Operating Systems
(Honor Track)

Scheduling 3: Scheduling & Deadlock

Xin Jin

Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: Ensuring Progress

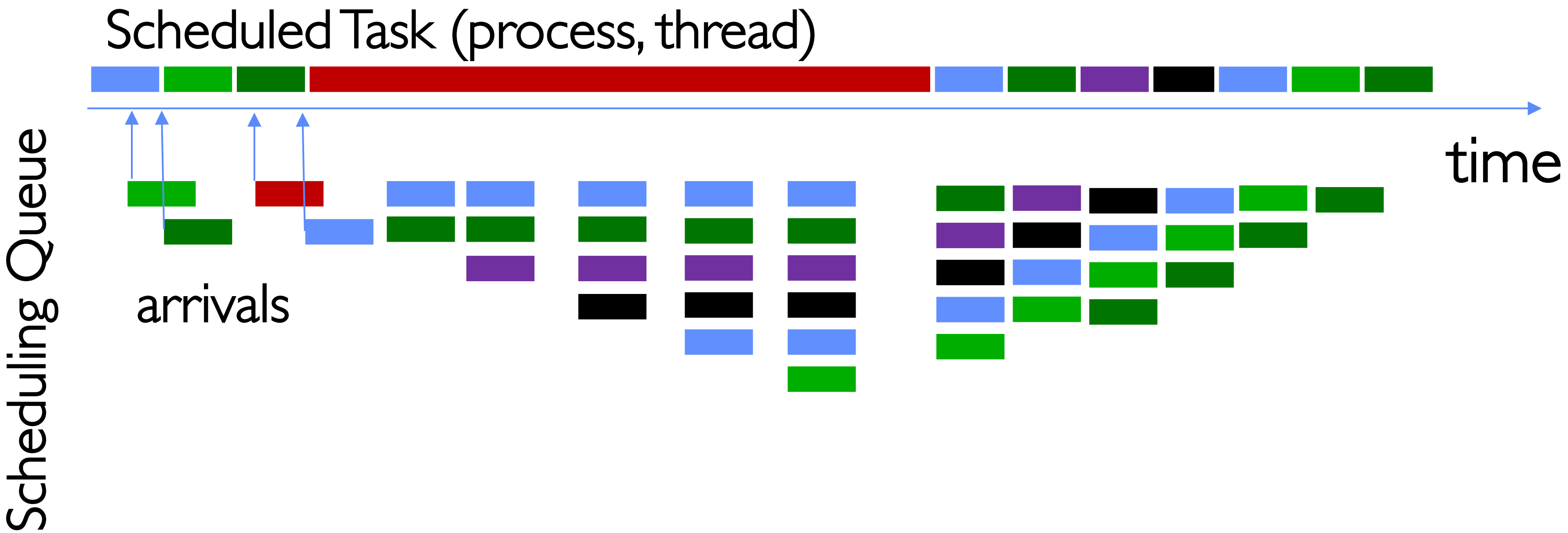
- Starvation: thread fails to make progress for an indefinite period of time

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

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

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

Recap: Is FCFS Prone to Starvation?



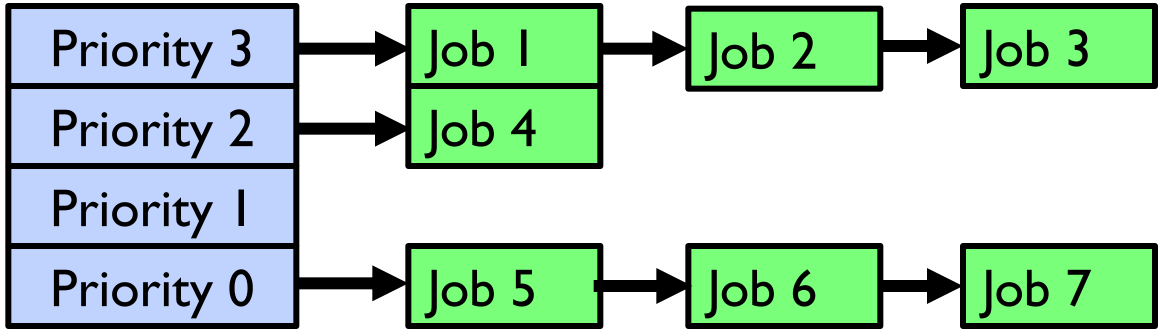
- If a task never yields (e.g., goes into an infinite loop), then other tasks don't get to run
- Problem with all non-preemptive schedulers…
  - And early personal OSes such as original MacOS, Windows 3.1, etc

Recap: Is Round Robin (RR) Prone to Starvation?

- Each of *N* processes gets ~1/*N* of CPU (in window)
  - With quantum length *Q* ms, process waits at most *(N-1)*Q* ms to run again
  - So a process can't be kept waiting indefinitely

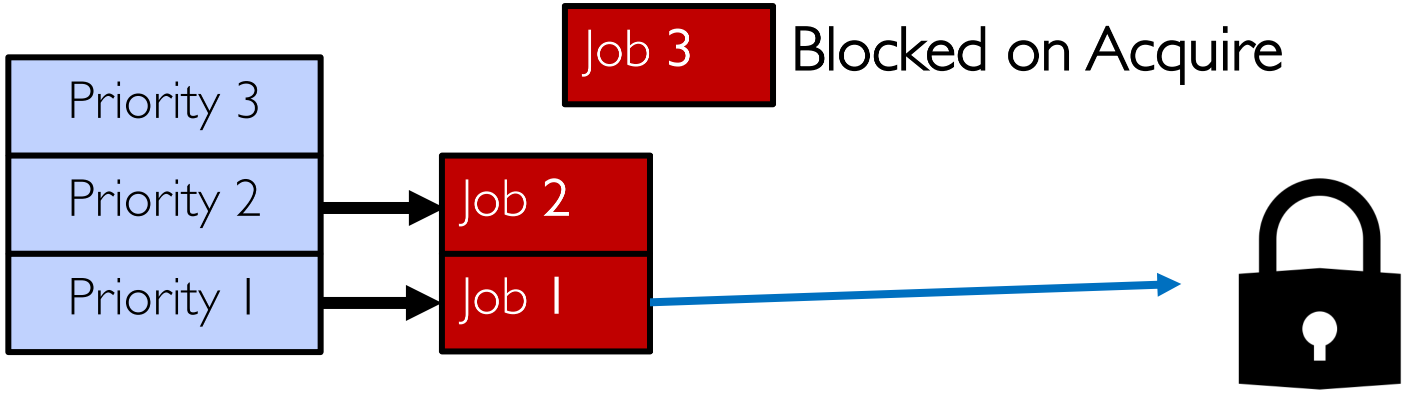
- So RR is fair in terms of *waiting time*
  - Not necessarily in terms of throughput…

Recap: Is Priority Scheduling Prone to Starvation?



