CS162
Operating Systems and Systems Programming
Lecture 12

Scheduling 2: Classic Policies (Con't), Case Studies, Realtime, Starvation

October 10th, 2024

Prof. Ion Stoica

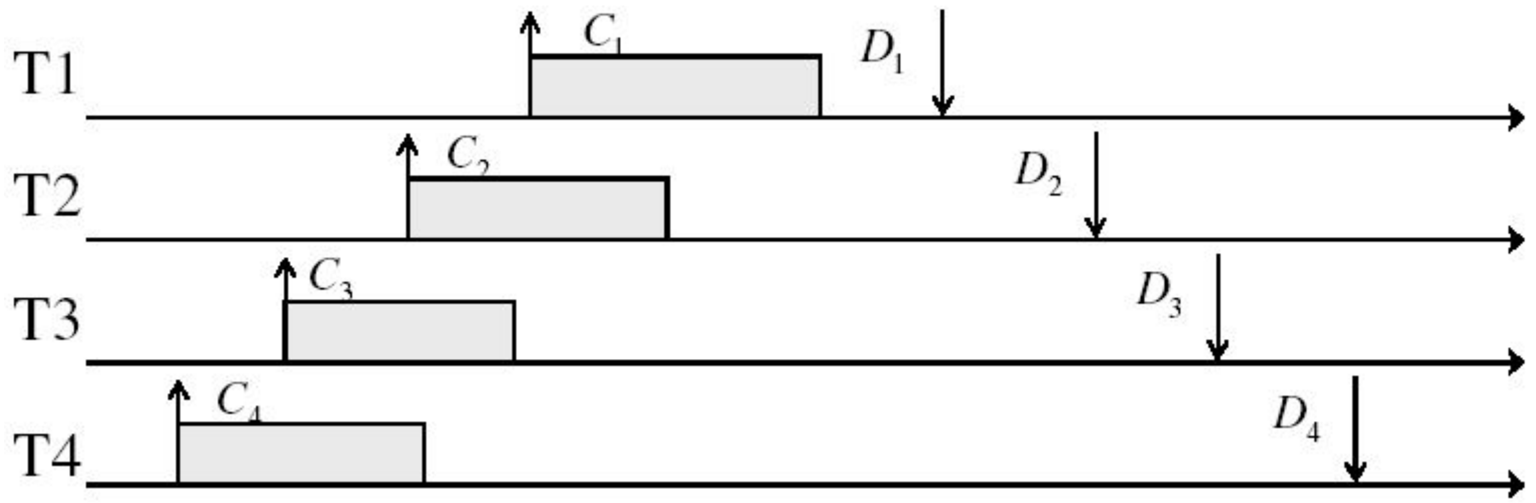
http://cs162.eecs.Berkeley.edu

Real-Time Scheduling

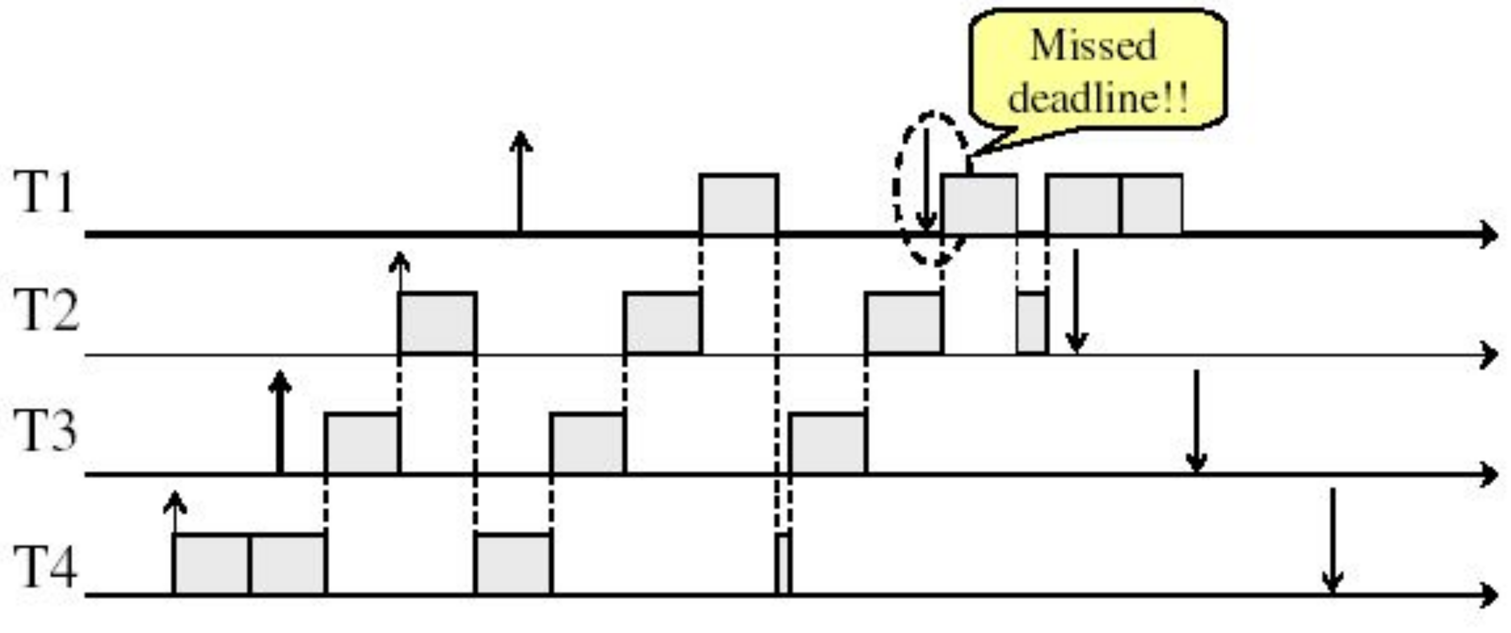
- Goal: Predictability of Performance!
  - We need to predict with confidence worst case response times for systems!
  - In RTS, performance guarantees are:
    - » Task- and/or class centric and often ensured a priori
  - In conventional systems, performance is:
    - » System/throughput oriented with post-processing (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!
- Hard real-time: for time-critical safety-oriented systems
  - Meet all deadlines (if at all possible)
  - Ideally: determine in advance if this is possible
  - Earliest Deadline First (EDF), Least Laxity First (LLF), Rate-Monitonic Scheduling (RMS), Deadline Monotonic Scheduling (DM)
- Soft real-time: for multimedia
  - Attempt to meet deadlines with high probability
  - Constant Bandwidth Server (CBS)

Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:



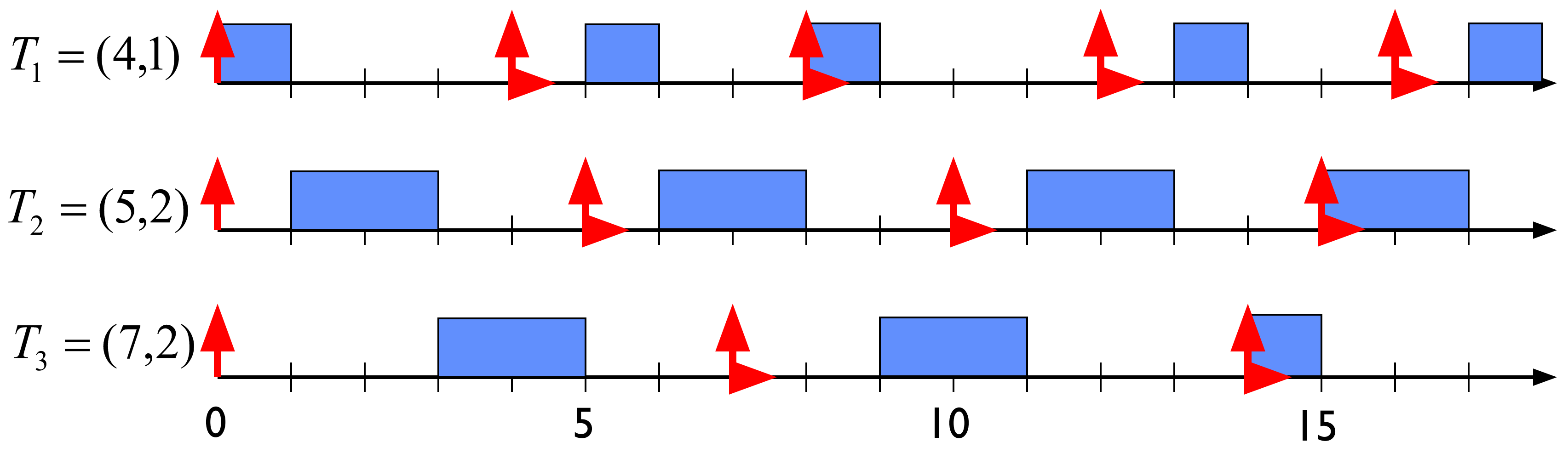
Example: Round-Robin Scheduling Doesn't Work



Time

Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P_i, C_i) for each task i
- Preemptive priority-based dynamic scheduling:
  - Each task is assigned a (current) priority based on how close the absolute deadline is (i.e. $D_i^{t+1} = D_i^t + P_i$for each task!)
  - The scheduler always schedules the active task with the closest absolute deadline



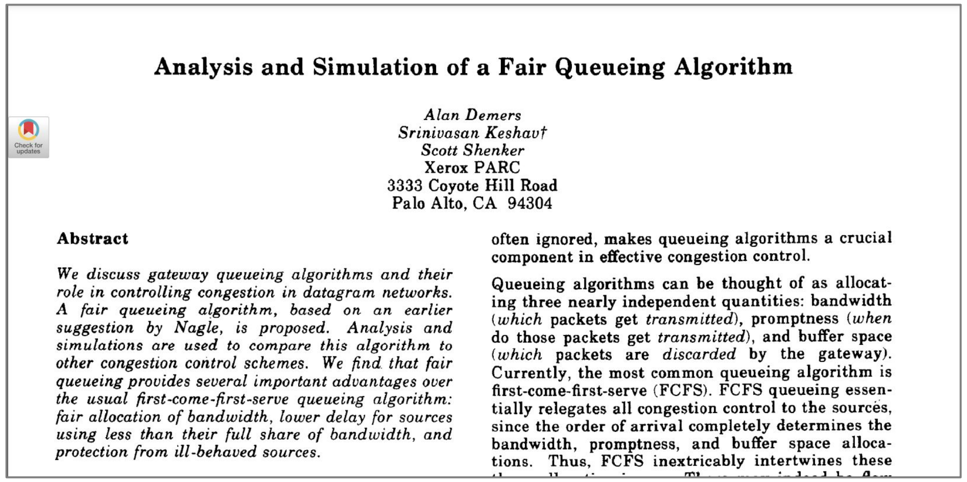
EDF Feasibility Testing

- Even EDF won't work if you have too many tasks
- For n tasks with computation time C and deadline D, a feasible schedule exists if:

$$\sum^{n}\left(\frac{C_i}{D_i}\right) \leq 1$$

Scheduling Fairness

- Two identical processes should get
  - the "same" service, experience the "same" performance
- *Work-conserving* scheduler: does not leave the CPU idle when there is work to do
- Note: fairness avoid starvation
- Example: round-robin
- Next:
  - Fair sharing / Fair queueing / Max-min fairness)
  - Originated in networking
  - Later applied to CPU scheduling and other resources (lottery scheduling)



