

EECS 482: Introduction to Operating Systems

Lecture 10: Deadlock

Prof. Ryan Huang

Honor code reminder

Using ChatGPT (or other AI tools) for help specific to 482 projects is an honor code violation

- Generate code, comments, test cases, verifiers

The consequences are serious

We were relatively lenient on P1 but will actively deal with violations going forward

- we have taken actions already for P1

Deadlock

Interviewer to EECS 482 student: *"Explain deadlock to me, and we'll give you the job"*.

EECS 482 student to interviewer: *"Give me the job, and I'll explain deadlock to you"*.

Deadlock

So far, we have made programs correct by **constraining schedules**

- Allow only correct interleavings

But, also possible to **over-constrain** schedules

- A must happen before B
- B must happen before A
- Result is called **deadlock**

Two types of correctness

- **Safety**: all actions that occur must be correct
- **Liveness**: actions must keep occurring. Deadlock violates liveness.

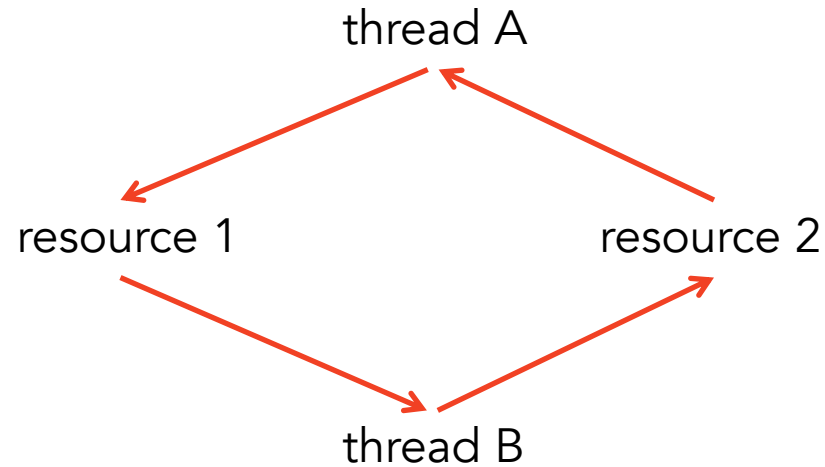
Deadlock

Resources

- Things that a thread needs and may need to wait for
- Examples: locks, disk space, memory, CPU

Deadlock: a cyclical waiting for resources

Wait-for graph



Cycle represents a deadlock

Traffic example



Class example

Alice is in EECS 482, Bob is in EECS 485, and they want to switch

Resources are seats in class

Both Alice and Bob will wait forever

Deadlock always leads to starvation of the threads involved

Is starvation always caused by deadlock?

