Linux scheduling

CSE 536 Fall 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

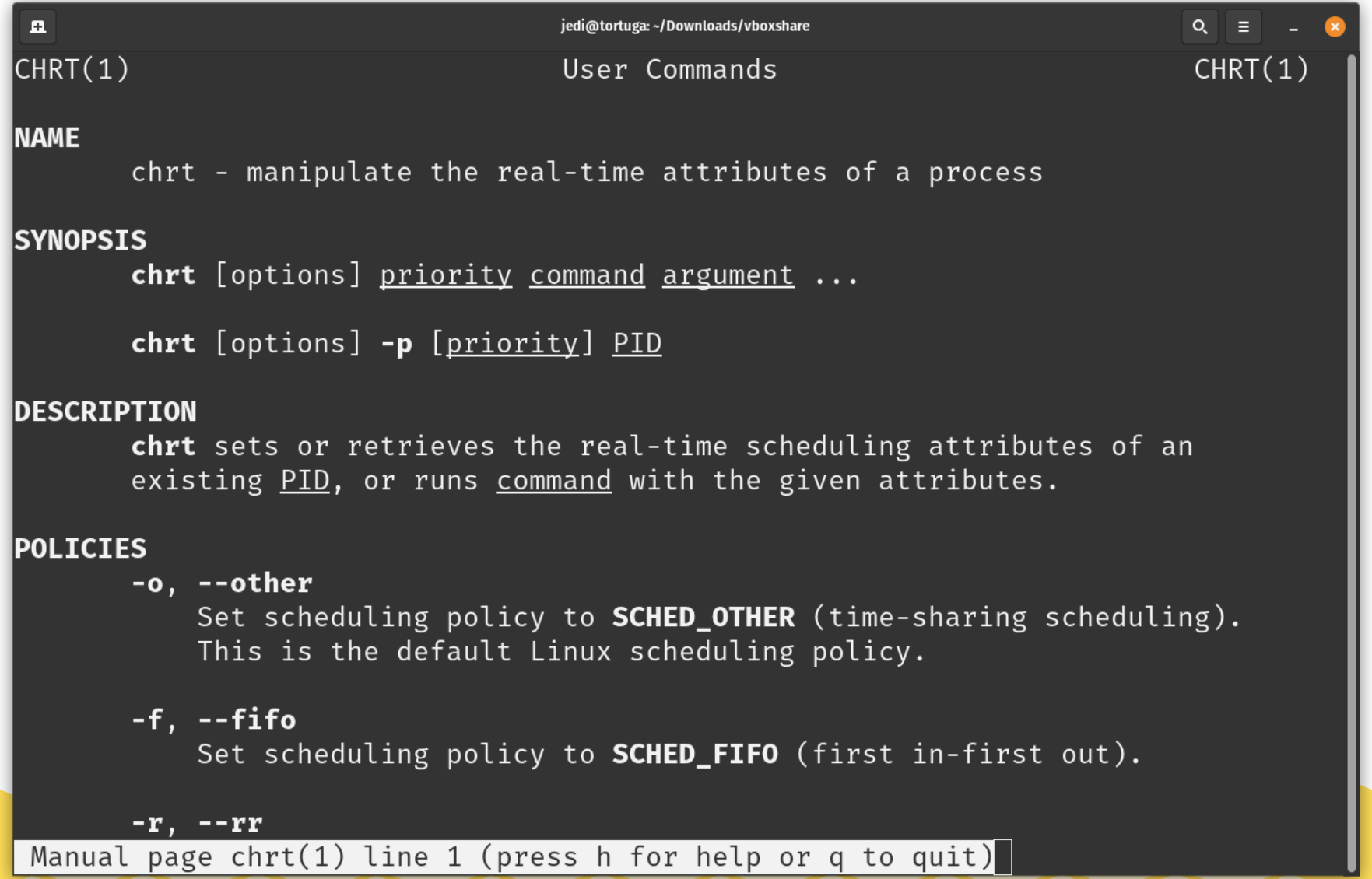
Task Type	User Scale	Internal Kernel Priority
Real-Time	1 to 99	0 to 98
Standard (Nice)	-20 to 19	100 to 139

- **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

```
-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]
```

```
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]
```

```
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
```

Earliest Eligible Virtual Deadline First

- Lag is how far behind on its fair CPU time a process is compared to others
  - Average lag is 0
- Lag ≥ 0 makes a process eligible (different from CFS)
  - At least one process is always eligible
- Among eligible processes, choose the soonest virtual deadline (also different from CFS)
  - We'll see how this works later in these slides...

