CS310 Operating Systems

Lecture 29: Deadlock - Introduction

Ravi Mittal

IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, University of California, Berkeley
  - Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II
    - Chapter 5.6
  - CS4410: Operating Systems, 2019sp, Slides - Deadlock

Reading

- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II, Chapter 6.5
- Book: Operating System Concepts, 10th Edition, by Silberschatz, Galvin, and Gagne
- CS4410: Operating Systems, 2019sp, Slides - Deadlock

Today, we will study

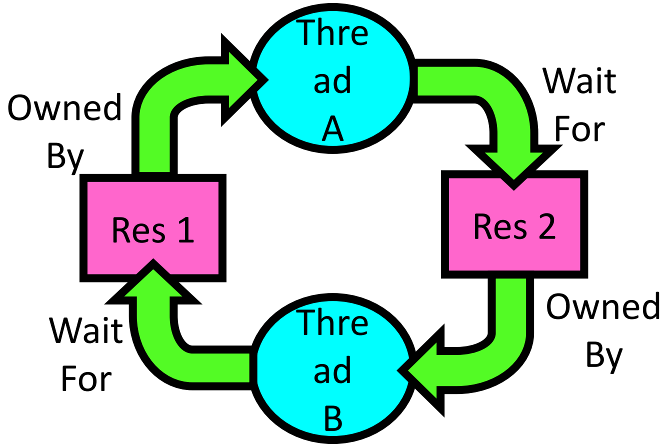
- What is deadlock?
- Why deadlock?
- Conditions for a deadlock
- Detecting a deadlock

What is a Deadlock?

Ensuring Progress

- Liveness:
  - A set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle
- Deadlock and Starvation are both liveness concerns
- Starvation: thread fails to make progress for an indefinite period of time
- 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

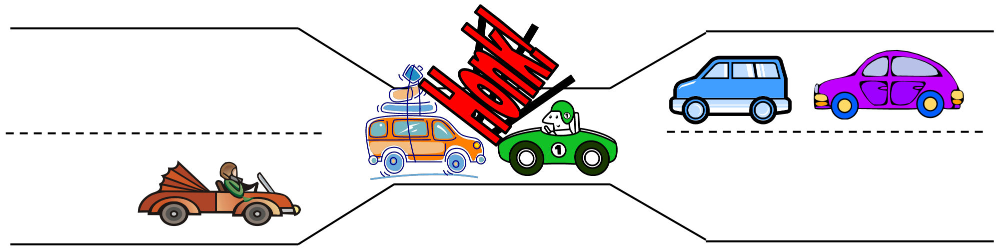
Deadlock: A Type of Starvation



- Starvation – thread fails to make progress for an indefinite period of time
- Deadlock – starvation due to a *cycle of waiting* among a set of threads
  - Each thread waits for some other thread in the cycle to take some action
- Deadlock implies starvation (example: dinning philosophers)
- Starvation does not imply deadlock

