# CS162
# Operating Systems and Systems Programming
# Lecture 12

## Scheduling
## Linux Schedulers and Fair Sharing

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

# Goals for Today

- How do Linux schedulers work?

- Proportional fair sharing and Linux EEVDF

# Recall: What We Want from Schedulers

1) Minimize average waiting time for IO/interactive tasks
(tasks with short CPU bursts)

2) Minimize average completion time

3) Maximize throughput
(includes minimizing context switches)

4) Remain fair/starvation-free

# History of Schedulers in Linux

O(n) scheduler
Linux 2.4 to Linux 2.6

O(1) scheduler
Linux 2.6 to 2.6.22

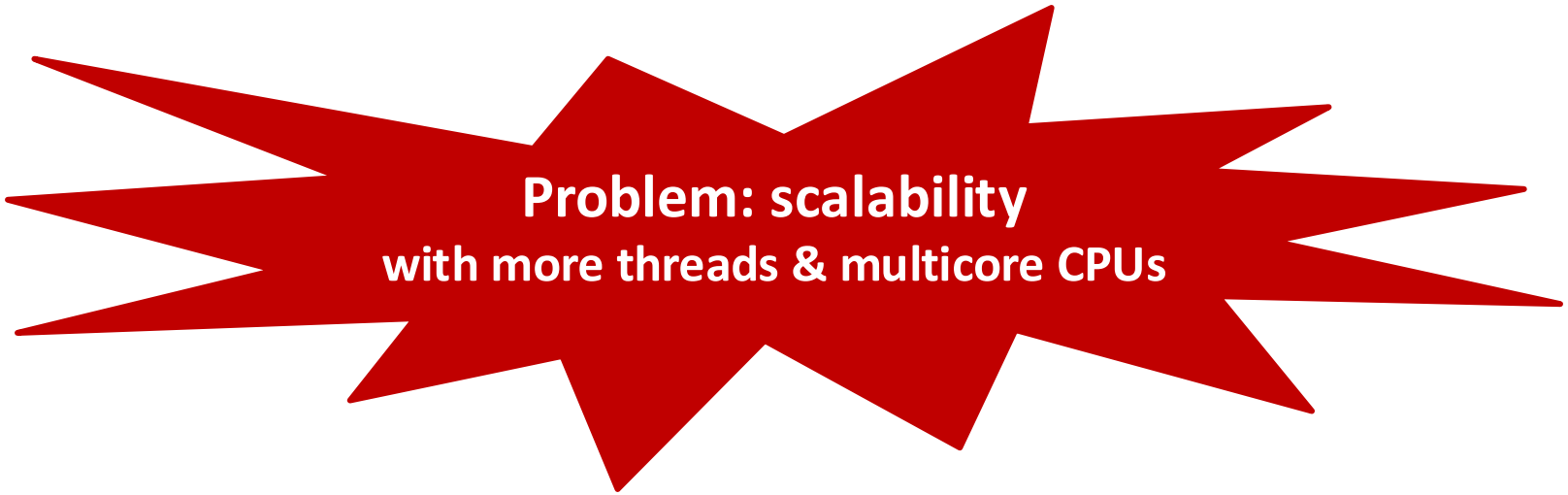
Completely Fair Scheduler (CFS)
Linux 2.6.23 to 6.6

EEVDF scheduler
Linux 6.6 onwards

# Recall: Linux O(n) Scheduler

At every context switch:

- Scan full list of processes in the ready queue
- Compute a priority score for each one
- Select the best process to run



# Recall: Linux O(1) Scheduler (MLFQ-like)


Next process to run is chosen in constant time

Priority-based scheduler with 140 different priorities

Real-time/kernel tasks assigned priorities 0 to 99 (0 is highest priority)

User tasks (interactive/batch) assigned priorities 100 to 139

- Set by via "nice" command, whose values range from -20 to 19

# O(1) Scheduler for User Tasks

Per priority-level, each CPU has two ready queues

An active queue, for processes which have not used up their time quanta

An expired queue, for processes who have

Timeslices/priorities/interactivity credits all computed when jobs finishes timeslice

Timeslice depends on priority

# O(1) Scheduler – Priority Adjustment

User-task priority adjusted ±5 based on heuristics

- » p->sleep_avg = sleep_time – run_time
- » Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)

Interactive Credit

- » Earned when a task sleeps for a "long" time
- » Spend when a task runs for a "long" time
- » IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior

However, "interactive tasks" get special dispensation

- » To try to maintain [illegible]activity
- » Placed back into active [illegible] for too long…

**Problem: Very complex to reason about and tune**

# A Different Approach: Proportional Fair Sharing

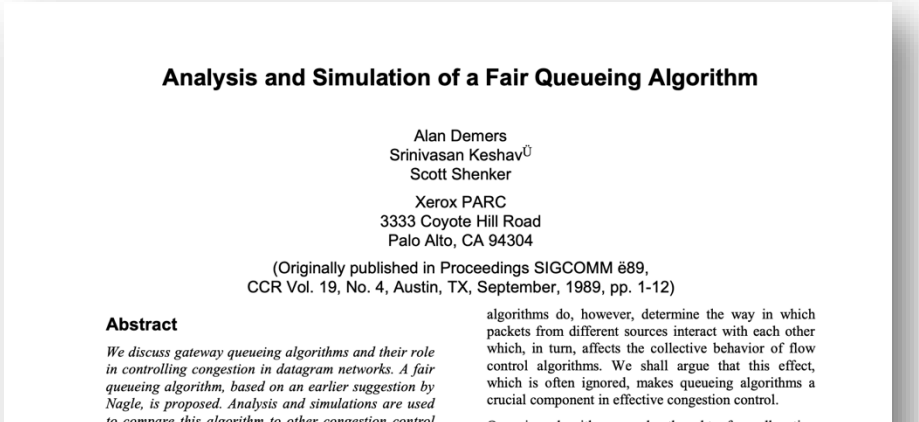
Share the CPU *proportionally* to per-job "weights"

Give each job a share of the CPU according to its priority

Low-priority jobs get to run less often

But all jobs can make progress (no starvation)

Originated in networking



**Analysis and Simulation of a Fair Queueing Algorithm**

Alan Demers
Srinivasan KeshavÛ
Scott Shenker

Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

(Originally published in Proceedings SIGCOMM ë89,
CCR Vol. 19, No. 4, Austin, TX, September, 1989, pp. 1-12)

**Abstract**

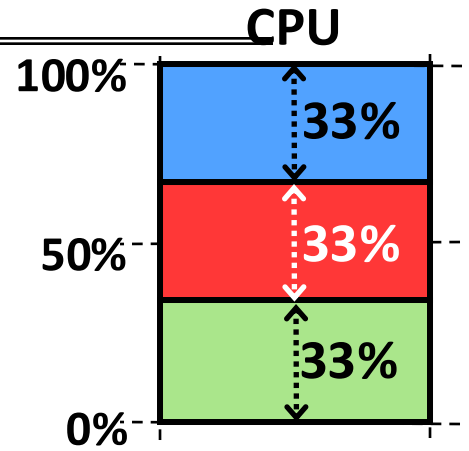
*We discuss gateway queueing algorithms and their role in controlling congestion in datagram networks. A fair queueing algorithm, based on an earlier suggestion by Nagle, is proposed. Analysis and simulations are used*

algorithms do, however, determine the way in which packets from different sources interact with each other which, in turn, affects the collective behavior of flow control algorithms. We shall argue that this effect, which is often ignored, makes queueing algorithms a crucial component in effective congestion control.