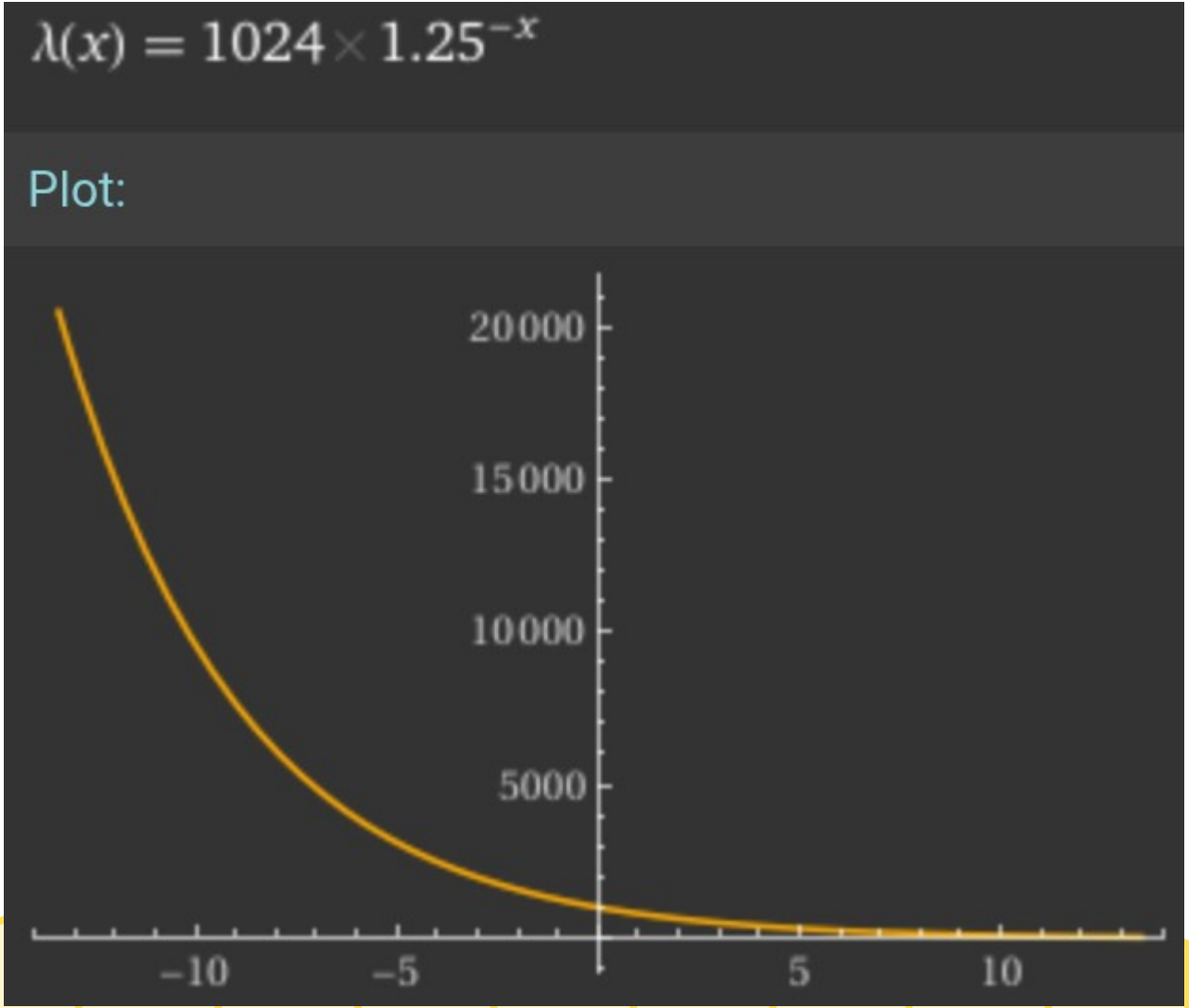
https://www.vittoriozaccaria.net/blog/notes-on-linux-eevdf

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

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

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

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.

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

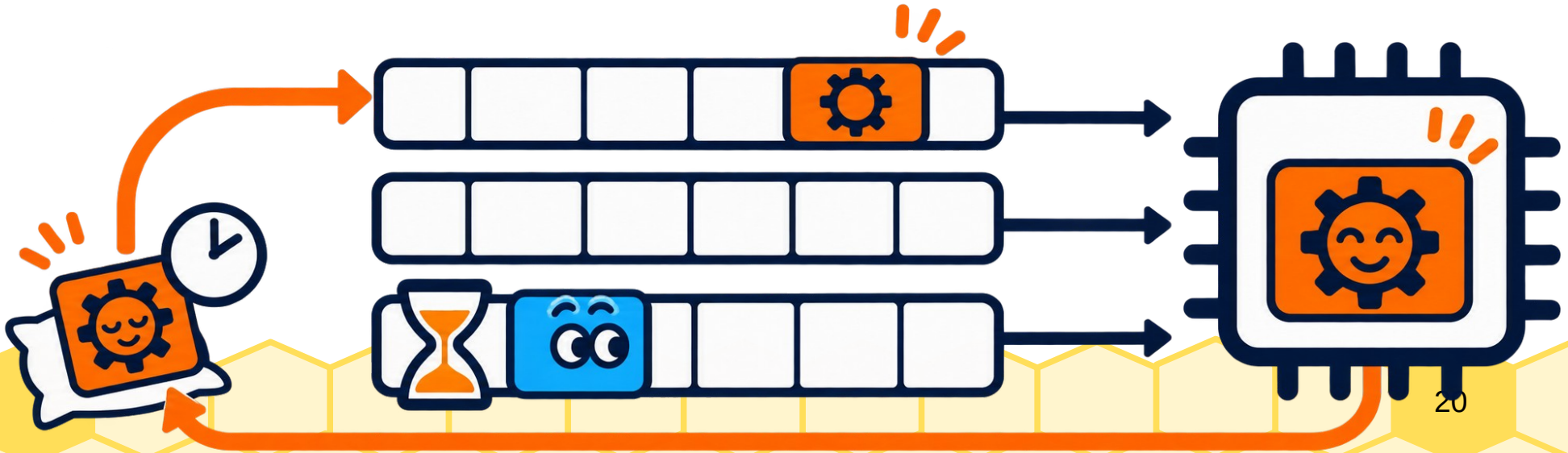
CFS introduced a **virtual runtime** `(vruntime)` variable ρ_p