Lock example

Thread A

`x.lock()`

`y.lock()`

`...`

`y.unlock()`

`x.unlock()`

Thread B

`y.lock()`

`x.lock()`

`...`







`x.unlock()`

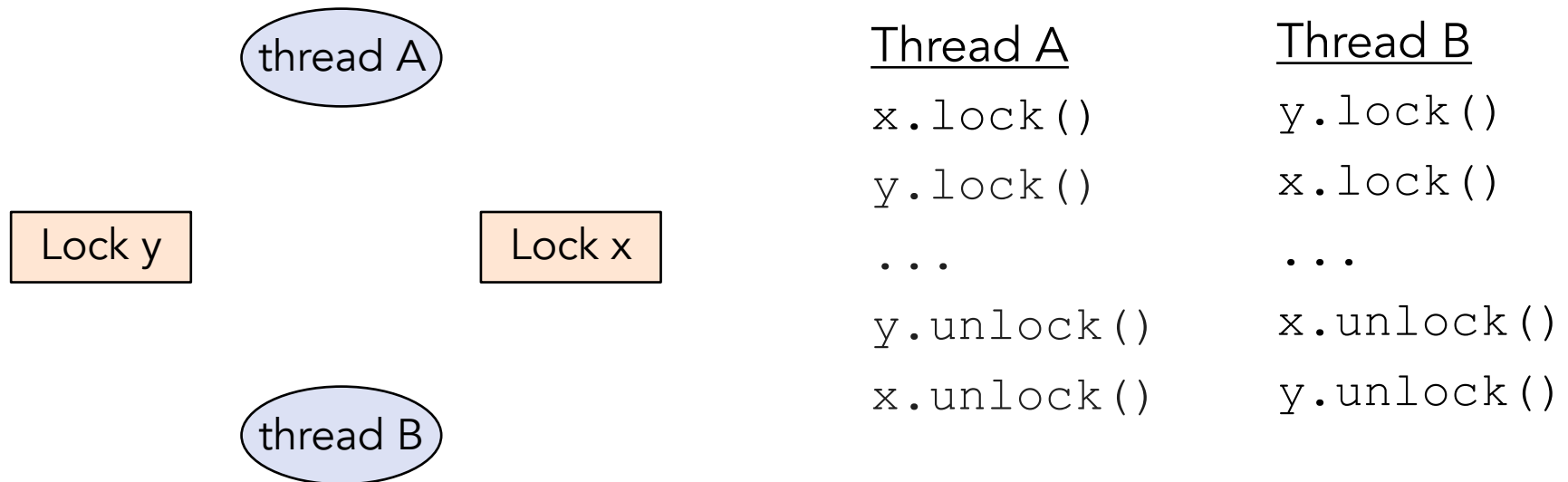
`y.unlock()`

Can deadlock occur?

Will deadlock always occur?

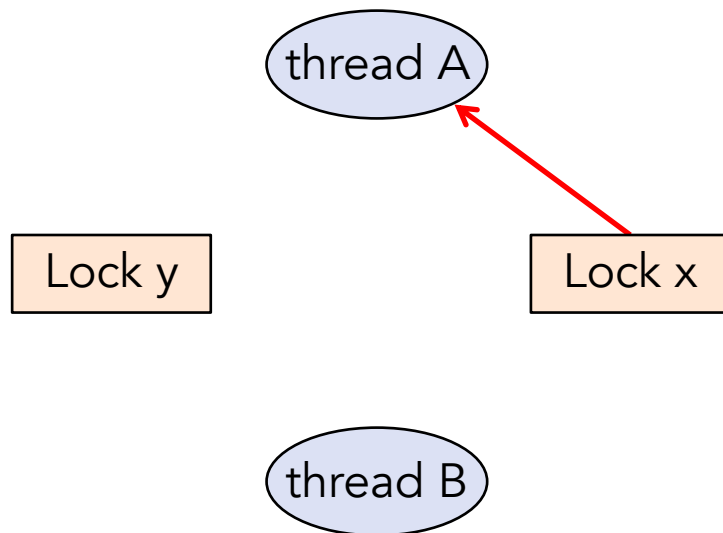
Wait-for graph

- Vertices: threads , resources 
- Edges:  \rightarrow  (x waits for y),  \rightarrow  (y holds x),



Cycle represents a deadlock

Wait-for graph



Thread A

x.lock()

y.lock()

...

y.unlock()

x.unlock()

Thread B

y.lock()

x.lock()

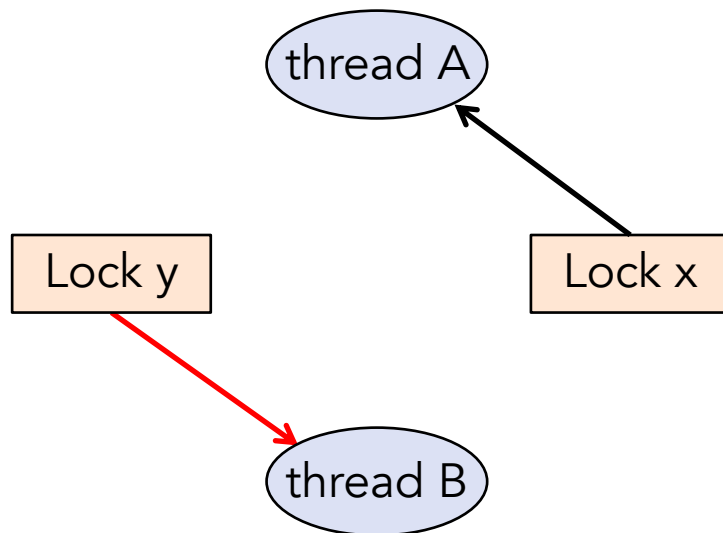
...

x.unlock()

y.unlock()

Cycle represents a deadlock

Wait-for graph



Thread A

`x.lock()`

`y.lock()`

...

