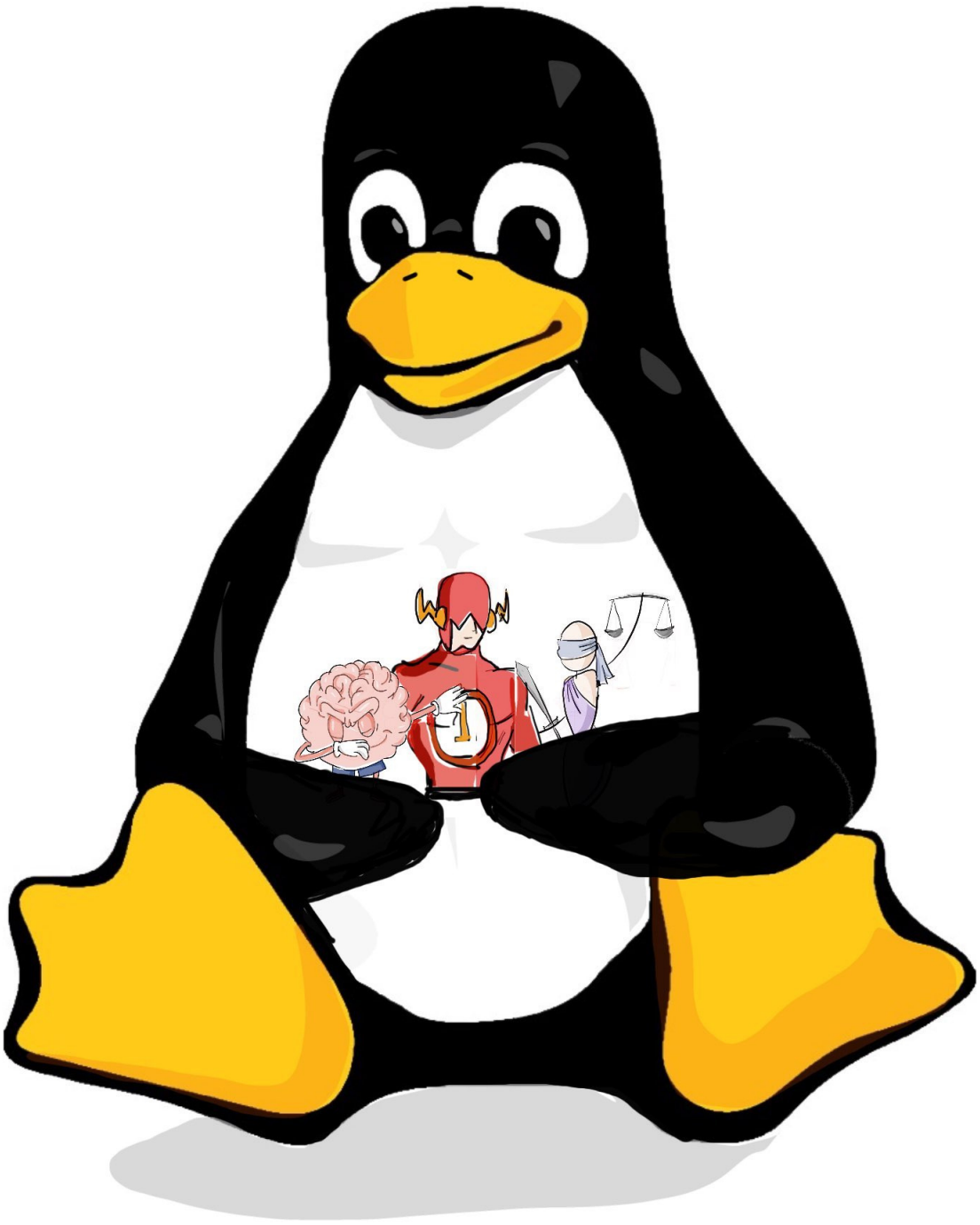
Linux Schedulers

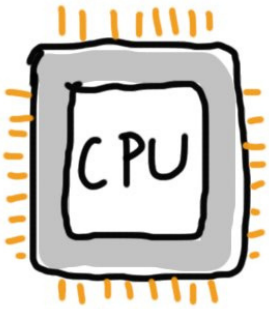


by Soham & Tanvi, 2021

Note to the reader:

This is a light-hearted summary of popular Linux schedulers. While we have tried to avoid unnecessary jargon, an elementary understanding of Operating Systems may be useful!