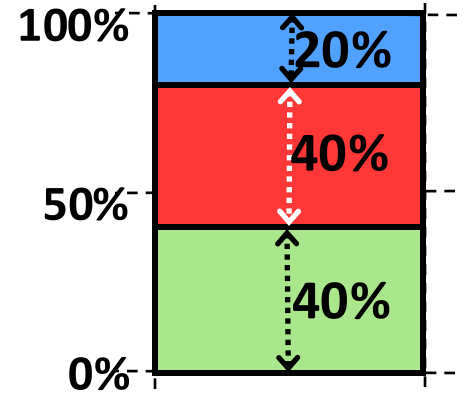
Fair sharing aims to maintain a global property (fairness) instead of reasoning about queues and priorities!

# Fair Sharing Variants

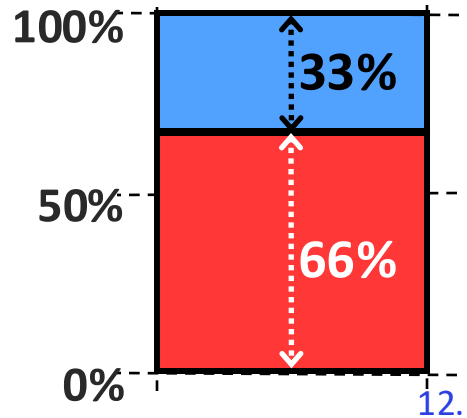
- *n* users want to share a resource (e.g. CPU)
  - Solution: give each *1/n* of the shared resource




- Generalized by ***max-min fairness***
  - Handles case where a user needs less than its fair share
  - E.g. user 1 needs no more than 20%




- Generalized by ***weighted/proportional*** *max-min fairness*
  - Give weights to users based on their importance
  - E.g. first user has weight 1, second user has weight 2



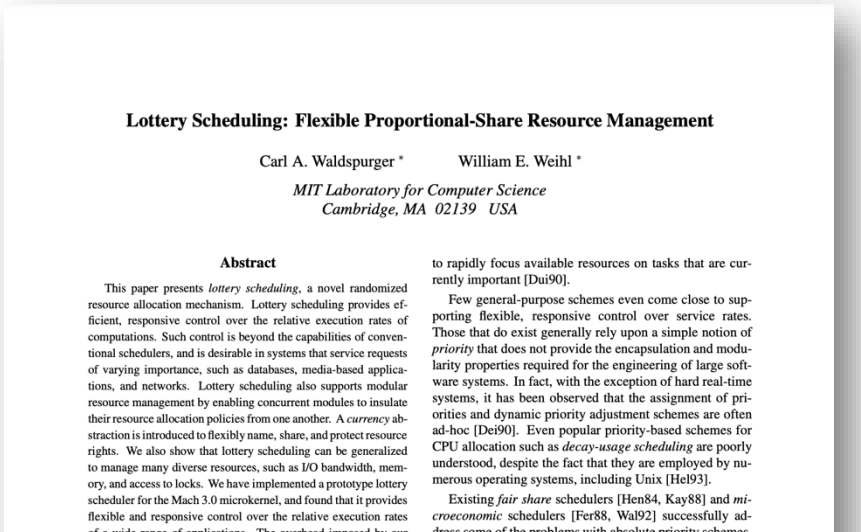
# Early Example: Lottery Scheduling

Give each job some number of lottery tickets (≥1)

On each time slice, randomly pick a winning ticket

On average, each job's CPU time is proportional to the number of tickets it has

Running time? O(1)

**Lottery Scheduling: Flexible Proportional-Share Resource Management**

Carl A. Waldspurger * William E. Weihl *

*MIT Laboratory for Computer Science*
*Cambridge, MA 02139 USA*

**Abstract**

This paper presents *lottery scheduling*, a novel randomized resource allocation mechanism. Lottery scheduling provides efficient, responsive control over the relative execution rates of computations. Such control is beyond the capabilities of conventional schedulers, and is desirable in systems that service requests of varying importance, such as databases, media-based applications, and networks. Lottery scheduling also supports modular resource management by enabling concurrent modules to insulate their resource allocation policies from one another. A *currency* abstraction is introduced to flexibly name, share, and protect resource rights. We also show that lottery scheduling can be generalized to manage many diverse resources, such as I/O bandwidth, memory, and access to locks. We have implemented a prototype lottery scheduler for the Mach 3.0 microkernel, and found that it provides flexible and responsive control over the relative execution rates

to rapidly focus available resources on tasks that are currently important [Dui90].

Few general-purpose schemes even come close to supporting flexible, responsive control over service rates. Those that do exist generally rely upon a simple notion of *priority* that does not provide the encapsulation and modularity properties required for the engineering of large software systems. In fact, with the exception of hard real-time systems, it has been observed that the assignment of priorities and dynamic priority adjustment schemes are often ad-hoc [Dei90]. Even popular priority-based schemes for CPU allocation such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

Existing *fair share* schedulers [Hen84, Kay88] and *microeconomic* schedulers [Fer88, Wal92] successfully ad-

# How to Assign Tickets?


Give Job A 50% of CPU, Job B 25%, Job C 25%

How can we use tickets to allow IO/interactive tasks to run quickly?
Assign them more tickets!

Can lottery scheduling lead to starvation?
a) Yes b) No

Can lottery scheduling lead to priority inversion?
a) Yes b) No

# Temporary Unfairness

Lose control over which job gets scheduled next.

Can suffer temporary bouts of unfairness

Given two jobs A and B with the same quantum that are each supposed to receive 50%,

U = finish time of first / finish time of last

As a function of quantum

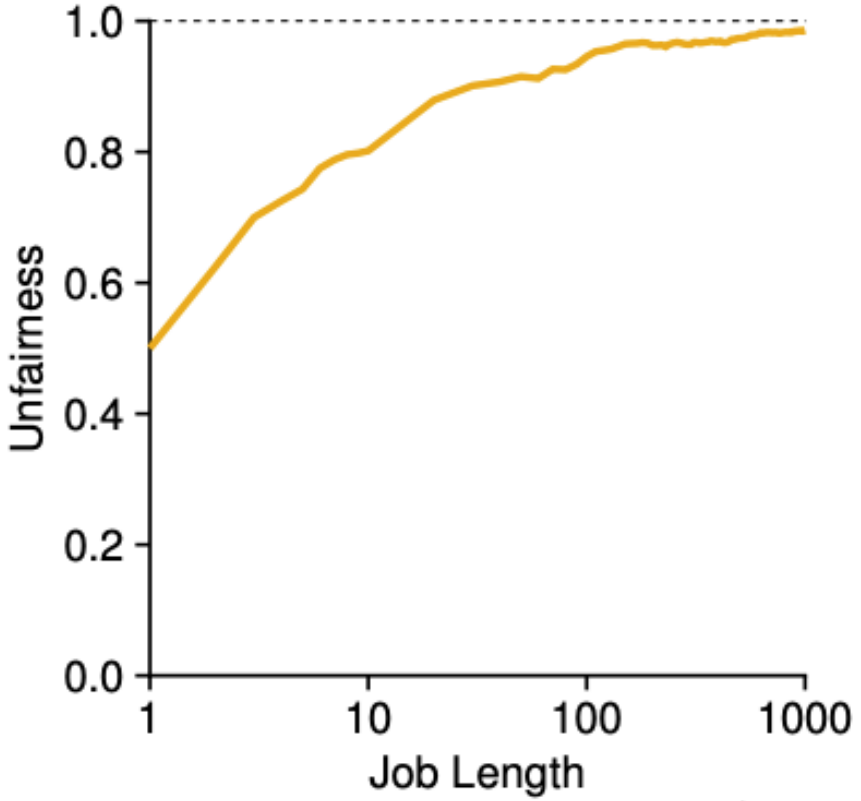



Figure 9.2: **Lottery Fairness Study**

# Stride Scheduling

Deterministic proportional fair sharing

Stride of each job is $\frac{big\ number\ W}{N_i}$

The larger your share of tickets $N_i$, the smaller your stride

W = 10,000,

A=100 tickets, B=50, C=250

A stride: 100, B: 200, C: 40

# Stride Scheduling

Each job as a pass counter.

