

Linux Kernel: monitoring the scheduler by trace_sched* events

Marco Perronet (Università degli Studi di Torino)

Relatore: Enrico Bini

July 2019

GNU/Linux is a free, open source and community developed operating system. Among GNU/Linux versions, Android is the most popular today, running on most smartphone devices.

Why is this important?

Thanks to its open source nature, it's possible to study the code and get a full understanding of operating systems. This wasn't possible before GNU/Linux.

Objectives

- ① Illustrate how scheduling works in a real operating system
 - Different scheduling algorithms used in the kernel
 - Implementation of the modern scheduler (CFS, the Completely Fair Scheduler)
- ② Write documentation for the scheduler events
 - `ftrace` usage (function and event tracing)

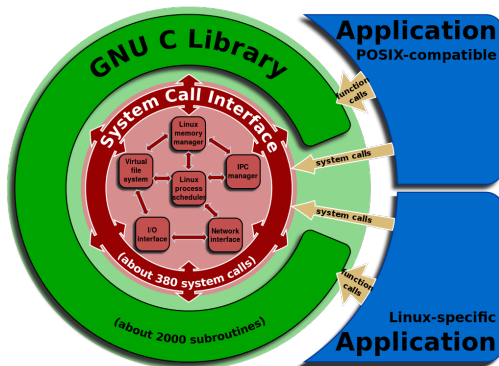


Figure 1: Userspace and Kernelspace

Why "GNU/Linux"?

- Linux is the kernel
- GNU is the application software, running on top of the kernel

The kernel is the core of the operating system. It is the software intended to manage the hardware resources. Some of its roles are:

- Responding to I/O requests
- Managing memory allocation
- Deciding how the CPU time is shared among the demanding processes (*scheduling*)

Scheduler

- Scheduling classes are an extensible hierarchy of scheduler modules
- A task from a scheduling class can be chosen to run only if there are no runnable tasks in classes higher in the hierarchy
- Each process has a scheduling policy associated

Scheduling classes and policies

Scheduling classes	Scheduling policies
stop_sched_class	
dl_sched_class	SCHED_DEADLINE
rt_sched_class	SCHED_FIFO SCHED_RR
fair_sched_class	SCHED_NORMAL SCHED_BATCH SCHED_IDLE
idle_sched_class	

Tuning and extending the scheduler

The workload on servers is different from the workload on desktops (CPU bound vs I/O bound). The scheduler must be changed accordingly:

- Implementing a new scheduling policy. Scheduling classes are made to be extensible, so scheduling policies are handled by the scheduler without the core code assuming about them too much.
- Changing the existing scheduler's behavior with tunable values

Source files

- `dl_sched_class` – `kernel/sched/deadline.c`
- `rt_sched_class` – `kernel/sched/rt.c`
- `fair_sched_class` – `kernel/sched/fair.c`
- Core code shared by all classes – `kernel/sched/core.c`

Tuning and extending the scheduler

```
1 // Code from ./kernel/sched/fair.c
2 /* The idea is to set a period in which each task runs once.
3  *
4  * When there are too many tasks (sched_nr_latency)
5  * we have to stretch this period
6  * because otherwise the slices get too small.
7  */
8 static u64 __sched_period(unsigned long nr_running) {
9     if (unlikely(nr_running > sched_nr_latency))
10         return nr_running * sysctl_sched_min_granularity;
11     else
12         return sysctl_sched_latency;
13 }
```

- `sysctl_sched_min_granularity` – The minimum time a task is allowed to run on a CPU before being preempted
- `sysctl_sched_latency` – The default scheduler period

What if we tweak the granularity?

Completely Fair Scheduler (CFS)

CFS tries to model an ideal multi-tasking CPU where each task runs for the same amount of time. Total fairness would mean that with n tasks, every task receives $\frac{1}{n}$ of the processor's time.

In order to know which task deserves to run next, every task keeps track of the total amount of time that it has spent running (*runtime*).

Picking the next task

When scheduling, CFS chooses the task with the smallest total runtime. However, this approach ignores the priorities of the tasks. Instead, the runtime of each task is weighted with its priority: we call this value *virtual runtime*.

Completely Fair Scheduler (CFS)

Virtual runtime

Vruntime is the absolute runtime of the task, weighted based on task priority. For high priority tasks, *vruntime* is less than the real time spent on the CPU. In this case, *vruntime* grows slower than real time. The opposite is true for low priority tasks.

The runqueue is implemented through a red-black tree, which is ordered with the virtual runtime.