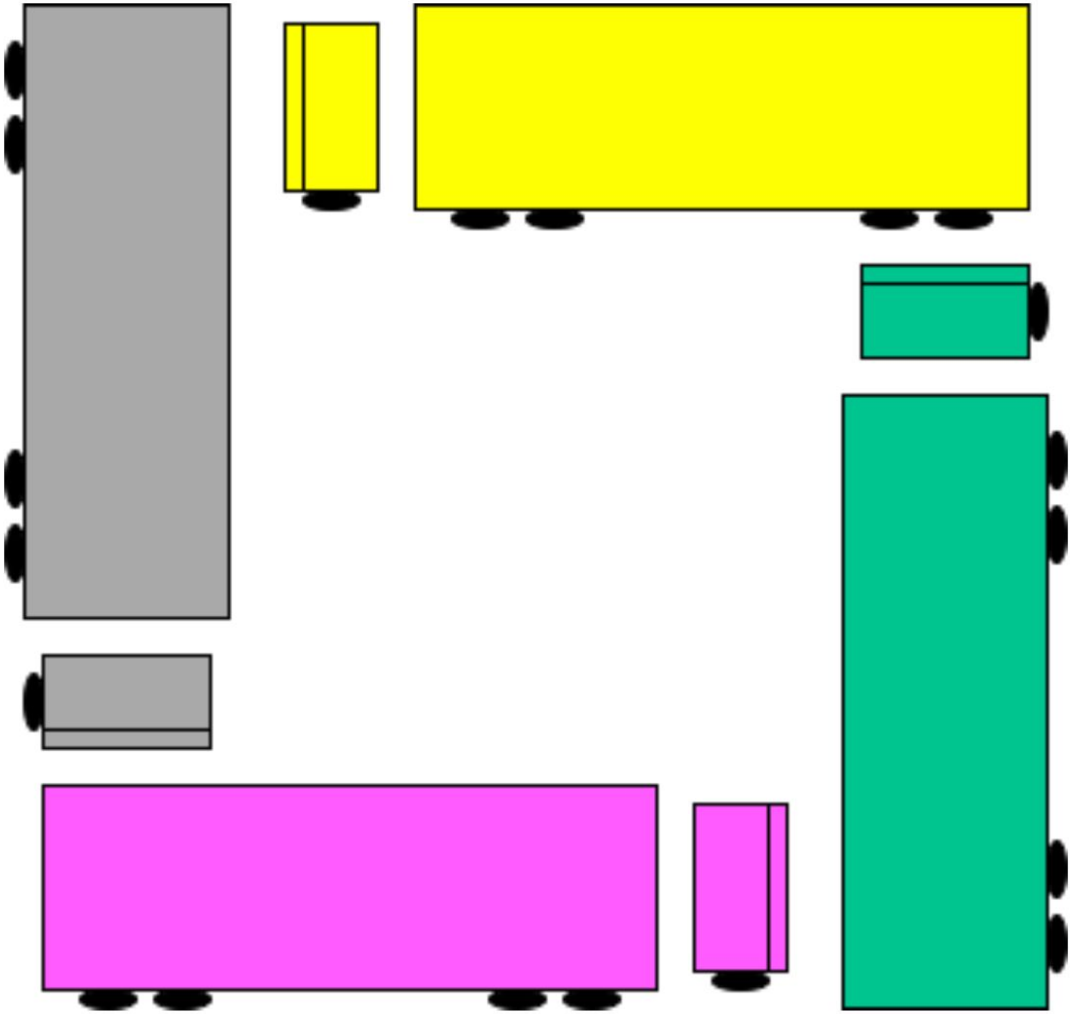
Example: Single-Lane Bridge Crossing

Bridge Crossing Example

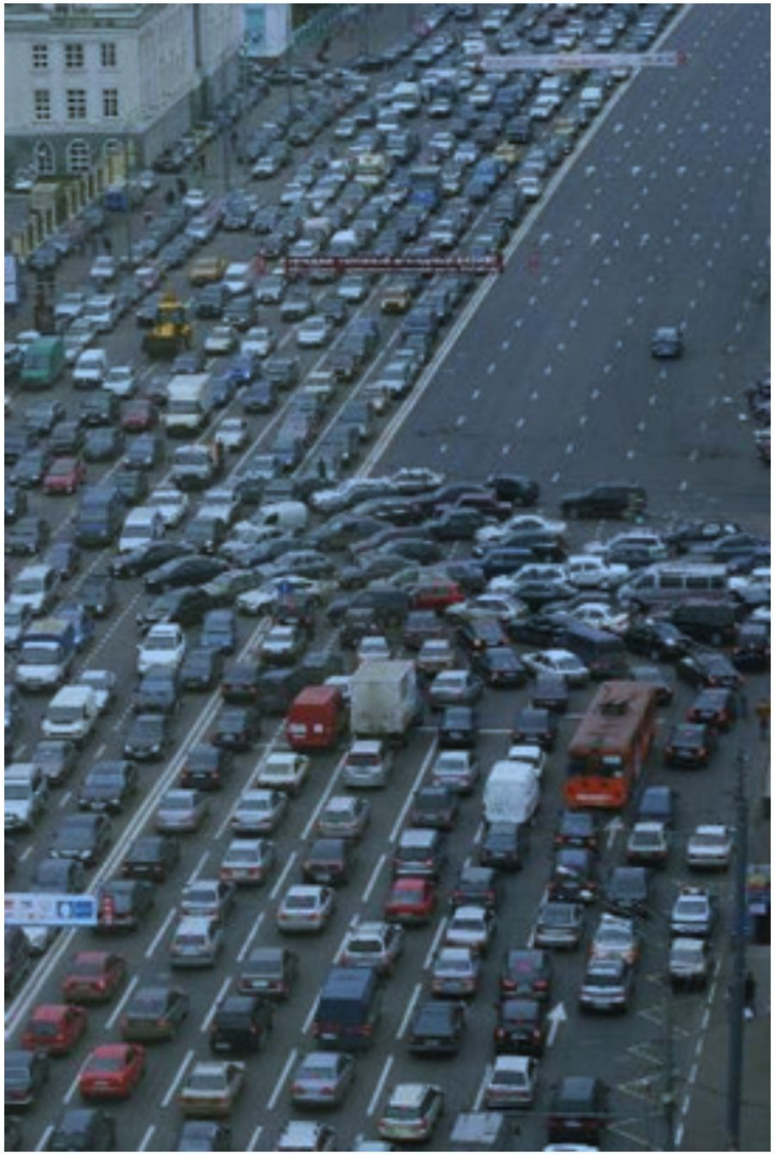


- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- **Deadlock:** Two cars in opposite directions meet in middle
- **Starvation** (not deadlock): Eastbound traffic doesn't stop for westbound traffic

Is this a deadlock?



Is this a deadlock?



Why Deadlock?

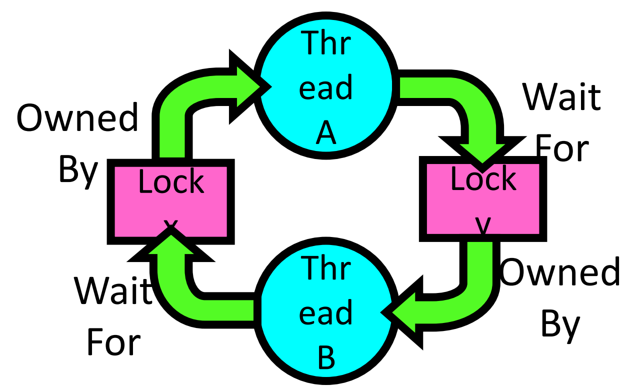
Deadlock with Locks

Thread A

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

Thread B

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



Nondeterministic Deadlock

Deadlock with Locks: Unlucky Case

Thread A

x.Acquire();

y.Acquire(); ***<stalled>***

<unreachable>

…

y.Release();

x.Release();

Thread B

y.Acquire();

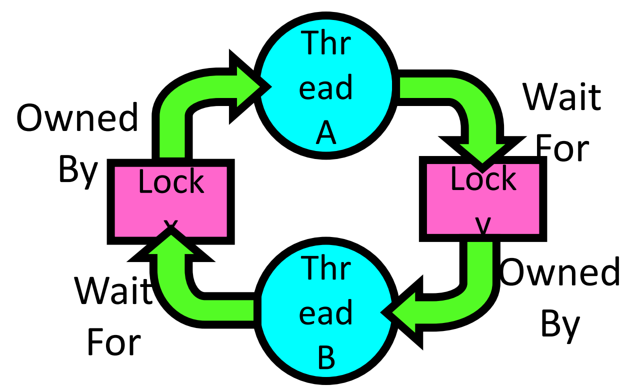
x.Acquire(); ***<stalled>***

<unreachable>

…

x.Release();

y.Release();