Analysis and Simulation of a Fair Queueing Algorithm

Alan Demers
Srinivasan Keshav†
Scott Shenker
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

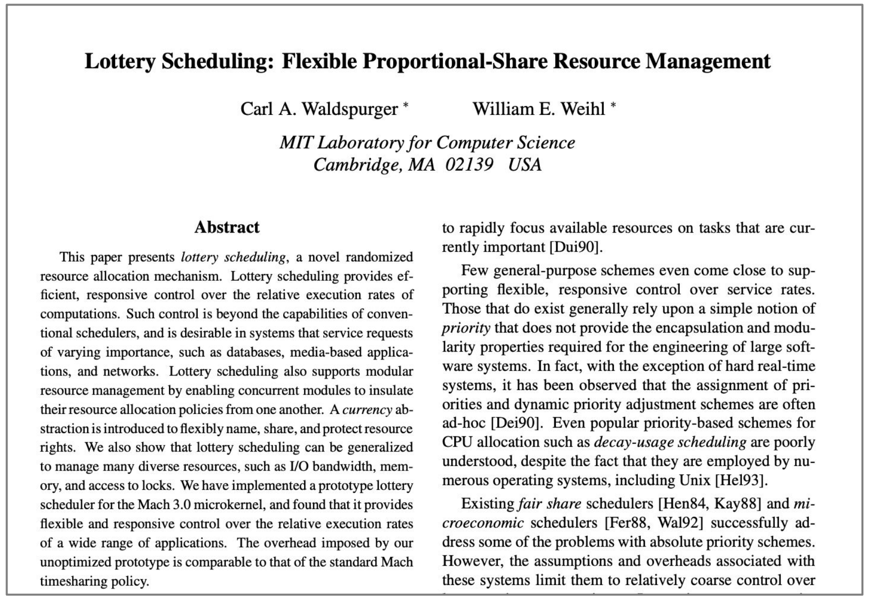
Abstract

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 to compare this algorithm to other congestion control schemes. We find that fair queueing provides several important advantages over the usual first-come-first-serve queueing algorithm: fair allocation of bandwidth, lower delay for sources using less than their full share of bandwidth, and protection from ill-behaved sources.

often ignored, makes queueing algorithms a crucial component in effective congestion control.

Queueing algorithms can be thought of as allocating three nearly independent quantities: bandwidth (*which* packets get *transmitted*), promptness (*when* do those packets get *transmitted*), and buffer space (*which* packets are *discarded* by the gateway). Currently, the most common queueing algorithm is first-come-first-serve (FCFS). FCFS queueing essentially relegates all congestion control to the sources, since the order of arrival completely determines the bandwidth, promptness, and buffer space allocations. Thus, FCFS inextricably intertwines these

SIGCOMM '89



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 of a wide range of applications. The overhead imposed by our unoptimized prototype is comparable to that of the standard Mach timesharing policy.

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

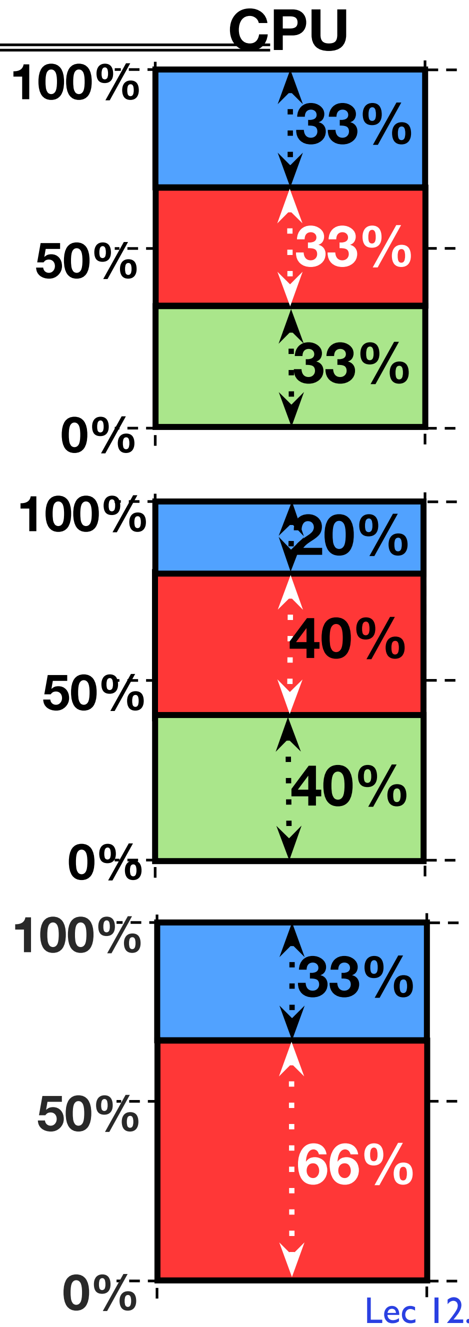
Few general-purpose schemes even come close to supporting flexible, responsive control over service rates. Those that do exist generally rely upon a simple notion of *priority* that does not provide the encapsulation and modularity properties required for the engineering of large software systems. In 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 address some of the problems with absolute priority schemes. However, the assumptions and overheads associated with these systems limit them to relatively coarse control over

OSDI '94

Single Resource: Fair Sharing

- *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 if a user wants less than its fair share
  - E.g. user 1 wants no more than 20%
- Generalized by ***weighted max-min fairness***
  - Give weights to users according to importance
  - User 1 gets weight 1, user 2 weight 2



Why Max-Min Fairness?

- *Weighted Fair Sharing / Proportional Shares*
  - User 1 gets weight 2, user 2 weight 1
- *Priorities*
  - Give user 1 weight 1000, user 2 weight 1
- *Revervations*
  - Ensure user 1 gets 10% of a resource
  - Give user 1 weight 10, sum weights ≤ 100
- *Deadline-based scheduling*
  - Given a user job's demand and deadline, compute user's reservation/weight
- *Isolation*
  - Users cannot affect others beyond their share

Widely Used

- *OS:* proportional sharing, lottery, Linux's cfs, …
- *Networking:* wfq, wf2q, sfq, drr, csfq, ...
- *Datacenters:* Hadoop's fair sched, Dominant Resource Fairness (DRF)

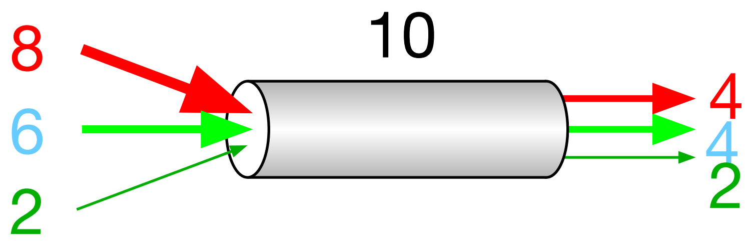
Fair Queueing: Max-min Fairness originated in Networking

- Fair queueing explained in a fluid flow system: reduces to bit-by-bit round robin among flows
  - Each flow receives $min(r_i, f)$, where
    - » r_i – flow arrival rate
    - » f – link fair rate (see next slide)
- Weighted Fair Queueing (WFQ) – associate a weight with each flow [Demers, Keshav & Shenker '89]
  - In a fluid flow system it reduces to bit-by-bit round robin
- WFQ in a fluid flow system □ Generalized Processor Sharing (GPS) [Parekh & Gallager '92]

Fair Rate Computation

- If link congested, compute f such that

$$\sum_i \min(r_i, f) = C$$

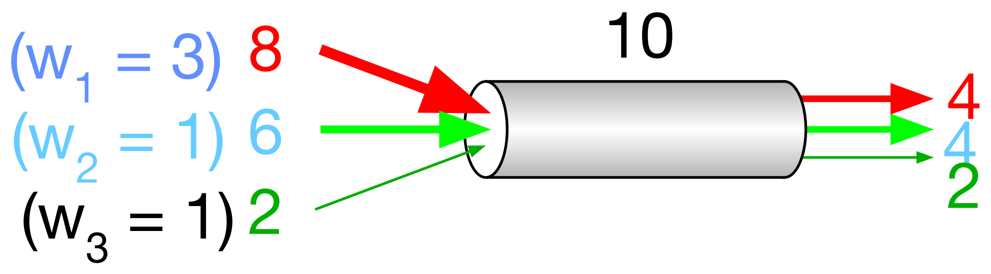


f = 4:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

Fair Rate Computation

- Associate a weight w_i with each flow i
- If link congested, compute f such that