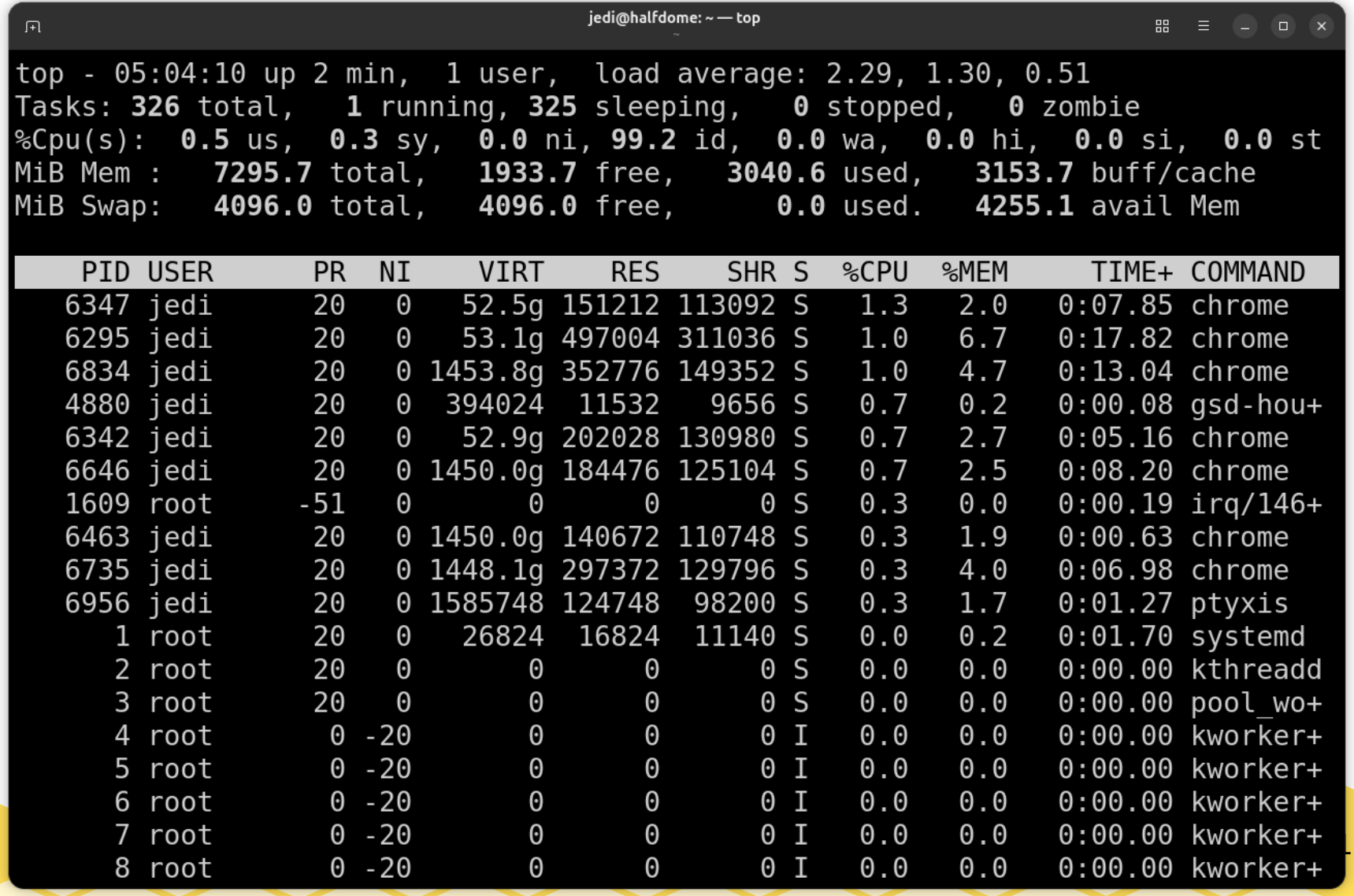
This is the total increase in its `vruntime` if it consumes its full CFS slice:

$$\Delta\rho_p = \frac{\tau_p}{\lambda_p} = \tau_p \frac{1024}{w_p}.$$

Scheduler	Does sleeping automatically boost its numeric priority?	What changes its scheduling order?
Old Linux O(1) scheduler	Yes, through interactivity heuristics	Dynamic priority
CFS	No	Virtual runtime and sleeper placement
EEVDF	No	Eligibility and virtual deadline