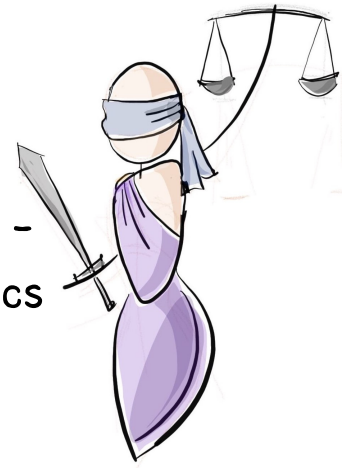
Meet the Cast



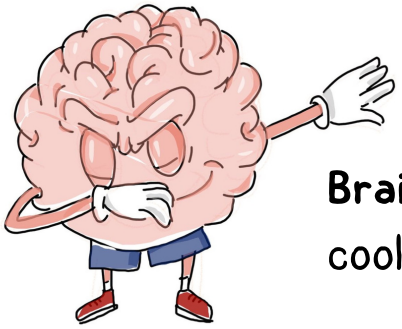
The CPU - everyone wants a (time)slice of this!

Tux - your friendly neighbourhood penguin

O(1) Scheduler - It's got no chill!



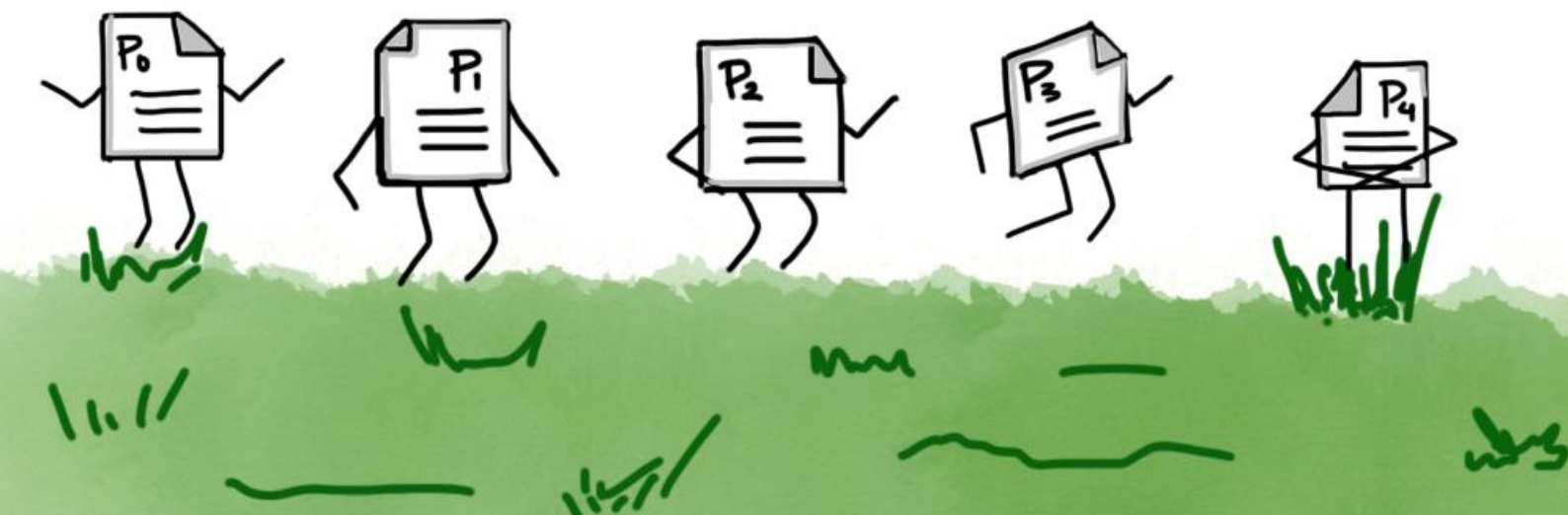
Completely Fair Scheduler - Scholar of Morality and Ethics



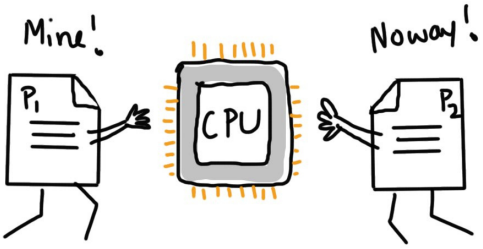
Brain F*ck Scheduler - The coolest kid on the block

Processes - Keep dreaming about that sweet, sweet CPU timeslice

Contents



Scheduling Basics



A **scheduler** is a manager who decides which process gets to run next on the CPU. While doing so, it tries to convince each process that it has its own CPU

Scheduling policy is a balancing act between multiple **goals**, such as:

Low Latency:
How long does a process wait till it gets to run on the CPU

Progress:
How much does the process accomplish in a given timeframe

Fairness:
Who gets how much and why?

