CS162
Operating Systems and Systems Programming
Lecture 11

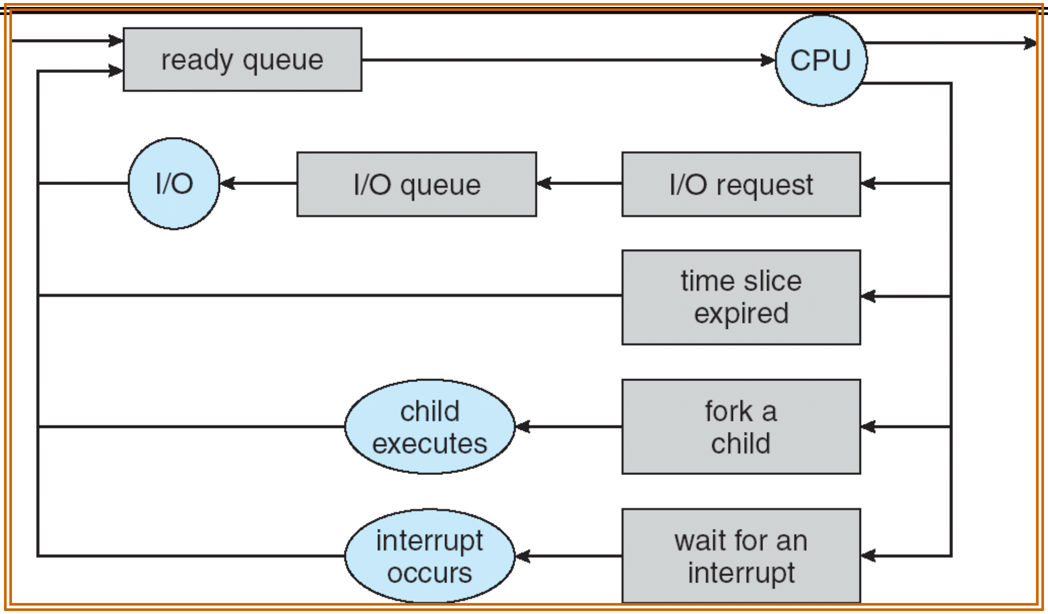
Scheduling 1:
Concepts and Classic Policies

October 8th, 2024

Prof. Ion Stoica

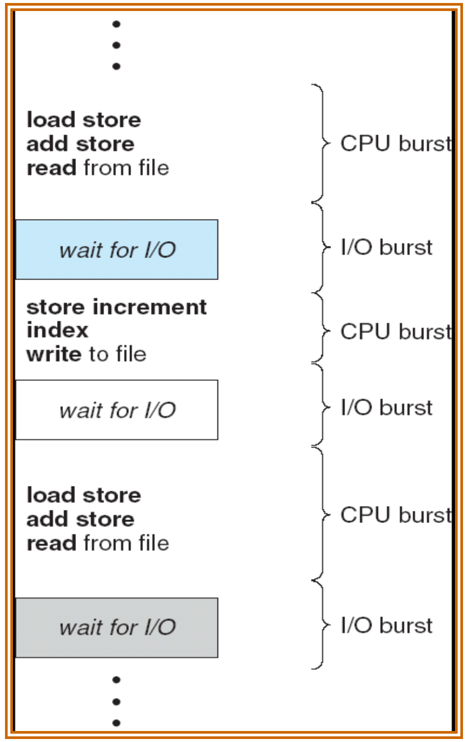
http://cs162.eecs.Berkeley.edu

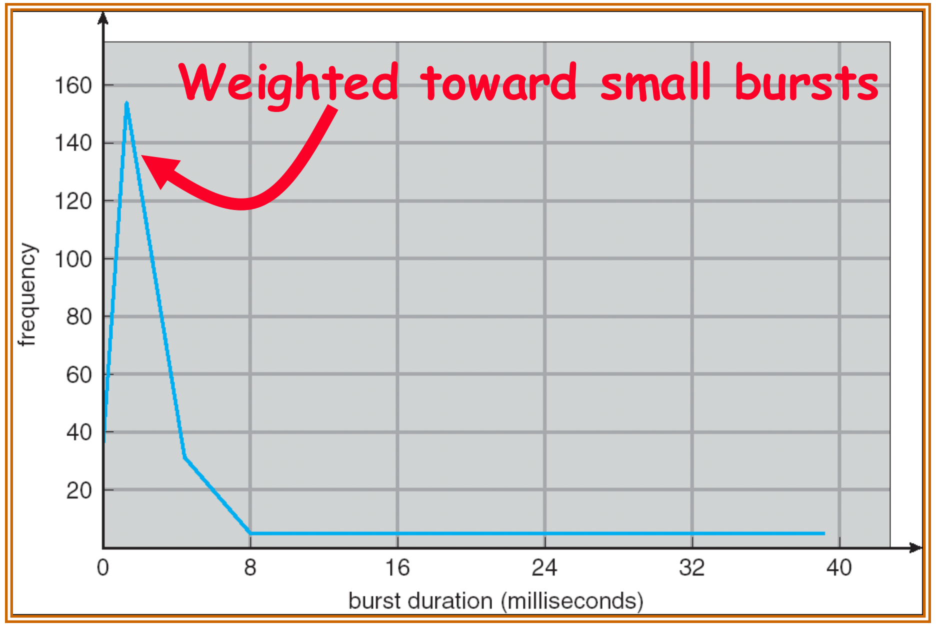
Recall: Scheduling



- Question: How is the OS to decide which of several tasks to take off a queue?
- Scheduling: deciding which threads are given access to resources from moment to moment
  - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

Recall: Assumption: CPU Bursts





- Execution model: programs alternate between bursts of CPU and I/O
  – Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  – Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  – With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Recall: Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

Recall: First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:

  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process stuck behind long process

Recall: FCFS Scheduling (Cont.)

- Example continued:
  – Suppose that processes arrive in order: P2 , P3 , P1
    Now, the Gantt chart for the schedule is:

  – Waiting time for P1 = 6; P2 = 0; P3 = 3
  – Average waiting time: (6 + 0 + 3)/3 = 3
  – Average Completion time: (3 + 6 + 30)/3 = 13
- In second case:
  – Average waiting time is much better (before it was 17)
  – Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  – Simple (+)
  – Short jobs get stuck behind long ones (-)
    » Safeway: Getting milk, always stuck behind cart full of items!
      Upside: get to read about Space Aliens!

Round Robin (RR) Scheduling

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
- Round Robin Scheme: Preemption!
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - n processes in ready queue and time quantum is $q \Rightarrow$
    - » Each process gets $1/n$ of the CPU time
    - » In chunks of at most q time units
    - » No process waits more than $(n-1)q$ time units

RR Scheduling (Cont.)

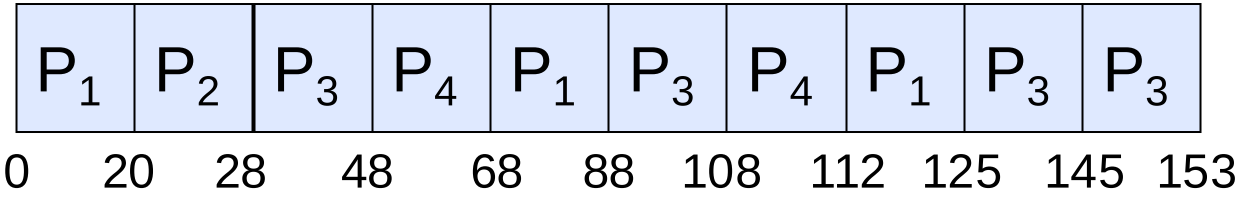
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ Interleaved (really small $\Rightarrow$ hyperthreading?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  – The Gantt chart is:



  – Waiting time for $P_1=(68-20)+(112-88)=72$
  $P_2=(20-0)=20$
  $P_3=(28-0)+(88-48)+(125-108)=85$
  $P_4=(48-0)+(108-68)=88$