$$\sum_i \min(r_i, f \times w_i) = C$$



f = 2:
min(8, 2*3) = 6
min(6, 2*1) = 2
min(2, 2*1) = 2

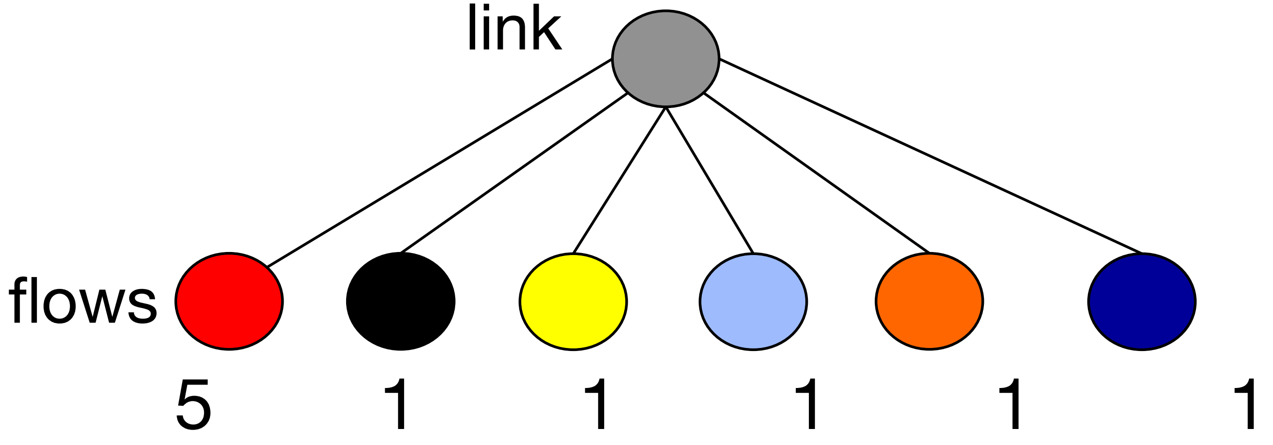
Fluid Flow System

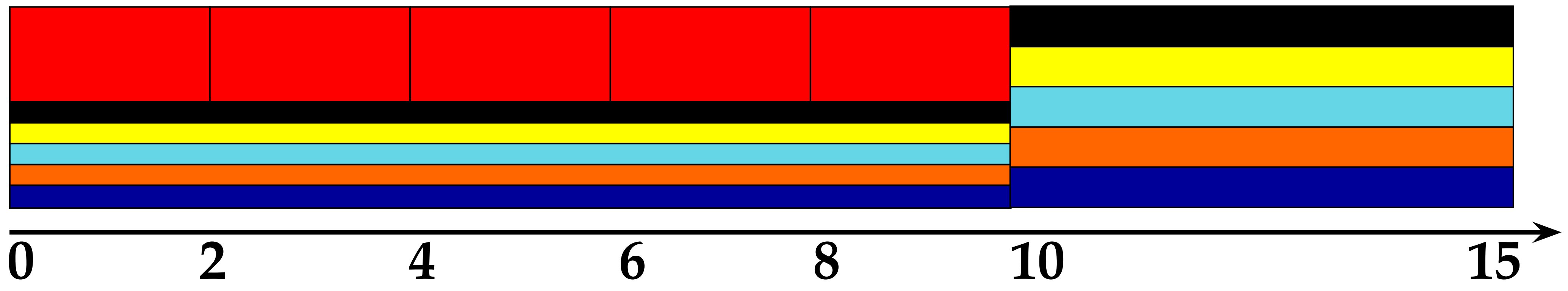
- Flows can be served one bit at a time
  - Fluid flow system, also known as Generalized Processor Sharing (GPS) [Parekh and Gallager '93]

- WFQ can be implemented using bit-by-bit weighted round robin in GPS model
  - During each round from each flow that has data to send, send a number of bits equal to the flow's weight

Generalized Processor Sharing Example

- Red session has packets backlogged between 0 and 10
  – Other sessions have packets continuously backlogged
- Each packet has size 1
- Link capacity is 1



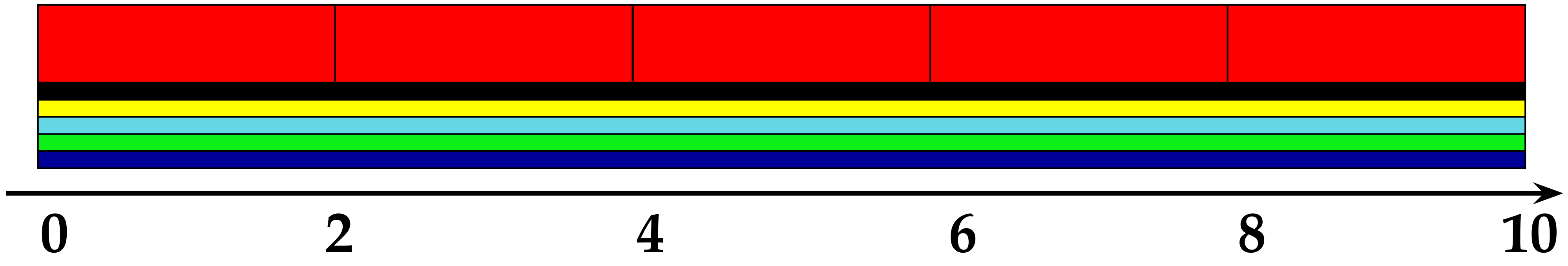


Packet Approximation of GPS

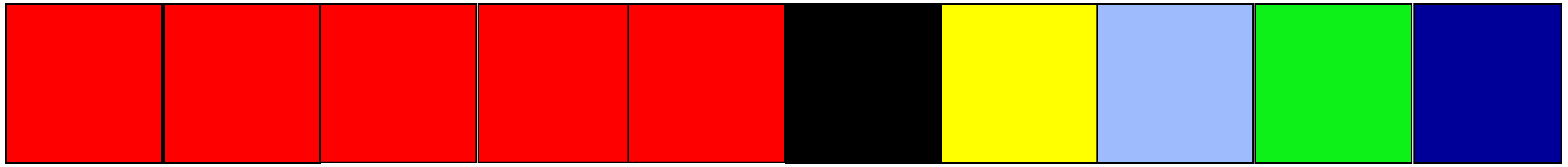
- Emulate GPS
- Select packet that finishes first in GPS assuming that there are no future arrivals

Approximating GPS with WFQ

- Fluid GPS system service order



- Weighted Fair Queueing
  - select the first packet that finishes in GPS

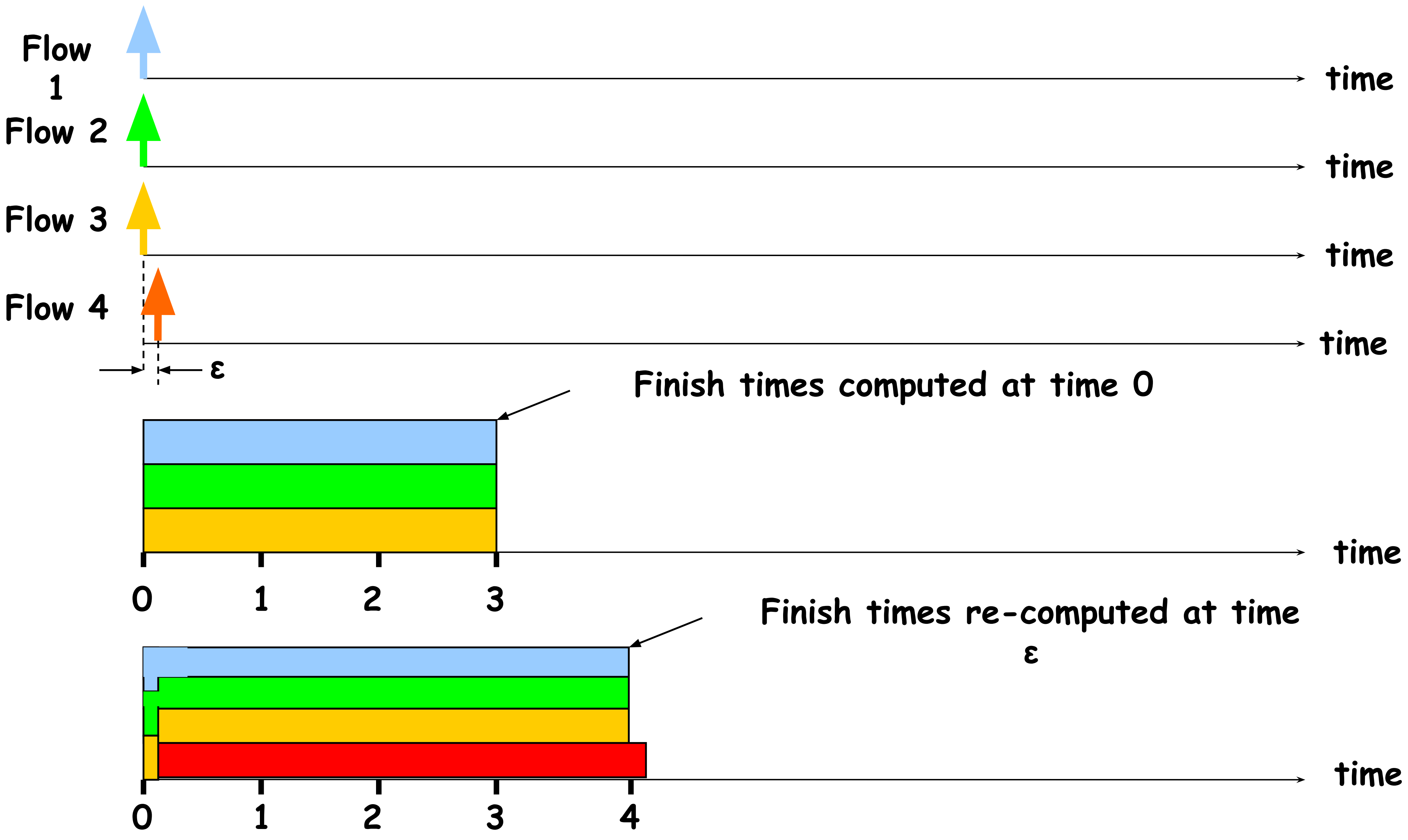


Implementation Challenge

- 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
  – But this is very expensive; a high-speed router may need to handle hundred of thousands of flows!

Example: Each flow has weight 1

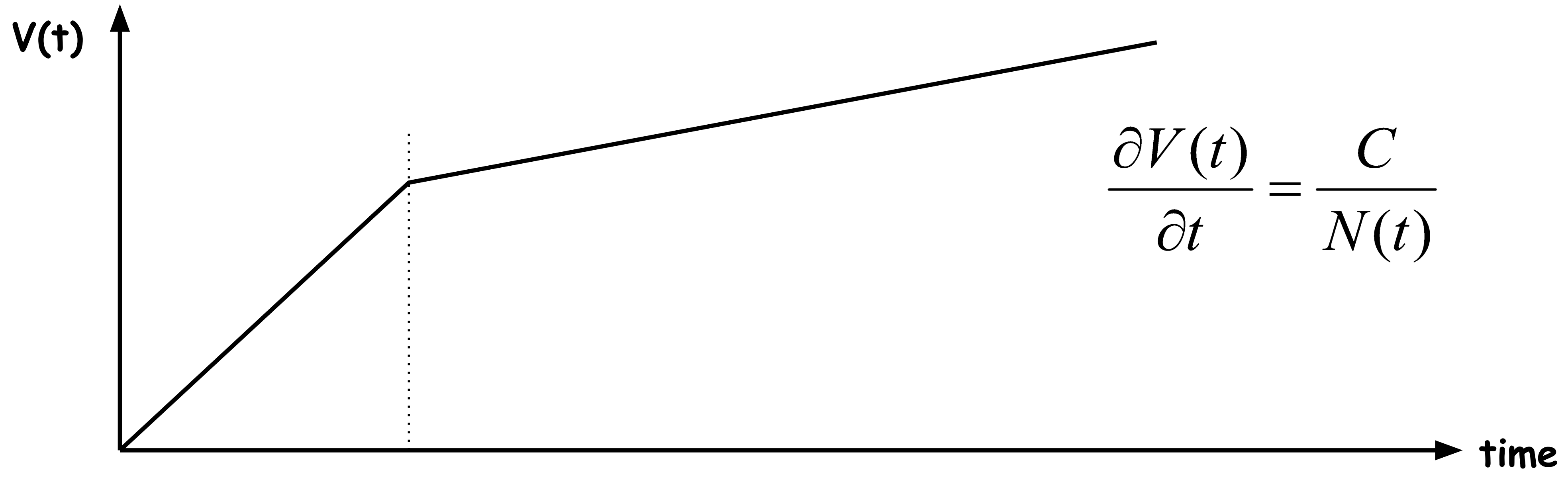

Flow 1
Flow 2
Flow 3
Flow 4
time
time
time
time
ε
Finish times computed at time 0
0
1
2
3
time
Finish times re-computed at time ε
0
1
2
3
4
time

