Student notes

Linux
Performance Analysis and Tuning

06. Processors: usage

Nijmegen, The Netherlands

Version: 12a

Student notes

During its life time, a thread within a process is almost continously switching states. The current state is shown in the S column of the `ps -l` output:

```
$ ps -Lel
F S   UID   PID  PPID   LWP  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
....
0 T  5101 25246 17523 25246  0  80   0 - 27542 -      pts/130  00:00:00 more
0 S  1049 26143 26142 26143  0  80   0 - 37235 do_wai pts/181  00:00:00 bash
0 R  1049 26164 26143 26164  0  80   0 - 29222 -      pts/181  00:00:00 ps
```

State *Running* — on CPU (tool shows R)
: At this very moment, the tread is burning CPU cycles, in user mode or in kernel mode. The number of CPUs obviously puts an upper boundary to how many threads can be in this state simultaneously. The output shown above is from a single-CPU system. Enevitably, the `ps` command itself will be on the CPU in this case while it is producing this output.

State *Sleeping* (tool shows S or D)
: The thread is waiting for some event to happen. This situation only occurs during the execution of a system call in kernel mode. The kernel programmer decides to make this state interruptable by a signal (S) or not (D). Chapter already covered this "Sleep and Wakeup" mechanism extensively.
The thread leaves this sleeping state when a different piece of kernel code issues a wakeup call, because the waited-for event has occurred, or because a signal arrived (the latter only in case of S).

State *Running* — wants CPU (tool shows R)
: A thread leaving the sleeping state wants to go back to a CPU. However, the number of CPUs is limited and most probably all CPUs are occupied by other threads. Therefore, a woken-up thread initially enters this "waiting room" state[1]. As soon as a CPU becomes available, the CPU scheduler will select an appropriate candidate from this waiting room and forces a context switch.

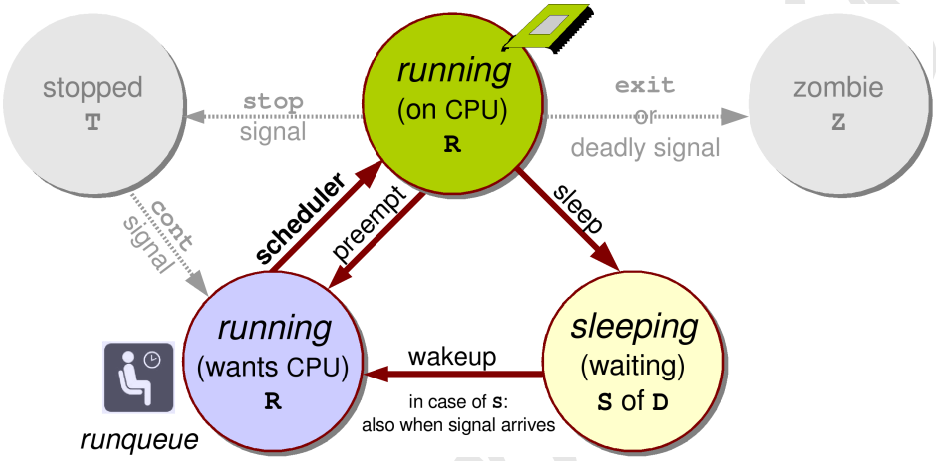
If a thread is about to switch from "running — wants CPU" to "running — on CPU", it gets assigned a *timeslice*: a maximum amount of CPU time it will be allowed to consume. If a thread voluntarily enters the sleeping state before its slice is over, that's called a "voluntary context switch" and is considered to be good behavior. If the thread does *not* voluntarily give up its CPU before its slice is over, it will be forcibly removed (*pre-empted*). It will be sent back to the waiting room with "running — wants CPU" threads and the CPU scheduler will select another thread to run. This is called an "involuntary context switch" and is considered bad behavior.
File `/proc/pid/tasks/tid/status` contains the relevant counters.

1. Unfortunately, Linux uses the R both for "Running — on CPU" and "Running - wants CPU". Other members of the UNIX family distinguish between "running" and "runnable" states.