`y.unlock()`

`x.unlock()`

Thread B

`y.lock()`

`x.lock()`

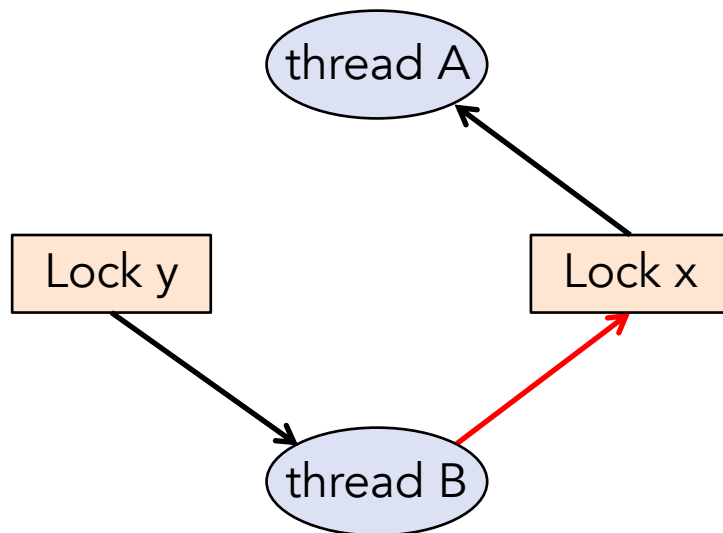
...

`x.unlock()`

`y.unlock()`

Cycle represents a deadlock

Wait-for graph



Thread A

`x.lock()`

`y.lock()`

...

`y.unlock()`

`x.unlock()`

Thread B

`y.lock()`

`x.lock()`

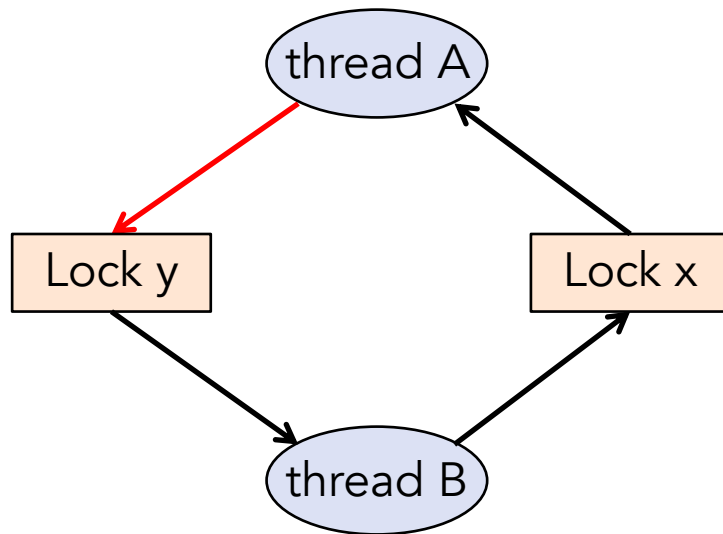
...

`x.unlock()`

`y.unlock()`

Cycle represents a deadlock

Wait-for graph



Thread A

`x.lock()`

`y.lock()`

...

`y.unlock()`

`x.unlock()`

Thread B

`y.lock()`

`x.lock()`

...