Deadlock with Locks: Lucky Case

Sometimes, schedule won't trigger deadlock

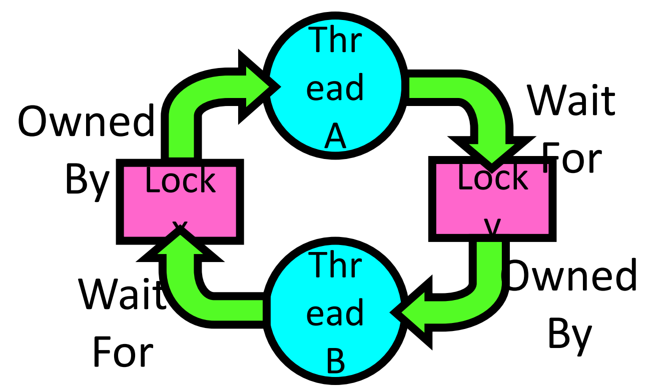
Thread A

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

Thread B

```
y.Acquire();

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



Other Types of Deadlock

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

- Threads often block waiting for other thread

Deadlock with Space

```
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)
```

If only 2 MB of space, we can get in to deadlock situation

Deadlock happens sometimes

- A system may be live without starvation or deadlock for most of the time
- However, a particular workload or unlucky interleaving may cause starvation or deadlock
- Example: Dining philosophers may succeed in eating for a long time before hitting the unlucky sequence of events that causes them to deadlock.
- Testing may not discover problems. Hence, it is important to design a system that is deadlock free

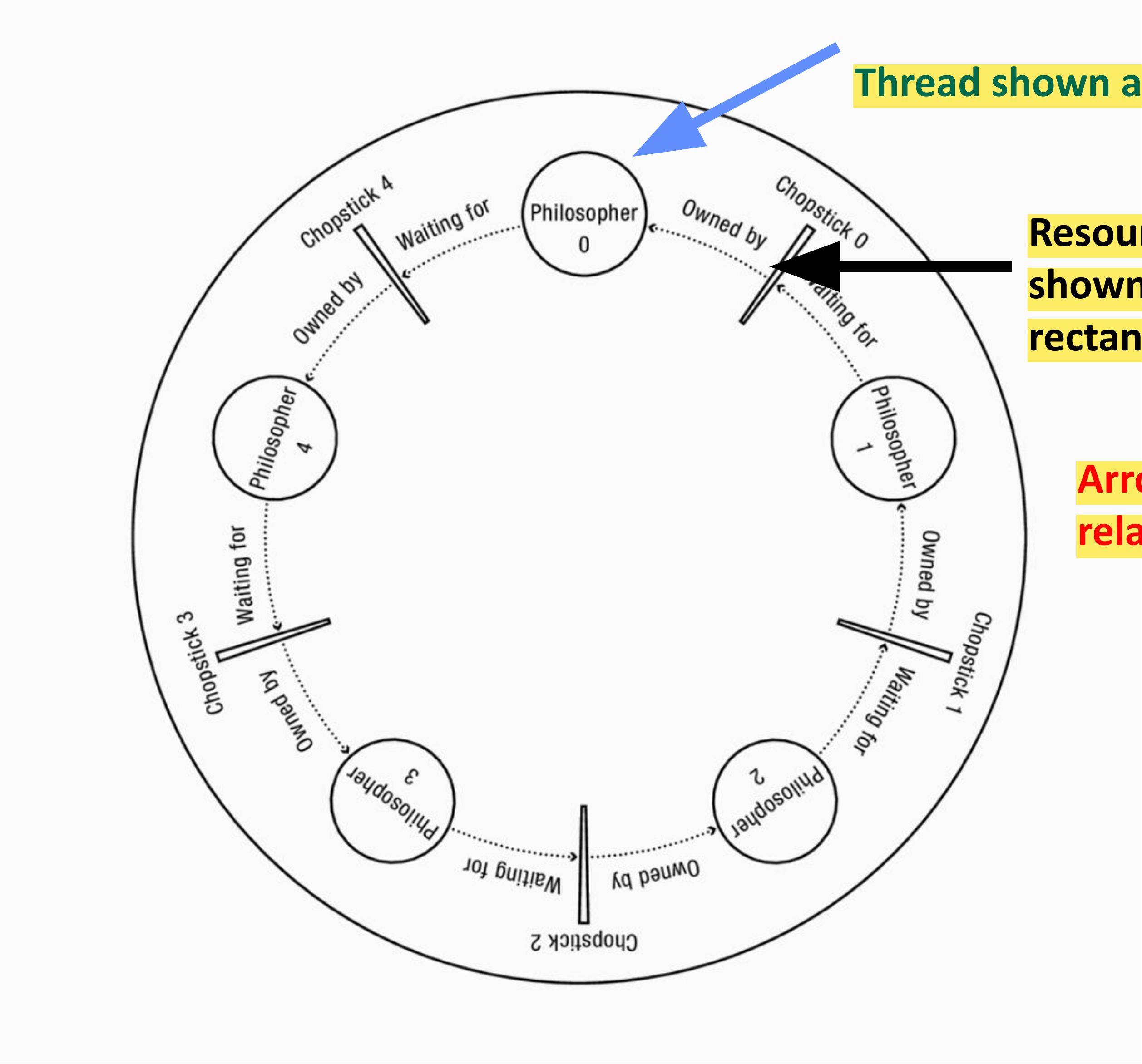
Conditions for a Deadlock !

Necessary Conditions for Deadlock

- There are 4 necessary conditions for deadlock to occur