Scheduler picks a job with lowest *pass*, runs it,
add its *stride* to its *pass*

Low-stride jobs (lots of tickets) run more often

# Stride Scheduling

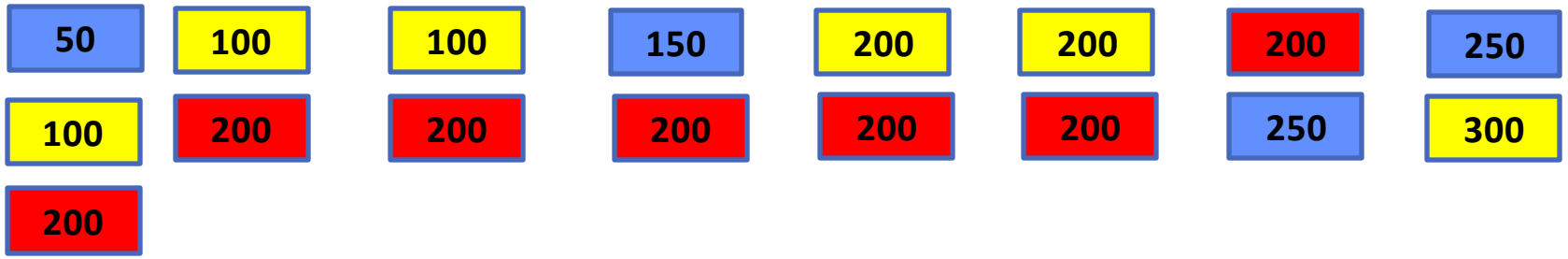
W = 10,000,

A=200 tickets, B=100 tickets, C=50 tickets

Strides:


Schedule


Ready Queue



# Complications with Stride Scheduling

What do you do when a job doesn't use its full time slice?

What do you do when jobs arrive or leave?

Long delays until a job is rescheduled

# Fluid Flow Model

Suppose that our processor/resource supported infinitely fine-grained context switching, e.g. after every instruction (for a CPU) or every bit sent (for a network link)

- Known as a "fluid flow system" in networking or Generalized Processor Sharing (GPS)

Proportional fair sharing could be implemented via weighted bit-by-bit round-robin in such a system

- During each round when a client has work, do a number of work units equal to its weight (e.g. run that many CPU instructions, or send that many bits on the network)

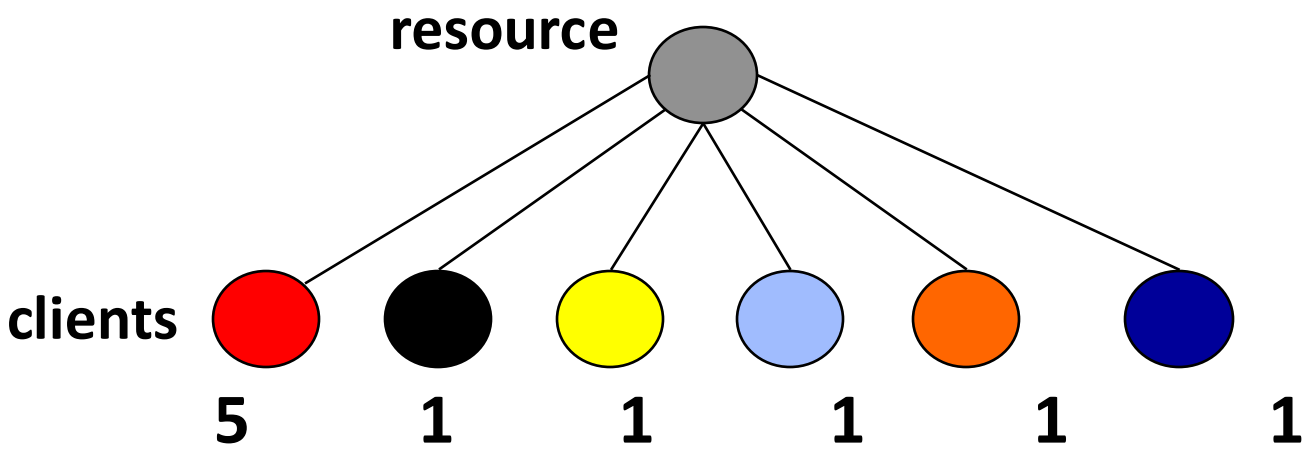
# GPS Example

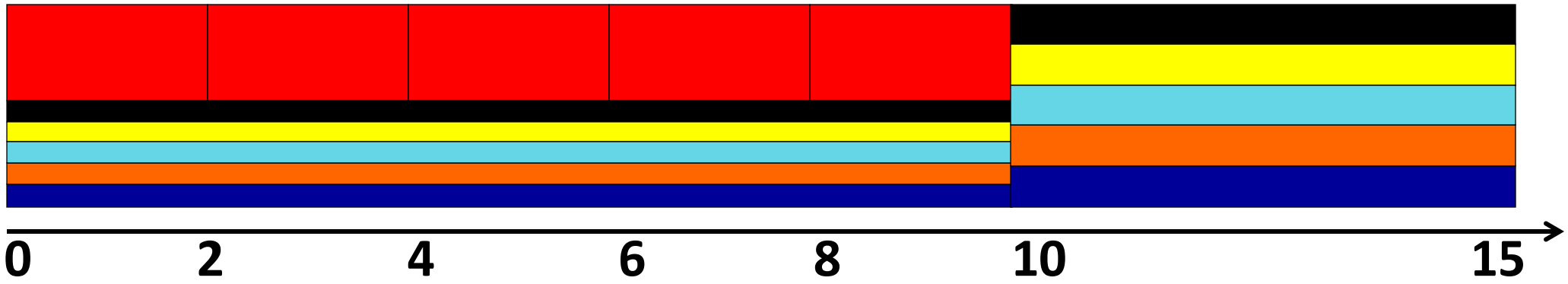
Red client has weight 5, and has 5 "packets" of work at time 0

– Other clients have weight 1 and always have work backlogged

Each packet has size 1 (for now)

Link capacity is 1 packet/second



# Packet Approximation of GPS

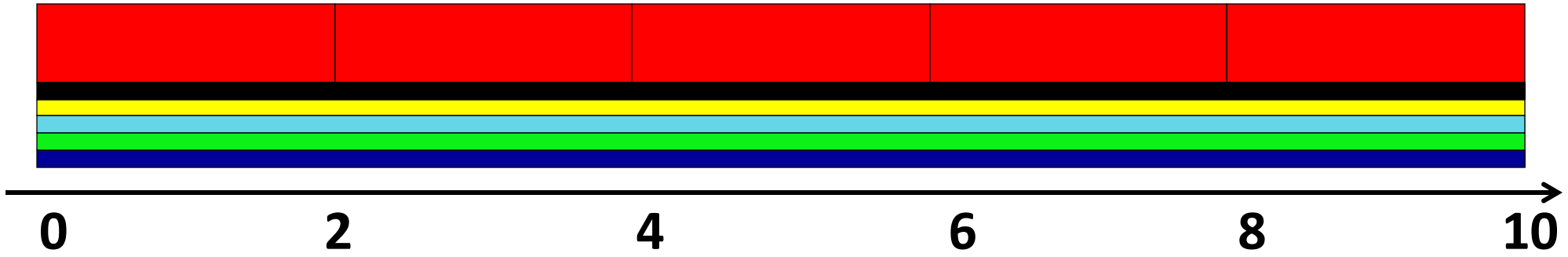
Emulate GPS

Select packet that finishes first in GPS assuming that there are no future arrivals

