LINUX KERNEL INTERNALS

THE LINUX KERNEL CPU SCHEDULER

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license**.
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

2000-2020 Kaiwan N Billimoria
kaiwanTECH, Bangalore, India.

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

Table of Contents

Good introductory notes on the kernel scheduler:
openSUSE: Tuning the Task Scheduler.

<<
Instructor Note:
Briefly describe the Linux kernel process state machine.
>>

Linux Scheduler Visualization

I With perf

SIDEBAR :: A “visual” map of scheduling

The very powerful and useful Linux ***perf*** – performance monitoring and analysis – tool lets us “see” a map of processes (threads) as they run on various processor cores. Below is some sample output from running the 'perf sched map' command from a previously grabbed

```
perf sched record <command>
```

command. (Tip: to see complete system, leave out the 'command' argument).

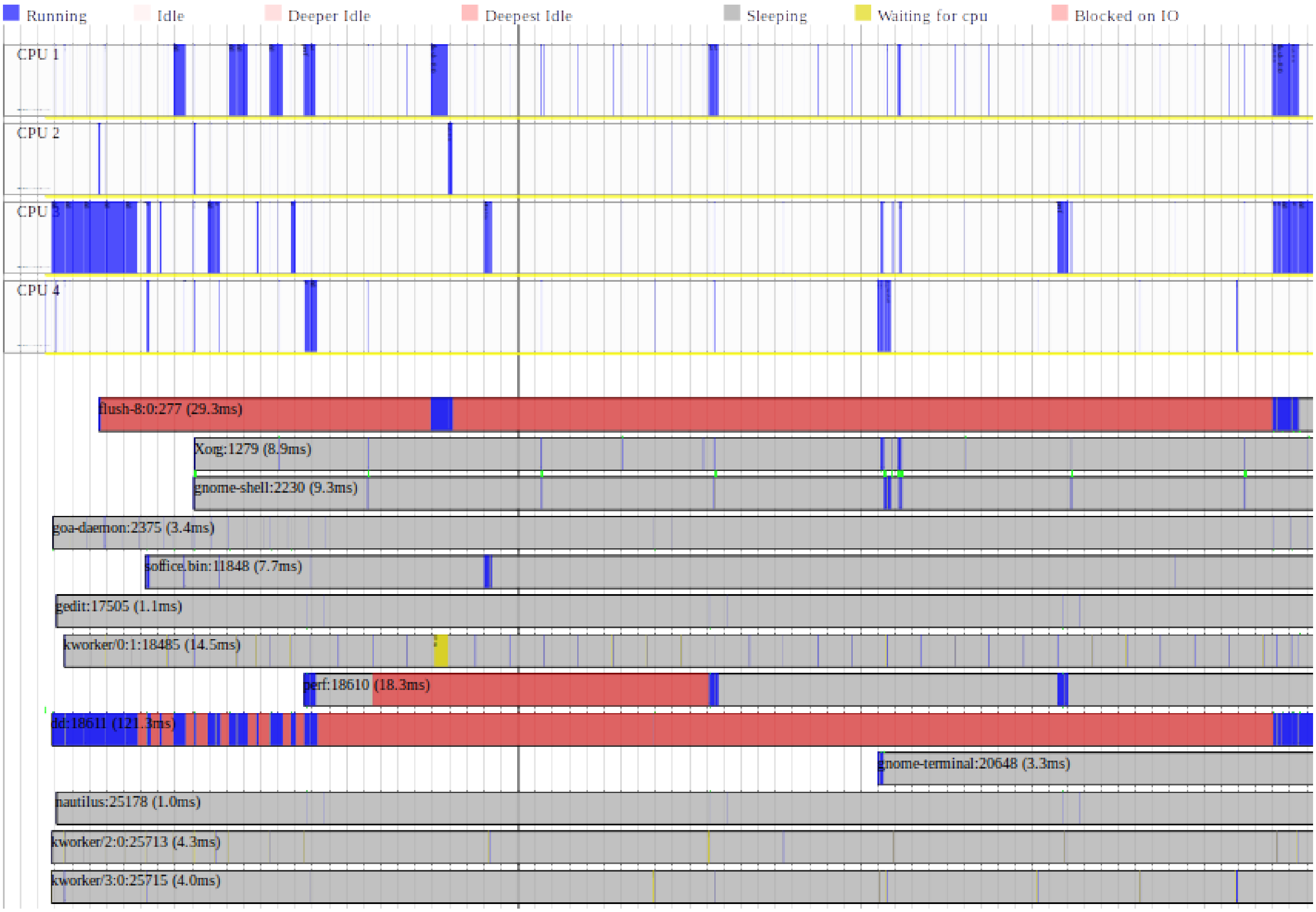
Columns stand for individual CPUs, and the two-letter shortcuts stand for tasks that are running on a CPU. A '*' denotes the CPU that had the event, and a dot signals an idle CPU.

```
<< Below is a sample run on a quad-core laptop:
perf sched map
    CPU core#            Timestamp          Legend
  0   1    2   3
>>

              *A0       97848.724561 secs A0 => perf:18611
              *B0       97848.724690 secs B0 => migration/3:25655
          *A0  B0       97848.724789 secs
  *.    .   A0  B0       97848.724791 secs C0 => swapper:0
   .       A0 *.       97848.724793 secs
  *D0      A0  .       97848.724798 secs D0 => ksoftirqd/0:3
  *.       A0  .       97848.724800 secs
  *D0      A0  .       97848.724803 secs
  *.       A0  .       97848.724805 secs
  *D0      A0  .       97848.724807 secs
  *.       A0  .        97848.724809 secs
   .       A0 *E0       97848.725292 secs E0 => ksoftirqd/3:25657
   .       A0 *.        97848.725309 secs

--snip--

   .   .  *A1  .        97849.464359 secs
  *X0  .   A1  .        97849.464359 secs
   X0  .  *.   .        97849.464360 secs
   X0  .  *A1  .        97849.464363 secs
   X0  .  *.   .        97849.464364 secs

--snip--
```

Want it more visual? No problem.
The 'perf timechart' command interprets the perf data file (output by the previous 'perf sched record <command>' command and renders the visual representation as an SVG image file.

[P. T. O.]

A sample:

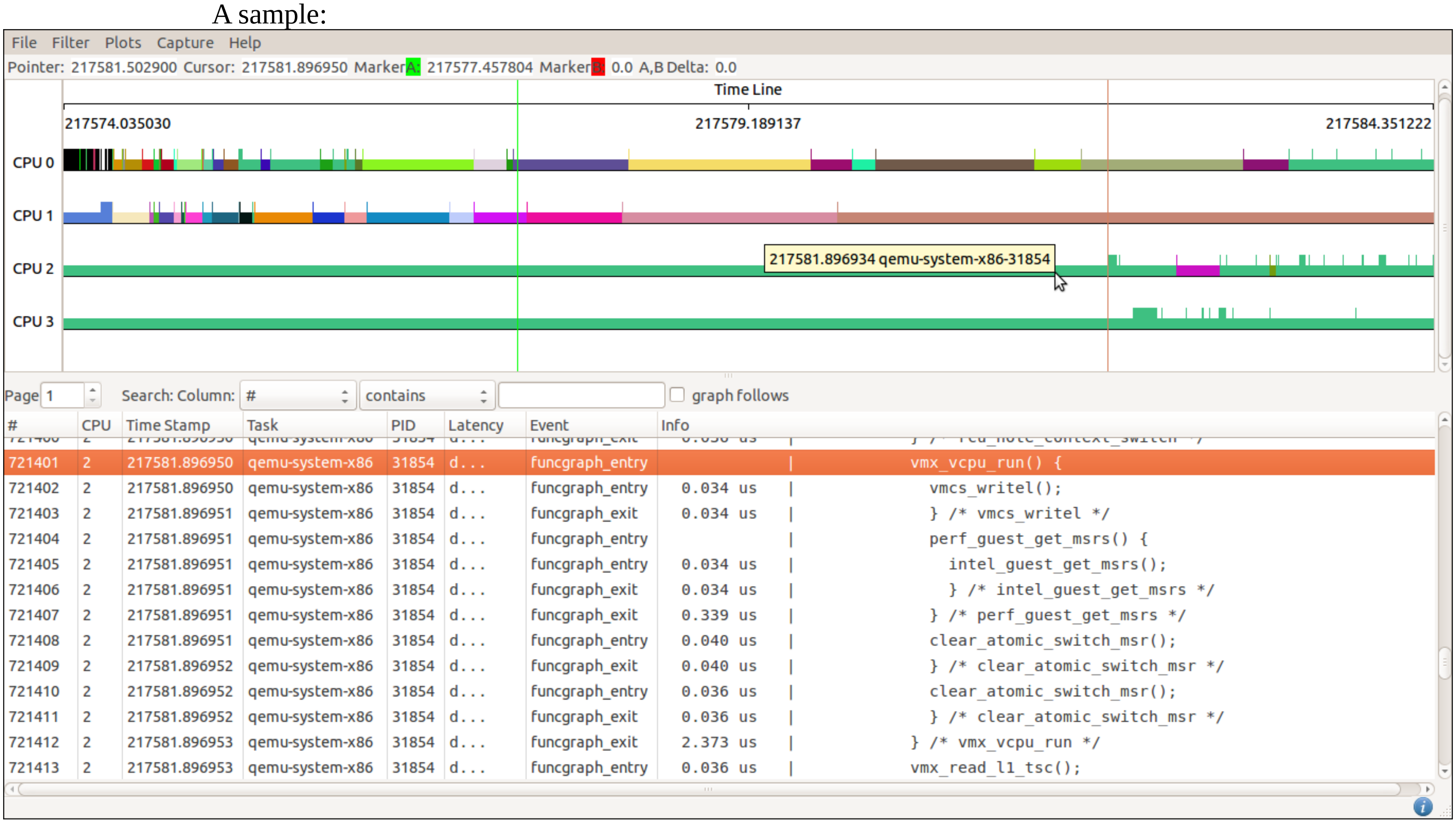


<< See the next method too.. >>

II With ftrace and KernelShark

Steps:
- Use the kernel ftrace (raw) or the trace-cmd front-end to collect data samples
- Visualization with the KernelShark GUI.

A sample:



Pointer: 217581.502900 Cursor: 217581.896950 MarkerA: 217577.457804 MarkerB: 0.0 A,B Delta: 0.0

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
721400	2	217581.896950	qemu-system-x86	31854	d...	funcgraph_exit	0.030 us | } /* rcu_note_context_switch */
721401	2	217581.896950	qemu-system-x86	31854	d...	funcgraph_entry	| vmx_vcpu_run() {
721402	2	217581.896950	qemu-system-x86	31854	d...	funcgraph_entry	0.034 us | vmcs_writel();
721403	2	217581.896951	qemu-system-x86	31854	d...	funcgraph_exit	0.034 us | } /* vmcs_writel */
721404	2	217581.896951	qemu-system-x86	31854	d...	funcgraph_entry	| perf_guest_get_msrs() {
721405	2	217581.896951	qemu-system-x86	31854	d...	funcgraph_entry	0.034 us | intel_guest_get_msrs();
721406	2	217581.896951	qemu-system-x86	31854	d...	funcgraph_exit	0.034 us | } /* intel_guest_get_msrs */
721407	2	217581.896951	qemu-system-x86	31854	d...	funcgraph_exit	0.339 us | } /* perf_guest_get_msrs */
721408	2	217581.896951	qemu-system-x86	31854	d...	funcgraph_entry	0.040 us | clear_atomic_switch_msr();
721409	2	217581.896952	qemu-system-x86	31854	d...	funcgraph_exit	0.040 us | } /* clear_atomic_switch_msr */
721410	2	217581.896952	qemu-system-x86	31854	d...	funcgraph_entry	0.036 us | clear_atomic_switch_msr();
721411	2	217581.896952	qemu-system-x86	31854	d...	funcgraph_exit	0.036 us | } /* clear_atomic_switch_msr */
721412	2	217581.896953	qemu-system-x86	31854	d...	funcgraph_exit	2.373 us | } /* vmx_vcpu_run */
721413	2	217581.896953	qemu-system-x86	31854	d...	funcgraph_entry	0.036 us | vmx_read_l1_tsc();

FYI: Linux Scheduler Visualization on YouTube

LTTng.

Linux and Real-Time

Firstly, the Linux OS as originally designed and implemented, is *not* a "hard real-time" OS, as in an RTOS (Real-Time-OS; more follows on this below: see the sidebar on the PREEMPT-RT patch).

Vanilla-Linux as such is a GPOS (a General-Purpose-OS, like Windows, Mac, Unix'es, etc).

Soft Realtime Capabilities

However, the (vanilla) Linux OS's performance is easily high enough to have it qualify as an eminently capable *soft real-time (SRT)* system. This basically means that it cannot guarantee to meet every single deadline; rather, it performs on a *best-effort* basis. SRT is exactly the kind of system required for many applications in the real world, many being within the consumer electronics domain.

How does an application developer leverage this capability?
The POSIX standard specifies three scheduling policies, one of which is the "usual" or normal policy and is always the default. The other two are (soft) realtime scheduling policies. They are:

- **SCHED_NORMAL (or SCHED_OTHER)** ← ***Default scheduling policy***
- **SCHED_RR**
- **SCHED_FIFO**

- SCHED_RR and SCHED_FIFO are the realtime policies.
- SCHED_NORMAL is the non-realtime policy and is always the default.
- It's important to understand that, on the Linux OS, the *kernel schedulable entity (KSE) is a thread*. Thus scheduling policy can be set programatically on a *per thread* basis.
- A task (KSE) requires superuser privileges to set the scheduler policy (< 2.6.12). Actually, from 2.6.12, subject to some constraints, an unprivileged process can also modify scheduling policies and priorities via a non-standard resource called RLIMIT_RTPRIO. If it's non-zero, the process can make changes to *it's own* realtime scheduling policy and priority.

Linux's so-called Real-Time scheduling policies actually provides soft real-time capability*.
This is often called '***Fixed Priority Preemptive Scheduling***' *: the highest priority ready-to-run task will be the task that is run.*

** From the 2.6.18 kernel onward, a set of patches for providing true Hard RT capability on Linux exists; in effect, we can use Linux as an RTOS. The patch is routinely called preempt-rt.*

Scheduling Policy Behaviour

SCHED_FIFO:

A running SCHED_FIFO task can only be preempted under the following three conditions:

- It (in)voluntarily yields the processor (technically, it moves out from the RUNNING/RUNNABLE state); this happens when a task issues **a blocking call** or invokes a system call like *sched_yield(2)*
- It dies or stops
- A higher priority realtime task becomes runnable.

In the first case above, the task is then placed at the end of it's runqueue (for that priority level). In the third case, the task remains at the head of the runqueue for it's priority level.

SCHED_RR behaviour is identical to that of SCHED_FIFO above *except that:*

- It has a timeslice, and will thus (also) be preempted when it's timeslice expires
- When preempted, the task is moved to the tail of the runqueue for it's priority level, ensuring that all SCHED_RR tasks at the same priority level are executed in turn.

<< FYI: 3.9 kernel: sched: Add a tuning knob to allow changing SCHED_RR timeslice (commit)
: /proc/sys/kernel/sched_rr_timeslice_ms
>>

SCHED_OTHER / SCHED_NORMAL :

All threads run with this policy by default. It is a decidedly non-realtime policy, the emphasis being on “fairness and overall throughput”. It's implementation from kernel ver 2.6.0 upto 2.6.22 was via the so called “O(1) scheduler”; from 2.6.23 onward, it's implemented via the scheduling class called CFS. (Discussed in detail later).

Priority Range

The chrt utility can display the scheduling class priority range.

```
# chrt -m
SCHED_OTHER min/max priority       : 0/0
SCHED_FIFO min/max priority : 1/99
SCHED_RR min/max priority   : 1/99
SCHED_BATCH min/max priority       : 0/0
SCHED_IDLE min/max priority : 0/0
#
```

From *include/linux/sched/rt.h*

```
…
/*
 * Priority of a process goes from 0..MAX_PRIO-1, valid RT
 * priority is 0..MAX_RT_PRIO-1, and SCHED_NORMAL/SCHED_BATCH
 * tasks are in the range MAX_RT_PRIO..MAX_PRIO-1. Priority
 * values are inverted: lower p->prio value means higher priority.
 *
 * The MAX_USER_RT_PRIO value allows the actual maximum
 * RT priority to be separate from the value exported to
 * user-space.  This allows kernel threads to set their
 * priority to a value higher than any user task. Note:
 * MAX_RT_PRIO must not be smaller than MAX_USER_RT_PRIO.
 */

#define MAX_USER_RT_PRIO    100
#define MAX_RT_PRIO         MAX_USER_RT_PRIO

#define MAX_PRIO        (MAX_RT_PRIO + 40)
#define DEFAULT_PRIO    (MAX_RT_PRIO + 20)
…
```

[P. T. O.]

Hence, we can represent the priority scale – as seen in the kernel (inverted!) – as:

```
[---------------...--------------------------|-----|-----]
1                                            99   120   140
^                                            ^^    ^     ^
A                                            BC    D     E
[----------- RT Prio Range-------------------][- Non-RT -]
                                               (nice val)
BEST                                                      WORST
```

```
Legend-
A = RT (real-time) Max priority (=  1)   ← RT best
B = RT Min priority             (= 99)   ← RT worst
C = Non-RT Min priority         (=100)   ← Non-RT best     [ nice -20]
D = Non-RT default priority     (=120)   ← Non-RT default  [ nice   0]
E = Non-RT Max priority         (=140)   ← Non-RT worst    [ nice +20]
```

Also, realize that:
RT (real-time) = SCHED_FIFO | SCHED_RR
Non-RT = SCHED_[OTHER|NORMAL] | SCHED_BATCH | SCHED_IDLE

And represent the priority scale – as seen (maps) in the user-mode – as:

```
[---------------...--------------------------|-----|-----]
1                                            99   120   140
^                                            ^^    ^     ^
A                                            BC    D     E
[----------- RT Prio Range-------------------][- Non-RT -]
                                               (nice val)
```

```
Legend-
A = RT (real-time) Min priority (=  1)  ← RT worst
B = RT Max priority             (= 99)  ← RT best
C = Non-RT Max priority         (=100)  ← Non-RT worst    [ nice +20]
D = Non-RT default priority     (=120)  ← Non-RT default  [ nice   0]
E = Non-RT Min priority         (=140)  ← Non-RT best     [ nice -20]
```

<<
The /proc/sched_debug output clearly shows tasks on each core's runqueue with their priorities.
>>

Note on the RT Runqueue Data Structure

The RT scheduler runqueue data structure is a priority queue: essentially an array of linked lists, managed in priority order. Each array “slot” or index represents a real-time priority level and has a pointer to a linked list of SCHED_FIFO and/or SCHED_RR runnable tasks. It is managed such that index 0 is the highest priority queue and 99 the lowest.