Solaris Time-Sharing (TS) scheduler

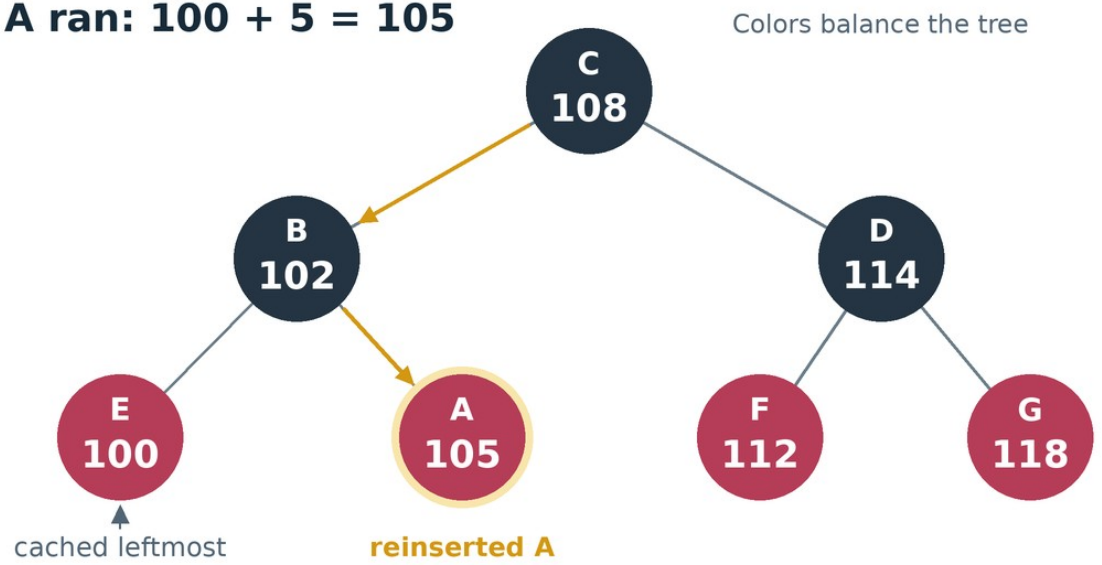



jedi@halfdome: ~ — top
top - 05:04:10 up 2 min, 1 user, load average: 2.29, 1.30, 0.51
Tasks: 326 total,   1 running, 325 sleeping,   0 stopped,   0 zombie
%Cpu(s): 0.5 us, 0.3 sy, 0.0 ni, 99.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem :   7295.7 total,   1933.7 free,   3040.6 used,   3153.7 buff/cache
MiB Swap: 4096.0 total,   4096.0 free,   0.0 used.   4255.1 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
6347 jedi 20 0 52.5g 151212 113092 S 1.3 2.0 0:07.85 chrome
6295 jedi 20 0 53.1g 497004 311036 S 1.0 6.7 0:17.82 chrome
6834 jedi 20 0 1453.8g 352776 149352 S 1.0 4.7 0:13.04 chrome
4880 jedi 20 0 394024 11532 9656 S 0.7 0.2 0:00.08 gsd-hou+
6342 jedi 20 0 52.9g 202028 130980 S 0.7 2.7 0:05.16 chrome
6646 jedi 20 0 1450.0g 184476 125104 S 0.7 2.5 0:08.20 chrome
1609 root -51 0 0 0 0 S 0.3 0.0 0:00.19 irq/146+
6463 jedi 20 0 1450.0g 140672 110748 S 0.3 1.9 0:00.63 chrome
6735 jedi 20 0 1448.1g 297372 129796 S 0.3 4.0 0:06.98 chrome
6956 jedi 20 0 1585748 124748 98200 S 0.3 1.7 0:01.27 ptyxis
1 root 20 0 26824 16824 11140 S 0.0 0.2 0:01.70 systemd
2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
3 root 20 0 0 0 0 S 0.0 0.0 0:00.00 pool_wo+
4 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker+
5 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker+
6 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker+
7 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker+
8 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker+

CFS: virtual runtime and the red-black tree

One runqueue, sorted by se->vruntime. Smaller values run sooner.

A ran: 100 + 5 = 105



Insertion compares 105 with 108, then 102.

Operation Worst case; n = nodes in this tree	Cost
Charge running task	O(1)
Read cached minimum	O(1)
Update min_vruntime	O(1)
Compute wakeup clamp	O(1)
Insert / reinsert	O(lg n)
Remove for dispatch	O(lg n)

Insertion / removal include rebalancing.

min_vruntime: monotonic reference

Uses current task + cached minimum

Off the CPU, still runnable

Running task stays outside the tree.

```
v += CPU_time × NICE_0_LOAD / weight
```

Reinsert its updated v. Waiting values stay fixed.

Returning from sleep

```
v = max(v_old, min_vruntime − bonus)
```

Example: max(80, 100 − 5) = 95

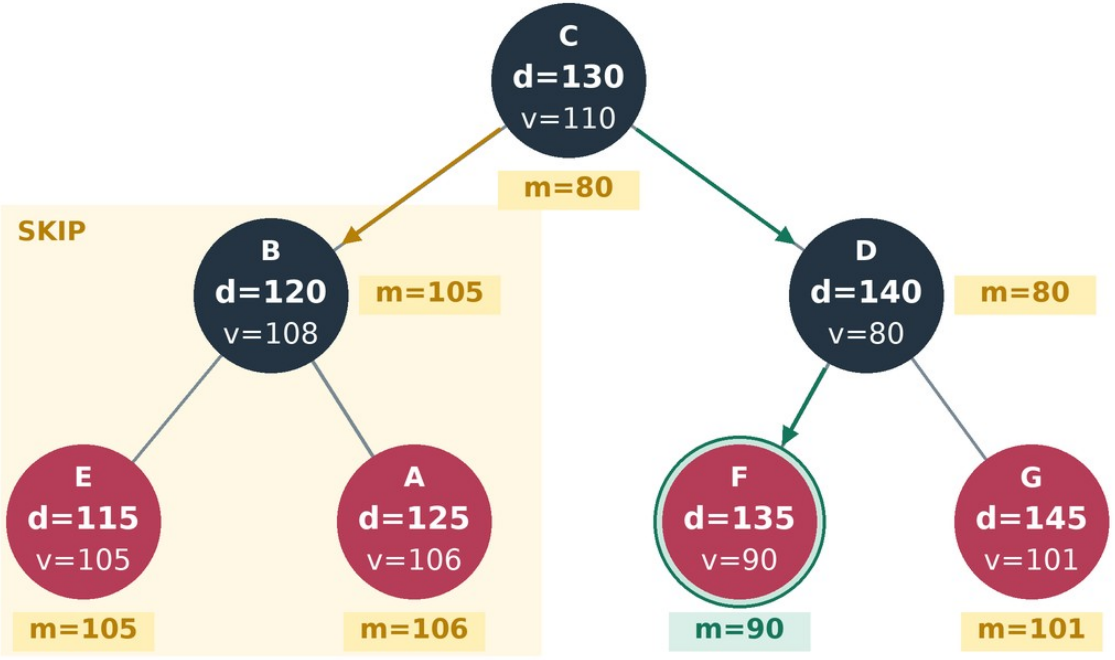
Insert before 100. Sleeper credit is capped.