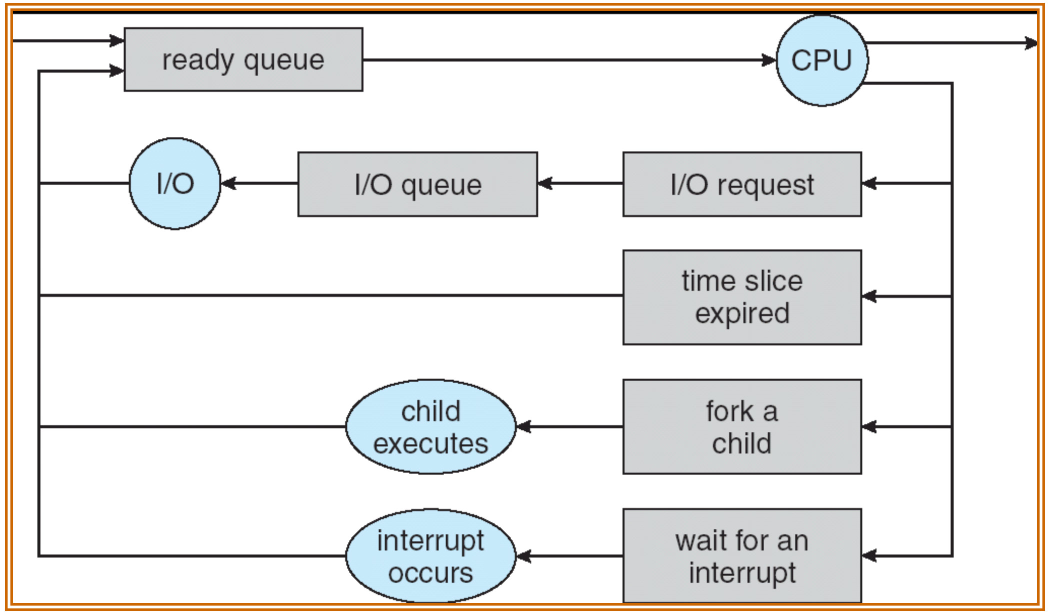
  – Average waiting time = (72+20+85+88)/4=66¼

  – Average completion time = (125+28+153+112)/4 = 104½

- Thus, Round-Robin Pros and Cons:
  – Better for short jobs, Fair (+)
  – Context-switching time adds up for long jobs (-)

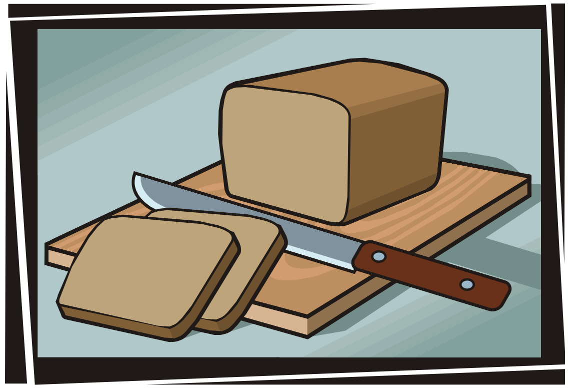
How to Implement RR in the Kernel?

- FIFO Queue, as in FCFS
- But preempt job after quantum expires, and send it to the back of the queue
  – How? Timer interrupt!
  – And, of course, careful synchronization



Project 2: Scheduling

Round-Robin Discussion



- How do you choose time slice?
  - What if too big?
    - » Response time suffers
  - What if infinite (∞)?
    - » Get back FIFO
  - What if time slice too small?
    - » Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - » Worked ok when UNIX was used by one or two people.
    - » What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput:
    - » Typical time slice today is between 10ms – 100ms
    - » Typical context-switching overhead is 0.1ms – 1ms
    - » Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

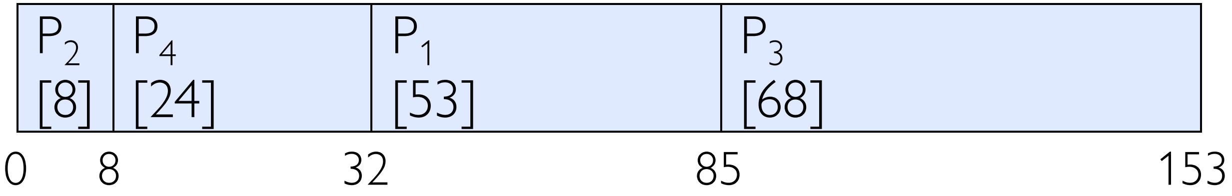
- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  – Both RR and FCFS finish at the same time
  – Average completion time is much worse under RR!
    » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  – Total time for RR longer even for zero-cost switch!

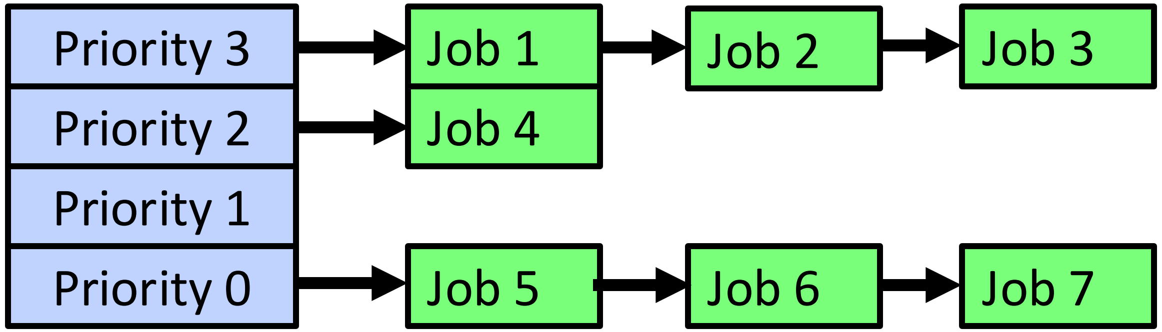
Earlier Example with Different Time Quantum

Best FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Handling Differences in Importance: Strict Priority Scheduling



- Execution Plan
  - Always execute highest-priority runable jobs to completion
  - Each queue can be processed in RR with some time-quantum
- Problems:
  - Starvation:
    - » Lower priority jobs don't get to run because higher priority jobs
  - Deadlock: Priority Inversion
    - » Happens when low priority task has lock needed by high-priority task
    - » Usually involves third, intermediate priority task preventing high-priority task from running
- How to fix problems?
  - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - » long running jobs may never get CPU
    - » Urban legend: In Multics, shut down machine, found 10-year-old job ⇒ Ok, probably not…
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!

Scheduling Fairness

- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - » What if one long-running job and 100 short-running ones?
    - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - » What is done in some variants of UNIX
    - » This is ad hoc—what rate should you increase priorities?
    - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

What if we Knew the Future?

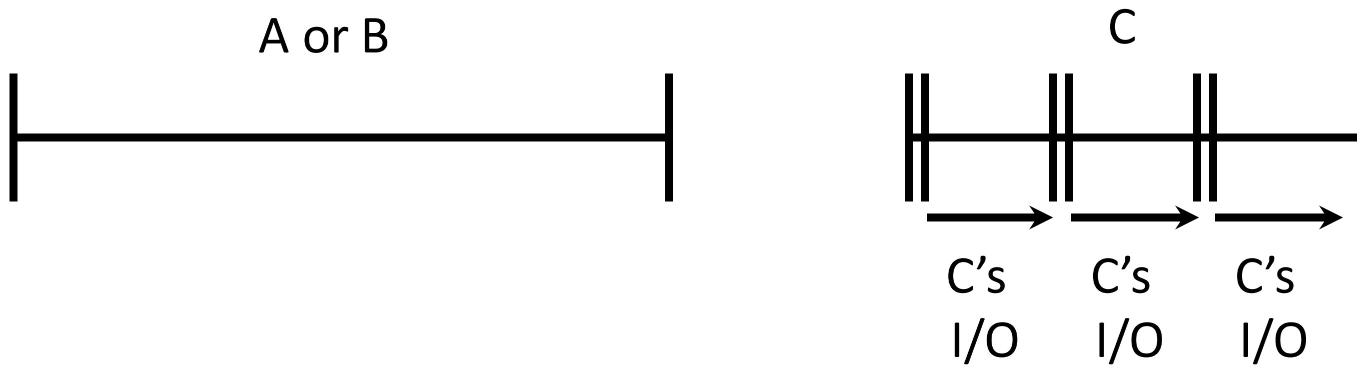
- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied to whole program or current CPU burst
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Discussion

- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF

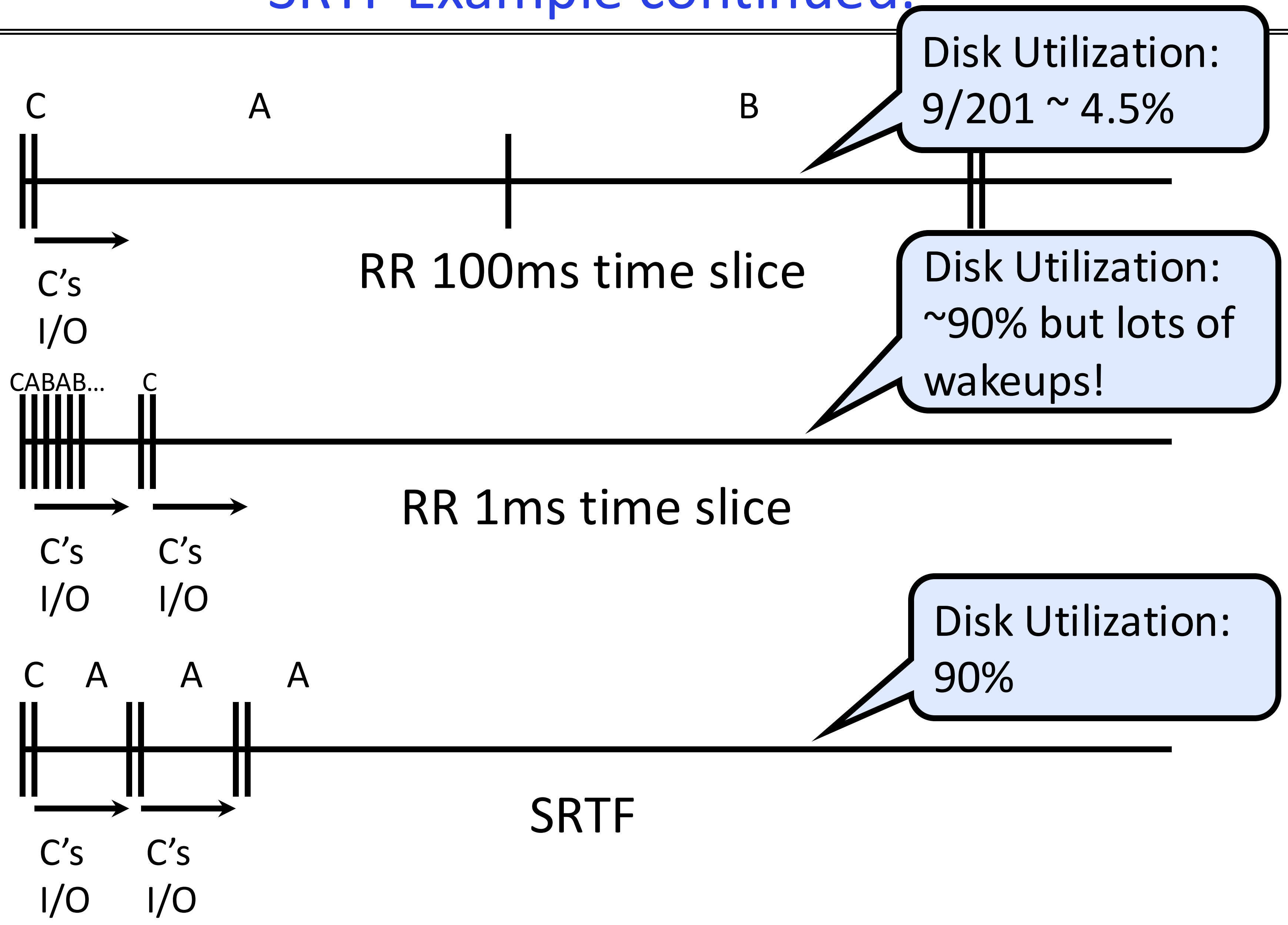
- Comparison of SRTF with FCFS
  - What if all jobs the same length?
    - » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - » SRTF: short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A, B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FCFS:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:

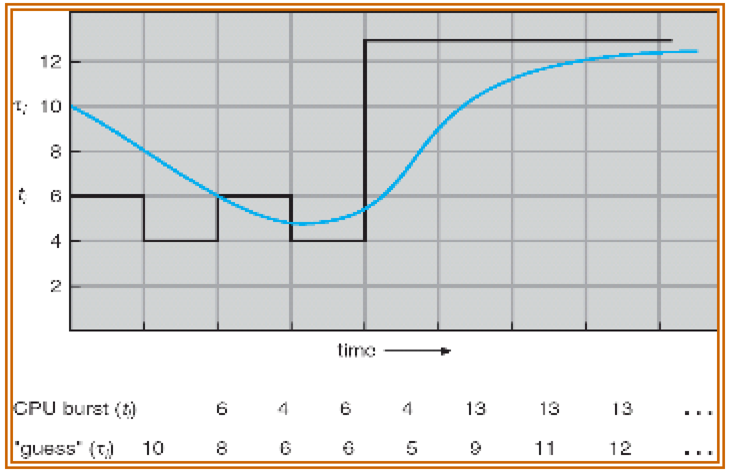


SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - » When you submit a job, have to say how long it will take
    - » To stop cheating, system kills job if takes too long
  - But: hard to predict job's runtime even for non-malicious users
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

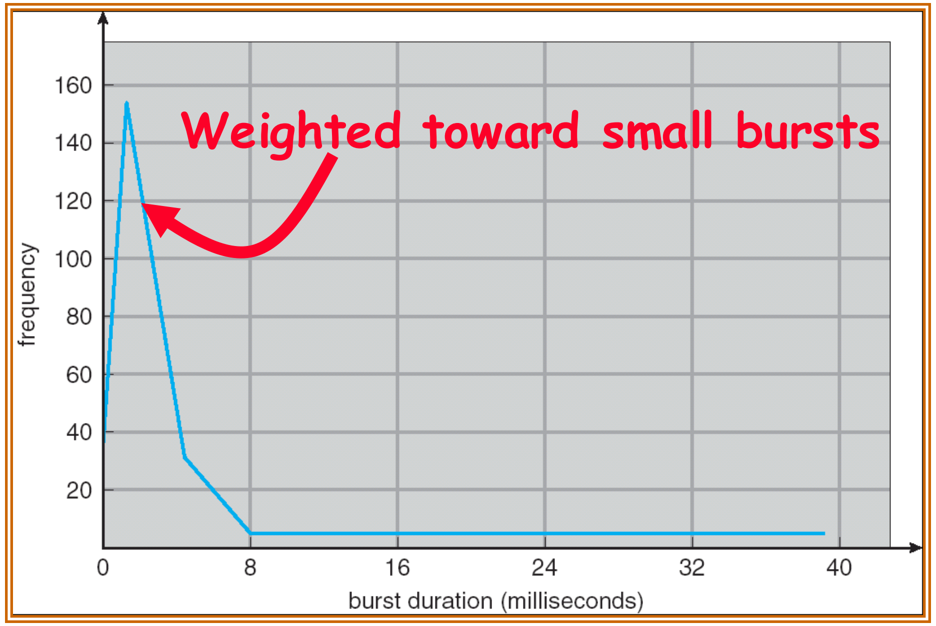
Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - » If program was I/O bound in past, likely in future
    - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths.
    Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    $\tau n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
    with $(0<\alpha\leq1)$