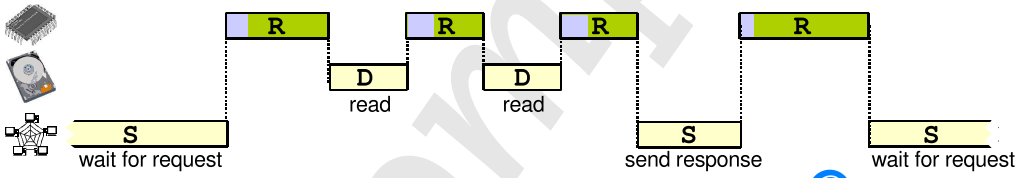
Threads — state transitions

Thread state

- shown by column **S** in
  - **atop**
  - **top**
  - **ps -l**
  - ....



- *example:* database server transaction



v12a – h06 – 1

@ at computing

Notes

Figure 1

Student notes

Each thread runs with a specific scheduling policy. This is inherited from the parent process via the `fork` system call. The `systemd` or `init` daemon (PID 1) runs with scheduling policy SCHED_NORMAL (a.k.a. SCHED_OTHER) which makes this the default for all its descendents.

The SCHED_FIFO and SCHED_RR are for *realtime* processes. The threads of these processes get a fixed priority on a scale from 1 (lowest) to 99 (highest). Suppose that we start two realtime processes: one with priority `10` and the other one with priority `11`, and that we are running on a single CPU system. The thread with priority `10` will only be able to run if the priority `11` thread is sleeping. This holds true both for the SCHED_FIFO and SCHED_RR policies.
However, if two realtime processes are started both with priority `10`, then a difference between SCHED_FIFO and SCHED_RR shows up:

SCHED_FIFO
: If one of the threads has got hold of the CPU, then the other can only get his turn if the first one (the one on the CPU) enters the sleep state voluntarily, or executes the special system call `sched_yield`.

SCHED_RR
: If one of the threads has got hold of the CPU, and then consumes his entire timeslice, the CPU will be taken away, and the thread will be put on the runqueue behind the last thread of equal priority. The result is that the other thread will get the CPU first. If only two threads share the same priority and always fully consume their timeslice, they will thus get alternate turns on the CPU.

The SCHED_NORMAL policy is meant for *timesharing* threads. These all get priority `0`, which is below any realtime priority. However, these time sharing threads also need some priority mechanism between them. This is regulated by their *nice* values. The same holds for SCHED_BATCH.

The `atop` command shows the current policy per process/thread when using keystroke `s` (or flag `-s`).
The `chrt` command is available to change policy and priority of a process. Setting a higher priority or selecting a realtime policy requires privileges.
Examples:

```
# chrt -r 17 myappl&          ← start new program realtime, round-robin, priority 17
[1] 15463
# chrt -p 15463               ← show policy and priority of process 15463
pid 15463's current scheduling policy: SCHED_RR
pid 15463's current scheduling priority: 17
# chrt -p -f 19 15463         ← set policy fifo and priority 19 for process 15463
```

Scheduling policy

POSIX.1b: every thread runs with *scheduling policy*

- ***SCHED_FIFO*** **— realtime first-in-first-out**
  - realtime with priority 1 (low) till 99 (high)
  - take highest priority, take longest waiter with that priority
  - only rescheduling after voluntary switch (sleep)

- ***SCHED_RR*** **— realtime round-robin**
  - realtime with priority 1 (low) till 99 (high)
  - take highest priority, take longest waiter with that priority
  - rescheduling after voluntary switch (sleep) or
    after involuntary switch (preempt after timeslice expires)

only for privileged users

- ***SCHED_NORMAL*** *a.k.a. SCHED_OTHER* **— *timesharing***
  SCHED_BATCH / SCHED_IDLE
  - timesharing with priority 0

Policy and priority can be defined with command **`chrt`**

Notes

Figure 2

Student notes

Each thread (process) has a *nice* value, which is shown in the NI column of the ps -l output:

```
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1049 15317 15312  0  76   0 -  1156 wait4  pts/1    00:00:00 bash
0 R  1049 15331 15317  0  95   0 -   551 -      pts/1    00:00:00 ps
```