– Known as “Weighted Fair Queuing”, WFQ

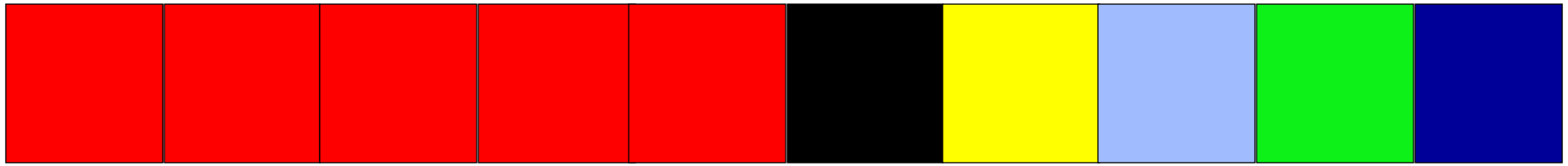
# Approximating GPS

Fluid GPS system service order




Approximation: select the first packet finishes in GPS

– Known as “Weighted Fair Queueing”, WFQ



# Nice Property of WFQ

The finish time of each packet in WFQ will be at most q + its finish time in GPS (where q is the maximum time quantum)

Every client receives at most q extra delay compared to GPS

# Implementation Challenges

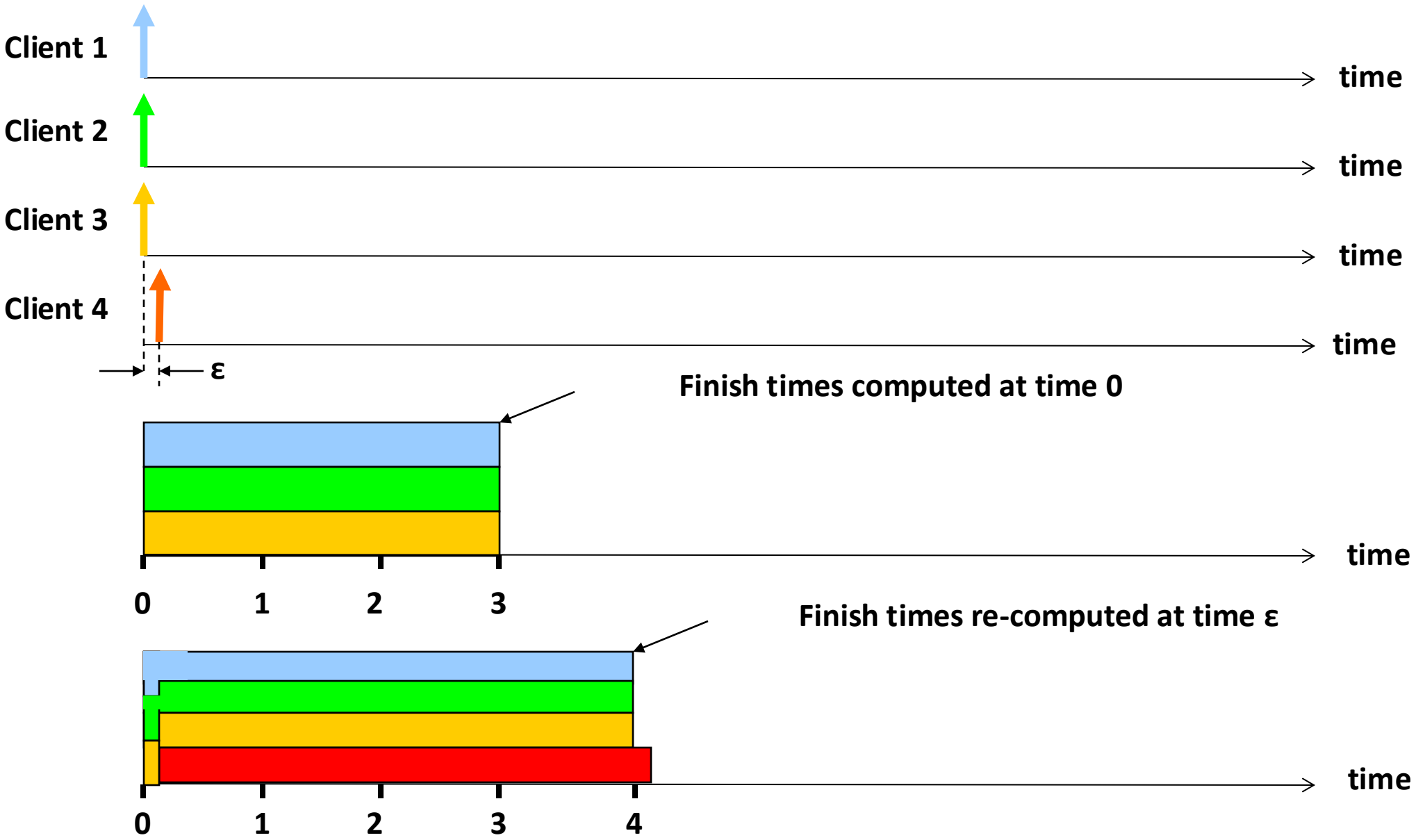
Need to compute the finish time of a packet in the fluid flow system…

… but the finish time may change as new packets arrive!

Need to update the finish times of *all* packets that are in service in the fluid flow system when a new packet arrives

– Very expensive; we might have 1000s of clients (threads, flows, etc)!

# Example: Each client has weight 1



# Solution: Virtual Time

Observation: while the finish times of packets may change when a new packet arrives, the order in which packets finish doesn't!

- Only the order is important for scheduling

Solution: instead of tracking packet finish times, maintain the number of rounds needed to send the remaining bits of the packet in GPS (virtual finish time)

- Virtual finish time doesn't change when other packets arrive; it is always equal to (length of packet) / $w_i$, where i is the client's weight, since each round of weighted bit-by-bit round-robin sends $w_i$ work from client i

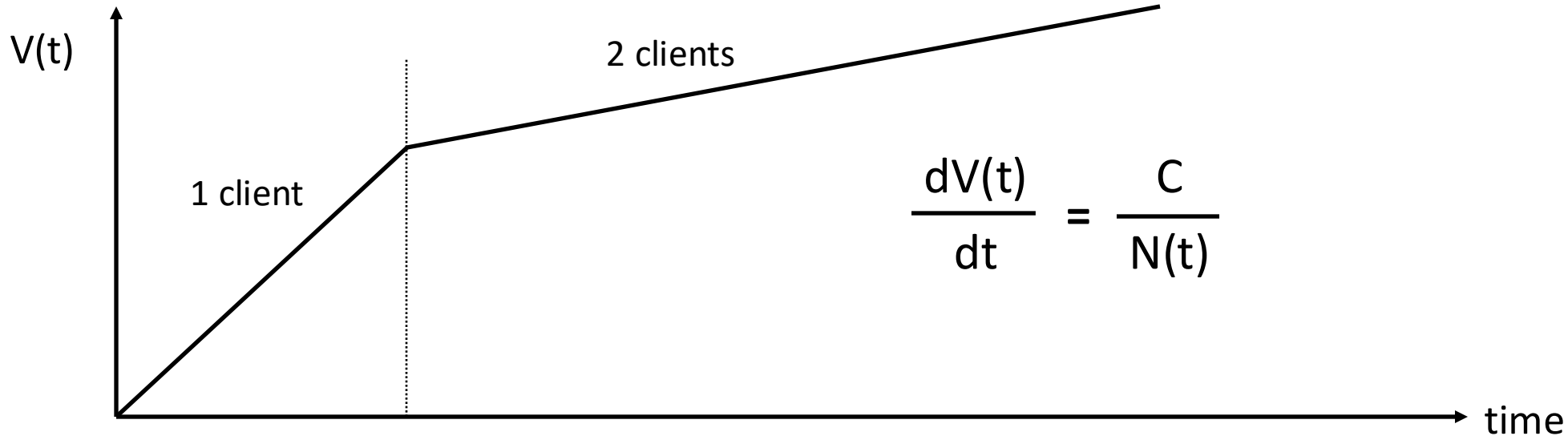
System virtual time – index of current round in weighted bit-by-bit round-robin