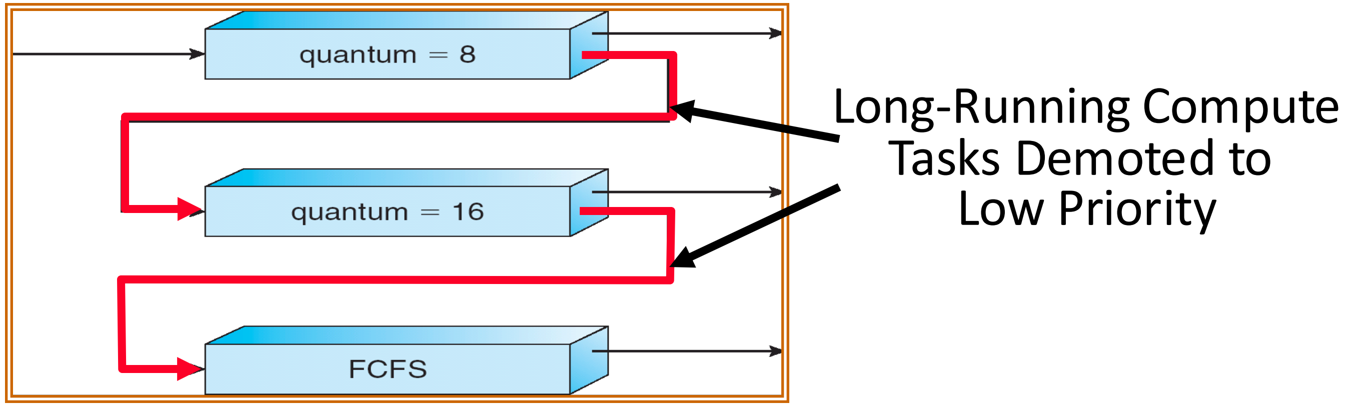


How to Handle Simultaneous Mix of Diff Types of Apps?

- Consider mix of interactive and high throughput apps:
  – How to best schedule them?
  – How to recognize one from the other?
    » Do you trust app to say that it is "interactive"?
  – Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- For instance, is Burst Time (observed) useful to decide which application gets CPU time?
  – Short Bursts ⇒ Interactivity ⇒ High Priority?
- Assumptions encoded into many schedulers:
  – Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
  – Apps that compute a lot should get low(er?) priority, since they won't notice intermittent bursts from interactive apps
- Hard to characterize apps:
  – What about apps that sleep for a long time, but then compute for a long time?
  – Or, what about apps that must run under all circumstances (say periodically)

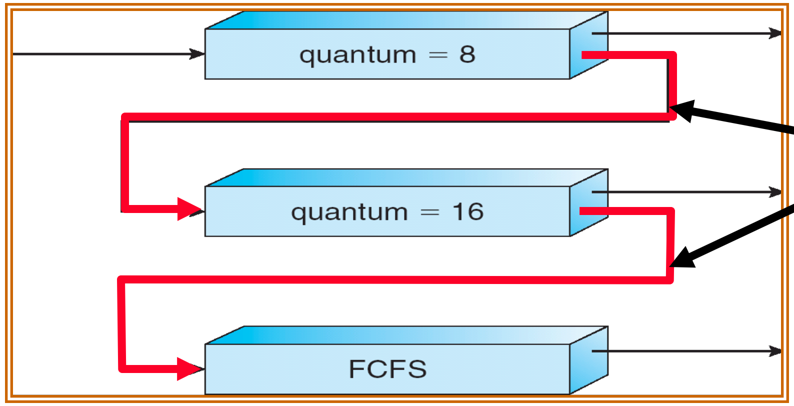


Multi-Level Feedback Scheduling



- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next: 2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

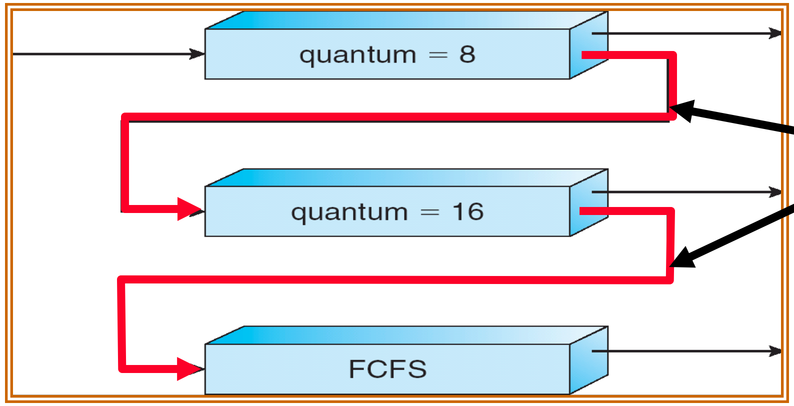
Scheduling Details



Long-Running Compute Tasks Demoted to Low Priority

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - » serve all from highest priority, then next priority, etc.
  - Time slice:
    - » each queue gets a certain amount of CPU time
    - » e.g., 70% to highest, 20% next, 10% lowest

Scheduling Details



Long-Running Compute Tasks Demoted to Low Priority

- Countermeasure: user action that can foil intent of the OS designers
  – For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  – Of course, if everyone did this, wouldn't work!
- Example of Othello program:
  – Playing against competitor, so key was to do computing at higher priority the competitors.
    » Put in printf's, ran much faster!

Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower priority value ⇒ higher priority (for realtime values)
  - Highest priority value ⇒ Lower priority (for nice values)
  - All algorithms O(1)
    - » Timeslices/priorities/interactivity credits all computed when job finishes time slice
    - » 140-bit bit mask indicates presence or absence of job at given priority level
- Two separate priority queues: “active” and “expired”
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
- Timeslice depends on priority – linearly mapped onto timeslice range
  - Like a multi-level queue (one queue per priority) with different timeslice at each level
  - Execution split into “Timeslice Granularity” chunks – round robin through priority

O(1) Scheduler Continued

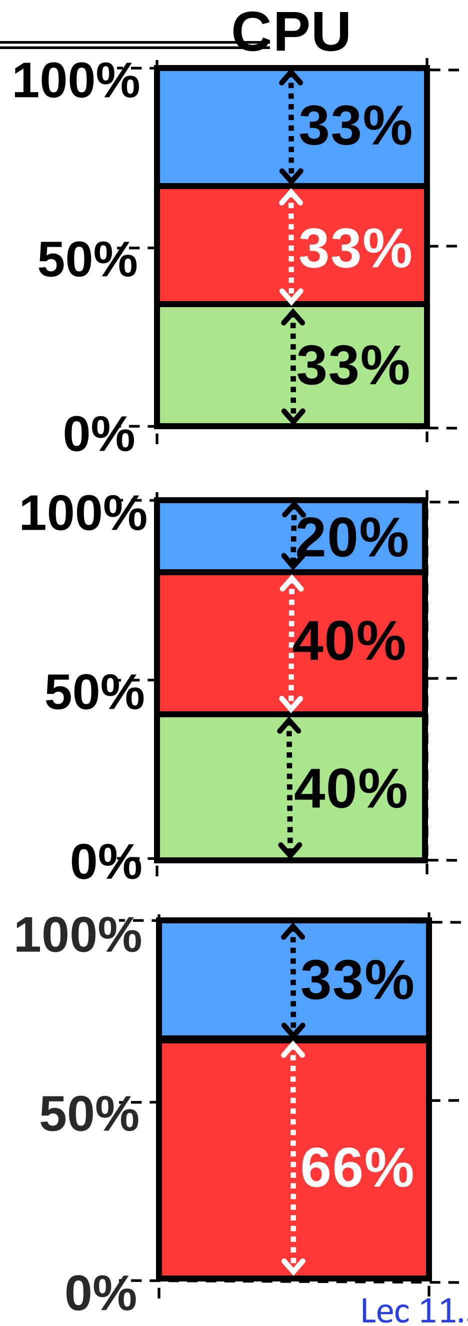
- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    - » p->sleep_avg = sleep_time – run_time
    - » Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    - » Earned when a task sleeps for a "long" time
    - » Spend when a task runs for 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 interactivity
    - » Placed back into active queue, unless some other task has been starved for too long…
- Real-Time Tasks
  - Always preempt non-RT tasks
  - No dynamic adjustment of priorities
  - Scheduling schemes:
    - » SCHED_FIFO: preempts other tasks, no timeslice limit
    - » SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority

Administrivia

- Midterm I:
  - Grading done today by end of the weel. 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!
- How does this work?
  - You get 20 points/partner to distribute as you want:
    Example—4 person group, you get 3 x 20 = 60 points
    - » If all your partners contributed equally, give the 20 points each
    - » Or, you could do something like:
      - 22 points partner 1
      - 22 points partner 2
      - 16 points partner 3
  - DO NOT GIVE YOURSELF POINTS!
    - » You are NOT an unbiased evaluator of your group behavior

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)

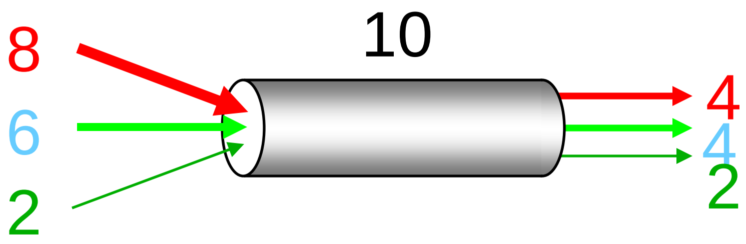
Fair Queueing: Max-min Fairness originated in Networking

- 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

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

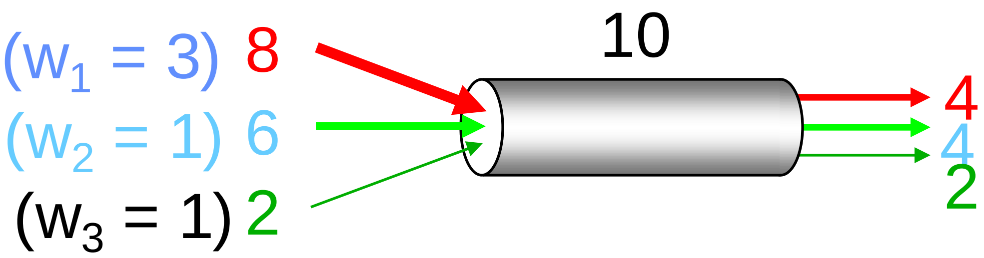


f = 4:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

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

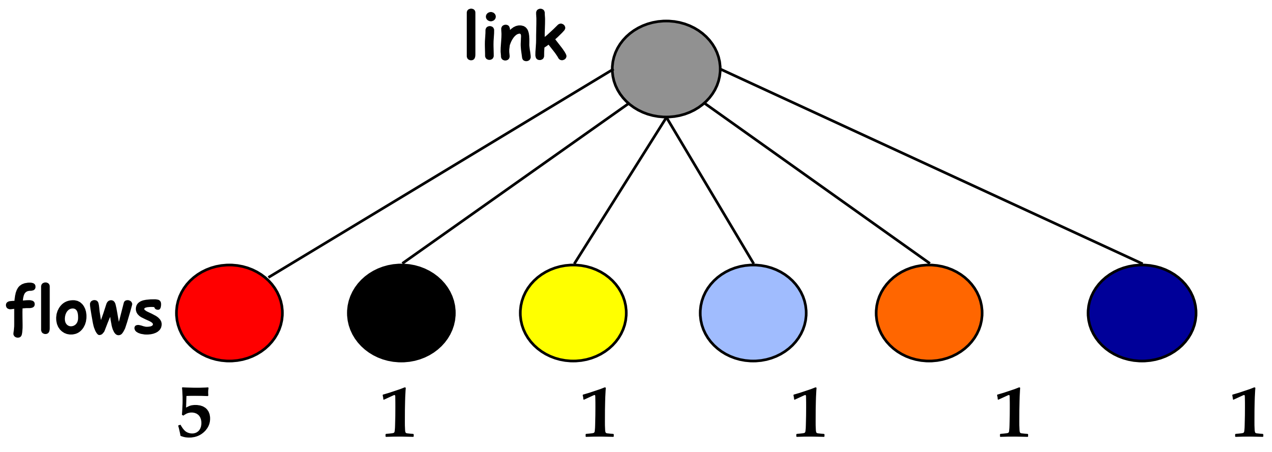
Fluid Flow System

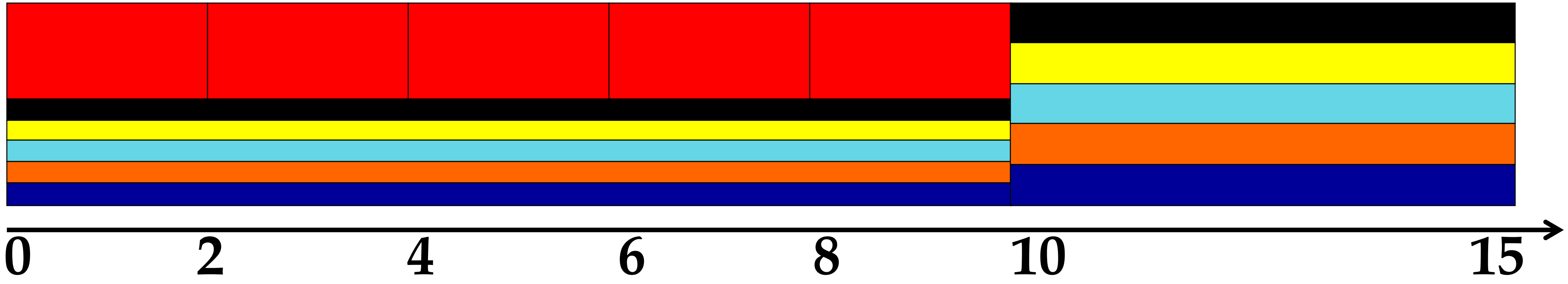
- Flows can be served one bit at a time
  - Fluid flow system, also known as Generalized Processor Sharing (GPS) [Parekh and Gallager '93]

- 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



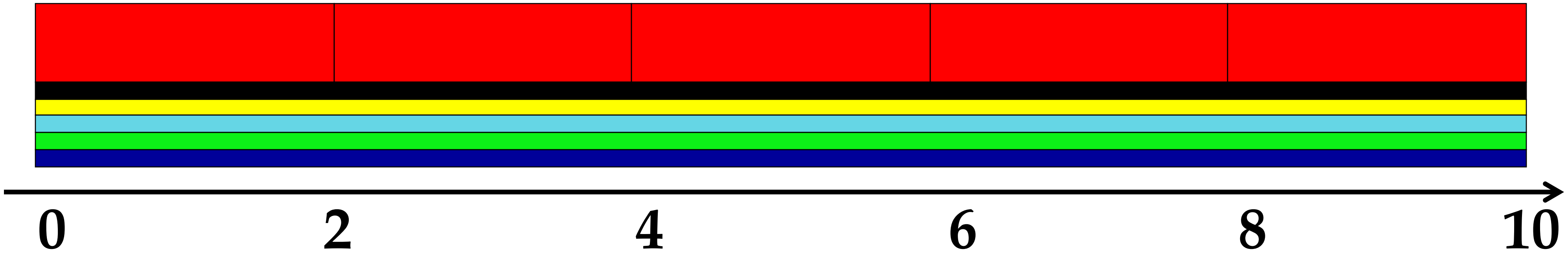


Packet Approximation of GPS

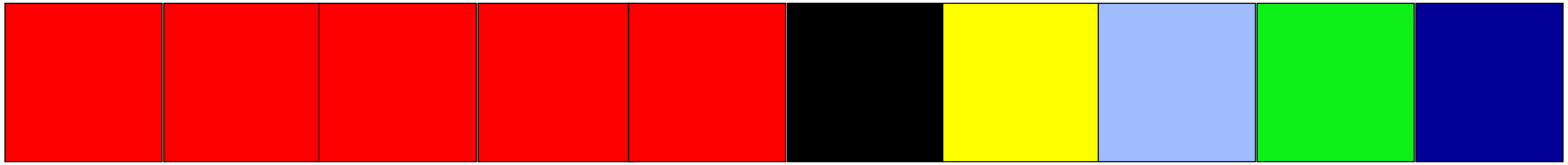
- Emulate GPS
- Select packet that finishes first in GPS assuming that there are no future arrivals