Using a bitmap (very much like the earlier 2.6.0-2.6.22 O(1) kernel scheduler algorithm), the first runnable thread at the lowest position in the queues can be very efficiently determined; in fact, the algorithmic time-complexity is O(1), which is deterministic and crucial for real-time characteristics.

In *kernel/sched/sched.h*

```
...
/*
 * This is the priority-queue data structure of the RT scheduling class:
 */
struct rt_prio_array {
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
    struct list_head queue[MAX_RT_PRIO];
};
<< Recollect that MAX_RT_PRIO = 100 >>
…
```

Example Application: a multithreaded application with three threads

Lets write a small multithreaded application that will clearly demonstrate the use and behaviour of the scheduling policies.

Our application (call it sched_pthread), will have a total of three threads; it will spawn two new threads, in addition to main() (considered the first thread):

```
Thread 1 (main(), really):
       Runs as SCHED_NORMAL (or SCHED_OTHER). It:

       Queries the priority range of SCHED_FIFO, printing out the values.
       Creates two threads (Thread 2 and Thread 3 below); they will
       automatically inherit the scheduling policy and priority of main.
       Prints the character “m” to the terminal in a loop.
       Terminates.


Thread 2:
       Sleeps for 2 seconds.
       Changes it's scheduling policy to SCHED_FIFO, setting it's real-

     time priority to the value passed on the command line.
       Prints the character “2” to the terminal in a loop.
       Terminates.


Thread 3:
       Changes it's scheduling policy to SCHED_FIFO, setting it's real-

      time priority to the value passed on the command line plus 10.
       Sleeps for 4 seconds.
       Prints the character “3” to the terminal in a loop.
       Terminates.
```

Remember, all the above three threads run in parallel (pseudo-parallel on a UP system).

<< Source available on the participant CD. >>

Note: On an SMP box, the threads will still run "as usual" - all together, in paralle. To prevent this (and see the point of this application), disable all processors but one to see the true effect of this demo.

There are (at least) two ways to do this:

1. Using sysfs:

As root, write '0' to the sysfs node (file) representing the online state of a CPU you wish to disable. The example below disables CPU 1 on a dual-core system:

```
# cat /sys/devices/system/cpu/cpu1/online
1
# echo -n "0" > /sys/devices/system/cpu/cpu1/online
# cat /sys/devices/system/cpu/cpu1/online
0
#
```

2. Using the taskset utility.

The *taskset* utility lets a user set (and query) the CPU affinity mask of a process. It's usage is simple:

```
$ taskset
taskset (util-linux-ng 2.13.1)
usage: taskset [options] [mask | cpu-list] [pid | cmd [args...]]
set or get the affinity of a process

  -p, --pid                  operate on existing given pid
  -c, --cpu-list             display and specify cpus in list format
  -h, --help                 display this help
  -V, --version              output version information

The default behavior is to run a new command:
  taskset 03 sshd -b 1024
You can retrieve the mask of an existing task:
  taskset -p 700
Or set it:
  taskset -p 03 700
List format uses a comma-separated list instead of a mask:
  taskset -pc 0,3,7-11 700
Ranges in list format can take a stride argument:
  e.g. 0-31:2 is equivalent to mask 0x55555555

$
```

See the man page for details.

Eg. Using taskset below to run the sched_pthrd application with CPU mask set to 01 :

```
$ sudo taskset 01 ./sched_pthrd 5
sched_pthrd.c:main : SCHED_FIFO priority range is 1 to 99

Note: to create true RT threads, you need to run this program as
superuser
main thread (11389): now creating realtime pthread p2..
main thread (11389): now creating realtime pthread p3..
m  RT Thread p3 (LWP 11389) here in function thrd_p3
   setting sched policy to SCHED_FIFO and RT priority HIGHER to 15
in 4 seconds..
```

```
  RT Thread p2 (LWP 11389) here in function thrd_p2
   setting sched policy to SCHED_FIFO and RT priority to 5 in 2 
seconds..
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm  p2: working
2222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222  p3: working
3333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333333
333333333  p3: exiting..
2222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222222222222222222222222222222222222222222222
2222222222222222222222222  p2: exiting..
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm$
```

Using POSIX capabilities

There's a far better approach to this than using sudo; sudo gives the process *root capabilities*. This interests hackers :-)
We use the powerful ***POSIX Capabilities model*** instead! This way, the process (and threads) get _only_ the capabilities they require and nothing more. Reduces the attack surface...

The man page on *capabilities(7)* says the **CAP_SYS_NICE** is the appropriate capability to use in this circumstance:

[...]

```
 CAP_SYS_NICE
              * Raise process nice value (nice(2), setpriority(2)) and change 
the nice value for arbitrary processes;
              * set  real-time  scheduling  policies  for  calling  process,  
and  set  scheduling policies and priorities for arbitrary processes
                (sched_setscheduler(2), sched_setparam(2), sched_setattr(2));
              * set CPU affinity for arbitrary processes 
(sched_setaffinity(2));
              * set I/O scheduling class and priority for arbitrary processes 
(ioprio_set(2));
```

```
              * apply migrate_pages(2) to arbitrary processes and allow
processes to be migrated to arbitrary nodes;
              * apply move_pages(2) to arbitrary processes;
* use the MPOL_MF_MOVE_ALL flag with mbind(2) and move_pages(2).
[...]


$ sudo setcap CAP_SYS_NICE+eip ./sched_pthrd_rtprio_dbg
$ getcap ./sched_pthrd_rtprio_dbg
./sched_pthrd_rtprio_dbg = cap_sys_nice+eip

# Still have to run it on exactly one core though!!

$ taskset -c 01 ./sched_pthrd_rtprio_dbg 20
[...]
```

FYI / Optional

Kernel Threads

What about changing scheduling policy and priority on a kernel thread? Certainly, can be done. As an example, the IRQ framework now uses threaded handlers to handle interrupts. The code that sets up the threaded handler is here:

kernel/irq/manage.c:

```
static int
setup_irq_thread(struct irqaction *new, unsigned int irq, bool secondary)
{
        struct task_struct *t;
        struct sched_param param = {
                .sched_priority = MAX_USER_RT_PRIO/2, << to 50 >>
        };
...
if (!secondary) {
                t = kthread_create(irq_thread, new, "irq/%d-%s",
                                   irq, new→name);
...
sched_setscheduler_nocheck(t, SCHED_FIFO, &param);
...
```

The chrt utility

chrt(1) sets or retrieves the real-time scheduling attributes of an existing PID or runs COMMAND with the given attributes. Both policy (one of SCHED_OTHER, SCHED_FIFO, SCHED_RR, or SCHED_BATCH) and priority can be set and retrieved.

<< Tip: For an informative read on using *chrt* and in general manipulating the real-time attributes of a process (and threads within it), see the article "signaltest: Using the RT priorities" by Arnaldo Carvalho de Melo, Red Hat Inc. >>

```
$ chrt
chrt (util-linux-ng 2.13.1)
usage: chrt [options] [prio] [pid | cmd [args...]]
manipulate real-time attributes of a process
  -b, --batch                        set policy to SCHED_BATCH
  -f, --fifo                         set policy to SCHED_FIFO
  -p, --pid                          operate on existing given pid
  -m, --max                          show min and max valid priorities
  -o, --other                        set policy to SCHED_OTHER
  -r, --rr                           set policy to SCHED_RR (default)
  -h, --help                         display this help
  -v, --verbose                      display status information
  -V, --version                      output version information

You must give a priority if changing policy.

Report bugs and send patches to <rml@tech9.net>
$
```

chrt syntax is a bit non-intuitive. The '-p' switch implies (real-time / static) priority level, not pid. PID follows priority.

chrt -p <prio> <pid>

First, we need a convenient process to test upon. For this purpose, we cook up a small shell script:

```
$ cat task.sh
#!/bin/bash

if [ $# -ne 1 ]; then
      echo "Usage: $0 char-to-display"
      exit 1
fi

echo "PID: $$"
echo
```

```
while [ true ]
do
        echo -n $1
        #usleep 1000
        # do some junk, spend some time...
        for i in `seq 1 700`
        do
                tmp=$(($i+5))
        done
done

$ ./task.sh t
PID: 6878

tttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
tttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
tttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
tttttttttttttttttttttttttttttttttttttttttttttttttttttt-snip--
$ sudo /bin/bash
Password:
#

ps -C <name-of-process> -To <columns to display>

# ps -C task.sh -To pid,rtprio,cmd
  PID RTPRIO CMD
 6878      - /bin/bash ./task.sh t

# chrt -f -p 50 6878          # Make PID 6878 SCHED_FIFO pri 50
# ps -C task.sh -To pid,rtprio,cmd
  PID RTPRIO CMD
 6878     50 /bin/bash ./task.sh t
# chrt -f -p 99 6878
# ps -C task.sh -To pid,rtprio,cmd
  PID RTPRIO CMD
 6878     99 /bin/bash ./task.sh t
#
```

Show priority range

```
# chrt -m
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority     : 1/99
SCHED_RR min/max priority       : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority     : 0/0
#
```

Change policy to round-robin

```
# chrt -r -p 6878
pid 6878's current scheduling policy: SCHED_FIFO
pid 6878's current scheduling priority: 99
```

```
# chrt -r -p 10 6878                        # set prio to 10...
# ps -C task.sh -To pid,rtprio,cmd
  PID RTPRIO CMD
 6878     10 /bin/bash ./task.sh t
# chrt -p 6878                              # check values
pid 6878's current scheduling policy: SCHED_RR
pid 6878's current scheduling priority: 10
#
```

Kill the RT process.

```
# chrt -p 6878
sched_getscheduler: No such process
failed to get pid 6878's policy
#
```

What are these other two (non-POSIX) policies?

SCHED_BATCH:
Added in 2.6.16. Similar to SCHED_OTHER; the difference is that this policy causes tasks that wake up frequently to be given a downslide in priority. Meant for batch-style jobs.

SCHED_IDLE:
Added in 2.6.23. Intended to be used for tasks which must run only when no other task wants the processor (i.e. during idle time).

Hard Realtime

Real-time processing *fails* if not completed within a specified deadline relative to an event; deadlines must always be met, regardless of system load.
…

Criteria for real-time computing[edit]

A system is said to be *real-time* if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed.[5] Real-time systems, as well as their deadlines, are classified by the consequence of missing a deadline:

- *Hard* – missing a deadline is a total system failure.
- *Firm* – infrequent deadline misses are tolerable, but may degrade the system's quality of service. The usefulness of a result is zero after its deadline.

- *Soft* – the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service.

...

Examples of hard realtime systems include: mission-critical applications, many types of transport (people-movers: fly-by-wire aircraft, ABS in a car, trains, escalators, etc), military uses, space flight, medical electronics and software, factory floor systems, stock exchanges, etc.

Take *stock exchanges* as a good example of a hard realtime system; see these articles (and book)!
Million Dollar Microsecond
BSE becomes world's fastest stock exchange: Ashishkumar Chauhan , 13 Oct 2015
Stock Trades at the Speed of Light
The excellent book "Automate This: How Algorithms Came to Rule our World", by Christopher Steiner; the first chapter - "Wall Street: the first domino".

Linux firmly falls into the domain of a GPOS – General-Purpose-OS – and hence, at best, can meet *Soft Real-time* criteria (in fact, vanilla Linux is *really good* at this and thus a natural for many consumer electronics devices!).

But what about *Firm* and *Hard real-time*? Not vanilla Linux, but PREEMPT-RT Linux, yes indeed!

Linux Hard Realtime (RTOS) Capabilities

The Linux PREEMPT-RT Patch – *Now a Linux Foundation (LF) Project!*

Now called the LF RTL (RealTime Linux) project on LF.

The real time (-rt) tree, also called the CONFIG_PREEMPT_RT patch-set, implements low-latency modifications to the kernel. The patch-set, downloadable from www.kernel.org/pub/linux/kernel/projects/rt, allows most of the kernel to be preempted, partly by replacing many spinlocks with mutexes. It also incorporates high-resolution timers. Several -rt features have been integrated into the mainline kernel. You will find detailed documentation at the project's wiki hosted at http://rt.wiki.kernel.org/.

Systems Based on RT Preempt Linux

FAQs: https://rt.wiki.kernel.org/articles/f/r/e/Frequently_Asked_Questions_7407.html

Other possibly useful online resources:
A realtime preemption overview, LWN, Aug 2005, Paul McKenny
How Realtime is Linux 2.6 ?
Writing Real-Time Device Drivers for Telecom Switches, Part 1
Intro to Real-Time Linux for Embedded Developers, March 2013
List of RTOS (Wikipedia)

Intro to Real-Time Linux for Embedded Developers, (interview with Steven Rostedt, maintainer of the stable release of Linux real-time patch) March 2013

...

For those that need hard real-time, it’s usually to control something that will end up killing people if something goes wrong. Everything else pretty much does not need that classic "Hard" real-time OS.

That said, PREEMPT_RT gives you something that's very close. I will be the first to tell you that I wouldn't want the PREEMPT_RT kernel to be controlling whether or not the plane I'm flying on crashes. But it's good enough for robotics, stock exchanges, and for computers that have to interface with the "Hard" real-time software. PREEMPT_RT has been used on computers that have gone into space.

PREEMPT_RT is a "hardening" of Linux. It's far from mathematically provable, but if there's an unbounded latency that's in PREEMPT_RT, we consider that a bug, and work hard to fix it.

…

One place that PREEMPT_RT comes in handy is with musicians. They were our earliest testers. I asked one person who reported a bug to test a patch, and he asked me for some advice in applying it. I asked him if he was a computer programmer, and he replied, "No, I'm a guitarist". He told me that the RT patch was great for getting reliable recordings, as jitter from mainline Linux would cause a scratching sound. I thought it was really cool that non computer focused people had a use for our work.

...

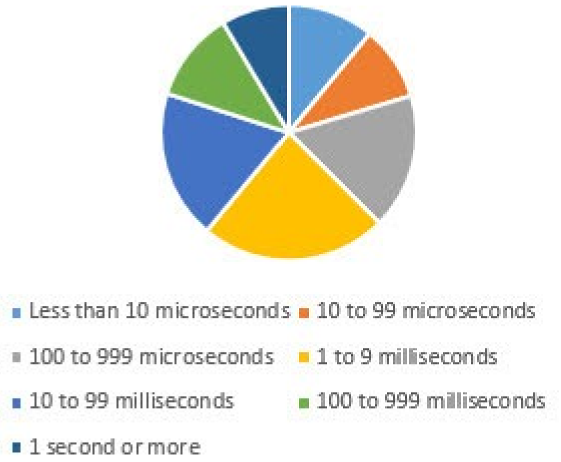
Source: The Embedded Muse 341, 03 Jan 2018, Jack Ganssle.

…

VDC Survey Results

VDC completed their survey of the embedded space and sent a copy to me. I believe the results are not generally available, but they gave me permission to cite factoids I found interesting.

Some 70% of respondents have at least some hard real-time deadlines. ***Distribution of real-time responses is:***

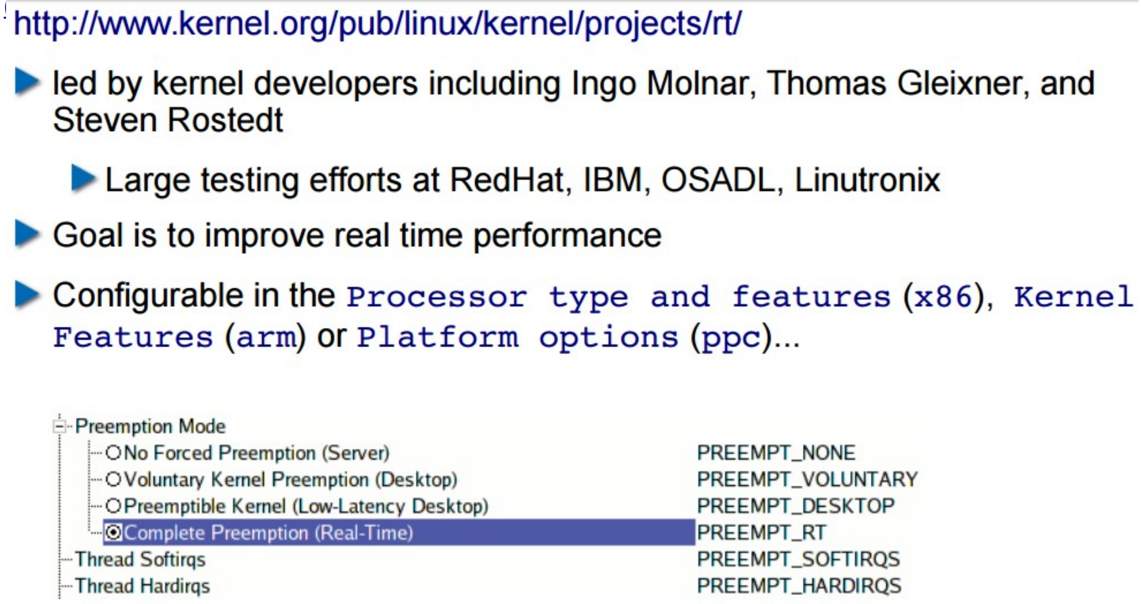


20% of us have sub-100 µs deadlines.

[...]

Additional Information

Source: RTMux: A Thin Multiplexer To Provide Hard Realtime Applications For Linux (“Making Linux do Hard Real-time”)

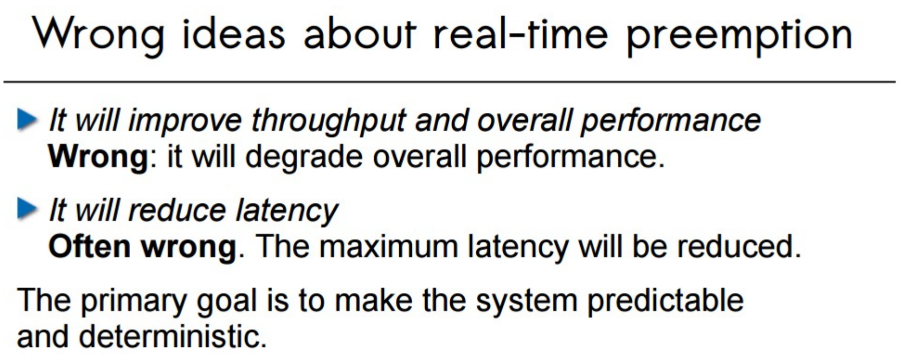


http://www.kernel.org/pub/linux/kernel/projects/rt/

- led by kernel developers including Ingo Molnar, Thomas Gleixner, and Steven Rostedt
  - Large testing efforts at RedHat, IBM, OSADL, Linutronix
- Goal is to improve real time performance
- Configurable in the `Processor type and features` (`x86`), `Kernel Features` (`arm`) or `Platform options` (`ppc`)...

```
Preemption Mode
  ○ No Forced Preemption (Server)                PREEMPT_NONE
  ○ Voluntary Kernel Preemption (Desktop)        PREEMPT_VOLUNTARY
  ○ Preemptible Kernel (Low-Latency Desktop)     PREEMPT_DESKTOP
  ◉ Complete Preemption (Real-Time)              PREEMPT_RT
Thread Softirqs                                  PREEMPT_SOFTIRQS
Thread Hardirqs                                  PREEMPT_HARDIRQS
```

...

