CS162
Operating Systems and Systems Programming
Lecture 12

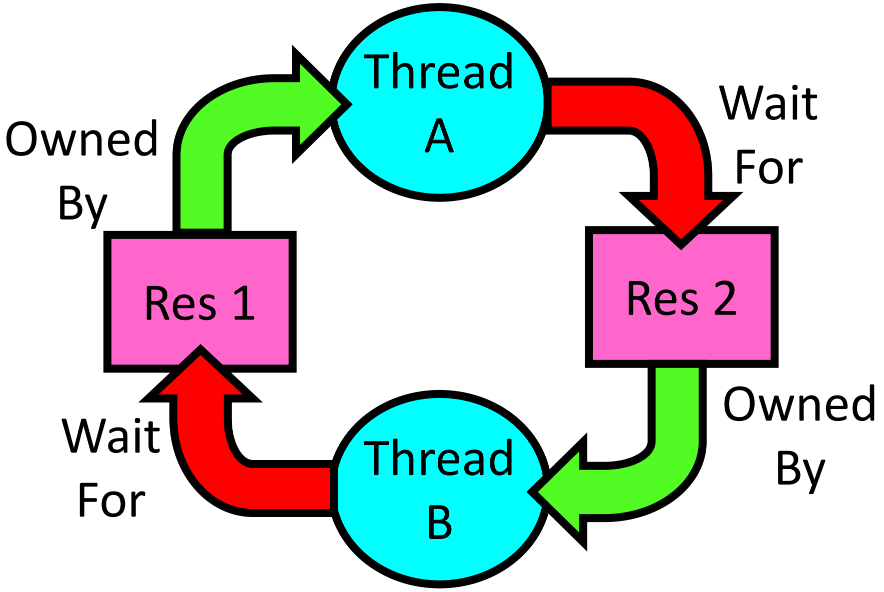
Scheduling 3: Deadlock

Recall: 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)	Linux CFS
Fairness – Wait Time to Get CPU	Round Robin
Favoring Important Tasks	Priority

Deadlock: A Deadly type of Starvation

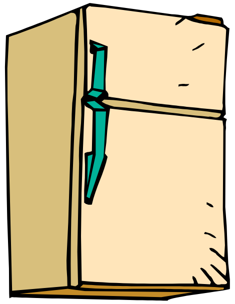
- Deadlock: cyclic waiting for resources
- Thread A owns Res 1 and is waiting for Res 2
- Thread B owns Res 2 and is waiting for Res 1



Deadlock: A Deadly type of Starvation

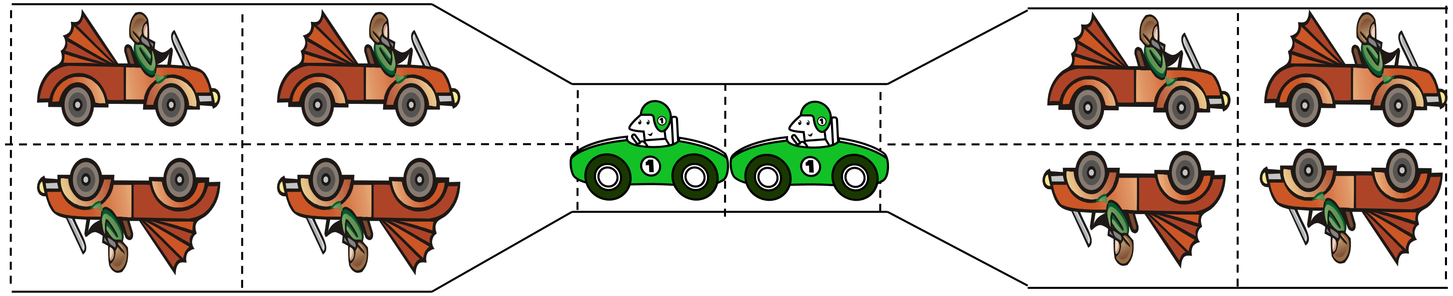
- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock implies starvation but starvation does not imply deadlock
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Example: Single-Lane Bridge Crossing