EEVDF: sort by deadline, prune by virtual runtime

`Sort key: se->deadline`

`Cache: se->min_vruntime = subtree min v`

Equal weights: V = 100. Eligible if v ≤ 100.



d: deadline v: vruntime m: subtree minimum v

Find the earliest eligible deadline

1 Left subtree: m = 105 > 100
Skip all 3 tasks.

2 Root: v = 110 > 100. Skip.

3 Go right, then left.
Pick F: d = 135, v = 90.

Tree operation	Cost
Worst case, per runqueue; n = nodes	
Charge task / update V	**O(1)**
Check task or subtree	**O(1)**
Read earliest deadline	**O(1)**
Find earliest eligible	**O(lg n)**
Insert / remove + caches	**O(lg n)**

Cached leftmost eligible? O(1) fast path.

Maintain each changed ancestor’s cache

```
m = min(v, m_left, m_right)
```

V changes? No full-tree scan.

Compare current V against saved minima.

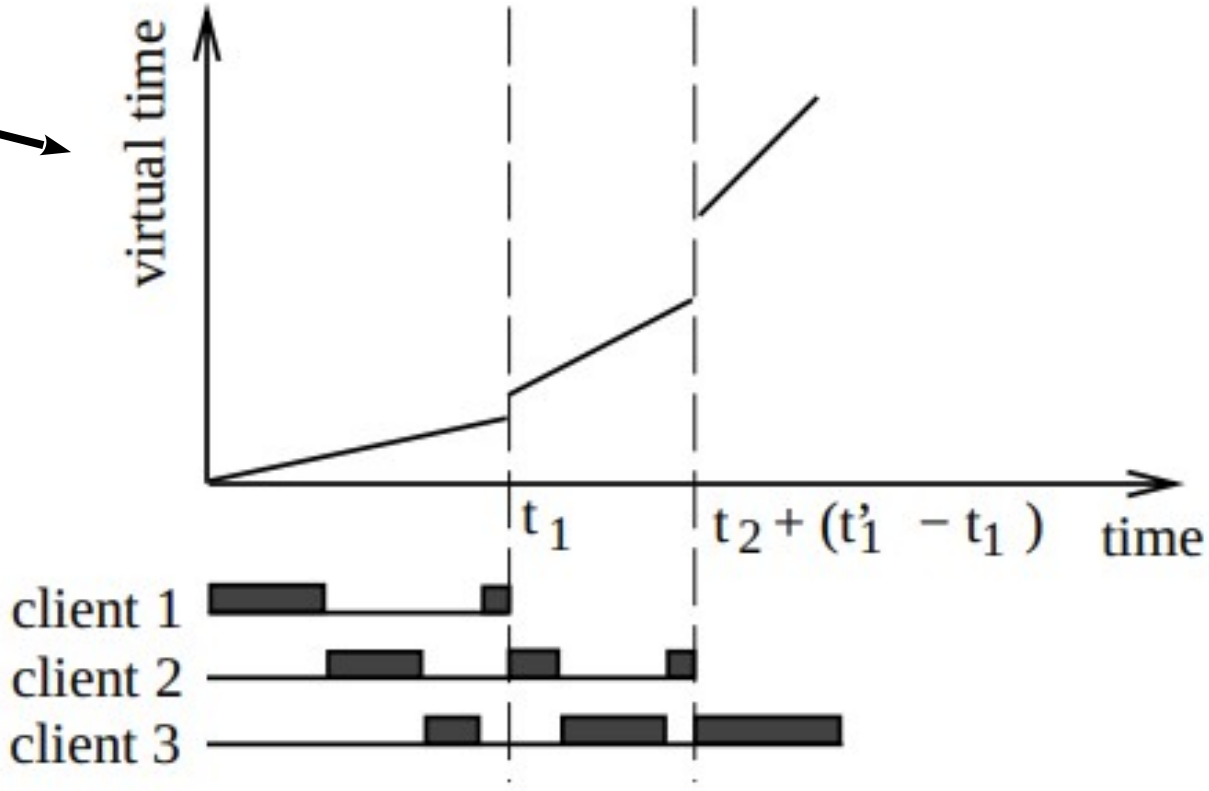
Common to CFS and EEVDF: vruntime

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:

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

shared/
global



http://www.cs.unc.edu/~jeffay/papers/RTSS-96a.pdf

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: $\bar{\rho} = \hat{\rho}/\Lambda + \rho_{min}$.

The main issue is the changing number and weight of runnable tasks—even when the CPU stays 100% busy. For example:

- With **two equally weighted tasks**, each deserves **0.5 ms** of every millisecond.
- With **four equally weighted tasks**, each deserves **0.25 ms**.