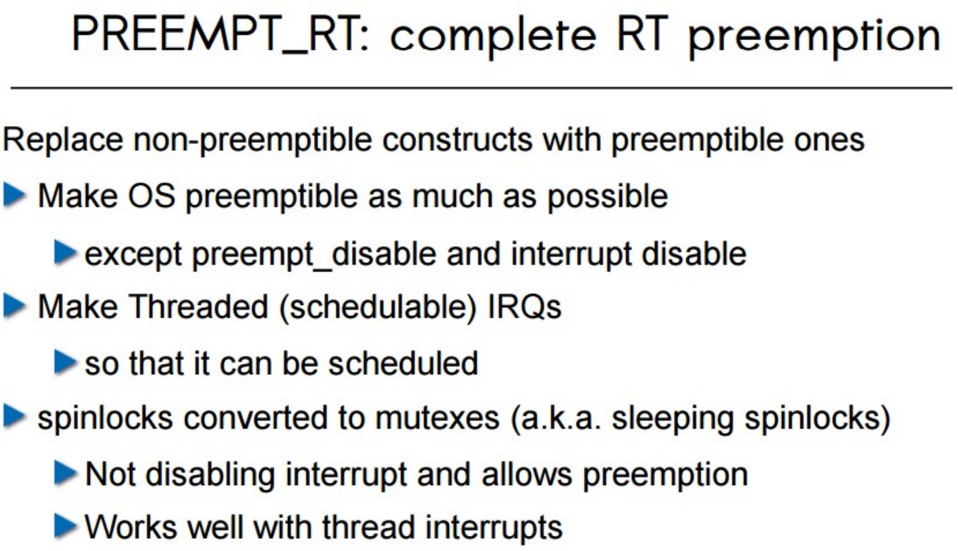


Wrong ideas about real-time preemption

- *It will improve throughput and overall performance*
  Wrong: it will degrade overall performance.
- *It will reduce latency*
  Often wrong. The maximum latency will be reduced.

The primary goal is to make the system predictable and deterministic.

…



PREEMPT_RT: complete RT preemption

Replace non-preemptible constructs with preemptible ones

- Make OS preemptible as much as possible
  - except preempt_disable and interrupt disable
- Make Threaded (schedulable) IRQs
  - so that it can be scheduled
- spinlocks converted to mutexes (a.k.a. sleeping spinlocks)
  - Not disabling interrupt and allows preemption
  - Works well with thread interrupts

Update: 05 Oct 2015:

The Linux Foundation Announces Project to Advance Real-Time Linux

By Linux_Foundation - October 5, 2015 – 8:14am

Real-Time Linux on the go, OSADL

“… OSADL is looking forward to a fruitful collaboration in the Linux Foundation RTL Working Group. We very much hope that a day will come in the foreseeable future when Linux mainline will immediately contain - without any further patching - the PREEMPT_RT configuration option. And we can only appeal to the other members of the RTL Working Group to not let Linux users wait too long. OSADL certainly will continue to go for it.”

OSADL Project: Realtime Linux

The future of the realtime patch set [LWN, Oct 21 2014]

“In a followup to last year's report on the future of realtime Linux, Thomas Gleixner once again summarized the status of the long-running patch set. The intervening year did not result in the industry stepping up to fund further work, which led Gleixner to declare that realtime Linux is now just his hobby. That means new releases will be done as his time allows and may eventually lead to dropping the patch set altogether if the widening gap between mainline and realtime grows too large.   …”

Understanding the Latest Open-Source Implementations of Real-Time Linux for Embedded Processors By: Michael Roeder, Field Applications Engineer, Future Electronics. (2 page PDF).

Other real-time Linux projects:
XENOMAI
RTAI

Deadline scheduling: coming soon? [LWN, Dec '13]

[Indeed, it has been merged into mainline kernel ver 3.14].

[FYI / OPTIONAL]

Source: LKD3: Ch 4 "The Linux Scheduling Algorithm".

The Scheduling Policy in Action

Consider a system with two runnable tasks: a text editor and a video encoder. The text editor is I/O-bound because it spends nearly all its time waiting for user key presses (no matter how fast the user types, it is not that fast). Despite this, when the text editor does receive a key press, the user expects the editor to respond immediately. Conversely, the video encoder is processor-bound. Aside from reading the raw data stream from the disk and later writing the resulting video, the encoder spends all its time applying the video codec to the raw data, easily using 100% of the processor. The video encoder does not have any strong time constraints on when it runs—if it started running now or in half a second, the user could not tell and would not care. Of course, the sooner it finishes the better, but latency is not a primary concern.

In this scenario example, ideally the scheduler gives the text editor a higher priority and larger timeslice than the video encoder receives because the text editor is interactive. This ensures that the text editor has plenty of timeslice available. Furthermore, because the text editor has a higher priority, it is capable of preempting the video encoder when needed—say, the instant the user presses a key. This guarantees that the text editor is capable of responding to user key presses immediately. This is to the detriment of the video encoder, but because the text editor runs only intermittently, when the user presses a key, the video encoder can monopolize the remaining time. This optimizes the performance of both applications.

From the 2.6.23 kernel onward (to now, which as of this writing, is the 3.x.y series kernel), Linux now has the concept of a scheduling class and multiple possible modular schedulers. The most important of these, and the default, is probably the CFS – Completely Fair Scheduler - (implemented in *kernel/sched_fair.c*).

Scheduler Classes

The Linux scheduler is modular, enabling different algorithms to schedule different types of processes. This modularity is called scheduler classes.

<<
Effectively, scheduler classes implement differing scheduling policies in a modular fashion. In other words, each "scheduling class" implements a certain scheduling policy.
>>

Scheduler classes enable different, pluggable algorithms to coexist, scheduling their own types of processes. Each scheduler class has a priority. The base scheduler code, which is defined in *kernel/sched.c* , iterates over each scheduler class in order of priority. (In practice, the scheduling classes reside on a linked list that is followed). The highest priority scheduler class that has a runnable process wins, selecting who runs next.

The Completely Fair Scheduler (CFS) is the registered scheduler class for normal processes, called SCHED_NORMAL in Linux (and SCHED_OTHER in POSIX). CFS is defined in *kernel/sched_fair.c*.

<<

The currently existing (as of 3.14) scheduler classes, in priority order, are:

Class Name	***sched_class Data Structure Name***	***Defined in***
1. Stop-sched *	**stop_sched_class**	**kernel/sched/stop_task.c**
2. Deadline [+]	**dl_sched_class**	**kernel/sched/deadline.c**
3. RT (Real-Time)	**rt_sched_class**	**kernel/sched/rt.c**
4. CFS	**fair_sched_class**	**kernel/sched/fair.c**
5. Idle	**idle_sched_class**	**kernel/sched/idle_task.c**

* The “stop-sched” class appears to be a recent addition; it is designed to preempt everything and be preempted by nothing. A “Simple, special scheduling class for the per-CPU stop task”. Apparently used by *kernel/stop_machine.c* .

[+] The deadline scheduler merged into mainline in kernel ver 3.14. Some details.
[Source] : “A new deadline scheduling policy (SCHED_DEADLINE) has been added. In order to control the scheduling of processes under this policy, two new system calls have been added: *sched_setattr()* and *sched_getattr()*. These are more generalized versions of the *sched_setscheduler()* and *sched_getscheduler()* system calls: they allow setting scheduling policy and parameters for all of the previously existing scheduling policies as well as the new SCHED_DEADLINE policy. Documentation can be found in the *sched_setattr(2)* and *sched(7)* manual pages.”

From *kernel/sched/sched.h* *<< as of kernel ver 3.14 >>*

```
...
#define sched_class_highest (&stop_sched_class)
#define for_each_class(class) \
   for (class = sched_class_highest; class; class = class->next)

extern const struct sched_class stop_sched_class;
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;
extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;
```

…

>>

CFS

The typical “old-school” approach to scheduling algorithms', is one where we manage a thread's CPU resource consumption on the basis of two driving factors: it's priority and timeslice.

However, this approach has it's problems and pathological cases.
Assigning absolute timeslices yields a constant switching rate but variable fairness. The approach taken by CFS is a radical (for process schedulers) rethinking of timeslice allotment: Do away with timeslices completely and assign each process a proportion of the processor. CFS thus yields constant fairness but a variable switching rate.

For details on it's design and working, please see the following resource(s).

***Documentation/scheduler/sched-design-CFS.txt* (extract below):**

1. OVERVIEW

CFS stands for "Completely Fair Scheduler," and is the new "desktop" process scheduler implemented by Ingo Molnar and merged in Linux 2.6.23. It is the replacement for the previous vanilla scheduler's SCHED_OTHER interactivity code.

80% of CFS's design can be summed up in a single sentence: *CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.*

"Ideal multi-tasking CPU" is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/nr_running speed. For example: if there are 2 tasks running, then it runs each at 50% physical power --- i.e., actually in parallel.

On real hardware, we can run only a single task at once, so we have to introduce the concept of "**virtual runtime**." The virtual runtime of a task specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the virtual runtime of a task is its actual runtime normalized to the total number of running tasks.

2. FEW IMPLEMENTATION DETAILS

In CFS the virtual runtime is expressed and tracked via the per-task **p->se.vruntime** (nanosec-unit) value. This way, it's possible to accurately timestamp and measure the "expected CPU time" a task should have gotten.

Implementation Details – what exactly is **vruntime**

The vruntime variable stores the virtual runtime of a process, which is the actual runtime (the amount of time spent running) normalized (or weighted) by the number of runnable processes. The virtual runtime's units are nanoseconds and therefore vruntime is decoupled from the timer tick.

Because processors are not capable of perfect multitasking and we must run each process in succession, CFS uses vruntime to account for how long a process has run and thus how much longer it ought to run.

The function *kernel/sched_fair.c:update_curr()*, manages this accounting.

```
<<

struct task_struct {
--snip--

<-- post-2.6.23 ; the CFS scheduler >

        struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
--snip--
}
...

struct sched_entity {
--snip--
          u64          vruntime;
```

<< vruntime – virtual runtime – actual runtime on the CPU weighted by the number of runnable processes. Essentially, the measure of how much time the process has run. Updated in the routine update_curr(). *This is done often: on every timer interrupt and when a process becomes runnable or blocking...*

The CFS algorithm then boils down to a simple fact: when selecting a process (during *schedule()), **pick the task with the smallest vruntime value**!*

```
>>
--snip--
};
```

CFS's task picking logic is based on this p->se.vruntime value and it is thus very simple: it always tries to run the task with the smallest p->se.vruntime value (i.e., the task which executed least so far). CFS always tries to split up CPU time between runnable tasks as close to "ideal multitasking hardware" as possible.

Most of the rest of CFS's design just falls out of this really simple concept, with a few add-on embellishments like nice levels, multiprocessing and various algorithm variants to recognize sleepers.

3. THE RBTREE

CFS's design is quite radical: it does not use the old data structures for the runqueues, but it uses a time-ordered rbtree to build a "timeline" of future task execution, and thus has no "array switch" artifacts (by which both the previous vanilla scheduler and RSDL/SD are affected).

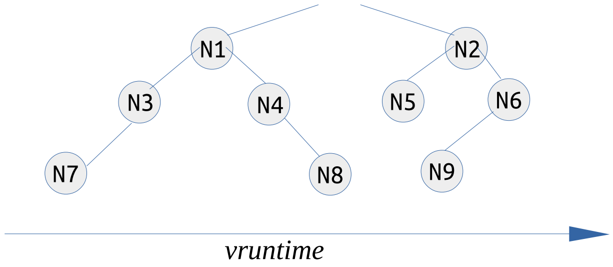
<<
An *rbtree*, or red-black tree, is a type of self-balancing binary search tree (BST). The tree holds nodes of arbitrary data identified by a specific key. Searching for a node given by a key within the rbtree is efficient – it's algorithmic time complexity is O(log n).
>>

--snip--

CFS maintains a time-ordered rbtree, where all runnable tasks are sorted by the p->se.vruntime key (there is a subtraction using rq->cfs.min_vruntime to account for possible wraparounds). CFS picks the "leftmost" task from this tree and sticks to it.

As the system progresses forwards, the executed tasks are put into the tree more and more to the right --- slowly but surely giving a chance for every task to become the "leftmost task" and thus get on the CPU within a deterministic amount of time.

<< see some implementation details below >>



Summing up, CFS works like this: it runs a task a bit, and when the task schedules (or a scheduler tick happens) the task's CPU usage is "accounted for": the (small) time it just spent using the physical CPU is added to p->se.vruntime. Once p->se.vruntime gets high enough so that another task becomes the "leftmost task" of the time-ordered rbtree it maintains (plus a small amount of "granularity" distance relative to the leftmost task so that we do not over-schedule tasks and trash the cache), then the new leftmost task is picked and the current task is preempted.

--snip--

The vruntime will obviously need to be "refreshed" - in effect, reset and recalculated – every so often (it can't be purely monotonic; it would then overflow). The value is a kernel tunable – cfs_period_us:

cpu.cfs_period_us - The default value is 100000 and refers to the time period in which the standard scheduler "accounts" the process in microseconds. It does little on its own.
[Source]

So, the default CFS period is 100000 us = 100ms.

vruntime Calculation

(*Para below based on notes from "OS Concepts", Silberschatz, Galvin, Gagne, 9th ed)*
…
The virtual run time is associated with a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks. For tasks at normal priority (nice values of 0), virtual run time is identical to actual physical run time.

Thus, if a task with default priority runs for 200 milliseconds, its vruntime will also be 200 milliseconds. However, if a lower-priority task runs for 200 milliseconds, its vruntime will be higher than 200 milliseconds. Similarly, if a higher-priority task runs for 200 milliseconds, its vruntime will be less than 200 milliseconds.
...

(Para below based on notes from "Professional Linux Kernel Architecture", Maurerer)

- Most of the work is done in *kernel/sched_fair.c:__update_curr()*
- Called on timer tick
- Updates the physical and virtual time 'current' has just spent on the processor
    - For tasks that run at default priority, i.e., *nice* value 0, the physical and virtual time spent is identical
    - Not so for tasks at other priority (nice) levels; thus the calculation of vruntime is affected by the priority of current using a *load weight* factor

```
delta_exec = (unsigned long)(now - curr->exec_start);
...
delta_exec_weighted = calc_delta_fair(delta_exec, curr);
curr->vruntime += delta_exec_weighted;
```

Neglecting some rounding and overflow checking, what *calc_delta_fair* does is to compute the value given by the following formula:

*delta_exec_weighed = delta_exec * (NICE_0_LOAD / curr->load.weight)*

The thing is, more important tasks (those with a lower nice value) will have *larger* weights; thus, by the above equations, the vruntime accounted to them will be *smaller* (thus having them enqueued more to the left on the rbtree!).

CFS Runqueue (rbtree) Updation

Elements with a smaller key (the *key* is the value based on which nodes are enqueued into the rbtree) will be placed more to the left, and thus be scheduled more quickly. This way, the kernel implements two antagonistic mechanisms:

1. When a process is running, its vruntime will steadily increase, so it will finally move rightward in the red-black tree. Because vruntime will increase more slowly for more important processes, they will also move rightward more slowly, so their chance to be scheduled is bigger than for a less important process — just as required.

2. If a process sleeps, its vruntime will remain unchanged. Because the per-queue min_vruntime increases in the meantime (recall that it is monotonic!), the sleeper will be placed more to the left after waking up because the key got smaller.

In practice, both effects naturally happen simultaneously, but this does not influence the interpretation.

Trying CFS: a simple practical experiment

The *task.sh* shell script used below is simple - it prints the character provided as an argument in an infinite loop.

So, lets run several of these (cpu-bound) tasks - via the wrapper shell script *rq.sh*, which in turn invokes the *show_runnable.sh* shell script (below):

Note:
The output depends upon the CONFIG_SCHED_DEBUG kernel configuration option turned On.

```
$ cat rq.sh
#!/bin/sh
trap 'pkill t1; pkill t2; pkill t3' INT QUIT EXIT

[ $# -ne 1 ] && {
  echo "Usage: $0 0|1
 0 => show Only our three runnable/running processes t1, t2, t3
 1 => show system-wide runnable/running processes "
  exit 1
}

taskset -c 02 ./t1 a > /dev/null &
taskset -c 02 ./t2 a > /dev/null &
taskset -c 02 ./t3 a > /dev/null &

HDR="
            task   PID         tree-key  switches  prio     exec-runtime         sum-exec
sum-sleep       Task Group
------------------------------------------------------------------------------------------------
-------------------------------------"
echo "${HDR}"
[ $1 -eq 0 ] && ./show_rq.sh |grep 't[1-3]' || [ $1 -eq 1 ] && ./show_rq.sh
$
$ cat show_rq.sh
#!/bin/bash
DELAYSEC=.25
while [ true ]
do
        egrep '^R ' /proc/sched_debug # show only runnable/running
        sleep $DELAYSEC
done
$

$ ./rq.sh
Usage: ./rq.sh 0|1
 0 => show Only our three runnable/running processes t1, t2, t3
 1 => show system-wide runnable/running processes
$ ./rq.sh 0

            task   PID         tree-key  switches  prio     exec-runtime         sum-exec
sum-sleep       Task Group
------------------------------------------------------------------------------------------------
-------------------------------------
R             t1 26413   3475532.352651       112   120   3475532.352651        20.687059
115.716940 /user/1000.user/c2.session
R             t2 26414   3477807.106308      1545   120   3477807.106308       268.955042
1140.697697 /user/1000.user/c2.session
R             t3 26415   3478713.069441      2120   120   3478713.069441       364.976408
1550.692176 /user/1000.user/c2.session
R             t1 26413   3479170.849802      2400   120   3479170.849802       415.478226
1759.018635 /user/1000.user/c2.session
R             t3 26415   3480073.009576      2976   120   3480073.009576       513.500122
2165.578822 /user/1000.user/c2.session
R             t2 26414   3480521.064251      3262   120   3480521.064251       566.501957
2366.334153 /user/1000.user/c2.session
^C$
$ ./rq.sh 1

            task   PID         tree-key  switches  prio     exec-runtime         sum-exec
sum-sleep       Task Group
------------------------------------------------------------------------------------------------
-------------------------------------
R    ibus-daemon 24041   3145825.132149    412387   120   3145825.132149     65954.854606
82152509.875089 /user/1000.user/c2.session
R          gdbus 24172   3145824.641853    125964   120   3145824.641853     19580.511122
82203266.143444 /user/1000.user/c2.session
R             t2 30406   3481898.900168         6   120   3481898.900168         3.316237
8.550844 /user/1000.user/c2.session
R          egrep 30410   2394206.619548         5   120   2394206.619548         5.536256
0.000000 /user/1000.user/c2.session
R          egrep 30690   2526207.836664         1   120   2526207.836664         0.546226
0.000000 /user/1000.user/c2.session
R            seq 30691   3482356.604352         0   120   3482356.604352         0.000000
0.000000 /user/1000.user/c2.session
R          egrep 30971   2526219.924814         1   120   2526219.924814         0.092399
0.000000 /user/1000.user/c2.session
R             t1 30405   3482798.353227       573   120   3482798.353227       101.270099
415.536354 /user/1000.user/c2.session
...
$
```

The actual CFS RQ (in effect the nodes on the cfs rbtree) are seen under the title 'runnable tasks' for each core.
The 'tree-key' value is actually the se->vruntime for that task!
So, we can in effect visualize the tree: smallest 'tree-key' implies leftmost node, next smallest 'tree-key' impiles the node on the right of the leftmost node, and so on...