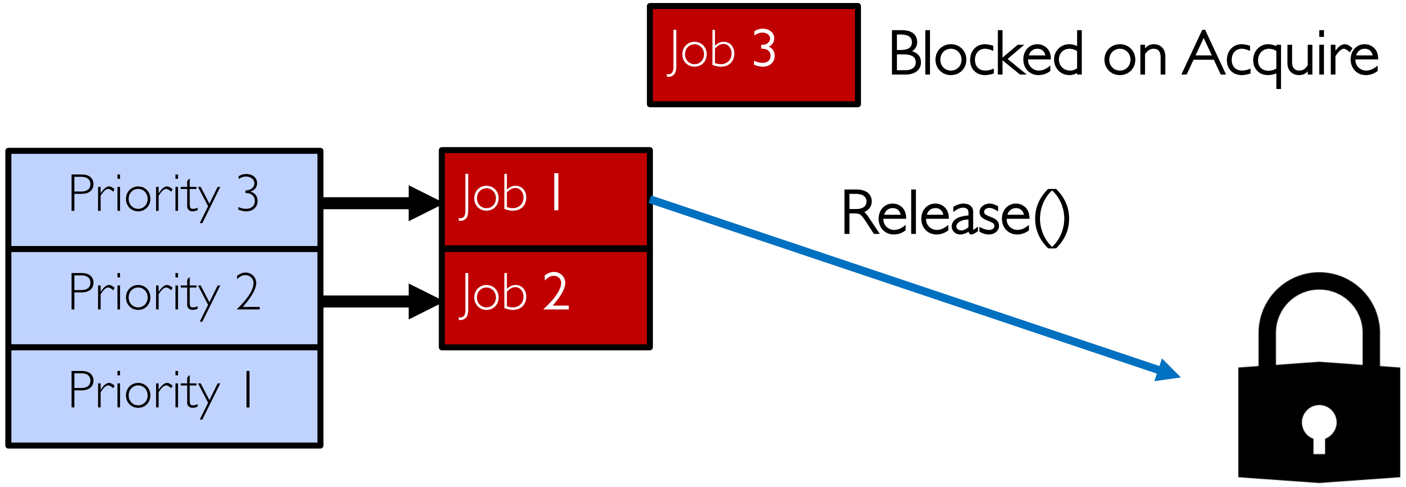
- Recall: Priority Scheduler always runs the thread with highest priority
  - Low priority thread might never run!
  - Starvation…
- But there are more serious problems as well…
  - Priority inversion: even high priority threads might become starved

Recap: Priority Inversion



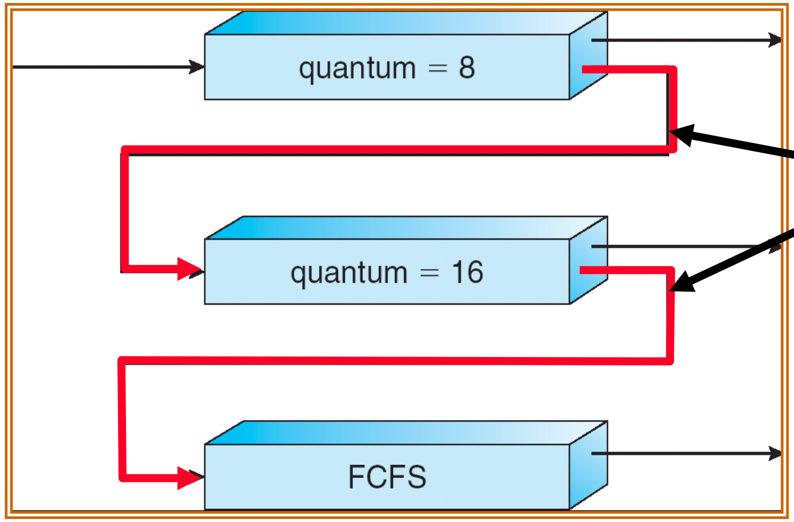
- At this point, which job does the scheduler choose?
- Job 2 (Medium Priority)
- Priority Inversion

Recap: One Solution: Priority Donation/Inheritance



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

Are SRTF and MLFQ Prone to Starvation?



Long-Running Compute Tasks Demoted to Low Priority

- In SRTF, long jobs are starved in favor of short ones
  - Same fundamental problem as priority scheduling
- MLFQ is an approximation of SRTF, so it suffers from the same problem