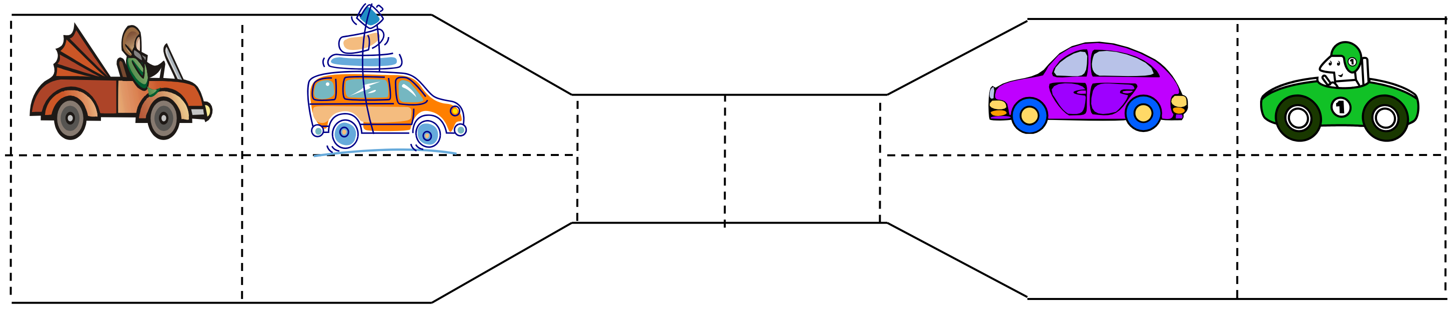
Bridge Crossing Example

- Each segment of road can be viewed as a resource



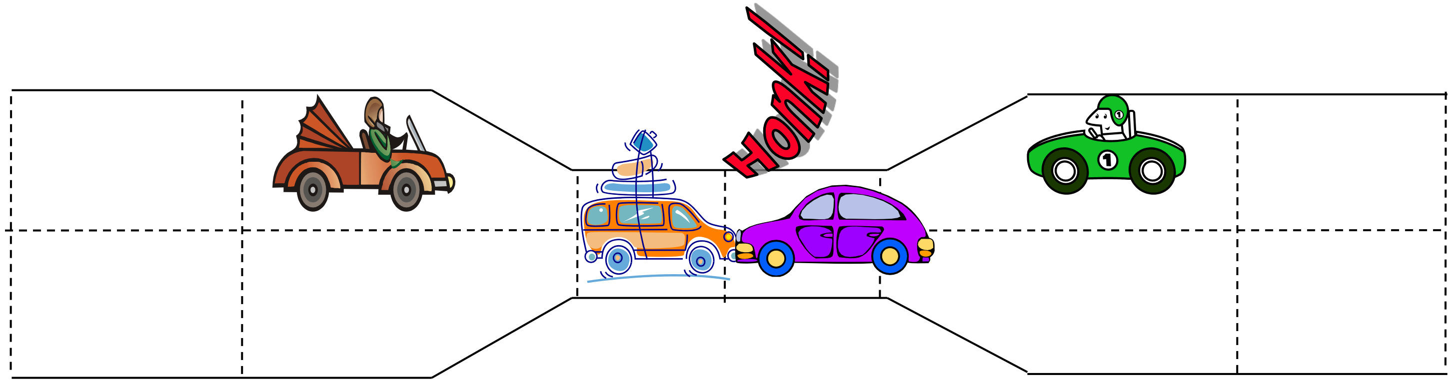
- Rules:
  – Car must own the segment under them
  – Must acquire segment that they are moving into
  – For bridge: traffic only in one direction at a time