Solution: Virtual Time

- Key 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 the packet finish time maintain the number of rounds needed to send the remaining bits of the packet (virtual finishing time)
  - Virtual finishing time doesn't change when the packet arrives
- System virtual time – index of the round in the bit-by-bit round robin scheme

System Virtual Time: *V*(t)

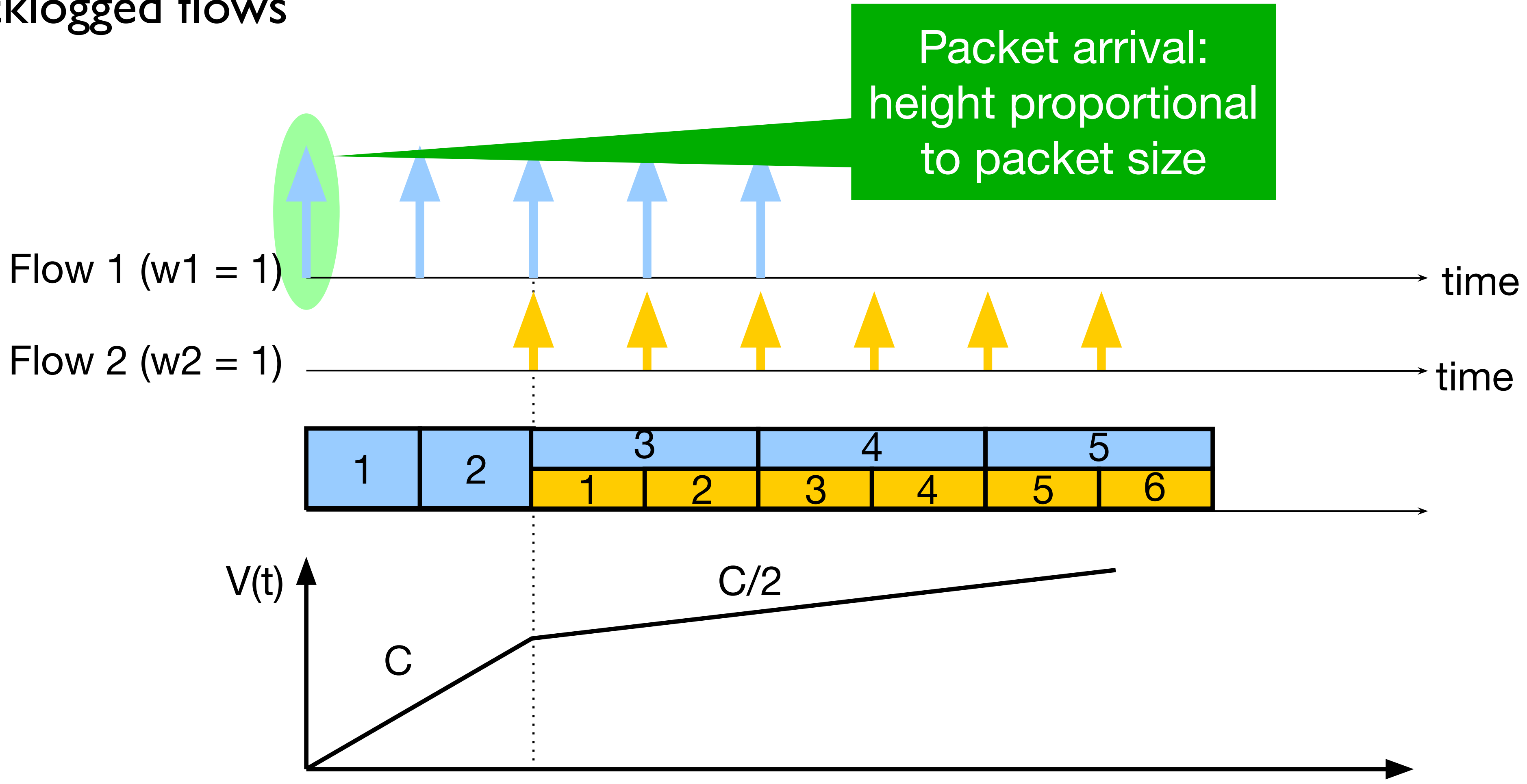
- Measure service, instead of time
- V(t) slope – normalized rate at which every backlogged flow receives service in the fluid flow system
  - *C* – link capacity
  - *N*(t) – total weight of backlogged flows in fluid flow system at time t



$$\frac{\partial V(t)}{\partial t} = \frac{C}{N(t)}$$

System Virtual Time (V(t)): Example

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



Fair Queueing Implementation

- Define
  - F_i^k virtual finishing time of packet k of flow i
  - a_i^k arrival time of packet k of flow i
  - L_i^k length of packet k of flow i
  - w_i weight of flow i
- The finishing time of packet $k+1$ of flow i is

Current round

Round when last packet of flow *i* finishes

# of rounds it takes to serve new packet (i.e., packt *k+1* of flow *i*)

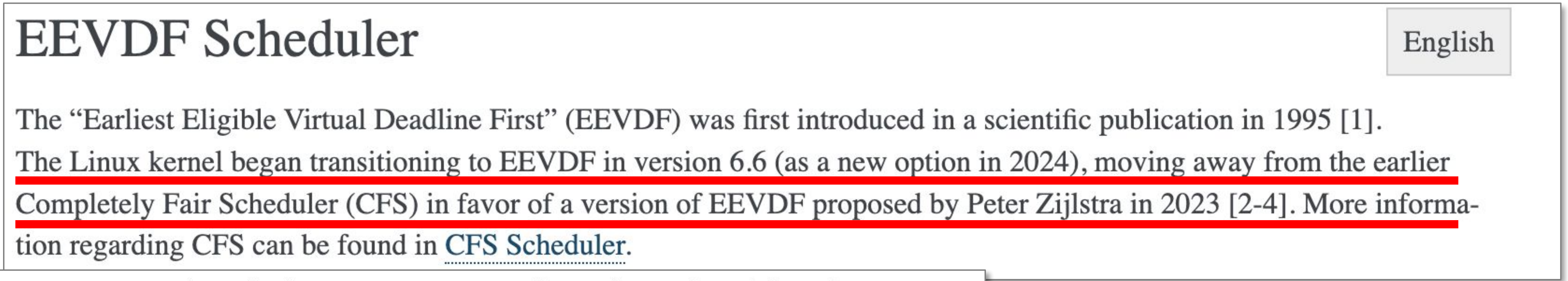
$$F_i^{k+1} = \max(V(a_i^{k+1}), F_i^k) + L_i^{k+1} / w_i$$

Round by which packet *k+1* is served

Administrivia

- Midterm I:
  - Grading done today by end of the week. Sorry for the delay!
  - Solutions will be up off the Resources page
- Project 1 final report is due Monday, October 14th
- Also due Monday: Peer evaluations
  - These are a required mechanism for evaluating group dynamics
  - Project scores are a zero-sum game
    - » In the normal/best case, all partners get the same grade
    - » In groups with issues, we may take points from non-participating group members and give them to participating group members!
- Homework 2 due Sunday 10/13
- Homework 3:
  - Release on Monday 10/14
  - Can be done in Rust (if you want!)

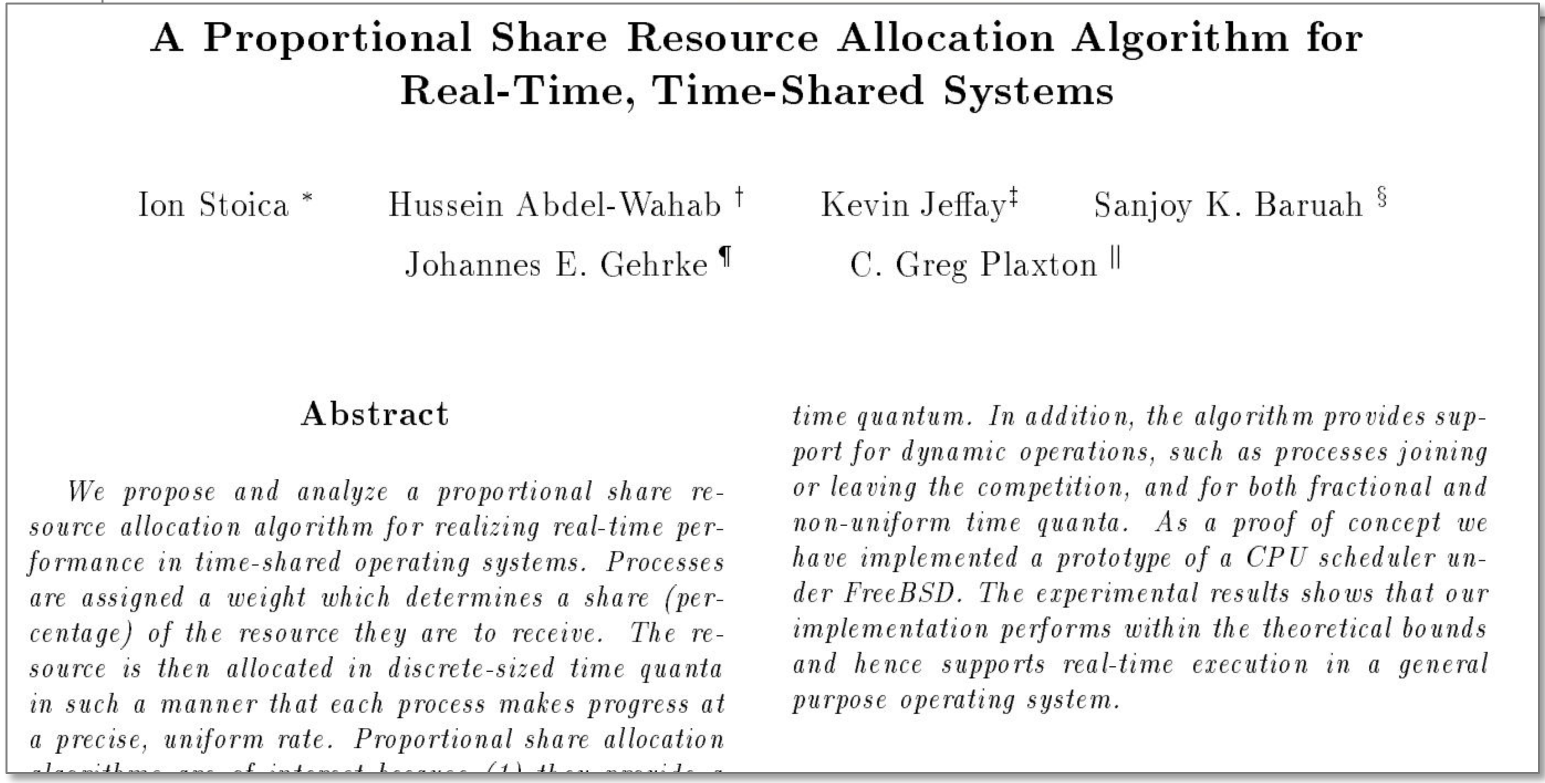
Early Eligible Virtual Deadline First (EEVDF)



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.



