INF113: MLFQ and CFS

Kirill Simonov

12.09.2025

Assignment 1

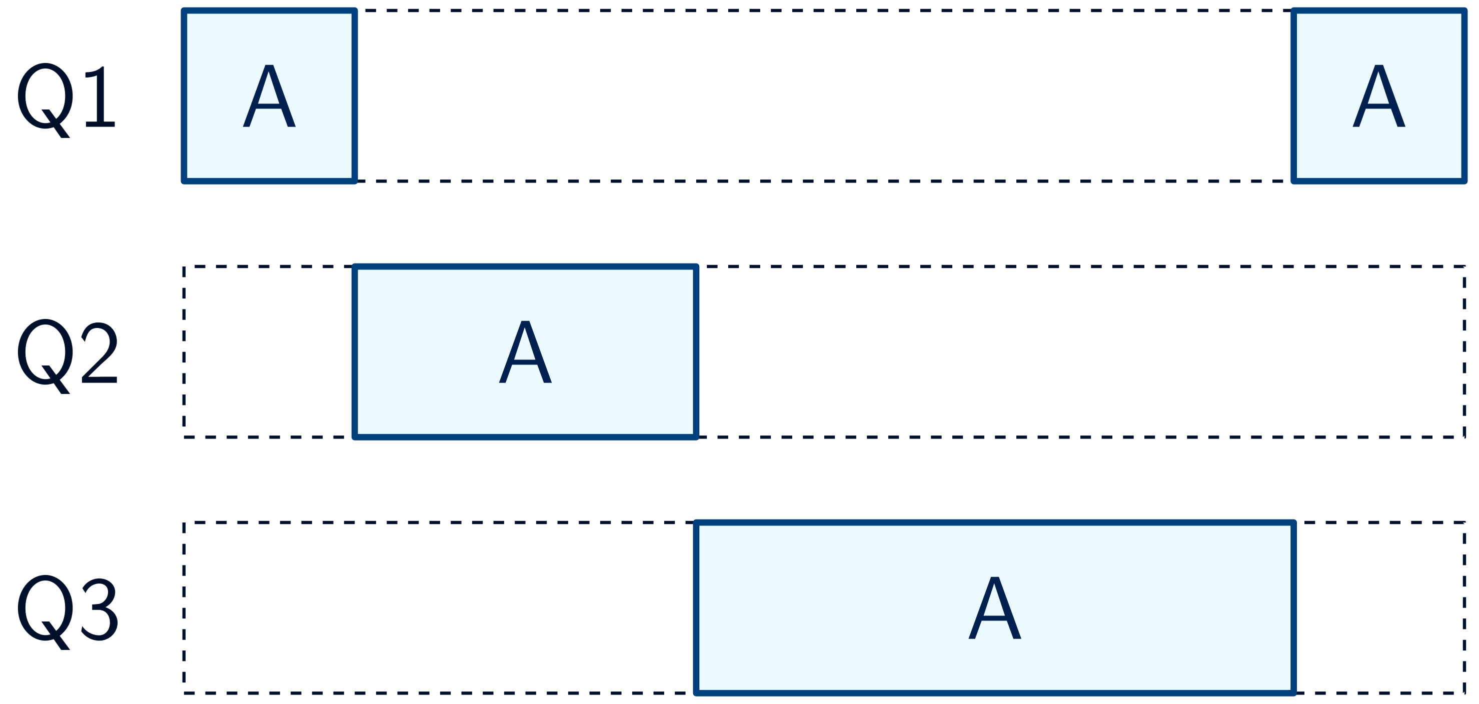
- Will be published later today
  Deadline: Friday September 26 (two weeks)
- We provide task descriptions and code “skeletons”
- You fill the skeletons in, and deliver the files via Mitt/CodeGrade
- In case of issues/if you are stuck
  - Write on Discord
  - Come to a group session
- The assigment is individual—submitting someone else’s code is not allowed!
- LLM-produced code is not allowed either
- No lectures next week

Why wait on I/O

Minute		
L1 cache reference	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee
Hour		
Main memory reference	100 s	Brushing your teeth
Compress 1K bytes with Zippy	50 min	One episode of a TV show (including ad breaks)
Day		
Send 2K bytes over 1 Gbps network	5.5 hr	From lunch to end of work day
Week		
SSD random read	1.7 days	A normal weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A medium vacation
Read 1 MB sequentially from SSD	11.6 days	Waiting for almost 2 weeks for a delivery
Year		
Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a new human being
The above 2 together	1 year	
Decade		
Send packet CA→Netherlands→CA	4.8 years	Average time it takes to complete a bachelor's degree