The ordinary clock advances by the same millisecond in both cases, but each task's fair entitlement advances differently. Changing priorities changes those entitlements too.

Idleness matters as well: wall-clock time continues passing when no CPU service is being delivered.

Lag and virtual deadline

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

$$\text{virtual lag}_p = \underbrace{V}_{\text{runqueue's weighted-average vruntime}} - \underbrace{v_p}_{\text{task's vruntime}} .$$

$$d_p = v_p + \frac{\text{remaining CPU-time budget}}{\lambda_p}$$

Quantity	Meaning
v_p, stored in `se->vruntime`	The task's priority-normalized CPU progress.
`se->vlag`	A saved, bounded version of $V - v_p$, used to preserve its balance across sleep and wakeup.

- While the task **waits**, v_p stays fixed. It becomes eligible if V catches up to it.
- While the task **runs**, v_p advances. It can move ahead of V, becoming ineligible until the reference catches up again.

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

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

CFS

`se->vruntime`

Implicit credit / debt

WHILE ASLEEP

vruntime pauses
Others advance

− debt → **0** → **+ credit**

ON WAKEUP

“Sleeper bonus”

Cap credit on wakeup

```
v = max(v_old, min_vruntime - bonus)
```

Misleading name

EEVDF

`se->vlag`

Saved virtual lag

WHILE ASLEEP

+ credit → **saved** (bounded)

− debt → **0** as others run

ON WAKEUP

Restore saved lag

No sleeper bonus

Sleep earns no new credit

Apps need no “interactive” label: Linux observes blocking and waking.
Apps can also request shorter EEVDF slices: sched_setattr(), sched_attr.sched_runtime (ns).

Ensuring latency-sensitive operation

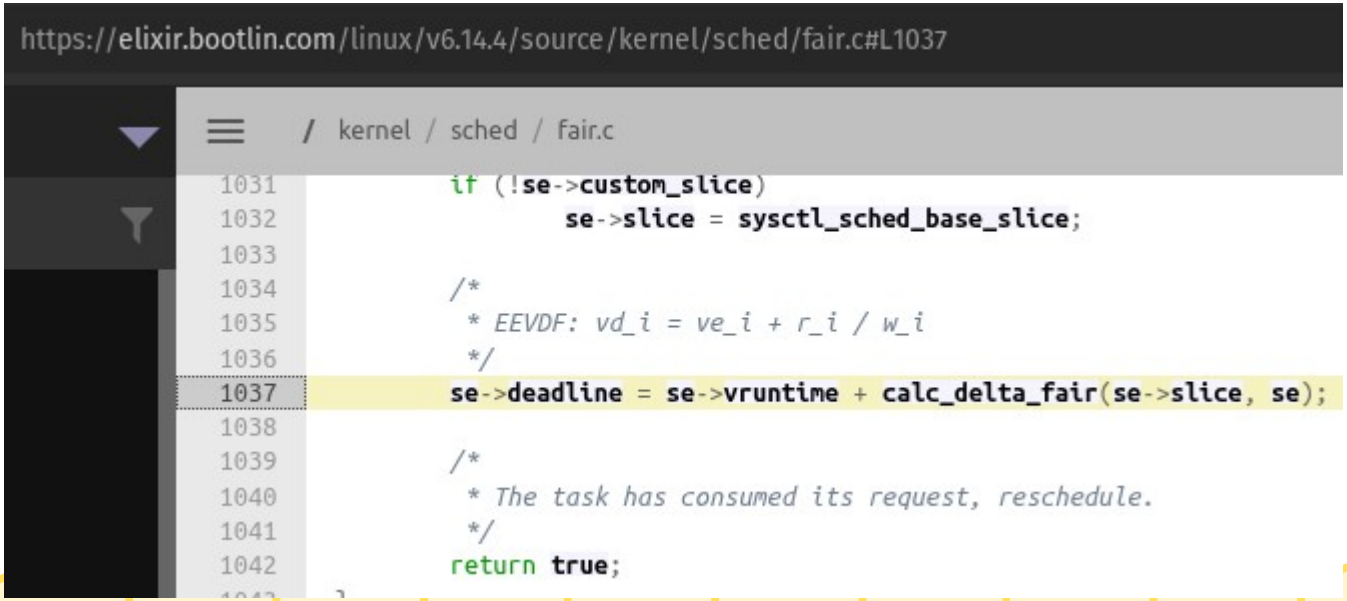
To prioritize among eligible tasks, EEVDF uses **virtual deadlines** to decide which process to run next. The virtual deadline represents the urgency with which a process should be scheduled, and the process with the **earliest** (smallest) virtual deadline δ is chosen to run first. The deadline δ is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