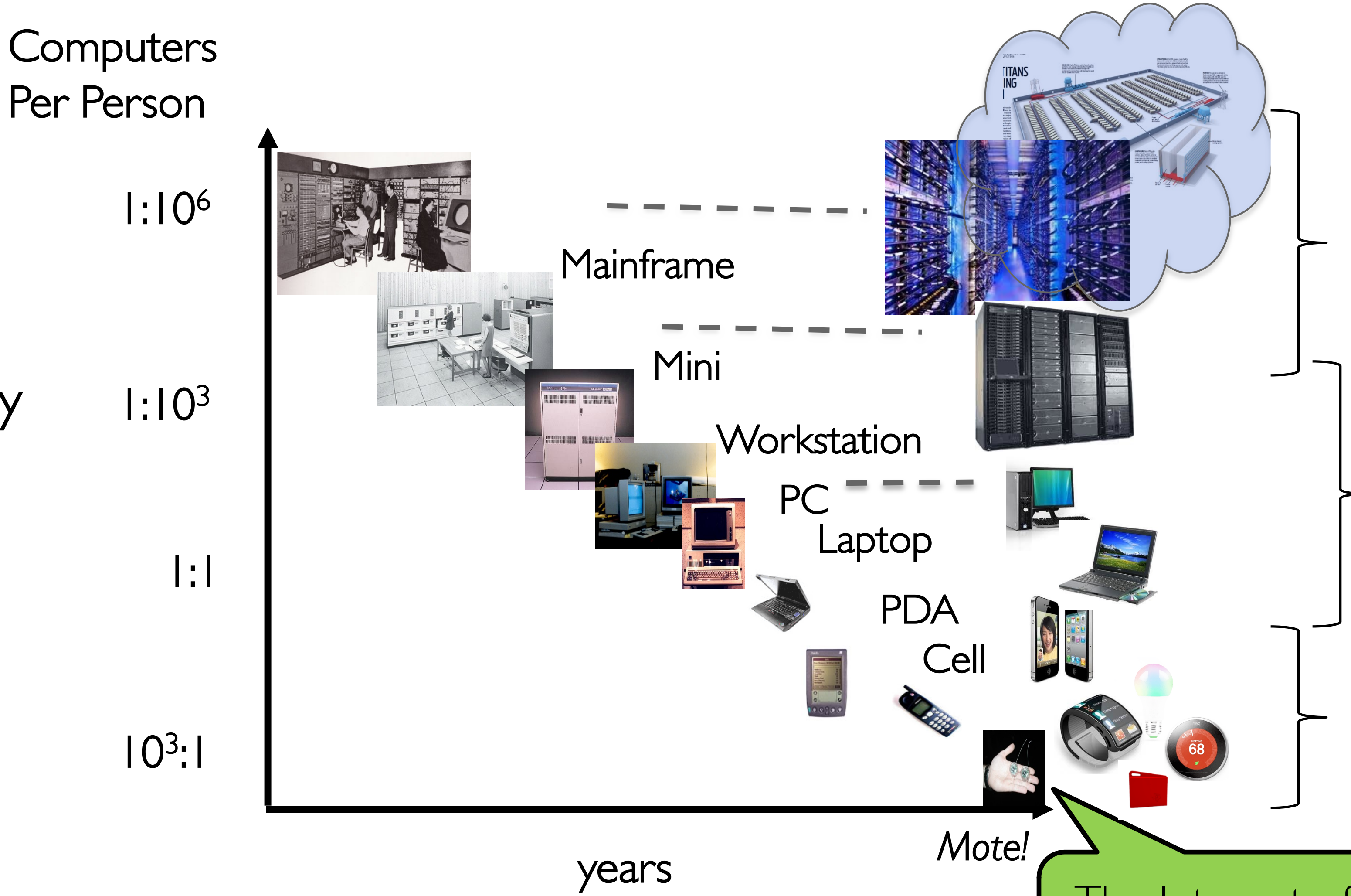
Cause for Starvation: Priorities?

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

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

Recall: Changing Landscape…

Bell's Law: New computer class every 10 years



Number crunching, Data Storage, Massive Inet Services, ML, …

Productivity, Interactive

Streaming from/to the physical world

The Internet of Things!

Changing Landscape of Scheduling

- Priority-based scheduling rooted in “time-sharing”
  - Allocating precious, limited resources across a diverse workload
    - CPU bound vs. interactive vs. I/O bound
- 80’s brought about personal computers, workstations, and servers on networks
  - Different machines of different types for different purposes
  - Shift to fairness and avoiding extremes (starvation)
- 90’s emergence of the web, rise of internet-based services, the data-center-is-the-computer
  - Server consolidation, massive clustered services, huge flashcrowds
  - It’s about predictability, 95th percentile performance guarantees

Priority in Unix – Being Nice

- The industrial operating systems of the 60s and 70s provided priority to enforced desired usage policies.
  - When it was being developed at Berkeley, instead it provided ways to “be nice”.
- `nice` values range from -20 to 19
  - Negative values are “not nice”
  - If you wanted to let your friends get more time, you would nice up your job
- Scheduler puts higher nice-value tasks (lower priority) to sleep more …
  - In O(1) scheduler, this translated fairly directly to priority (and time slice)

