CS162
Operating Systems and Systems Programming
Lecture 13

Scheduling 3: Proportional Share Scheduling, Deadlock

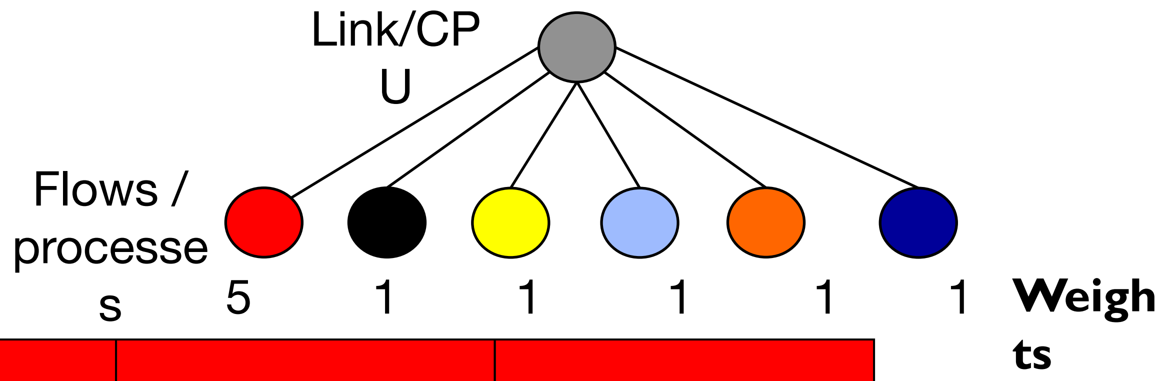
October 14th, 2024
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu

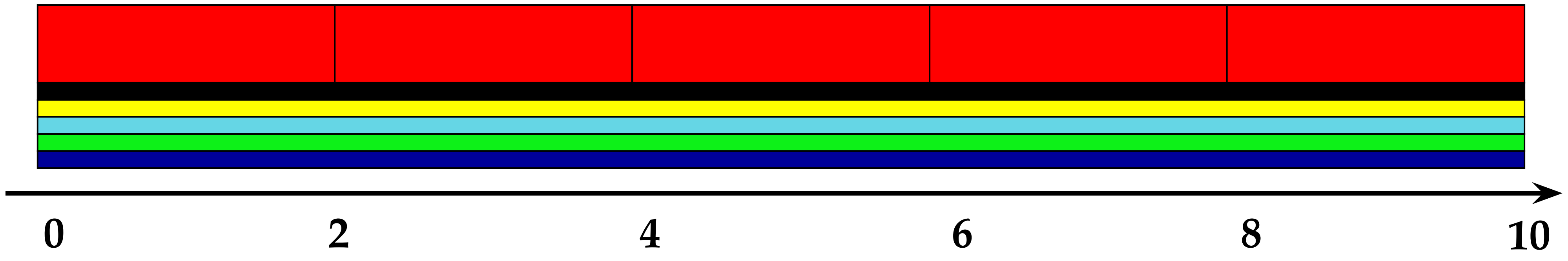
Recall: Why Max-Min Fairness?

- *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

Recall: Weighted Fair Sharing

- Fluid system:
  – Networking: bit-by-bit round-robin
  – CPU: round robin with arbitrary small time quanta

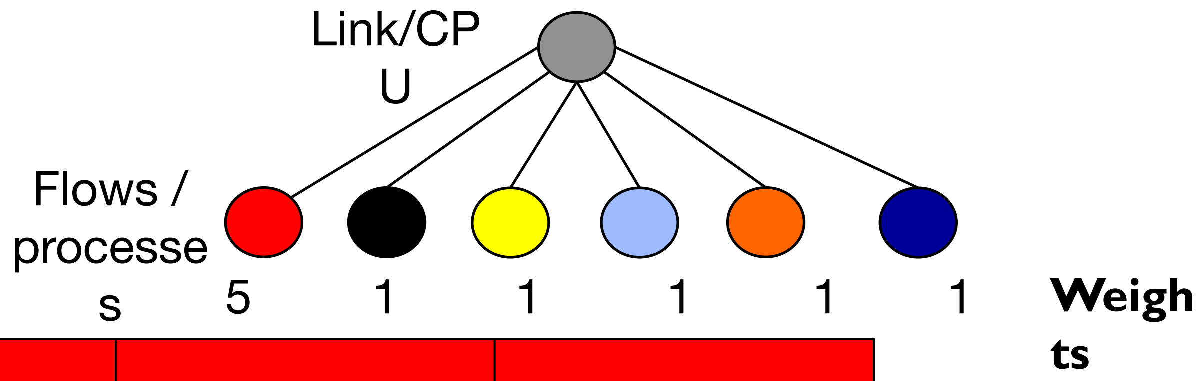


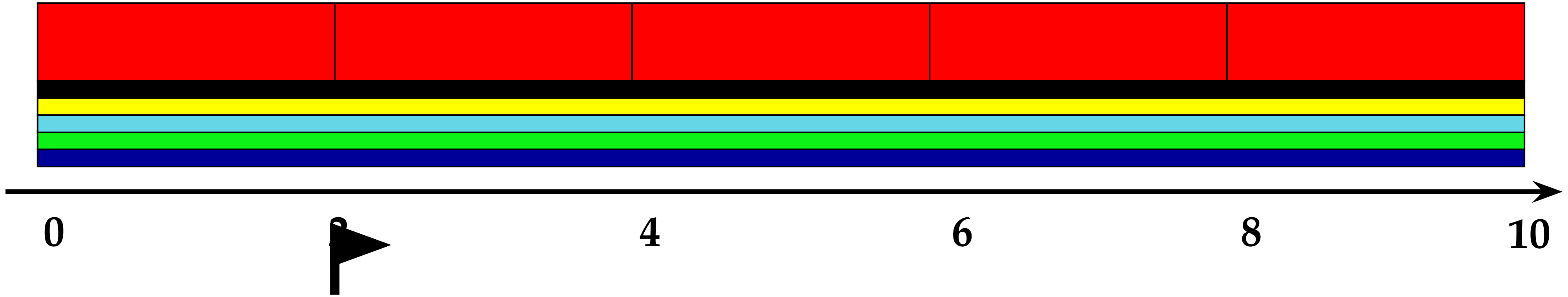


- Link/CPU capacity = 1
- Each unit of work (packet/quanta) = 1
- Total weight: 5 + 1 + 1 + 1 + 1 + 1 = 10
- Red flow: weight 5 □ gets 5/10 = 50% of network capacity or 0.5
- Every other flow: weight 1 □ gets 1/10 = 10% of network capacity or 0.1
- Since each packet's size of 1, it takes:
  - 1/0.5 = 2 time units to send a red packet
  - 1/0.1 = 10 time units to send every other packet

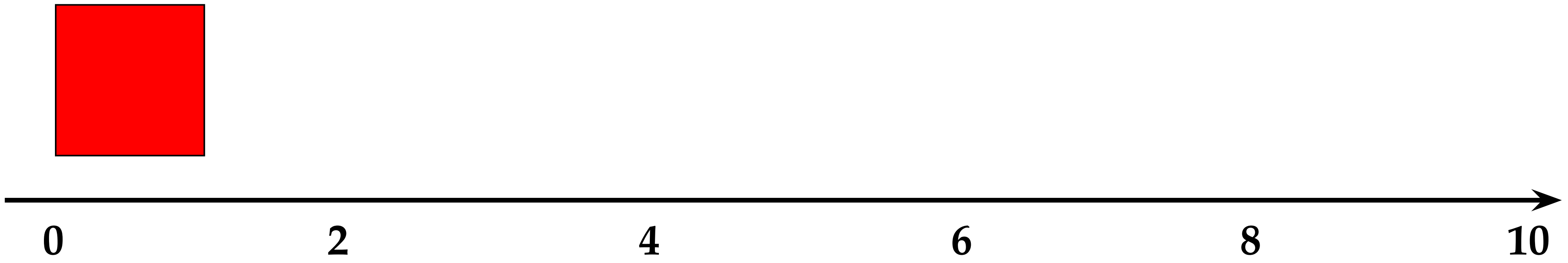
Recall: Weighted Fair Sharing

- Fluid system:
  – Networking: bit-by-bit round-robin
  – CPU: round robin with arbitrary small time quanta



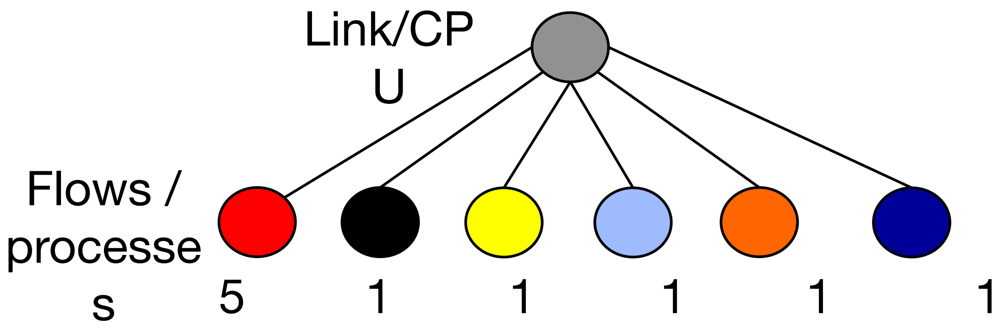


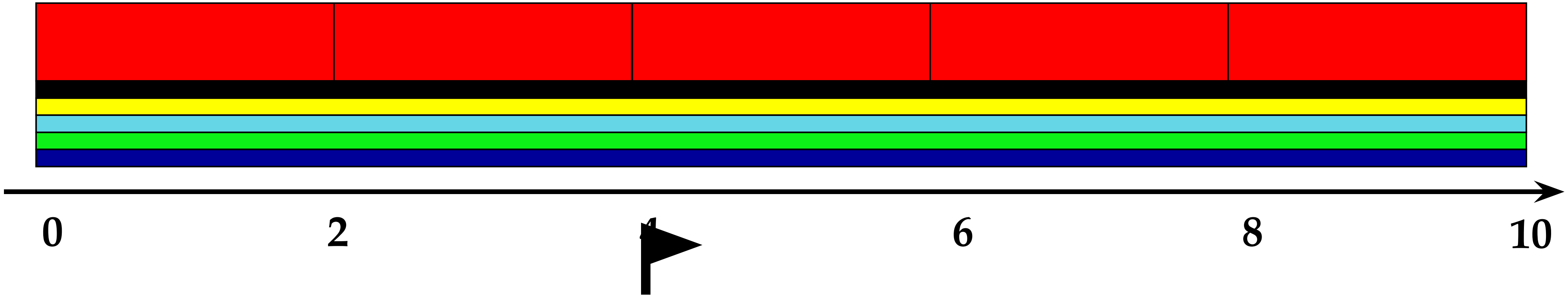
- Weighted Fair Queueing: schedule in the order of finishing time in fluid flow system



