More on Linux scheduling

CSE 536 Spring 2026

jedimaestro@asu.edu

Old: Completely Fair Scheduler (CFS)

New: Earliest Eligible Virtual Deadline First (EEVDF)

https://web.archive.org/web/20260225090858/https://citeseerx.ist.psu.edu/document?doi=805acf7726282721504c8f00575d91ebfd750564&repid=rep1&type=pdf

Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*

Ion Stoica, Hussein Abdel-Wahab
Department of Computer Science, Old Dominion University
Norfolk, Virginia, 23529-0162
{stoica, wahab}@cs.odu.edu

TR-95-22

Priority (PR) *vs.* niceness (NI)

- Niceness comes from the user space (like `nice` and `renice`) and ranges from -20 to 19
  - Positive niceness is less priority
  - Negative niceness requires root
- For normal processes:
  - PR = 20 + NI
- Real-time processes have PR ranging from -100 to -1
  - Think cars, industrial controls, telecom equipment, multimedia, kernel threads

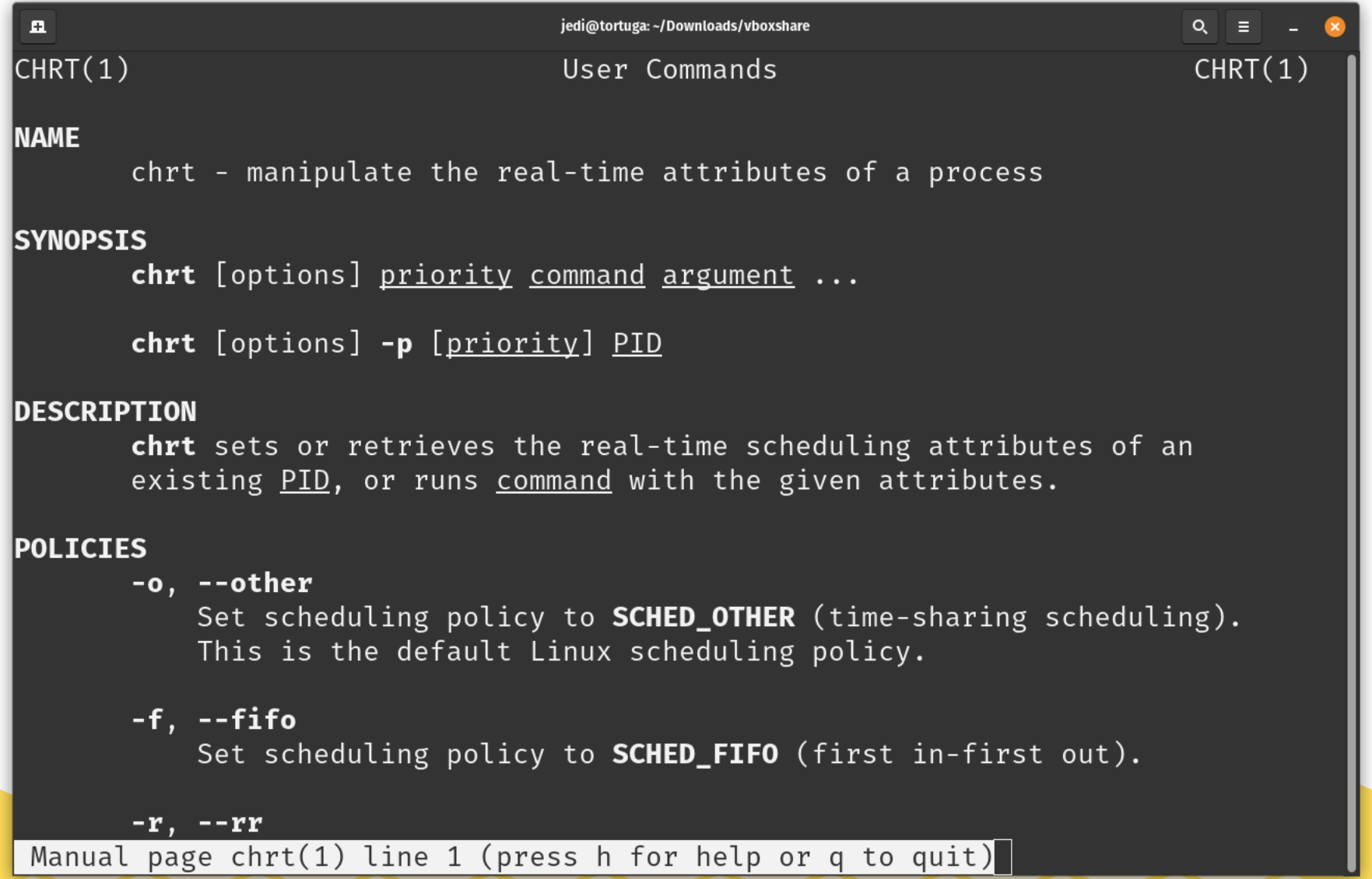
Task Type	User Scale	Internal Kernel Priority
Real-Time	1 to 99	0 to 98
Standard (Nice)	-20 to 19	100 to 139