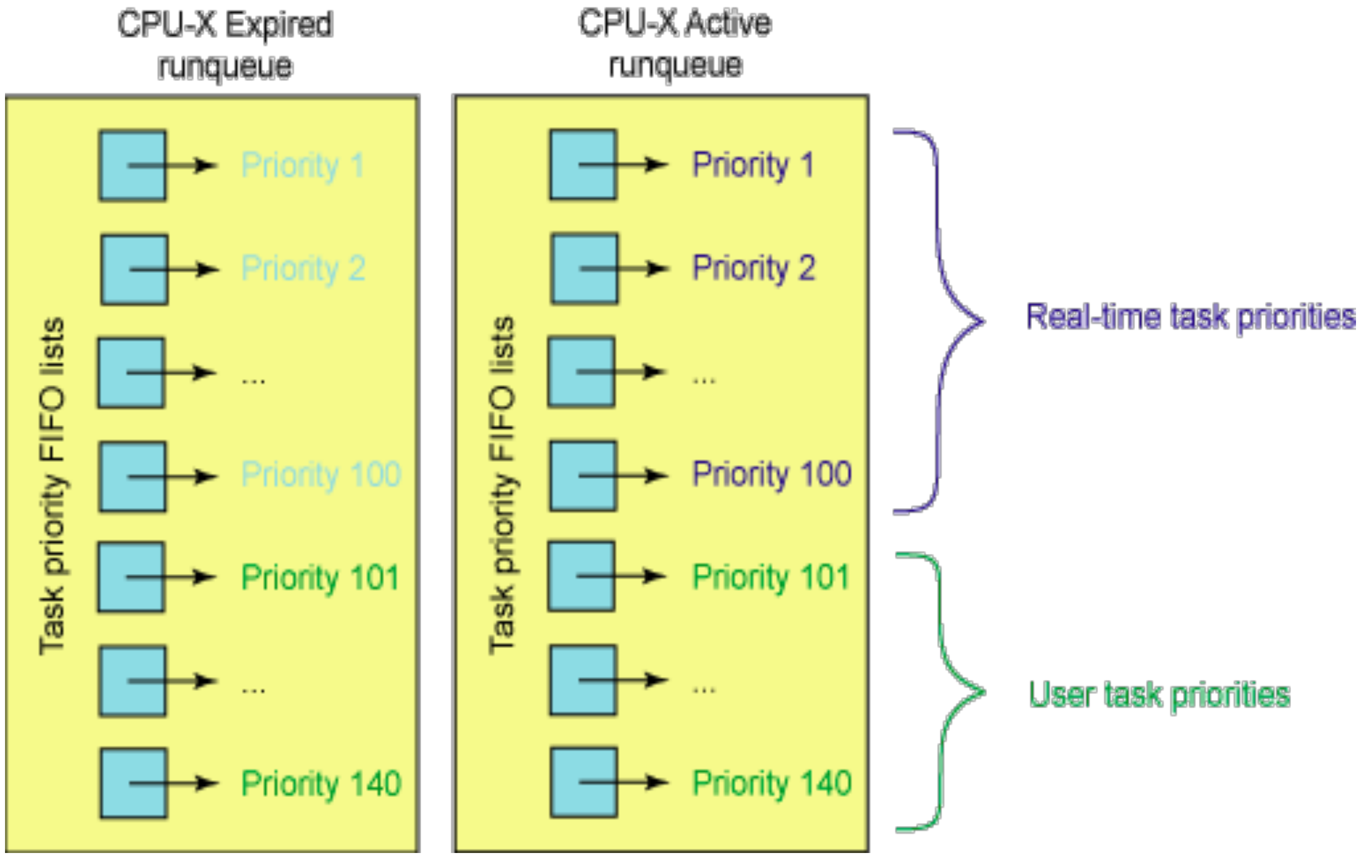
Case Study: Linux O(1) Scheduler

Kernel/Realtime Tasks | User Tasks

0 100 139

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower nice value ⇒ higher priority
  - Higher nice value ⇒ lower priority
  - All algorithms O(1)
    - Timeslices/priorities/interactivity credits all compute 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

Linux O(1) Scheduler



- Lots of ad-hoc heuristics
  - Try to boost priority of I/O-bound tasks
  - Try to boost priority of starved tasks

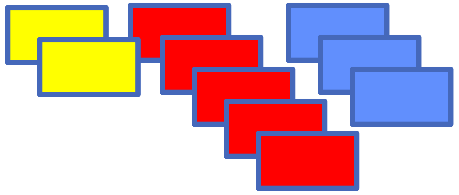
O(1) Scheduler Continued

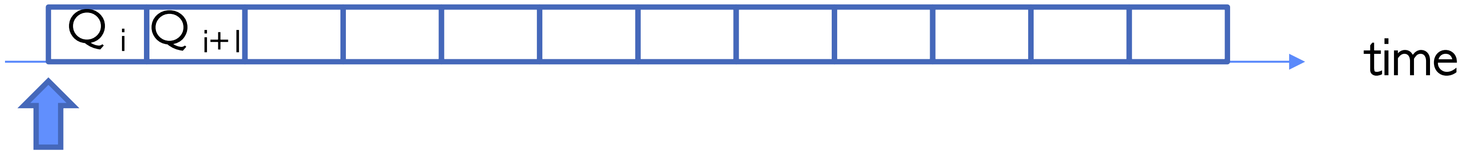
- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    - P→sleep_avg = (sleep_time – run_time) x coefficient
    - Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    - Earned when a task sleeps for a “long” time
    - Spend when a task 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

Proportional-Share Scheduling

- Instead using priorities, share the CPU *proportionally*
  - 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)

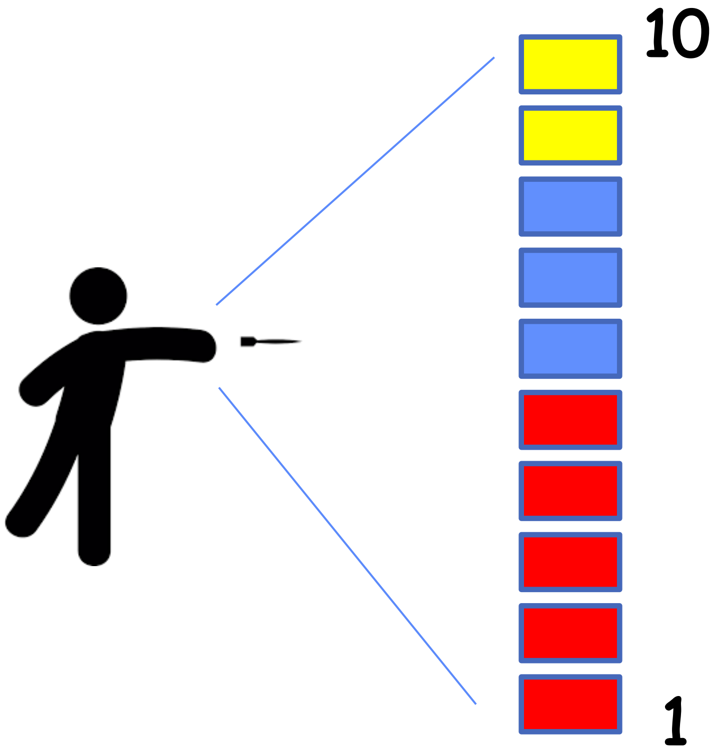
Recall: Lottery Scheduling





- Given a set of jobs (the mix), provide each with a share of a resource
  - e.g., 50% of the CPU for Job A, 30% for Job B, and 20% for Job C
- Idea: Give out tickets according to the proportion each should receive,
- Every quantum (tick): draw one at random, schedule that job (thread) to run

Lottery Scheduling: Simple Mechanism

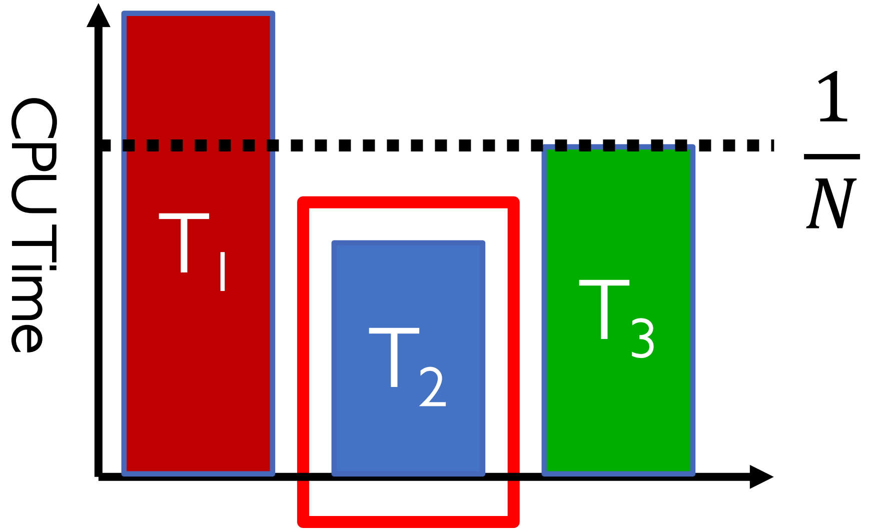


- $N_{ticket} = \sum \mathrm{N_i}$
- Pick a number d in $1\ ..\ N_{ticket}$ as the random "dart"
- Jobs record their $\mathrm{N_i}$ of allocated tickets
- Order them by $\mathrm{N_i}$
- Select the first j such that $\sum \mathrm{N_i}$ up to j exceeds *d.*

Linux Completely Fair Scheduler (CFS)

- Basic Idea: track CPU time per thread and schedule threads to match up average rate of execution
- Scheduling Decision:
  - "Repair" illusion of complete fairness
  - Choose thread with minimum CPU time
  - Closely related to Fair Queueing
- Use a heap-like scheduling queue for this…
  - O(log N) to add/remove threads, where N is number of threads
- Sleeping threads don't advance their CPU time, so they get a boost when they wake up again…
  - Get interactivity automatically!

