CS162
Operating Systems and Systems Programming
Lecture 13

Fair Scheduling Continued & Deadlock

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

- Proportional fair sharing and Linux EEVDF

- A deeper look at deadlock

Recall: Proportional Fair Sharing

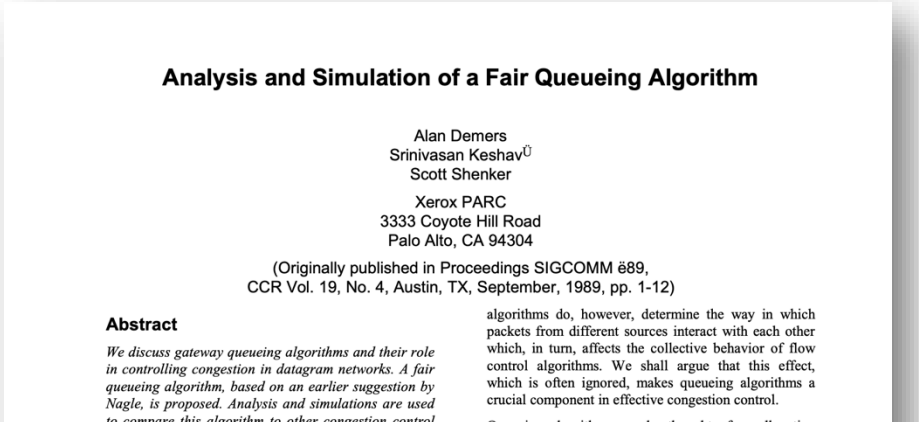
Share the CPU *proportionally* to per-job "weights"

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 Keshav[Ü]
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

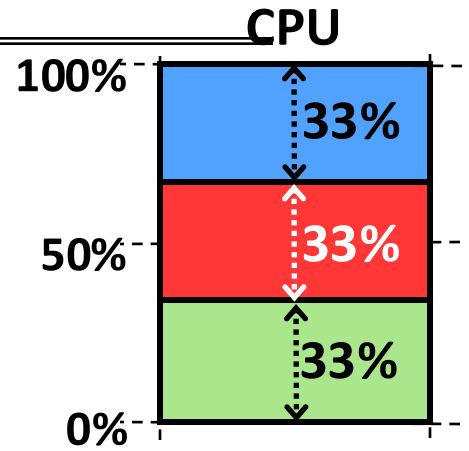
We discuss gateway queueing algorithms and their role in controlling congestion in datagram networks. A fair queueing algorithm, based on an earlier suggestion by Nagle, is proposed. Analysis and simulations are used

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.

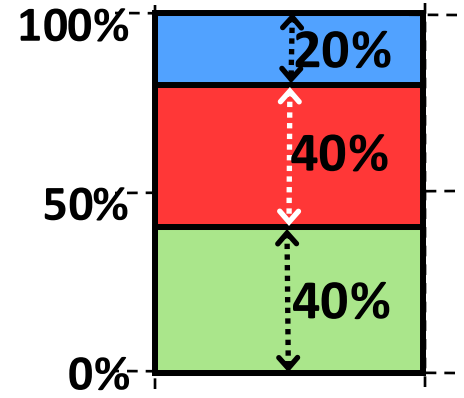
Fair sharing aims to maintain a global property (fairness) instead of reasoning about queues and priorities!

Fair Sharing Variants

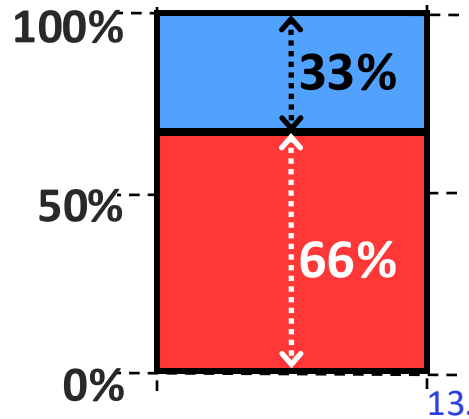
- *n* users want to share a resource (e.g. CPU)
  - Solution: give each *1/n* of the shared resource



- 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



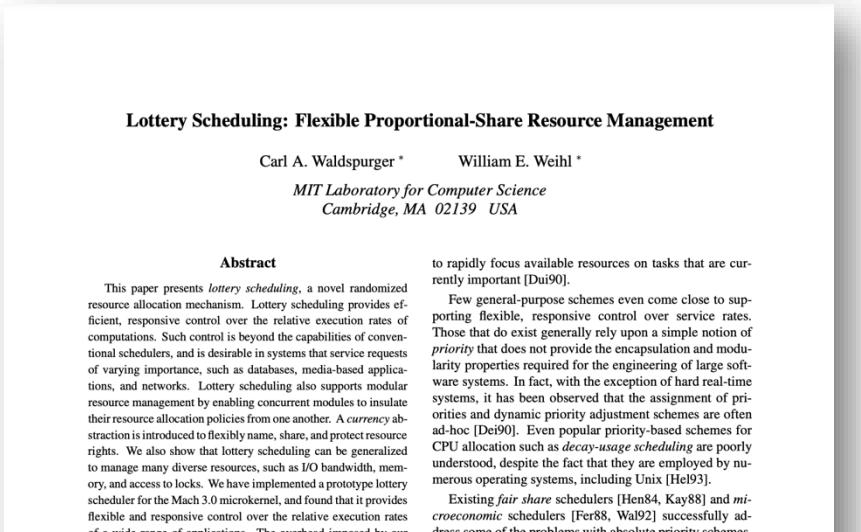
Early Example: Lottery Scheduling

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-

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

Maintain a "pass" counter for each job, and advance by stride each time it runs

– But gets tricky when jobs sleep or join or leave

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" or Generalized Processor Sharing (GPS)

Fair sharing could then be done via weighted bit-by-bit round-robin

- 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)

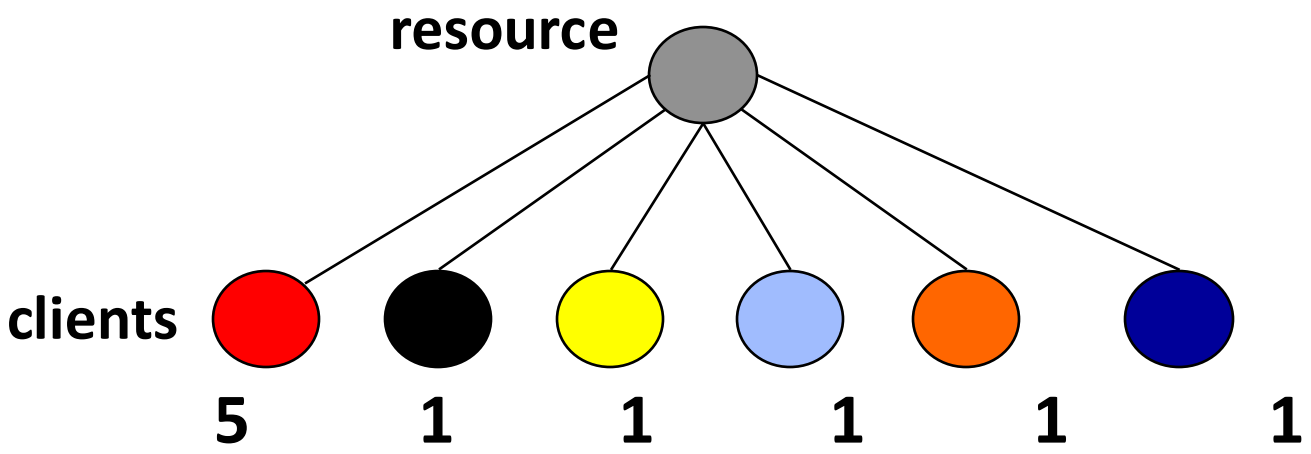
GPS Example

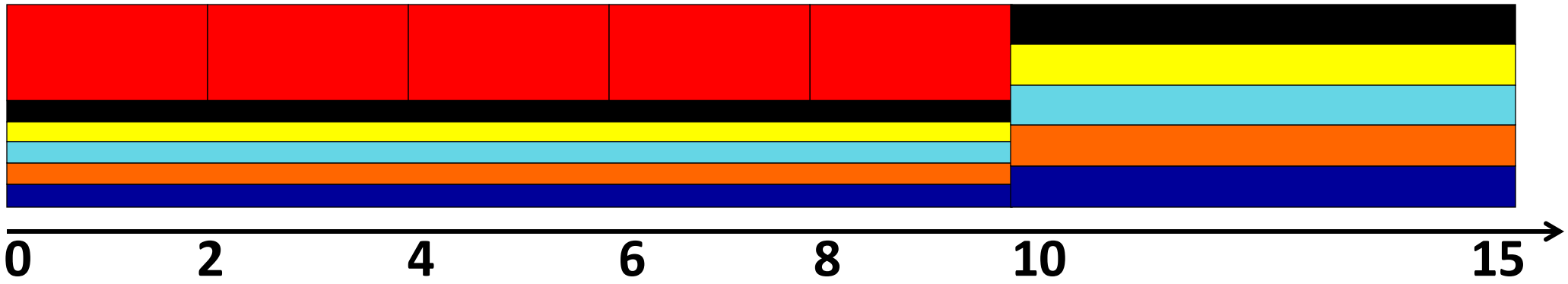
Red client has weight 5, and has 5 "packets" of work at time 0