- **Hard vs. Soft RT:** While standard Linux is best-effort, a properly configured PREEMPT_RT kernel provides guaranteed response times.

1. Scheduling Policies

The kernel uses different "scheduling classes" that fundamentally change how priority is handled. Most standard apps use `SCHED_OTHER`, but others can be assigned:

- **Real-Time Policies (`SCHED_FIFO`, `SCHED_RR`)**: These processes use a separate priority scale (1 to 99) and always run before any standard process, regardless of its niceness. You can set these using the `chrt` command.
- **Deadline Policy (`SCHED_DEADLINE`)**: This is the highest priority class in Linux. It ignores traditional priority numbers entirely and instead prioritizes tasks based on their specific timing deadlines.
- **Idle Policy (`SCHED_IDLE`)**: This is for extremely low-priority background jobs that should only run when the CPU has absolutely nothing else to do. Linux Foundation +5


jedi@tortuga: ~/Downloads/vboxshare
CHRT(1) User Commands CHRT(1)
NAME
chrt - manipulate the real-time attributes of a process
SYNOPSIS
chrt [options] priority command argument ...
chrt [options] -p [priority] PID
DESCRIPTION
chrt sets or retrieves the real-time scheduling attributes of an existing PID, or runs command with the given attributes.
POLICIES
-o, --other
Set scheduling policy to SCHED_OTHER (time-sharing scheduling). This is the default Linux scheduling policy.
-f, --fifo
Set scheduling policy to SCHED_FIFO (first in-first out).
-r, --rr
Manual page chrt(1) line 1 (press h for help or q to quit)


jedi@tortuga: ~/Downloads/vboxshare
jedi@tortuga:~/Downloads/vboxshare$ ps axo ni,pri,args | sort -n | less -S

“Hard” real time...

```
-   90 [irq/25-ACPI:Event]
-   90 [irq/26-ACPI:Event]
-   90 [irq/27-ACPI:Event]
-   90 [irq/28-ACPI:Event]
-   90 [irq/29-ACPI:Event]
-   90 [irq/30-ACPI:Event]
-   90 [irq/31-ACPI:Event]
-   90 [irq/32-ACPI:Event]
-   90 [irq/33-aerdrv]
-   90 [irq/34-aerdrv]
-   90 [irq/35-aerdrv]
-   90 [irq/36-aerdrv]
-   90 [irq/38-aerdrv]
-   90 [irq/9-acpi]
-   90 [watchdogd]
```

Soft real time...

```
-20  39 [kworker/u65:6-hci0]
-20  39 [kworker/u65:8-ttm]
-15  34 /usr/bin/pipewire
-15  34 /usr/bin/pipewire-pulse
-10  29 [krfcommd]
 -9  28 /usr/bin/wireplumber
 -3  22 /usr/bin/gnome-shell
 -3  22 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority>
  0  19 bash
  0  19 bash
  0  19 [cpuhp/0]
```