Completely Fair Scheduler (CFS)

Both the timeslice and the absolute runtime (`delta_exec`) must be weighted.

$$assigned_time = target_latency \frac{task_weight}{total_weight} \quad (1)$$

$$vruntime = delta_exec \frac{weight_of_nice_0}{task_weight} \quad (2)$$

- *Equation 1 ensures fairness*: the task's timeslice is proportional to its weight
- *Equation 2 ensures good interactivity*: interactive tasks will have a low vruntime, and will be often picked by CFS

ftrace (Function tracer)

ftrace is a debug tool embedded in the kernel. It is also useful to approach and understand the code.

ftrace can perform function and event tracing.

Function tracing

Traces the path taken by kernel functions. Entry and exit point of the functions are traced, so the total duration of the function can be calculated. This allows latencies to be easily detected.

Event tracing

Based on static tracepoints in the code, which are called just like functions. Unlike function tracing, tracepoints are static, so they cannot be toggled at runtime.

Function tracing

- Thanks to code instrumentation, function tracing creates zero overhead when it's not used.
- It's possible to filter what is being traced: we can dynamically activate tracing only on functions from a single subsystem, or on one function alone.

Code instrumentation

At compile time, extra assembly instructions are generated to help debuggers and analysis tools. `ftrace` exploits `gcc`'s code instrumentation feature, and uses *runtime injection* for dynamic toggling.

Function tracing

```
1      0)          | scheduler_tick() {
2      0)    0.094 us |     _raw_spin_lock();
3      0)    0.116 us |     update_rq_clock.part.84();
4      0)          |     task_tick_fair() {
5      0)          |         update_curr() {
6      0)    0.086 us |             update_min_vruntime();
7      0)    0.093 us |             cpuacct_charge();
8      0)    1.631 us |         } /* update_curr */
9      0)    0.074 us |         update_cfs_shares();
10     0)    0.124 us |         hrtimer_active();
11     0)    4.320 us |     } /* task_tick_fair */
12     0)          |     cpu_load_update_active() {
13     0)    0.069 us |         tick_nohz_tick_stopped();
14     0)          |         cpu_load_update() {
15     0)    0.088 us |             sched_avg_update();
16     0)    0.940 us |         } /* cpu_load_update */
17     0)    2.419 us |     } /* cpu_load_update_active */
18     0)    0.102 us |     calc_global_load_tick();
19     0)          |     trigger_load_balance() {
20     0)    0.094 us |         raise_softirq();
21     0)    0.183 us |         nohz_balance_exit_idle.part.85();
22     0)    1.890 us |     } /* trigger_load_balance */
23     0) + 13.238 us | } /* scheduler_tick */
```

Event tracing

Event tracing is performed at specific points in the code known as *tracepoints*. It's less efficient than function tracing because it doesn't use runtime injection.

Tracepoint functions are generated by the `TRACE_EVENT(...)` macro, which allows developers to quickly declare their own events to trace from outside the kernel.

Event tracing

```
1  static void update_curr(struct cfs_rq *cfs_rq) {
2      struct sched_entity *curr = cfs_rq->curr;
3      u64 now = rq_clock_task(rq_of(cfs_rq));
4      u64 delta_exec = now - curr->exec_start;
5
6      if (unlikely((s64)delta_exec <= 0))
7          return;
8
9      curr->exec_start = now; // Reset exec_start
10     schedstat_set(curr->statistics.exec_max, max(delta_exec,
11     ↪ curr->statistics.exec_max));
12     curr->sum_exec_runtime += delta_exec; // Non-weighted runtime
13     schedstat_add(cfs_rq->exec_clock, delta_exec);
14     curr->vruntime += calc_delta_fair(delta_exec, curr); // Applies
15     ↪ vruntime equation
16     update_min_vruntime(cfs_rq);
17
18     if (entity_is_task(curr)) {
19         struct task_struct *curtask = task_of(curr);
20         trace_sched_stat_runtime(curtask, delta_exec,
21         ↪ curr->vruntime); // Tracepoint
22     }
23 }
```


Event tracing

```
1 # tracer: nop
2 #
3 # entries-in-buffer/entries-written: 116546/459475   #P:4
4 #
5 #           _-----=> irqs-off
6 #           / _-----=> need-resched
7 #           | / _---=> hardirq/softirq
8 #           || / _--=> preempt-depth
9 #           ||| /      delay
10 #  TASK-PID   CPU#  | |||   TIMESTAMP  FUNCTION
11 #   | |       |   | |||   |           |
12 <idle>-0     [000] d...   611.283814: sched_switch:
↪ prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==>
↪ next_comm=Xorg next_pid=1450 next_prio=120
13 Xorg-1450    [000] d...   611.283921: sched_stat_runtime: comm=Xorg
↪ pid=1450 runtime=117083 [ns] vruntime=17539094302 [ns]
14 Xorg-1450    [000] d...   611.283924: sched_switch: prev_comm=Xorg
↪ prev_pid=1450 prev_prio=120 prev_state=S ==> next_comm=swapper/0
↪ next_pid=0 next_prio=120
15 <idle>-0     [000] d...   611.283957: sched_switch:
↪ prev_comm=swapper/0 prev_pid=0 prev_prio=120 prev_state=R ==>
↪ next_comm=Xorg next_pid=1450 next_prio=120
```