The nice value is partly responsible for the priority of the thread and thus can be used to impose an artifical handicap or advantage upon a certain process (thread), with respect to the other processes. The nice value as specified with the nice command acts *relative* to the inherited value, as this example shows:

```
$ nice -n 14 bash            ← activate new shell with lower priority
$ nice -n 3  ps -l           ← start ps from new shell with even lower prio
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1049 15317 15312  0  76   0 -  1156 wait4  pts/1    00:00:00 bash
0 S  1049 15347 15317  0  89  14 -  1156 wait4  pts/1    00:00:00 bash
0 R  1049 15363 15347  0  95  17 -   551 -      pts/1    00:00:00 ps
```

Specifying a negative value with the nice command is permitted with root privileges. This will increase the priority of the program:

```
# nice -n -13 sleep 10&      ← sleep at high priority
# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0 15385 15368  0  67 -13 -   986 -      pts/1    00:00:00 sleep
```

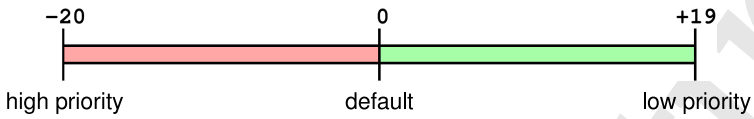
The renice command is available to modify the nice value of an already running process. The value specified with it is an *absolute* value:

```
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1049 15429 15317  0  76   0 -   986 -      pts/1    00:00:00 sleep
$ renice 7 -p 15429          ← renice based on PID
15429: old priority 0, new priority 7
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1049 15429 15317  0  87   7 -   986 -      pts/1    00:00:00 sleep

$ renice 7 -u gerlof         ← renice all processes owned by this user
1049: old priority 0, new priority 7
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1049 15317 15312  0  82   7 -  1156 wait4  pts/1    00:00:00 bash
0 S  1049 15429 15317  0  87   7 -   986 -      pts/1    00:00:00 sleep
```

Nice value

Priority for timesharing threads determined by *nice* value



- extent to which thread is nice to other threads
- effectively determines timeslice
- inherited from parent
- shown by **ps** (**-l**), **top**, **atop**, ...
- can be changed with
  - command **nice**: **nice -n *relvalue* *cmd parl ....***
  - command **renice**: **renice *absvalue* -p *pid***

Notes

Figure 3

Student notes

The CPU scheduler by itself is *not* a process nor a thread; it is a kernel subroutine (named `schedule`) that is called by the thread that still has a CPU, but wants to switch to sleeping state. The subroutine is also called when a thread is taken off the CPU because its timeslice is over. This `schedule` subroutine selects a successor to the current thread from the runqueue of that CPU (i.e. the threads in "Running — wants CPU" state). Then it arranges for the context switch to that successor thread.

The scheduler has a number of considerations to take into account:

- Always prefer a realtime thread (policy SCHED_FIFO or SCHED_RR) over a timeharing thread (policy SCHED_NORMAL or SCHED_BATCH).
- Make sure that interactive processes get a good response time, even if they compete with heavy batch processes that don't have an artificially increased nice value (decreased priority).
- Make sure that a CPU-intensive timesharing thread does not monopolize its CPU. This is the purpose of the timeslice, which we have discussed earlier.
- Allow a user to artificially influence a priority, via the nice mechanism for time sharing processes, and the POSIX-priority for realtime processes.
- Prevent total starvation of a low priority thread (e.g. a thread running with nice value `+19`). They shouldn't get much CPU time, but a few crumbs are a minimum.
- *Multiprocessor consideration:*
  Preferably assign a thread to the same CPU that it had last time, to seize upon the opportunity that some of its text or data may still be present in the L1, L2, or L3 caches, and because its MMU translation tables might still be present in the "Translation Lookaside Buffer" (TLB). Assigning a thread to a different CPU causes the context switch to be more laborious, and requires text and/or data to be retrieved from main memory into the caches of that newly assigned CPU again.
- *Multiprocessor consideration:*
  A user should have the possibility to explicitly assign the thread(s) of a process to a particular CPU. The default is to select an arbitrary CPU for each context switch, obviously taking CPU affinity (see previous bullet) into account.
- *NUMA consideration:*
  Preferably assign a thread to the a CPU on the same NUMA node that also holds the memory pages of of the process involved.
- *NUMA consideration:*
  A user should have the possibility to explicitly assign the thread(s) of a process to a particular node.