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 a precise, uniform rate. Proportional share allocation

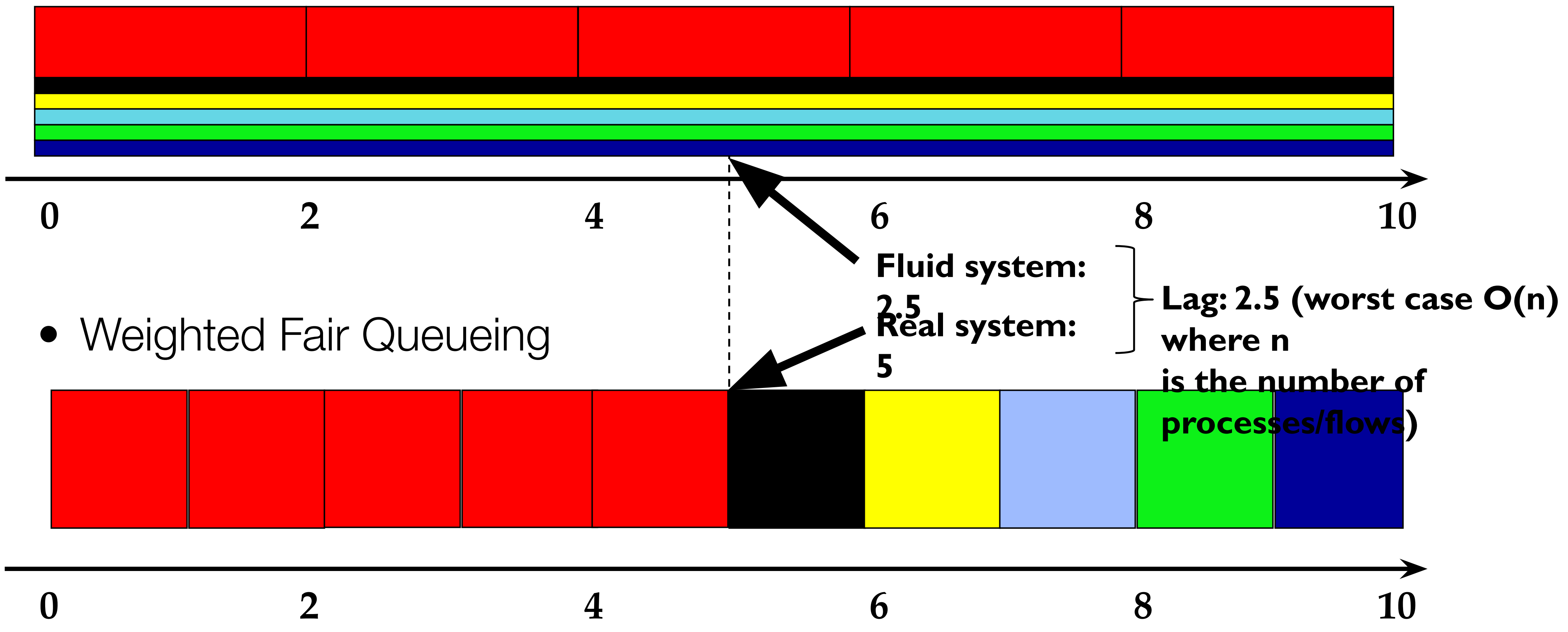
time quantum. In addition, the algorithm provides support for dynamic operations, such as processes joining or leaving the competition, and for both fractional and non-uniform time quanta. As a proof of concept we have implemented a prototype of a CPU scheduler under FreeBSD. The experimental results shows that our implementation performs within the theoretical bounds and hence supports real-time execution in a general purpose operating system.

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

What problem does EEVDF try to solve?

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

- Fluid system service order
- Weighted Fair Queueing

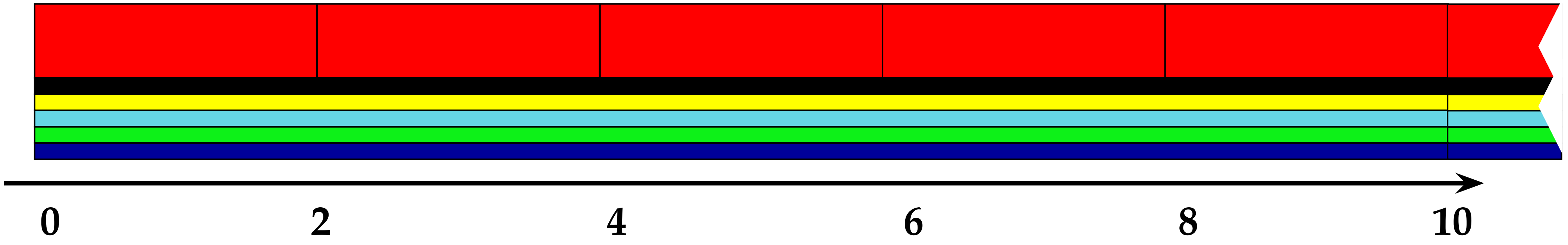


0
2
4
6
8
10
Fluid system: 2.5
Real system: 5
Lag: 2.5 (worst case O(n) where n is the number of processes/flows)
0
2
4
6
8
10

Why is this bad?

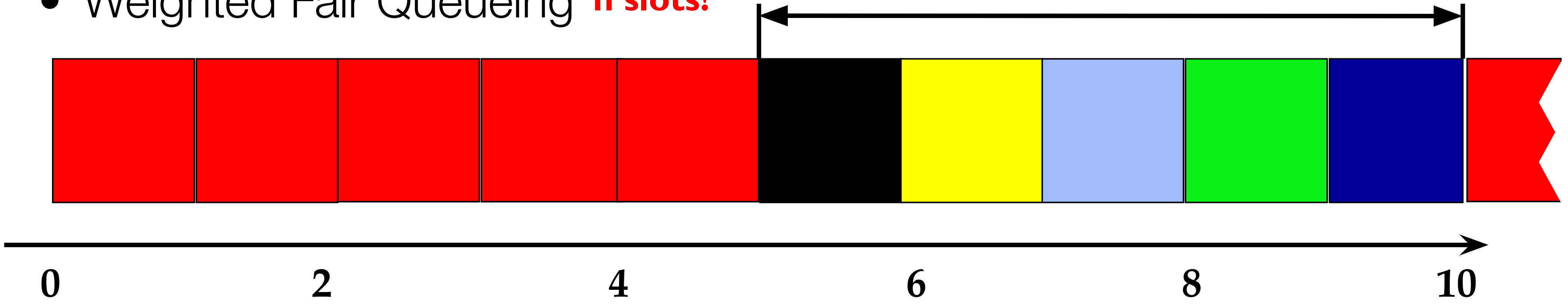
Minimize lag: the difference between service received in real system vs fluid flow (idealized) system

- Fluid system service order



- Weighted Fair Queueing

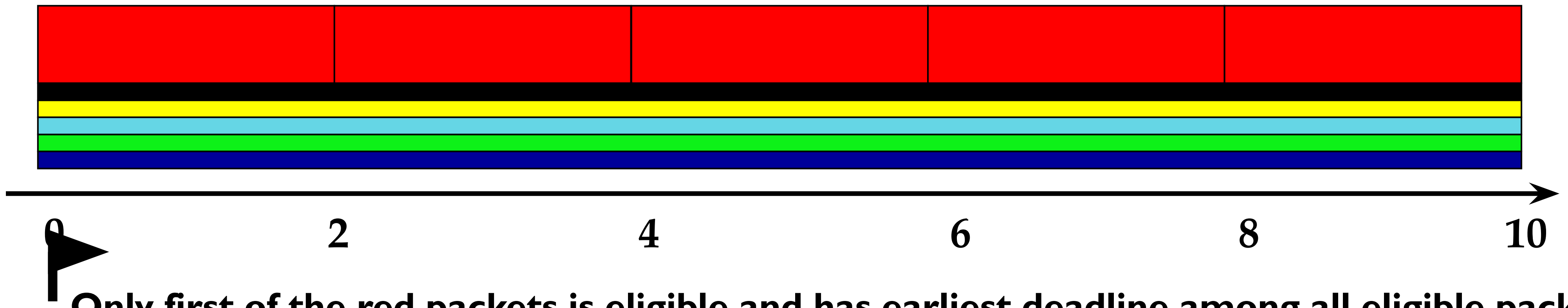
Red flow has 1/2 of resource capacity but can be denied service for n slots!



How?

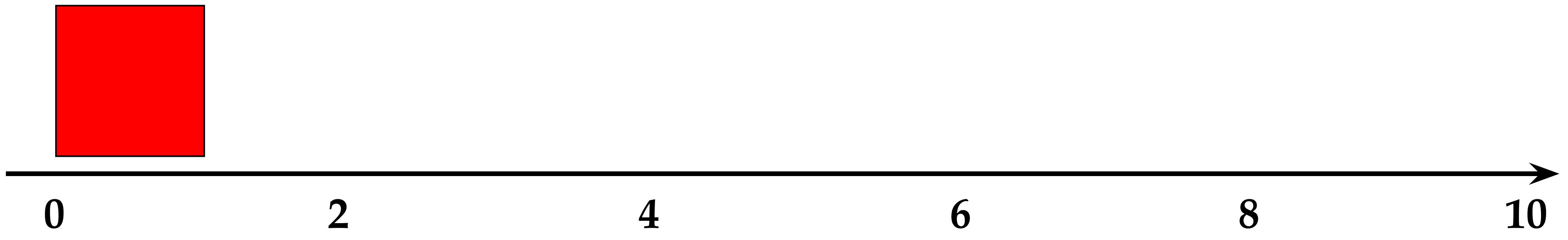
Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time greater than virtual time

- Fluid system service order



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

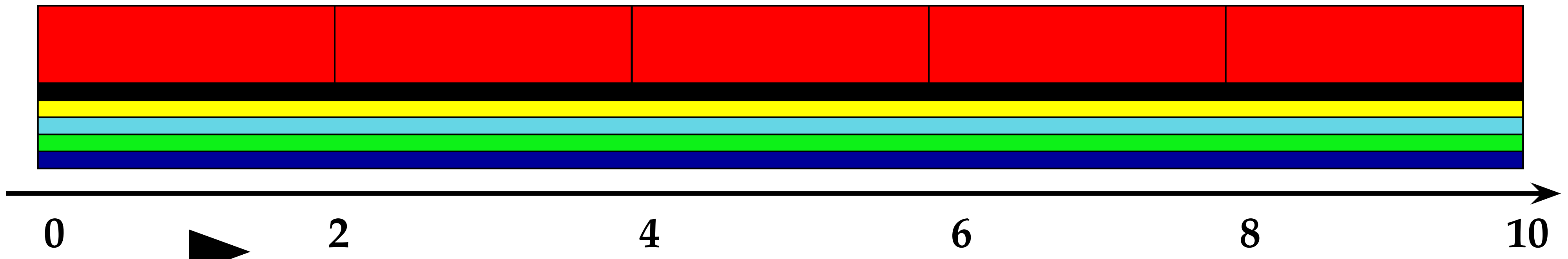
- Weighted Fair Queueing

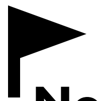


How?