Timesharing Schedulers

Goal is to provide **low latency** for (IO-bound) interactive tasks. They assume interactivity is inversely related to CPU-usage.

Since these do not prioritize progress, special care has to be taken to prevent **starvation** (no progress being made)

Proportional-Share Schedulers

Main goal is **fairness** in CPU allocation. Every process is allocated a fraction* of CPU resources in proportion to their weights (relative importance)

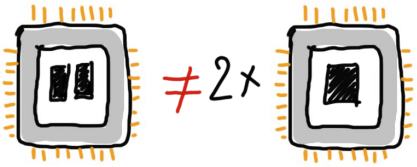
These may be either stochastic (lottery-based) or deterministic

*such an illusion is created

Multicore Issues

Since 2004, sinking transistors no longer resulted in faster processors. This led to **multicore processors** - they increase performance without increase clock-speed.

However, that adds a bunch of new goals to attain, such as:

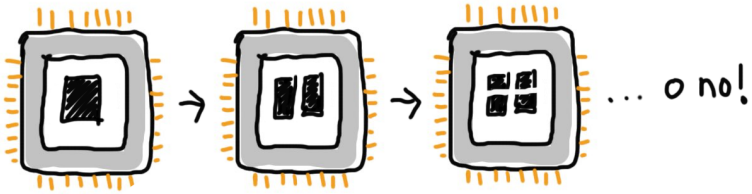


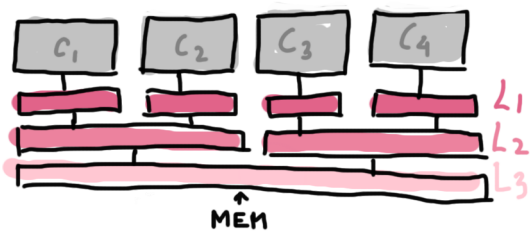
Matching Single Processor Policy

Given enough parallel processes to run, performance in an 8-core machine should be equivalent to a 1-core machine with an 8 times faster processor

Scalability

Architecture should be such, and overheads should be low enough that additional cores translate to increased performance





Maximizing Hardware Features

Processes continually scheduled on the same processor are likely to find data in the processors' cache. Cache/ Processor-affinity may thus, significantly affect performance

Multicore Issues

There are 2 main architectures in use, which address some of these issues in multi-core processors:

Global Queue

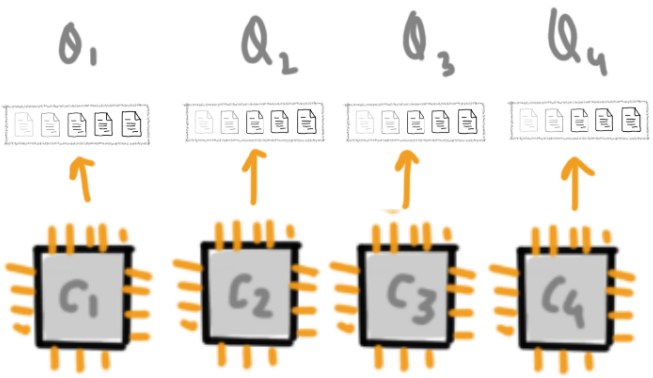
Each processor selects processes from a shared global run-queue. This natively tries to match single processor policy.

Drawbacks include complicated synchronization among processors and poor scalability

Distributed Queue

Each process gets its own run-queue. This makes this approach easily scalable and also uses cache hierarchies effectively.

The major drawback is the need for a complicated load-balancing mechanism across all run-queues, to match the single processor policy.



O(1)

Who's this?

A stable, timesharing scheduler, used in Linux from 2.6 through 2.6.22. It was also used internally by Google till at least 2009

What's Cool?

The next task to run is selected in O(1). In fact every step of the algorithm happens in O(1) - it's fast!

What's Not?

Uses very complex heuristics (such as average sleep-time) to reward/penalize interactive and non-interactive processes