– 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

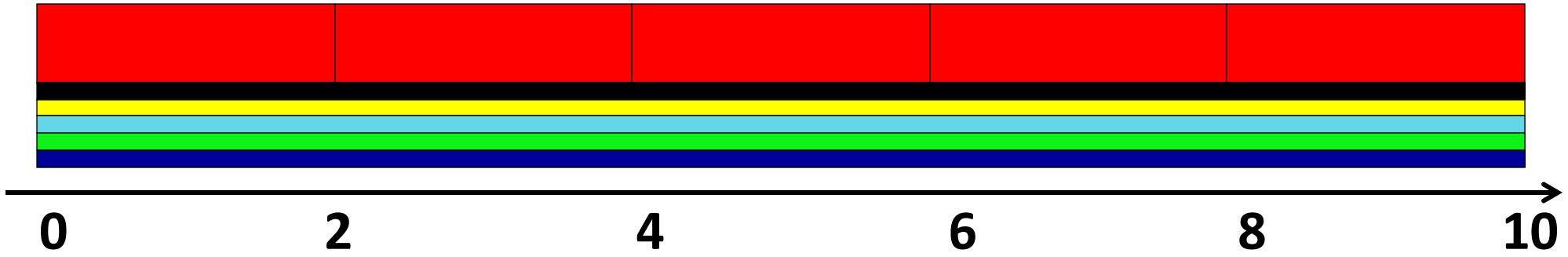
Emulate GPS

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

– Known as “Weighted Fair Queuing”, WFQ

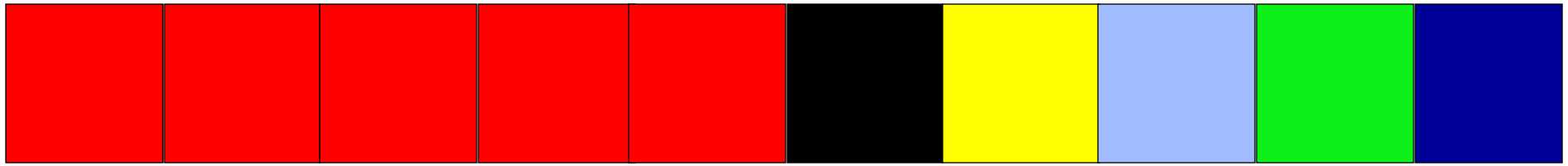
Approximating GPS

Fluid GPS system service order



Approximation: select the first packet finishes in GPS

– Known as “Weighted Fair Queueing”, WFQ



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)!

Solution: Virtual Time

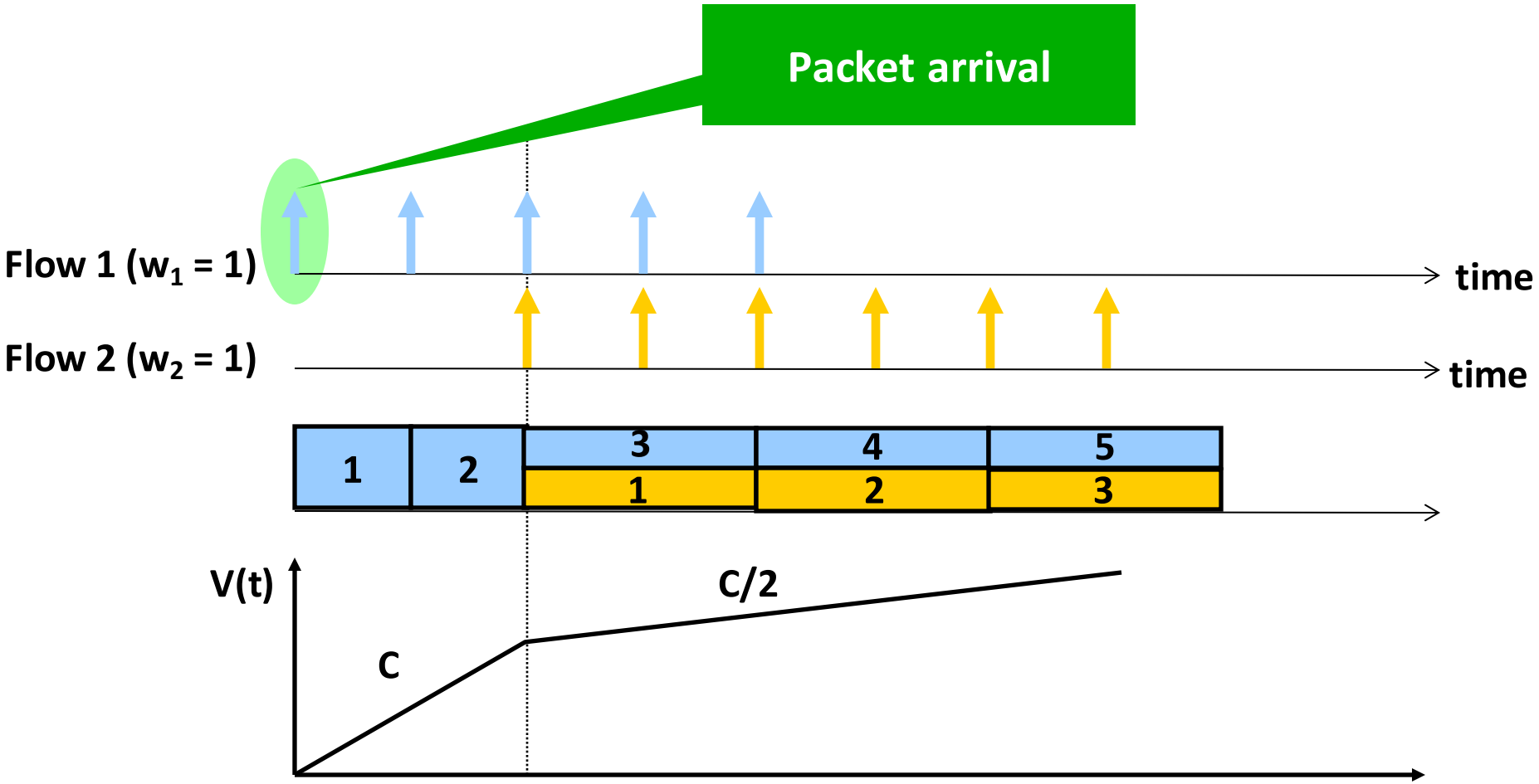
Instead of computing packet finish times, track the number of rounds needed to send the remaining bits of the packet in GPS (virtual finish time)

- 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: Example

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



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

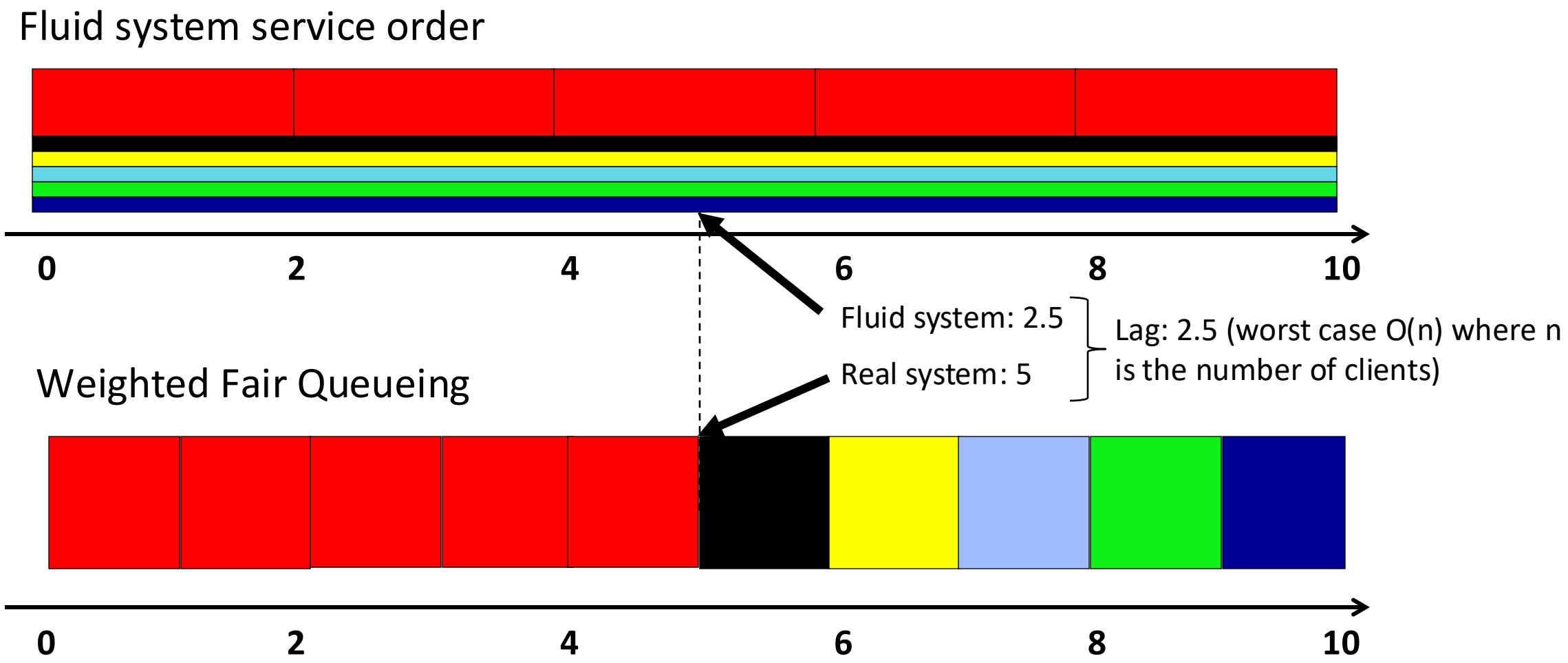
Nice Property of WFQ

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 packet experiences at most q extra delay compared to GPS

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?