The Queues in MLFQ

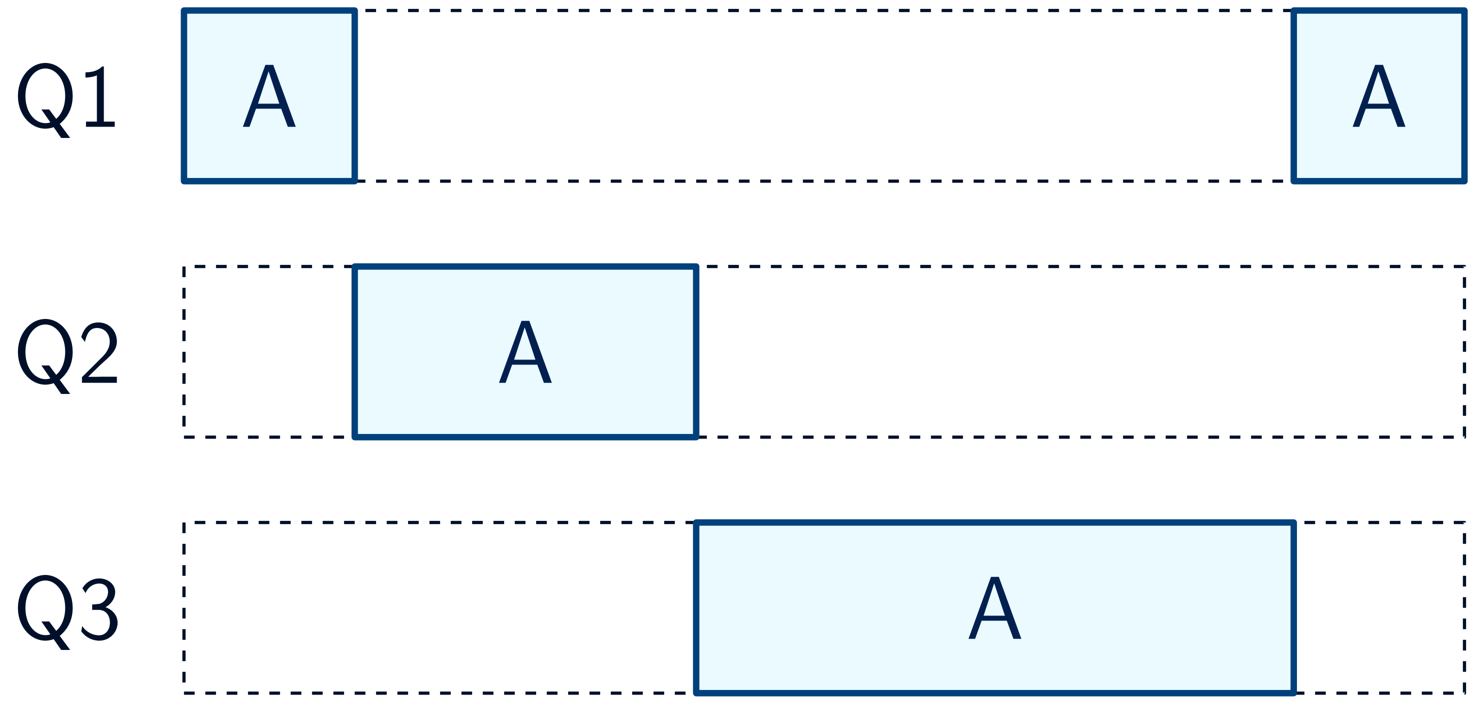
- There are several queues, modelling different priorities
- Run the processes from the highest non-empty queue in RR
- When a process comes, put it in the topmost queue
- If a process uses up its allotment, move it down
- Priority boost: after time S, move all jobs to topmost queue