The code that drives this output seen above (i.e. output of the /proc/sched_debug file) is here:

```
kernel/sched_debug.c:sched_debug_show() -->
       for_each_online_cpu(cpu)
```

```
        print_cpu(m, cpu);
              print_cpu() → ... → print_rq() → print_task()
```

<<
There must exist a relationship between the scheduler-related Control Group CONFIG_FAIR_SCHED_GROUP and the CFS algorithm: indeed there is. FYI, the code corresponding to this is shown in Appendix C.
>>

Scheduling Latency

The kernel guarantees that every runnable task will run at least once in a given time interval. This interval is called the *scheduling latency*.

The scheduling latency is a system configurable parameter: sched_latency_ns

Eg. on a recent 3.2 kernel Ubuntu 12.04 x86 desktop system:

```
# cat /proc/sys/kernel/sched_latency_ns
18000000
<< i.e 18 ms >>
# cat /proc/sys/kernel/sched_min_granularity_ns
2250000
<< i.e  2.25 ms >>
#
```

Thus, the *scheduling latency* here is 18ms; implying that every runnable task will get a chance to run at least once within an 18ms interval.

However, if the number of runnable tasks at a given point in time grows large, then the situation becomes untenable - we will have too little a time slice to allocate within the latency period. Hence, if this is indeed the case, the scheduling latency (or period) is dynamically increased.

<< Below, FYI / OPTIONAL.
Source: The openSUSE documentation article “Tuning the Task Scheduler” >>

sched_latency_ns

The scheduling period (latency, or targeted preemption latency) for CPU bound tasks. Increasing this variable increases a CPU bound task's timeslice.

```
# cat /proc/sys/kernel/sched_latency_ns
18000000
```

```
#                       << i.e. it's 18 ms; on a recent 3.2 Ubuntu 12.04, x86
desktop >>
```

A task's timeslice is its weighted fair share of the scheduling period:

timeslice = scheduling period * (task's weight/total weight of tasks in the run queue)

The task's weight depends on the task's nice level and the scheduling policy. Minimum task weight for a SCHED_OTHER task is 15, corresponding to nice 19. The maximum task weight is 88761, corresponding to nice -20.

Timeslices become smaller as the load increases. When the number of runnable tasks exceeds sched_latency_ns/sched_min_granularity_ns, (works out to ~ 9 for the above numbers), the slice becomes
number_of_running_tasks * sched_min_granularity_ns.

Prior to that, the (minimum) slice is equal to sched_latency_ns.

```
# cat /proc/sys/kernel/sched_min_granularity_ns
2250000
#          << i.e. it's 2.25 ms; on a recent 3.2 Ubuntu 12.04, x86 desktop >>
```

…

[Other scheduler configuration parameters follow...]

sched_wakeup_granularity_ns

The wake-up preemption granularity. Increasing this variable reduces wake-up preemption, reducing disturbance of compute bound tasks. Lowering it improves wake-up latency and throughput for latency critical tasks, particularly when a short duty cycle load component must compete with CPU bound components. The default value is 5000000 (ns) (= 5ms).

```
# cat /proc/sys/kernel/sched_wakeup_granularity_ns
3000000
#              << i.e. it's 3 ms; on a recent 3.2 Ubuntu 12.04, x86 desktop >>
```

WARNING!
Settings larger than half of sched_latency_ns will result in zero wake-up preemption and short duty cycle tasks will be unable to compete with CPU hogs effectively.

Throttling

Realtime SCHED_FIFO Throttling

sched_rt_period_us

Period over which real-time task bandwidth enforcement is measured. The default value is 1000000 (µs) (= 1s).

```
# cat /proc/sys/kernel/sched_rt_period_us
1000000
#
```

sched_rt_runtime_us

Quantum allocated to real-time tasks during sched_rt_period_us. Setting to -1 disables RT bandwidth enforcement. By default, RT tasks may consume 95% CPU/sec, thus leaving 5% CPU/sec or 0.05s to be used by SCHED_OTHER tasks.

```
# cat /proc/sys/kernel/sched_rt_runtime_us
950000
#
```

From the article "SCHED_FIFO and realtime throttling"
…
Kernels shipped since 2.6.25 have set the *rt_bandwidth* value for the default group to be 0.95 out of every 1.0 seconds. In other words, the group scheduler is configured, by default, to reserve 5% of the CPU for non-SCHED_FIFO tasks.
…
So, it is argued, the rt_bandwidth limit is an important safety breaker. With it in place, even a runaway SCHED_FIFO cannot prevent the administrator from (eventually) regaining control of the system and figuring out what is going on. In exchange for this safety, this feature only robs SCHED_FIFO tasks of a small amount of CPU time - the equivalent of running the application on a slightly weaker processor.
..."

CFS Bandwidth Throttling

Details:
"CPU bandwidth control for CFS"
http://lxr.free-electrons.com/source/Documentation/scheduler/sched-bwc.txt

[OPTIONAL/FYI] *<< Below, source: “Professional Linux Kernel Architecture”, Maurerer >>*

__sched_period determines the length of the latency period, which is usually just sysctl_sched_latency (nothing but sched_latency_ns), but is extended linearly if more processes are running. In this case, the period length is

sysctl_sched_latency × (nr_running / sched_nr_latency)

Distribution of the time among active processes in one latency period is performed by considering the relative weights of the respective tasks. The slice length (in effect, timeslice) for a given process as represented by a schedulable entity is computed by the function sched_slice().

SCHEDULING CLASSES

The new CFS scheduler has been designed in such a way to introduce "Scheduling Classes," an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming too much about them.

sched_fair.c implements the CFS scheduler described above.

sched_rt.c implements SCHED_FIFO and SCHED_RR semantics, in a simpler way than the previous vanilla scheduler did. It uses 100 runqueues (for all 100 RT priority levels, instead of 140 in the previous scheduler) and it needs no expired array.

Scheduling classes are implemented through the sched_class structure, which contains hooks to functions that must be called whenever an interesting event occurs.

--snip--

Implementation Details:

In *<linux/sched.h>* :

```
...
struct sched_class {
    const struct sched_class *next;

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int
flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int
flags);
    void (*yield_task) (struct rq *rq);
    bool (*yield_to_task) (struct rq *rq, struct task_struct *p,
bool preempt);
```

```
    void (*check_preempt_curr) (struct rq *rq, struct task_struct
*p, int flags);

    struct task_struct * (*pick_next_task) (struct rq *rq);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
    int  (*select_task_rq)(struct task_struct *p, int sd_flag, int
flags);

    void (*pre_schedule) (struct rq *this_rq, struct task_struct
*task);
    void (*post_schedule) (struct rq *this_rq);
    void (*task_waking) (struct task_struct *task);
    void (*task_woken) (struct rq *this_rq, struct task_struct
*task);

    void (*set_cpus_allowed)(struct task_struct *p,
                 const struct cpumask *newmask);

    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);
#endif

    void (*set_curr_task) (struct rq *rq);
    void (*task_tick) (struct rq *rq, struct task_struct *p, int
queued);
    void (*task_fork) (struct task_struct *p);

    void (*switched_from) (struct rq *this_rq, struct task_struct
*task);
    void (*switched_to) (struct rq *this_rq, struct task_struct
*task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct
*task, int oldprio);

    unsigned int (*get_rr_interval) (struct rq *rq,
                     struct task_struct *task);

#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*task_move_group) (struct task_struct *p, int on_rq);
#endif
};
```

…

<<
Source

…

Scheduling classes are implemented through the sched_class structure, which contains hooks to functions that must be called whenever an interesting event occurs.

This is the (partial) list of the hooks:

- enqueue_task(...)

  Called when a task enters a runnable state.
  It puts the scheduling entity (task) into the red-black tree and increments the nr_running variable.

- dequeue_task(...)

  When a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree. It decrements the nr_running variable.

- yield_task(...)

  This function is basically just a dequeue followed by an enqueue, unless the compat_yield sysctl is turned on; in that case, it places the scheduling entity at the right-most end of the red-black tree.

- check_preempt_curr(...)

  This function checks if a task that entered the runnable state should preempt the currently running task.

- pick_next_task(...)

  This function chooses the most appropriate task eligible to run next.

- set_curr_task(...)

  This function is called when a task changes its scheduling class or changes its task group.

- task_tick(...)

  This function is mostly called from time tick functions; it might lead to process switch. This drives the running preemption.

…

>>

Thus, for CFS, in *kernel/sched_fair.c* :

```
...
/*
 * All the scheduling class methods:
 */
static const struct sched_class fair_sched_class = {
    .next               = &idle_sched_class,
    .enqueue_task       = enqueue_task_fair,
    .dequeue_task       = dequeue_task_fair,
    .yield_task         = yield_task_fair,
    .yield_to_task      = yield_to_task_fair,

    .check_preempt_curr = check_preempt_wakeup,

    .pick_next_task     = pick_next_task_fair,
    .put_prev_task      = put_prev_task_fair,

#ifdef CONFIG_SMP
    .select_task_rq     = select_task_rq_fair,

    .rq_online          = rq_online_fair,
    .rq_offline         = rq_offline_fair,

    .task_waking        = task_waking_fair,
#endif

    .set_curr_task      = set_curr_task_fair,
    .task_tick          = task_tick_fair,
    .task_fork          = task_fork_fair,

    .prio_changed       = prio_changed_fair,
    .switched_from      = switched_from_fair,
    .switched_to        = switched_to_fair,

    .get_rr_interval    = get_rr_interval_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
    .task_move_group    = task_move_group_fair,
#endif
};
```

FYI: Queuing a task onto the CFS rbtree is achieved by (call graph):

enqueue_task_fair → enqueue_entity → __enqueue_entity

The __enqueue_entity function sets 'cfs_rq->rb_leftmost' to the “left-most” node in the rbtree, thus caching it.

Implementation Details – Picking the Next Task

Let's start with the assumption that we have a red-black tree populated with every runnable process in the system where the key for each node is the runnable process's virtual runtime. We'll look at how we build that tree in a moment, but for now let's assume we have it.

Given this tree, the process that CFS wants to run next, which is the process with the smallest vruntime, is the leftmost node in the tree. That is, if you follow the tree from the root down through the left child, and continue moving to the left until you reach a leaf node, you find the process with the smallest vruntime. (Again, if you are unfamiliar with binary search trees, don't worry. Just know that this process is efficient.)

CFS's process selection algorithm is thus summed up as "run the process represented by the leftmost node in the rbtree." The function that performs this selection is *__pick_next_entity()*, defined in *kernel/sched_fair.c*:

```
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
        struct rb_node *left = cfs_rq->rb_leftmost;

        if (!left)
                return NULL;

        return rb_entry(left, struct sched_entity, run_node);
}
```

Note that __pick_next_entity() does not actually traverse the tree to find the leftmost node, because the value is cached by rb_leftmost. Although it is efficient to walk the tree to find the leftmost node—O(height of tree), which is O(log N) for N nodes if the tree is balanced—it is even easier to cache the leftmost node.

The return value from this function is the process that CFS next runs. If the function returns NULL, there is no leftmost node, and thus no nodes in the tree. In that case, there are no runnable processes, and the kernel schedules the idle task.

schedule() - the Scheduler entry point

The main entry point into the process schedule is the function schedule(), defined in *kernel/sched.c*. This is the function that the rest of the kernel uses to invoke the process scheduler, deciding which process to run and then running it. schedule() is generic with respect to scheduler classes.That is, it finds the highest priority scheduler class with a runnable process *and asks it* what to run next.

Given that, it should be no surprise that schedule() is simple. The only important part of the function—which is otherwise too uninteresting to reproduce here—is its invocation of *pick_next_task()*, also defined in *kernel/sched.c*. The *pick_next_task()* function goes through each scheduler class, starting with the highest priority, and selects the highest priority process in the highest priority class:

```
/*
* Pick up the highest-prio task:
*/
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
        const struct sched_class *class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in
         * the fair class we can call that function directly:
         */
        if (likely(rq->nr_running == rq->cfs.nr_running)) {
                p = fair_sched_class.pick_next_task(rq);
                if (likely(p))
                        return p;
        }

        class = sched_class_highest;
        for ( ; ; ) {
                p = class->pick_next_task(rq); << "ask" the sched
          class for the most suitable task to run next >>
                if (p)
                        return p;
                /*
                 * Will never be NULL as the idle class always
                 * returns a non-NULL p:
                 */
                class = class->next;
        }
}
```

Note the optimization at the beginning of the function. Because CFS is the scheduler class for normal processes, and most systems run mostly normal processes, there is a small hack to quickly select the next CFS-provided process if the number of runnable processes is equal to the number of CFS runnable processes (which suggests that all runnable processes are provided by CFS).

The core of the function is the for() loop, which iterates over each class in priority order, starting with the highest priority class. Each class implements the *pick_next_task()* function, which returns a pointer to its next runnable process or, if there is not one, NULL. The first class to return a non-NULL value has selected the next runnable process. CFS's implementation of *pick_next_task()* calls *pick_next_entity()*, which in turn calls the *__pick_next_entity()* function that we discussed in the previous section.

Preemption and Context Switching

Source:

Lowering Latency in Linux: Introducing a Preemptible Kernel, Robert Love

Performance measurements come in two flavors, throughput and latency.

The former is like the width of an expressway: the wider the expressway, the more cars that can travel on it.

The latter is like the speed limit: the faster it is, the sooner cars get from point A to point B.

Obviously, both quantities are important to any task. Many jobs, however, require more of one quality than of the other. Sticking to our roadway analogy, long-haul trucking may be more sensitive to throughput, while a courier service may be more demanding on latency. Lowering latency and increasing system response, through good old-fashioned kernel work, is the topic of this article.

Audio/video processing and playback are two common beneficiaries of lowered latency. Increasingly important to Linux, however, is its benefit to interactive performance. With high latency, user actions, such as mouse clicks, go unnoticed for too long—not the snappy responsive desktop users expect. The system cannot get to the important process fast enough.

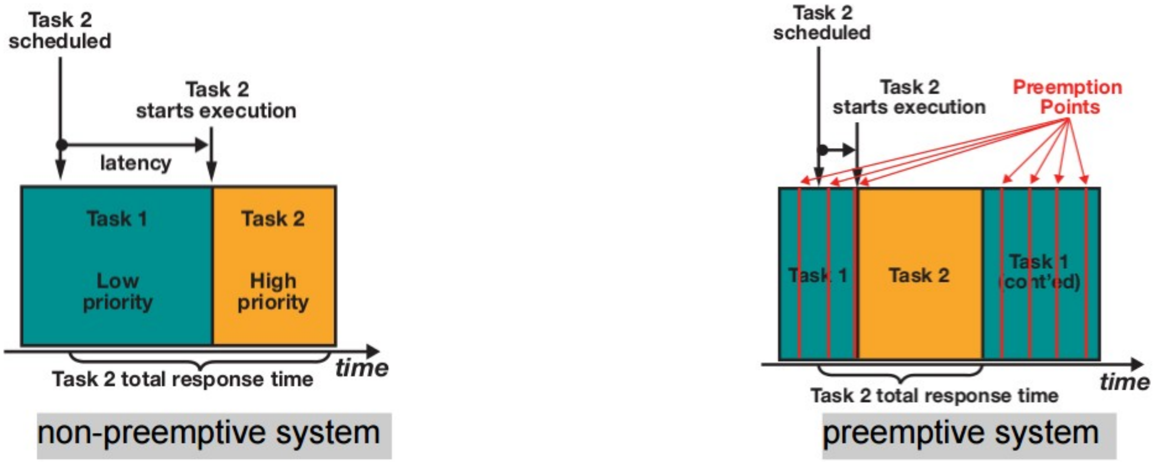
The problem, at least as far as the kernel is concerned, is the nonpreemptibility of the kernel itself. Normally, if something sufficiently important happens, such as an interactive event, the receiving application will get a priority boost and subsequently find itself running. This is how a preemptively multitasked OS works. Applications run until they use up some default amount of time (called a timeslice) or until an important event occurs. The alternative is cooperative multitasking, where applications must explicitly say, “I'm done!”, before a new process can run. The problem, when running in the kernel, is that scheduling is effectively cooperative.

Applications operate in one of two modes: either in user space, executing their own code, or within the kernel, executing a system call or otherwise having the kernel work on their behalf. When operating in the kernel, the process continues running until it decides to stop, ignoring timeslices and important events. If a more important process becomes runnable, it cannot be run until the current process, if it is in the kernel, gets out. This process can take hundreds of milliseconds.

<<

Source: RTMux: A Thin Multiplexer To Provide Hard Realtime Applications For Linux

...



A concept linked to that of real time is preemption: the ability of a system to interrupt tasks at many “preemption points”.The longer the non-interruptible program units are, the longer is the waiting time (‘latency’) of a higher priority task before it can be started or resumed. GNU/Linux is “user-space preemptible”: it allows user tasks to be interrupted at any point. The job of realtime extensions is to make system calls << *their kernel code* >> preemptible as well.

…

>>

Latency Solutions

The first and simplest solution to latency problems is to rewrite kernel algorithms so that they take a minimal, bounded amount of time. The problem is that this is already the goal; system calls are written to return quickly to user space, yet we still have latency problems. Some algorithms simply do not scale nicely.

The second solution is to insert explicit scheduling points throughout the kernel. This approach, taken by the low-latency patches, finds problem areas in the kernel and inserts code to the effect of “Anyone need to run? If so, run!” The problem with this solution is that we cannot possibly hope to find and fix all problem areas. Nonetheless, testing shows that these patches do a good job. What we need, however, is not a quick fix but a solution to the problem itself.

The Preemptible Kernel

A proper solution is removing the problem altogether by making the kernel preemptible. Thus, if something more important needs to run, it will run, regardless of what the current process is doing.

The obstacle here, and the reason Linux did not do this from the start, is that the kernel would need to be re-entrant. Thankfully, the issues of preemption are solved by existing SMP (symmetric multiprocessing) support. By taking advantage of the SMP code, in conjunction with some other simple modifications, the kernel can be made preemptible.