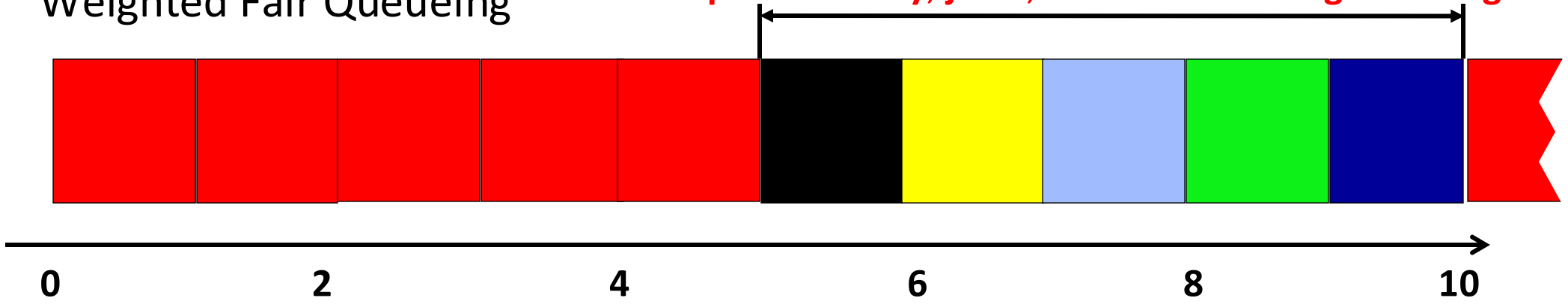
Minimize lag: the difference between service received in real system vs fluid flow (idealized) system

Fluid system service order

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

Bad for **predictability**, **jitter**, and **network congestion algorithms**

Weighted Fair Queueing



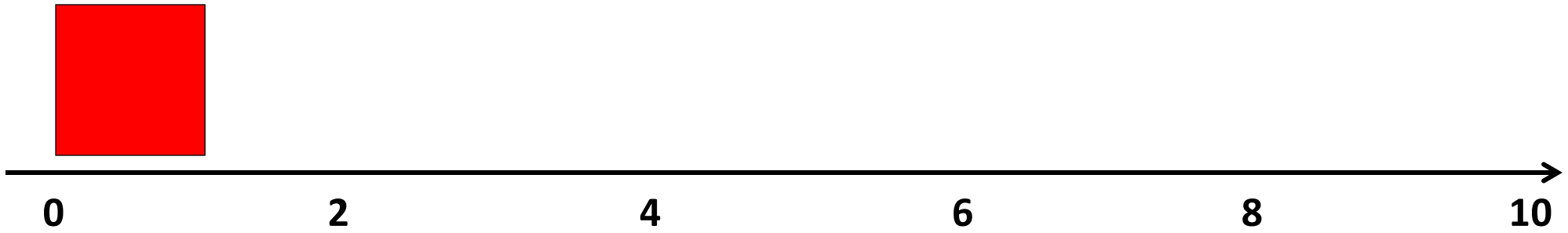
How?

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

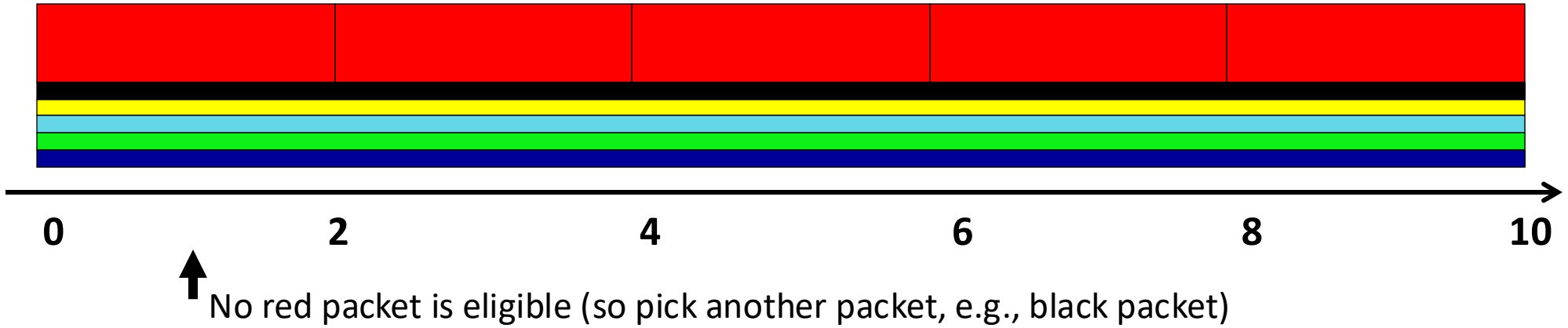
EEVDF



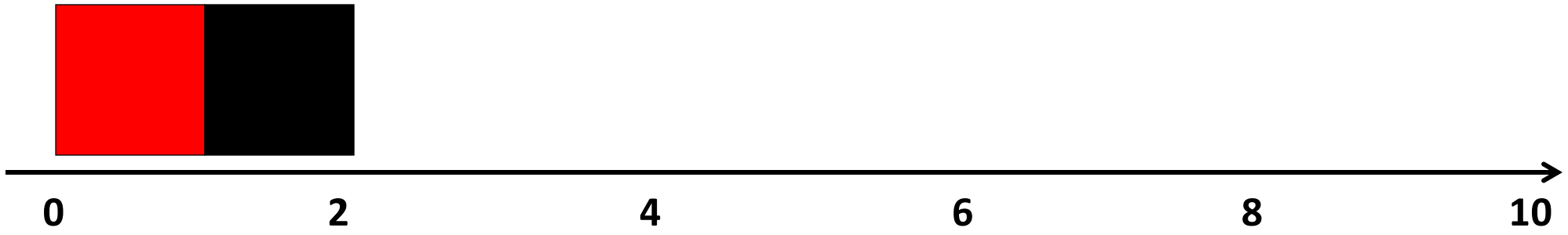
How?

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



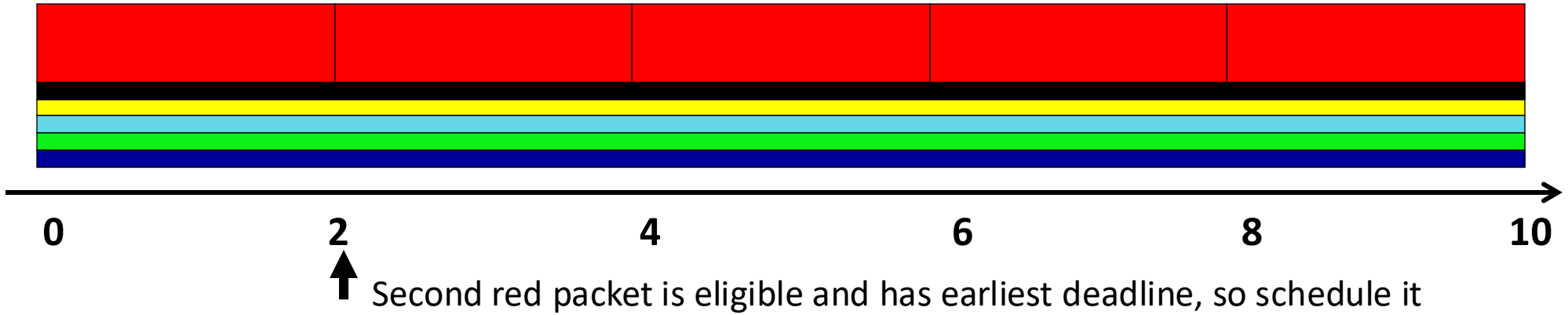
EEVDF



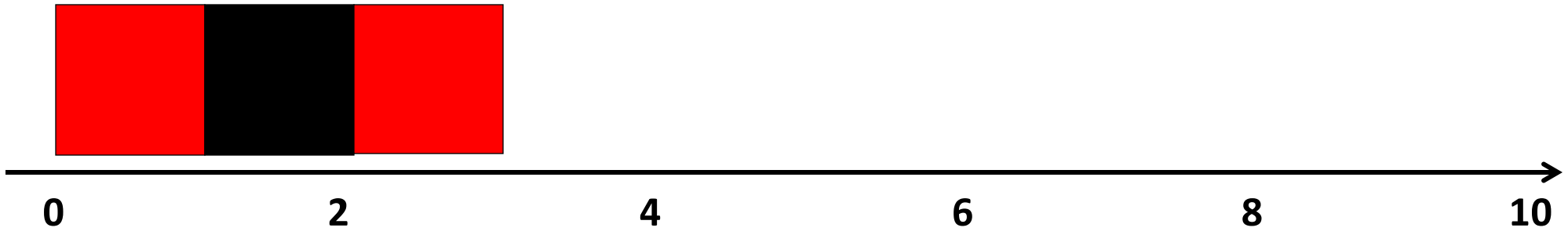
How?

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



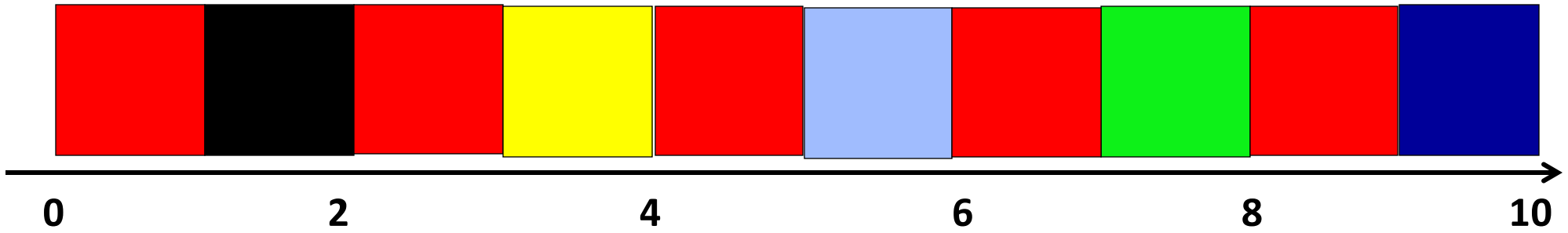
How?