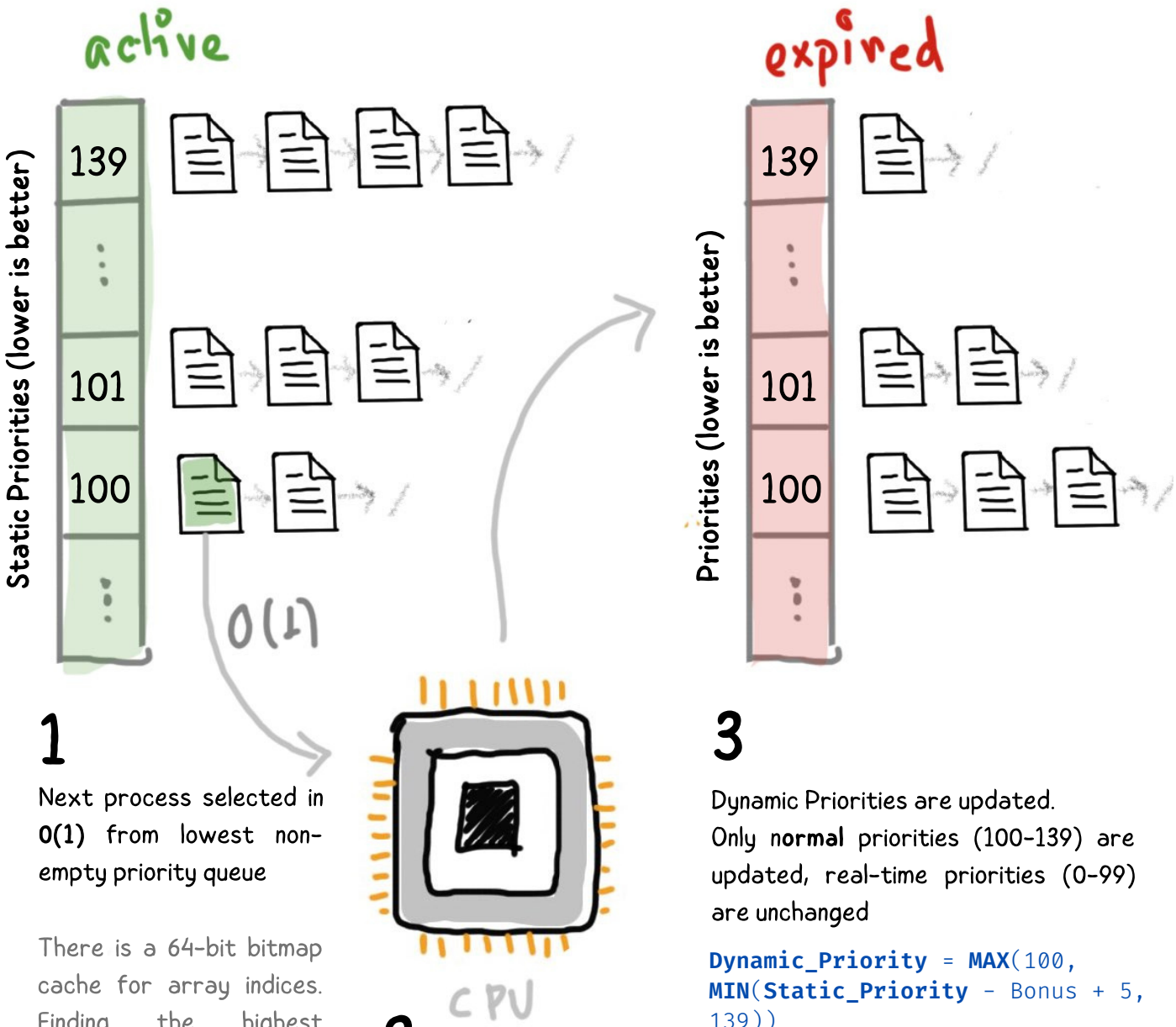
Fun Fact!

Ingo Molnar first implemented an O(1) scheduler for Linux 2.6

O(1)

Tell me more!

Each active process is given a fixed timeslice to run, after which it is moved to the expired array. When there are no active processes, the active and expired arrays are swapped. Here's the full process:



1

Next process selected in **O(1)** from lowest non-empty priority queue

There is a 64-bit bitmap cache for array indices. Finding the highest priority task is thus a matter of two x86 BSFL bit-search instructions

Default priority of a new process is 120. This can be changed using **nice** as shown:

```
$nice -n [-20,+19] ./a.out
```

2

CPU Timeslice calculated based on dynamic priority. Interactive (IO-bound) Processes get more time than Batch (CPU-bound) Processes (unlike MLFQ)

Process removed on timeslice expiration

3

Dynamic Priorities are updated.
Only **normal** priorities (100-139) are updated, real-time priorities (0-99) are unchanged

```
Dynamic_Priority = MAX(100, MIN(Static_Priority - Bonus + 5, 139))
```