The programmers at MontaVista provided the initial implementation of kernel preemption. First, the definition of a spin lock was modified to include marking a “nonpreemptible” region. Therefore, we do not preempt while holding a spin lock, just as we do not enter a locked region concurrently under SMP. Of course, on uniprocessor systems we do not actually make the spin locks anything other than the preemption markers. Second, code was modified to ensure that we do not preempt inside a bottom half or inside the scheduler itself. Finally, the return from interrupt code path was modified to reschedule the current process if needed.

On UP, spin_lock is defined as preempt_disable, and spin_unlock is defined as preempt_enable. On SMP, they also perform the normal locking. So what do these new routines do?

The nestable preemption markers preempt_disable and preempt_enable operate on preempt_count, a new integer stored in each task_struct.

preempt_disable effectively is:

```
++current->preempt_count;
barrier();
```

and **preempt_enable** is:

```
--current->preempt_count;
barrier();
if (unlikey(!current->preempt_count
    && current->need_resched))
       preempt_schedule();

...

#ifdef CONFIG_PREEMPT
/*
 * this is the entry point to schedule() from in-kernel preemption
 * off of preempt_enable. Kernel preemptions off return from interrupt
 * occur there and call schedule directly.
 */
asmlinkage __visible void __sched notrace preempt_schedule(void)
{
    /*
     * If there is a non-zero preempt_count or interrupts are disabled,
     * we do not want to preempt the current task. Just return..
     */
    if (likely(!preemptible()))
        return;

    preempt_schedule_common();
}
NOKPROBE_SYMBOL(preempt_schedule);
EXPORT_SYMBOL(preempt_schedule);
...
```

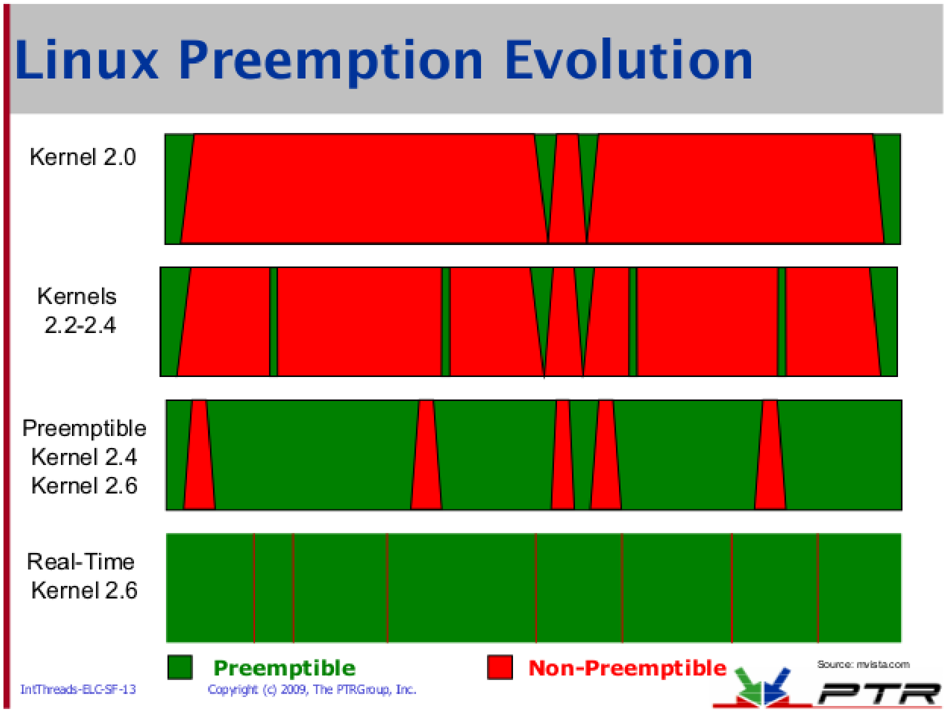
The result is we do not preempt when the count is greater than zero. Because spin locks are already in place to protect critical regions for SMP machines, the preemptible kernel now has its protection too.

…

The other entry to preempt_schedule is via the interrupt return path. When an interrupt handler returns, it checks the preempt_count and need_resched variables, just as preempt_enable does (although the interrupt return code in entry.S is in assembly). The ideal scenario is to cause a preemption here because it is an interrupt that typically sets need_resched due to a hardware event. It is not always possible, however, to preempt immediately off the interrupt, as a lock may be held. That is why we also check for preemption off preempt_enable.

…

Source



The Results

Thus, with the preemptive kernel patch, we can reschedule tasks as soon as they need to be run, not only when they are in user space. What are the results of this?

Process-level response is improved twentyfold in some cases. (See Figure 1, a standard kernel, vs. Figure 2, a preemptible kernel.) These graphs are the output of Benno Senoner's useful latencytest tool, which simulates the buffering of an audio sample under load. The red line in the graphs represents the amount of latency beyond which audio dropouts are perceptible to humans. Notice the multiple spikes in the graph in Figure 1 compared to the smooth low graph in Figure 2.

<< Figures on next page >>

The improvement in latencytest corresponds to a reduction in both worst-case and average latency. Further tests show that the average system latency over a range of workloads is now in the 1-2ms range.

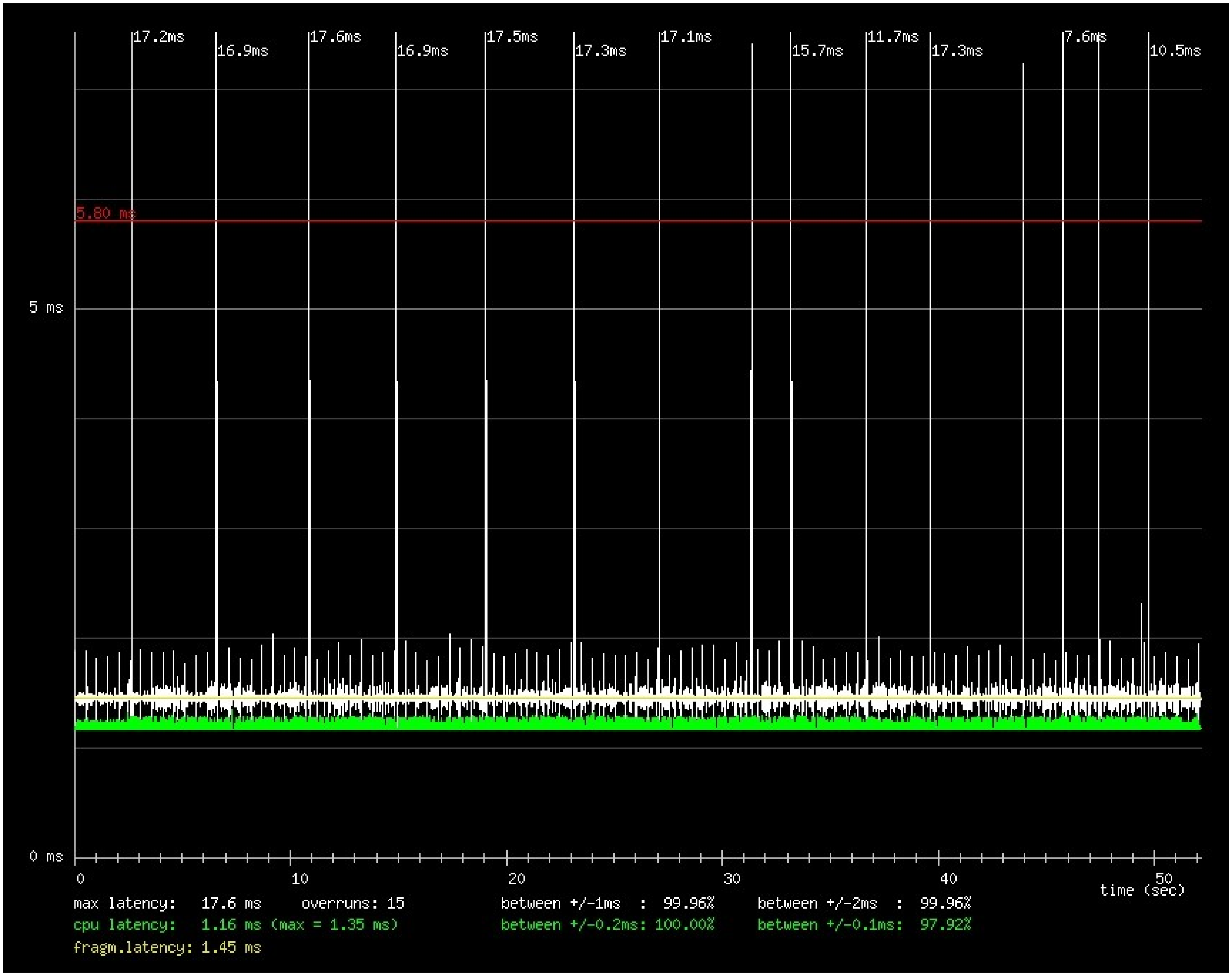


Figure 1. Result of a Latency Test Benchmark on a Standard (vanilla 2.6) Kernel

[P.T.O.]

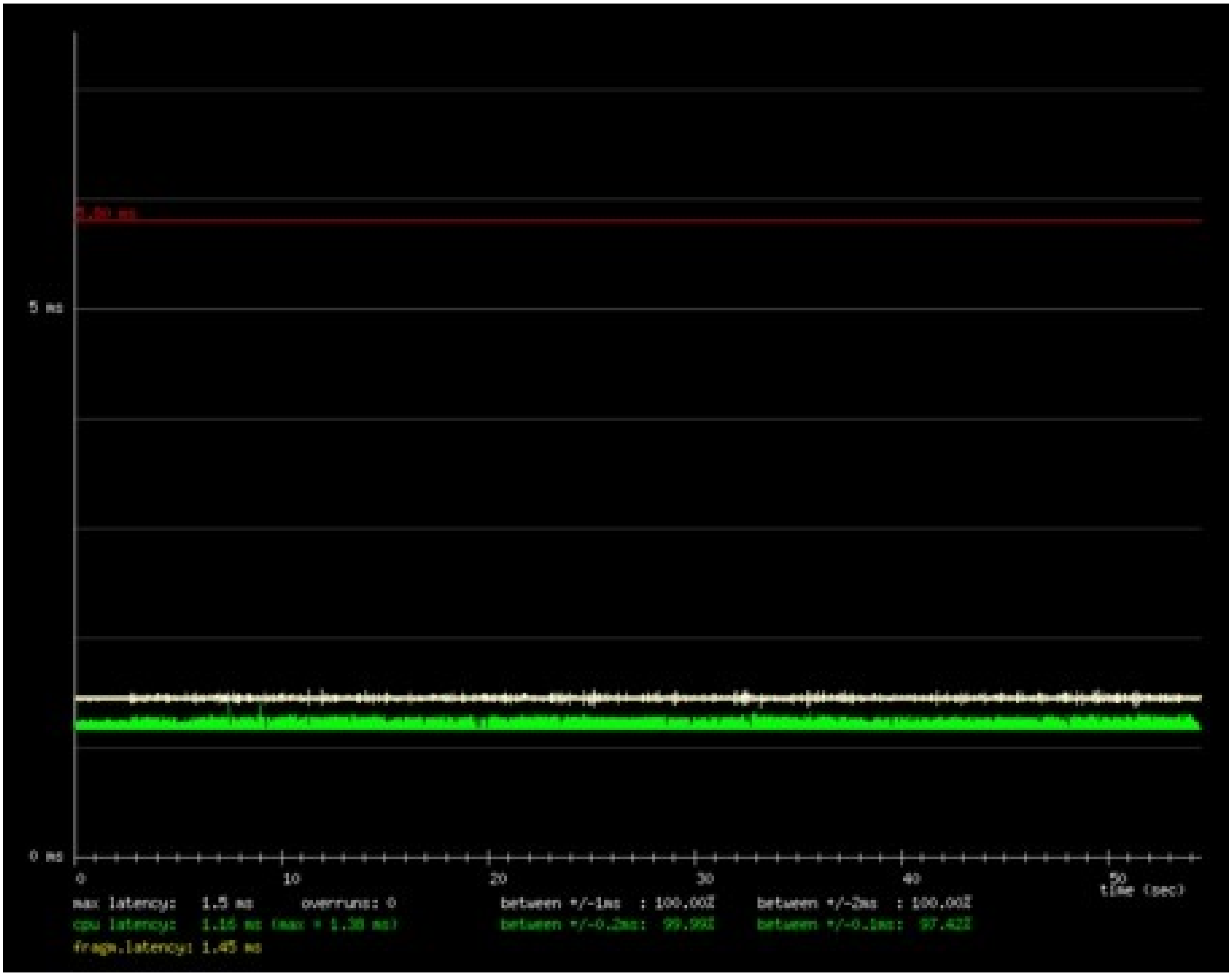


Figure 2. Result of a Latency Test Benchmark on a 2.6 Preemptible Kernel

A common complaint against the preemptible kernel centers on the added complexity. Complexity, opponents argue, decreases throughput. Fortunately, the preemptive kernel patch improves throughput in many cases (see Table 1). The theory is that when I/O data becomes available, a preemptive kernel can wake an I/O-bound process more quickly. The result is higher throughput, a nice bonus. The net result is a smoother desktop, less audio dropout under load, better application response and improved fairness to high-priority tasks.

Table 1. Throughput Test: dbench Runs

Kernel	Throughput
2.5.2	24.3813 MB/s
2.5.2-preempt	28.5920 MB/s

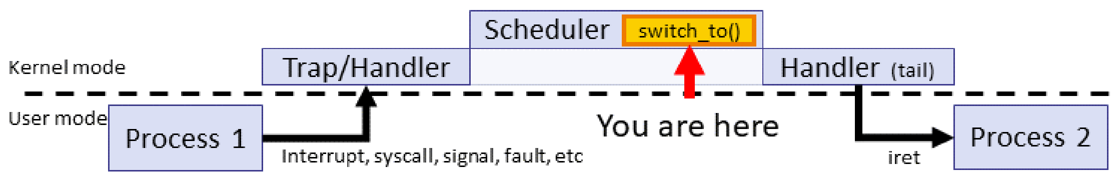
Context Switching

SIDEBAR : A good resource

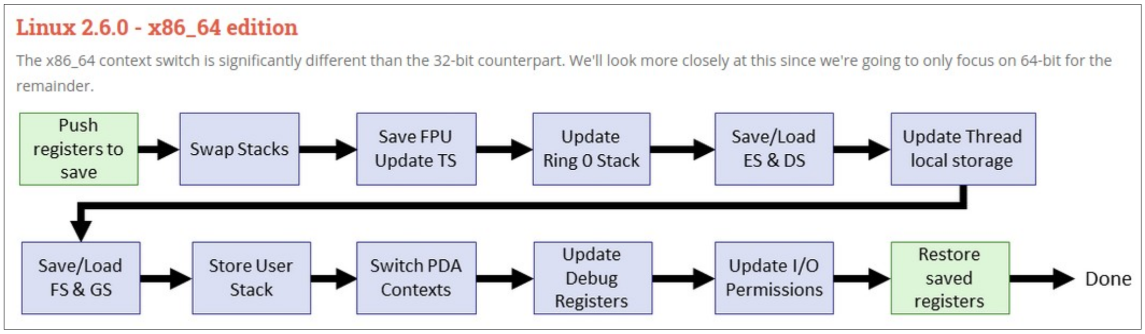
An excellent detailed code-level walkthrough article on the Linux kernel *context switching* code; spans several kernel versions beginning with the 0.01 to one of the latest LTS kernels ver 4.14.67!

Evolution of the x86 context switch in Linux, Maizure, Sept 2018

Below: earliest (v0.01)



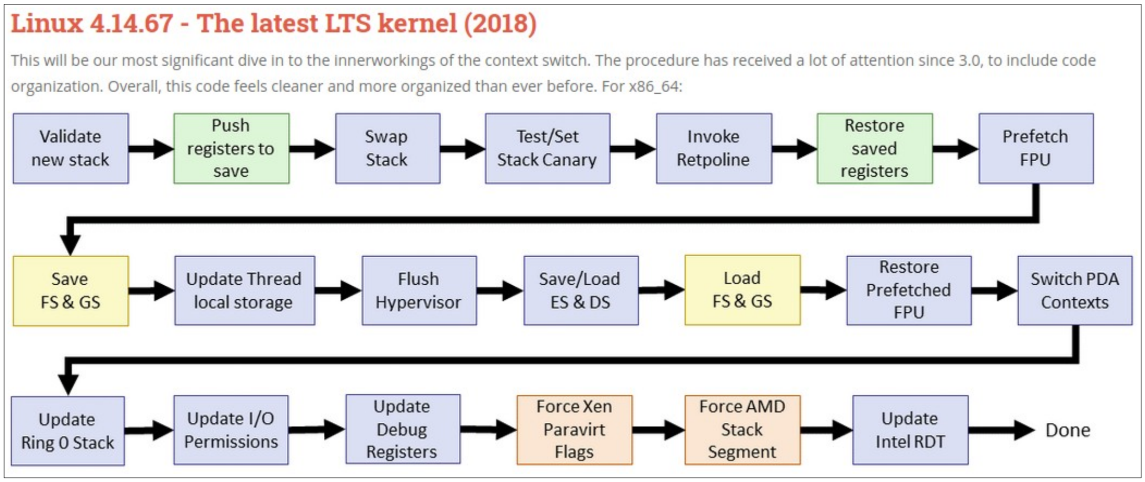
…



Linux 2.6.0 - x86_64 edition

The x86_64 context switch is significantly different than the 32-bit counterpart. We'll look more closely at this since we're going to only focus on 64-bit for the remainder.

…



Linux 4.14.67 - The latest LTS kernel (2018)

This will be our most significant dive in to the innerworkings of the context switch. The procedure has received a lot of attention since 3.0, to include code organization. Overall, this code feels cleaner and more organized than ever before. For x86_64:

Context switching, the switching from one runnable task to another, is handled by the *context_switch()* function defined in *kernel/sched.c*. It is called by *schedule()* when a new process has been selected to run. It does two basic jobs:

- Calls *switch_mm()*, which is declared in <*asm/mmu_context.h*>, to switch the virtual memory mapping from the previous process's to that of the new process.
- Calls *switch_to()*, declared in <*asm/system.h*>, to switch the processor state from the previous process's to the current's. This involves saving and restoring stack information and the processor registers and any other architecture-specific state that must be managed and restored on a per-process basis.

<<

PLKA, Mauerer

The hardware-dependent part of context switching takes place once the scheduler has decided to instruct the current process to relinquish the CPU so that another process can run. For this purpose, all architectures must provide the switch_to function or a corresponding macro with the following prototype in <*asm-arch/system.h*> :

```
<asm-arch/system.h>
void switch_to(struct task_struct *prev, struct task_struct *next, struct
task_struct *last)
```

The function performs a context switch by saving the state of the process specified by prev and activating the process designated by next . Although the last parameter may initially appear to be superfluous, it is used to find the process that was running immediately prior to the function return. Note that *switch_to* is not a function in the usual sense because the system state can change in any number of ways between the start and end of the function.