# System Virtual Time: *V*(t)

Measure total # of bit-by-bit round-robin rounds served, instead of time

V(t) slope = normalized rate at which every backlogged client receives service in the fluid flow system

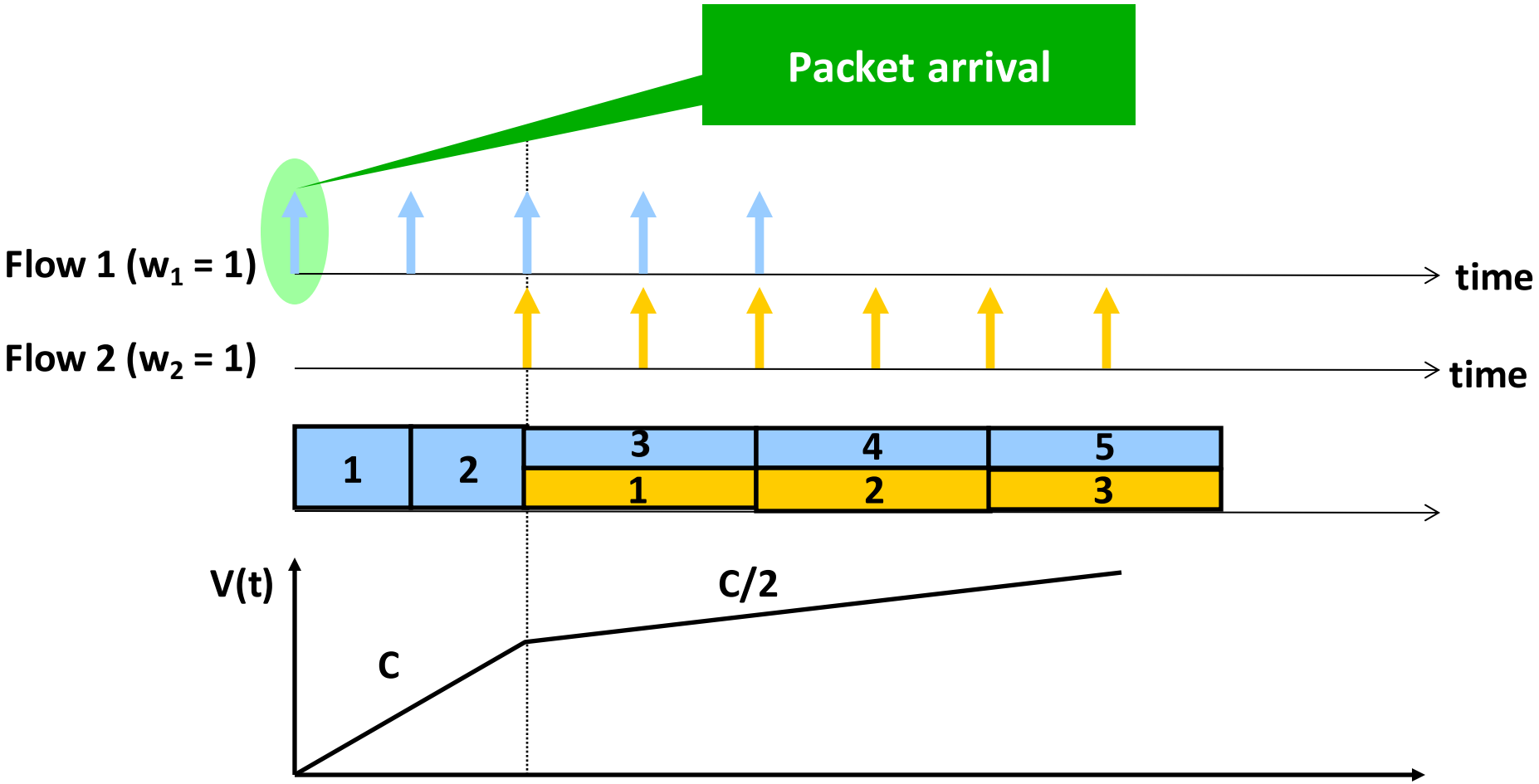
- *C*: resource capacity
- *N*(t): total weight of backlogged clients in fluid flow system at time t




$$\frac{dV(t)}{dt} = \frac{C}{N(t)}$$

# System Virtual Time V(t): Example

V(t) increases inversely proportionally to the sum of the weights of the backlogged flows



# Implementing WFQ with Virtual Finish Times

Define

- $F_i^k$ virtual finish time of packet $k$ of client $i$
- $a_i^k$ arrival time of packet $k$ of client $i$
- $L_i^k$ length of packet $k$ of client $i$
- $w_i$ weight of client $i$

The finish time of packet $k+1$ of client $i$ is:

$$F_i^{k+1} = \max(V(a_i^{k+1}), F_i^k) + L_i^{k+1} / w_i$$

Arrival round

Round when last packet of client *i* finishes

# of rounds it takes to serve packet *k+1*

Round by which packet *k+1* is served

# Implementing WFQ with Virtual Finish Times

Each time a client gets a new packet of work (e.g., on the CPU, whenever a process becomes runnable), compute the virtual finish time of that packet

- Assume the duration is a full time quantum for CPU scheduling (can have other heuristics to identify IO-bound tasks)

Maintain a red-black tree of runnable processes, sorted by virtual finish time

Always schedule the process with the earliest virtual finish time

# Early Eligible Virtual Deadline First (EEVDF)

## EEVDF Scheduler

English

The "Earliest Eligible Virtual Deadline First" (EEVDF) was first introduced in a scientific publication in 1995 [1]. The Linux kernel began transitioning to EEVDF in version 6.6 (as a new option in 2024), moving away from the earlier Completely Fair Scheduler (CFS) in favor of a version of EEVDF proposed by Peter Zijlstra in 2023 [2-4]. More information regarding CFS can be found in CFS Scheduler.