Source: http://pages.cs.wisc.edu/~jmeehean/Papers/meehean-thesis11.pdf

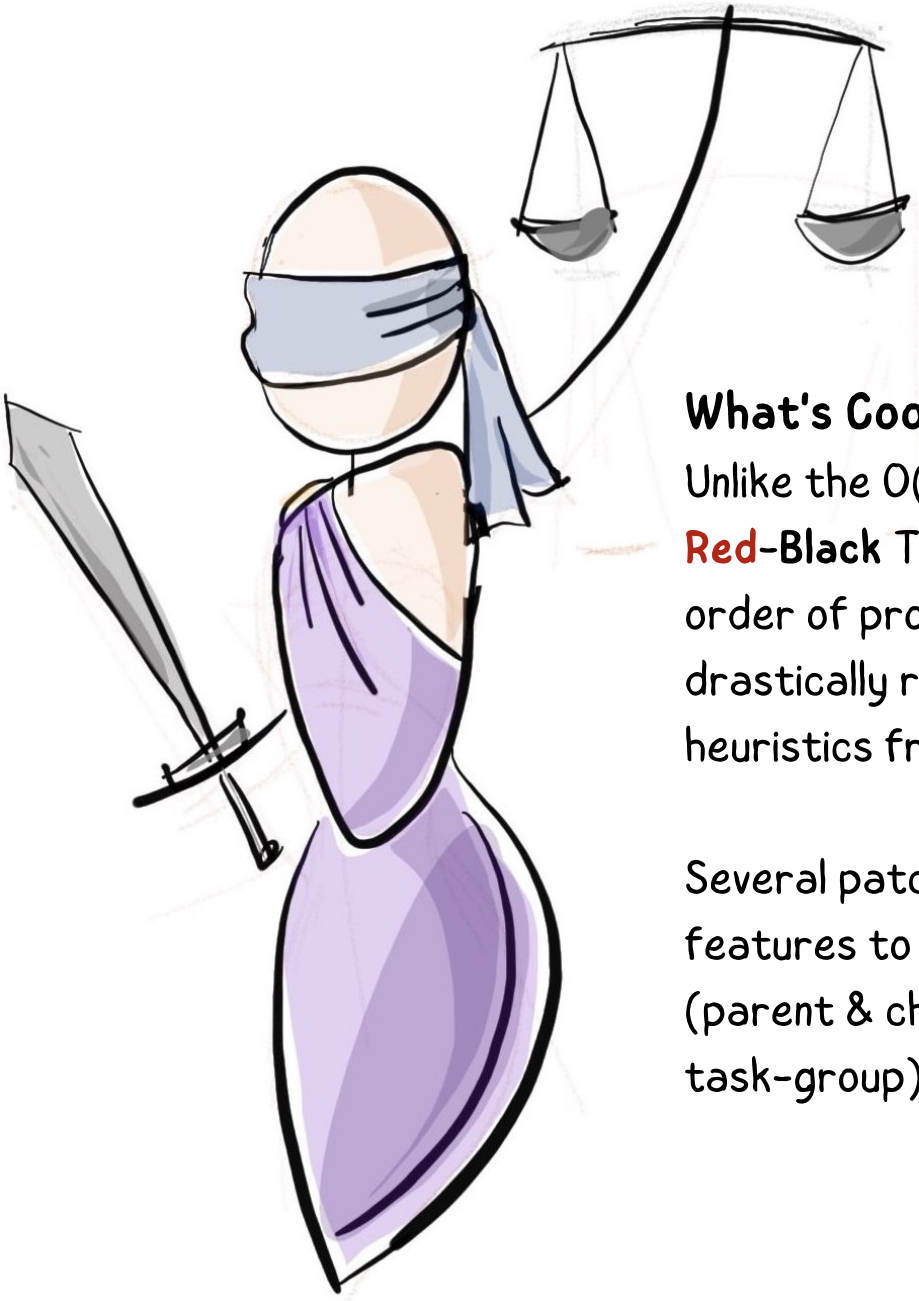
Completely Fair

Who's this?

A proportional-share scheduler still under active development, used as the official replacement for O(1) in Linux (from 2.6.23 onwards).

Fun Fact!

Ingo Molnár calls it a model of an 'ideal precise multitasking CPU'