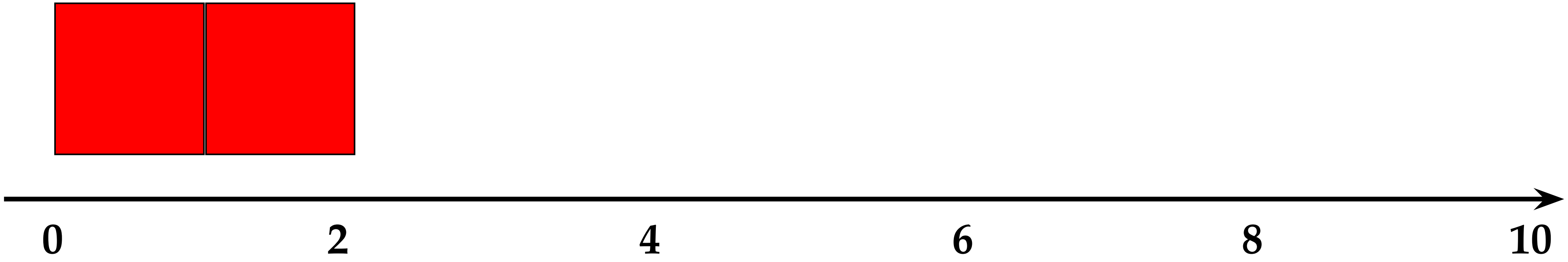
Recall: Weighted Fair Sharing

- Fluid system:
  - Networking: bit-by-bit round-robin
  - CPU: round robin with arbitrary small time quanta



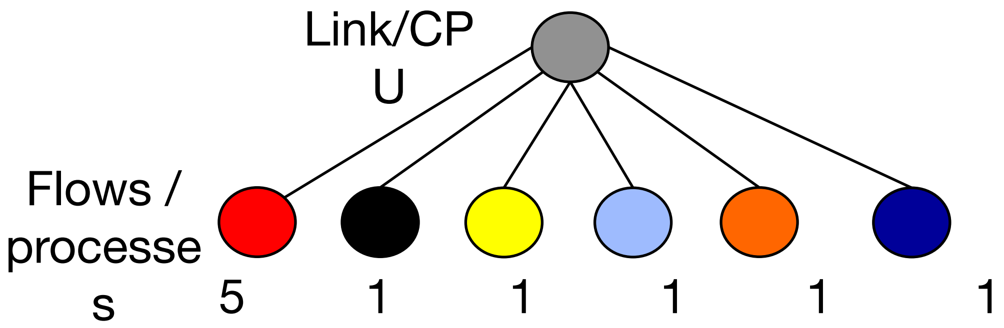


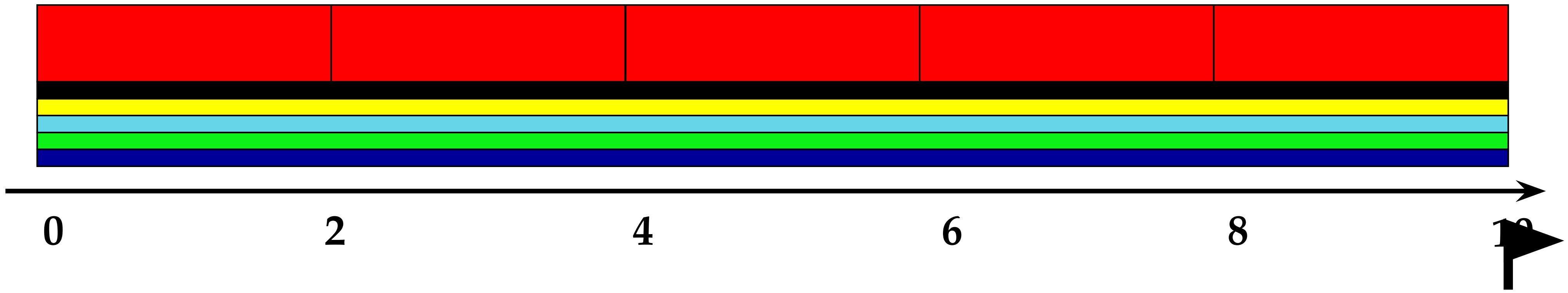
- Weighted Fair Queueing: schedule in the order of finishing time in fluid flow system



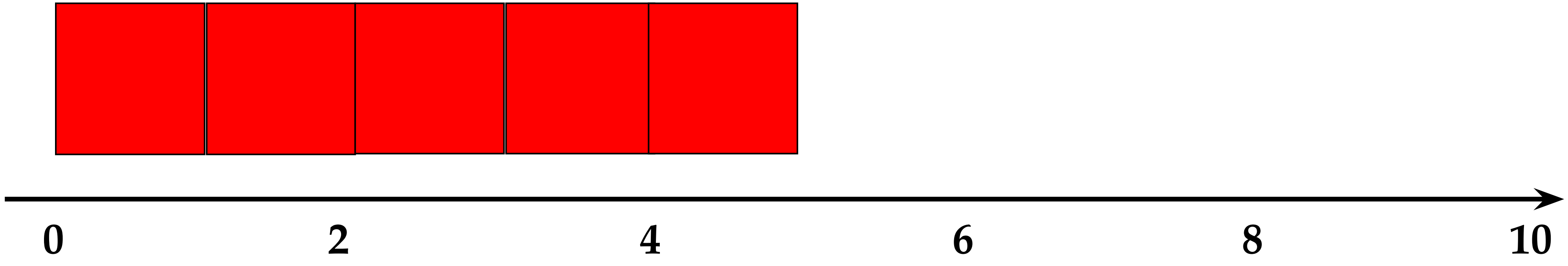
Recall: Weighted Fair Sharing

- Fluid system:
  - Networking: bit-by-bit round-robin
  - CPU: round robin with arbitrary small time quanta



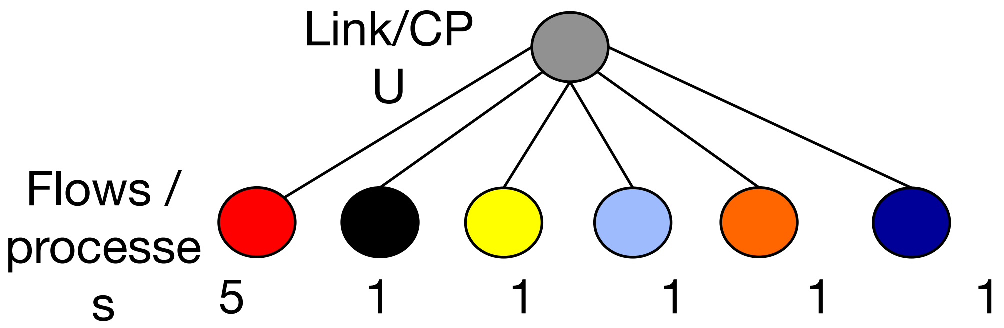


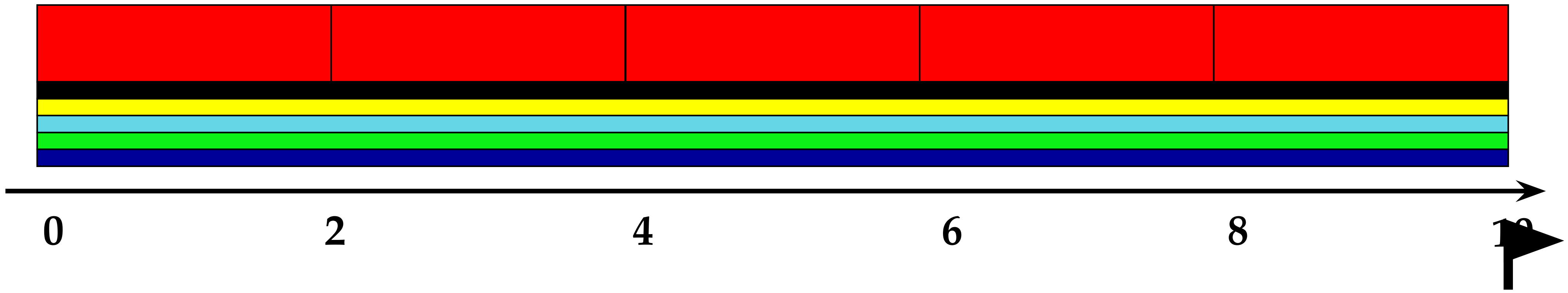
- Weighted Fair Queueing: schedule in the order of finishing time in fluid flow system