Approximating GPS with WFQ

- Fluid GPS system service order



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

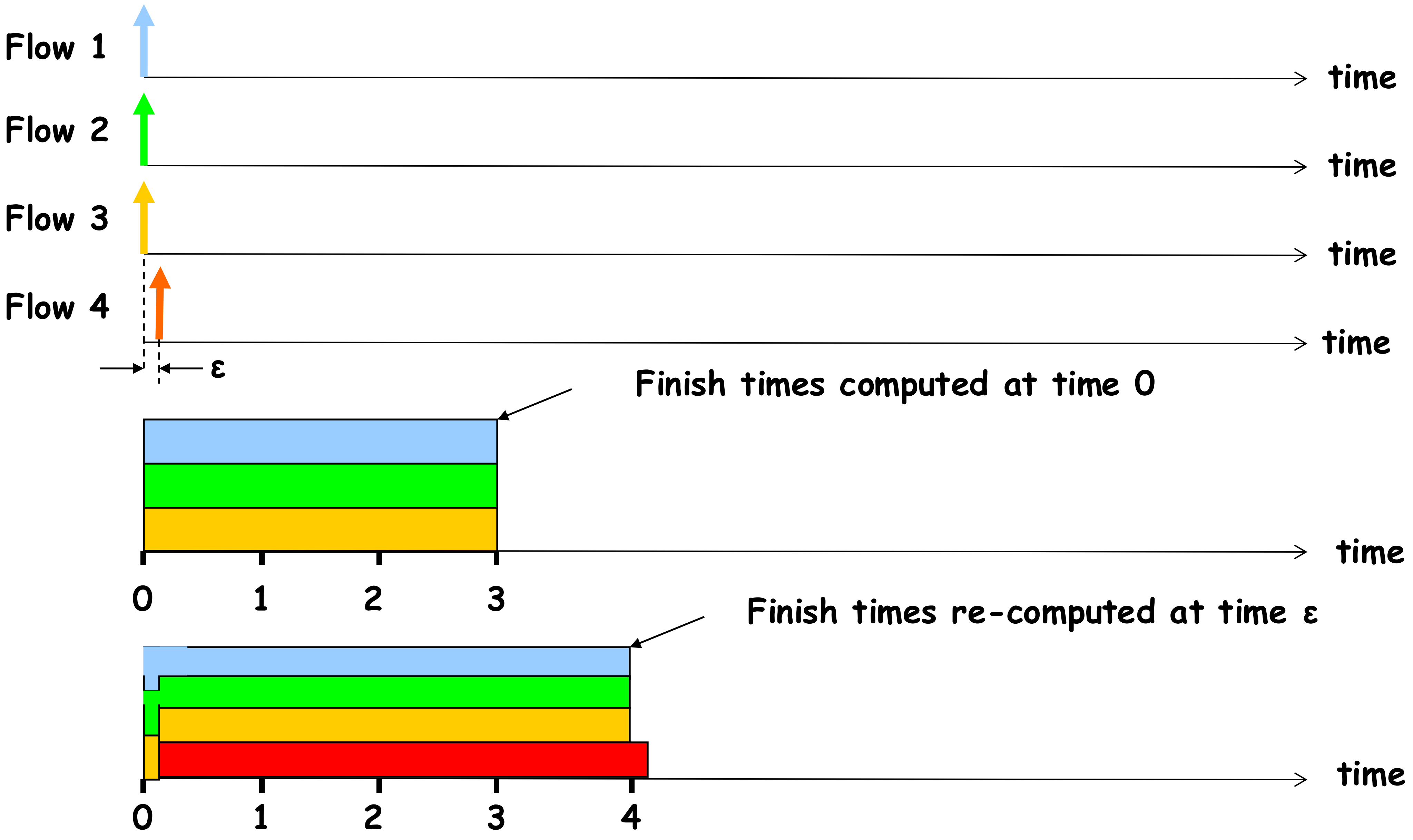


Implementation Challenge

- 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

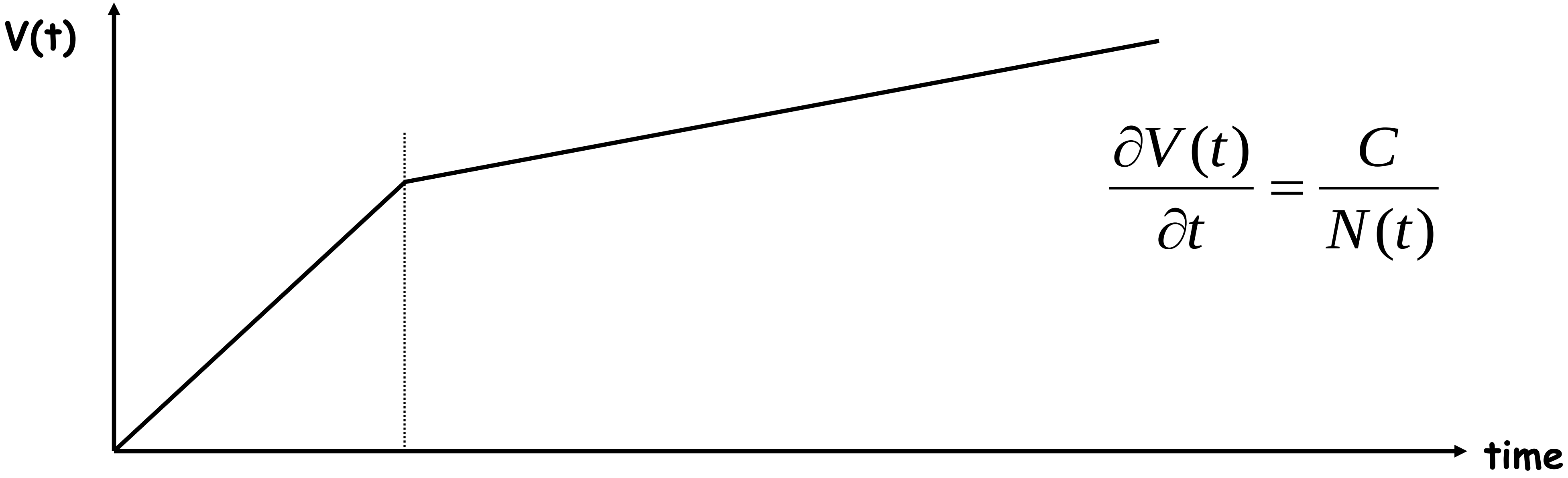


Solution: Virtual Time

- **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)