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

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

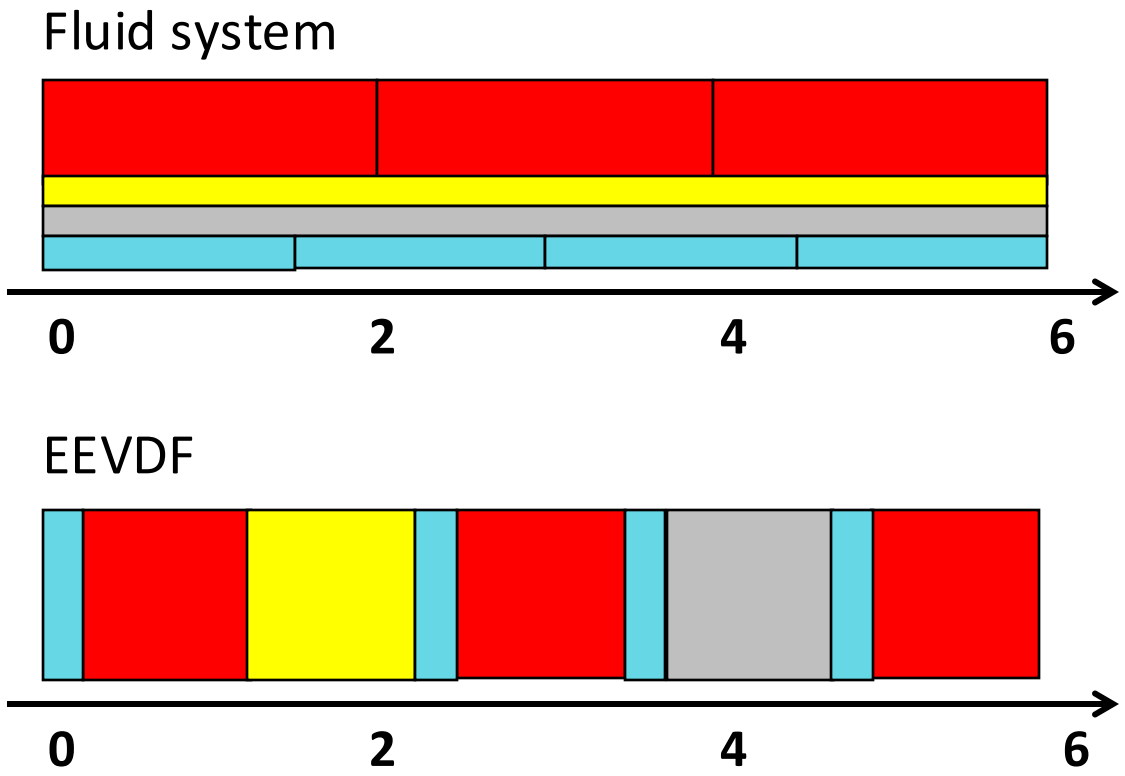
Configuring “Packet Lengths” in CPU Scheduling

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 0.25s



Configuring “Packet Lengths” in CPU Scheduling

Can clients cheat and get more total 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_vtime + vtime_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 Linux 2.4 to Linux 2.6	Did not scale with large number of processes
O(1) scheduler Linux 2.6 to 2.6.22	MLFQ, but got very complex
CFS scheduler Linux 2.6.23 to 6.6	Proportional Fair Sharing, but can have suboptimal lag for interactive tasks
EEVDF scheduler Linux 6.6 onward	Proportional Fair Sharing with low lag, fewer heuristics than CFS

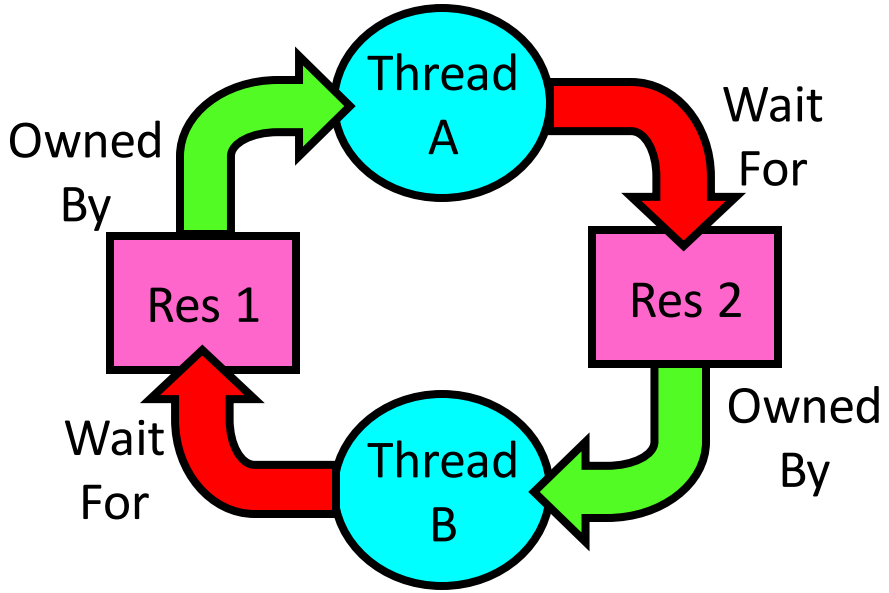
Understanding Deadlock

I will if you will

I will if you will

Deadlock: A Deadly type of Starvation

Deadlock: cyclic waiting for resources



Thread A owns Res 1 and is waiting for Res 2

Thread B owns Res 2 and is waiting for Res 1

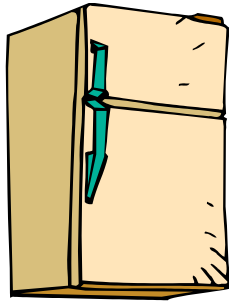
Deadlock: A Deadly type of Starvation

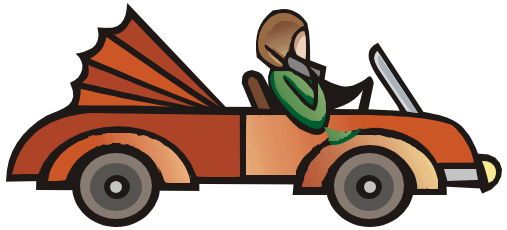
Starvation: thread waits indefinitely

Deadlock implies starvation
but starvation does not imply deadlock

Starvation can end (but doesn't have to)
Deadlock can't end without external intervention

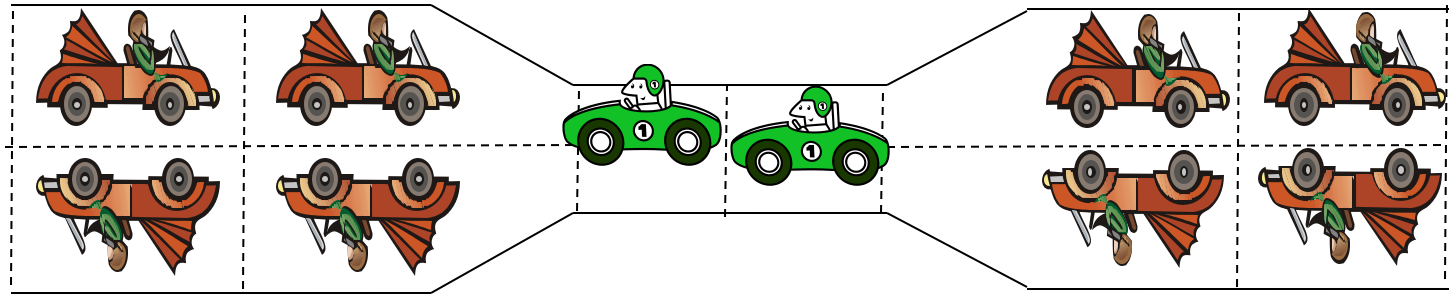
Example: Single-Lane Bridge Crossing





Bridge Crossing Example

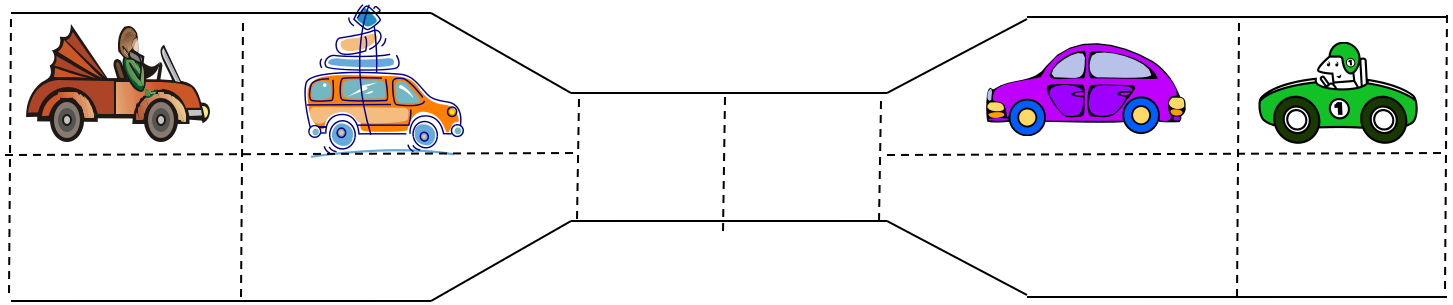
Each segment of road can be viewed as a resource



Rules:

- Car must own the segment under them
- Must acquire segment that they are moving into
- For bridge: traffic only in one direction at a time