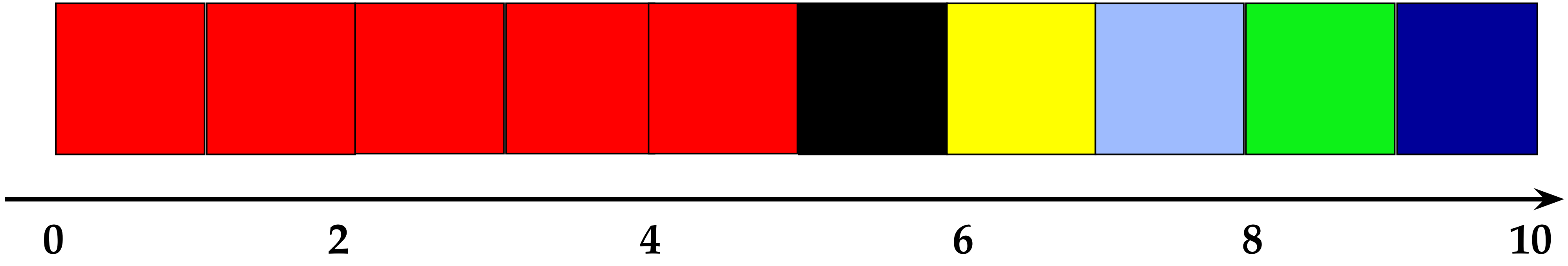
Recall: Weighted Fair Sharing

- Fluid system:
  - Networking: bit-by-bit round-robin
  - CPU: round robin with arbitrary small time quanta





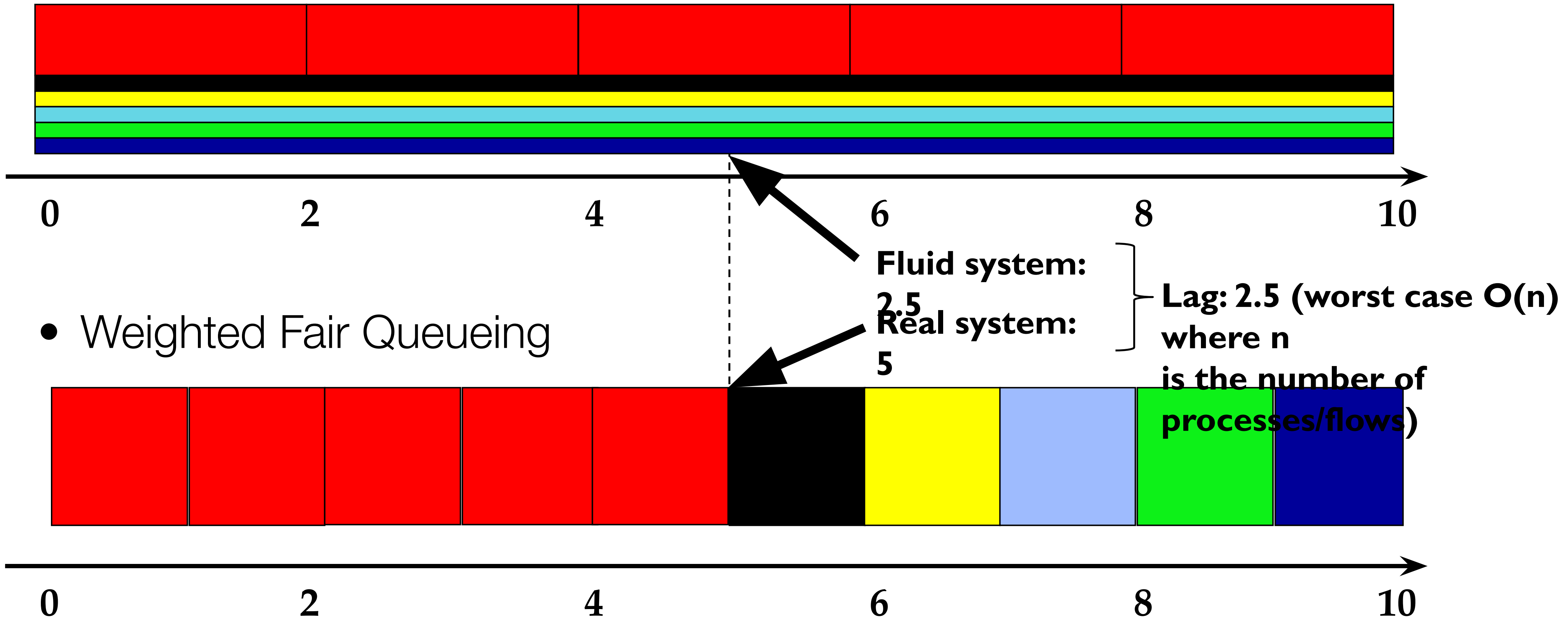
- Weighted Fair Queueing: schedule in the order of finishing time in fluid flow system



Recall: Service lag

Lag: difference between service received in real system vs fluid flow system

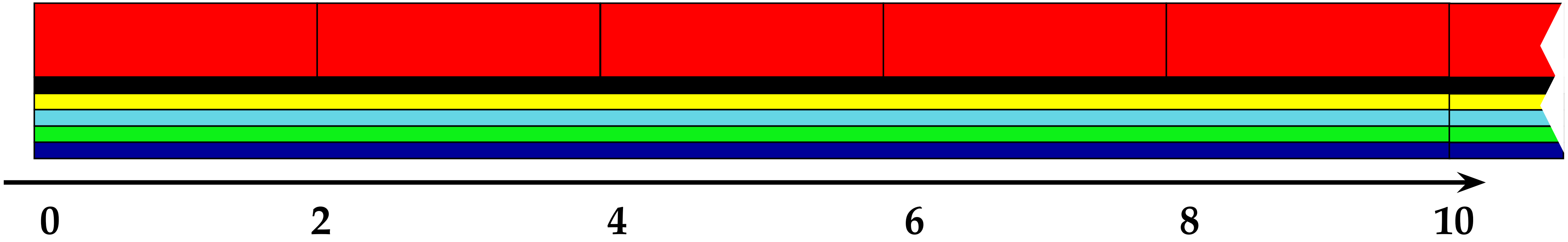
- Fluid system service order
- Weighted Fair Queueing



Recall: Why is this bad?

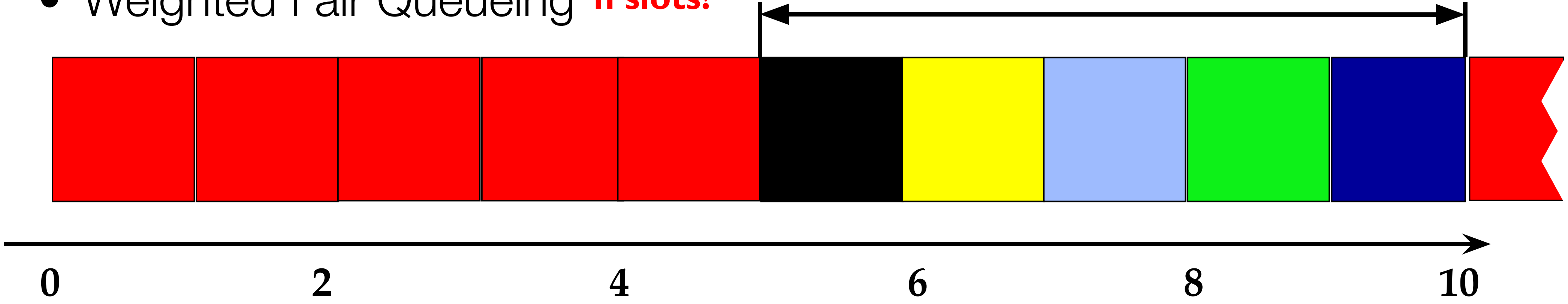
Lag: difference between service received in real system vs fluid flow system

- Fluid system service order



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

- Weighted Fair Queueing



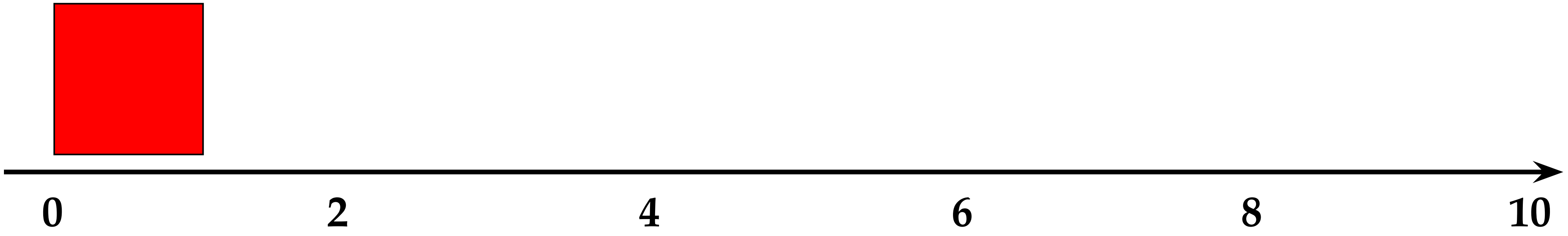
Recall: EEVDF

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



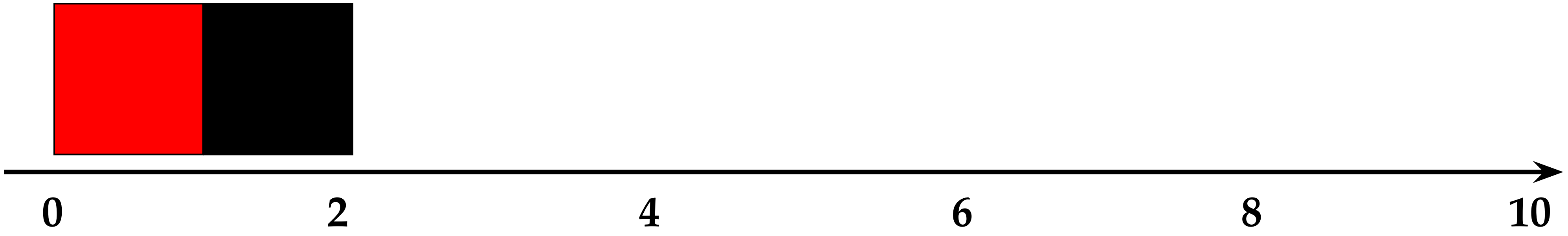
Recall: EEVDF

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)