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

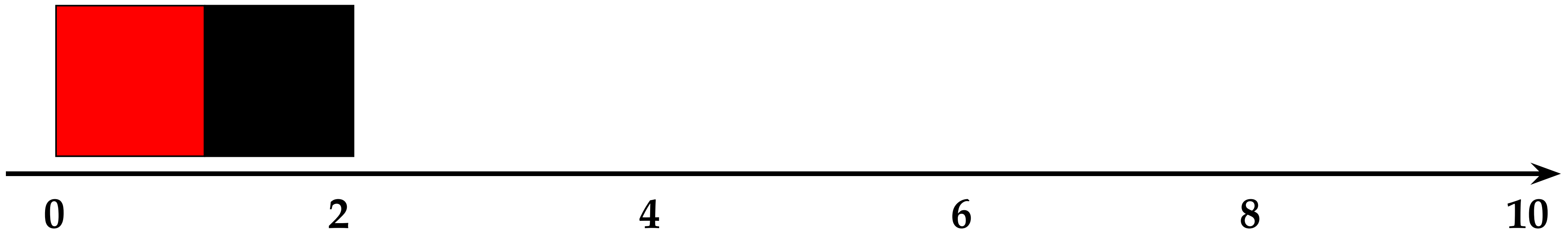
- Fluid system service order





No red packet is eligible (so pick another packet, e.g., black packet)

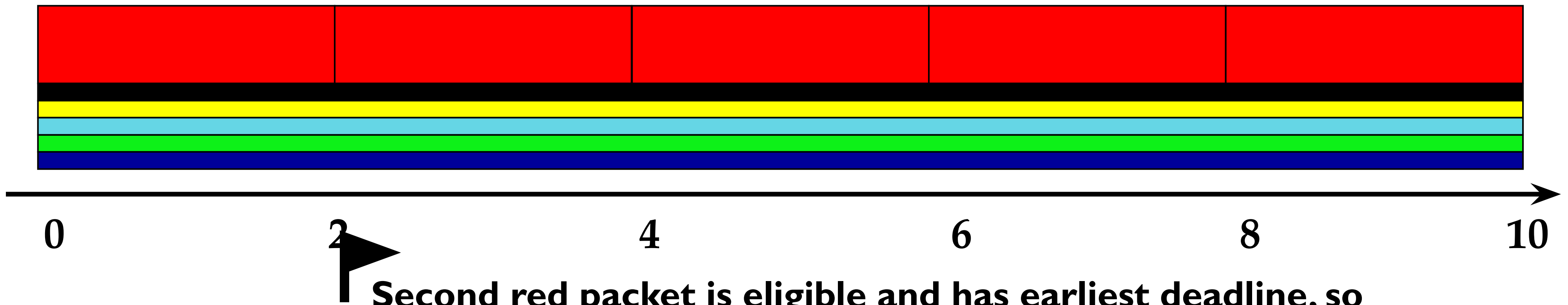
- Weighted Fair Queueing



How?

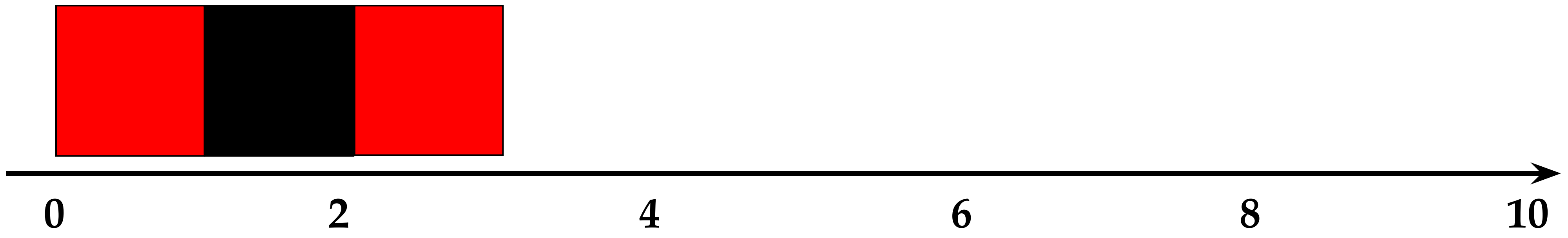
Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time greater than virtual time

- Fluid system service order



Second red packet is eligible and has earliest deadline, so schedule it

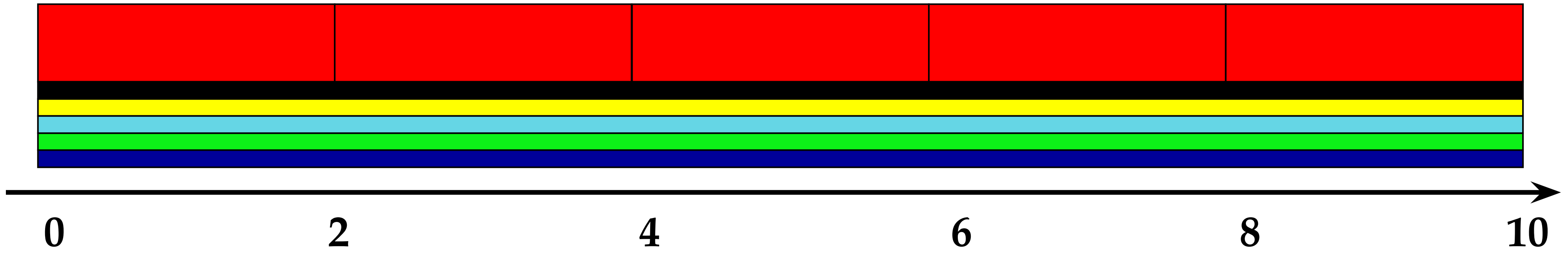
- Weighted Fair Queueing



How?

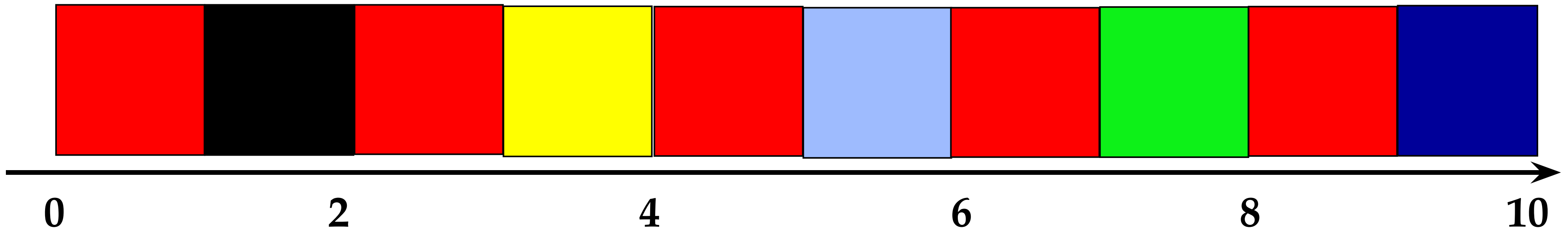
Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time greater than virtual time

- Fluid system service order



Lag <= 0.5 (independent on number of flows)

- Weighted Fair Queueing



Lottery Scheduling

An approximation of weighted fair sharing

– Weight number of tickets

– Scheduling decision probabilistic: give a slot to a process proportionally to its weight

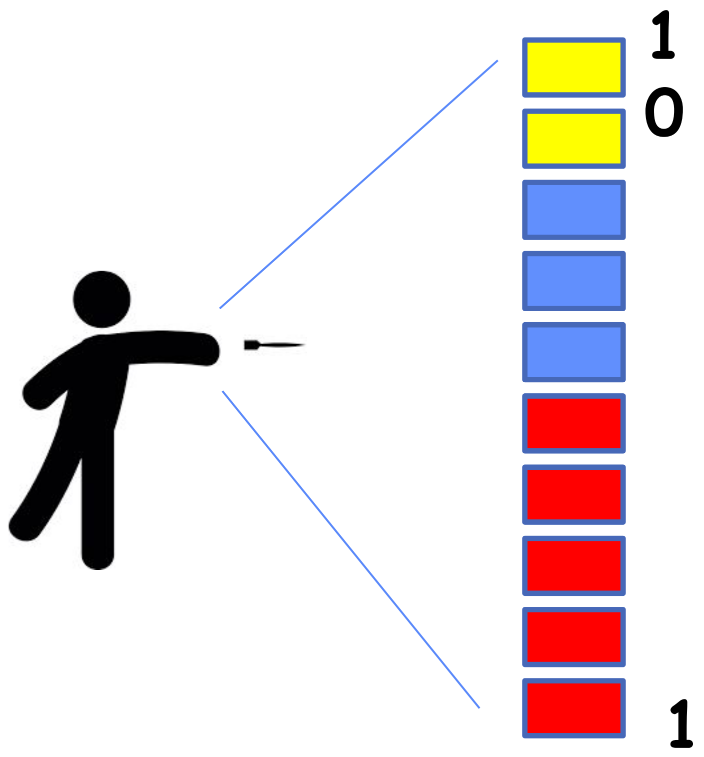
Lottery Scheduling Example (Cont.)

- Lottery Scheduling Example
  – Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  – What if too many short jobs to give reasonable response time?
    » If load average is 100, hard to make progress
    » One approach: log some user out

Lottery Scheduling: Simple Mechanism



- $N_{ticket} = \sum N_i$
- Pick a number d in $1 \; .. \; N_{ticket}$ as the random "dart"
- Jobs record their N_i of allocated tickets
- Order them by N_i
- Select the first j such that $\sum N_i$ up to j exceeds *d*.