- If you can prevent any one of these conditions, you can eliminate the possibility of deadlock

1. Bounded Resources
    - There are a finite number of threads that can simultaneously use a resource
2. No preemption
    - Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it
3. Wait while Holding
    - A thread holds one resource while waiting for another
        - Also called multiple independent requests
4. Circular Waiting
    - A set of waiting threads such that each thread is waiting for a resource held by another

Case: Dinning Philosopher Problem



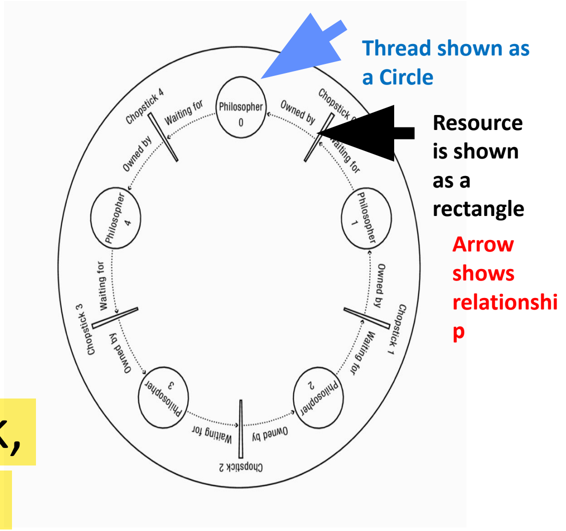
Thread shown as a Circle

Resource is shown as a rectangle

Arrow shows relationship

Case: Dinning Philosopher Problem

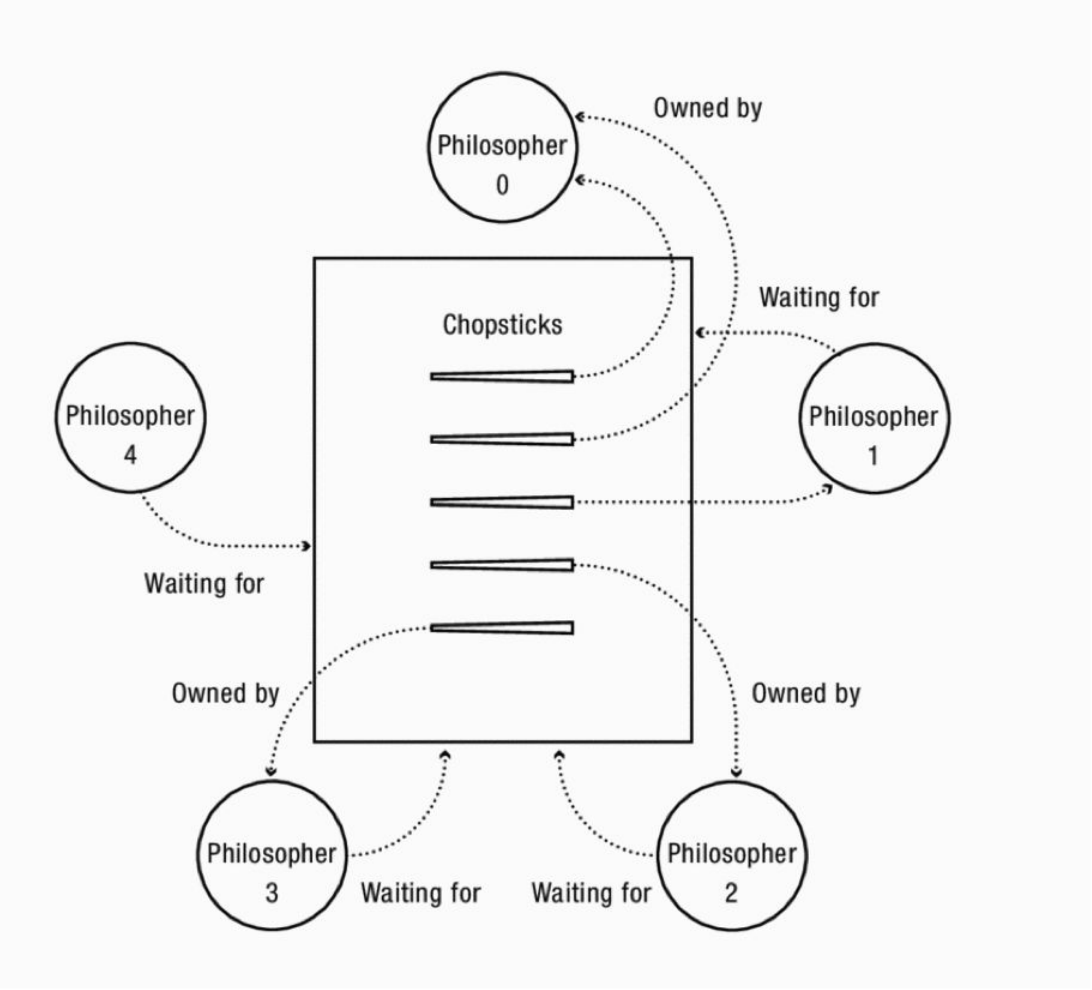
- Bounded Resources
  - Each chopstick can be held by a single philosopher at a time
- No preemption
  - Once a philosopher picks up a chopstick, she does not release it until she is done eating even it means no one will ever eat
- Wait while holding
  - When a philosopher needs to wait for a chopstick, she continues to hold onto any chopstick she has already picked up