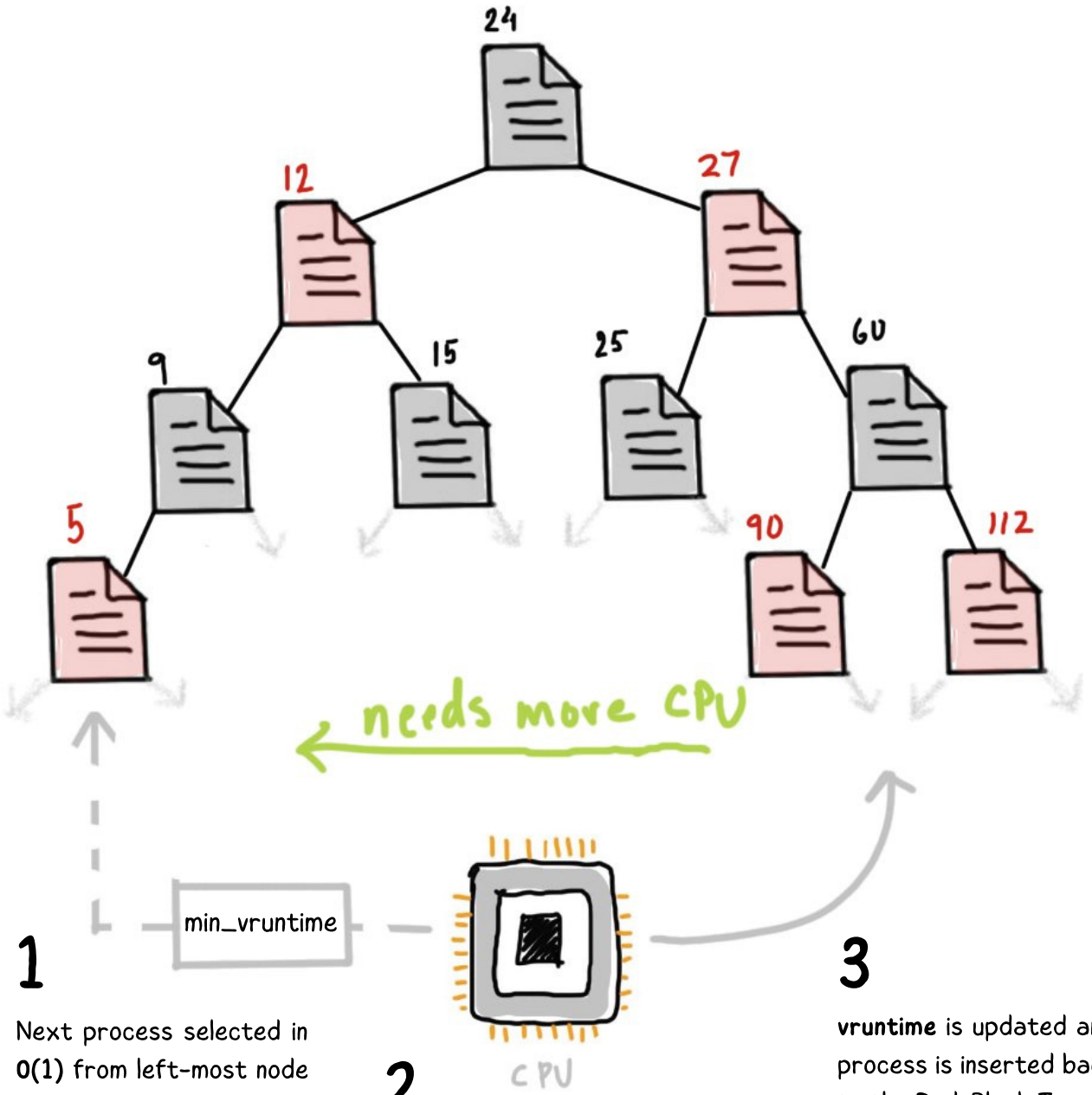
What's Cool?

Unlike the O(1) scheduler, CFS uses **Red-Black** Trees to generate the order of process execution. This drastically reduces the complex heuristics from the O(1) scheduler.

Several patches have added other features to CFS such as autogrouping (parent & child processes in the same task-group) and better load-balancing

Tell me more!

Processes are maintained in a (self-balancing) Red-Black Tree, indexed by their **vruntime** (the total time a process has run for). At every context-switch, process with minimum **vruntime** is selected (in O(1)). Here's the full process:



1

Next process selected in **O(1)** from left-most node of the Red-Black Tree (a pointer is maintained)

2

Process runs for 't' ms (dynamic timeslice)

Interactive (IO-bound) processes run smaller 't' before yielding, thus the increase in vruntime is smaller

3

vruntime is updated and process is inserted back to the Red-Black Tree in **O(logn)**

```
vruntime += t * (weight, based on nice value)
```

Source: https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf

BrainF*ck

Who's this?

A proportional-share scheduler, designed by Con Kolivas in 2009 as an alternative to the CFS. It wasn't intended to be integrated with the mainline Linux kernel

Fun Fact!