Unfairness

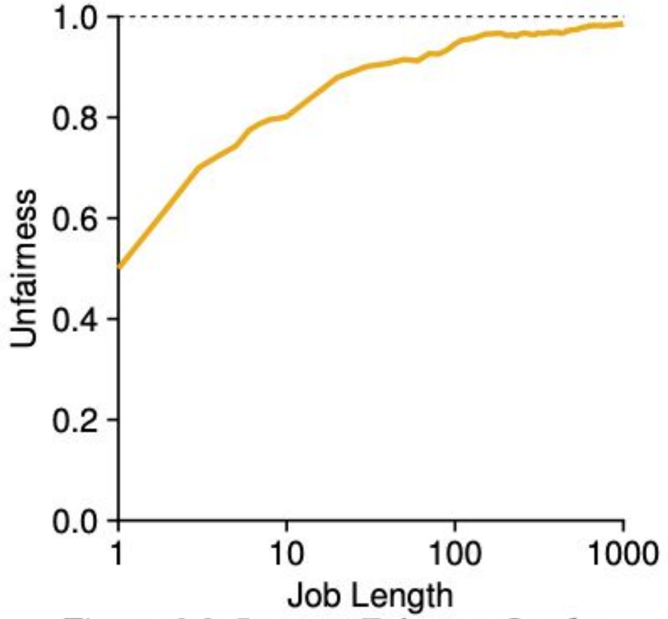


Figure 9.2: **Lottery Fairness Study**

- E.g., Given two jobs A and B of same run time (# Qs) that are each supposed to receive 50%,

  U = finish time of first / finish time of last

- As a function of run time

Stride Scheduling

- Achieve proportional share scheduling without resorting to randomness and overcome the "law of small numbers" problem.
- "Stride" of each job is $\frac{big\#W}{N_i}$
  - The larger your share of tickets, the smaller your stride
  - Ex: W = 10,000, A=100 tickets, B=50, C=250
  - A stride: 100, B: 200, C: 40
- Each job has a "pass" counter
- Scheduler: pick job with lowest *pass*, runs it, add its *stride* to its *pass*
- Low-stride jobs (lots of tickets) run more often
  - Job with twice the tickets gets to run twice as often
- Some messiness of counter wrap-around, new jobs, …

Multi-Core Scheduling

- 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: 1 set of queues/core with background rebalancing

Recall: *Spinlocks for multiprocessing* (cont'd)

- Spinlock implementation:

```
int value = 0; // Free
Acquire() {
   while (test&set(&value)) {}; // spin while busy
}
Release() {
   value = 0;                    // atomic store
}
```

- Spinlock doesn't put the calling thread to sleep—it just busy waits
  - When might this be preferable?
    - » Waiting for limited number of threads at a barrier in a multiprocessing (multicore) program
    - » Wait time at barrier would be greatly increased if threads must be woken inside kernel
- Every `test&set()` atomic read & write
  - Makes value ping-pong around between core-local caches (using lots of memory!)

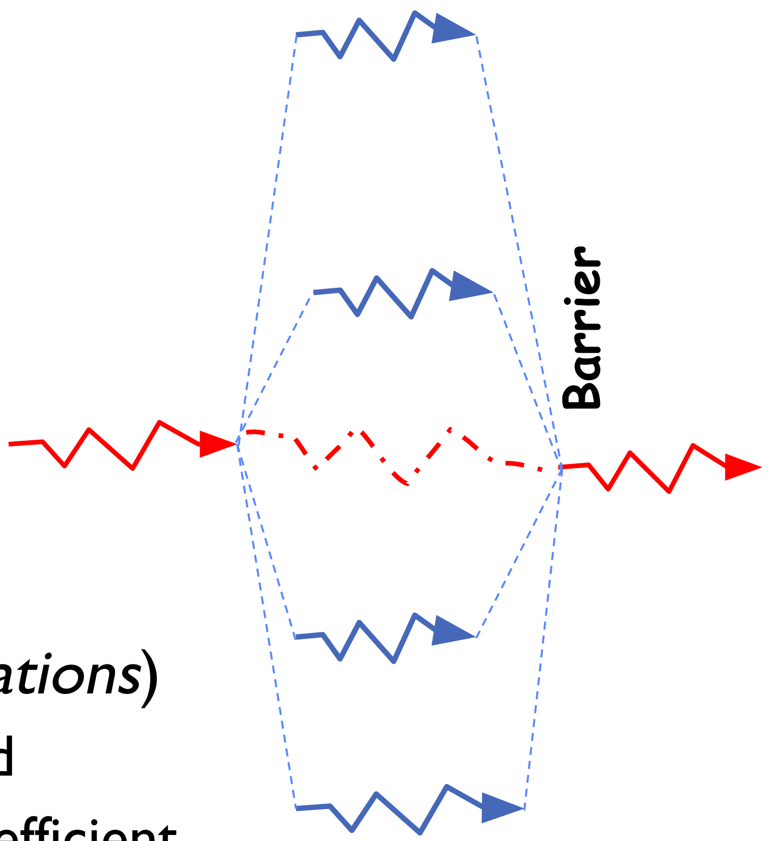
Recall: *Spinlocks for multiprocessing*

- Want `test&test&set()` !
  - First test just reads the lock; no memory contention!
  - Only if free do atomic `test&set()`
- The extra read eliminates the ping-ponging issues:

```
// Implementation of test&test&set():
Acquire() {
   do {
      while(value);       // wait until might be free
   } while (test&set(&value)); // exit if acquire lock
}
```

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



Ensuring Progress

- Starvation: thread fails to make progress for an indefinite period of time

- Starvation ≠ **Deadlock because starvation** because starvation *could* resolve under right circumstances
  - Deadlocks are unresolvable, cyclic requests for resources

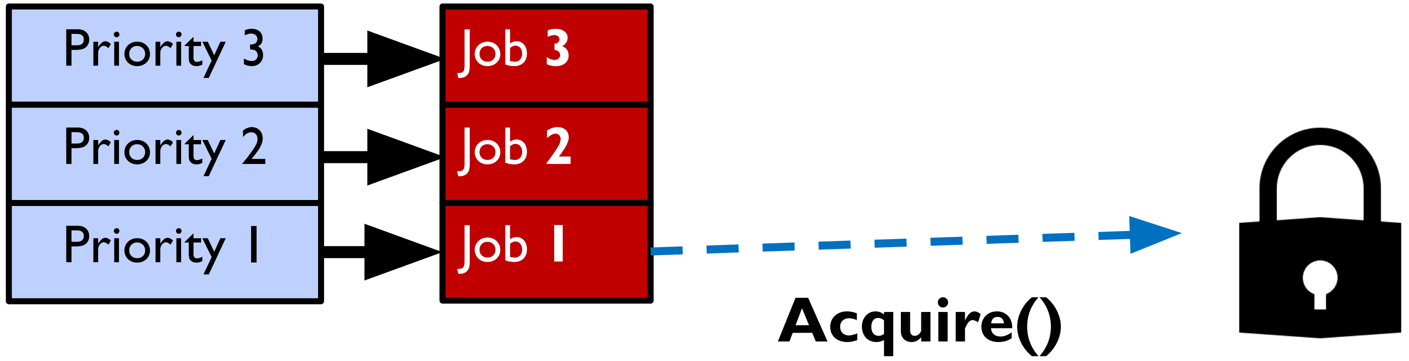
- Causes of starvation:
  - Scheduling policy never runs a particular thread on the CPU
  - Threads wait for each other or are spinning in a way that will never be resolved

- Let's explore what sorts of problems we might encounter and how to avoid them…

Can following policies lead to starvation?

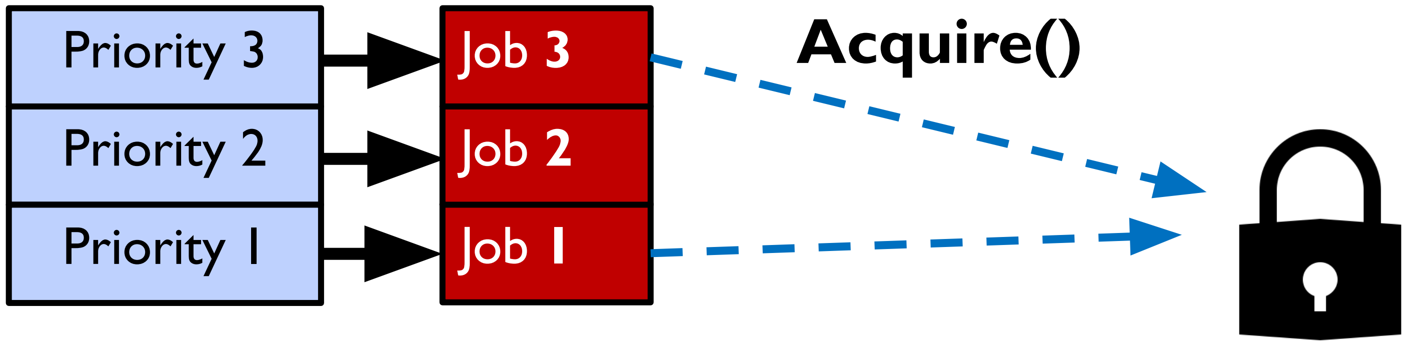
- FCFS ?
- LCFS (Last Come First Served) ?
- Round robin ?
- Shortest remaining time first (SRTF) ?
- Priority scheduling ?
- Early deadline first (EDF) ?
- Fair queuening ?
- Early eligible deadline first (EEVDF) ?

Priority Inversion



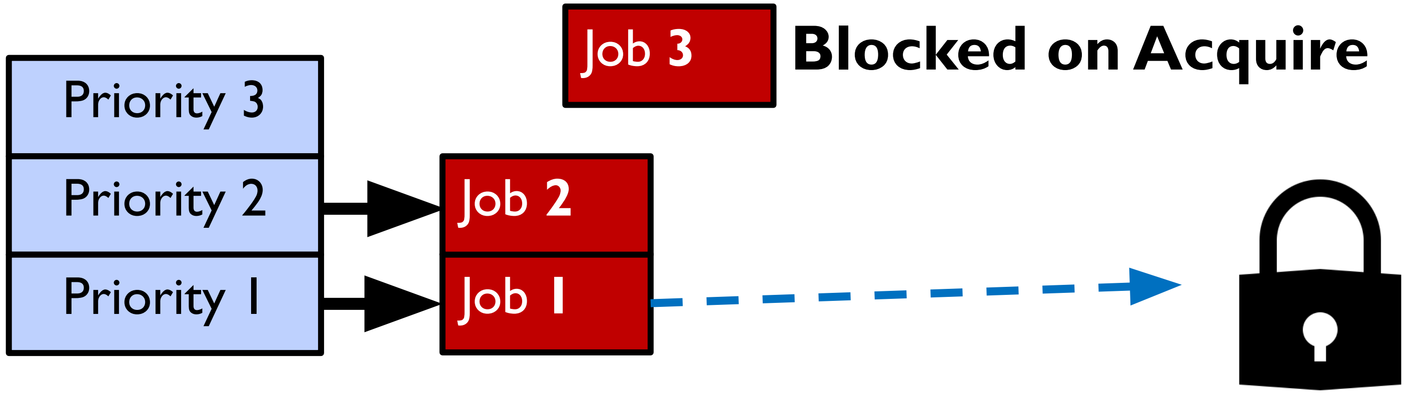
- **At this point, which job does the scheduler choose?**
- Job 3 (Highest priority)