CFS: Average rate of execution $= \frac{1}{N}$:



Linux CFS: Responsiveness/Starvation Freedom

- In addition to fairness, we want **low waiting time** and starvation freedom
  - Make sure that everyone gets to run at least a bit!
- Constraint 1: *Target Latency*
  - Period of time over which every process gets service
  - Quanta = Target_Latency / n (n: number of processes)
- Target Latency: 20 ms, 4 Processes
  - Each process gets 5ms time slice
- Target Latency: 20 ms, 200 Processes
  - Each process gets 0.1ms time slice (!!!)
  - Recall Round-Robin: large context switching overhead if slice gets to small

Linux CFS: Throughput

- Goal: Throughput
  - Avoid excessive overhead
- Constraint 2: Minimum Granularity
  - Minimum length of any time slice

- Target Latency 20 ms, Minimum Granularity 1 ms, 100 processes
  - Each process gets 1 ms time slice

Linux CFS: Proportional Shares

- What if we want to give more CPU to some and less to others in CFS (proportional share) ?
  - Allow different threads to have different *rates* of execution (cycles/time)
- Use weights: assign a weight w_i to each process i to compute the switching quanta Q_i
  - Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$
  - Weighted Share: $Q_i = \left({w_i}/{\sum_p w_p} \right) \cdot \text{Target Latency}$
- Reuse `nice` value to reflect share, rather than priority,
  - Remember that lower nice value ⇒ higher priority
  - CFS uses nice values to scale weights exponentially: Weight=1024/$(1.25)^{\text{nice}}$
    - » Two CPU tasks separated by nice value of 5 ⇒
      Task with lower nice value has 3 times the weight, since $(1.25)^5 \approx 3$

Choosing the Right Scheduler

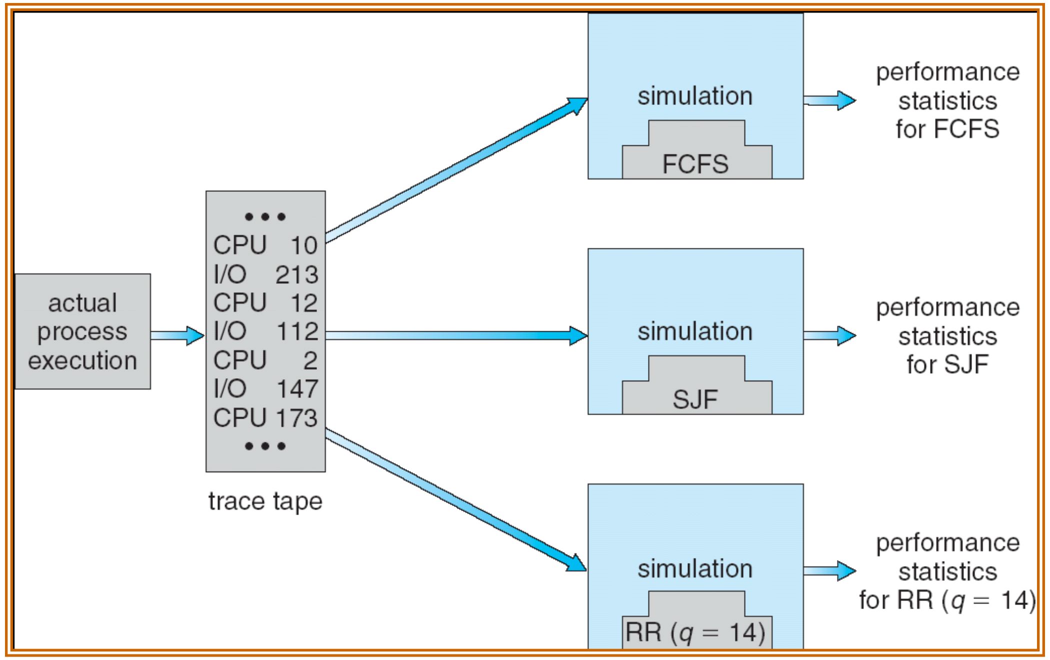
I Care About:	Then Choose:
CPU Throughput	
Avg. Completion Time	
I/O Throughput	
Fairness (CPU Time)	
Fairness (Wait Time to Get CPU)	
Meeting Deadlines	
Favoring Important Tasks	

Choosing the Right Scheduler

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Completion Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness (Wait Time to Get CPU)	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

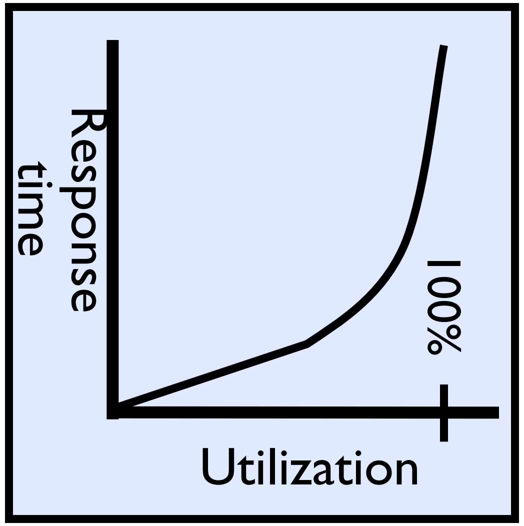
How to Evaluate a Scheduling algorithm?