Normal...

```
  0   19 dbus-broker --log 4 --controller 10 --machine-id ce91caf6376626ac106d3b>
  0   19 dbus-broker --log 4 --controller 9 --machine-id ce91caf6376626ac106d3be>
  0   19 dbus-broker --log 4 --controller 9 --machine-id ce91caf6376626ac106d3be>
  0   19 [dmcrypt_write/252:0]
  0   19 [ecryptfs-kthread]
  0   19 [iprt-VBoxTscThread]
  0   19 [jbd2/dm-1-8]
  0   19 [kauditd]
```

Nice and background...

```
1  18 /usr/libexec/rtkit-daemon
5  14 [ksmd]
6  13 cat
6  13 cat
6  13 fusermount3 -o rw,nosuid,nodev,fsname=portal,auto_unmount,subtype=porta>
6  13 gdm-session-worker [pam/gdm-password]
6  13 gjs /usr/share/gnome-shell/extensions/ding@rastersoft.com/ding.js -E -P>
6  13 /opt/google/chrome/chrome
6  13 /opt/google/chrome/chrome_crashpad_handler --monitor-self --monitor-sel>
6  13 /opt/google/chrome/chrome_crashpad_handler --no-periodic-tasks --monito>
6  13 /opt/google/chrome/chrome --type=gpu-process --ozone-platform=x11 --cra>
6  13 /opt/google/chrome/chrome --type=renderer --crashpad-handler-pid=3665 -->
6  13 /opt/google/chrome/chrome --type=renderer --crashpad-handler-pid=3665 -->
6  13 /opt/google/chrome/chrome --type=renderer --crashpad-handler-pid=3665 -->
6  13 /opt/google/chrome/chrome --type=renderer --crashpad-handler-pid=3665 -->
```

```
12    7 /usr/libexec/switcheroo-control
12    7 /usr/libexec/udisks2/udisksd
12    7 /usr/libexec/upowerd
12    7 /usr/lib/system76-firmware/system76-firmware-daemon
12    7 /usr/sbin/acpid
12    7 /usr/sbin/atd -f
12    7 /usr/sbin/chronyd -F 1
12    7 /usr/sbin/chronyd -F 1
12    7 /usr/sbin/cron -f -P
12    7 /usr/sbin/cups-browsed
12    7 /usr/sbin/cupsd -l
12    7 /usr/sbin/gdm3
12    7 /usr/sbin/ModemManager
12    7 /usr/sbin/NetworkManager --no-daemon
12    7 /usr/sbin/rsyslogd -n -iNONE
```

AI Overview

The Earliest Eligible Virtual Deadline First (EEVDF) scheduler is a Linux kernel CPU algorithm (introduced in v6.6) that improves upon the CFS scheduler by balancing strict fairness with latency-sensitive task prioritization. It works by assigning a "virtual deadline" to tasks based on their lag (time they are owed vs. time received), prioritizing tasks that are behind on CPU time. Kernel docs +2

