Linux Kernel Development, Technologies

Understanding process thread priorities in Linux

October 3, 2023 | 13 minute read

Imran Khan

1. Introduction

On systems supporting multiple runnable tasks, each task is assigned a “priority” or rank that determines how often it gets the CPU. The idea is that higher priority runnable tasks get CPU before the lower priority ones and tasks with the same priority get CPU in a round robin way.

So on the face of it, priority of a process seems a fairly simple idea. However tracking and managing a task’s priority is quite complicated because the Linux kernel needs to accommodate different use cases. For example it needs to schedule tasks with different scheduling policies (i.e different timing constraints and/or different task selection criteria), it may have to temporarily boost the priority of low priority tasks to unblock a high priority task or there can be several other scenarios where just giving a rank or priority to a task will not give optimum overall performance.

Further a lot of user space tools used for observing and/or changing priority of tasks work with other UNIX variants as well. So these may not show the exact priority values used by the Linux kernel. In order to interpret the results of these tools correctly, we need to understand the relationship between priority values shown by these tools and actual priority values maintained by the kernel.

This article aims to explain different priority values maintained and used internally by the Linux kernel and how to interpret priority values shown by common tools.

2. Linux scheduling policies/classes

Before looking at task priorities, I will briefly describe different scheduling classes/policies supported in the Linux kernel. A scheduling class or policy determines how the next task to run on CPU is selected. It also determines for how long the selected task gets the CPU, if the task does not block or voluntarily gives up the CPU. The Linux scheduler works for both real-time and non real-time tasks. Real time tasks need to respond to an event under a specific time frame, in most of the cases as soon as possible.

For real-time tasks 3 policies are supported, SCHED_RR, SCHED_FIFO and SCHED_DEADLINE.

SCHED_RR (i.e Round Robin) and SCHED_FIFO (i.e First In First Out) correspond to POSIX realtime policies. Tasks under these policies are

selected in order of their priority i.e if there are multiple SCHED_FIFO or SCHED_RR tasks, tasks with higher priority are selected first.

For the other real time policy i.e SCHED_DEADLINE (deadline), tasks are selected based on their deadline.

This means that when there are multiple SCHED_DEADLINE tasks, the task with the earliest deadline is selected first. For SCHED_DEADLINE tasks priority does not impact task selection.

For non real-time tasks 3 policies are supported, SCHED_OTHER, SCHED_BATCH and SCHED_IDLE. Among these SCHED_OTHER is used in almost all of the cases. SCHED_BATCH and SCHED_IDLE are special case policies for non real-time tasks. For SCHED_BATCH tasks, the scheduler assumes that these tasks are CPU intensive and does not require low latency scheduling in response to wake up events.

SCHED_IDLE is used for tasks which get CPU only if there is nothing else to run.

Scheduling policies also have priorities amongst themselves in the following (higher to lower) order:

SCHED_DEADLINE > SCHED_RR/SCHED_FIFO > SCHED_OTHER > SCHED_BATCH > SCHED_IDLE

As can be seen above SCHED_RR and SCHED_FIFO tasks have the same priority and tasks of these policies get CPU in order of their priority.

A task inherits its scheduling policy from its parent and if needed scheduling policy can be changed using relevant system calls.

From this point onwards, tasks with SCHED_OTHER scheduling policy will be called normal tasks, tasks with SCHED_FIFO and SCHED_RR scheduling policy will be called real time tasks and tasks with SCHED_DEADLINE scheduling policy will be called deadline tasks. I will explicitly refer to other scheduling classes as and when needed.

3. Task priorities in the Linux kernel

Linux uses separate priority ranges for normal and real time tasks. For normal tasks a priority range (or nice value) of -20 to +19 is used. Lower nice corresponds to higher priority. In other words the task is being less nicer to other tasks in the system. For real time tasks a priority range of 0 to 99 is used. In this case a higher number indicates higher priority.

Since the deadline tasks are not selected based on their priority, no priority range is specified for these. All deadline tasks get a priority of -1.

Internally, the Linux kernel uses a single range (0-139) that maps each of the above 2 ranges (as shown in picture below). In this range lower value means higher priority.

Fig 1: Priority Range used by the Linux Kernel

As mentioned earlier, the kernel needs to accommodate different use cases that depend on the task's priority. All of such use cases can't be efficiently served using just class based priority values. Hence for each task the kernel maintains 4 types of priorities in that task's task_struct.

These priorities are named as:

- static_prio
- rt_priority
- normal_prio
- prio

static_prio and rt_priority reflect scheduling class based priority values. normal_prio is used under specific scenarios (described below) and prio is the effective priority that the kernel actually uses in its scheduling decisions.

3.1. static_prio

This maps the priority range used for normal tasks and is the priority according to the nice value of a task.

The relationship between **nice value** and **static_prio** can be expressed as:

static_prio = 120 + nice

So for nice values in the range -20 to +19 we have static_prio in the range 100 to 139.

The static_prio of a user task is set at the time of its inception (fork , exec etc.), the kernel does not change it on its own. It can be modified by the user using relevant system calls.

