `cyclictest` man page).

- `hackbench`: an idle kernel (with few running tasks) will tend to show much lower scheduling latencies, it's essential to put some load or stress conditions on it to get a realistic result. This can be done with another utility in the `rt-tests` package called `hackbench`. It works by creating multiple pairs of threads or processes, that pass data between themselves either over **sockets** or **pipes**. To make it run longer add the `-l` parameter, `hackbench -l 1000000` for example (cf. `hackbench` man page).

This tutorial was mainly based on the Wiki article available in [3]

6 Support for Ada locking policies

This section specifies which policies specified by `pragma Locking_Policy` are supported and on which platforms of execution.

GNAT supports the standard `Ceiling_Locking` policy (*Higest Locker Protocol* HLP, or *Immediate Ceiling Priority Protocol* ICPP in the Ada jargon), and the implementation defined `Inheritance_Locking` and `Concurrent_Readers_Locking` policies.

`Ceiling_Locking` is supported on all platforms if the operating system supports it. In particular, it is not supported on VxWorks. `Inheritance_Locking` is supported on Linux, Darwin (Mac OS X), LynxOS 178, and VxWorks. `Concurrent_Readers_Locking` is supported on Linux.

On Linux, if the process is running as root, `Ceiling_Locking` is used by default. If the capabilities facility is installed (`sudo apt-get assume-yes install libcap-dev` on Debian-based distributions, for example), and the program is linked against that library (using `-largs -lcap`), and the executable file has the `cap_sys_nice` capability (`sudo /sbin/setcap cap_sys_nice=ep exe`), then `Ceiling_Locking` is used. Otherwise, it is ignored [2].

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Adacore. Implementation of Specific Ada Features. `https://docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm/implementation_of_specific_ada_features.html`.

[3] Linux Real-Time Wiki. Basic Linux from a Real-Time Perspective. `http://linuxrealtime.org/index.php/Main_Page`.

[4] Russell, Rusty and Zhang, Yanmin and Molnar, Ingor and Sommerseth, David and Gleixner, Thomas and Williams, Clark and Kacur, John. Latency testing utilities `rt-tests`. `https://www.kernel.org/pub/linux/utils/rt-tests/`.