MLFQ: Potential issues

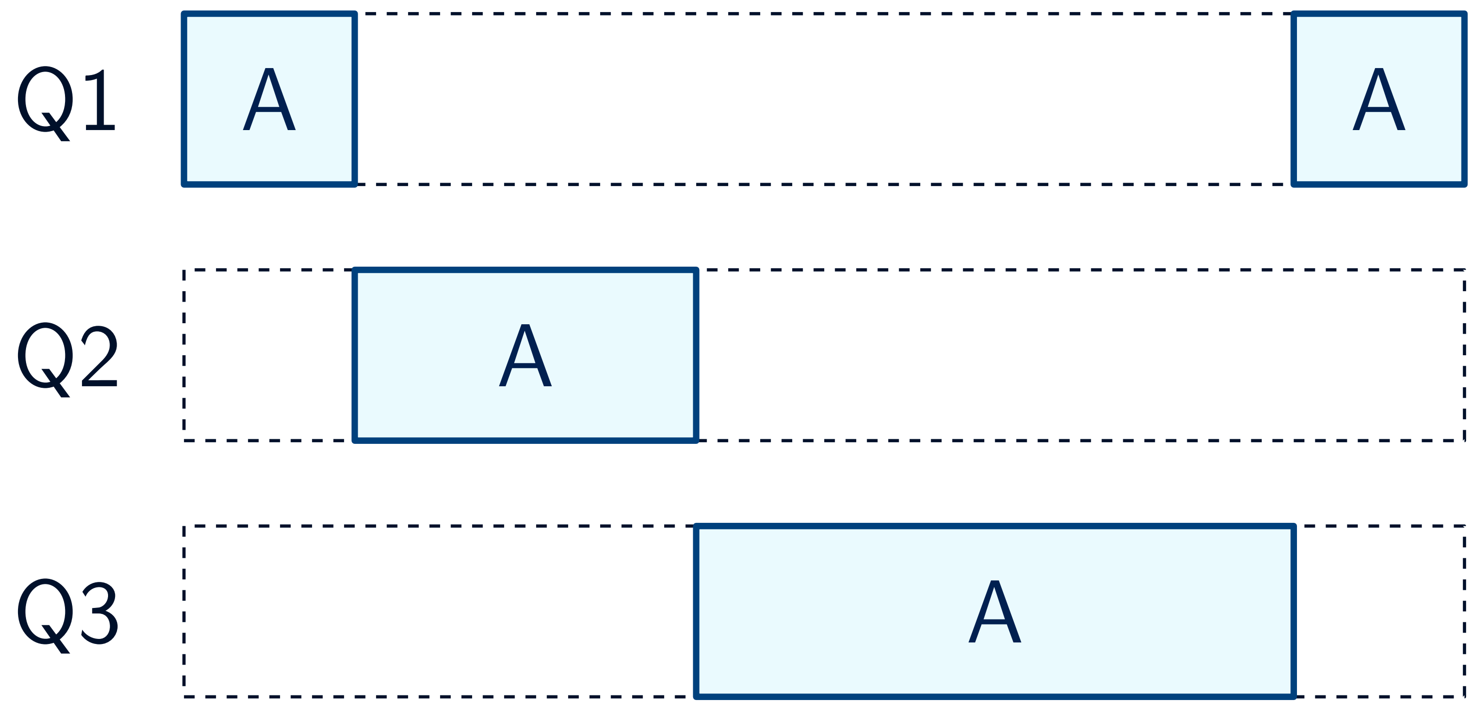
- Starvation: bottom-queue jobs never get to run
- **Solution:** Priority boost
- Two-phase job: first compute, then lots of I/O
- **Solution:** Priority boost

- Cheating: Issuing I/O right before allotment ends
- **Solution:** Allotment is not reset while waiting



MLFQ: Customization

- Number of queues
- Allotments on each level
- Priority boost time
- Scheduler within the queue:
  e.g., FIFO or Round Robin
- Additional rules for moving up or down
  e.g., moving one level up after I/O



- Computing priority directly
  e.g. FreeBSD: formula based on usage
  usage decays over time
- User-advised priorities
  `man nice`