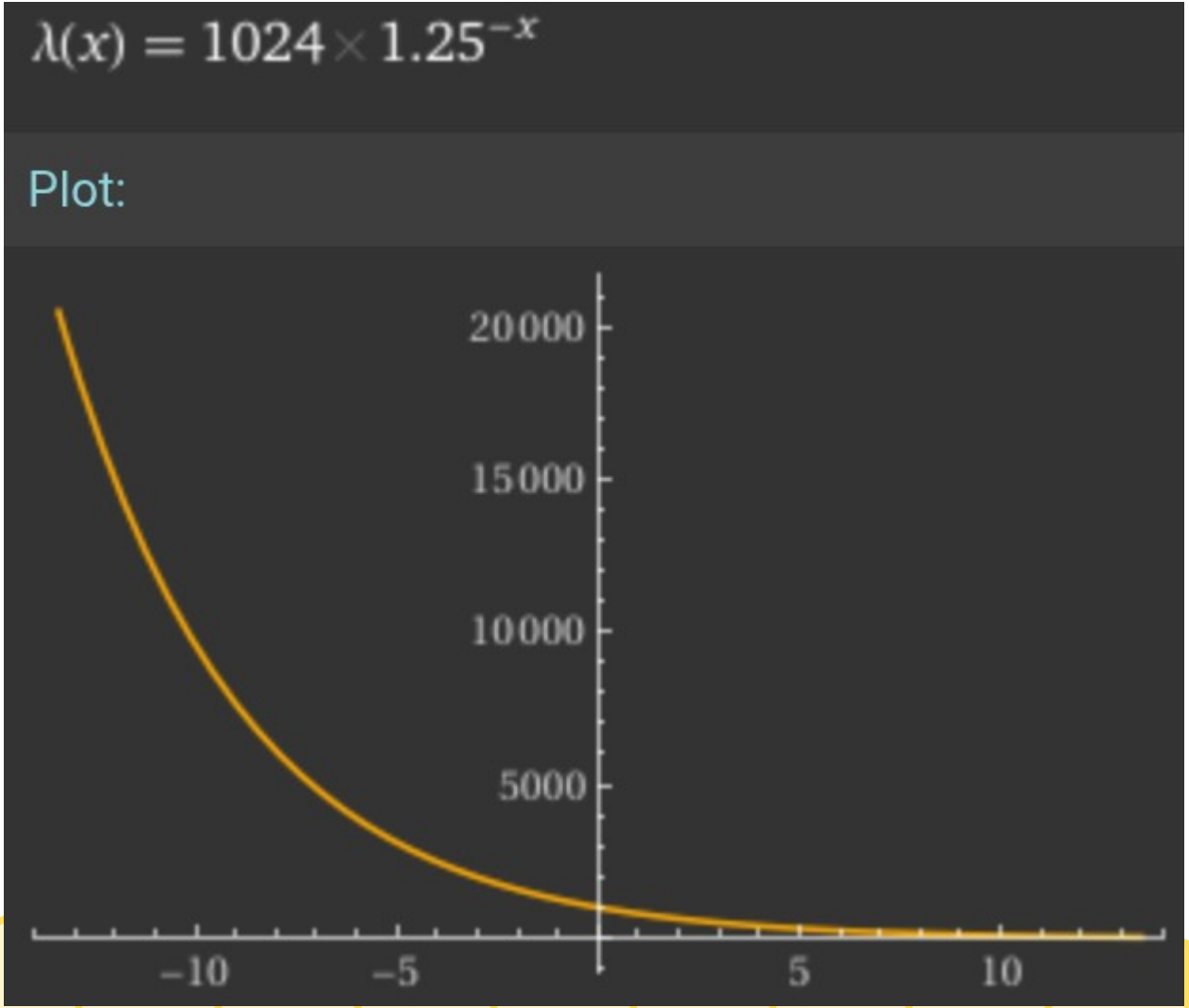
https://www.vittoriozaccaria.net/blog/notes-on-linux-eevdf

$$\lambda_i \simeq b^{-\nu_i} \text{ (b=1.25)}$$

$$w_i = 2^{10}\lambda_i$$

$$\lambda(x) = 1024 \times 1.25^{-x}$$

Plot:



https://chunyu.sh/blog/thinking-about-eevdf/

EEVDF, and well as it's predecessor, CFS rely on this concept of `vruntime`. This is the "current" virtualized time of a task. It is amount of CPU time given to a task. For example, given a task with niceness (think of niceness like priority levels) of 0.