- 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



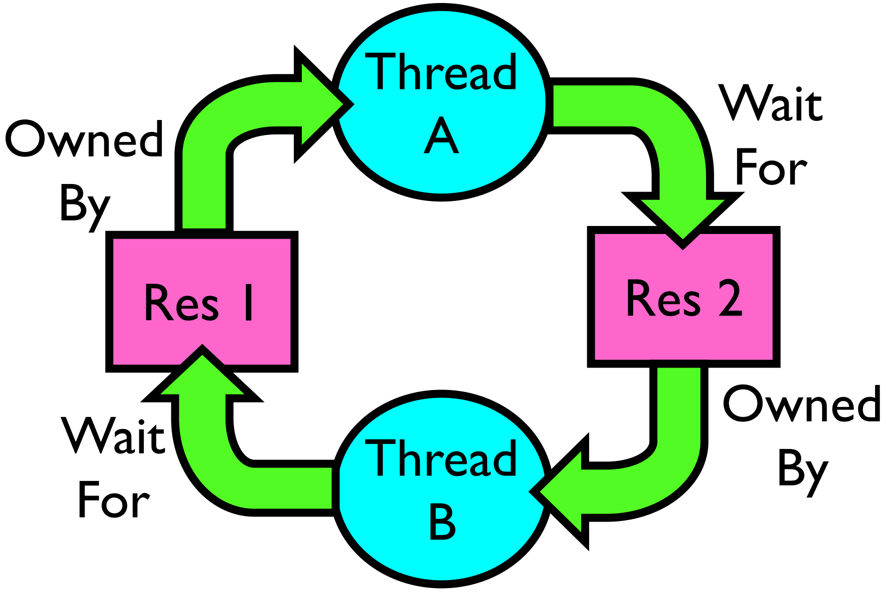
A Final Word On Scheduling

- 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⇒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



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



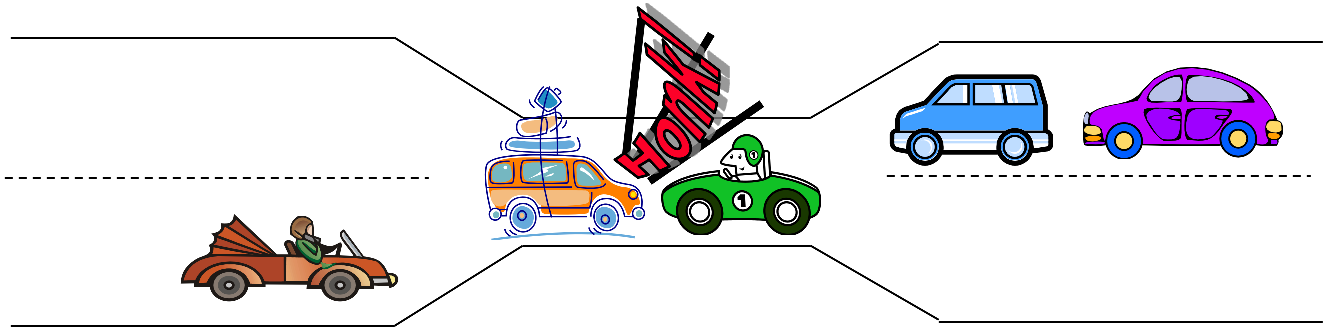
- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