Bridge Crossing Example

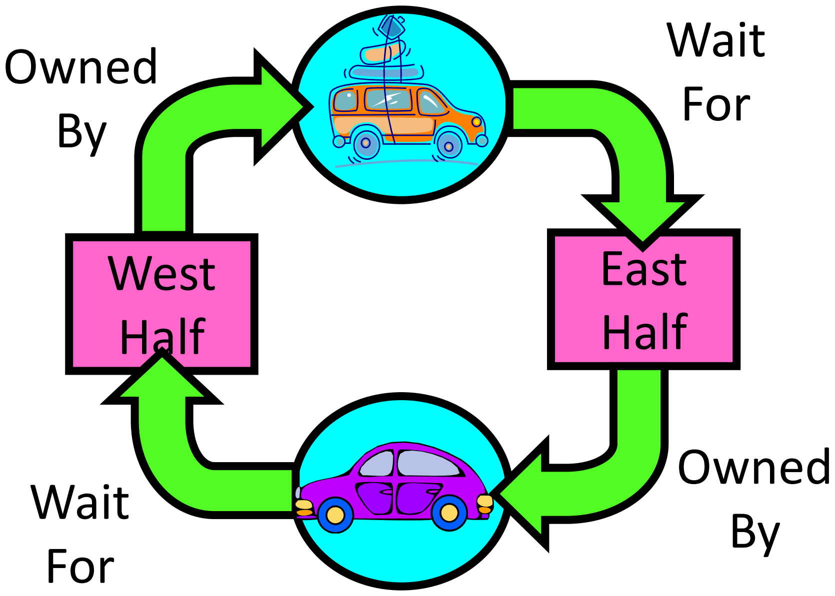


- Car must own the segment under them
- Must acquire segment that they are moving into