CPU scheduler – general approach

Kernel subroutine **`schedule`** — thread scheduler

- called by executing thread to choose and activate successor
- searches for running thread in runqueue and forces context switch

- design considerations
  - realtime always higher priority than timesharing
  - garantee good response times for interactive processes, even when running in combination with batch processes
  - preempt CPU-intensive threads after timeslice expired
  - offer possibility to influence priority manually (nice)
  - avoid starvation of low-priority threads
  - multi core and NUMA behavior
    - thread preferably activated on same CPU ("CPU affinity") and same node
    - offer possibility to bind threads (processes) to specific CPU(s) or node(s)

Notes

Figure 4

Student notes

The so-called "O(1)" scheduler implementation was introduced in the 2.6 kernel and survived until version 2.6.22. It has 140 priorities numbered `0-139`. A lower priority number means a higher effective priority.

The range `0-99` is reserved for realtime threads (SCHED_FIFO and SCHED_RR). The POSIX 1.b standard defines priority number `0` to be the lowest, and `99` to be the highest. The Linux kernel has inverted this. Therefore, the POSIX priority `rt_prio` is subtracted from `99` to calculate the Linux priority: POSIX priority `99` thus becomes Linux priority `0` (highest) and POSIX priority `1` becomes Linux priority `98` (lowest). Real time priorities are static, and therefore independent of the thread's behavior.
The nice value has no influence on the priority of realtime threads, although the nice value is used to calculate the length of the timeslice for threads with SCHED_RR policy, similar to the timeslice calculation for timesharing threads.

The priority range `100-139` has been reserved for timesharing threads (SCHED_NORMAL and SCHED_BATCH). This corresponds to priority `0` of the POSIX 1.b standard. The calculation of the priority of a timesharing thread is based upon the nice value (more or less static) and the recent behavior (varies). The nice value, with `120` added to it, is used as the base for the calculation. Since the range of the nice values is `−20` till `+19` this new range becomes `100-139`, which produces the timesharing range that we saw at the start of this paragraph.
To get at the actual priority, a *behavioral component* is added to this base value. This component ranges from `+5` (bad behavior) to `−5` (good behavior). Bad behavior means that this thread recently used much CPU time (state "Running — on CPU"). Good behavior means that the thread recently spent sufficient time in the sleeping state. This mechanism makes sure that an interactive process can peacefully co-exist with a heavy batch process, even if they have identical nice values: the batch process consumes much CPU time, which decreases its own priority.

Whenever a timesharing thread has been chosen by the scheduler, a *timeslice* will be calculated — the maximum time that this thread is allowed to use the CPU. This also holds for a realtime thread with policy SCHED_RR (in case there is another running thread with the same priority). The duration of the timeslice is determined by the nice value, according to two disjunct linear scales, one for the positive and one for the negative nice values:

Nice value	*Time slice*	
−20	800 msec	
|	|	← 35 milliseconds per nice-point
0	100 msec	
|	|	← 5 milliseconds per nice-point
+19	5 msec	

Notice that the timeslice values are independent of the number of threads in the runqueue, which means that the scaling is bad.

O(1) scheduler implementation

O(1) scheduler < kernel 2.6.23

- 140 priorities
  low value is high priority
  - realtime 0 - 99

    static priority:
    `prio = 99 – rt_prio`

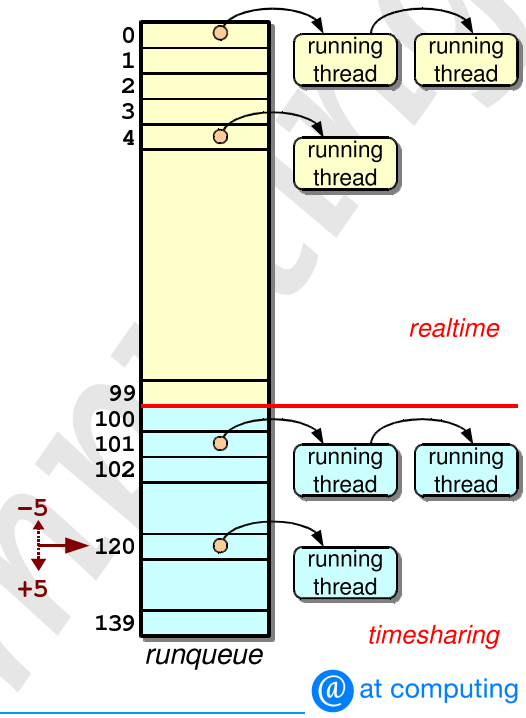
  - timesharing 100 - 139

    variable priority:
    `prio = (nice + 120) + behavior`

    behavior varies from +5 (bad behavior)
    till −5 (good behavior)