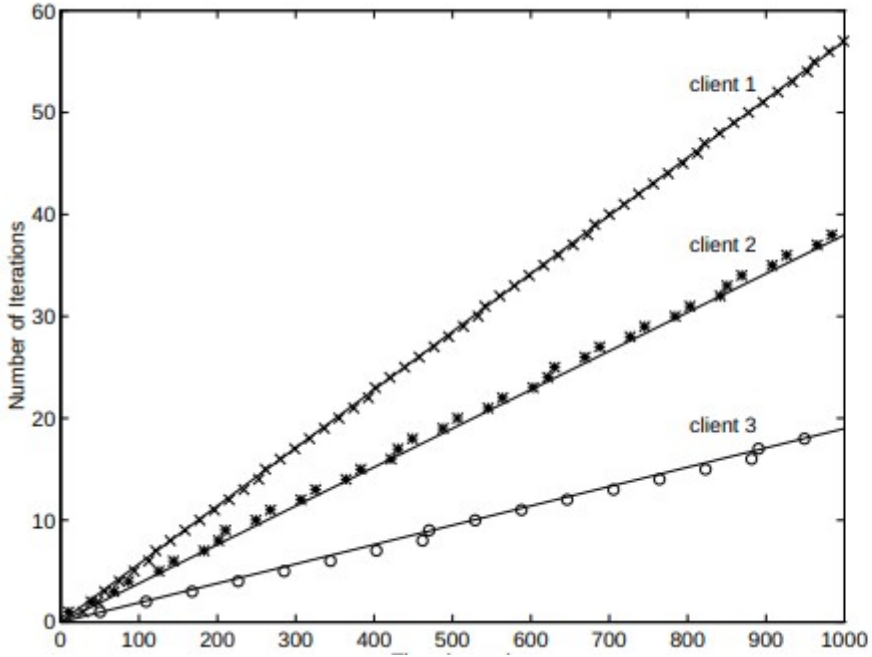
Think of "eligible time" as the moment a task has officially "caught up" to its fair share of the CPU.

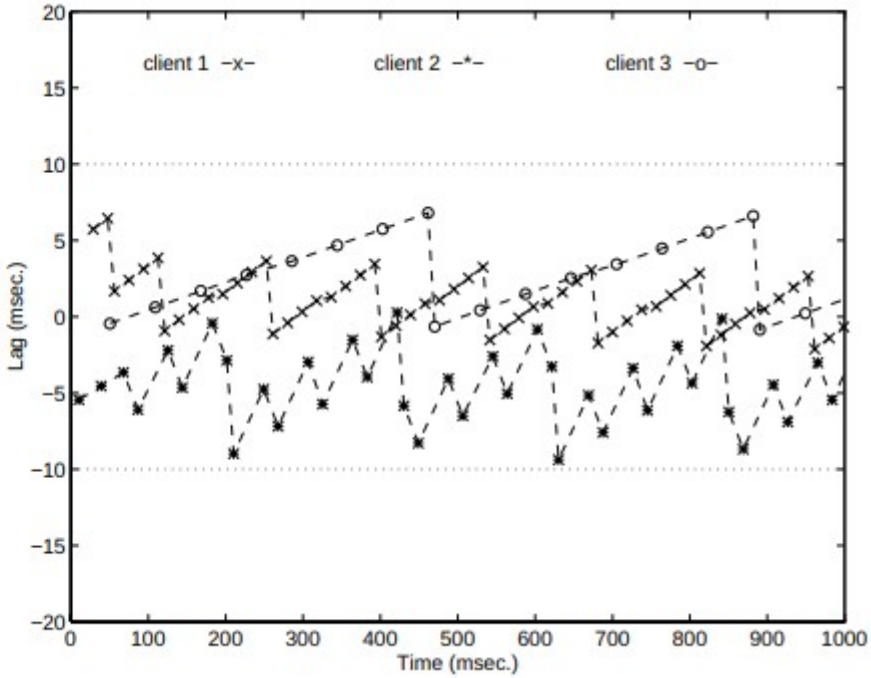


Caveats

- 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)





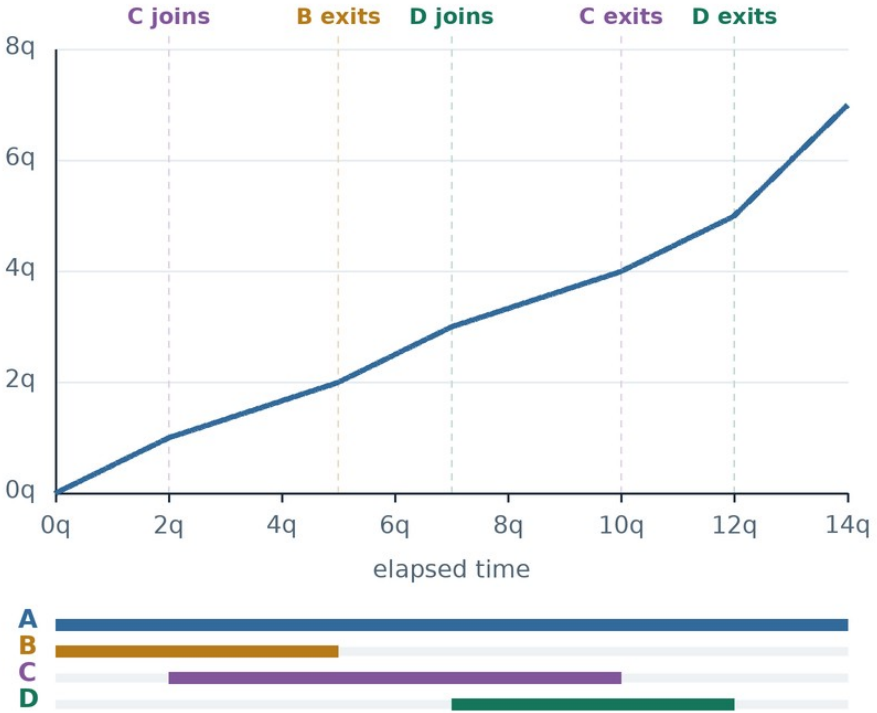
http://www.cs.unc.edu/~jeffay/papers/RTSS-96a.pdf

Tasks come and go, so each task’s ideal CPU share changes over time.