Bridge Crossing Example

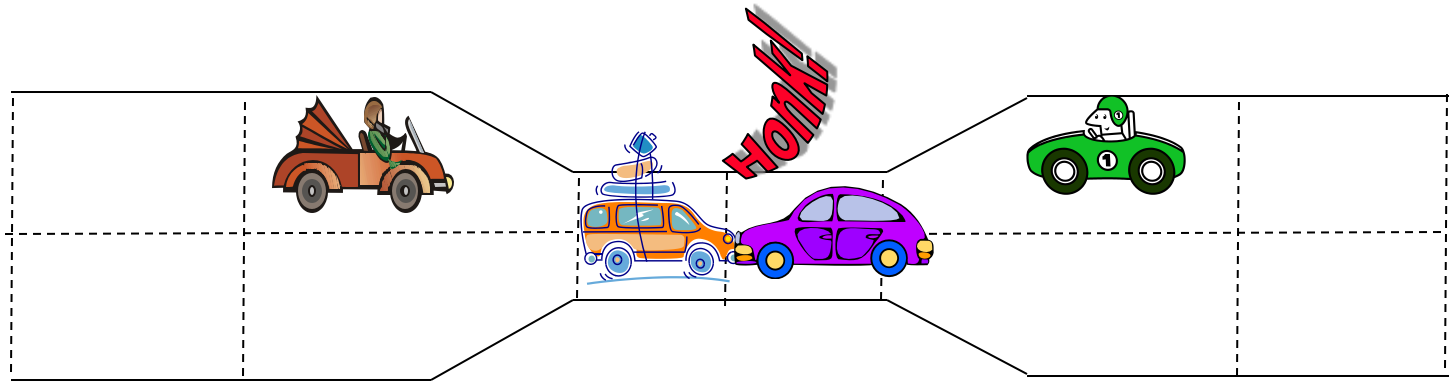


Car must own the segment under them

Must acquire segment that they are moving into

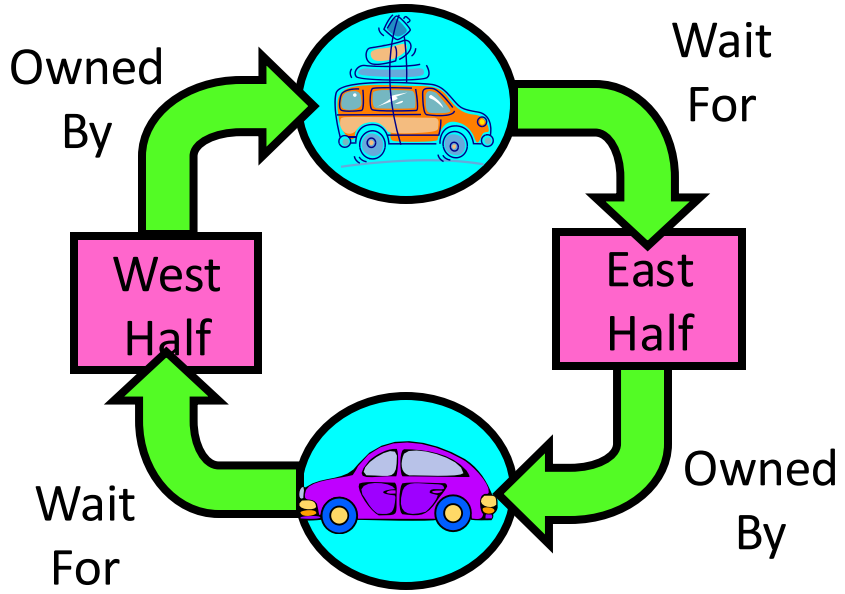
For bridge: traffic only in one direction at a time

Bridge Crossing Example



Deadlock:

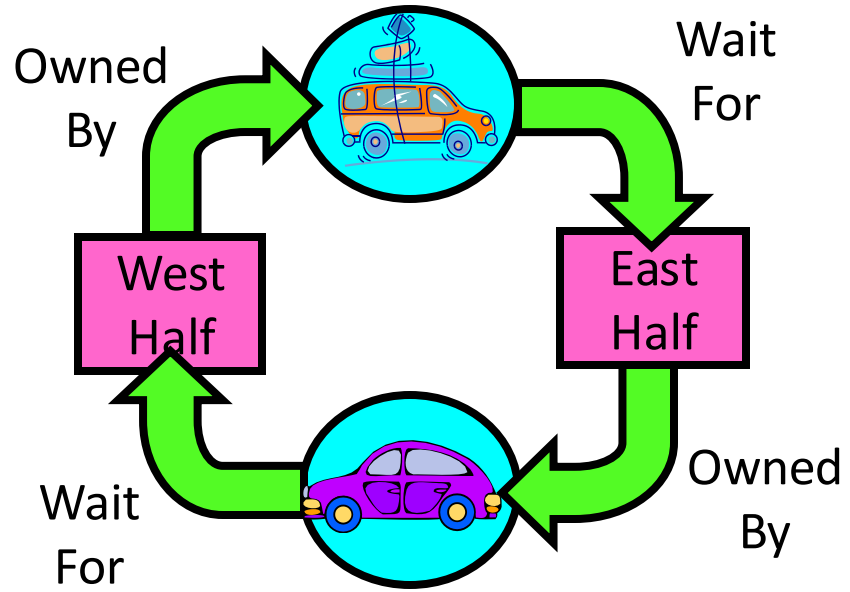
Circular waiting for resources



Bridge Crossing Example

Deadlock:

Circular waiting for resources

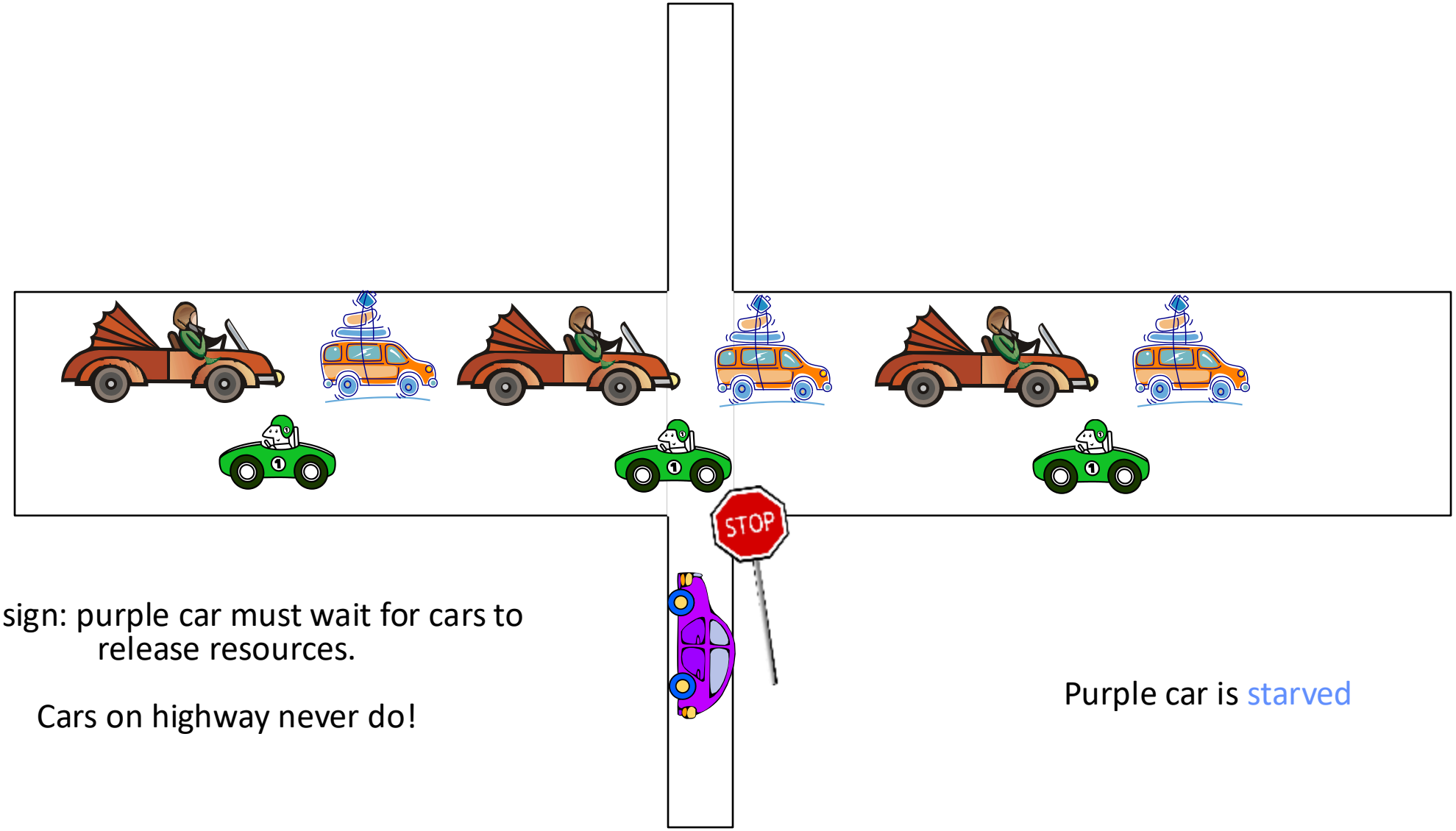


Could be resolved by "external" intervention:

- fork-lifting a car off the bridge (equivalent to killing a thread)

- Asking cars to back up (equivalent to removing the resource from the thread)

Starvation does not mean deadlock!



Stop sign: purple car must wait for cars to release resources.

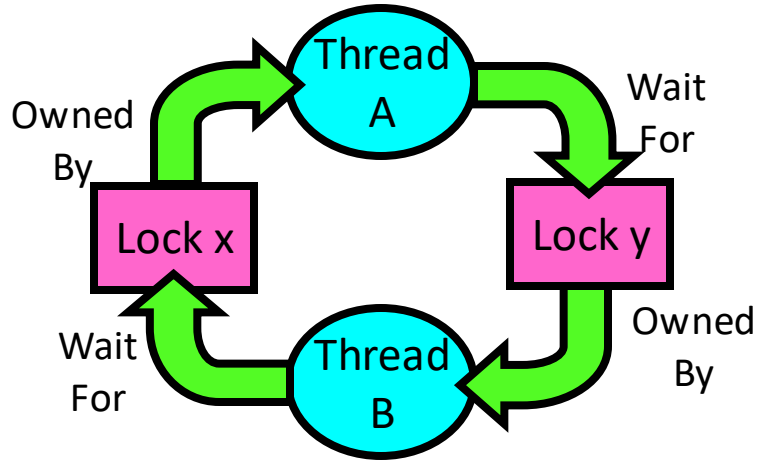
Cars on highway never do!