`x.unlock()`

`y.unlock()`

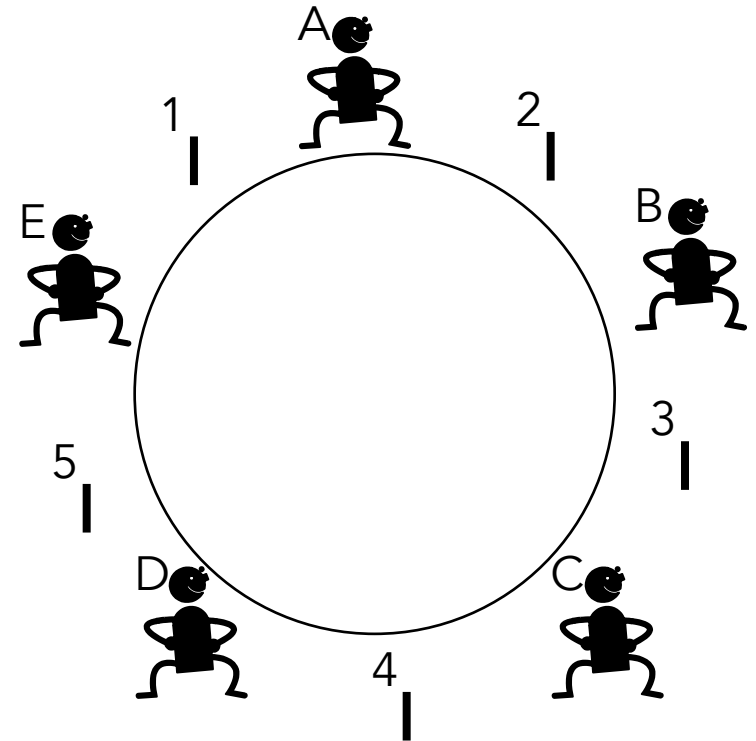
Cycle represents a deadlock

Dining philosophers

5 philosophers sit at round table

1 chopstick between each pair of philosophers

Each philosopher needs 2 chopsticks to eat



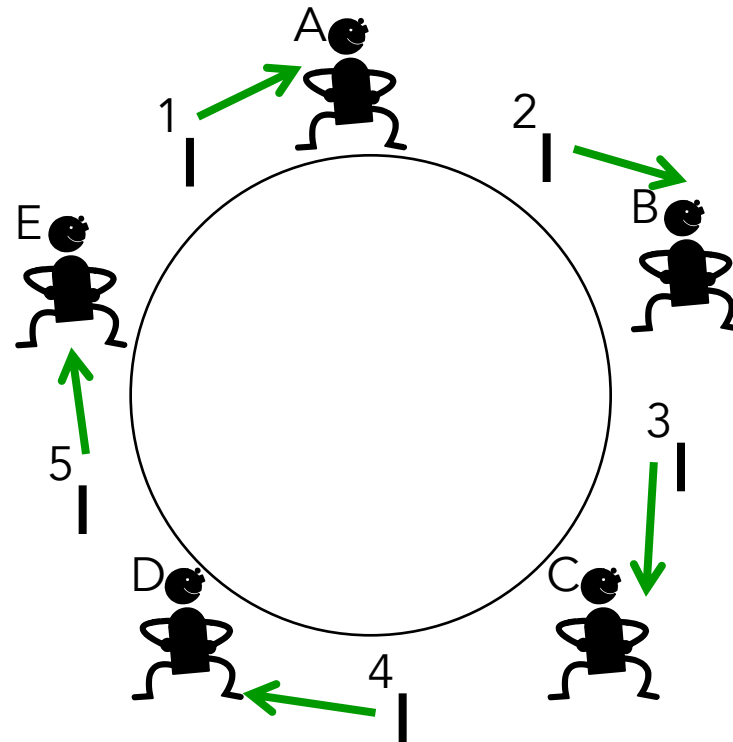
Dining philosophers

Algorithm for each philosopher:

```
while (true) {  
    think()  
    wait for chopstick on right to be free  
    pick up chopstick on right  
    wait for chopstick on left to be free  
    pick up chopstick on left  
    eat()  
    put both chopsticks down  
}
```

Can this deadlock?

Dining philosophers



Generic example of multi-threaded program

phase 1

```
while (!done) {  
  
    acquire some resource  
  
    work  
  
}
```

phase 2

```
release all resources
```

What to do about deadlocks

Ignore

- Typical OS strategy for application deadlocks
- Deadlocked threads consume no CPU time

What to do about deadlocks

Ignore

Detect and fix

- Detect: look for cycle in wait-for graph
- Fix #1
 - Kill one of the threads involved in the deadlock
 - What about broken invariants?
- Fix #2
 - Roll back actions of one of the threads, then try again
 - This fixes broken invariants
 - Is it always possible to roll back a thread?