By default the Kernel tasks (kthreads) are created with static_prio corresponding to a nice value of 0. For certain kthreads the kernel may change the static_prio later on. For example for worker threads, kthreads are first created with default static_prio and later the kernel changes static_prio according to nice value specified in worker_pool attribute (pool.attr.nice). This is done after worker kthreads have been created but before they get attached to the worker pool.

In the absence of temporary priority boosting from the kernel, effective priority (i.e prio) of normal tasks is the same as their static_prio.

For normal tasks static_prio effectively determines how often and for how long a runnable task gets CPU. static_prio does not impact CPU time of real-time or deadline tasks. But it impacts the load weight of all tasks. The load on a CPU is determined by using the load weight of all tasks on its runqueue.

3.2. rt_priority

This maps the priority range for real time tasks and indicates real time priority.

Real time priority range supported by Linux, gets mapped to internal priority range using the formula:

MAX_RT_PRIO-1 - rt_priority

Here, MAX_RT_PRIO is 100. So rt_priority of 0 to 99 maps to the internal priority value of 99 to 0. It must be remembered that high rt_priority value signifies high priority and in the kernel low priority value signifies high priority. Hence the above relationship. In absence of any temporary priority boosting from the kernel, effective priority (i.e prio) of a real time task is related to its rt_priority as per above relationship.

rt_priority impacts priority of real-time tasks only and is ignored for normal tasks. The rt_priority of the task gets set at the time of its creation (fork, exec etc) and later can be changed using relevant system calls.

3.3. normal_prio

This indicates priority of a task without any temporary priority boosting from the kernel side. For normal tasks it is the same as static_prio and for RT tasks it is directly related to rt_priority as:

normal_prio = MAX_RT_PRIO-1 - rt_priority

The main purpose of normal_prio is to prevent CPU starvation for other tasks, due to children of one or more tasks of boosted priorities.

In absence of normal_prio, children of a priority boosted task will get boosted priority as well and this will cause CPU starvation for other tasks. To avoid such a situation, the kernel maintains **nomral_prio** of a task. Forked tasks usually get their effective prio set to normal_prio of the parent and hence don't get boosted priority.

3.4. prio

This is the effective priority of a task and is used in all scheduling related decision makings.

As mentioned earlier, its value ranges from 0 to 139 and a lower value represents a higher priority. Usually it depends on static_prio for normal tasks and rt_priority for real-time tasks but under certain special cases (like RT mutex, boosted RCU read side critical sections etc.) the kernel can boost the prio of a task without changing its static_prio or rt_priority.

By having an effective priority the kernel achieves 2 things:

1. It can map diverse priority ranges of different scheduling classes into a single range and utilize that range to get the same end result.

2. In case of temporary priority boosting, the kernel can again revert to usual priority values once the need of temporary priority boosting is done.

Now if we map different priority values maintained by the Linux kernel to different priority ranges of different scheduling policies, we get the following diagram:

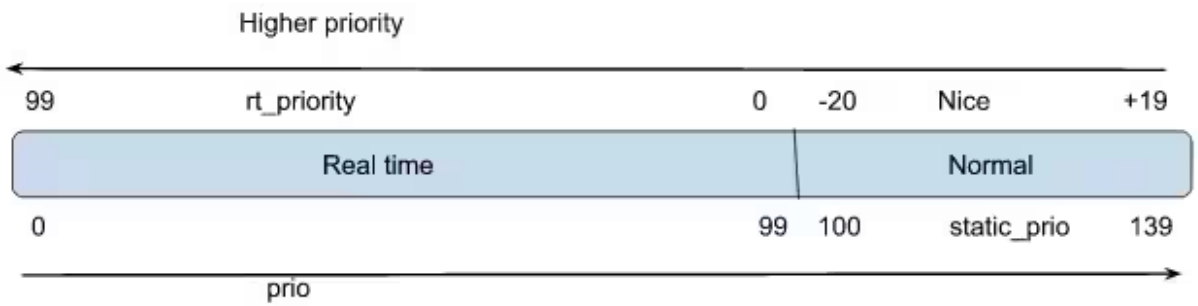

Fig 2: Relationship between priority ranges supported by Linux and different priority values maintained by the Linux kernel

4. How to interpret priorities shown by common tools

Diagnostic tools such as top, ps etc. get process priority values from `procfs` but they don't show the priority values as read from `procfs`. This is because some of these tools work with other Unix variants as well and hence their output conforms to the UNIX standard and this can be different from Linux specific interpretation. So in order to correctly interpret the output of these tools in the context of Linux, one should have a clear understanding of the relationship between the priority values shown by these tools, priority values maintained by the kernel and priority values supported by Linux. Since these tools make use of `procfs`, lets first see what priority values are exported by the kernel to `procfs` and where.

4.1. Task priorities seen in procfs

The task priority can be seen in the `procfs` as well but same priority value is not seen at all places. Hence it is important to understand which/what priority values are visible in `procfs`.