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Ran for 1000 time units…

Niceness of 0 → 1000(1024/1024) = 1000 time units added

Niceness of -1 → 1000(1024/1277) = 802 time units added

Niceness of 19 → 1000(1024/15) = 68267 time units added

Niceness of -20 → 1000(1024/88761) = 12 time units added

- **CFS:** Uses `vruntime` and selects the task with the smallest `vruntime`.
- **EEVDF:** Uses `vruntime` to determine eligibility, then selects based on the lowest virtual deadline.
- **Result:** EEVDF allows interactive tasks to run more frequently in smaller, shorter bursts without giving them more total CPU time, directly solving latency issues without needing separate "latency-nice" configurations. » Linux Magazine +1

CFS time slice calculation

https://www.vittoriozaccaria.net/blog/notes-on-linux-eevdf

$$\tau_p = f(\nu_0, \ldots, \nu_p, \ldots, \nu_{n-1}, \bar{\tau}, \mu) \sim max(\frac{\lambda_p \bar{\tau}}{\sum \lambda_i}, \mu)$$

- $\bar{\tau}$ was called *schedule latency*. Configurable, default 6ms. It was the targeted time for each process to get a chance to run.
- μ is called *minimum granularity*. Configurable, default 0.75ms. This ensured that that every low-priority process got a minimum amount of CPU time.
- $\frac{\lambda_p}{\sum \lambda_i}$ is the *process share* of time.

$$\forall p,\ \Delta\rho_p = \frac{\tau_p}{\lambda_p} = \frac{\bar{\tau}\lambda_p}{\lambda_p \sum \lambda_i} = \frac{\bar{\tau}}{\sum \lambda_i}$$

CFS introduced a **virtual runtime** `(vruntime)` variable ρ_p

Common to CFS and EEVDF: vruntime

New to EEVDF: lag and virtual deadlines

EEVDF, like CFS, tracks the virtual runtime of each process, but introduces also the concept of **global virtual time** at time t:

$$\rho(t) = \frac{t}{\sum_p \lambda_p}$$

In Linux, there is not an explicit variable that tracks ρ. It is usually approximated with a weighted average of virtual runtimes of all running processes:

$$\bar{\rho}(t) = \frac{\sum_i \lambda_i \rho_i}{\sum_p \lambda_p}$$

For numerical stability the kernel stores $\bar{\rho}$ in two different variables called `avg_vruntime` and `avg_load` $\hat{\rho}, \Lambda$:

$$\hat{\rho} = \sum_i w_i * (\rho_i - \rho_{min}), \quad \Lambda = \sum_p w_p$$

Here, `min_vruntime` $\rho_{min}(t)$ is used to reduce the magnitudes involved in computations (it is always equal to the minimum value of runtime you can find in the run queue at any given moment). Implicitly the value of $\bar{\rho}$ could always be reconstructed from $\hat{\rho}, \Lambda$ and ρ_{min} as: $\bar{\rho} = \hat{\rho}/\Lambda + \rho_{min}$.

Process lag = Process current weighted vruntime - weighted average of every task's vruntime

$$VD = \text{Eligible Time} + \frac{\text{Remaining Slice}}{\text{Weight}}$$

As in CFS, tasks that are behind in virtual runtime must be given priority. In EEVDF, this means only *eligible* tasks (those with positive lag) can be scheduled, which ensures fairness.

To compute eligibility for process p, the kernel measures the actual process **lag**, i.e., how much actual time do we owe to a process p at time t relative to the current virtual time:

$$\alpha_p(t) = w_p(\bar{\rho}(t) - \rho_p(t))$$

A task p is considered eligible when $\rho_p(t) \leq \bar{\rho}(t)$. The kernel tries hard to avoid loss in precision to compute if a task is eligible and in particular the actual formula used is is