Interfacing with ftrace

```
marco@turing-machine:/proc$ ls
1      1324    142      2      27047    393      518      949      ioports    slabinfo
10     1339    1431     20      27059    3990     519      9725     irq        softirqs
103    1349    1433     21      27505    3999     520      9727     kallsyms   stat
1045   1354    1434     2103    27506    40       521      99       kcore      swaps
1046   1359    14414    2106    27553    4005     533      acpi      keys       sys
105     136     1442     2107    27707    402      5362     asound    key-users  sysrq-trigger
1052   1362    14766    2108     28      405      5381     buddyinfo kmsg       sysvipc
1054   1368    1494     218     296     41       540      bus       kpagecgrou thread-self
106     1375    15       22      3        412     546      cgroups   kpagecount timer_list
107     1376    1571     220     30      415     586      cmdline   kpageflags tty
1095   13880   16       22715    31      42      62       consoles loadavg     uptime
1097   139     17515    22763    32      43      64       cpuinfo   locks       version
11     1390    1763     23285    32192    47       6468     crypto    meminfo     vmallocinfo
1101   1393    17765    23486    324     48       664      devices   misc        vmstat
12     14     17815    23705    329     49       670      diskstats modules      zoneinfo
1275   140     179     23751    33      5        699     dma        mounts
1297   1400    18      24      331     50      7        driver    mtrr
1298   1404    180     248     34      511     7981     execdomains net
13     1408    18500    25      35      512     7995     fb         pagetypeinfo
1302   141     19      25180    355     513     8        filesystems partitions
1310   1413    1968     26      36      514     9        fs         sched_debug
1318   1418    1970     27      38      516     9087     interrupts schedstat
1322   14186    19736    27045    39      517     92       iomem     self
```

Figure 2: The procfs special filesystem

Interfacing with ftrace

```
root@turing-machine:/sys/kernel/debug/tracing/events/sched# ls
enable          sched_process_fork  sched_stick_numa
filter          sched_process_free  sched_swap_numa
sched_kthread_stop sched_process_hang   sched_switch
sched_kthread_stop_ret sched_process_wait   sched_wait_task
sched_migrate_task sched_stat_blocked    sched_wake_idle_without_ipi
sched_move_numa    sched_stat_iowait     sched_wakeup
sched_pi_setprio   sched_stat_runtime    sched_wakeup_new
sched_process_exec sched_stat_sleep       sched_waking
sched_process_exit sched_stat_wait
```

Figure 3: Scheduler events in the tracefs special filesystem

- Enable scheduler events:

```
echo 1 > enable
```

- Enable only the sched_stat_runtime event:

```
echo 1 > sched_stat_runtime/enable
```

KernelShark

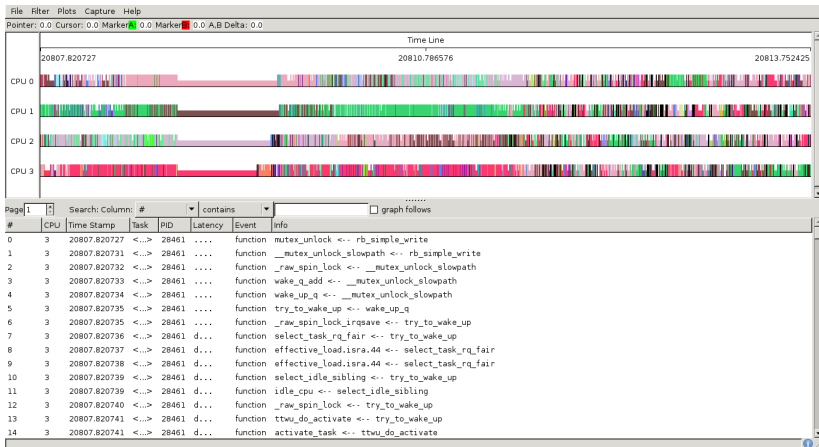


Figure 4: Function tracing with KernelShark

KernelShark



Figure 5: Event tracing with KernelShark

Thank you!