Bridge Crossing Example

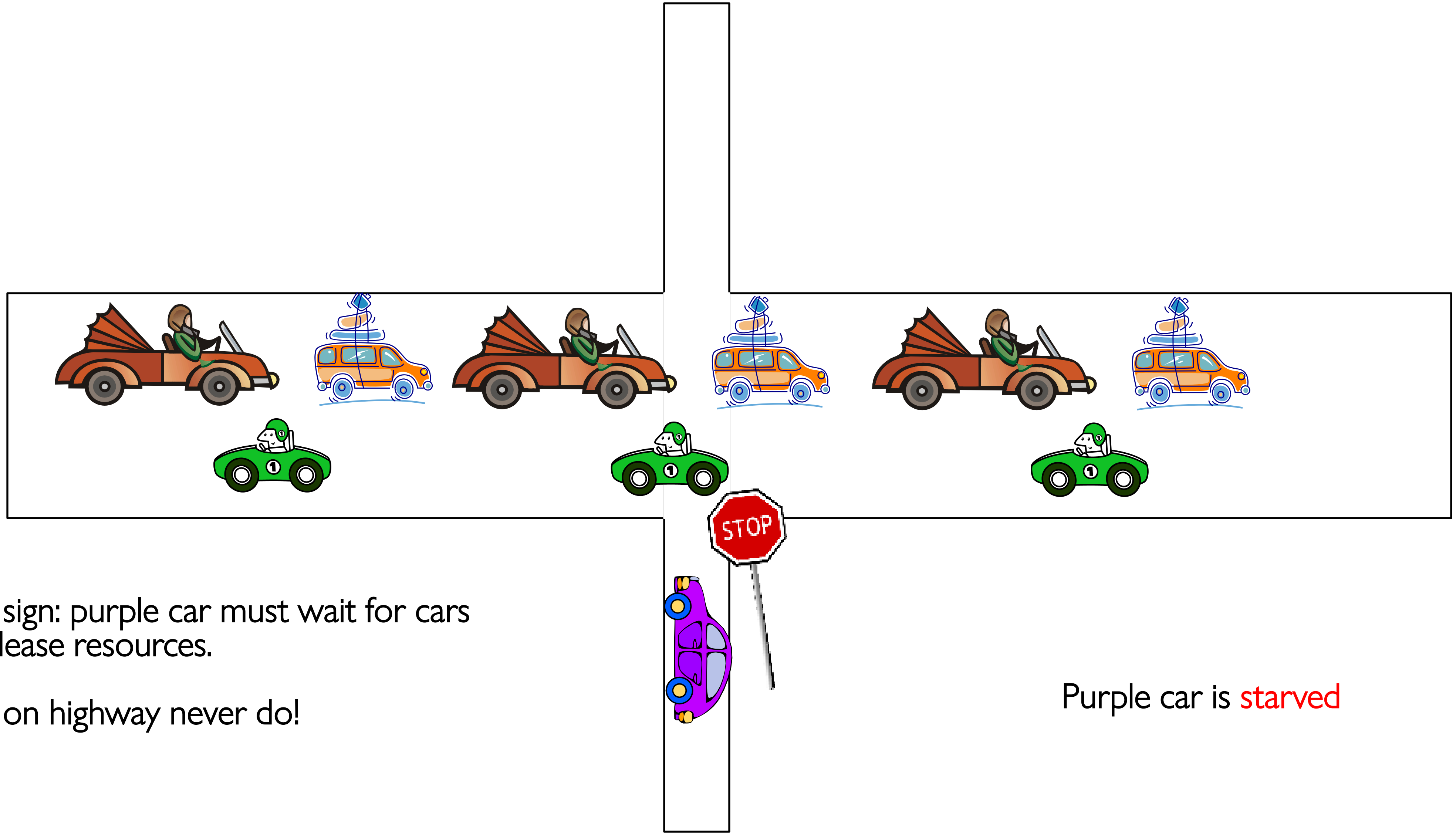


Deadlock: Circular waiting for resources



Could be resolved by "external" intervention:
- fork-lifting a car of the bridge (equivalent to killing a thread)
- Asking cars to backup (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

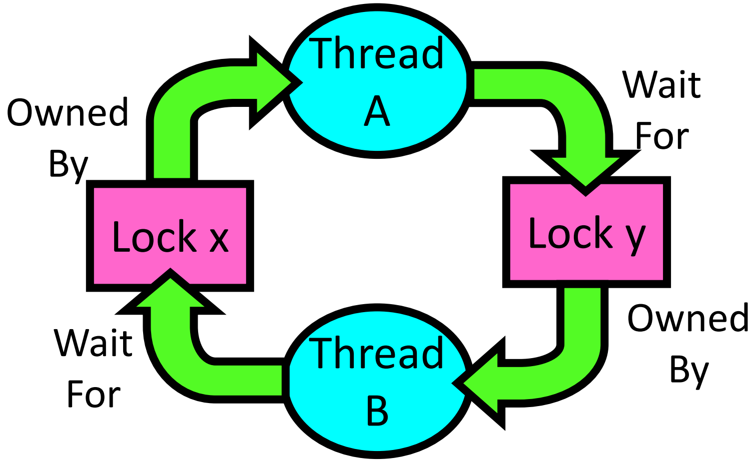
Deadlock with Locks

Thread A:

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

Thread B:

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



- This lock pattern exhibits *non-deterministic deadlock*
  - Sometimes it happens, sometimes it doesn't!

- 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

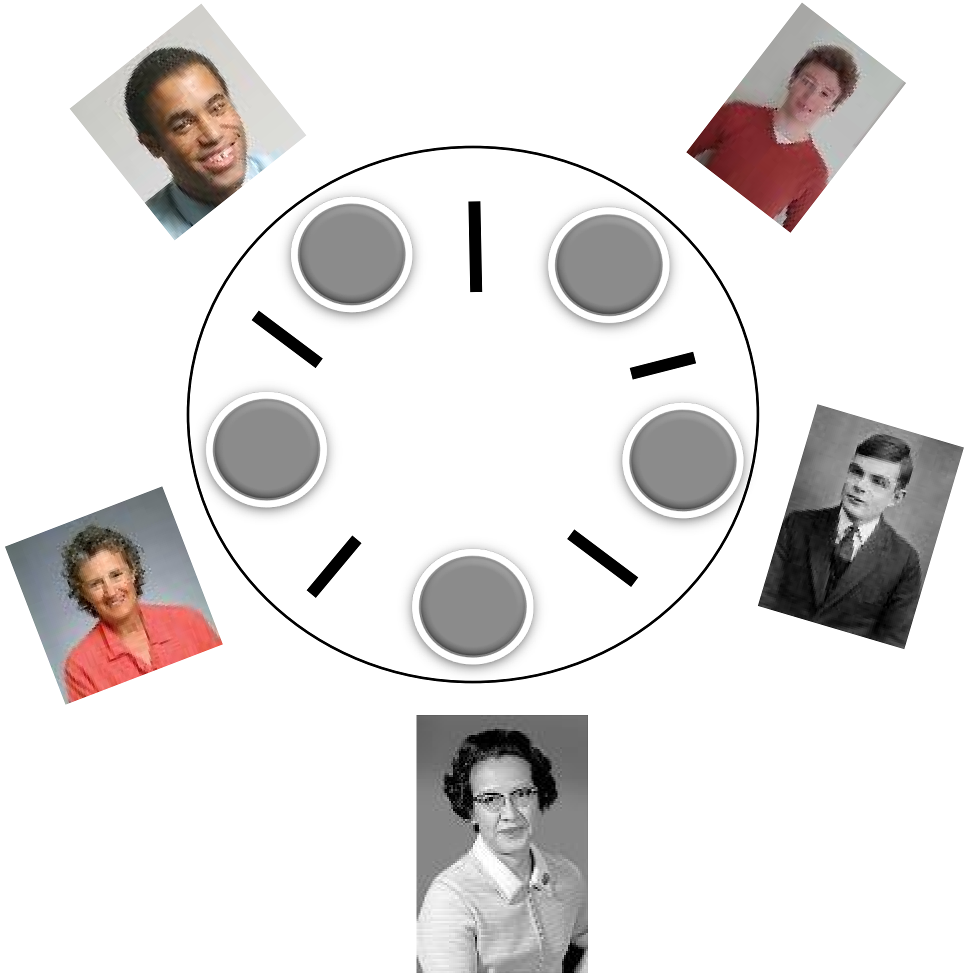
- Threads often block waiting for resources
  – Locks
  – Terminals
  – Printers
  – CD drives
  – Memory