What to do about deadlocks

Ignore

Detect and fix

- Detect by looking for cycle in wait-for graph
- Fix by killing or rolling back one or more threads

Prevent

- Think about what conditions are necessary for deadlock to occur

Four necessary conditions for deadlock

Limited resource

- Not enough to serve all threads simultaneously
- Unlimited resource → no waiting for that resource

No preemption

- Can't force threads to give up resources
- Preemption → no indefinite waiting

Hold and wait

- Hold resources while waiting to acquire others
- No hold and wait → no multi-edge path in wait-for graph

Cycle of hold-wait requests

- No cyclical chain → no cycle in wait-for graph

How to eliminate limited resources?

Increase # of resources

- E.g., buy more machines
- E.g., increase # of locks?

How to eliminate no preemption?

Enable preemption

- Preempt CPU
- Preempt memory (next lecture topic)
- Preempt lock?

How to eliminate hold and wait?

First step:

- Move resource acquisition to beginning

```
phase 1          all needed
                       
                       
while (!done) {
    acquire some resource
    work
}
```

```
phase 2
    release all resources
```

How to eliminate hold and wait?

Second step

- Option A: Acquire all needed resources **atomically**

phase 1

```
acquire all needed resources atomic
```

```
while (!done) {  
    work  
}
```

phase 2

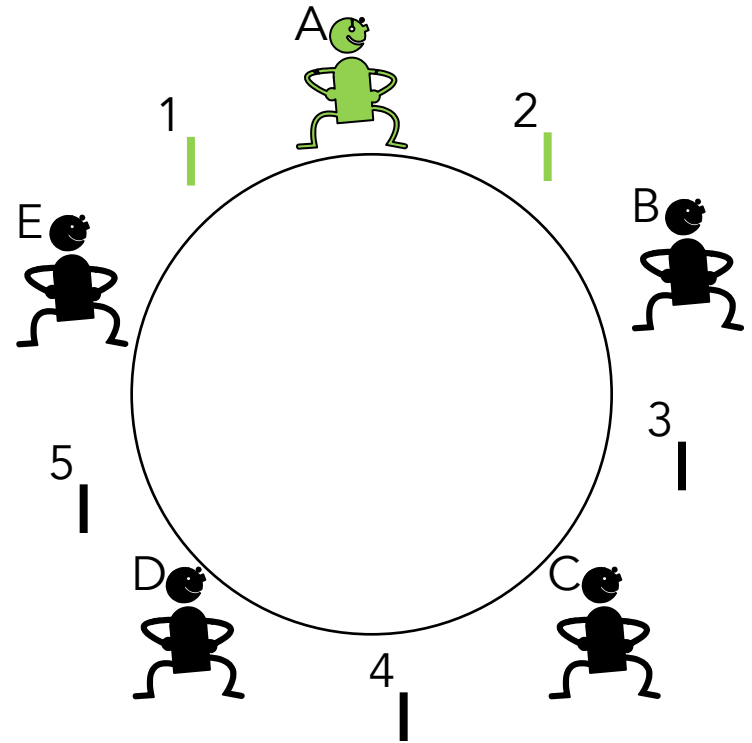
```
release all resources
```

Atomic acquisition

```
L.lock()  
while right chopstick busy or left chopstick busy  
    cv.wait (L)  
pick up right chopstick  
pick up left chopstick  
L.unlock()  
<eat>  
L.lock()  
drop left chopstick  
drop right chopstick  
cv.broadcast()  
L.unlock()
```