Priority Inversion



- Job 3 attempts to acquire lock held by Job 1

Priority Inversion

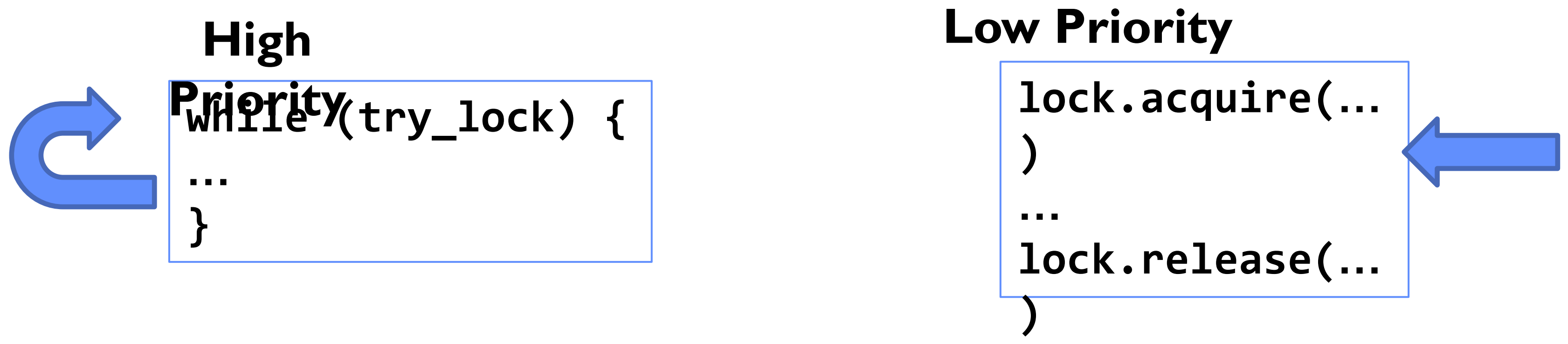


- **At this point, which job does the scheduler choose?**
- Job 2 (Medium Priority)
- Priority Inversion

Priority Inversion

- Where high priority task is blocked waiting on low priority task
- Low priority one ***must*** run for high priority to make progress
- Medium priority task can starve a high priority one

- When else might priority lead to starvation or "live lock"?



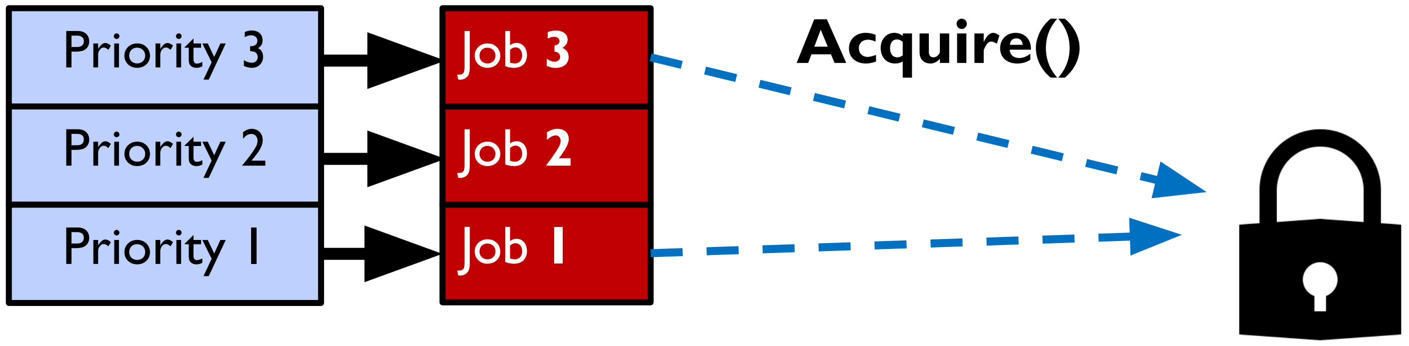
High Priority

```
while (try_lock) {
…
}
```

Low Priority

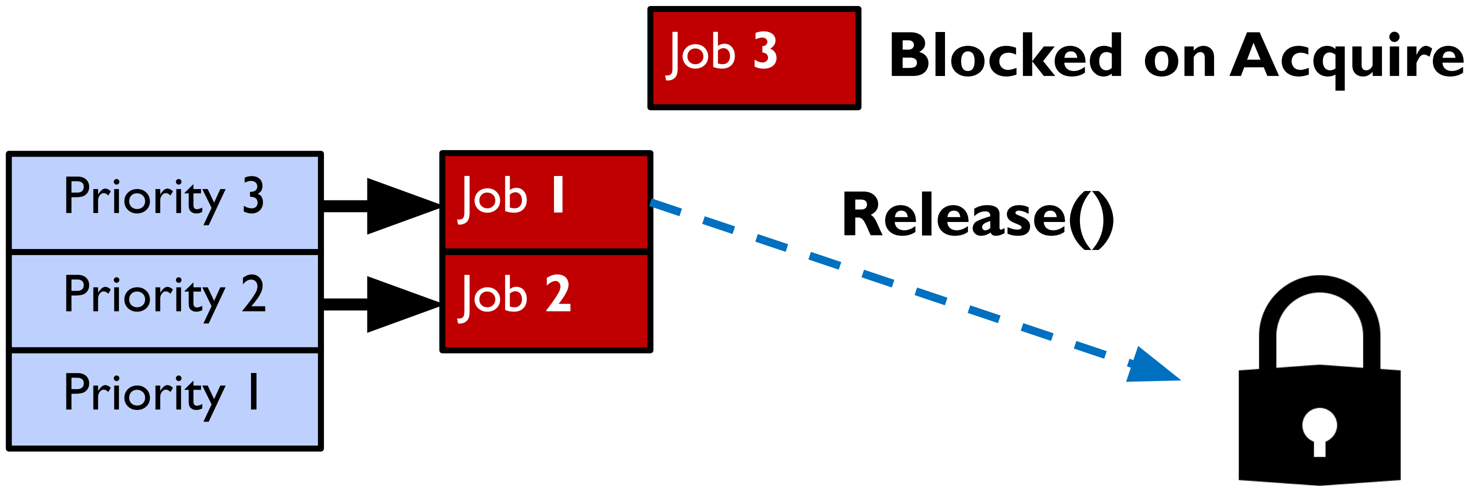
```
lock.acquire(…
)
…
lock.release(…
)
```

One Solution: Priority Donation/Inheritance



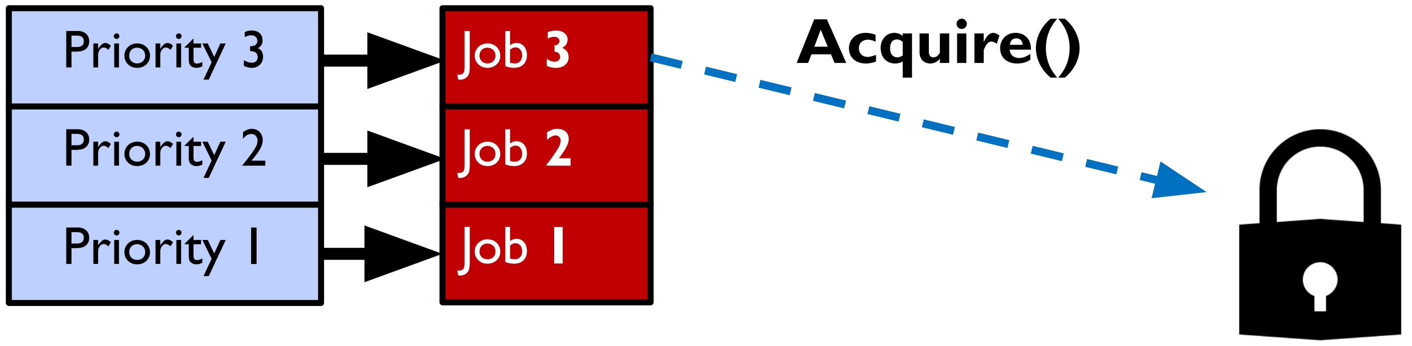
- Job 3 temporarily grants Job 1 its “high priority” to run on its behalf

One Solution: Priority Donation/Inheritance



- Job 3 temporarily grants Job 1 its "high priority" to run on its behalf

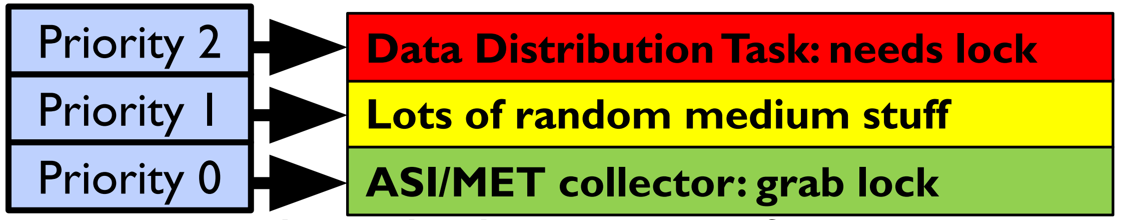
One Solution: Priority Donation/Inheritance



- Job 1 completes critical section and releases lock
- Job 3 acquires lock, runs again
- How does the scheduler know?

Case Study: Martian Pathfinder Rover

- July 4, 1997 – Pathfinder lands on Mars
  - First US Mars landing since Vikings in 1976; first rover
  - Novel delivery mechanism: inside air-filled balloons bounced to stop on the surface from orbit!
- And then…a few days into mission…:
  - Multiple system resets occur to realtime OS (VxWorks)
  - System would reboot randomly, losing valuable time and progress
- Problem? Priority Inversion!
  - Low priority task grabs mutex trying to communicate with high priority task:



Priority 2
Data Distribution Task: needs lock
Priority 1
Lots of random medium stuff
Priority 0
ASI/MET collector: grab lock

  - Realtime watchdog detected lack of forward progress and invoked reset to safe state
    - » High-priority data distribution task was supposed to complete with regular deadline
- Solution: Turn priority donation back on and upload fixes!
- Original developers turned off priority donation (also called priority inheritance)
  - Worried about performance costs of donating priority!

Cause for Starvation: Priorities?

- The policies we've studied so far:
  - **Always prefer to give the CPU to a prioritized job**
  - Non-prioritized jobs may never get to run

- But priorities were a means, not an end
- Our end goal was to serve a mix of CPU-bound, I/O bound, and Interactive jobs effectively on common hardware
  - Give the I/O bound ones enough CPU to issue their next file operation and wait (on those slow discs)
  - Give the interactive ones enough CPU to respond to an input and wait (on those slow humans)
  - Let the CPU bound ones grind away without too much disturbance

Conclusion

- Realtime Schedulers such as EDF
  - Guaranteed behavior by meeting deadlines
  - Realtime tasks defined by tuple of compute time and period
  - Schedulability test (admission control): is it possible to meet deadlines with proposed set of processes?
- Fair Sharing Scheduling
  - Give each job a share of the CPU according to its priority
  - Low-priority jobs get to run less often
  - But all jobs can at least make progress (no starvation)
- Priority Inversion
  - A higher-priority task is prevented from running by a lower-priority task
  - Often caused by locks and through the intervention of a middle-priority task