This can be best understood via an example in which the kernel switches from process A to process B. prev points to A and next to B. Both are local variables in context A. After process B has executed, the kernel switches to other processes and finally arrives at process X; when this process ends, the kernel reverts to process A. Because process A was exited in the middle of switch_to , execution resumes in the second half of the process.

The local variables are retained (the process is not allowed to notice that the scheduler has reclaimed the CPU in the meantime), so prev points to A and next to B. However, this information is not sufficient to enable the kernel to establish which process was running immediately prior to activation of A — although it is important to know this at various points in the kernel. This is where the last variable comes in. The low-level assembler code to implement context switching must ensure that last points to the task structure of

the process that ran last so that this information is still available to the kernel after a switch has been made to process A.
>>

<<
FAQ: How long does a context switch take?
A good article on context-switching time on modern Intel processors: *How long does it take to make a context switch?*
>>

When is schedule() called?

The kernel, however, must know when to call *schedule()*. If it called *schedule()* only when code explicitly did so, user-space programs could run indefinitely. Instead, the kernel provides the *need_resched* flag (particularly, the TIF_NEED_RESCHED bit in the thread_info structure, which resides within the kernel mode stack of current) to signify whether a reschedule should be performed (see Table 4.1).

Table 4.1 - Functions for Accessing and Manipulating need_resched

Function	*Purpose*
`set_tsk_need_resched()`	Set the *need_resched* flag in the given process.
`clear_tsk_need_resched()`	Clear the *need_resched* flag in the given process.
`need_resched()`	Test the value of the *need_resched* flag; return true if set and false otherwise.

Two questions do come up:

Q1. When exactly (and where in the kernel codebase) is the TIF_NEED_RESCHED flag set?

Q2. Where exactly (in the kernel codebase) is the TIF_NEED_RESCHED flag checked to see if it's set (and thus possibly invoke the scheduler)?

Q1. When exactly (and where in the kernel codebase) is the TIF_NEED_RESCHED flag set?

A1. First, we find that the TIF_NEED_RESCHED flag is actually set within the code `void resched_task(struct task_struct *)`

Next, we use cscope (on a recent kernel- 3.10.24) to find all possible call points of *resched_task()* :

```
Functions calling this function: resched_task

  File        Function                 Line
0 core.c      resched_cpu               543 resched_task(cpu_curr(cpu));
1 core.c      check_preempt_curr        963 resched_task(rq->curr);
2 core.c      set_user_nice            3690 resched_task(rq->curr);
3 core.c      yield_to                 4493 resched_task(p_rq->curr);
4 core.c      normalize_task           7137 resched_task(rq->curr);
5 fair.c      check_preempt_tick       1850 resched_task(rq_of(cfs_rq)->curr);
6 fair.c      check_preempt_tick       1874 resched_task(rq_of(cfs_rq)->curr);
7 fair.c      entity_tick              1995 resched_task(rq_of(cfs_rq)->curr);
8 fair.c      __account_cfs_rq_runtime 2184 resched_task(rq_of(cfs_rq)->curr);
9 fair.c      unthrottle_cfs_rq        2336 resched_task(rq->curr);
a fair.c      hrtick_start_fair        2769 resched_task(p);
b fair.c      check_preempt_wakeup     3594 resched_task(curr);
c fair.c      task_fork_fair           5803 resched_task(rq->curr);
d fair.c      prio_changed_fair        5828 resched_task(rq->curr);
e fair.c      switched_to_fair         5885 resched_task(rq->curr);
f idle_task.c check_preempt_curr_idle    33 resched_task(rq->idle);
g rt.c        sched_rt_rq_enqueue       440 resched_task(curr);
h rt.c        sched_rt_rq_enqueue       534 resched_task(rq_of_rt_rq(rt_rq)-
>curr);
i rt.c        update_curr_rt            954 resched_task(curr);
j rt.c        check_preempt_equal_prio 1317 resched_task(rq->curr);
k rt.c        check_preempt_curr_rt    1328 resched_task(rq->curr);
l rt.c        push_rt_task             1648 resched_task(rq->curr);
m rt.c        push_rt_task             1695 resched_task(lowest_rq->curr);
n rt.c        switched_from_rt         1897 resched_task(rq->curr);

* 4 more lines - press the space bar to display more *
<space>
Functions calling this function: resched_task
```

```
  File         Function                 Line
0 rt.c         switched_to_rt           1935 resched_task(rq->curr);
1 rt.c         prio_changed_rt          1964 resched_task(p);
2 rt.c         prio_changed_rt          1968 resched_task(p);
3 rt.c         prio_changed_rt          1977 resched_task(rq->curr);

* Press the space bar to display the first lines again *
```

Q2. Where exactly (in the kernel codebase) is the TIF_NEED_RESCHED flag checked to see if it's set (and thus possibly invoke the scheduler)?

A2. [The kernel checks the flag, sees that it is set, and calls *schedule()* to switch to a new process. The flag is a message to the kernel that the scheduler should be invoked as soon as possible because another process deserves to run.]

When and where does the kernel check the TIF_NEED_RESCHED flag?
The kernel code-paths have some “opportunity points” where the check is made:

- upon returning to user-space (system call return path). If it is set, the kernel invokes the scheduler before continuing.
- upon returning from a hardware interrupt, the need_resched flag is checked. If it is set And preempt_count is zero (meaning we're in a preemtible region of the kernel and no locks are held), the kernel invokes the scheduler before continuing.

The flag is per-process, and not simply global, because it is faster to access a value in the process descriptor (because of the speed of current and high probability of it being cache hot) than a global variable. Historically, the flag was global before the 2.2 kernel. In 2.2 and 2.4, the flag was an int inside the task_struct. In 2.6, it was moved into a single bit of a special flag variable inside the thread_info structure.

SIDEBAR :: Comment in the kernel core scheduler code

(This comment appears just above the __schedule() function here:

[3.10.24]:kernel/sched/core.c:__schedule()

…

```
/*
 * __schedule() is the main scheduler function.
 *
 * The main means of driving the scheduler and thus entering this function are:
 *
 *   1. Explicit blocking: mutex, semaphore, waitqueue, etc.
 *
 *   2. TIF_NEED_RESCHED flag is checked on interrupt and userspace return
 *      paths. For example, see arch/x86/entry_64.S.
 *
 *     To drive preemption between tasks, the scheduler sets the flag in timer
 *     interrupt handler scheduler_tick().
 *
 *  3. Wakeups don't really cause entry into schedule(). They add a
 *     task to the run-queue and that's it.
 *
 *     Now, if the new task added to the run-queue preempts the current
 *     task, then the wakeup sets TIF_NEED_RESCHED and schedule() gets
 *     called on the nearest possible occasion:
 *
 *      - If the kernel is preemptible (CONFIG_PREEMPT=y):
 *
 *        - in syscall or exception context, at the next outmost
 *          preempt_enable(). (this might be as soon as the wake_up()'s
 *          spin_unlock()!)
 *
 *        - in IRQ context, return from interrupt-handler to
 *          preemptible context
 *
 *      - If the kernel is not preemptible (CONFIG_PREEMPT is not set)
 *        then at the next:
 *
 *         - cond_resched() call
 *         - explicit schedule() call
 *         - return from syscall or exception to user-space
 *         - return from interrupt-handler to user-space
 */
...
```

SIDEBAR :: Scheduling and Interrupt context

A “golden rule” of the Linux kernel:
WE CANNOT SCHEDULE IN INTERRUPT CONTEXT.

Why not?

From Robert Love (a kernel hacker):
http://mail.nl.linux.org/kernelnewbies/2003-07/msg00026.html
You cannot sleep in an interrupt handler because interrupts do not have a backing process context, and thus there is nothing to reschedule back into. In other words, interrupt handlers are not associated with a task, so there is nothing to "put to sleep" and (more importantly) "nothing to wake up". They must run atomically.

Source

In detail:
Why kernel code/thread executing in interrupt context cannot sleep?
How are system calls interrupted by signal?

FYI

CFS Email Announcement from Ingo Molnar

Reproduced below is the actual email (patch) sent by **Ingo Molnar** (the current maintainer of the scheduler) to the LKML:

From: Ingo Molnar [email blocked]

To: linux-kernel
Subject: [Announce] [patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]
Date: Fri, 13 Apr 2007 22:21:00 +0200

[announce] [patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]

i'm pleased to announce the first release of the "Modular Scheduler Core and Completely Fair Scheduler [CFS]" patchset:

http://redhat.com/~mingo/cfs-scheduler/sched-modular+cfs.patch

This project is a complete rewrite of the Linux task scheduler. My goal is to address various feature requests and to fix deficiencies in the vanilla scheduler that were suggested/found in the past few years, both for desktop scheduling and for server scheduling workloads.

[QuickStart: apply the patch to v2.6.21-rc6, recompile, reboot. The new scheduler will be active by default and all tasks will default to the new SCHED_FAIR interactive scheduling class.]

Highlights are:

- the introduction of Scheduling Classes: an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming about them too much.

- sched_fair.c implements the 'CFS desktop scheduler': it is a replacement for the vanilla scheduler's SCHED_OTHER interactivity code.

  i'd like to give credit to Con Kolivas for the general approach here: he has proven via RSDL/SD that 'fair scheduling' is possible and that it results in better desktop scheduling. Kudos Con!

  The CFS patch uses a completely different approach and implementation from RSDL/SD. My goal was to make CFS's interactivity quality exceed that of RSDL/SD, which is a high standard to meet :-) Testing feedback is welcome to decide this one way or another. [and, in any case, all of SD's logic could be added via a kernel/sched_sd.c module as well, if Con is interested in such an approach.]

  CFS's design is quite radical: it does not use runqueues, it uses a time-ordered rbtree to build a 'timeline' of future task execution, and thus has no 'array switch' artifacts (by which both the vanilla scheduler and RSDL/SD are affected).

CFS uses nanosecond granularity accounting and does not rely on any jiffies or other HZ detail. Thus the CFS scheduler has no notion of 'timeslices' and has no heuristics whatsoever. There is only one central tunable:

```
      /proc/sys/kernel/sched_granularity_ns
```

which can be used to tune the scheduler from 'desktop' (low latencies) to 'server' (good batching) workloads. It defaults to a setting suitable for desktop workloads. SCHED_BATCH is handled by the CFS scheduler module too.

due to its design, the CFS scheduler is not prone to any of the 'attacks' that exist today against the heuristics of the stock scheduler: fiftyp.c, thud.c, chew.c, ring-test.c, massive_intr.c all work fine and do not impact interactivity and produce the expected behavior.

the CFS scheduler has a much stronger handling of nice levels and SCHED_BATCH: both types of workloads should be isolated much more agressively than under the vanilla scheduler.

(another rdetail: due to nanosec accounting and timeline sorting, sched_yield() support is very simple under CFS, and in fact under CFS sched_yield() behaves much better than under any other scheduler i have tested so far.)

- sched_rt.c implements SCHED_FIFO and SCHED_RR semantics, in a simpler way than the vanilla scheduler does. It uses 100 runqueues (for all 100 RT priority levels, instead of 140 in the vanilla scheduler) and it needs no expired array.

- reworked/sanitized SMP load-balancing: the runqueue-walking assumptions are gone from the load-balancing code now, and iterators of the scheduling modules are used. The balancing code got quite a bit simpler as a result.

the core scheduler got smaller by more than 700 lines:

```
 kernel/sched.c | 1454 +++++++++++++++
+------------------------------------------------
 1 file changed, 372 insertions(+), 1082 deletions(-)
```

and even adding all the scheduling modules, the total size impact is relatively small:

```
 18 files changed, 1454 insertions(+), 1133 deletions(-)
```

most of the increase is due to extensive comments. The kernel size impact is in fact a small negative:

```
   text    data     bss     dec     hex filename
  23366    4001      24   27391    6aff kernel/sched.o.vanilla
  24159    2705      56   26920    6928 kernel/sched.o.CFS
```

(this is mainly due to the benefit of getting rid of the expired array and its data structure overhead.)

```
thanks go to Thomas Gleixner and Arjan van de Ven for review of this
patchset.

as usual, any sort of feedback, bugreports, fixes and suggestions are
more than welcome,

        Ingo
```

You can see the entire thread here on KernelTrap:
“Linux: The Completely Fair Scheduler”
April 18, 2007 - 10:51am
Submitted by Jeremy on April 18, 2007 – 10:51am.
http://kerneltrap.org/node/8059

Resources

- “Linux Kernel Development” 3rd Ed, by Robert M Love
- Resource: “Multiprocessing with the Completely Fair Scheduler” by Avinesh Kumar, IBM.
  http://www.ibm.com/developerworks/linux/library/l-cfs/

Dynticks and Tickless Kernels

Recommended LWN article: "Clockevents and dyntick"

(This feature touches a lot of low-level x86 code, so regressions are possible. If you have problems with 2.6.21, please report it)

Clockevents is a building block of dynticks. An example of a clock device is the device which makes timer interrupts. Previously, the handling of such devices was made in architecture-specific code preventing a unified method of using those devices. The clockevents patch unifies the clock device handling so the kernel can use the timer capabilities of those devices in a unified manner. This also allows to implement true high-resolution timers.

Dynticks (aka: dynamic ticks) is a configurable feature for x86 32bits (x86-64 and ARM support is already done but not ready for this release; PPC and MIPS support are in the works) that changes the heart of the mechanism that allows a system to implement multitasking.

To know what dyntick does, first you should know some basics: Traditionally, multitasking is implemented using a timer interrupt that is configured to fire N times in a second. Firing this interrupt causes a call to the operating system's process scheduler routines << *edit: not necessarily every time it fires, but rather on an on-needed basis.* >> .

The scheduler then decides which process should run next - either the process that was running before the timer interrupt was fired or another process. This is how true multitasking is implemented in all the general-purpose operating systems and is also what stops processes from being able to monopolize the CPU time: the timer interrupt will be fired regardless of what the process is doing and the operating system will be able to stop it (pre-emption).

N (the number of times the timer interrupt is fired in each second, aka 'HZ') is a hard coded compile-time architecture-dependent constant. For Linux 2.4 (and Windows), HZ=100 (the timer interrupt fires 100 times per second).

2.6 increased HZ to 1000 for several reasons. 100 was the HZ value that x86 had been using since forever and it didn't really make a lot of sense in modern CPUs that run much faster. Higher HZ means smaller process time slices, which improves the minimum latency and interactivity. The downside is higher timer overhead (negligible in modern hardware, although some server-oriented distros package kernels with HZ=100 because of minor performance gains) and high pitch noises in some systems due to cheap, low-quality capacitors.

Anyway, the issue is that the timer is fired HZ times in every second - even if the system is doing nothing. Dynticks is a feature that stops the system from having to always wake up HZ times per second. When the system is entering the idle loop it *disables the periodic timer interrupt* and programs the timer to fire the next time a timer event is needed.

<< How exactly?
In the idle loop, the kernel checks for when the next pending timer event will occour (using the "timer wheel" data structure (?)). If the event is further than 1 clock tick away, it disables the timer interrupt, programming it to next occour just before (1 tick before) the pending timer event is due.
>>

This means your system will be 'disabled' while there's nothing to do (unless an interrupt happens - e.g. an incoming packet through your network).

For now, this is all dynticks does. However, this infrastructure will enable the creation of an innovative power-saving feature - when dynticks is in "tickless" mode and the system is waiting for the timer interrupt, the power-saving feature of modern CPUs will be used for longer. This can save a few watts when a laptop is idle. It's also very useful for virtualization software - since the virtualization host has to execute all those timer notifications there's some overhead even when not doing anything (especially when there are lots of virtual cpus); with Dynticks the host handles less timer notifications when the virtual guest is doing anything.

Dynticks adds some nice configurable debugging features. */proc/timer_list* prints all the pending timers, allowing developers to check if their program is doing something when it should be doing nothing. */proc/timer_stat* collects some timer statistics allowing detection of sources of commonly-programmed timers.

Source.

Recent:

Source
"***Timerless multitasking : Kernel ver 3.10***

In the prehistory of computing, computers could only have one task running at one time. But people wanted to start other tasks without waiting for first one to end, and even switch between tasks, and thus multitasking was born.

First, multitasking was "collaborative", a process would run until its own code voluntarily decided to pause and allow other tasks to run. But it was possible to do multitasking better: the hardware could have a timer that fires up at regular intervals (called "ticks"); this timer could forcefully pause any program and run a OS routine that decides which task should continue running next. This is called preemptive multitasking, and it's what modern OSs do.

But preemptive multitasking had some side effects in modern hardware. CPUs of laptops and mobile devices require inactivity to enter in low power modes. Preemptive multitasking fires the the timer often, 1000 times per second in a typical Linux kernel, even when the system is not doing anything, so the CPUs could not save as much power as it was possible. Virtualization created more problems, since each Linux VM runs its own timer.In 2.6.21, released in April 2007, Linux partially solved this: the timer would fire off 1000 times per second as always when the system is running tasks, but it would stop completely the timer when the system is idle << *dynticks* >>.

But this is not enough. There are single task workloads like scientific number crunching or users of the real-time pachset whose performance or latency is hurt because they need to be temporally paused 1000 times per second for no reason.

This Linux release << 3.10 >> adds support for not firing the timer (tickless) even when tasks are running. With some caveats: in this release it's not actually fully tickless, it still needs the timer, but only fires up one time per second; the full tickless mode is disabled when a CPU runs more than one process; and a CPU must be kept running with full ticks to allow other CPUs to go into tickless mode.

For more details and future plans, it's strongly recommended to read this LWN article: '(Nearly) full tickless operation in 3.10' and the Documentation.

Code: (merge commit)”

Superbly illustrated article : “What does an idle CPU do?”, Gustavo Duarte

Linux kernel Scheduler | Upcoming

- 3.16 : “The power-aware scheduling for the Linux kernel has been something that's been in the works for many months and is nearing fruition.”

[FYI / OPTIONAL]

Appendix A :: CPU Scheduler Load Balancing

Source: The article “Load-Balancing for Improving User Responsiveness on Multicore Embedded Systems”, Lim, Min & Eom.