Dining philosophers

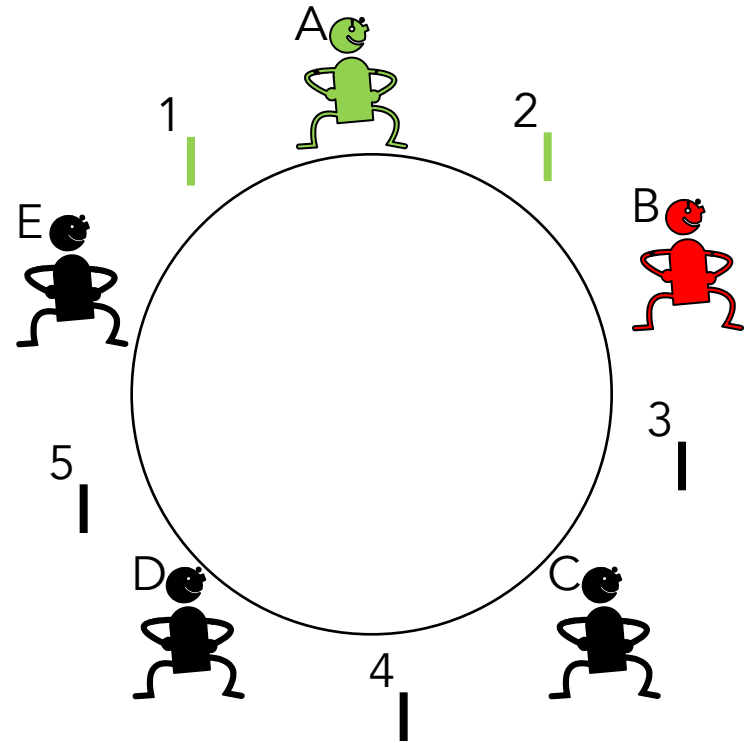
A starts to eat



Dining philosophers

A starts to eat

B waits

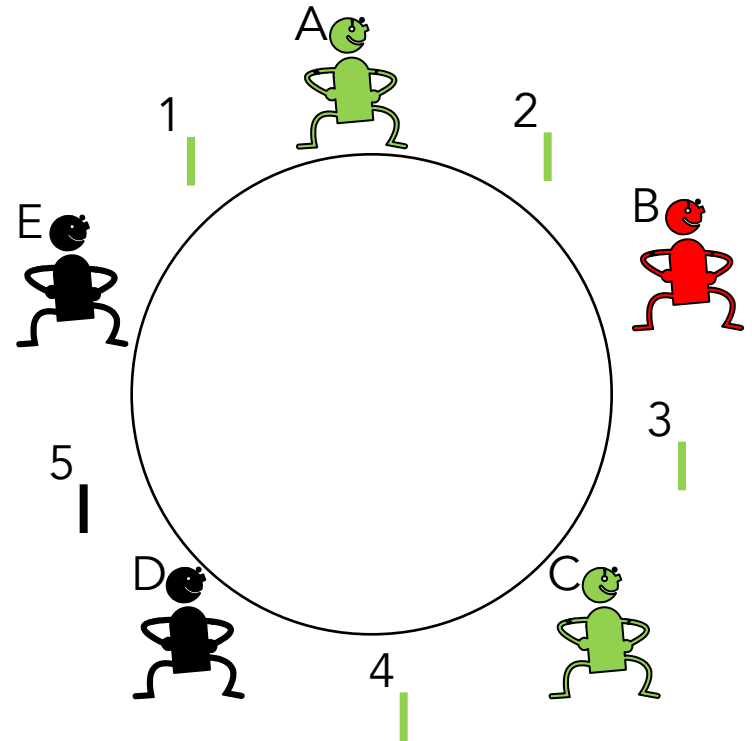


Dining philosophers

A starts to eat

B waits

C starts to eat



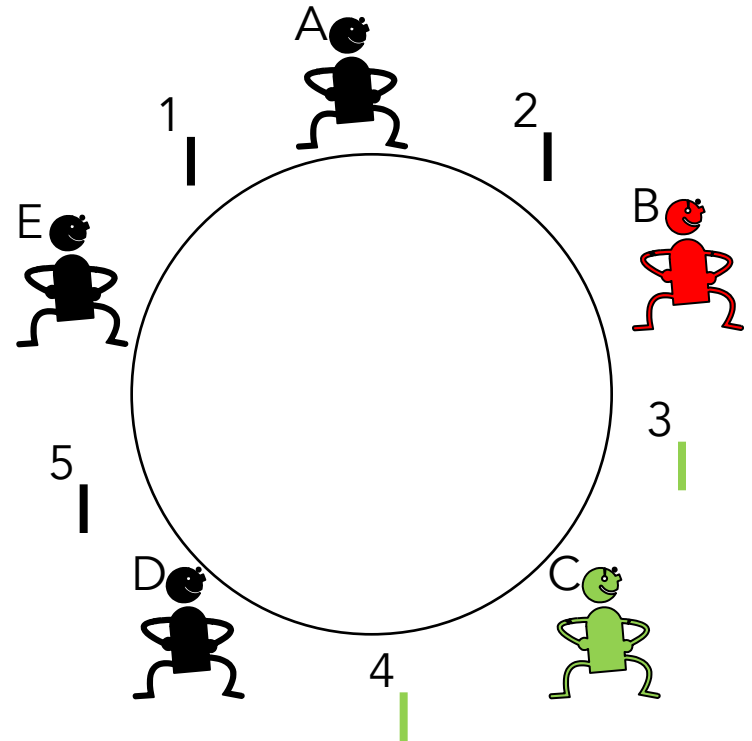
Dining philosophers

A starts to eat

B waits

C starts to eat

A finishes