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

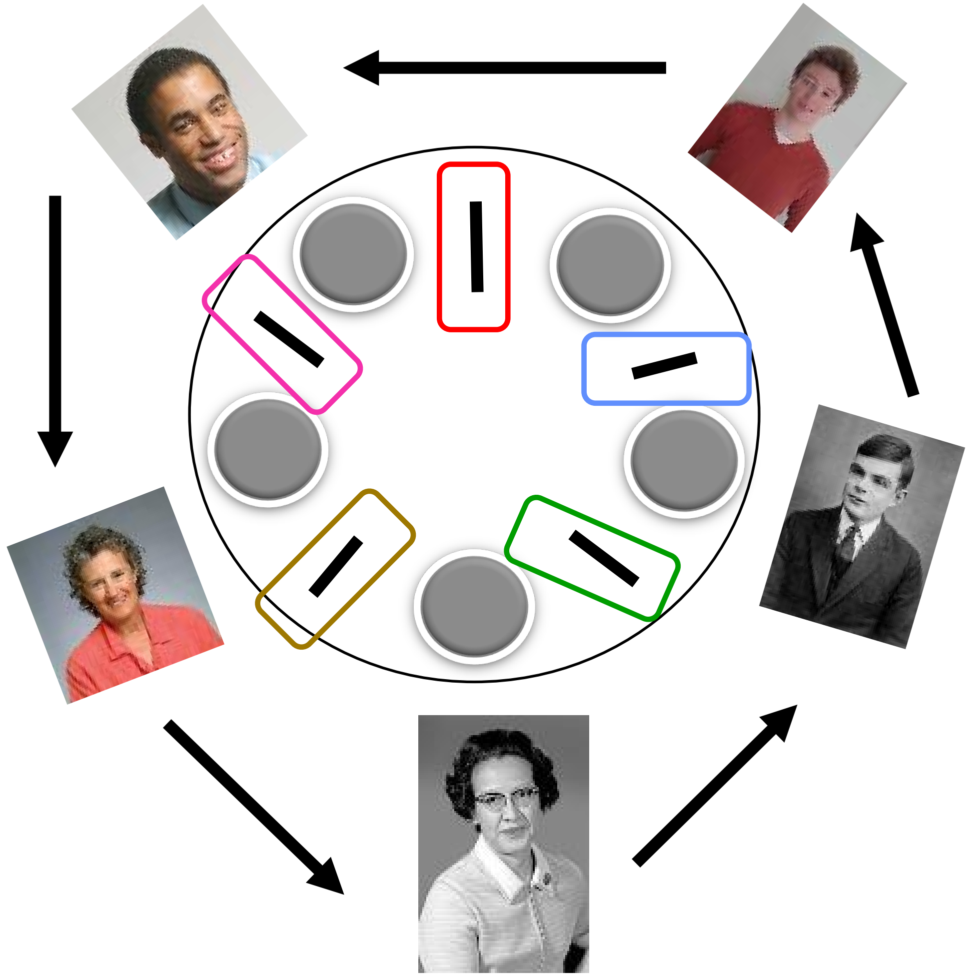
- You can deadlock on any of these!