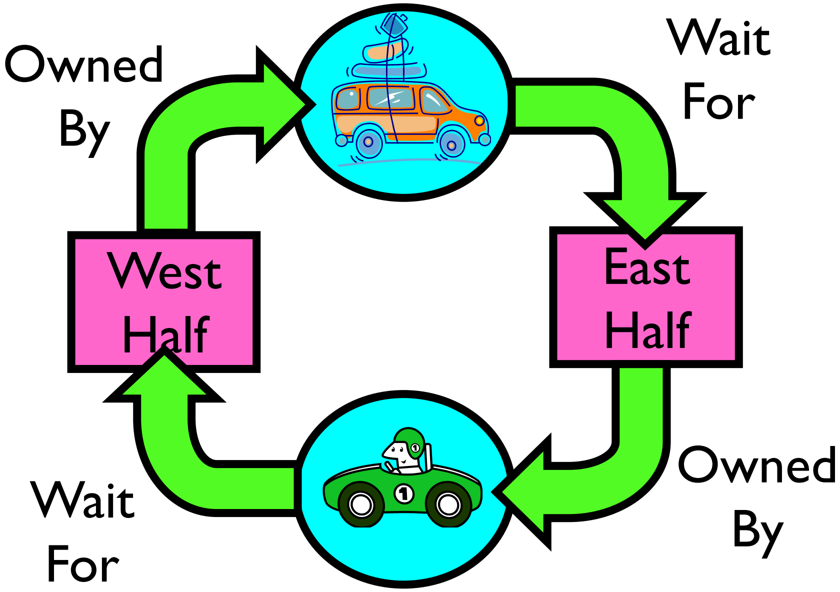
Example: Single-Lane Bridge Crossing

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





Owned By

Wait For

West Half

East Half

Owned By

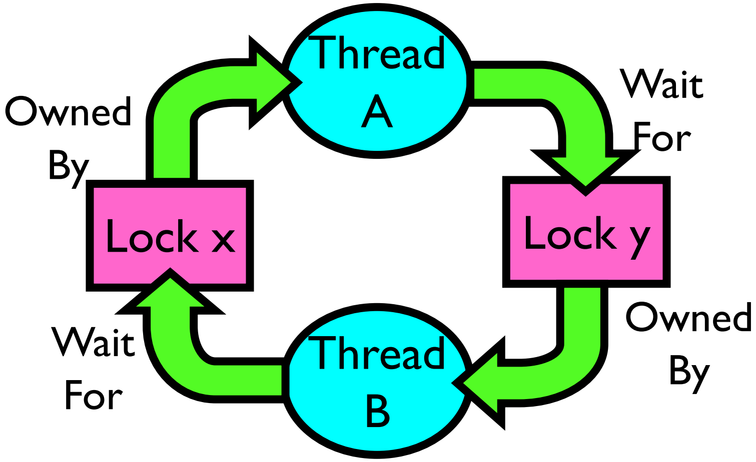
Wait For

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

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



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

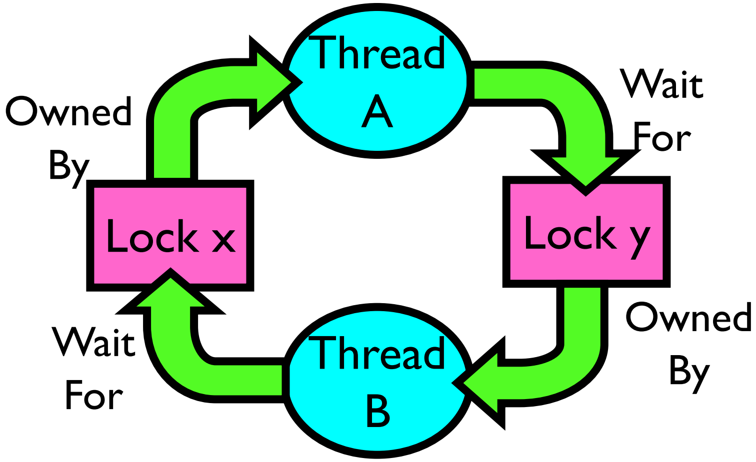
Deadlock with Locks: “Unlucky” Case

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();
```

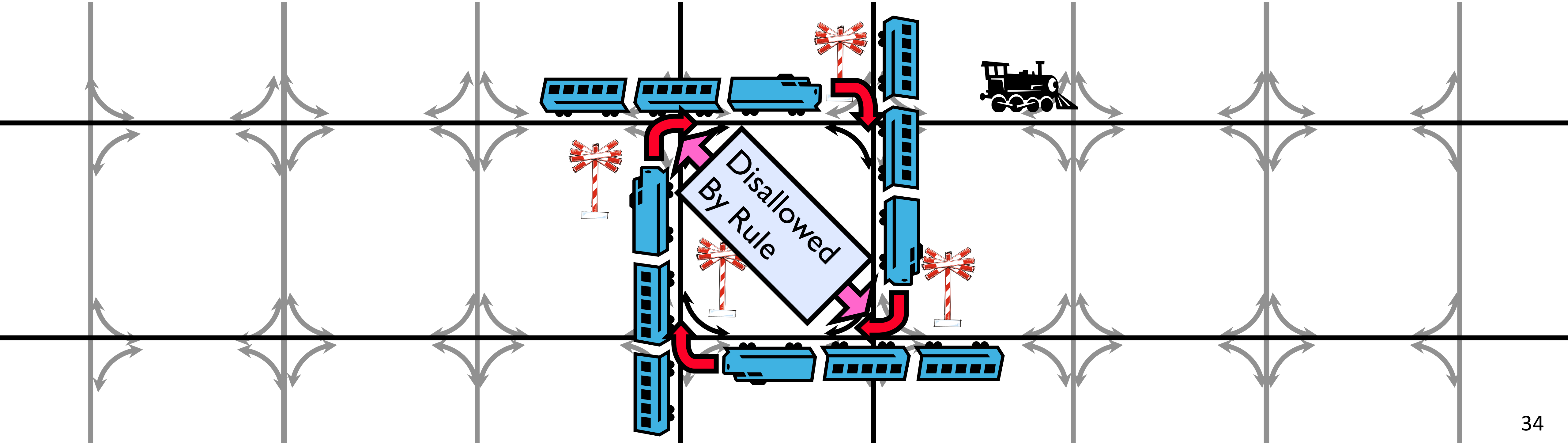
Thread B:

```
y.Acquire();


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

Sometimes, schedule won't trigger deadlock!

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)



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)
```

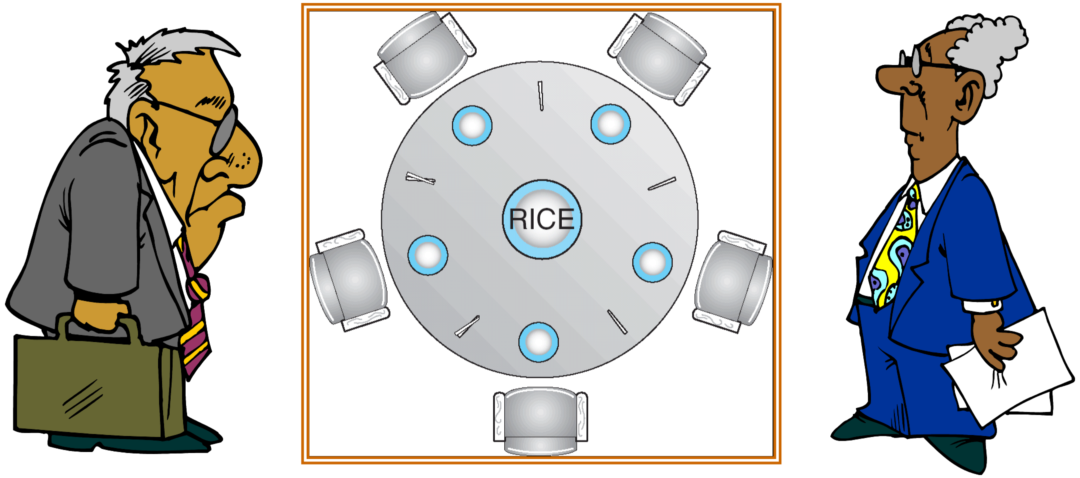
Thread B

```
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!

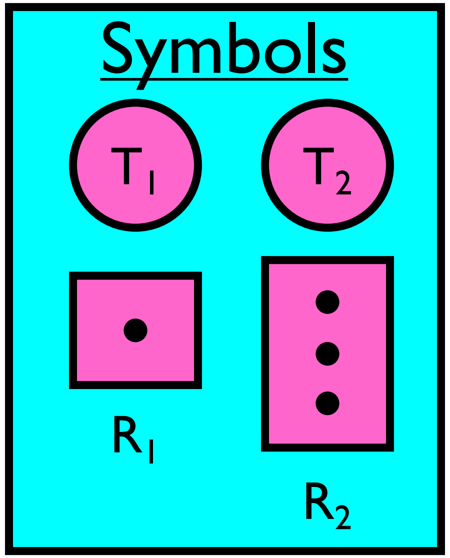
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat

- 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, …, 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