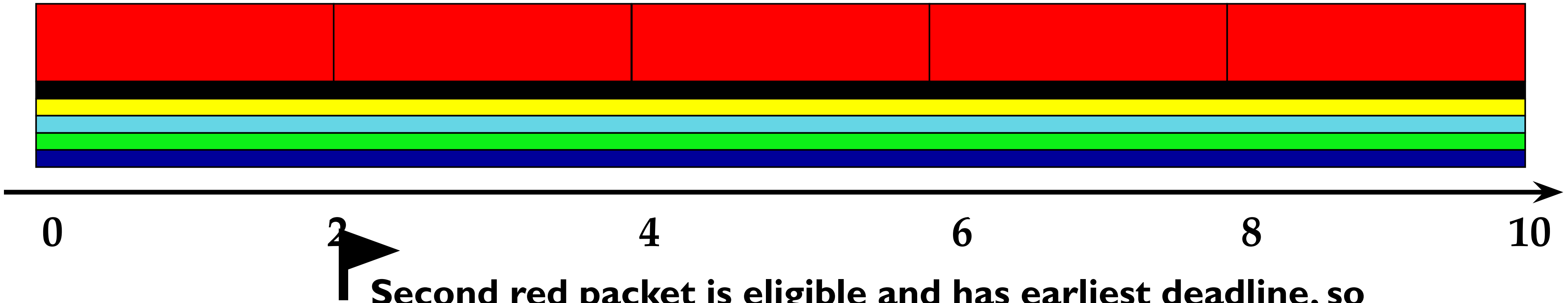
- Weighted Fair Queueing



Recall: EEVDF

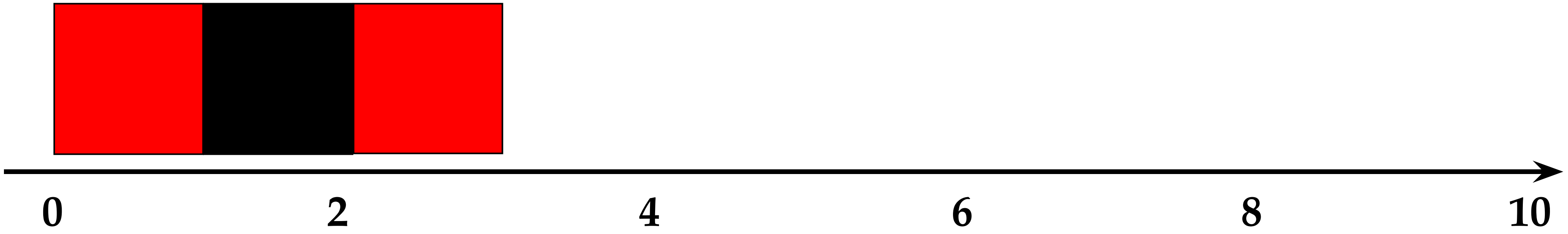
Schedule only the processes/packets that are eligible in fluid flow system, i.e., packets with virtual arrival time greater than virtual time

- Fluid system service order



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

- Weighted Fair Queueing



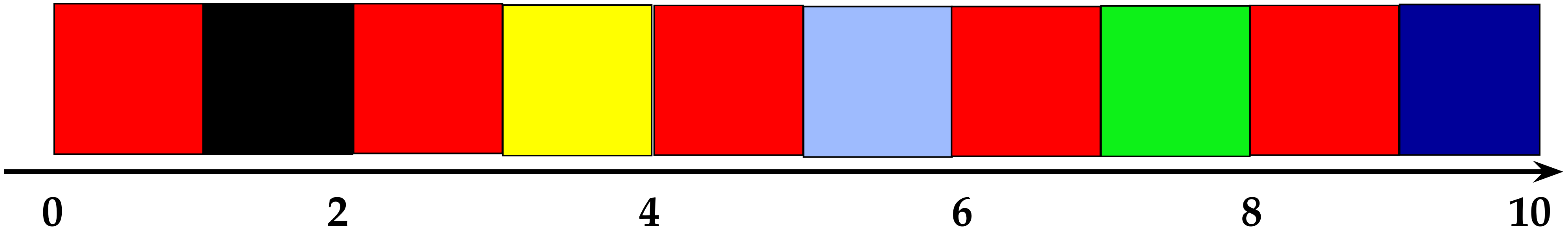
Recall: EEVDF

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



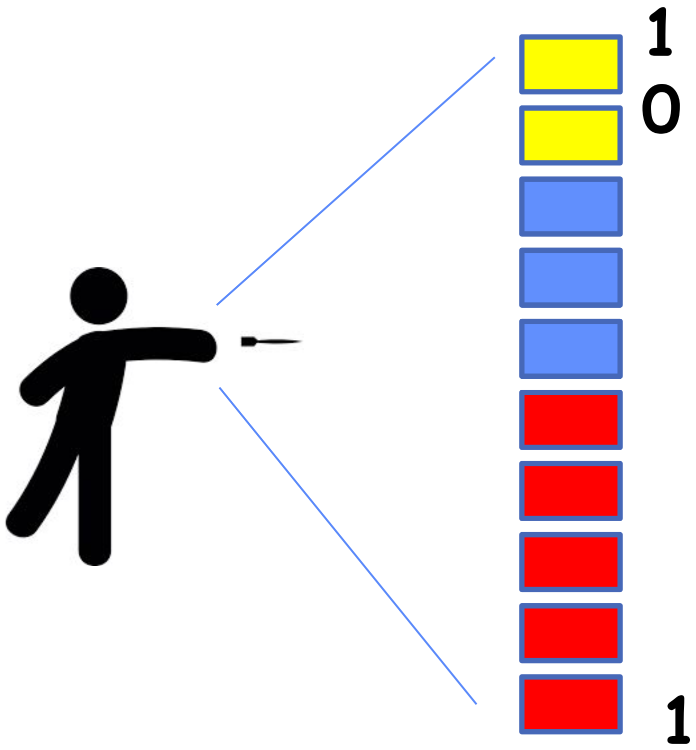
Recall: Lottery Scheduling

An approximation of weighted fair sharing

– Weight □ number of tickets

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

Recall: Lottery Scheduling: Simple Mechanism



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

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, …

Comparison of proportional share disciplines

- n: number of flows / processes / users
- q: length of max time quanta

	Missing the deadline in fluid flow system	Service lag to fluid flow systems	Scheduling complexity	Comments
Fair sharing / queueing	q	O(n)*q	O(log(n))	- For networking q is max packet size - Data structure: priority queue ordered by finishing virtual time
EEVDF	q	O(1)*q	O(log(n))	More complex data structure
Lottery scheduling	O(\sqrt(n))*q	O(\sqrt(n))*q	O(1)	
Stride scheduling	O(log(n))*q	O(log(n))*q	O(log(n))	
Round robin	q	O(n)*q	O(n)	- If only one flow has packets to send, it takes O(n) to check every empty queue - Hard to handle diff. size time quanta

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: one set of queues/core with background rebalancing

Spinlocks for multiprocessing

- 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!)

Spinlocks for multiprocessing (cont'd)

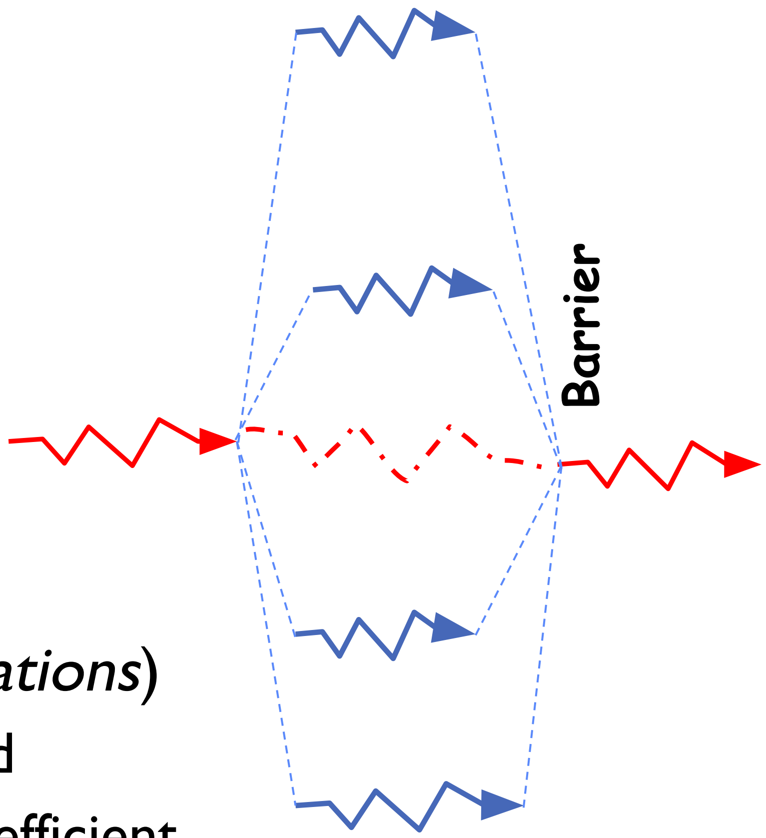
- Want `test&test&set()` !
  - First test just reads the lock; no memory contention!
  - Only if free do atomic `test&set()`