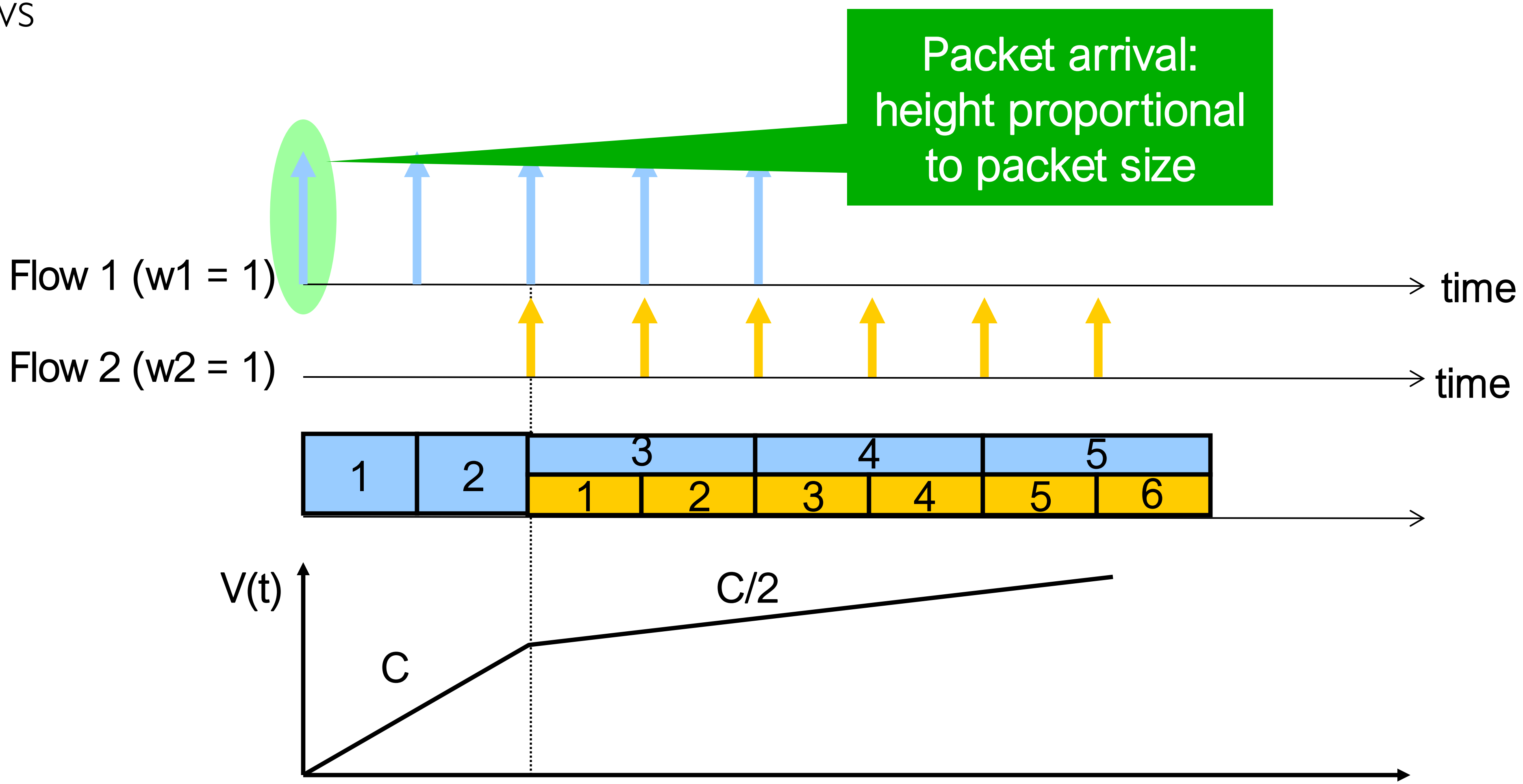
- Measure service, instead of time
- V(t) slope – normalized rate at which every backlogged flow receives service in the fluid flow system
  - *C* – link capacity
  - *N*(t) – total weight of backlogged flows in fluid flow system at time t



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

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

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.

https://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

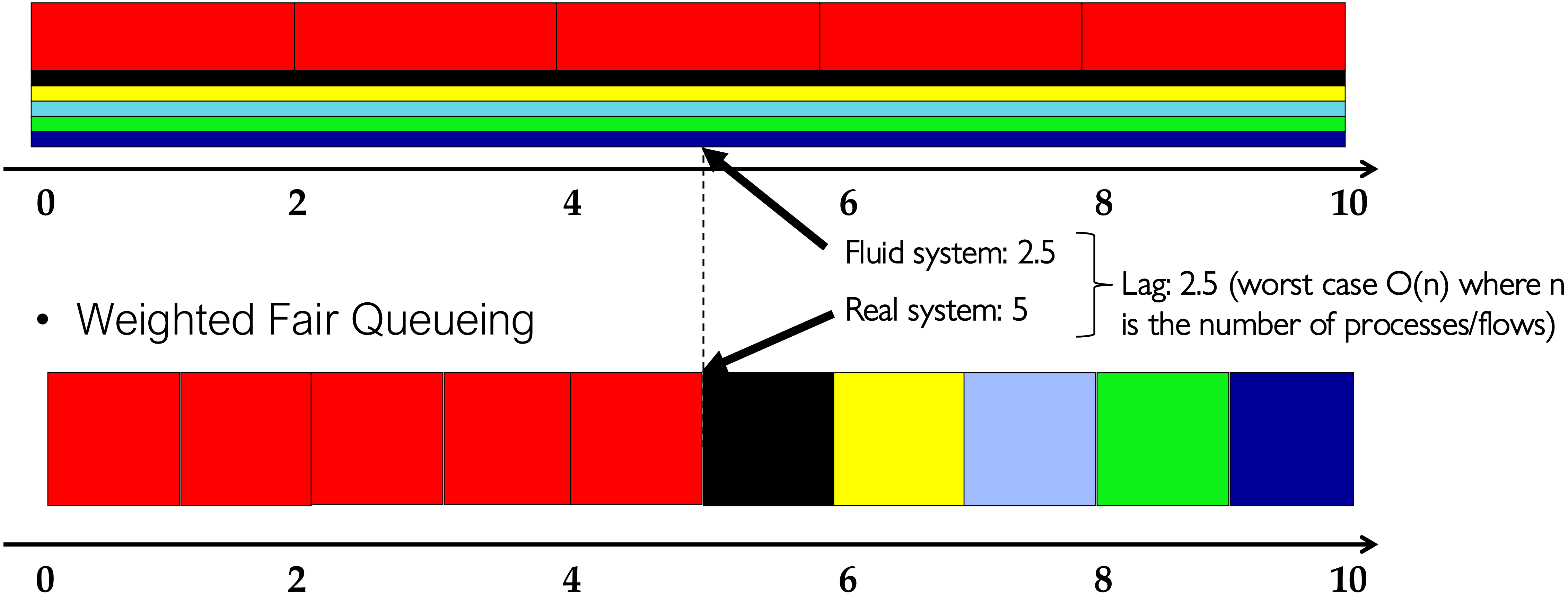
We propose and analyze a proportional share resource allocation algorithm for realizing real-time performance in time-shared operating systems. Processes are assigned a weight which determines a share (percentage) of the resource they are to receive. The resource is then allocated in discrete-sized time quanta in such a manner that each process makes progress at 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.