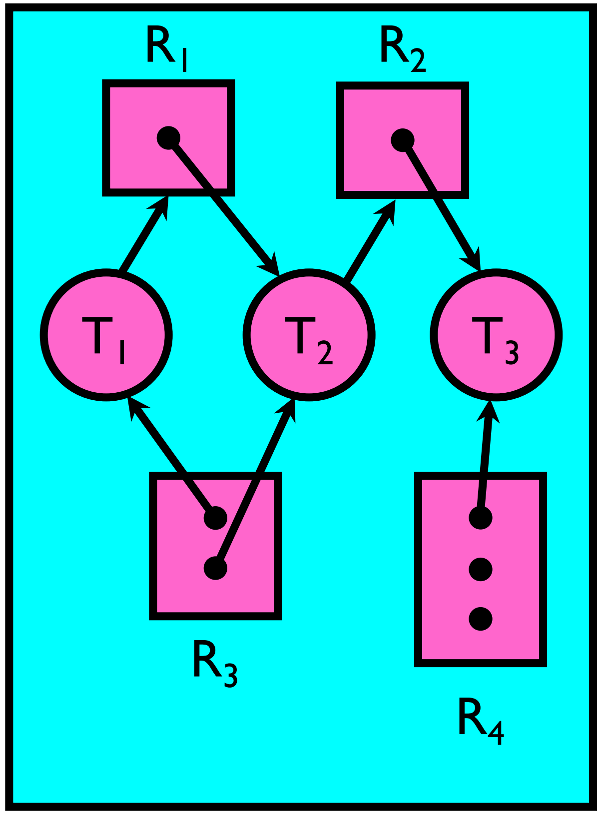


- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

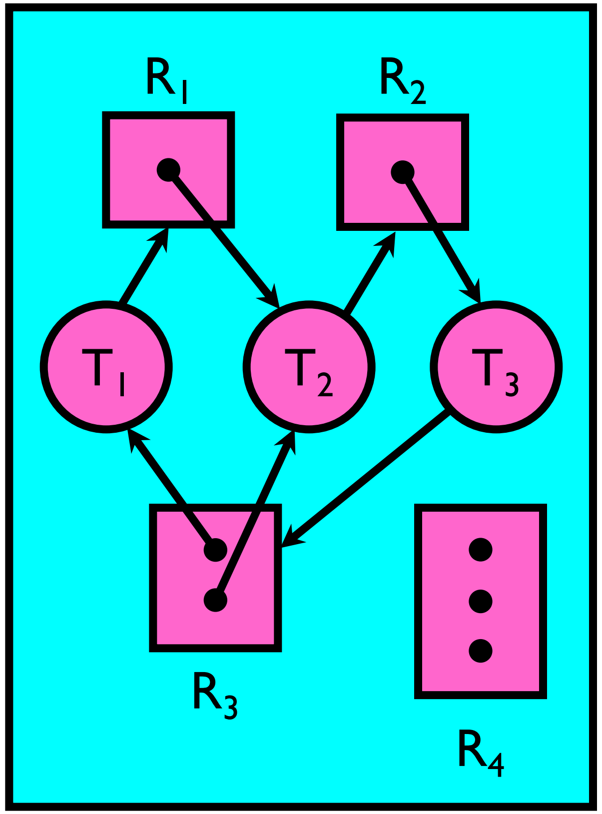
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the 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

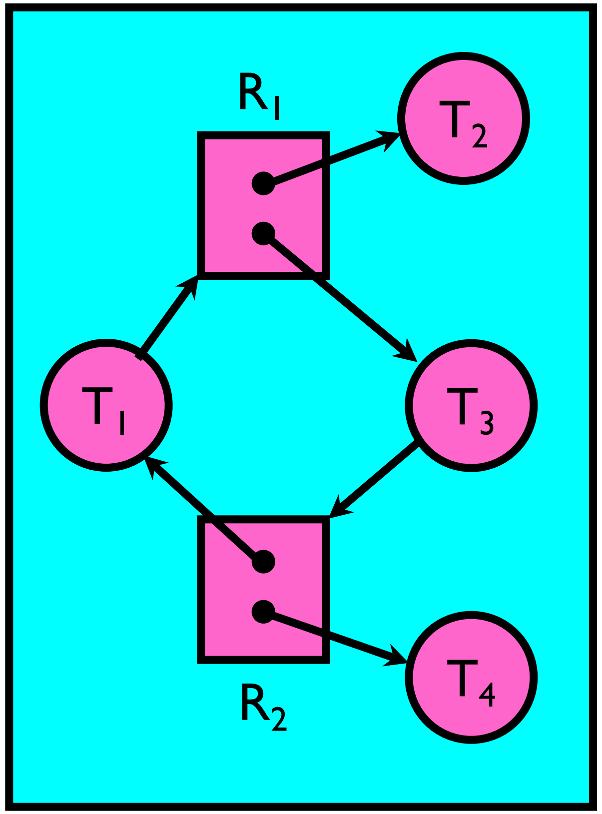
- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



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):

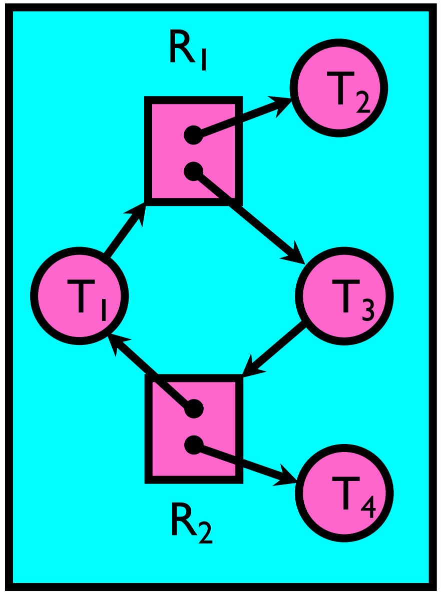
`[FreeResources]`: Current free resources each type

`[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
   For each 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"

Summary (1 of 3)

- 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

Summary (2 of 3)

- Realtime Schedulers such as EDF
  - Guaranteed behavior by meeting deadlines
  - Realtime tasks defined by tuple of compute time and period
  - Schedulability test: is it possible to meet deadlines with proposed set of processes?
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)
- Linux CFS Scheduler: Fair fraction of CPU
  - Approximates an “ideal” multitasking processor
  - Practical example of “Fair Queueing”

Summary (3 of 3)

- Four conditions for deadlocks
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Techniques for addressing Deadlock
  - Deadlock prevention:
    - write your code in a way that it isn’t prone to deadlock
  - Deadlock recovery:
    - let deadlock happen, and then figure out how to recover from it
  - Deadlock avoidance:
    - dynamically delay resource requests so deadlock doesn’t happen
    - Banker’s Algorithm provides on algorithmic way to do this
  - Deadlock denial:
    - ignore the possibility of deadlock