    example: for process with nice value 0,
    priority varies from 115 till 125

- static timeslice ☞ bad scaling



v12a – h06 – 5

@ at computing

Notes

Figure 5

Student notes

Kernel version 2.6.23 introduced an entirely new scheduling model. It contains different *scheduling classes*. This is a hierarchical model in which an overall regulator (the *scheduler core*) calls upon a number of "sub-schedulers" in a defined sequence. A "scheduling class" is the implementation of one of these "sub-schedulers". The first "Running — wants CPU" thread to be delivered by a sub-scheduler will be selected, and a context switch follows.

Currently, four standard scheduling classes are available.
The first class to be called upon is the *deadline class*. This class checks if a deadline thread in state "Running — wants CPU" needs some runtime on the processor. If so, it presents that thread to the scheduler core. Alternatively, it reports that no suitable thread could be found.

If no suitable deadline thread could be found, the scheduler core calls upon is the *realtime class*. It will check for the presence of realtime processes in "Running — wants CPU" state, and present the one with the highest priority to the scheduler core. Otherwise, it reports that no suitable thread could be found.

If no suitable realtime thread could be found, the scheduler core calls upon the *fair class*. This class scans the collection of time sharing threads in "Running — wants CPU" state, selects the best candidate, and delivers it to the scheduler core. But this again can possibly find no candidate at all.

If no suitable time sharing thread could be found, the *idle class* will be called upon. This always is the lowest one in the hierachy, and will deliver the so-called *idle thread* that each CPU has. This is a thread that stops the CPU until other threads switch to "Running — wants CPU" state again.

This model is open-ended, and allows for new classes, programmed as kernel modules, to be inserted in between the existing ones.

Every scheduling class implements one or more policies. In the next slides, the implementation of the deadline class, the realtime class and the fair class will be covered.
The following classes/policies have been defined:

Class	*Policy*	*Numerical policy*
realtime	SCHED_FIFO	1
realtime	SCHED_RR	2
fair	SCHED_NORMAL	0
fair	SCHED_BATCH	3
fair	SCHED_IDLE	5
deadline	SCHED_DEADLINE	6
(future)	SCHED_ISO	4

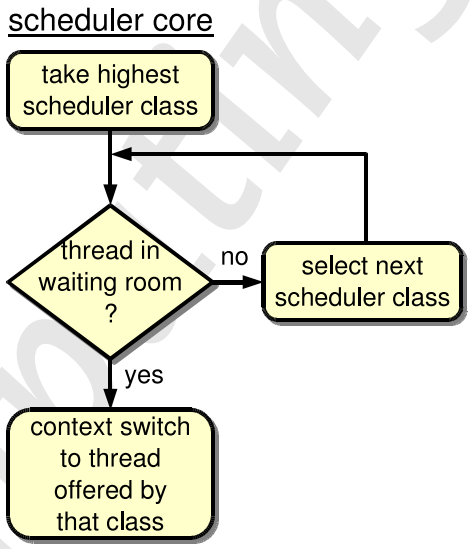
The current scheduling policy of a thread can be found in the file `/proc/`*`pid`*`/task/`*`tid`*`/sched` as the numerical value `policy` (numerical value of the scheduling policies as shown in the table above).

CFS scheduler – classes

Completely Fair Scheduler (CFS) >= kernel 2.6.23

- *scheduler classes*
  - offers possibility to implement separate algorithm per policy
  - open-ended and hierarchical

1. *deadline class* >= kernel 3.14
   policy SCHED_DEADLINE
2. *realtime class*
   policy SCHED_FIFO and SCHED_RR
3. *fair class*
   policy SCHED_NORMAL, SCHED_BATCH and SCHED_IDLE
4. *idle class*
   provides idle thread for CPU as final “fall back”



Notes

Figure 6

Student notes

From this page onwards, the mentioned CFS scheduler classes are discussed in detail....