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



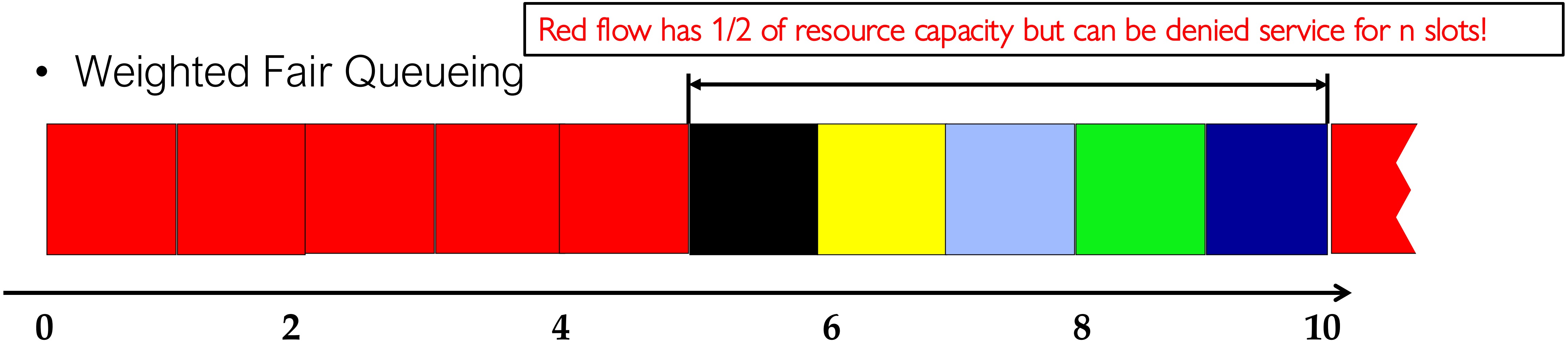
- Weighted Fair Queueing

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



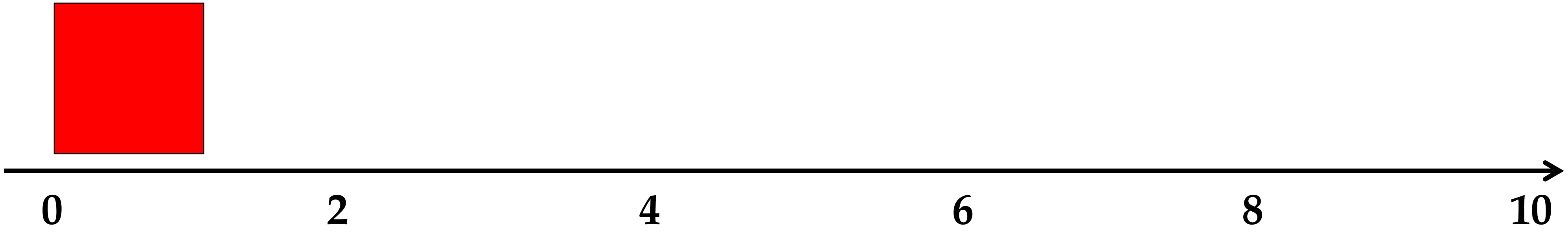
How?

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

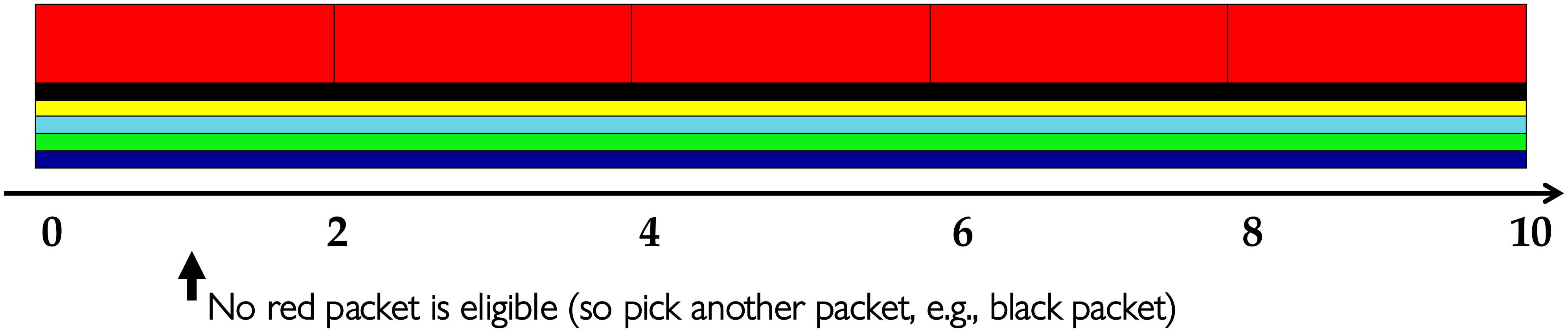
- Weighted Fair Queueing



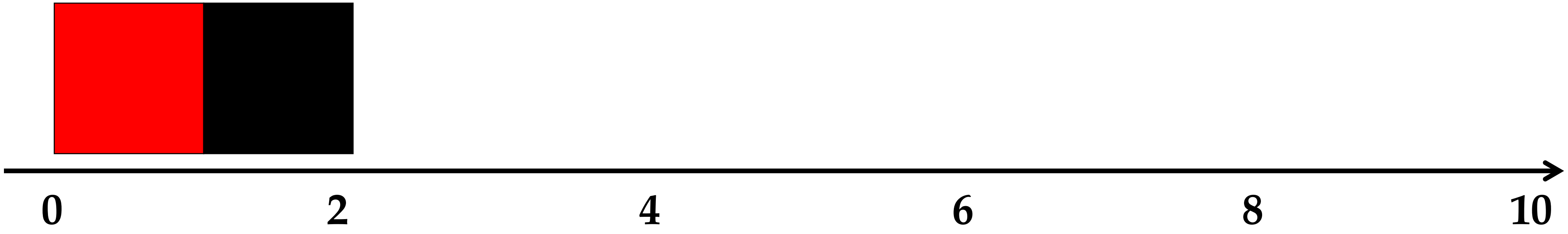
How?

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



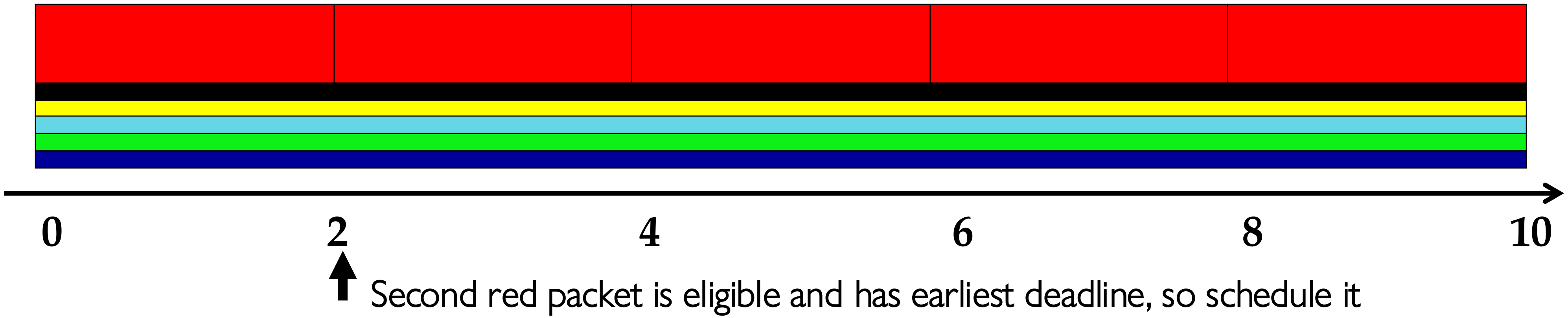
- Weighted Fair Queueing



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



- Weighted Fair Queueing

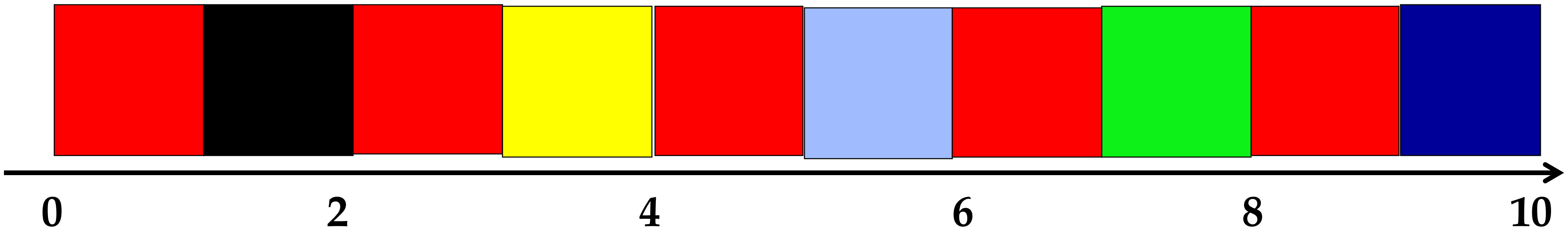
How?

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

Conclusion

- Scheduling Goals:
  - Minimize Response Time (e.g. for human interaction)
  - Maximize Throughput (e.g. for large computations)
  - Fairness (e.g. Proper Sharing of Resources)
  - Predictability (e.g. Hard/Soft Realtime)
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Fair sharing
  - Each process is allocated a share of the CPU
  - Can use weights to emulate other scheduling disciplines (e.g., priority scheduling)