$$(\rho_p - \rho_{min})\Lambda \leq \hat{\rho}$$

Ensuring latency-sensitive operation

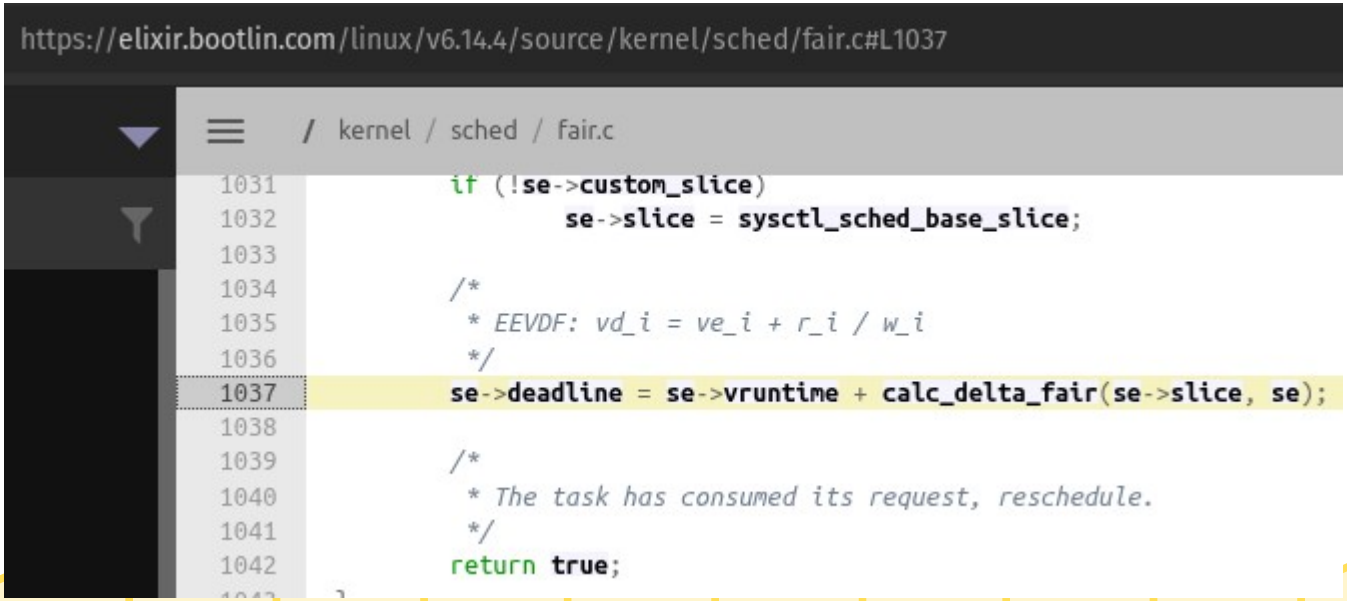
To prioritize among eligible tasks, EEVDF uses **virtual deadlines** to decide which process to run next. The virtual deadline represents the urgency with which a process should be scheduled, and the process with the **earliest** (smallest) virtual deadline δ is chosen to run first. The deadline δ is computed as

$$\delta_p = \frac{\tau_p}{\lambda_p} + \rho_p$$

Unlike CFS, the timeslice τ_p of the process can be a default timeslice or a custom one set via the `sched_setattr()` system call. The smaller the timeslice, the highest priority will be given to that eligible task.

Is ρ_p a type-o?

Think of "eligible time" as the moment a task has officially "caught up" to its fair share of the CPU.



Caveats

- Control groups (cgroups) limit CPU usage (and other resources)
- Priority inheritance
  - Remember Linux's mutex's?
- We've only considered the CPU (`ionice` for I/O)
- Latency nice
  - `sched_setattr()`

Feature	Old CFS	New EEVDF
New Task Baseline	Initialized to `min_vruntime`	Initialized to **Zero Lag**
Selection Logic	Lowest `vruntime`	Earliest **Virtual Deadline** among eligible tasks
Latency Handling	Heuristics (e.g., "sleeper bonus")	Algorithmic (via time slice & deadline math)


jedi@tortuga: ~/Downloads/vboxshare
jedi@tortuga:~/Downloads/vboxshare$ sudo ps axHo policy,pid,comm,nice,rtprio | sort | less