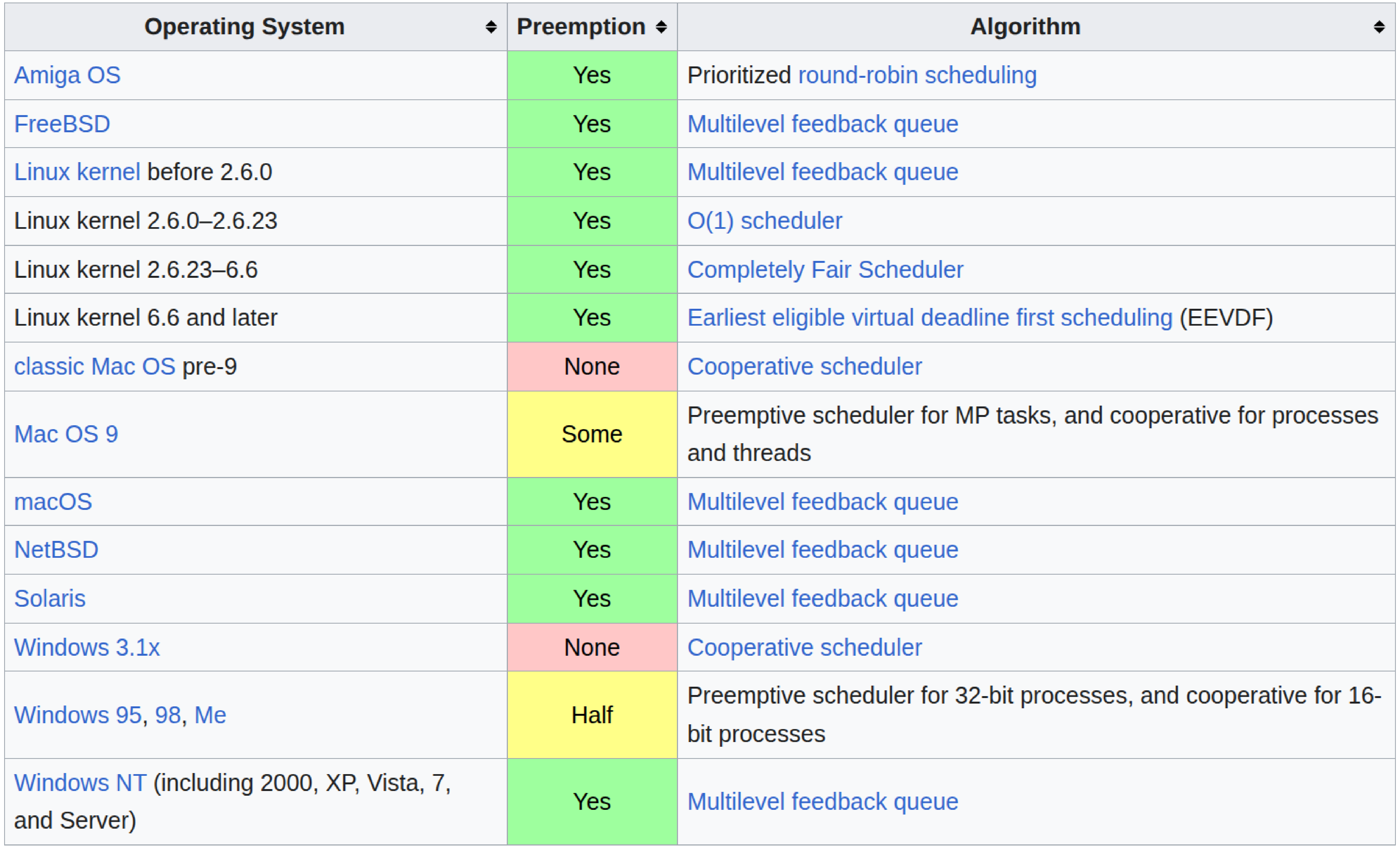
Lottery scheduling

- Each job holds **tickets**
- Job for the next timeslice is determined by a random draw
- Fairness: Each job gets proportional runtime, in expectation
- Easy to implement

Assigning tickets

- Big question: how to distribute/adjust tickets
- All tickets to shortest job: Shortest Job First
- Split tickets equally: Round Robin
- MLFQ-like: Start with equal tickets, remove tickets upon consuming CPU

Schedulers IRL



Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel 2.6.23–6.6	Yes	Completely Fair Scheduler
Linux kernel 6.6 and later	Yes	Earliest eligible virtual deadline first scheduling (EEVDF)
classic Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, and cooperative for processes and threads
macOS	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, and cooperative for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

from 1991
2003
2007
2023

https://en.wikipedia.org/wiki/Scheduling_(computing)

Completely Fair Scheduler (CFS)

- Main goal: fairly divide CPU among competing processes
- Based on counting **virtual runtime** `vruntime`
  - Process accumulates `vruntime` whenever it uses CPU
  - Process with the least `vruntime` gets to run
- Time slice: `sched_latency`
  - Once a process is scheduled, it runs for at least `sched_latency`
  - `sched_latency` is dynamic: 48ms divided by n
  - when n processes want to run
- If n is too large: `min_granularity`
  - `sched_latency` is always at least `min_granularity`
  - `min_granularity` is usually 6ms

Niceness

- All user processes are made equal

- We can make the process get more/less CPU by calling `nice` on it

```
$ nice -5 ./a
```

means that `./a` gets niceness 5, as opposed to default 0
priority 25, default 20

- Higher `nice` value—smaller share

```
$ nice -5 ./a
$ ./b
```

means that CPU share of `./a` and `./b` is 1:3

Weighting

- In fact, `nice` levels define weights

```
static const int 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,
};
```