www.kernel.org/doc/html/next/scheduler/sched-eevdf.html

**A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems**

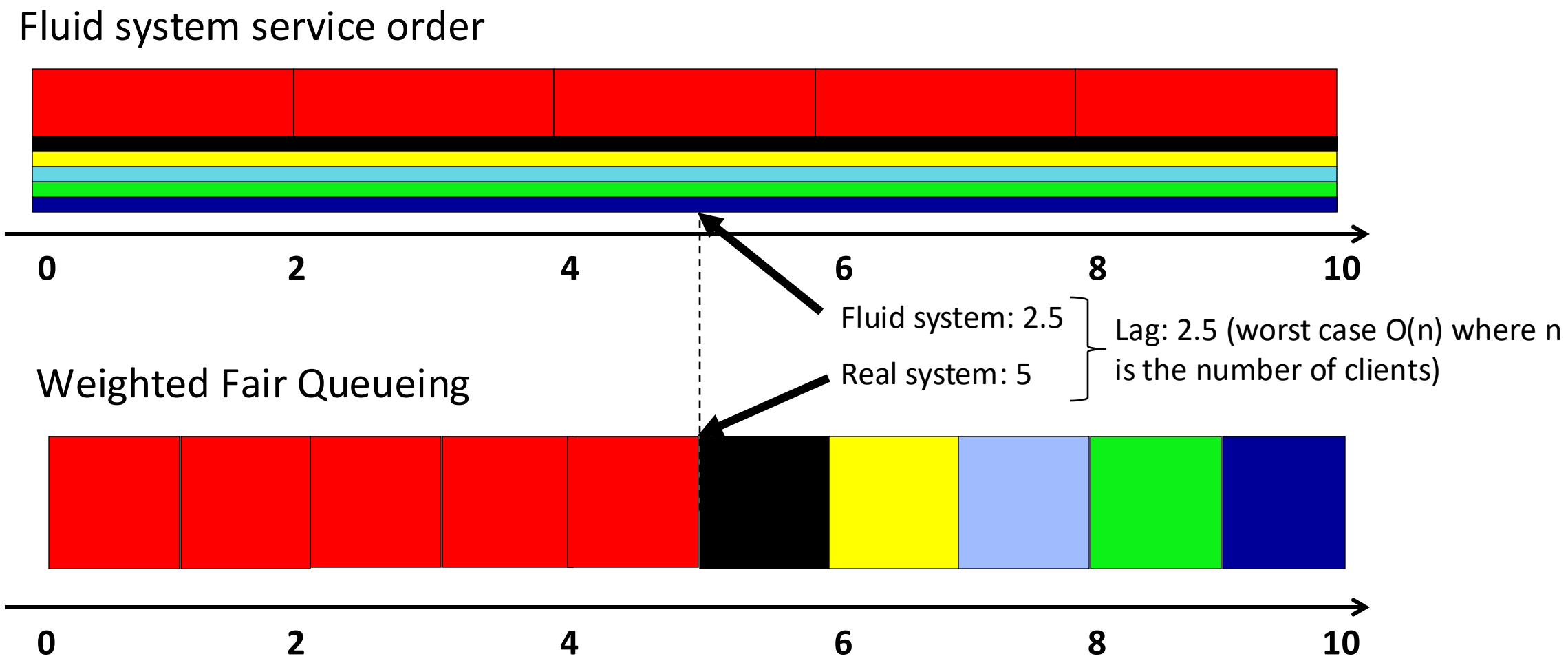
Ion Stoica * Hussein Abdel-Wahab † Kevin Jeffay‡ Sanjoy K. Baruah §
Johannes E. Gehrke ¶ C. Greg Plaxton ||

**Abstract**

*We propose and analyze a proportional share resource allocation algorithm for realizing real-time performance in time-shared operating systems. Processes are assigned a weight which determines a share (percentage) of the resource they are to receive. The resource is then allocated in discrete-sized time quanta in such a manner that each process makes progress at time quantum. In addition, the algorithm provides support for dynamic operations, such as processes joining or leaving the competition, and for both fractional and non-uniform time quanta. As a proof of concept we have implemented a prototype of a CPU scheduler under FreeBSD. The experimental results shows that our implementation performs within the theoretical bounds and hence supports real-time execution in a general purpose operating system.*

# What Problem Does EEVDF Try to Solve?

Minimize lag: the difference between service received in real system vs fluid flow (idealized) system



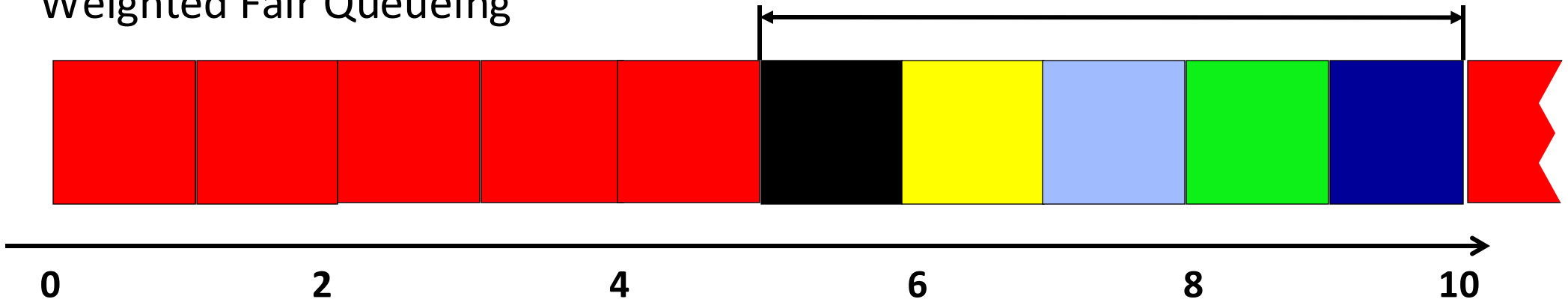
# Why is This Bad?

Minimize lag: the difference between service received in real system vs fluid flow (idealized) system

Fluid system service order


Red client has 1/2 of resource capacity but can be denied service for n slots!

Weighted Fair Queueing



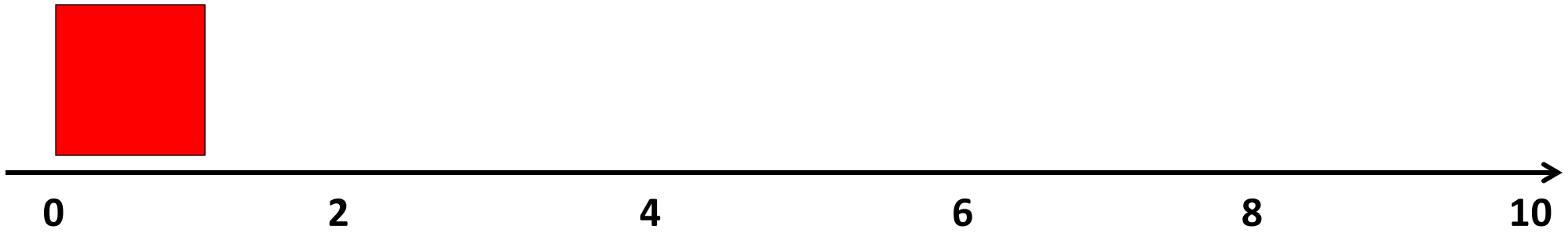
# How?

Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time <= current virtual time

Fluid system service order


Only first of the red packets is eligible and has earliest deadline among all eligible packets so schedule it

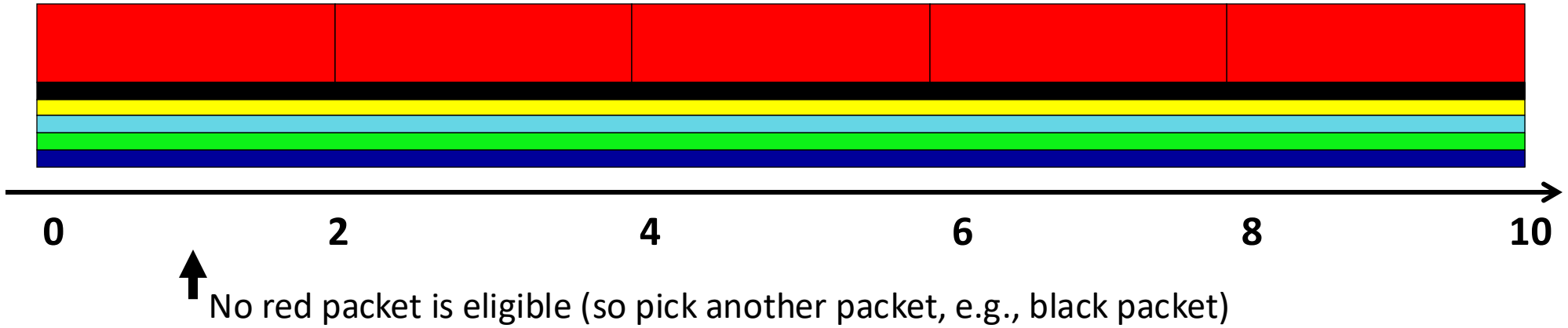
EEVDF



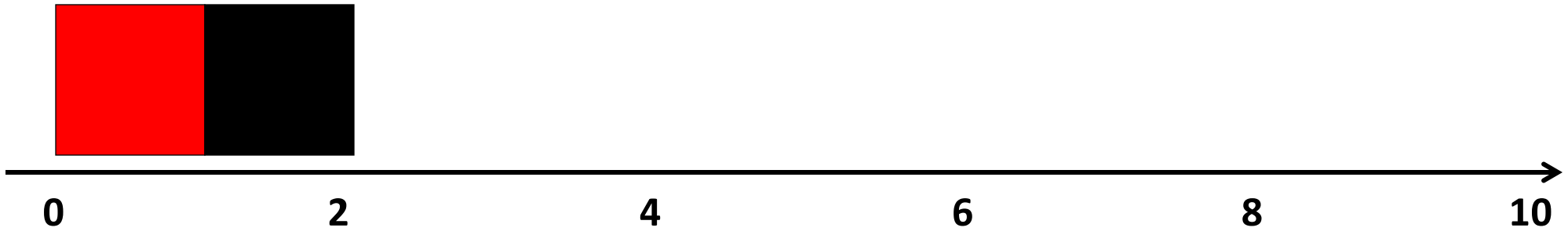
# How?

Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time <= current virtual time

Fluid system service order



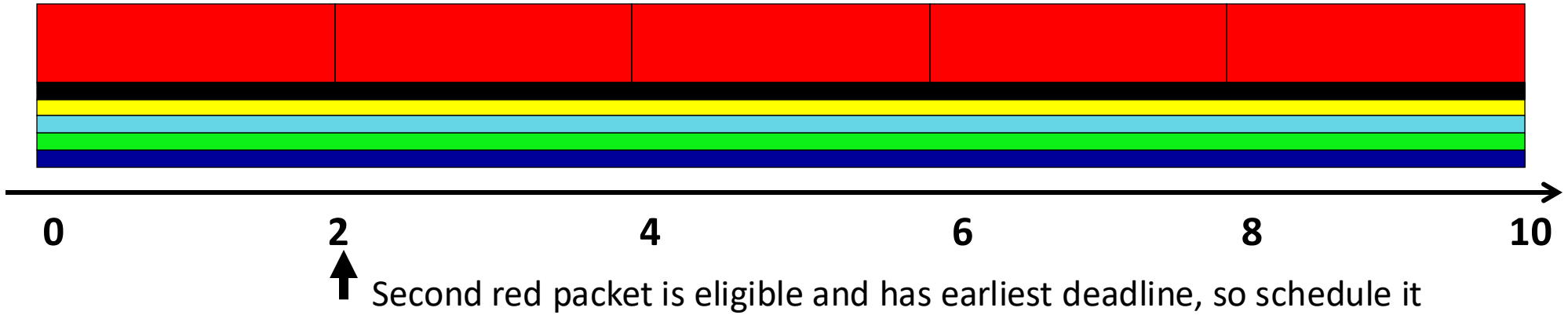

EEVDF



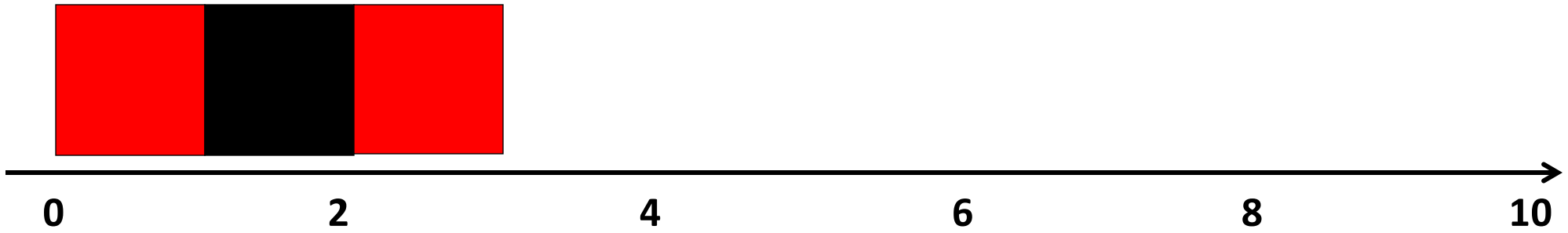
# How?

Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time <= current virtual time

Fluid system service order




EEVDF



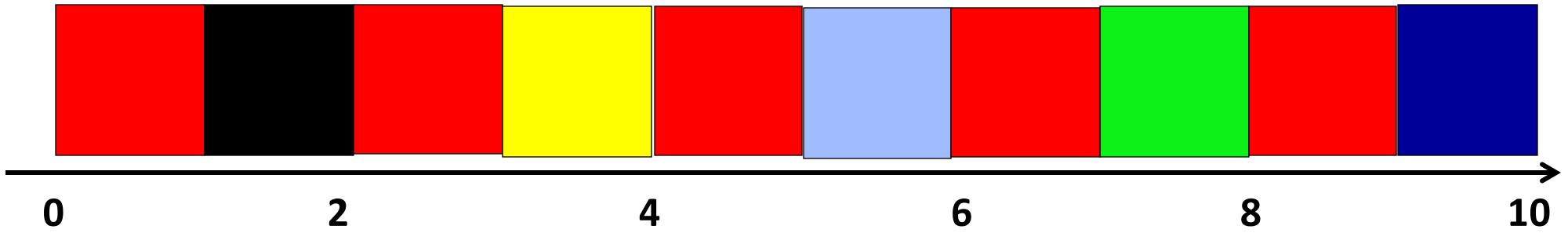
# How?

Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time <= current virtual time

Fluid system service order


Lag <= 1 (independent on number of clients)

EEVDF



# Comparison of Proportional Sharing Algorithms

n: number of clients

q: length of max time quantum

| | Finish time delay vs fluid flow system | Lag vs fluid flow system | Scheduler complexity |
|---|---|---|---|
| Lottery scheduling | O(sqrt(n))*q average case | O(sqrt(n))*q average case | O(1) |
| Stride scheduling | O(log(n))*q | O(log(n))*q | O(log(n)) |
| WFQ | q | O(n)*q | O(log(n)) |
| EEVDF | q | O(1)*q | O(log(n)) |

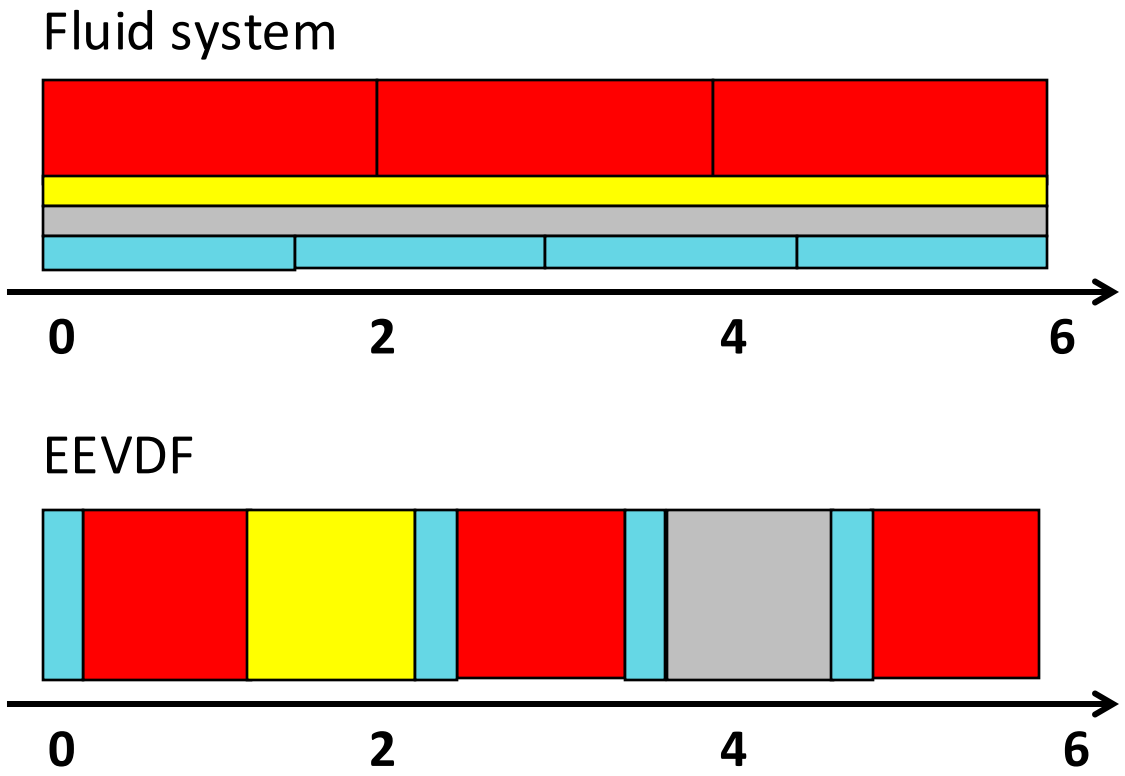
# Configuring "Packet Lengths" in CPU Scheduling

Remember that EEVDF considers packet lengths, aims to finish shorter packets faster!

In Linux EEVDF, tasks can specify their preferred time slice via sched_setattr(), so that developers can ask for interactive tasks to be scheduled faster

Example:

- client 1: weight 3, slice 1s
- client 2: weight 1, slice 1s
- client 3: weight 1, slice 1s
- client 4: weight 1, slice 1/4 s



# Configuring “Packet Lengths” in CPU Scheduling

Can clients “cheat” and get more CPU time by requesting a smaller time slice?

a) Yes  b) No

# Implementing EEVDF in the Kernel (Rough Sketch)

For each task, track it lag in virtual time, i.e. service it received minus service it should have received under GPS

– Positive means we owe it time, negative means it ran ahead of GPS

Only tasks with lag ≥ 0 are eligible; for each of those, compute a virtual deadline based on eligible_time + time_slice, and store them in a sorted red-black tree

Schedule the task with the lowest deadline from the red-black tree

As tasks that had lag < 0 become eligible, compute their deadlines and add to tree

More details: https://lwn.net/Articles/925371/, https://hackmd.io/@Kuanch/eevdf

# What to Do When Tasks Sleep?

If a task goes to sleep (e.g. on I/O), we probably want to remember its lag and return it with that lag when it wakes up

– What would be the problem if we reset lag to 0?


But if too many tasks wake up at once, this might result in arbitrarily delaying existing tasks in the system, which is not great

In practice, Linux decays the lag after some time (heuristics still being explored)

# How do “nice” Values Map to Weights in Linux?

```
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,
};
```

Each priority level is 1.25x the weight of the next lower one

i.e. weight = $1024 / 1.25^{nice}$

# Summary: Schedulers in Linux

| | |
|---|---|
| O(n) scheduler<br>Linux 2.4 to Linux 2.6 | Did not scale with large number of processes |
| O(1) scheduler<br>Linux 2.6 to 2.6.22 | MLFQ, but got very complex |
| CFS scheduler<br>Linux 2.6.23 to 6.6 | Proportional Fair Sharing, but can have suboptimal lag for interactive tasks |
| EEVDF scheduler<br>Linux 6.6 onward | Proportional Fair Sharing with low lag, fewer heuristics than CFS |

# Scheduling on Multicores

Algorithmically, not a huge difference from single-core scheduling

Implementation-wise, helpful to have *per-core* scheduling data structures

- Cache coherence

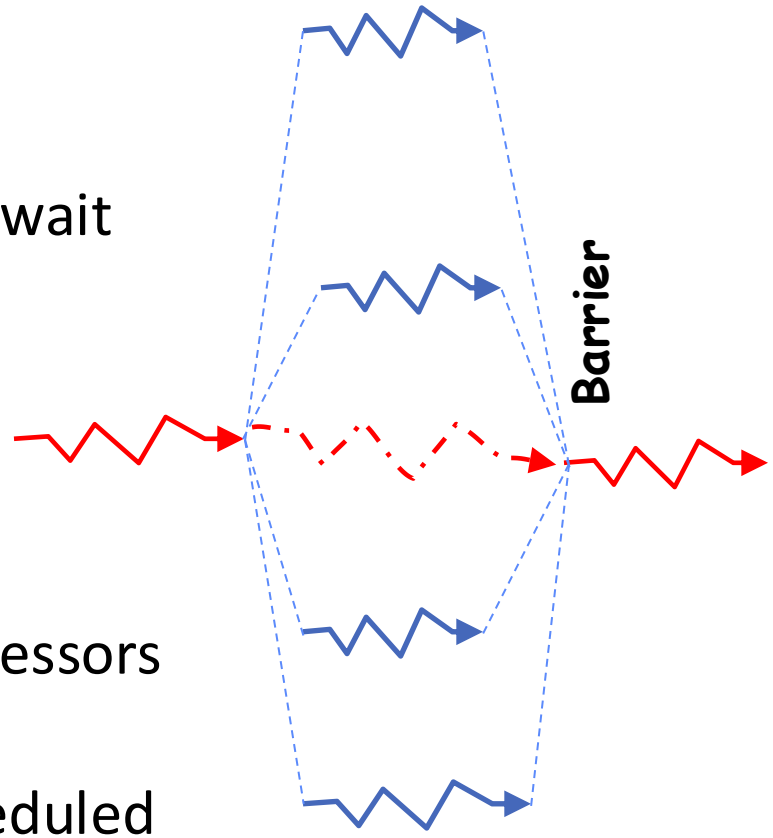
*Affinity scheduling*: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU

- Cache reuse, branch prediction
- Example for O(1) scheduler: one set of queues/core with background rebalancing

# Gang Scheduling and Parallel Applications

When multiple threads work together on a multi-core system, try to schedule them together

- Makes spin-waiting more efficient (inefficient to spin-wait for a thread that's suspended)
- Multiple phases of parallel and serial execution




Additionally: OS informs a parallel program how many processors its threads are scheduled on (*Scheduler Activations*)

- Application adapts to number of cores that it has scheduled
- "Space sharing" with other parallel programs can be more efficient, because parallel speedup is often sublinear with the number of cores