With requests no larger than q and zero-lag arrivals/departures,
EEVDF stays less than q ahead or behind each task’s ideal CPU time.

Fluid model

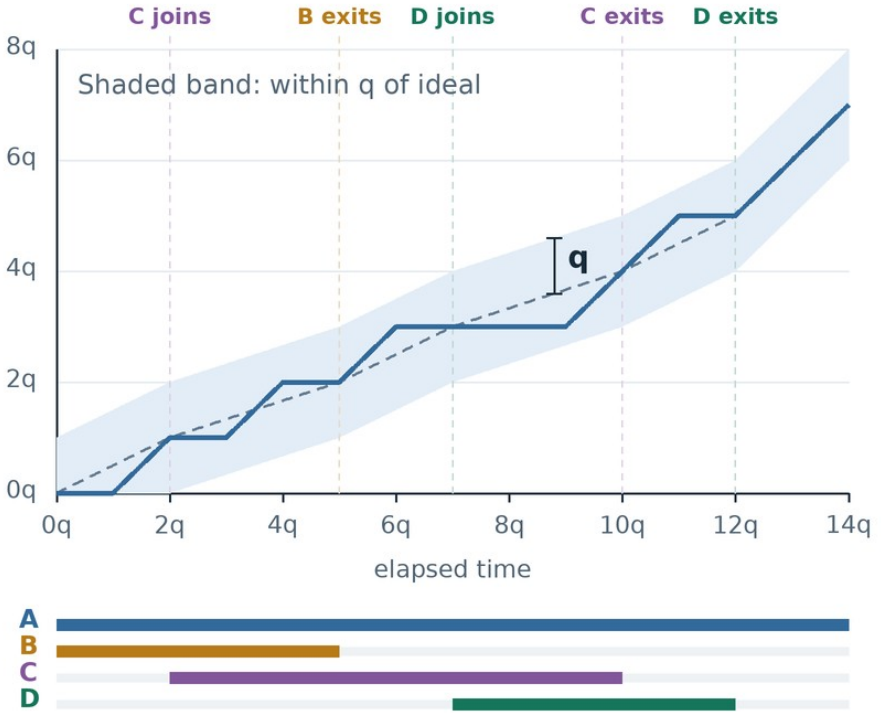
Ideal cumulative CPU time for A



A–D: equal-weight tasks; colored bars show when each is present.
Zero lag: settled CPU credit/debt. q: CPU quantum.

EEVDF

Actual A; fluid ideal dashed



Stoica & Abdel-Wahab, TR-95-22, §5–6


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

EEVDF: slide symbols → task fields

Deck's 6.14-era model, fair-class tasks. se = &p->se; p is the event's task.

In the slides	Kernel field / expression	Meaning / units
ν_p	`p->static_prio - 120`	nice, −20…19 (fair tasks)
w_p, λ_p	`se->load.weight`	kernel-scaled weight; λ shown below
ε_p	`se->sum_exec_runtime`	CPU time actually consumed (ns)
ρ_p, v_p	`se->vruntime`	CPU service adjusted for weight (virtual ns)
δ_p, VD	`se->deadline`	virtual ns; may be relative off-rq
τ_p, r_p	`se->slice`	requested CPU budget (ns), not time left
saved lag	`se->vlag`	saved / clamped virtual lag, not live

Live virtual lag: $g = V - \rho_p$. Eligible when $g \geq 0$.

CPU-service lag = $\lambda_p g$. Slide 27's raw-weight α_p is 1024 times this.

λ_p = load.weight / NICE_0_LOAD. nice 0: λ = 1; slide-table w = 1024.

Slide 14's curve is w. Slide 26 reverses lag's sign. Use $V - \rho_p$.

Source: schedlinux.odp, slides 13–30; Linux sched.h / kernel/sched/fair.c. Field names depend on the running kernel.

EEVDF: runqueue state and tracing

se->cfs_rq identifies the task's CPU/cgroup runqueue (group-scheduling builds).

V is computed by avg_vruntime(): a kernel function, not a BPF helper.

In the slides	6.14-era kernel name	Meaning / tracing detail
V, $\bar{\rho}$	`avg_vruntime(cfs_rq)`	computed average; includes current entity
$\hat{\rho}$	`cfs_rq->avg_vruntime`	weighted numerator, despite its name
Λ	`cfs_rq->avg_load`	weight sum; add current if on_rq
ρ_min	`cfs_rq->min_vruntime`	monotonic offset, not exact tree minimum
tree minimum	`se->min_vruntime`	subtree minimum for eligibility pruning

Later bookkeeping names (arithmetic also changes):

`avg_vruntime → sum_w_vruntime` `avg_load → sum_weight` `min_vruntime → zero_vruntime`

Task activity

`tp_btf/sched_switch`

prev / next supply task pointers.

Also trace wakeups and CPU migration.

Placement, weights and saved lag

`place_entity, reweight_entity, update_entity_lag`

fentry/kprobe if emitted and traceable.

Use running-kernel BTF + CO-RE. Log task ID, CPU and rq/cgroup. current may be the waker.

Check on_rq, sched_delayed and rel_deadline; older kernels may reuse vlag storage as vprot.

Sources: Linux fair.c, sched.h, include/trace/events/sched.h; libbpf documentation. Compare against your kernel's implementation.

Coming up… Real-time scheduling and how Linux can now claim to be “hard real time”...