Dining Computer Scientists Problem

- Five chopsticks/Five computer scientists
- Need two chopsticks to eat



Free for all leads to deadlock and (literal) starvation



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

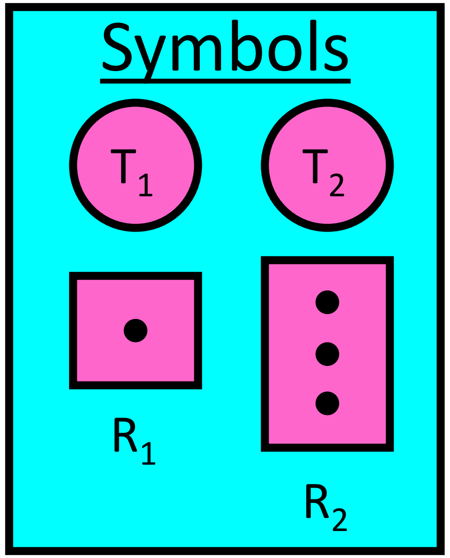
Four requirements for occurrence of Deadlock

- Mutual exclusion and bounded resources
  - 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

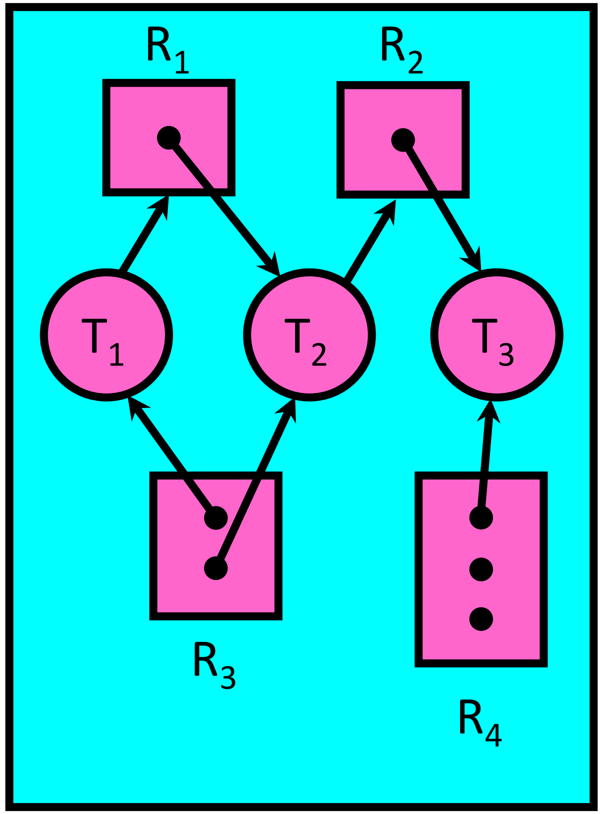
- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