Purple car is starved

Deadlock with Locks

Thread A:
```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:
```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```



Will threads deadlock

a) Always b) Never c) Sometimes d) I'm still trying to cross the road

This lock pattern exhibits *non-deterministic deadlock*

A system is subject to deadlock if deadlock can happen in any execution

Deadlock with Locks: "Lucky" Case

Thread A:

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:

```
y.Acquire();

x.Acquire();
…
x.Release();
y.Release();
```

Sometimes, schedule won't trigger deadlock!

Other Types of Deadlock

Threads can block waiting for resources

- Locks
- Terminals
- Printers
- Memory

Threads can block waiting for other threads

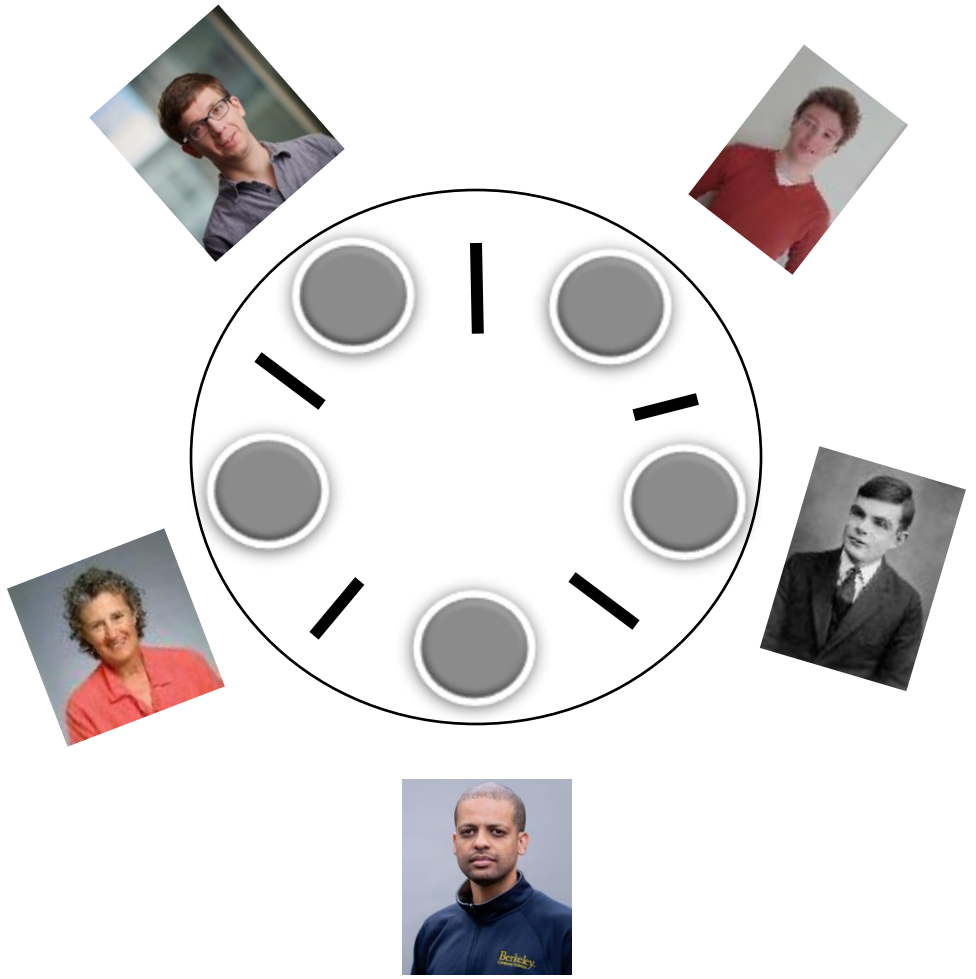
- Pipes
- Sockets
- pthread_join

You can deadlock on any of these!

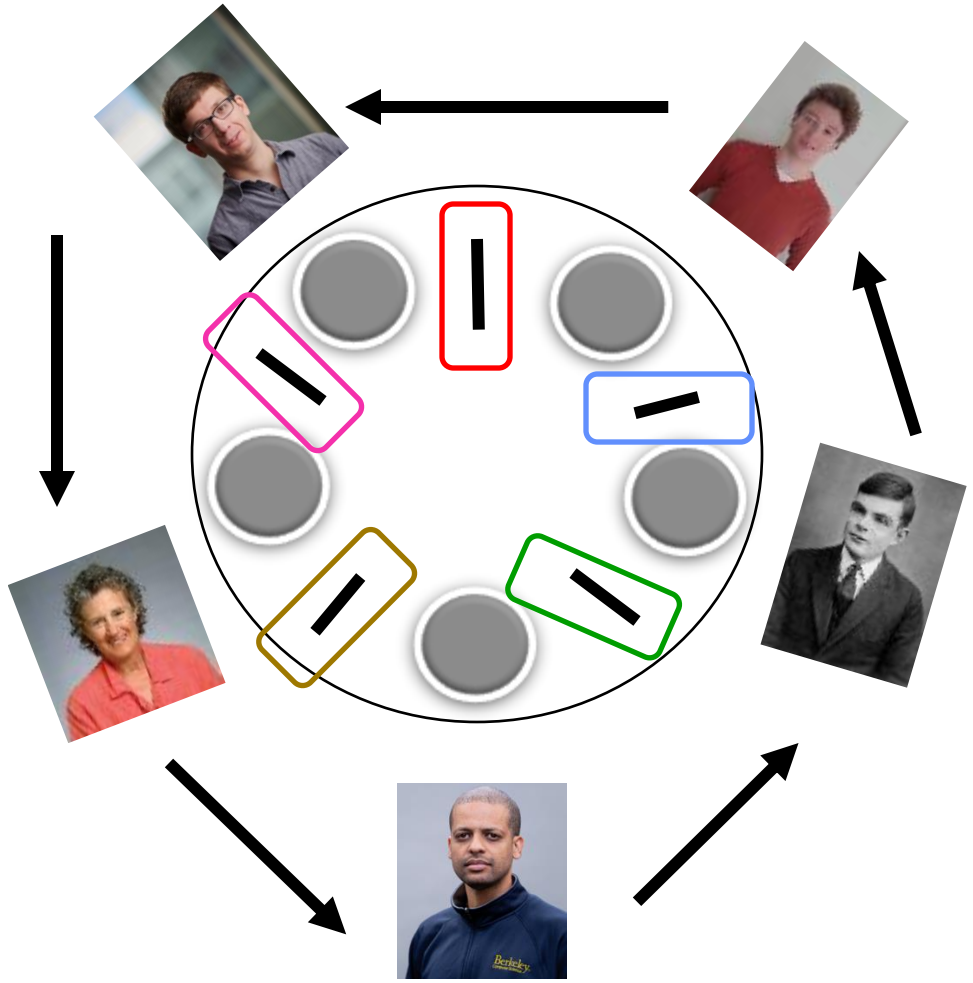
Dining Computer Scientists Problem

Five chopsticks/Five computer scientists

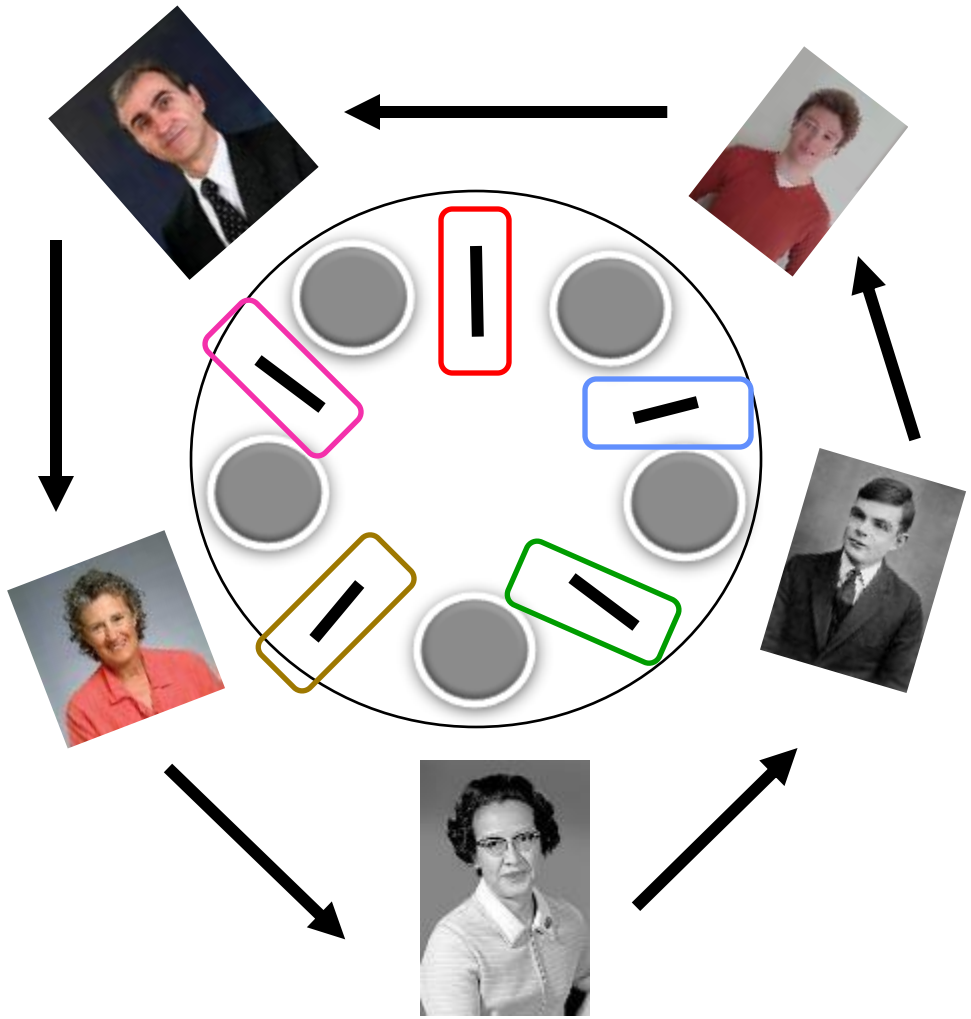
Need two chopsticks to eat



Free for all leads to deadlock



Intervention needed



Fixing deadlock needs external intervention!