Dining philosophers

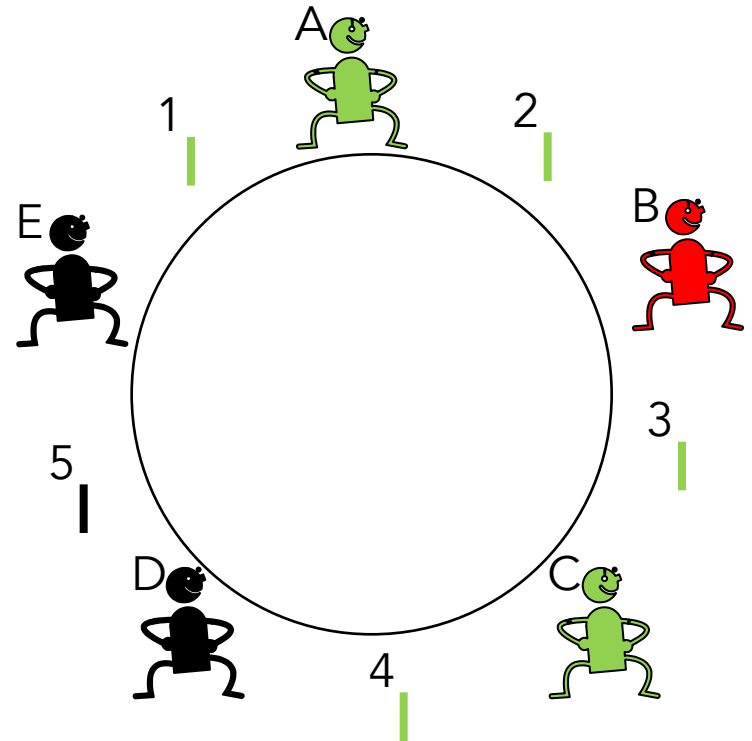
A starts to eat

B waits

C starts to eat

A finishes

A starts to eat



Dining philosophers

A starts to eat

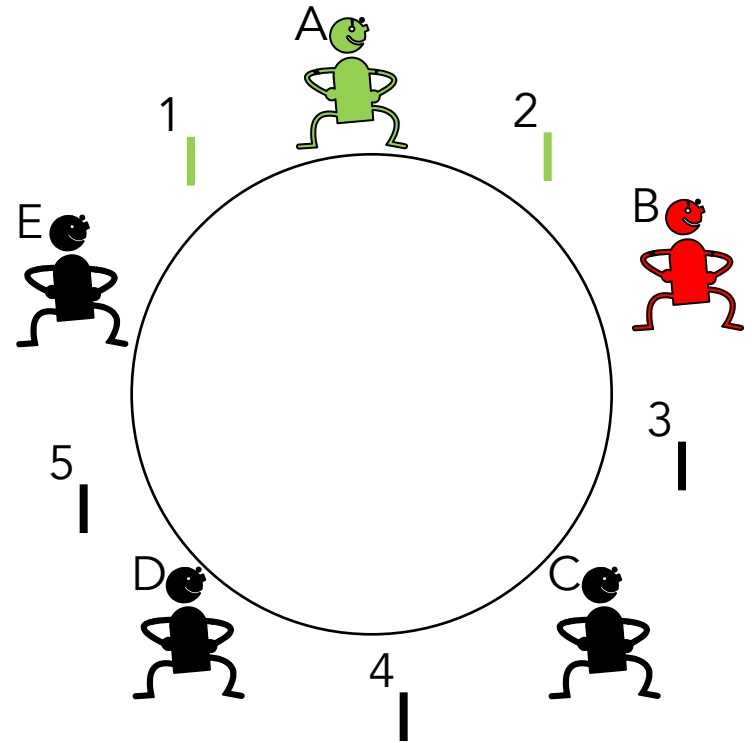
B waits

C starts to eat

A finishes

A starts to eat

C finishes



Dining philosophers

A starts to eat

B waits