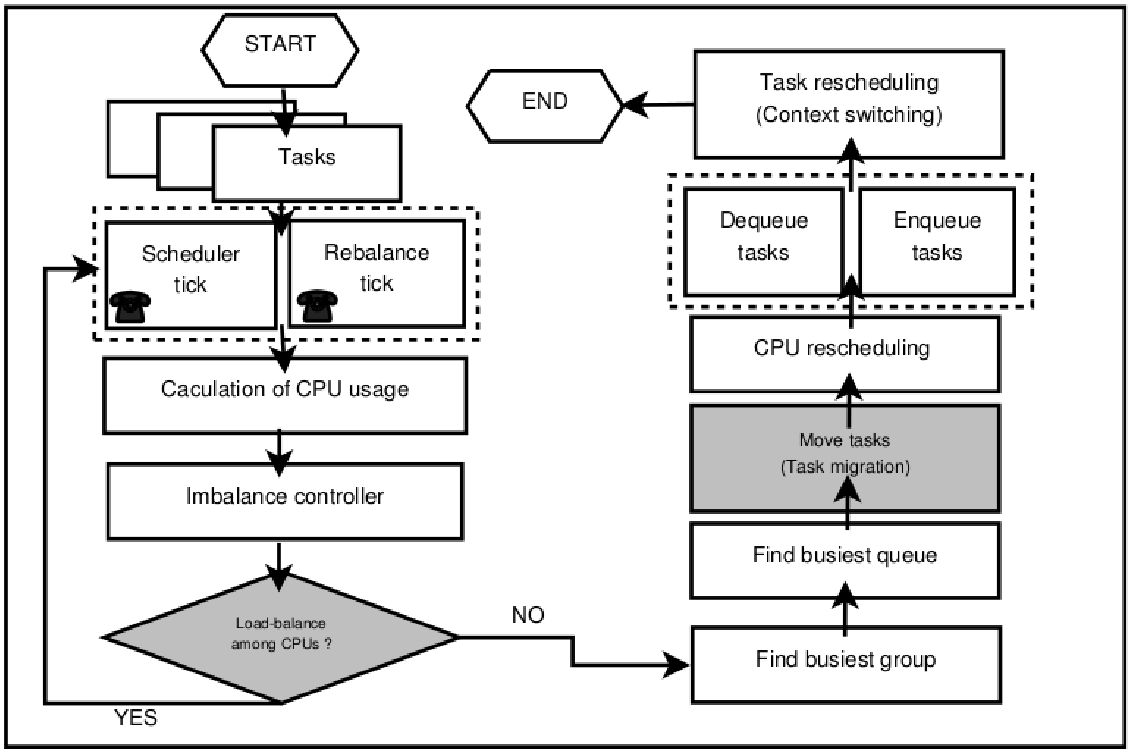
…

The current SMP scheduler in Linux kernel periodically executes the load-balancing operation to equally utilize each CPU core whenever load imbalance among CPU cores is detected. [Such aggressive load-balancing operations incur unnecessary task migrations even when the CPU cores are not fully utilized, and thus, they incur additional cache invalidation, scheduling latency, and power consumption.]

If the load sharing of CPUs is not fair, the multicore scheduler [10] makes an effort to solve the system’s load imbalance by entering the procedure for load-balancing [11]. *Figure 1* shows the overall operational flow when the SMP scheduler [2] performs the load-balancing.

At every timer tick, the SMP scheduler determines whether it needs to start load-balancing [20] or not, based on the number of tasks in the per-CPU run-queue.
At first, it calculates the average load of each CPU [12].

If the load imbalance between CPUs is not fair, the load-balancer selects the task with the highest CPU load [13], and then lets the migration thread move the task to the target CPU whose load is relatively low. Before migrating the task, the load-balancer checks whether the task can be instantly moved. If so, it acquires two locks, *busiest->lock* and *this_rq->lock*, for synchronization before moving the task.



After the successful task migration, it releases the previously held double-locks [5]. The definitions of the terms in Figure 1 are as follows [10] [18]:

- Rebalance_tick: update the average load of the runqueue.
- Load_balance: inspect the degree of load imbalance of the scheduling domain [27].
- Find_busiest_group: analyze the load of groups within the scheduling domain.
- Find_busiest_queue: search for the busiest CPU within the found group.
- Move_tasks: migrate tasks from the source runqueue to the target run-queue in *Figure 1*

other CPU.
…
…

Appendix B :: Scheduler Domains

Source

The new scheduler work is a response to the needs of modern hardware and, in particular, the fact that the processors in multi-CPU systems have unequal relationships with each other.

Virtual CPUs in a hyperthreaded set share equal access to memory, cache, and even the processor itself. Processors on a symmetric multiprocessing system have equal access to memory, but they maintain their own caches. NUMA architectures create situations where different nodes have different access speeds to different areas of main memory. A modern large system can feature all of these situations: each NUMA node looks like an SMP system which may be made up of multiple hyperthreaded processors.

One of the key problems a scheduler must solve on a multi-processor system is balancing the load across the CPUs. It doesn't do to have some processors being heavily loaded while others sit idle. But moving processes between processors is not free, and some sorts of moves (across NUMA nodes, for example, where a process could be separated from its fast, local memory) are more expensive than others. Teaching the scheduler to migrate tasks intelligently under many different types of loads has been one of the big challenges of the 2.5 development cycle.

The domain-based scheduler aims to solve this problem by way of a new data structure which describes the system's structure and scheduling policy in sufficient detail that good decisions can be made. To that end, it adds a couple of new structures:

A scheduling domain (*struct sched_domain*) is a set of CPUs which share properties and scheduling policies, and which can be balanced against each other. Scheduling domains are hierarchical; a multi-level system will have multiple levels of domains.

Each domain contains one or more CPU groups (*struct sched_group*) which are treated as a single unit by the domain. When the scheduler tries to balance the load within a domain, it tries to even out the load carried by each CPU group without worrying directly about what is happening within the group.

<<
Source: Documentation/vm/numa
...
The Linux scheduler is aware of the NUMA topology of the platform – embodied in the "scheduling domains" data structures [see *Documentation/scheduler/sched-domains.txt*]--and the scheduler attempts to minimize task migration to distant scheduling domains.

However, the scheduler does not take a task's NUMA footprint into account directly. Thus, under sufficient imbalance, tasks can migrate between nodes, remote from their initial node and kernel data structures.

System administrators and application designers can restrict a task's migration to improve NUMA locality using various CPU affinity command line interfaces, such as taskset(1) and numactl(1), and program interfaces such as sched_setaffinity(2). Further, one can modify the kernel's default local allocation behavior using Linux NUMA memory policy.
[see *Documentation/vm/numa_memory_policy.txt*.]

System administrators can restrict the CPUs and nodes' memories that a non-privileged user can specify in the scheduling or NUMA commands and functions using control groups and CPUsets. [see *Documentation/cgroups/cpusets.txt*]
...
>>

<<
From the StackOverflow Q&A “what does struct sched_domain stands for in include/linux/sched.h (scheduling domains in kernel)” :

…

Scheduling domains and scheduler groups/cpu groups help to ease the process of scheduling tasks like:
1. Load Balancing tasks across cpus.
2. Choosing a cpu for a new task to run on.
3. Choosing a cpu for a sleeping task to run when it wakes up.

It has a two fold advantage:
1. It organises the cpus in the system very well into groups and hierarchies.
2. It organises the cpus in such a way that it is useful.All cpus which share an L2 cache belong to one domain. All cpus which share an L3 cache belong to a higher level domain,which encompasses all the domains which share the L2 cache.

The advantages that you see with a tree like data structure are similar here to the advantages of scheduler domains and groups.

[P.T.O.]

Refer to following diagram:
[Imagine a system, with what appears to be 8 independent cpu cores. In reality, the hardware package is such that there are two physical cpu packages, each having two sockets (?), and within each socket there are two hyperthreaded processors. Thus we have a hierachy, which the scheduling domains represent.]

```
 ________ sd1 ________            ________ sd1 ________
/                     \          /                     \
-----------------------          -----------------------
       L3 cache                         L3 cache
-----------------------          -----------------------
---------   ----------           ---------   ----------
L2 cache     L2 cache            L2 cache     L2 cache
---------   ----------           ---------   ----------
cpu0 cpu1   cpu2 cpu3            cpu4 cpu5   cpu6 cpu7
\_______/   \________/           \_______/   \________/
   sd0         sd0                  sd0         sd0
```

What you see above is a scheduler domain hierarchy. sd1 encompasses sd0's which happen to be scheduler groups of sd1. Every cpu has a scheduler domain hierarchy associated with it. For eg.
cpu0->sd=sd0; sd0->parent=sd1. This way through a linked list we can iterate through all the scheduler domains to which a cpu belongs to.

How does this help?

1. Load balancing: Say cpu0 is idle and is ready to pull tasks upon itself to relieve any other burdened cpu. In the above approach,it first checks if the other cpus that belong to the first level sched domain ,needs to be relieved of load. Here, cpu1.If so it takes on tasks from cpu1,else it goes to the higher level domain sd1. If it chooses to migrate task from cpu1 it is the best thing,because the cache contents can be utilized;shared cache.no need to fetch from memory again. This is the first advantage: sched domains are formed based upon the advantages that hardware has to provide.

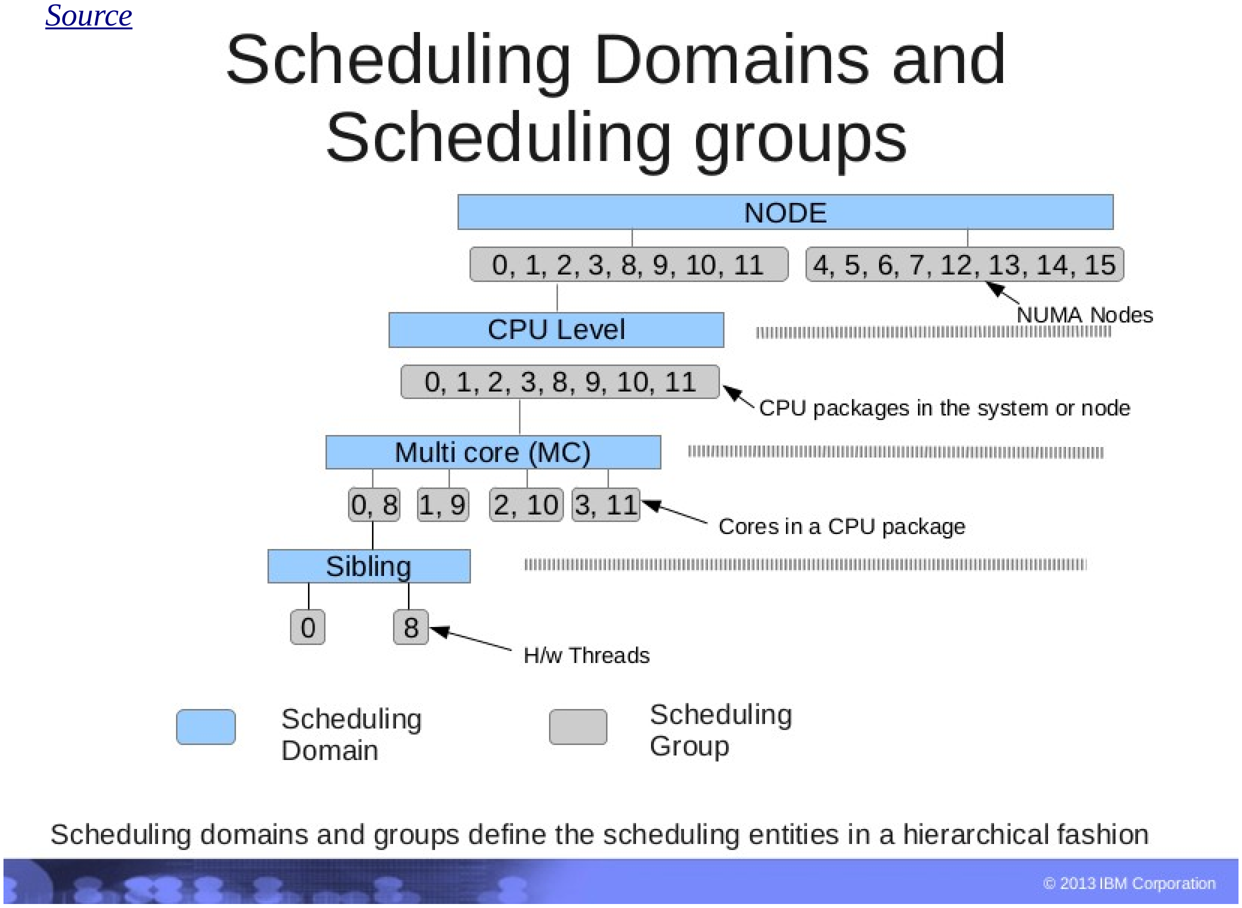
2. If it goes to sd1, then it probes sd1's 'groups',both the sd0s. Here is the next advantage. It needs information about the sched group alone and will not bother about the individual cpus in it.it checks if load(sd0[cpu2,cpu3]) > load(sd0[cpu0,cpu1]) Only if this is true does it go on to see if cpu2/3 is more loaded.If there were no scheduler domain or groups,we would have to see the states of cpu2 and cpu3 in two iterations instead of 1 iteration like we are doing now.

Now scale this problem and solution to 128 cpus! imagine what a mess it would have been if there was nothing to tell you which cpu would be the best to relieve load from,in the worst case you would have to iterate through all the 128 cpus.

But with scheduler domain or groups,say you divide the 128 cpus into groups of 16 cpus,you would have 8 groups.see which is the busiest,so that would be 8 iterations,then you would know the busiest group,then descend down.another 16 iterations. so worst case 8+16 = 24 iterations.And this decrease is only with one level of sched domain. Imagine if you had more levels,you would make the number of iterations even lower.

So in short the scheduler domains and groups are a 'divide and conquer ;but conquer as much as possible what is more useful' solution to scheduling related stuff.
>>

<<
Source



Scheduling Domains and Scheduling groups

Scheduling domains and groups define the scheduling entities in a hierarchical fashion

© 2013 IBM Corporation

On a NUMA system, the scheduling domains hierarchy can be pretty big!
>>

Each scheduling domain contains policy information which controls how decisions are made at that level of the hierarchy. The policy parameters include how often attempts should be made to balance loads across the domain, how far the loads on the component processors are allowed to get out of sync before a balancing attempt is made, how long a process can sit idle before it is considered to no longer have any significant cache affinity, and various policy flags. These policies tend to be set as follows:

At the hyperthreaded processor level: balancing attempts can happen often (every 1-2ms), even when the imbalance between processors is small. There is no cache affinity at all: since hyperthreaded processors share cache, there is no cost to moving a process from one to another. Domains at this level are also marked as sharing CPU power; we'll see how that information is used shortly.

At the physical processor level: balancing attempts do not have to happen quite so often, and they are curtailed fairly sharply if the system as a whole is busy. Processor loads must be somewhat farther out of balance before processes will be moved within the domain. Processes lose their cache affinity after a few milliseconds.

At the NUMA node level: balancing attempts are made relatively rarely, and cache affinity lasts longer. The cost of moving a process between NUMA nodes is relatively high, and the policy reflects that.

The scheduler uses this structure in a number of ways. For example, when a sleeping process is about to be awakened, the normal behavior would be to keep it on the same processor it was using before, on the theory that there might still be some useful cache information there. If that processor's scheduling domain has the SD_WAKE_IDLE flag set, however, the scheduler will look for an idle processor within the domain and move the process immediately if one is found. This flag is used at the hyperthreading level; since the cost of moving processes is insignificant, there is no point in leaving a processor idle when a process wants to run.

When a process calls exec() to run a new program, its current cache affinity is lost. At that point, it may make sense to move it elsewhere. So the scheduler works its way up the domain hierarchy looking for the highest domain which has the SD_BALANCE_EXEC flag set. The process will then be shifted over to the CPU within that domain with the lowest load. Similar decisions are made when a process forks.

If a processor becomes idle, and its domain has the SD_BALANCE_NEWIDLE flag set, the scheduler will go looking for processes to move over from a busy processor within the domain. A NUMA system might set this flag within NUMA nodes, but not at the top level.

The new scheduler does an interesting thing with "shared CPU" (hyperthreaded) processors. If one processor in a shared pair is running a high-priority process, and a low-priority process is trying to run on the other processor, the scheduler will actually idle the second processor for a while. In this way, the high-priority process is given better access to the shared package.

The last component of the domain scheduler is the active balancing code, which moves processes within domains when things get too far out of balance. Every scheduling domain has an interval which describes how often balancing efforts should be made; if the system tends to stay in balance, that interval will be allowed to grow. The scheduler "rebalance tick" function runs out of the clock interrupt handler; it works its way up the domain hierarchy and checks each one to see if the time has come to balance things out. If so, it looks at the load within each CPU group in the domain; if the loads differ by too much, the scheduler will try to move processes from the busiest group in the domain to the most idle group. In doing so, it will take into account factors like the cache affinity time for the domain.

Active balancing is especially necessary when CPU-hungry processes are competing for access to a hyperthreaded processor. The scheduler will not normally move running processes, so a process which just cranks away and never sleeps can be hard to dislodge. The balancing code, by way of the migration threads, can push the CPU hog out of the processor for long enough to allow it to be moved and spread the load more widely.

When the system is trying to balance loads across processors, it also looks at a parameter kept within the sched_group structure: the total "CPU power" of the group. Hyperthreaded processors look like independent CPUs, but the total computation power of a pair of hyperthreaded processors is far less than that of two separate packages. Two separate processors would have a "CPU power" of two, while a hyperthreaded pair would have something closer to 1.1. When the scheduler considers moving a process to balance out the load, it looks at the total amount of CPU power currently being exercised. By maximizing that number, it will tend to spread processes across physical processors and increase system throughput.