    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 $R_j \rightarrow T_i$

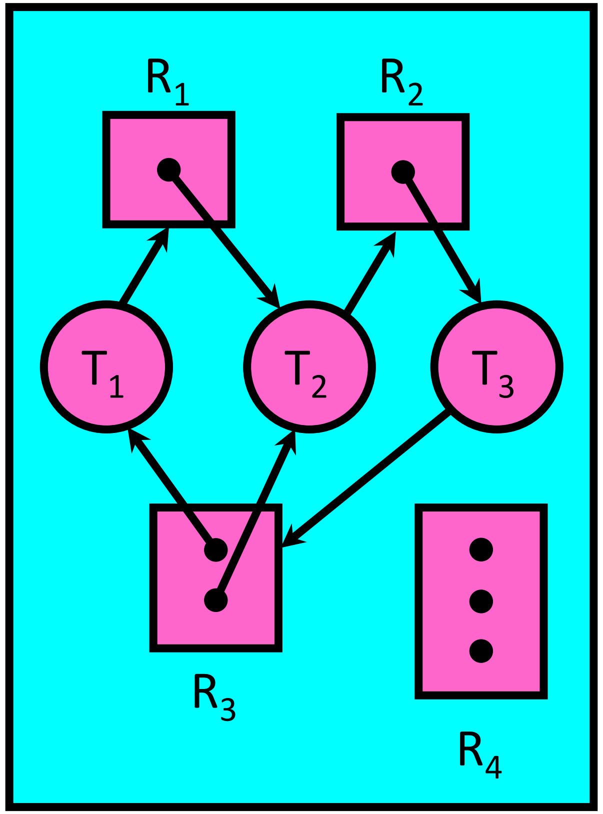


Resource-Allocation Graph Examples

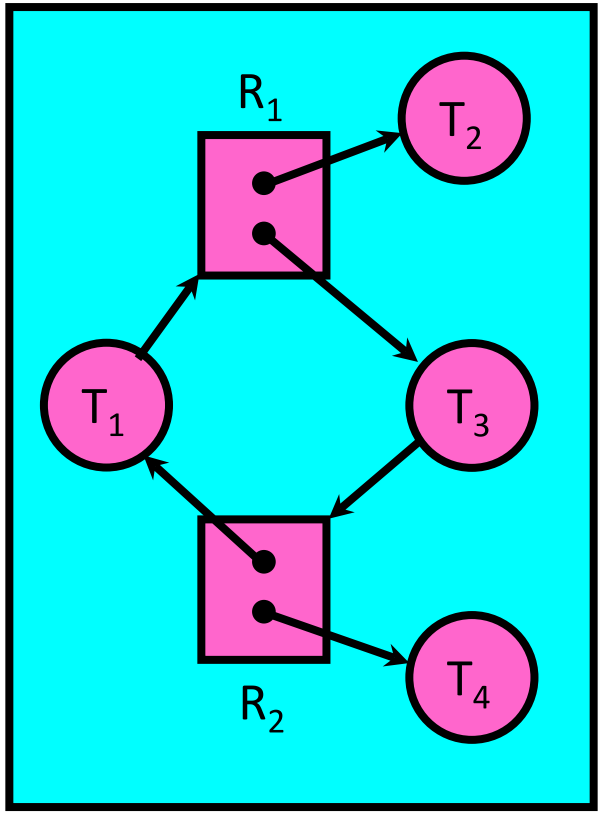
- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



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

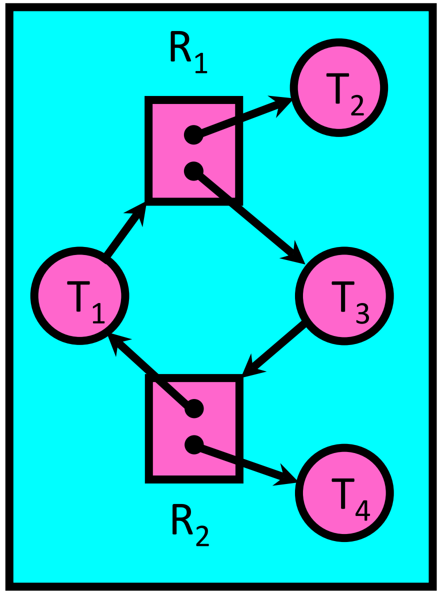
- See if tasks can eventually terminate on their own

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

- Threads left in `UNFINISHED` $\Rightarrow$ deadlocked

Deadlock Detection Algorithm

-
```
[Avail] = [FreeResources]
    Add all threads to UNFINISHED
    do {
        done = true
        Foreach thread in UNFINISHED {
            if ([Request_node] <= [Avail]) {
                remove thread from UNFINISHED
                [Avail] = [Avail] + [Alloc_node]
                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?

- 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

Deadlock prevention

- Condition 1: Mutual exclusion and bounded resources

  => Provide sufficient resources

- Condition 2: Hold and wait

  ⇒Abort request or acquire requests atomically

- Condition 3: No preemption

  => Preempt threads

- Condition 4: Circular wait

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

Condition 1: (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!

Condition 2: Request Resources Atomically

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();
```

Condition 3: Preemption

- Force thread to give up resource
- Common technique in databases using database aborts
  - A transaction is “aborted”: all of its actions are undone, and the transaction must be retried
- Common technique in wireless networks:
  - Everyone speaks at once. When a resource collision is detected, retry at a new, random time

Condition 4: Circular Waiting

- Force all threads to request resources in a particular order preventing any cyclic use of resources