- `nice 0` and `nice 5` process compare exactly the same as `nice 5` and `nice 10`

Using weights

- Time slice is proportional to weight, normalized to weights of all n processes

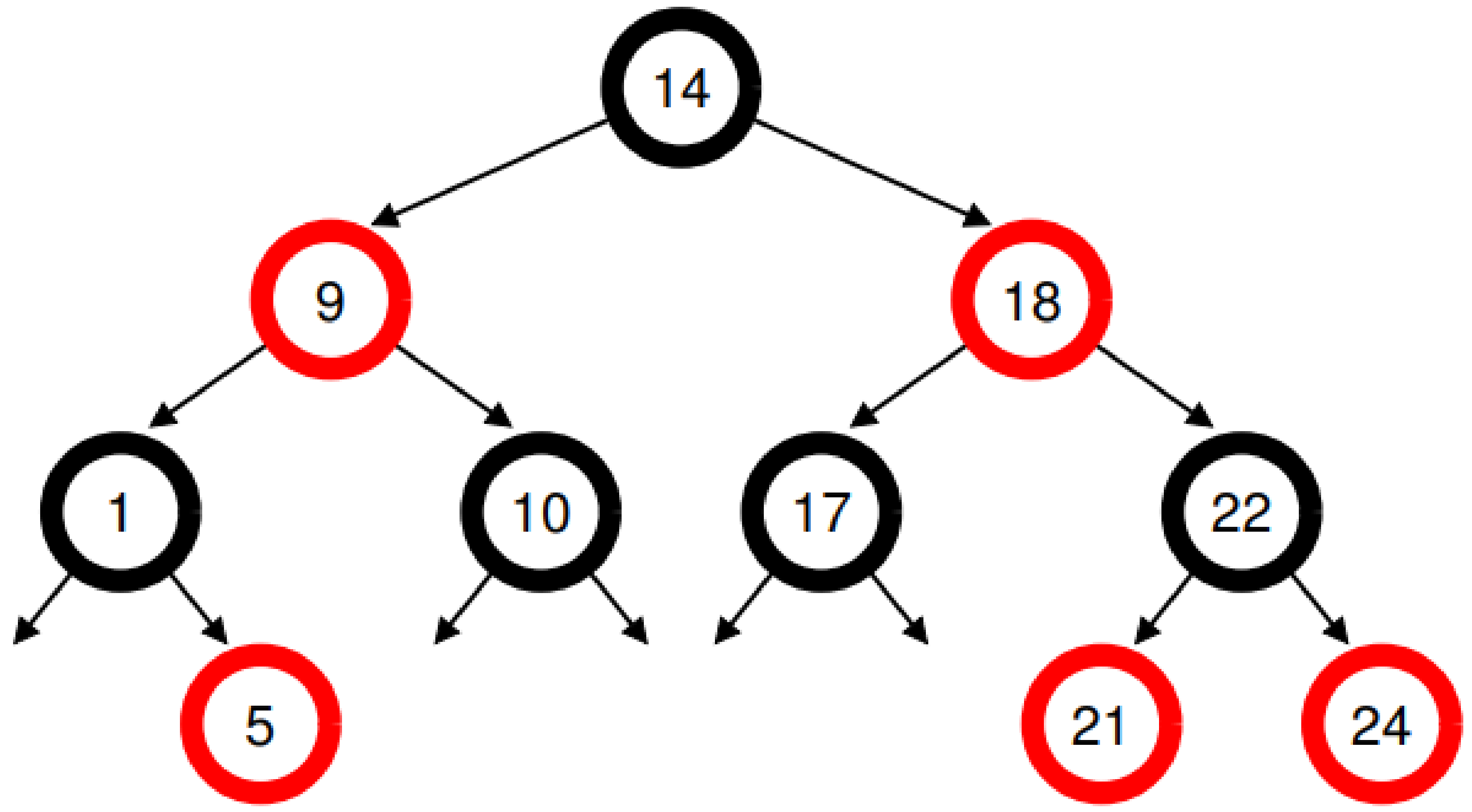
$$\texttt{time\_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \cdot \texttt{sched\_latency}$$

- `vruntime` accumulates faster for low-priority processes

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \cdot \texttt{runtime\_i}$$

Red-black trees

- CFS wants to get process with the lowest `vruntime` fast
- All active processes are stored in a **red-black tree**
- In time $O(\log n)$:
  - Get process with lowest/given `vruntime`
  - Add process
  - Remove process
  - Update `vruntime` of a process


14
9
18
1
10
17
22
5
21
24