<<
To “see” the scheduling domains available on the system (this assumes that CONFIG_SCHEDSTATS is enabled within the kernel):

```
$ cat /proc/schedstat
version 15
timestamp 20223772
cpu0 132 0 67892392 30908996 37756976 19188071 7226705509182 518734824999 36971782
domain0 03 16782 15089 1349 136937 391 9 1 15088 4967 4596 313 28257 97 0 0 4596 639296 626681 8783 788013 5271
156 173 626508 0 0 0 0 0 0 0 0 0 154527 15891 0
domain1 0f 14453 13082 1023 200090 438 1 702 3226 2988 2793 177 26538 20 0 109 1079 635464 580929 49089 10748500
6832 4 68692 512237 0 0 0 0 0 0 0 0 0 315501 26897 0
cpu1 87 0 46241558 20608152 23161330 8479999 5024141542305 484256301415 25623656
domain0 03 8449 6886 1116 136409 553 27 1 6885 6170 5163 786 74586 258 0 3 5160 392542 378127 9852 1024418 5545
210 157 377970 2 0 2 0 0 0 0 0 0 119226 8404 0
```

```
domain1 0f 7258 6766 307 56614 226 0 43 394 4333 4333 0 0 0 0 0 0 387979 340408 42738 9588209 6046 2 47724
292684 0 0 0 0 0 0 0 0 0 217676 9262 0
...
$
```

>>

Appendix C : Cgroups and CFS

Assuming the kernel is configured with Cgroup support and particularly CONFIG_FAIR_SCHED_GROUP, the output of /proc/sched_debug does show the cgroup the task belongs to:

```
...
R             t1 26413   3475532.352651       112   120   3475532.352651
20.687059       115.716940 /user/1000.user/c2.session
…
```

```
...
#ifdef CONFIG_CGROUP_SCHED
    SEQ_printf(m, " %s", task_group_path(task_group(p)));
#endif
...
```

task_struct ptr: p->sched_task_group;

is the scheduling group (cgroup) the task belongs to.
See the struct task_group :

kernel/sched/sched.h

```
...
/* task group related information */
struct task_group {
    struct cgroup_subsys_state css;

#ifdef CONFIG_FAIR_GROUP_SCHED
    /* schedulable entities of this group on each cpu */
    struct sched_entity **se;
    /* runqueue "owned" by this group on each cpu */
    struct cfs_rq **cfs_rq;
    unsigned long shares;

    atomic_t load_weight;
    atomic64_t load_avg;
    atomic_t runnable_avg;
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity **rt_se;
    struct rt_rq **rt_rq;

    struct rt_bandwidth rt_bandwidth;
#endif

    struct rcu_head rcu;
    struct list_head list;

    struct task_group *parent;
    struct list_head siblings;
    struct list_head children;

#ifdef CONFIG_SCHED_AUTOGROUP
    struct autogroup *autogroup;
#endif

    struct cfs_bandwidth cfs_bandwidth;
};
```

The weights are used in the CFS vruntime calculation; thus, the Cgroup weightage gets taken into account for scheduling onto the CFS rbtree!

kernel/sched/fair.c

```
...
static struct task_struct *pick_next_task_fair(struct rq *rq)
{
    struct task_struct *p;
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;

    if (!cfs_rq->nr_running)
        return NULL;

    do {
        se = pick_next_entity(cfs_rq);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);
...
```

We can see that the task picked is from 'cfs_rq'. cfs_rq in turn, is determined from *group_cfs_rq()* :

```
…
/* runqueue "owned" by this group */
static inline struct cfs_rq *group_cfs_rq(struct sched_entity *grp)
{
    return grp->my_q;
}
...
```

>>

Appendix D :: The O(1) Scheduler

The Linux 2.6 O(1) Scheduler

LKD2, Ch 4 "The Linux Scheduling Algorithm" section "The Linux Scheduling Algorithm" page 43.

The Linux Scheduling Algorithm

In the previous sections, we discussed process scheduling theory in the abstract, with only occasional mention of how Linux applies a given concept to reality. With the foundation of scheduling now built, we can dive into Linux's very own process scheduler.

The Linux scheduler is defined in `kernel/sched.c`. The scheduler algorithm and supporting code went through a large rewrite early in the 2.5 kernel development series.

Consequently, the scheduler code is entirely new and unlike the scheduler in previous kernels. The new scheduler was designed to accomplish specific goals:

- Implement fully `O(1)` scheduling. Every algorithm in the new scheduler completes in constant-time, regardless of the number of running processes.
- Implement perfect SMP scalability. Each processor has its own locking and individual runqueue.
- Implement improved SMP affinity. Attempt to group tasks to a specific CPU and continue to run them there. Only migrate tasks from one CPU to another to resolve imbalances in runqueue sizes.
- Provide good interactive performance. Even during considerable system load, the system should react and schedule interactive tasks immediately.
- Provide fairness. No process should find itself starved of timeslice for any reasonable amount of time. Likewise, no process should receive an unfairly high amount of timeslice.
- Optimize for the common case of only one or two runnable processes, yet scale well to multiple processors, each with many processes.

The new scheduler accomplished these goals.

Runqueues

The basic data structure in the scheduler is the runqueue. The runqueue is defined in `kernel/sched.c` as `struct rq`. The runqueue is the list of runnable processes on a given processor; there is one runqueue per processor. Each runnable process is on exactly one runqueue. The runqueue additionally contains per-processor scheduling information. Consequently, the runqueue is the primary scheduling data structure for each processor.

[3] Why `kernel/sched.c` and not `<linux/sched.h>`? Because it is desired to abstract away the scheduler code and provide only certain interfaces to the rest of the kernel. Placing the runqueue code in a header file would allow code outside of the scheduler to get at the runqueues, and this is not desired.

Let's look at the structure, with comments describing each field:

```
struct runqueue {
        spinlock_t  lock;   /* spin lock that protects this runqueue */
        unsigned long  nr_running;  /* number of runnable tasks */
        unsigned long  nr_switches; /* context switch count */
        unsigned long  expired_timestamp; /* time of last array swap */
        unsigned long  nr_uninterruptible;   /* uninterruptible tasks */
        unsigned long long timestamp_last_tick; /*last scheduler tick */
        struct task_struct  *curr;     /* currently running task */
        struct task_struct  *idle;     /* this processor's idle task */
        struct mm_struct    *prev_mm;  /* mm_struct of last ran task */
        struct prio_array   *active;   /* active priority array */
        struct prio_array   *expired;  /* the expired priority array */
        struct prio_array   arrays[2]; /* the actual priority arrays */
        struct task_struct  *migration_thread; /* migration thread */
        struct list_head    migration_queue;   /* migration queue*/
        atomic_t        nr_iowait; /* number of tasks waiting on I/O */
#ifdef CONFIG_SMP
...
};
```

<< Same on 2.6.37 below >>

```
struct rq {
 spinlock_t lock;

 /*
 * nr_running and cpu_load should be in the same cacheline because
 * remote CPUs use both these fields when doing load calculation.
 */
 unsigned long nr_running;
 unsigned long raw_weighted_load;
#ifdef CONFIG_SMP
        unsigned long cpu_load[3];
#endif
  unsigned long long nr_switches;

 /*
 * This is part of a global counter where only the total sum
 * over all CPUs matters. A task can increase this counter on
 * one CPU and if it got migrated afterwards it may decrease
 * it on another CPU. Always updated under the runqueue lock:
```

```
*/
unsigned long nr_uninterruptible;

unsigned long expired_timestamp;
/* Cached timestamp set by update_cpu_clock() */
unsigned long long most_recent_timestamp;
struct task_struct *curr, *idle;
unsigned long next_balance;
struct mm_struct *prev_mm;
struct prio_array *active, *expired, arrays[2];
int best_expired_prio;
...
};
```

Because runqueues are the core data structure in the scheduler, a group of macros are used to obtain the runqueue associated with a given processor or process. The macro `cpu_rq(processor)` returns a pointer to the runqueue associated with the given processor; the macro `this_rq()` returns the runqueue of the current processor; and the macro `task_rq(task)` returns a pointer to the runqueue on which the given task is queued.

...

The Priority Arrays

Each runqueue contains two priority arrays, the active and the expired array. Priority arrays are defined in `kernel/sched.c` as `struct prio_array`. Priority arrays are the data structures that provide `O(1)` scheduling. Each priority array contains one queue of runnable processes per priority level. These queues contain lists of the runnable processes at each priority level. The priority arrays also contain a priority bitmap used to efficiently discover the highest-priority runnable task in the system.

```
struct prio_array {
        int               nr_active; /* number of tasks in the queues */
        unsigned long     bitmap[BITMAP_SIZE];  /* priority bitmap */
        struct list_head  queue[MAX_PRIO];      /* priority queues */
};
```

`MAX_PRIO` is the number of priority levels on the system. By default, this is 140. Thus, there is one `struct list_head` for each priority.

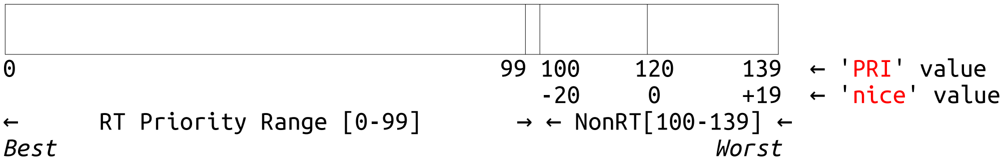
<code>

```
/*
 * Priority of a process goes from 0..MAX_PRIO-1, valid RT
 * priority is 0..MAX_RT_PRIO-1, and SCHED_NORMAL tasks are
 * in the range MAX_RT_PRIO..MAX_PRIO-1. Priority values
 * are inverted: lower p->prio value means higher priority.
 *
 * The MAX_USER_RT_PRIO value allows the actual maximum
 * RT priority to be separate from the value exported to
 * user-space.  This allows kernel threads to set their
 * priority to a value higher than any user task. Note:
 * MAX_RT_PRIO must not be smaller than MAX_USER_RT_PRIO.
 */

#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO

#define MAX_PRIO                (MAX_RT_PRIO + 40)

#define rt_task(p)              (unlikely((p)->prio < MAX_RT_PRIO))
…
```

</code> *<< kernel ver 2.6.11 >>*

<<

This is a kernel mapping of the user-space 'nice' value priority:



>>

BITMAP_SIZE is the size that an array of unsigned long typed variables would have to be to provide one bit for each valid priority level. With 140 priorities and 32-bit words, this is five. Thus, bitmap is an array with five elements and a total of 160 bits.

Each priority array contains a bitmap field that has at least one bit for every priority on the system. Initially, all the bits are zero. When a task of a given priority becomes runnable (that is, its state is set to TASK_RUNNING), the corresponding bit in the (active priority array's) bitmap is set to one. For example, if a task with priority seven is runnable, then bit seven is set. Finding the highest priority task on the system is therefore only a matter of finding the first set bit in the bitmap. Because the number of priorities is static, the time to complete this search is constant and unaffected by the number of running processes on the system.

Furthermore, each supported architecture in Linux implements a fast find first set algorithm to quickly search the bitmap. This method is called sched_find_first_bit(). Many architectures provide a find-first-set instruction that operates on a given word[4]. On these systems, finding the first set bit is as trivial as executing this instruction at most a couple of times.

[4] On the x86 architecture, this instruction is called `bsfl`. On PPC, `cntlzw` is used for this purpose.

Each priority array also contains an array named `queue` of `struct list_head` queues, one queue for each priority. Each list corresponds to a given priority and in fact contains all the runnable processes of that priority that are on this processor's runqueue. Finding the next task to run is as simple as selecting the next element in the list. Within a given priority, tasks are scheduled round robin.

The priority array also contains a counter, `nr_active`. This is the number of runnable tasks in this priority array.

<<

SIDEBAR

Where in the code are the priority arrays updated?

`kernel/sched.c:`

…

```
/*
 * Adding/removing a task to/from a priority array:
 */
static void dequeue_task(struct task_struct *p, prio_array_t *array)
{
        array->nr_active--;
        list_del(&p->run_list);
        if (list_empty(array->queue + p->prio))
                __clear_bit(p->prio, array->bitmap);
}

static void enqueue_task(struct task_struct *p, prio_array_t *array)
{
        sched_info_queued(p);
        list_add_tail(&p->run_list, array->queue + p->prio);
        __set_bit(p->prio, array->bitmap);
        array->nr_active++;
        p->array = array;
}
```

…

Kernel ver 2.6.11

>>

Recalculating Timeslices

Many operating systems (older versions of Linux included) have an explicit method for recalculating each task's timeslice when they have all reached zero. Typically, this is implemented as a loop over each task, such as

```
for (each task on the system) {
        recalculate priority
        recalculate timeslice
}
```

The priority and other attributes of the task are used to determine a new timeslice. This approach has some problems:

- It potentially can take a long time. Worse, it scales `O(n)` for `n` tasks on the system.
- The recalculation must occur under some sort of lock protecting the task list and the individual process descriptors. This results in high lock contention.
- The nondeterminism of a randomly occurring recalculation of the timeslices is a problem with deterministic real-time programs.
- It is just gross (which is a quite legitimate reason for improving something in the Linux kernel).

The new Linux scheduler alleviates the need for a recalculate loop. Instead, it maintains two priority arrays for each processor: both an active array and an expired array. The active array contains all the tasks in the associated runqueue that have timeslice left. The expired array contains all the tasks in the associated runqueue that have exhausted their timeslice. When each task's timeslice reaches zero, its timeslice is recalculated before it is moved to the expired array. Recalculating all the timeslices is then as simple as just switching the active and expired arrays. Because the arrays are accessed only via pointer, switching them is as fast as swapping two pointers. This is performed in `schedule()`:

```
struct prio_array *array = rq->active;
if (!array->nr_active) {
        rq->active = rq->expired;
        rq->expired = array;
}
```

This swap is a key feature of the new `O(1)` scheduler. Instead of recalculating each processes priority and timeslice all the time, the `O(1)` scheduler performs a simple two-step array swap. This resolves the previously discussed problems.

schedule()

The act of picking the next task to run and switching to it is implemented via the `schedule()` function. This function is called explicitly by kernel code that wants to sleep and it is invoked whenever a task is to be preempted. The `schedule()` function is run independently by each processor. Consequently, each CPU makes its own decisions on what process to run next.

The `schedule()` function is relatively simple for all it must accomplish. The following code determines the highest priority task:

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;

prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
```

First, the active priority array is searched to find the first set bit. This bit corresponds to the highest priority task that is runnable. Next, the scheduler selects the first task in the list at that priority. This is the highest priority runnable task on the system and is the task the scheduler will run.

If `prev` does not equal `next`, then a new task has been selected to run. The function `context_switch()` is called to switch from `prev` to `next`. Context switching is discussed in a subsequent section.

Two important points should be noted from the previous code. First, it is very simple and consequently quite fast. Second, the number of processes on the system has no effect on how long this code takes to execute. There is no loop over any list to find the most suitable process. In fact, nothing affects how long the `schedule()` code takes to find a new task. It is constant in execution time.

[FYI, OPTIONAL]

Calculating Priority and Timeslice

At the beginning of this chapter, you saw how priority and timeslice are used to influence the decisions that the scheduler makes. Additionally, you learned about I/O-bound and processor-bound tasks and why it is beneficial to boost the priority of interactive tasks. Now it's time to look at the actual code that implements this design.

Processes have an initial priority that is called the nice value. This value ranges from –20 to +19 with a default of zero. Nineteen is the lowest and –20 is the highest priority.

```
<<
$ ps -el|head -n1
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY      TIME CMD
$ gedit&
[1] 3705
$ ps -el|grep gedit
```

```
0 S  1001  3705  3555  9  80  0 - 23213 poll_s pts/2 00:00:00 gedit
$ renice 5 3705
3705: old priority 0, new priority 5
$ ps -el|grep gedit
0 S  1001  3705  3555  1  85 5 - 23213 poll_s pts/2 00:00:00 gedit
$ renice 0 3705
3705: old priority 5, new priority 0
$ ps -el|grep gedit
0 S  1001  3705  3555  1  80 0 - 23213 poll_s pts/2  00:00:00 gedit
$ renice -3 3705
renice: 3705: setpriority: Permission denied
$
>>
```

This value is stored in the `static_prio` member of the process's `task_struct`. The variable is called the static priority because it does not change from what the user specifies. The scheduler, in turn, bases its decisions on the dynamic priority that is stored in `prio`. The dynamic priority is calculated as a function of the static priority and the task's interactivity.

<<

i.e. ***dynamic priority* = *f(static priority, interactivity) ;***

where static priority is the *nice* value.

>>

The method `effective_prio()` returns a task's dynamic priority.

`activate_task() → recalc_task_prio() → effective_prio()`

Of course, the scheduler does not magically know whether a process is interactive. It must use some heuristic that is capable of accurately reflecting whether a task is I/O bound or processor bound. The most indicative metric is how long the task sleeps. If a task spends most of its time asleep, then it is I/O bound. If a task spends more time runnable than sleeping, it is certainly not interactive. This extends to the extreme: A task that spends nearly all the time sleeping is completely I/O bound, whereas a task that spends nearly all its time runnable is completely processor bound.

To implement this heuristic, Linux keeps a running tab on how much time a process is spent sleeping versus how much time the process spends in a runnable state. This value is stored in the `sleep_avg` member of the `task_struct`. It ranges from zero to `MAX_SLEEP_AVG`, which defaults to 10 milliseconds. When a task becomes runnable after sleeping, `sleep_avg` is incremented by how long it slept, until the value reaches `MAX_SLEEP_AVG`. For every timer tick the task runs, `sleep_avg` is decremented until it reaches zero.

This metric is surprisingly accurate. It is computed based not only on how long the task sleeps but also on how little it runs. Therefore, a task that spends a great deal of time sleeping, but also continually exhausts its timeslice, will not be awarded a huge bonus—the metric works not just to award interactive tasks but also to punish processor-bound tasks. It is also not vulnerable to abuse. A task that receives a boosted priority and timeslice quickly loses the bonus if it turns around and hogs the processor. Finally, the metric provides quick response. A newly created interactive process quickly receives a large `sleep_avg`. Despite this, because the bonus or penalty is applied against the initial nice value, the user can still influence the system's scheduling decisions by changing the process's nice value.

Timeslice, on the other hand, is a much simpler calculation. It is based on the static priority. When a process is first created, the new child and the parent split the parent's remaining timeslice. This provides fairness and prevents users from forking new children to get unlimited timeslice.

<<

timeslice = f(static priority) ;

where static priority is the *nice* value.

>>

The scheduler provides one additional aide to interactive tasks: If a task is sufficiently interactive, when it exhausts its timeslice it will not be inserted into the expired array, but instead reinserted back into the active array. Recall that timeslice recalculation is provided via the switching of the active and the expired arrays.

Normally, as processes exhaust their timeslices, they are moved from the active array to the expired array. When there are no more processes in the active array, the two arrays are switched: The active becomes the expired, and the expired becomes the active.

This provides `O(1)` timeslice recalculation. It also provides the possibility that an interactive task can become runnable but fail to run again until the array switch occurs because the task is stuck in the expired array. Reinserting interactive tasks back into the active array alleviates this problem. The task does not run immediately, but is scheduled round robin with the other tasks at its priority. The logic to provide this feature is implemented in `scheduler_tick()`, which is called via the timer interrupt (discussed in Chapter 10, "Timers and Time Management"):

```
<Timer Interrupt> →  tick_handle_periodic → tick_periodic → update_process_times
→ scheduler_tick

...
struct task_struct *task;
struct runqueue *rq;
task = current;
rq = this_rq();
```

```
if (!--task->time_slice) {    << timeslice exhausted! >>
        if (!TASK_INTERACTIVE(task) || EXPIRED_STARVING(rq))
                enqueue_task(task, rq->expired);
        else
                enqueue_task(task, rq->active);
}
```

First, the code decrements the process's timeslice and checks whether it is now zero. If it is, the task is expired and it needs to be inserted into an array, so this code first checks whether the task is interactive via the `TASK_INTERACTIVE()` macro. This macro computes whether a task is "interactive enough" based on its nice value. The lower the nice value (the higher the priority) the less interactive a task needs to be. A nice +19 task can never be interactive enough to be reinserted. Conversely, a nice –20 task would need to be a heavy processor hog not to be reinserted. A task at the default nice value, zero, needs to be relatively interactive to be reinserted, but it is not too difficult.

Next, the `EXPIRED_STARVING()` macro checks whether there are processes on the expired array that are starving—that is, if the arrays have not been switched in a relatively long time. If they have not been switched recently, reinserting the current task into the active array further delays the switch, additionally starving the tasks on the expired array. If this is not the case, the process can be inserted into the active array. Otherwise, it is inserted into the expired array, which is the normal practice.