- Circular Waiting
  - In mapped abstract graph, we can see circular waiting



Case: Dinning Philosopher Problem

- Four conditions are necessary but not sufficient for deadlock
- If there are multiple instances of a type of resource, there can be cycle of waiting without deadlock
  - A thread not in cycle may return resource that enables a waiting thread to proceed

No deadlock


Philosopher
0
Owned by
Chopsticks
Waiting for
Philosopher
4
Philosopher
1
Waiting for
Owned by
Owned by
Philosopher
3
Waiting for
Waiting for
Philosopher
2

No deadlock

- Philosopher 0 is with two chopsticks
- Philosophers 1, 2,3 : each as one chopstick
- Philosopher 4 doesn’t have any chopstick

- Bounded resources – 5 chopsticks
- No preemption – can’t forcibly take chopstick away from philosopher’s hand
- Wait while holding – philosophers 2, 3, 4 are holding chopsticks while waiting for another
- Circular Waiting - each of philosophers 2, 3, and 4 are waiting for a resource held by another of them
- Eventually, philosopher 1 will release it’s two chopsticks
  - Will allow philosopher 2 and 3 to eat and release chopsticks
  - Now philosophers 4 and 5 can eat

Detecting a Deadlock

Simple Deadlock detection mechanism

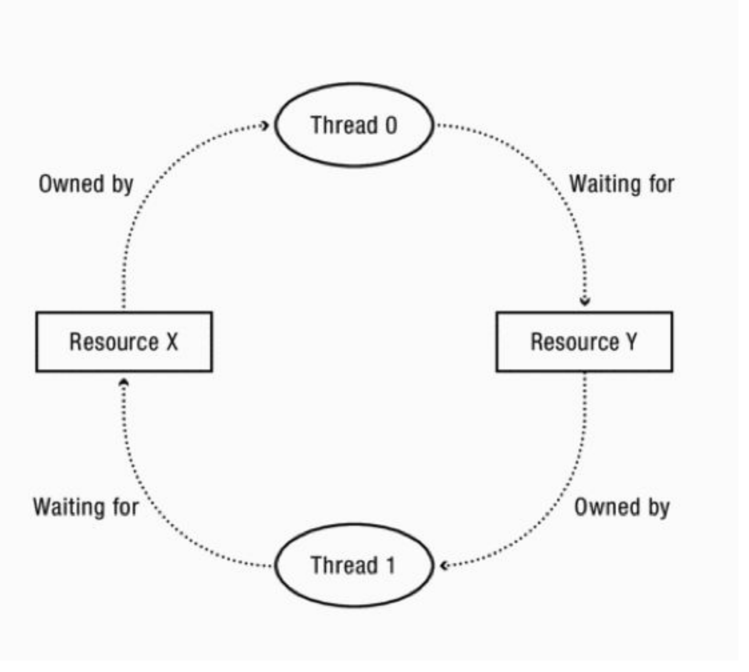
- Deadlock detection mechanisms must be simple
  - Even if it leads to occasional false positive
- A program can choose to wait only briefly before declaring that recovery is needed

Deadlock Detection – Simple approaches

- PSTN Networks: When making an International call (or local call), resources on the path are booked.
  - When unable to find a free circuit on the next hop – it cancels connection attempt and sends a message "all lines on the path are busy. Try after some time"
- Internet: When a router is overloaded it runs out of packet buffers – drops incoming packets
  - Alternative would be each router to wait to send a packet until it knows that the next router has space – could lead to deadlock – Identifying a deadlock would be expensive and time consuming instead of dropping packets

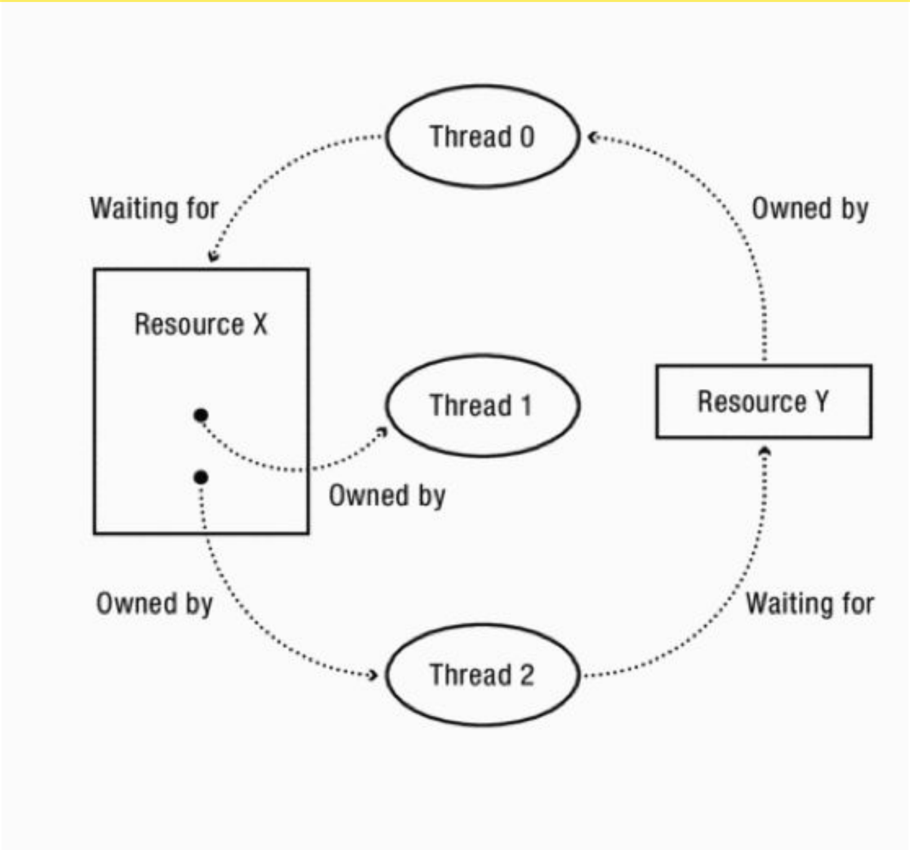
Analyzing - Resource Allocation Graph (RAG)

- Consider several resources and only one thread can hold each resource at a time