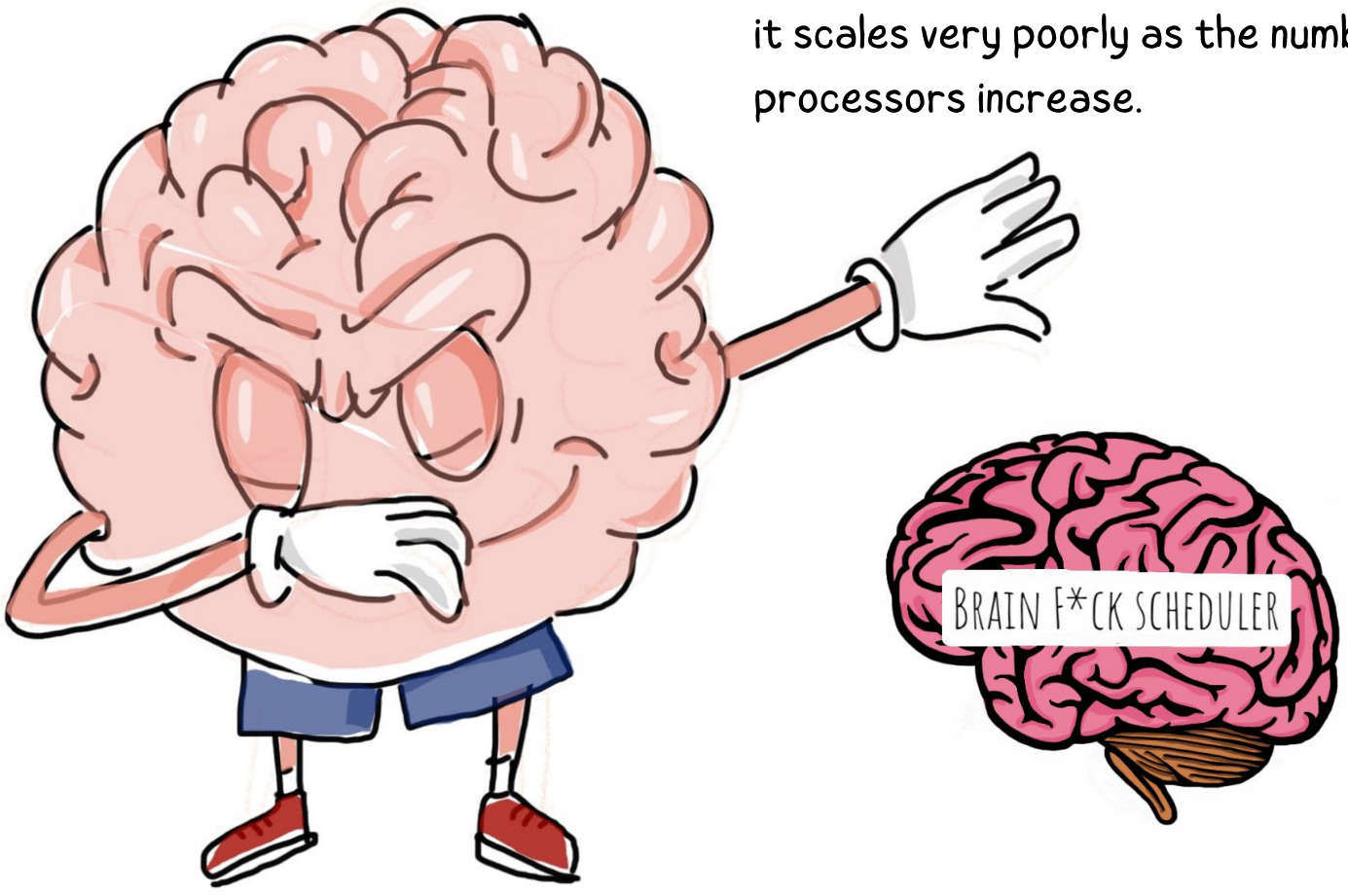
Kolivas was inspired by XKCD comic #619 to make the BF Scheduler

What's Cool?

Uses a fairly simple algorithm and a single unsorted global run-queue. Avoids all complex heuristics and tunable parameters

What's Not?

Since it uses a single global run-queue, it scales very poorly as the number of processors increase.