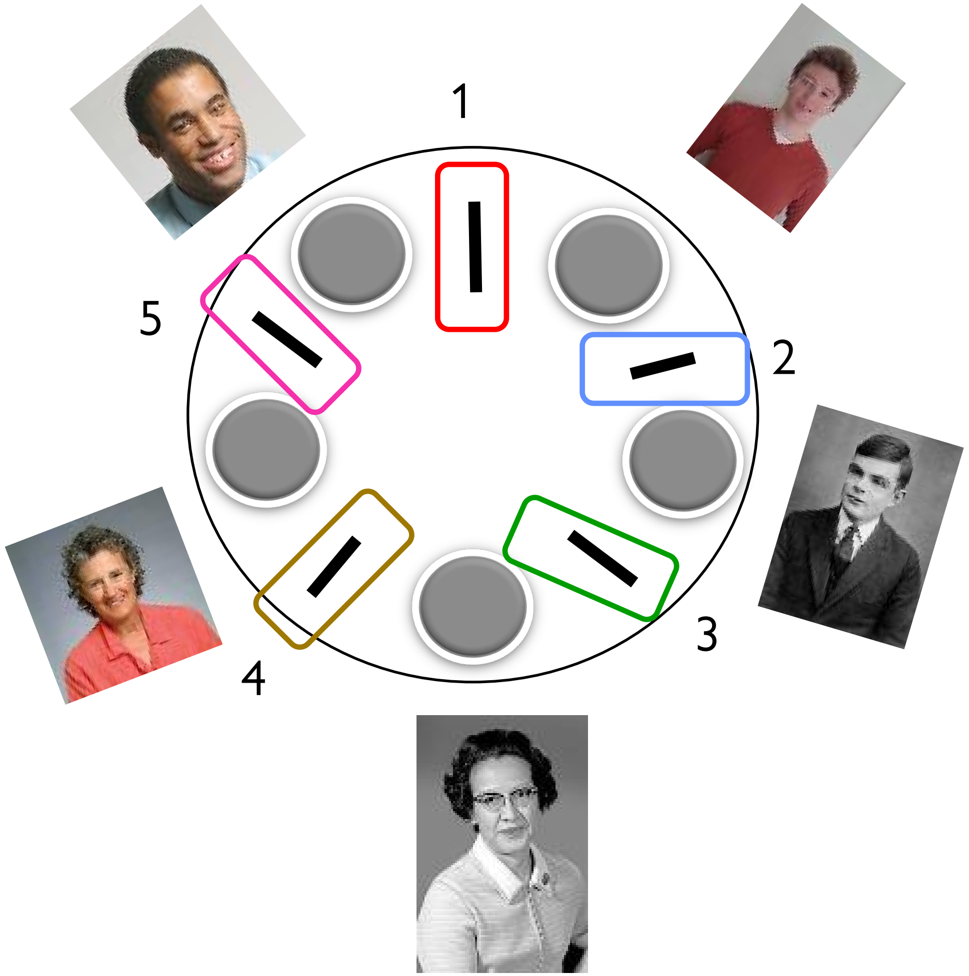
Thread A:
```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

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();
…
x.Release();
y.Release();
```

Condition 4: Circular Waiting



- Joseph: first 1 then 5
- Crooks: first 2 then 1
- Turing: first 3 then 2
- Johnson: first 4 than 3
- Liskov: first 5 then 4

If ensure that Joseph graphs chopstick 5 followed by 1, no deadlock!

Recall: 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

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:

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

Thread B:

```
y.Acquire();
x.Acquire();   // Wait?
…              // But it's already too late…
x.Release();
y.Release();
```

Blocks…

Wait?

But it's already too late…

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
  - Also considered "unsafe"

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

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:

Thread A:

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

Thread B:

```
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) $\geq$ 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

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)
```

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
     Add all threads to UNFINISHED
     do {
        done = true
        Foreach threads in UNFINISHED {
           if ([Max_threads]-[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 $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..

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_threads]-[Alloc_thread] <= [Avail]) {
                    remove thread from UNFINISHED
                    [Avail] = [Avail] + [Alloc_thread]
                    done = false
                }
            }
        } until(done)
```

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

When Thread A acquires x:

Run Algorithm:
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 A from UNFINISHED

Safe state!

When Thread B acquires y:

Run Algorithm:
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 => Starvation, 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, or denial
- Banker's algorithm for avoiding deadlock