  - Example: resources – one printer, one keyboard, one speaker
- We can detect a deadlock by analyzing simple graph



Multiple instances of one resource

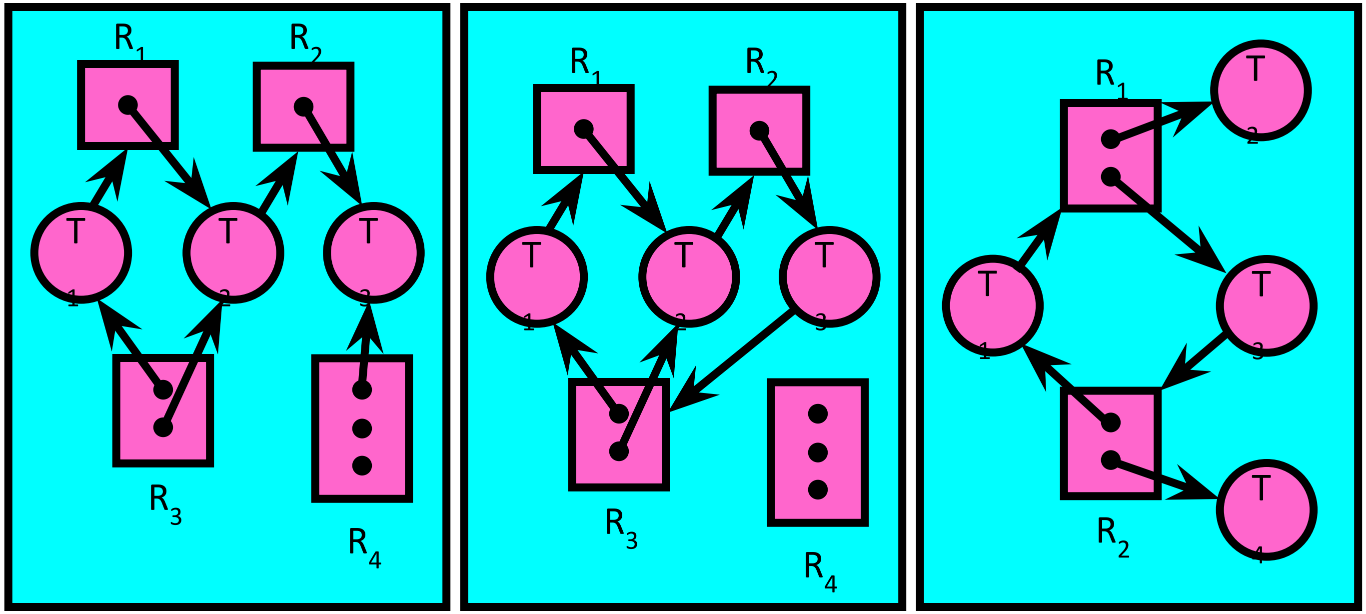
- Multiple instances of a resource can be represented as a resource with k interchangeable instances
    - Eg K equivalent printers as a node with k connections
- Cycle is necessary but not sufficient condition for deadlock



Resource-Allocation Graph Examples

Model: Directed Graph

- request edge
  - $T_i \rightarrow R_j$
- assignment edge
  - $R_j \rightarrow T_i$



Simple Resource Allocation Graph

Allocation Graph With Deadlock

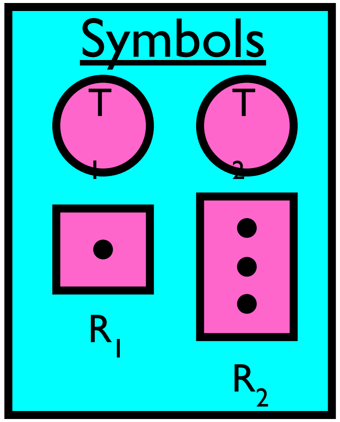
Allocation Graph With Cycle, but No Deadlock

Instead of thread T, we can also represent a process with a circle

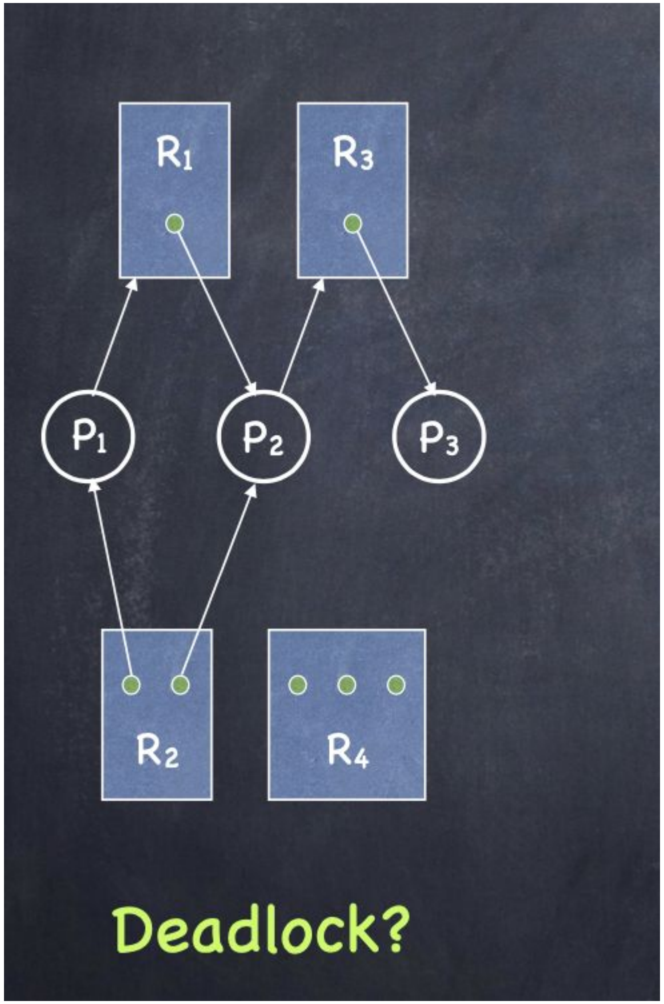
Resource-Allocation Graph

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

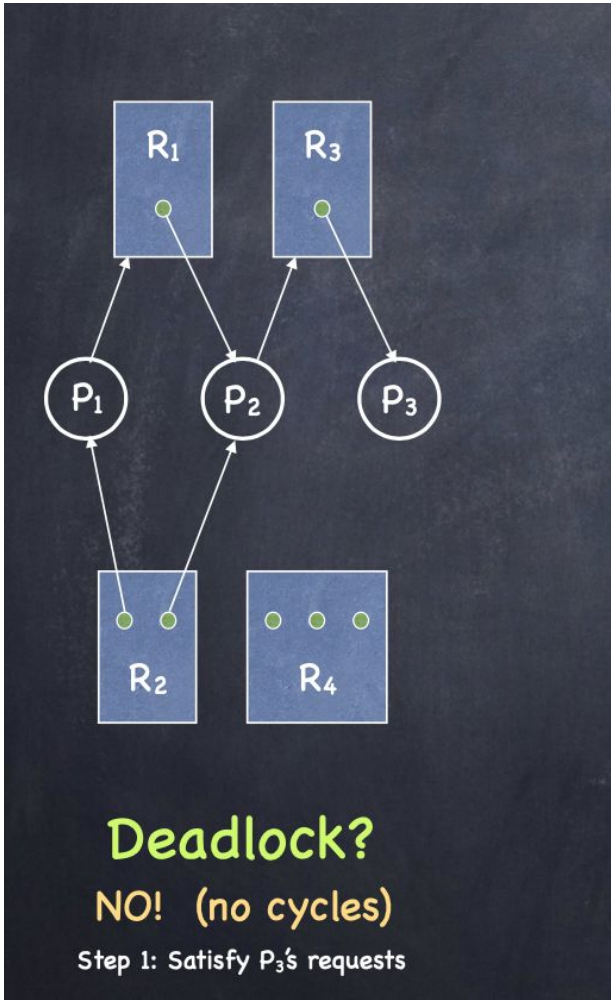
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances
  - Each thread utilizes a resource as follows:
    - `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