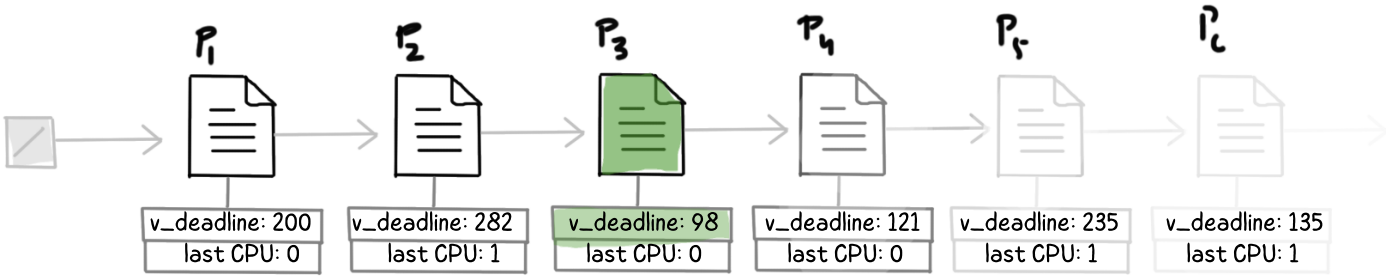


Tell me more!

Each runnable process is maintained in a single global run-queue, along with its virtual deadline (and the CPU it was last run on). The process with the earliest deadline is chosen to run next. Here's the full process:

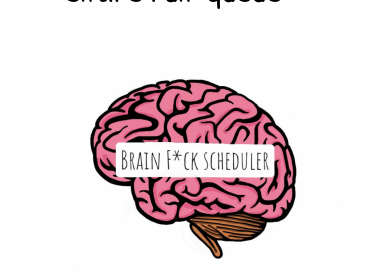
Each process is assigned a **virtual deadline** and stored in a system-wide linked-list

The assigned deadline is **virtual**, as it provides no guarantees - it is simply the longest time a process will have to wait to get a CPU timeslice

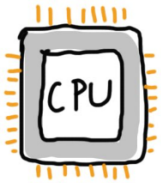


1

Next process selected in **O(n)** by searching for the earliest deadline in the entire run-queue



2

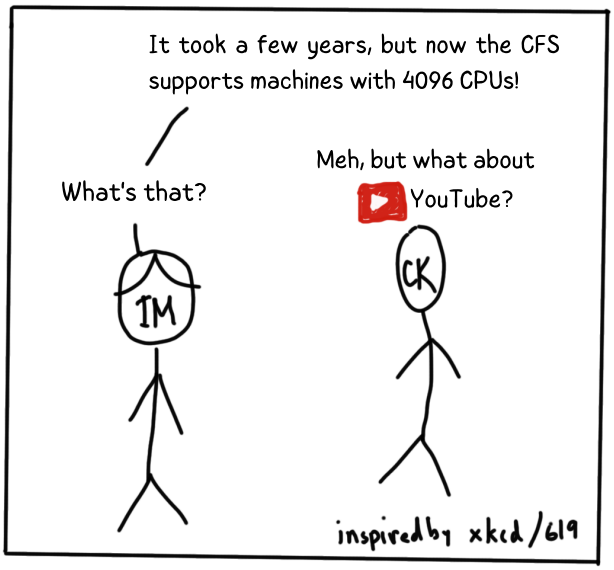


A process is assigned a timeslice when it requests CPU time. On using up the time, it is rescheduled and updated in the queue

The BrainF*ck Scheduler scales very poorly - that was OK for Kolivas

3

If the process blocks, then it keeps the remainder of its timeslice and deadline - this increases interactivity



Fun Fact!

Kolivas also created the Staircase Deadline Scheduler, which partly inspired the CFS - Kolivas left Linux soon after

Source: https://www.cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

Beyond Linux

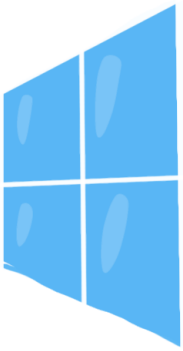
In this zine, we've covered some of the most popular Linux Schedulers. However, beyond Linux there's still lots left to explore ...

Solaris

Ships with both a timesharing scheduler (using a classic MLFQ) and a proportional share scheduler (based on decay-usage). Distributed run-queues are used in the multi-core setting

FreeBSD

FreeBSD's ULE scheduler uses decay-usage to implement a timesharing policy. It also uses distributed run-queues and a push/pull load-balancer



Windows

Uses a timesharing scheduler implemented as an MLFQ, with temporary priority boosts. Since 2003, Windows has moved from global run-queues to distributed ones for multi-core systems

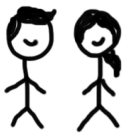
OSX

OSX uses a priority-based decay-usage scheduler to implement timesharing. It also uses distributed run-queues

Source: http://pages.cs.wisc.edu/~jmeehean/Papers/meehean-thesis11.pdf

Thank You!

We were inspired by Julia Evans to make this zine. We would also like to thank Professor Manu Awasthi.



All the illustrations in this zine are by Soham & Tanvi