How could we have prevented this?

- Give everyone two chopsticks

- Make everyone “give up” after a while

- Require everyone to pick up both chopsticks atomically

Four requirements for occurrence of deadlock

1) Mutual exclusion and bounded resources

Only one thread at a time can use a resource.

2) Hold and wait

Thread holding at least one resource is waiting to acquire additional resources held by other threads

Four requirements for occurrence of deadlock

3) No preemption

Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

4) Circular wait

There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads

» T_1 is waiting for a resource that is held by T_2

» T_2 is waiting for a resource that is held by T_3

…

» T_n is waiting for a resource that is held by T_1

Detecting Deadlock: Resource-Allocation Graph

System Model

A set of Threads T_1, T_2, . . ., T_n

Resource types R_1, R_2, . . ., R_m

CPU cycles, memory space, I/O devices

Each resource type R_i has W_i instances

Each thread

`Request()` / `Use()` / `Release()` a resource:

Detecting Deadlock: Resource-Allocation Graph

Resource-Allocation Graph

– Vertices of two types:

$T = \{T_1, T_2, \ldots, T_n\}$,

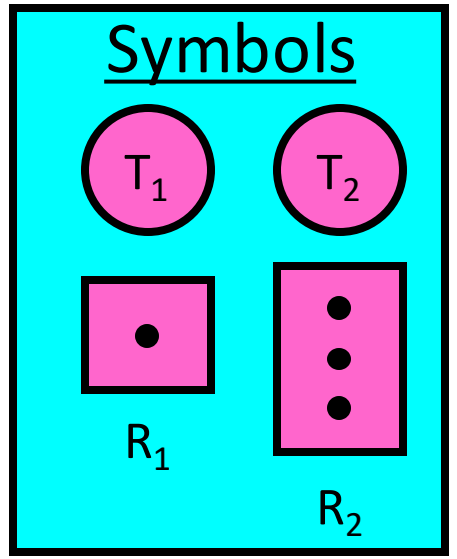
the set threads in the system.

$R = \{R_1, R_2, \ldots, R_m\}$,

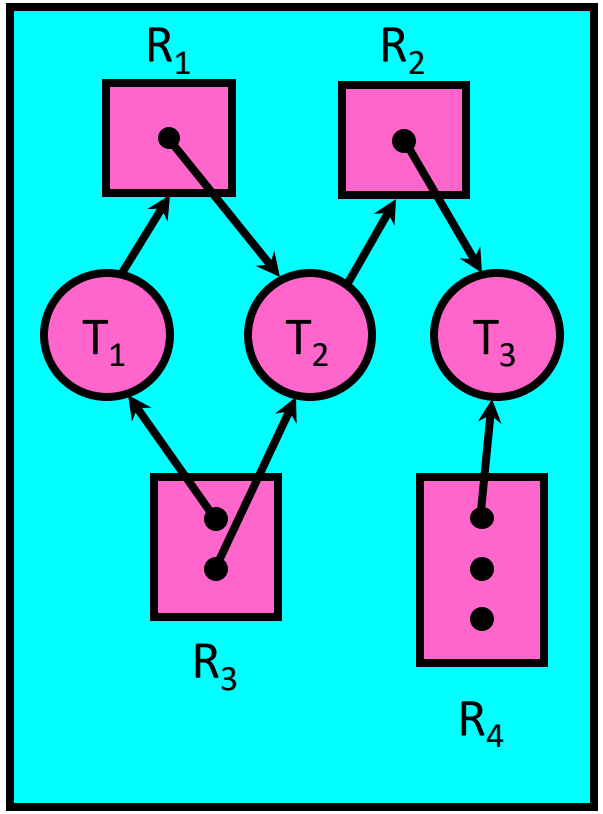
the set of resource types in system

– request edge – directed edge $T_1 \rightarrow R_j$

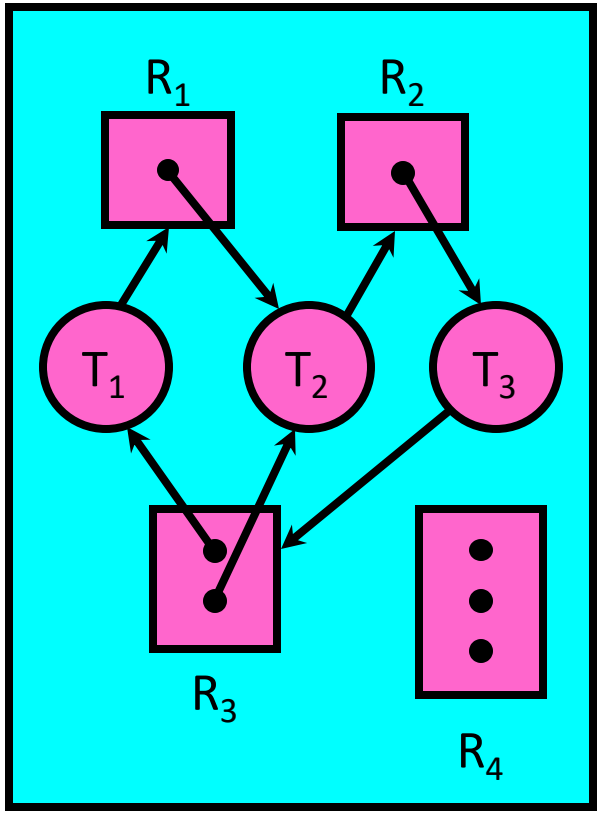
– assignment edge – directed edge $R_j \rightarrow T_i$



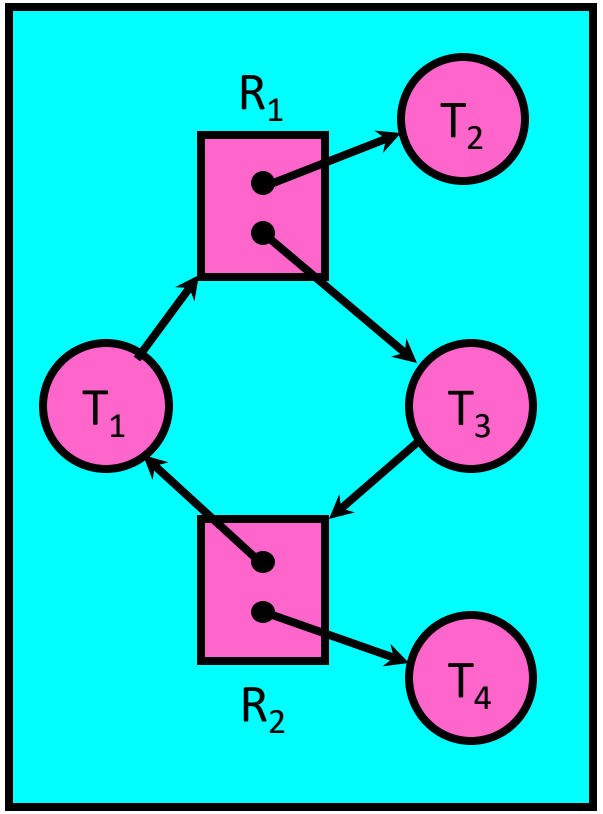
Resource-Allocation Graph Examples



Simple Resource Allocation Graph



Allocation Graph with Deadlock



Allocation Graph with Cycle, No Deadlock

Deadlock Detection Algorithm

Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type)

`[FreeResources]:`	Current free resources each type
`[Request`$_T$`]:`	Current requests from thread T
`[Alloc`$_T$`]:`	Current resources held by thread T

Deadlock Detection Algorithm

See if tasks can eventually terminate on their own

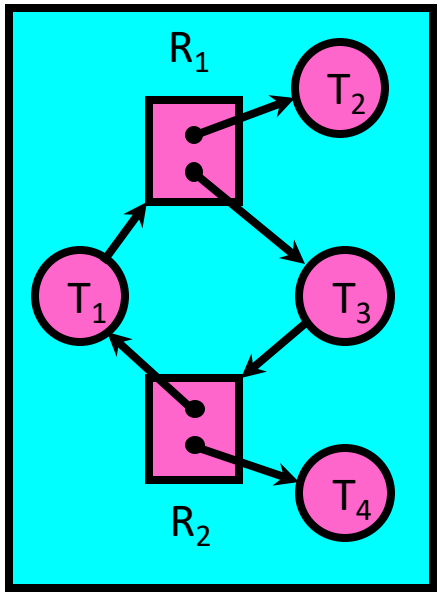
```
[Avail] = [FreeResources]
add all threads to UNFINISHED
do {
   done = true
   foreach thread in UNFINISHED {
      if ([Request_thread] <= [Avail]) {
         remove thread from UNFINISHED
         [Avail] = [Avail] + [Alloc_thread]
         done = false
      }
   }
} until(done)
```

Threads left in UNFINISHED ⇒ deadlocked

Deadlock Detection Algorithm

```
[Avail] = [FreeResources]
        add all threads to UNFINISHED
        do {
            done = true
            foreach thread in UNFINISHED {
                if ([Request_thread] <= [Avail]) {
                    remove thread from UNFINISHED
                    [Avail] = [Avail] + [Alloc_thread]
                    done = false
                }
            }
        } until(done)
```

Threads left in UNFINISHED ⇒ deadlocked



[Avail] = {0,0}
UNFINISHED = T1, T2, T3, T4

Looking at T1: [1,0] > [0,0]

Looking at T2: [0,0] <= [0,0]
Avail = [1,0]
UNFINISHED = T1,T3,T4

Looking at T3: [0,1] > [1,0]

Looking at T4
[0,0] <= [0,0]
Avail = [1,1]
UNFINISHED = T1, T3

Looking at T1: [1,0] <= [1,1]
Avail = [2,1]
UNFINISHED = T3

Looking at T3: [0,1] <= [2,1]
Avail = [2,2]
UNFINISHED = Empty!