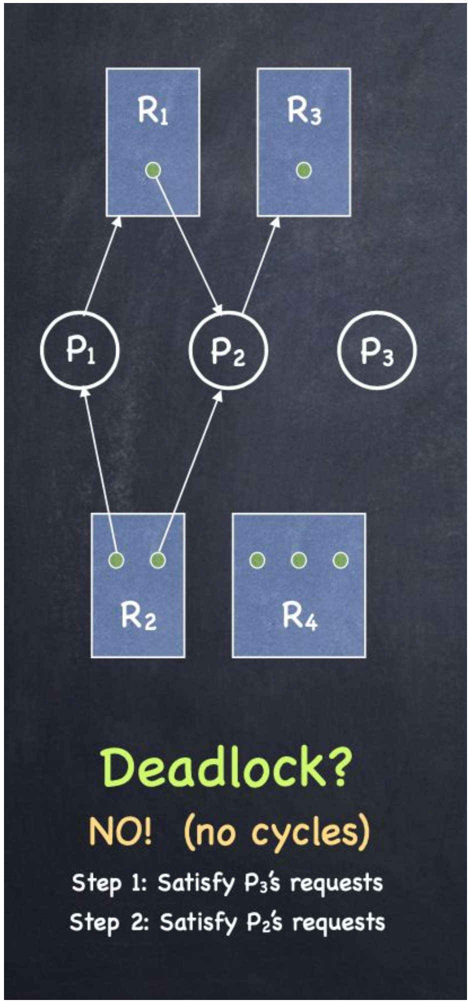
Example 1: RAG Reduction



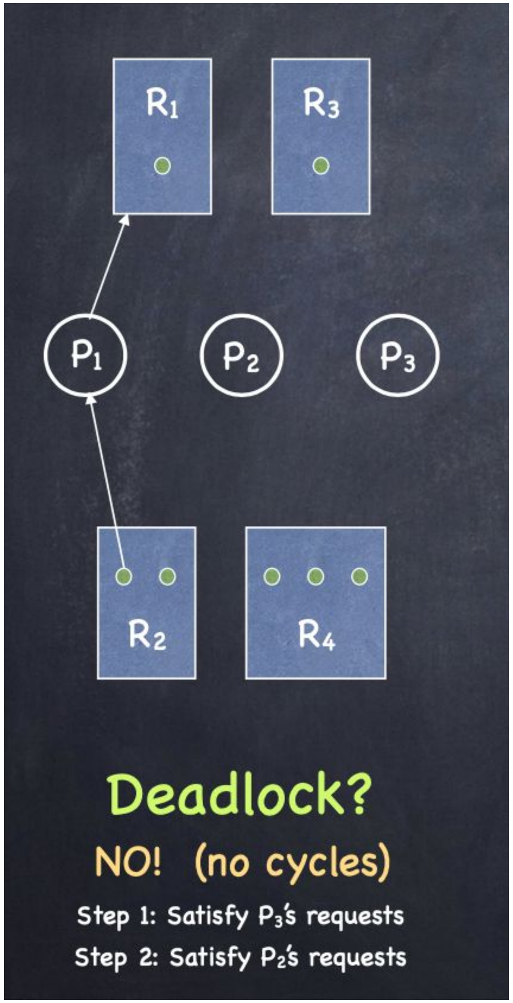
Example 1: RAG Reduction



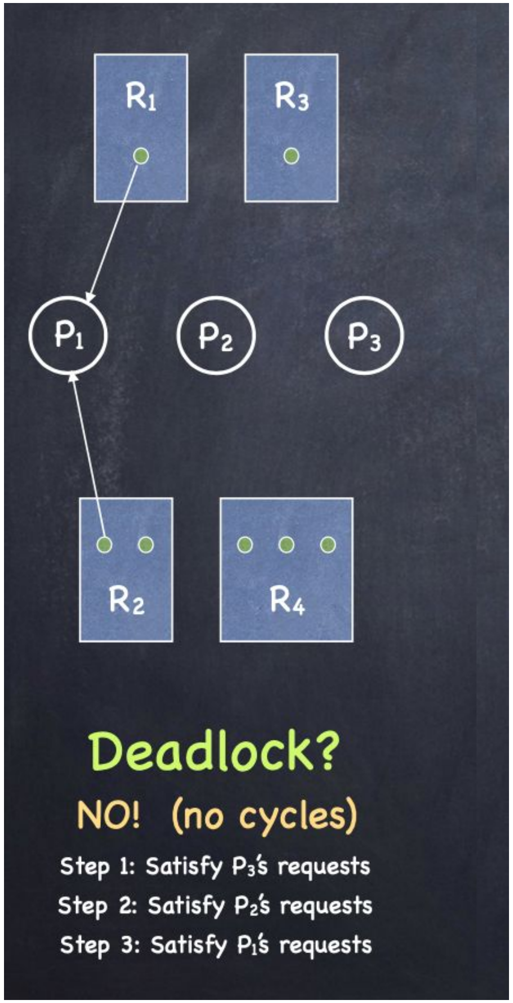
Example 1: RAG Reduction



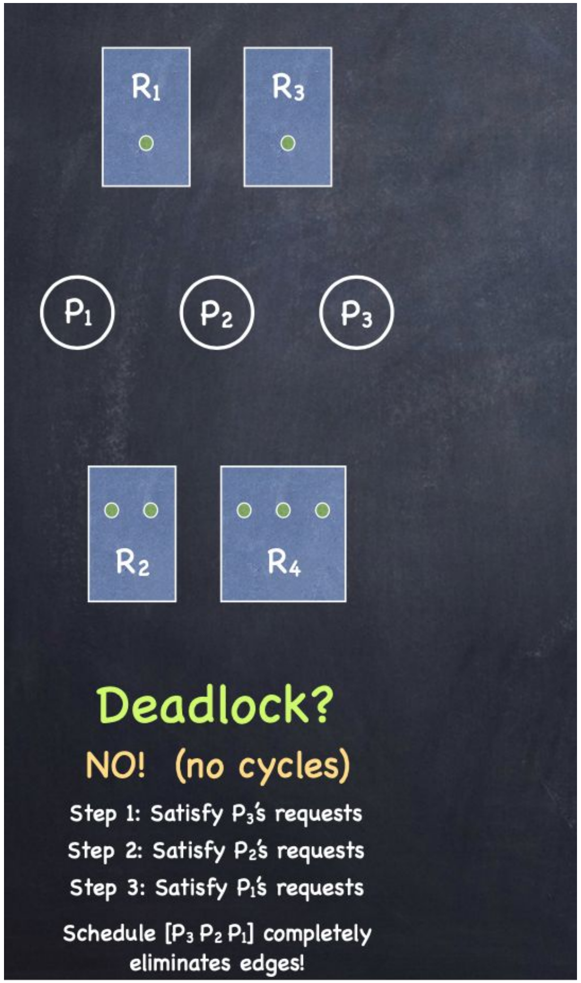
Example 1: RAG Reduction



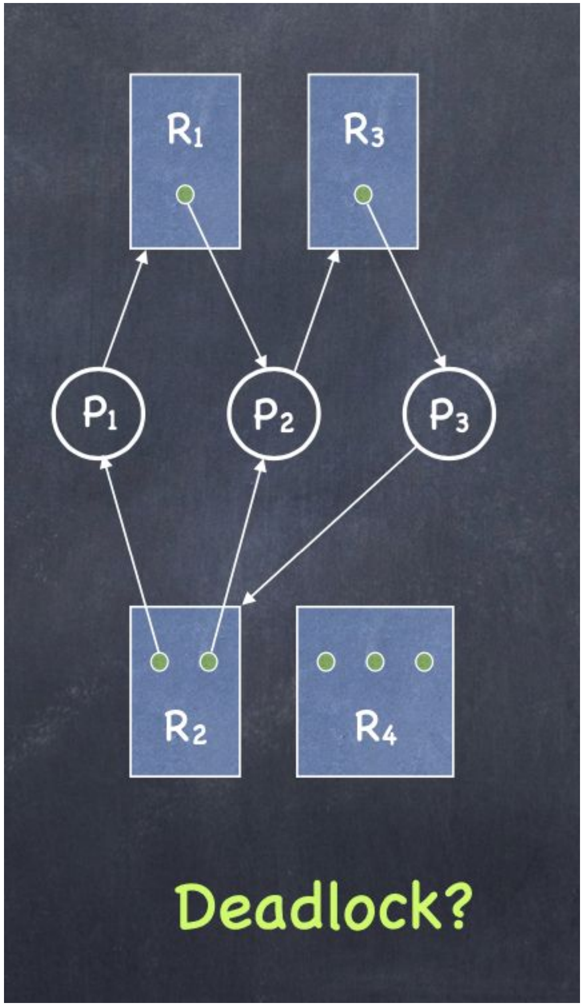
Example 1: RAG Reduction



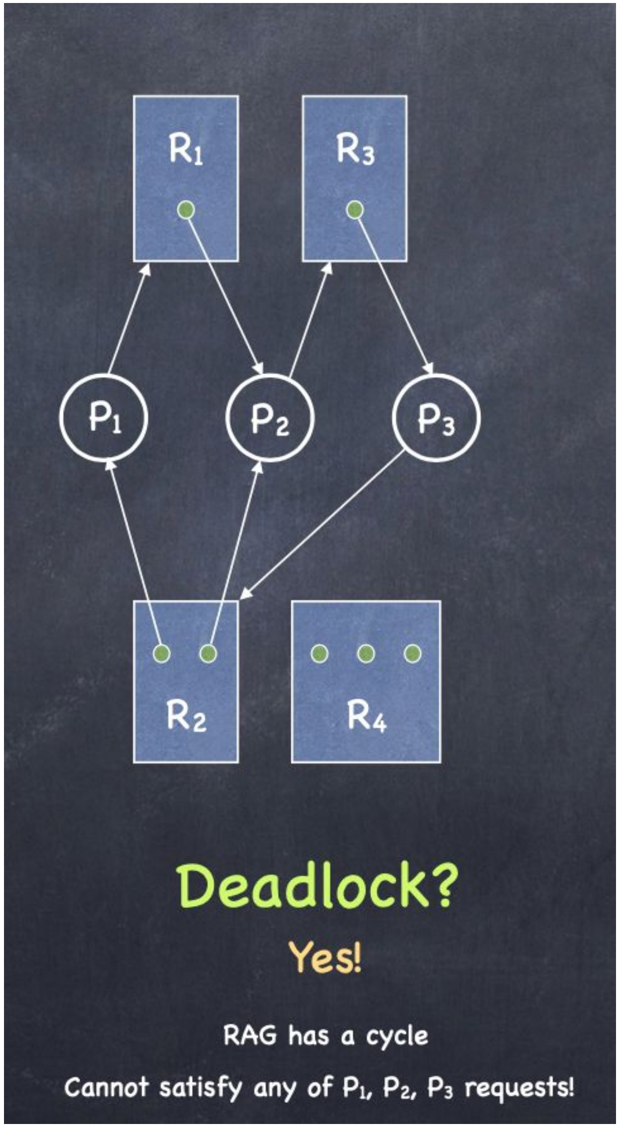
Example 1: RAG Reduction



Example 2: RAG Reduction



Example 2: RAG Reduction



Deadlock can be deadly!

- Does a deadlock disappear on it’s own?

Lecture Summary

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - Bounded Resources
  - 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 threads
  - Circular waiting
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Next class: Techniques for addressing deadlocks

The Dining Philosophers Problem

- Five chopsticks, five philosophers
  - Goal: Grab two chopsticks to eat
- **Deadlock** if they all grab chopstick to their right
- **How to fix deadlock?**
  - Make one of them give up a chopstick
- **How to prevent deadlock?**
  - Never take last chopstick if no hungry lawyer has two afterward