- 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
}
```

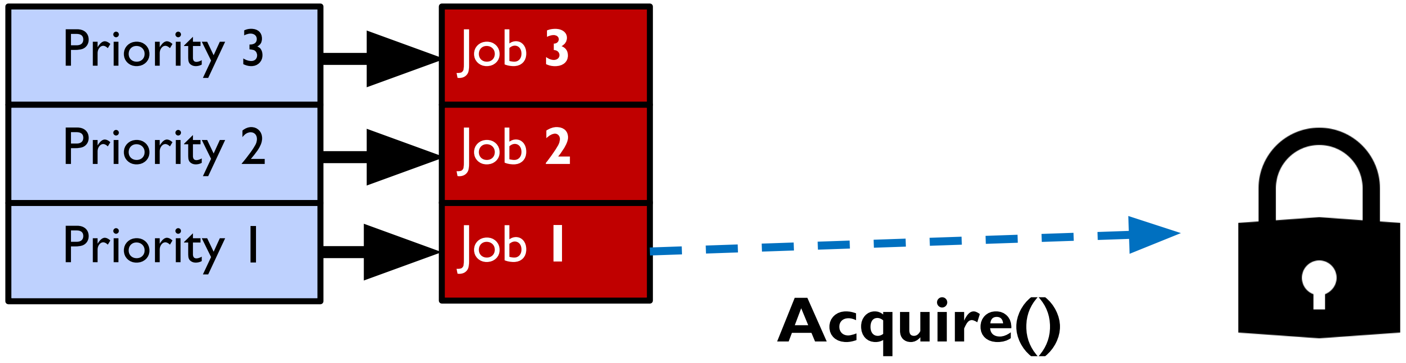
Gang Scheduling and Parallel Applications

- 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



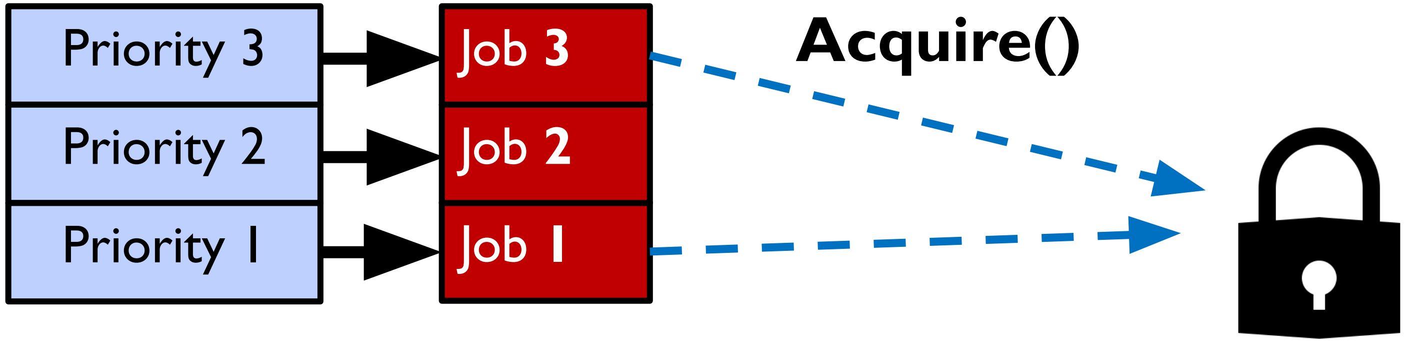
- Additionally: OS informs a parallel program how many processors its threads are scheduled on (*Scheduler Activations*)
  - Application adapts to number of cores that it has scheduled
  - "Space sharing" with other parallel programs can be more efficient, because parallel speedup is often sublinear with the number of cores

Priority Inversion



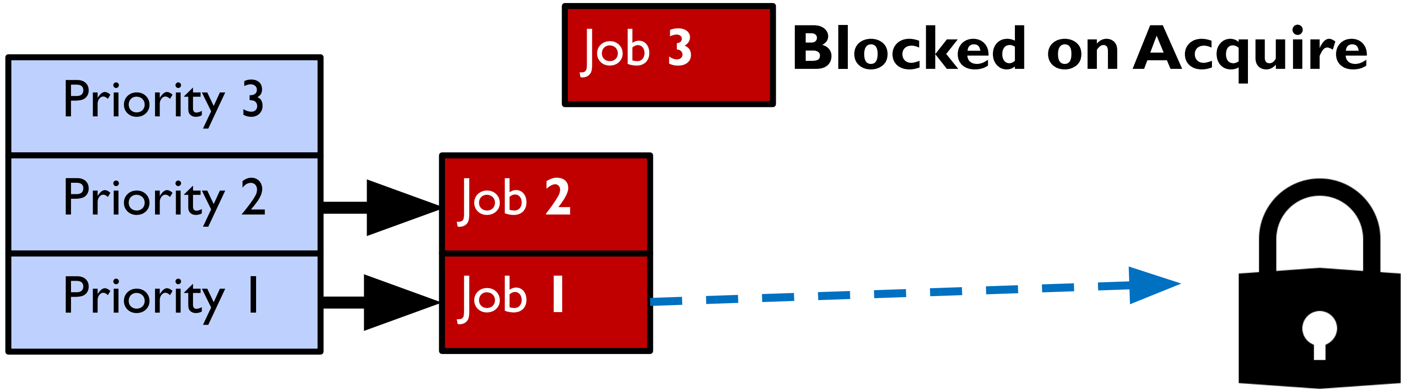
- **At this point, which job does the scheduler choose?**
- Job 3 (Highest priority)

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"?

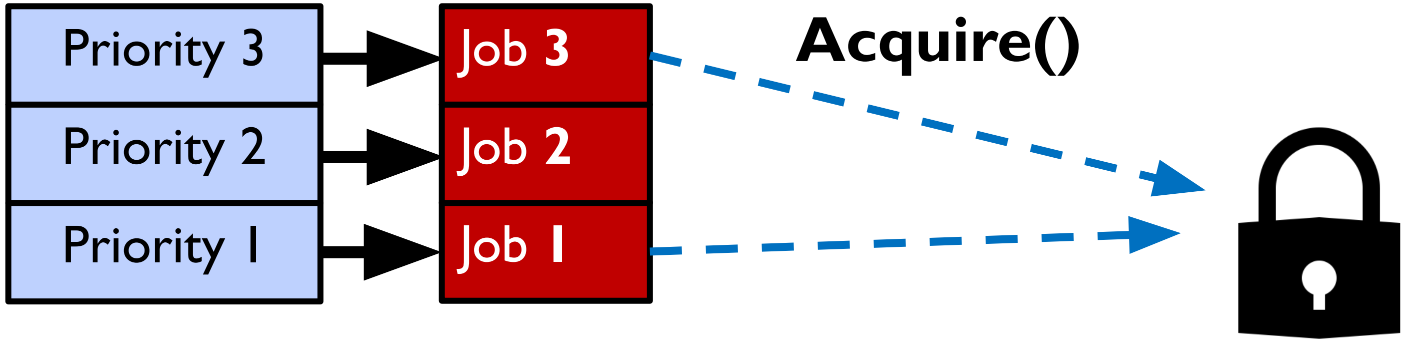
High Priority