How should a system deal with deadlock?

Deadlock prevention

Write your code in a way that it isn't prone to deadlock

Deadlock recovery

Let deadlock happen, and figure out how to recover from it

Deadlock avoidance

Dynamically delay resource requests so deadlock doesn't happen

~~Deadlock denial~~

~~Ignore the possibility of deadlock~~

Deadlock prevention

Condition 1: Mutual exclusion and bounded resources

=> Provide sufficient resources

Condition 2: Hold and wait

=> Abort requests or acquire all resources atomically

Condition 3: No preemption

=> Preempt threads

Condition 4: Circular wait

=> Order resources and always acquire resources in the same way

Condition 1 Fix: (Virtually) Infinite Resources

```
Thread A                    Thread B
AllocateOrWait(1 MB)        AllocateOrWait(1 MB)
AllocateOrWait(1 MB)        AllocateOrWait(1 MB)
Free(1 MB)                  Free(1 MB)
Free(1 MB)                  Free(1 MB)
```

With virtual memory we have “infinite” space so everything will always succeed

Condition 2 Fix: Request Resources Atomically

Rather than:

```
Thread A:                Thread B:
x.Acquire();             y.Acquire();
y.Acquire();             x.Acquire();
…                        …
y.Release();             x.Release();
x.Release();             y.Release();
```

Consider instead:

```
Thread A:                Thread B:
Acquire_both(x, y);      Acquire_both(y, x);
…                        …
y.Release();             x.Release();
x.Release();             y.Release();
```

Condition 3 Fix: Preemption

Force thread to give up resource

Common technique in databases using transaction aborts

- A transaction from a user can be "aborted" by the DB while running: all of its actions are undone, and user must retry the transaction

Common technique in wireless networks:

- Everyone speaks at once. When a resource collision is detected, retry at a new, random time

Condition 4 Fix: Circular Waiting

Force all threads to request resources

in the same order

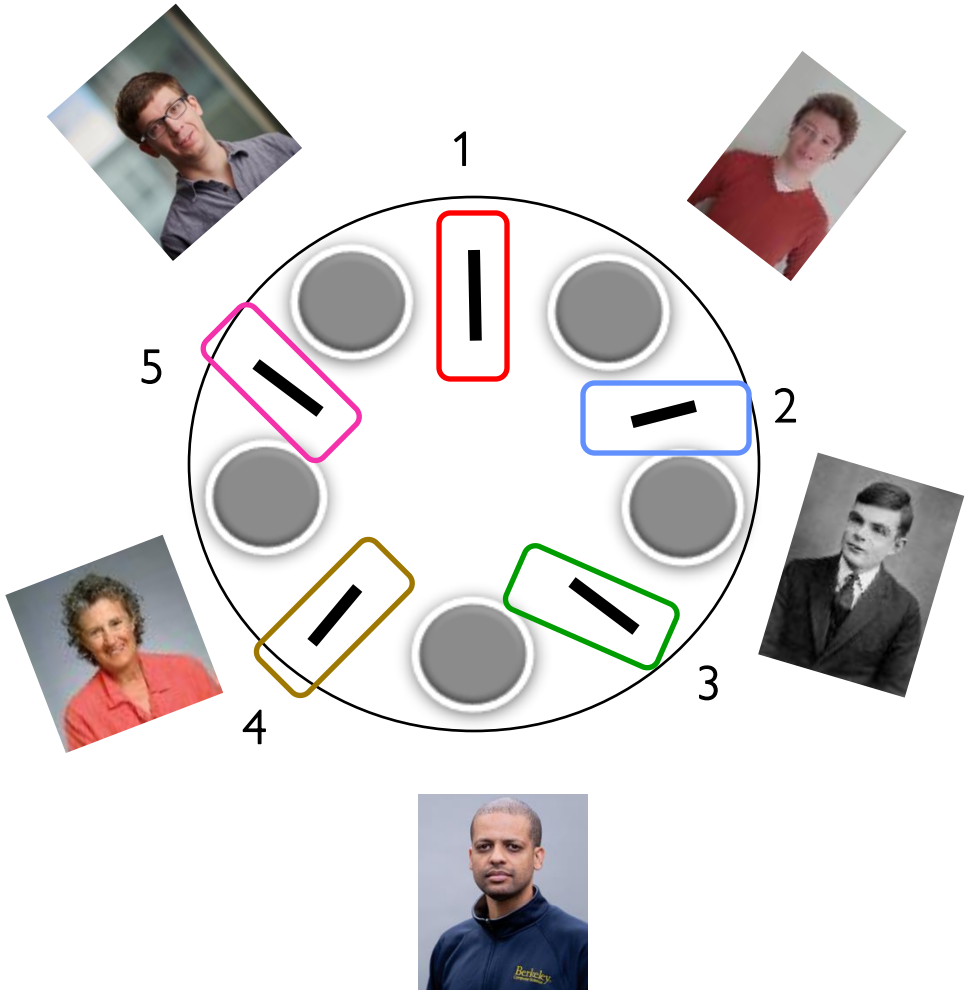
```
Thread A:
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

```
Thread B:
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

```
Thread A:
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

```
Thread B:
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Condition 4 Fix: Circular Waiting



Matei: first 1 then 5

Crooks: first 2 then 1

Turing: first 3 then 2

Nelson: first 4 than 3

Liskov: first 5 then 4

If we instead ensure that Matei always grabs chopstick 5 before 1, (higher ID first), no deadlock!

How should a system deal with deadlock?

Deadlock prevention

Write your code in a way that it isn't prone to deadlock

Deadlock recovery

Let deadlock happen, and figure out how to recover from it

Deadlock avoidance

Dynamically delay resource requests so deadlock doesn't happen

Techniques for Deadlock Avoidance

Attempt 1

When a thread requests a resource, OS checks if it would result in deadlock

If not, it grants the resource right away

If so, it waits for other threads to release resources

Techniques for Deadlock Avoidance

This does not work!

	Thread A:	Thread B:	
	`x.Acquire();`	`y.Acquire();`	
Blocks…	`y.Acquire();`	`x.Acquire();`	**Wait?**
	`…`	`…`	
	`y.Release();`	`x.Release();`	**But it's already too late…**
	`x.Release();`	`y.Release();`	

Deadlock Avoidance: Three States

Safe state

System can delay resource acquisition to prevent deadlock

Unsafe state

No deadlock yet…

But threads can request resources in a pattern that ***unavoidably*** leads to deadlock

Deadlocked state

There exists a deadlock in the system

Deadlock avoidance: prevent system from reaching an *unsafe* state

Deadlock Avoidance: Three States

Thread A:

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

A acquires x.

There exists a deadlock-free sequence: A-A(y), A-R(y), A-R(x), B-A(y), B-A(x), B-R(x), B-R(y) => safe state

B acquires y.

All sequences will lead to deadlock => unsafe state

Banker's Algorithm for Avoiding Deadlock

Banker's algorithm ensures we never enter an unsafe state

Evaluate each request and grant if some ordering of threads is still deadlock free afterward

Technique: pretend each request is granted, then run our deadlock detection algorithm

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
     add all threads to UNFINISHED
     do {
        done = true
        Foreach thread in UNFINISHED {
           if ([Request_thread] <= [Avail]) {
              remove thread from UNFINISHED
              [Avail] = [Avail] + [Alloc_thread]
              done = false
           }
        }
     } until(done)
```

```
[Avail] = [FreeResources]
     add all threads to UNFINISHED
     do {
        done = true
        Foreach threads in UNFINISHED {
           if ([Max_thread]-[Alloc_thread] <= [Avail]) {
              remove thread from UNFINISHED
              [Avail] = [Avail] + [Alloc_thread]
              done = false
           }
        }
     } until(done)
```

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
      add all threads to UNFINISHED
      do {
         done = true
         Foreach threads in UNFINISHED {
            if ([Max_thread]-[Alloc_thread] <= [Avail]) {
               remove thread from UNFINISHED
               [Avail] = [Avail] + [Alloc_thread]
               done = false
            }
         }
      } until(done)
```

Step 1: "Assume" request is made

Step 2: If request is made, is system still in SAFE state?

There exists a sequence $\{T_1, T_2, \ldots T_n\}$ such that all transactions finish

Step 3: If SAFE, grant resources. If UNSAFE, delay

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
        add all threads to UNFINISHED
        do {
            done = true
            Foreach threads in UNFINISHED {
                if ([Max_thread]-[Alloc_thread] <= [Avail]) {
                    remove thread from UNFINISHED
                    [Avail] = [Avail] + [Alloc_thread]
                    done = false
                }
            }
        } until(done)
```

Thread A:

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

When Thread A acquires x:

Avail = [0,1]
For A: [1,1] – [1,0] <= [0,1]
Update Avail to = 1,1. Remove A from UNFINISHED
For B:
[1,1] – [0,0] <= [1,1]
Update Avail to = [1,1]. Remove B from UNFINISHED

Safe state!

When Thread B acquires y:

Avail = [0,0]
For A: [1,1] – [1,0] <= [0,0]
For B: [1,1] – [0,1] <= [0,0]

UNFINISHED not empty

Unsafe state! Must delay acquiring y!

Summary

Deadlock implies starvation, but starvation does not imply deadlock

Four conditions for deadlocks:
Mutual exclusion
Hold and wait
No preemption
Circular wait

Techniques for addressing deadlock: prevention, recovery, avoidance

Banker's algorithm for avoiding deadlock