```
while (try_lock) {
…
}
```

Low Priority

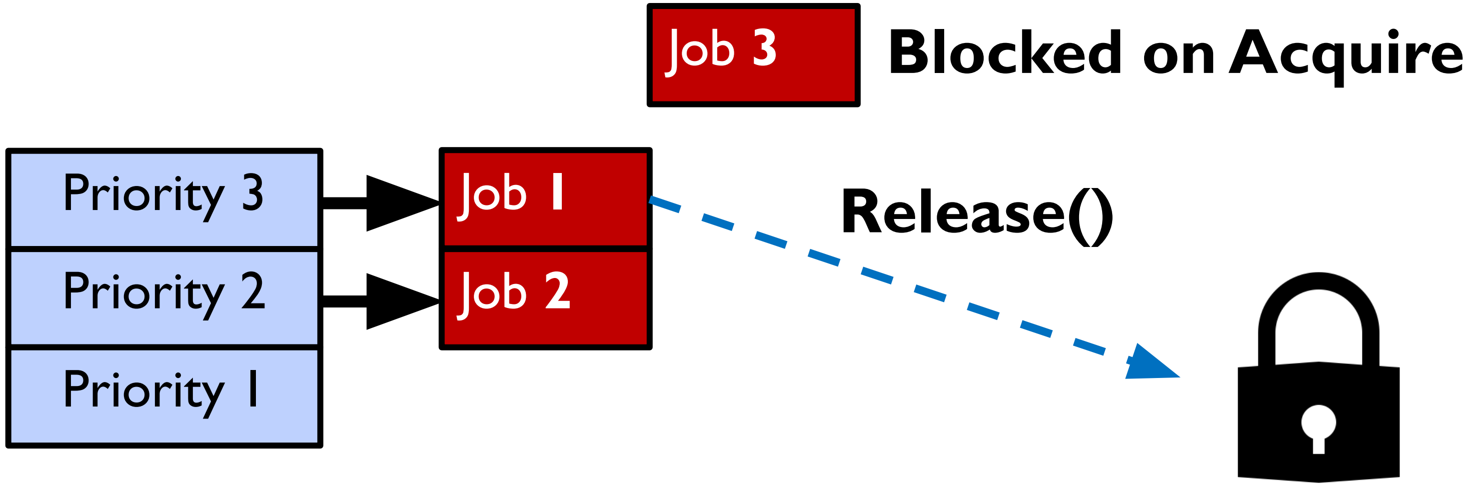
```
lock.acquire(…)
…
lock.release(…)
```

One Solution: Priority Donation/Inheritance



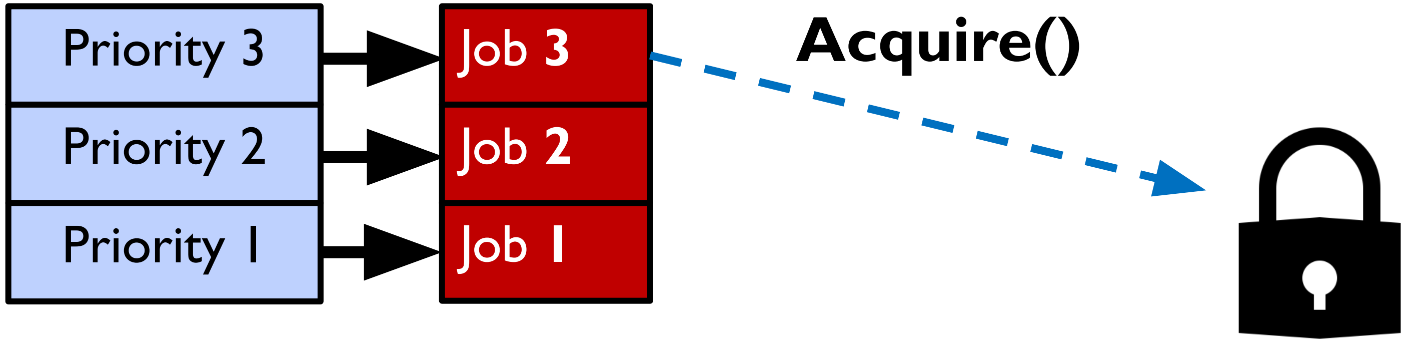
- Job 3 temporarily grants Job 1 its "high priority" to run on its behalf

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

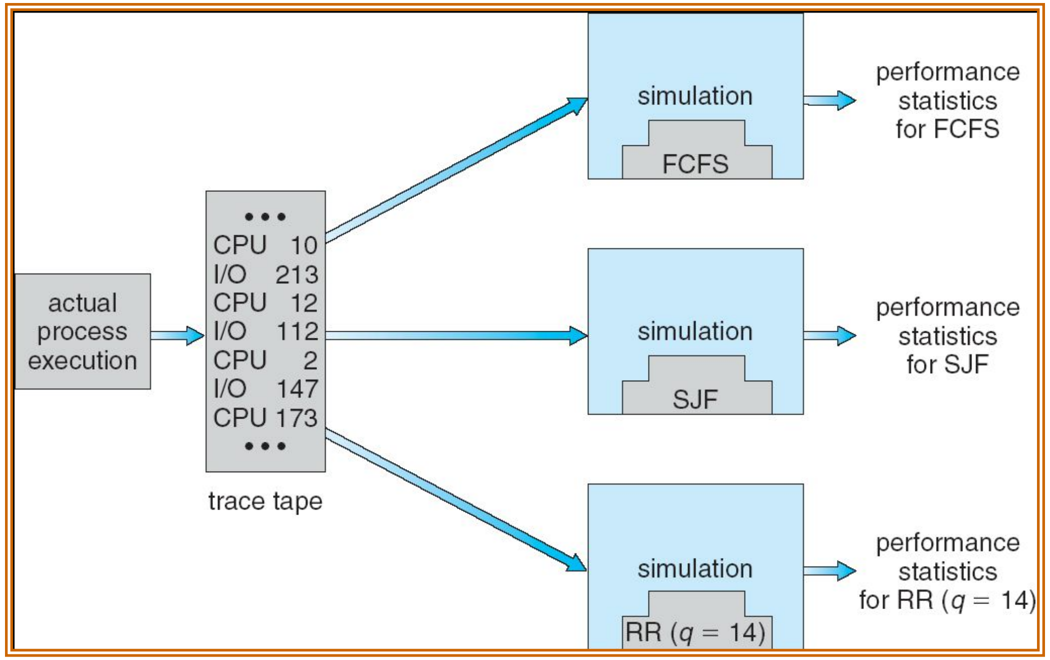
- July 4, 1997 – Pathfinder lands on Mars
  - First US Mars landing since Vikings in 1976; first rover
  - Novel delivery mechanism: inside air-filled balloons bounced to stop on the surface from orbit!
- And then…a few days into mission…:
  - Multiple system resets occur to realtime OS (VxWorks)
  - 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!

How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data
  - Most flexible/general

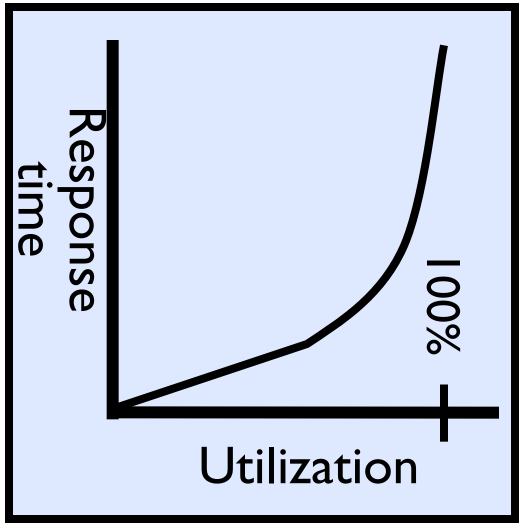


Choosing the Right Scheduler

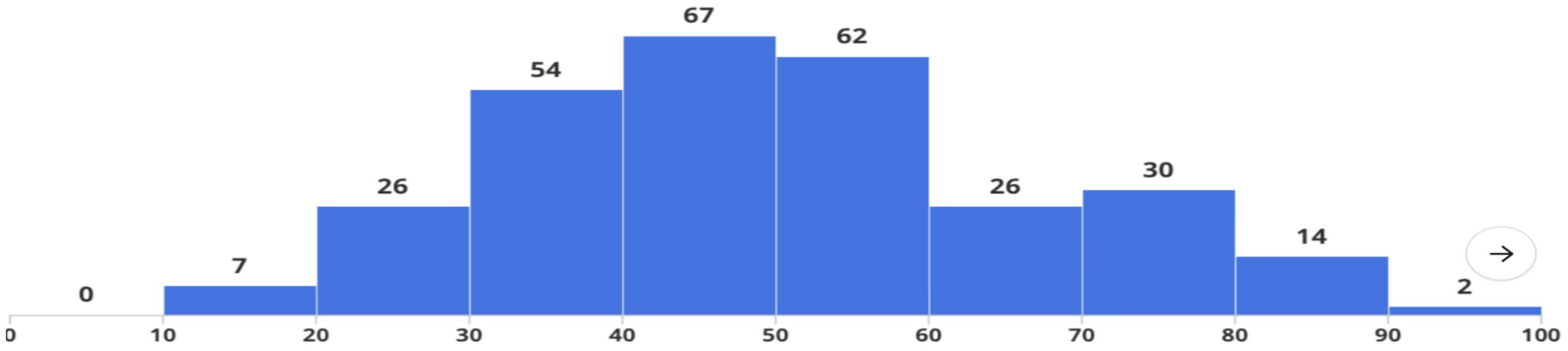
I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Round Robin, Fair Sharing, Lottery, EEVDF, …
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - » Perhaps you're paying for worse response time in reduced productivity, customer angst, etc…
    - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization $\Rightarrow$ 100%
- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve



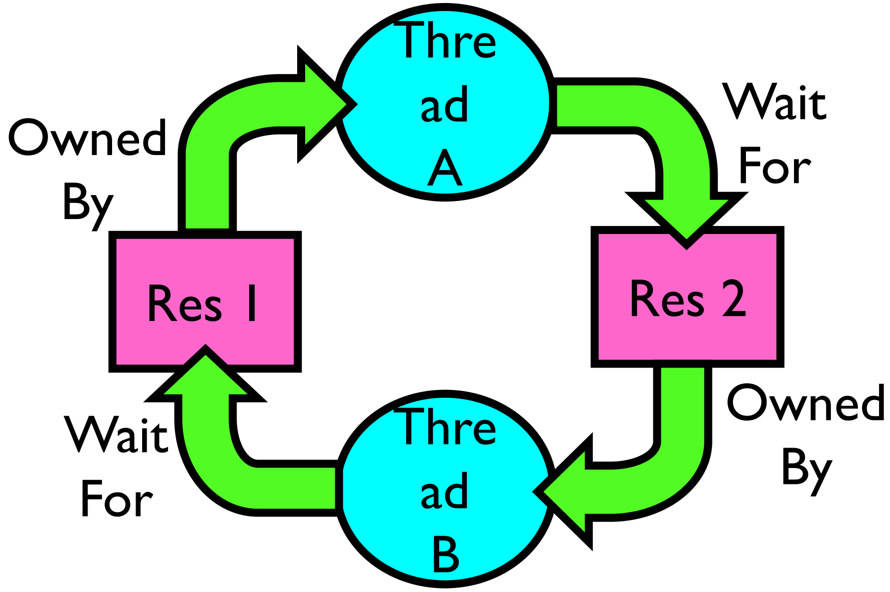
Administrivia



- Midterm 1 results: Mean: 50.1, StdDev: 16.9, Min: 12.9, Max: 92.5
- Ion out on Thursday (10/17); Professor Natacha Crooks will teach the lecture
- Welcome to Project 2
  - Please get started earlier than last time!
- Midterm 2
  - Coming up in 3 weeks! (11/5)
  - Everything up to the midterm is fair game (perhaps deemphasizing the lecture on the day before….)

Deadlock: A Deadly type of Starvation

- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular 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 ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

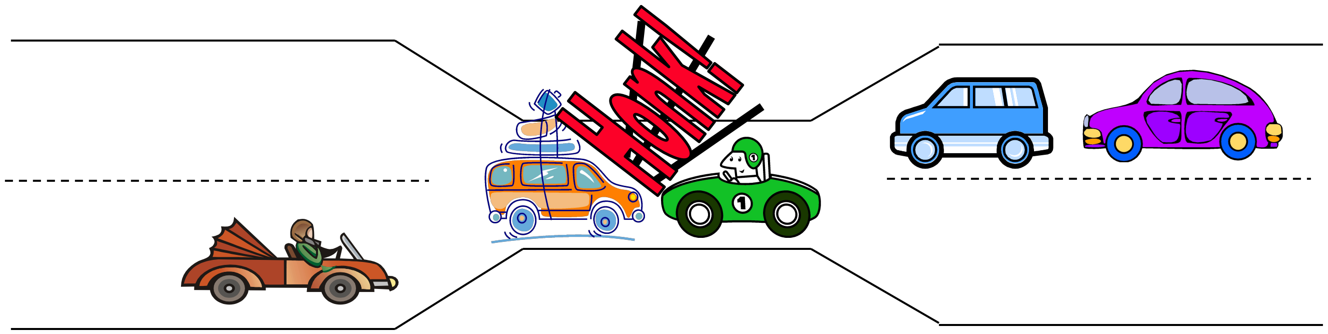


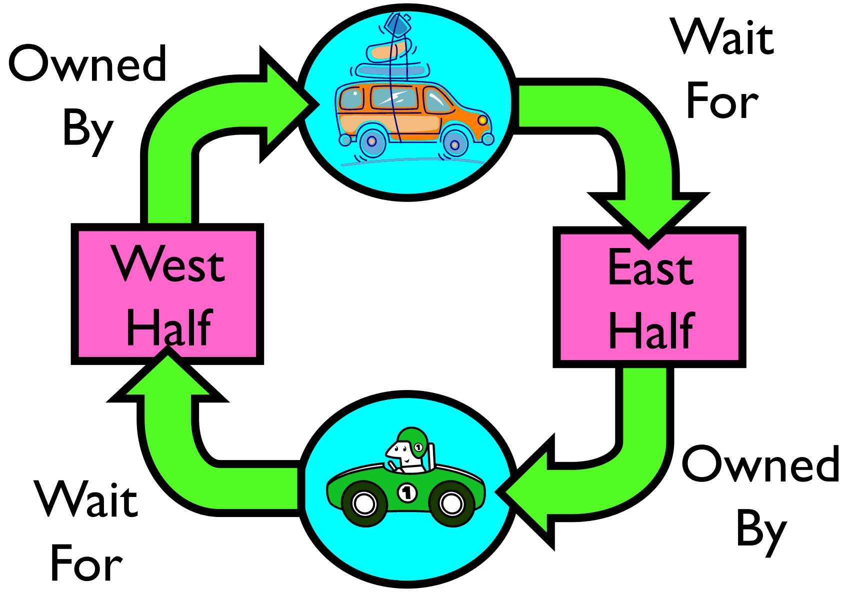
Example: Single-Lane Bridge Crossing

CA 140 to Yosemite National Park

Bridge Crossing Example

- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time

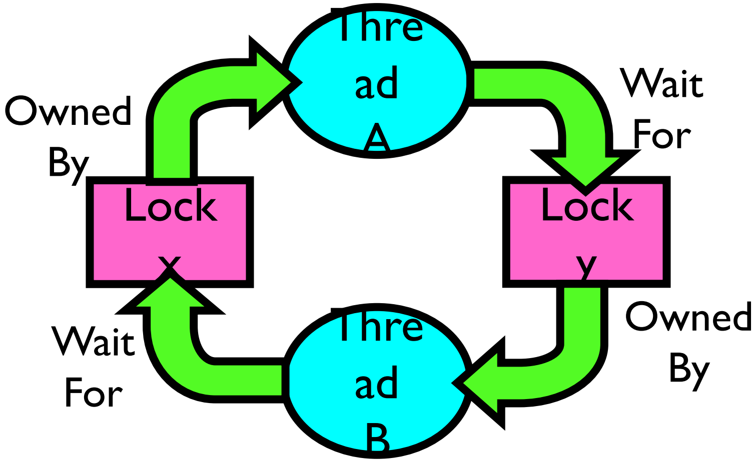




- Deadlock: Shown above when two cars in opposite directions meet in middle
  - Each acquires one segment and needs next
  - Deadlock resolved if one car backs up (preempt resources and rollback)
    - » Several cars may have to be backed up
- Starvation (not Deadlock):
  - East-going traffic really fast ⇒ no one gets to go west

Deadlock with Locks

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



- This lock pattern exhibits *non-deterministic deadlock*
  - Sometimes it happens, sometimes it doesn't!
- This is really hard to debug!

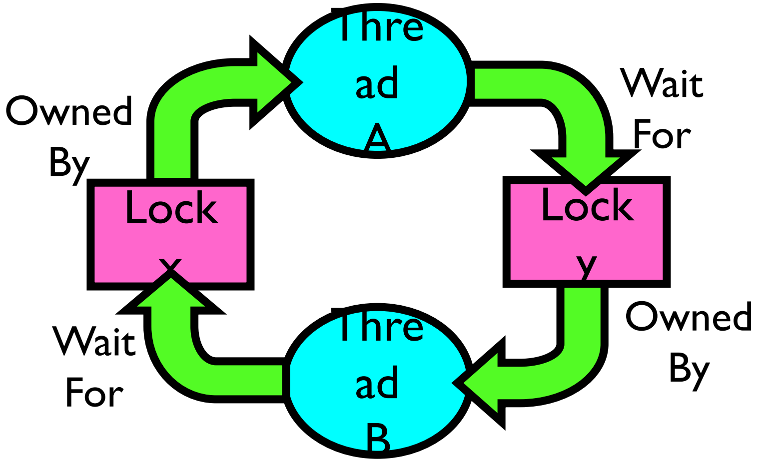
Deadlock with Locks: "Unlucky" Case

Thread A:

```
x.Acquire();

y.Acquire(); <stalled>
<unreachable>
…
y.Release();
x.Release();
```

Thread B:

```

y.Acquire();

x.Acquire(); <stalled>
<unreachable>
…
x.Release();
y.Release();
```



Neither thread will get to run ⇒ Deadlock

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 often block waiting for resources
  - Locks
  - Terminals
  - Printers
  - CD drives
  - Memory

- Threads often block waiting for other threads
  - Pipes
  - Sockets

- You can deadlock on any of these!

Deadlock with Space

Thread A:

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

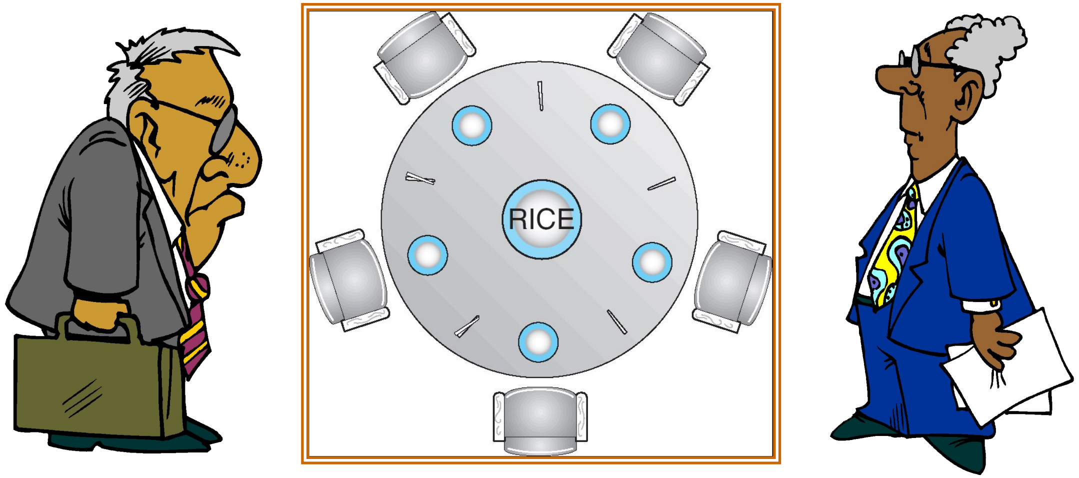
Thread B

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

If only 2 MB of space, we get same deadlock situation

Dining Lawyers Problem

- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat



- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards
  - Can we formalize this requirement somehow?

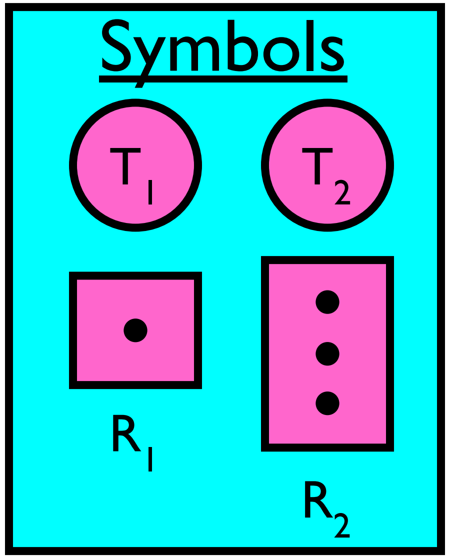
Four requirements for occurrence of Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- 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 $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

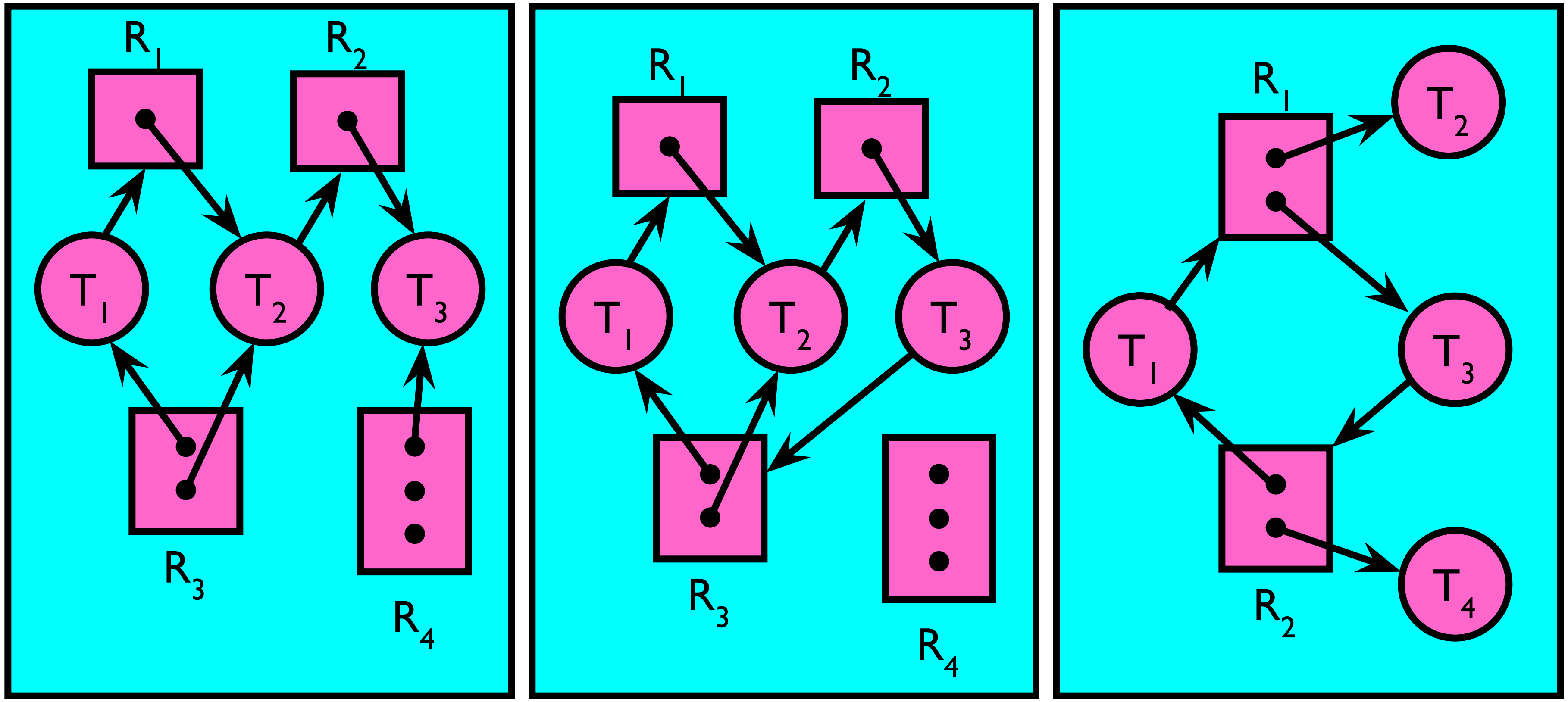
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`



- Resource-Allocation Graph:
  - V is partitioned into 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

- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock

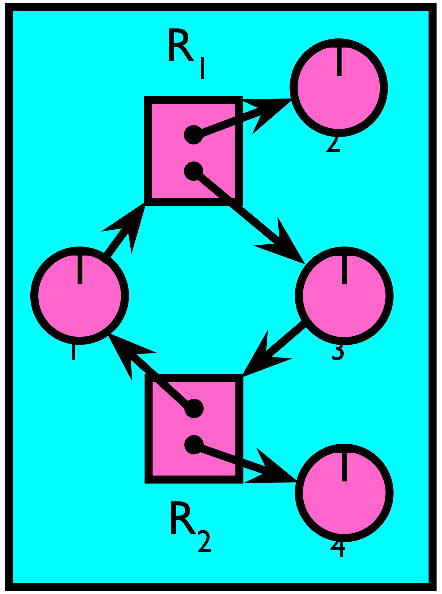
Deadlock Detection Algorithm

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

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

- See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_node] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
        }
    }
} until(done)
```



- Nodes left in UNFINISHED ⇒ deadlocked

How should a system deal with deadlock?

- Four different approaches:

1. Deadlock prevention: write your code in a way that it isn't prone to deadlock
2. Deadlock recovery: let deadlock happen, and then figure out how to recover from it
3. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
4. Deadlock denial: ignore the possibility of deadlock

- Modern operating systems:
  – Make sure the *system* isn't involved in any deadlock
  – Ignore deadlock in applications
    » "Ostrich Algorithm"

Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't actually have to be infinite, just large…
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call Mom in Toledo, works way through phone network, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    - » Consider: driving to San Francisco; when hit traffic jam, suddenly you're transported back home and told to retry!

(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 just succeed, thus above example won't deadlock
  - Of course, it isn't actually infinite, but certainly larger than 2MB!

Techniques for Preventing Deadlock

- Make all threads request everything they'll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (`x.Acquire()`, `y.Acquire()`, `z.Acquire()`,…)
    - » Make tasks request disk, then memory, then…
    - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Request Resources Atomically (1)

Rather than:

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

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

Consider instead:

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

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

Request Resources Atomically (2)

Or consider this:

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

Acquire Resources in Consistent Order

Rather than:

Thread A:

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

Thread B:

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

Consider instead:

Thread A:

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

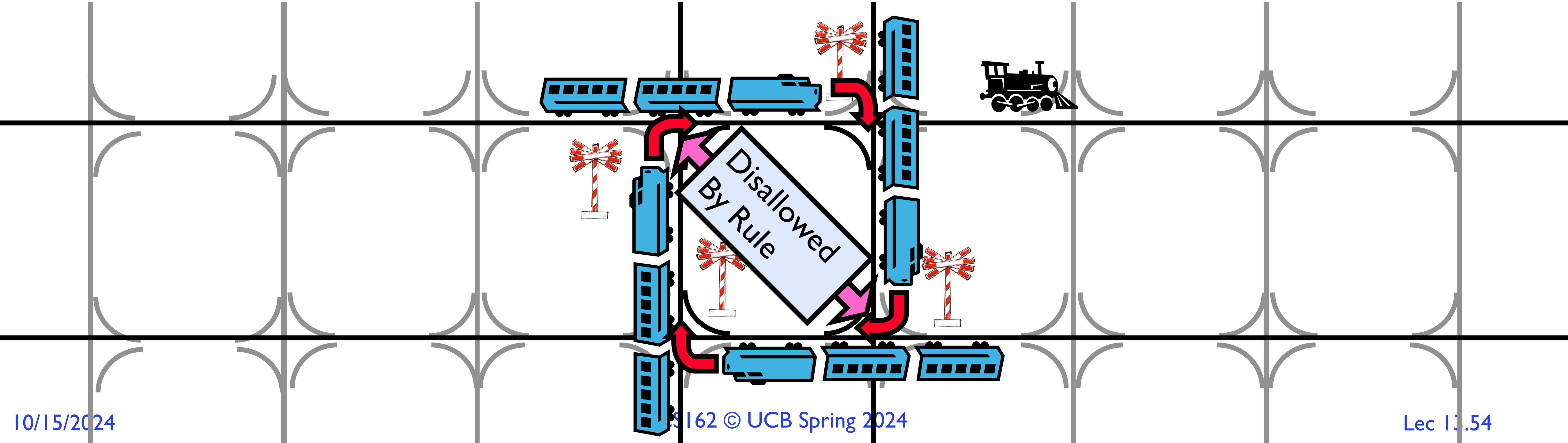
Thread B:

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

Does it matter in which order the locks are released?

Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right, but is blocked by other trains
- Similar problem to multiprocessor networks
  - Wormhole-Routed Network: Messages trail through network like a "worm"
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Techniques for Recovering from Deadlock

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Hold dining lawyer in contempt and take away in handcuffs
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

Another view of virtual memory: Pre-empting 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)`

- Before: With virtual memory we have "infinite" space so everything will just succeed, thus above example won't deadlock
  - Of course, it isn't actually infinite, but certainly larger than 2MB!
- Alternative view: we are "pre-empting" memory when paging out to disk, and giving it back when paging back in
  - This works because thread can't use memory when paged out

Techniques for Deadlock Avoidance

- Idea: 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

THIS DOES NOT WORK!!!!

- Example:

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

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

- Deadlocked state
  - There exists a deadlock in the system
  - Also considered "unsafe"

Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in ~~deadlock~~ an unsafe state
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

- Example:

<u>Thread A</u>:

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

<u>Thread B</u>:

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

Wait until Thread A releases mutex X

Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

      $([\text{Max}_{\text{node}}]-[\text{Alloc}_{\text{node}}] \text{ <= } [\text{Avail}])$ for $([\text{Request}_{\text{node}}] \text{ <= } [\text{Avail}])$

      Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
   Add all nodes to UNFINISHED
   do {
      done = true
      Foreach node in UNFINISHED {
         if ([Request_node] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
         }
      }
   } until(done)
```

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

» Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
  ([Max$_{node}$]-[Alloc$_{node}$] <= [Avail]) for ([Request$_{node}$] <= [Avail])
  Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
   Add all nodes to UNFINISHED
   do {
      done = true
      Foreach node in UNFINISHED {
         if ([Max_node]-[Alloc_node] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
         }
      }
   } until(done)
```

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

» Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
   ([Max$_{node}$]-[Alloc$_{node}$] <= [Avail]) for ([Request$_{node}$] <= [Avail])
   Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

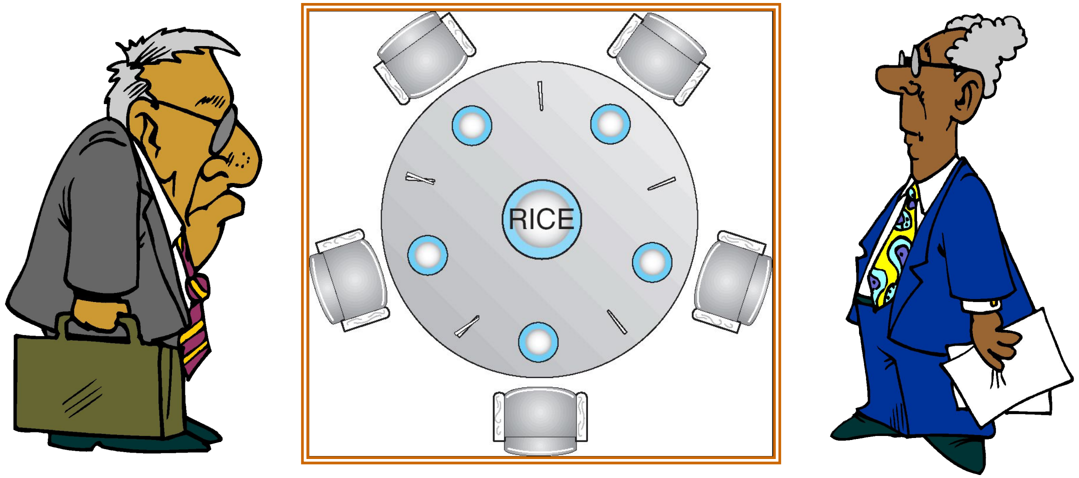
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

      ([Max_{node}]-[$\text{Alloc}_{\text{node}}$] <= [Avail]) for ([$\text{Request}_{\text{node}}$] <= [Avail])

      Grant request if result is deadlock free (conservative!)
  - Keeps system in a "SAFE" state: there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..

Banker's Algorithm Example

- Banker's algorithm with dining lawyers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards



  - What if k-handed lawyers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

Conclusion

- Proportional Share Scheduling (Fair Sharing, EEVDF, Lottery 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)
- Four conditions for deadlocks
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Techniques for addressing Deadlock
  - <u>Deadlock prevention</u>:
    - » write your code in a way that it isn't prone to deadlock
  - <u>Deadlock recovery</u>:
    - » let deadlock happen, and then figure out how to recover from it
  - <u>Deadlock avoidance</u>:
    - » dynamically delay resource requests so deadlock doesn't happen
    - » Banker's Algorithm provides on algorithmic way to do this
  - <u>Deadlock denial</u>:
    - » ignore the possibility of deadlock