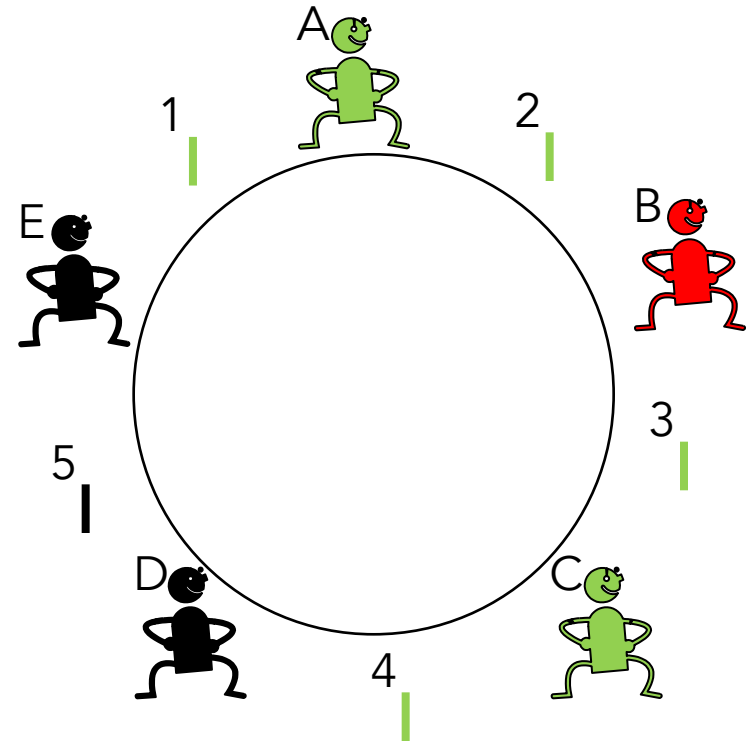
C starts to eat

A finishes

A starts to eat

C finishes

C starts to eat



Dining philosophers

A starts to eat

B waits

C starts to eat

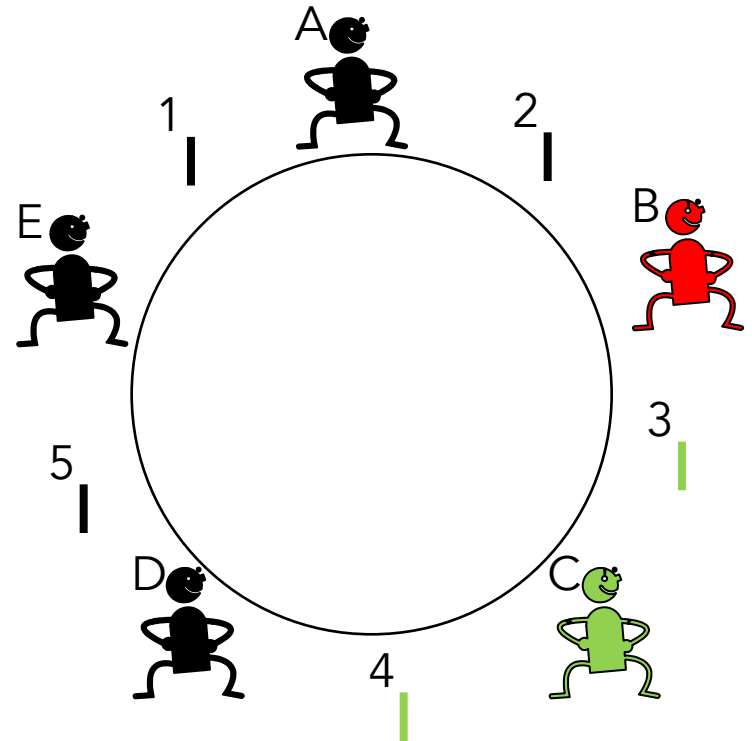
A finishes

A starts to eat

C finishes

C starts to eat

A finishes



Dining philosophers

A starts to eat

B waits

C starts to eat

A finishes

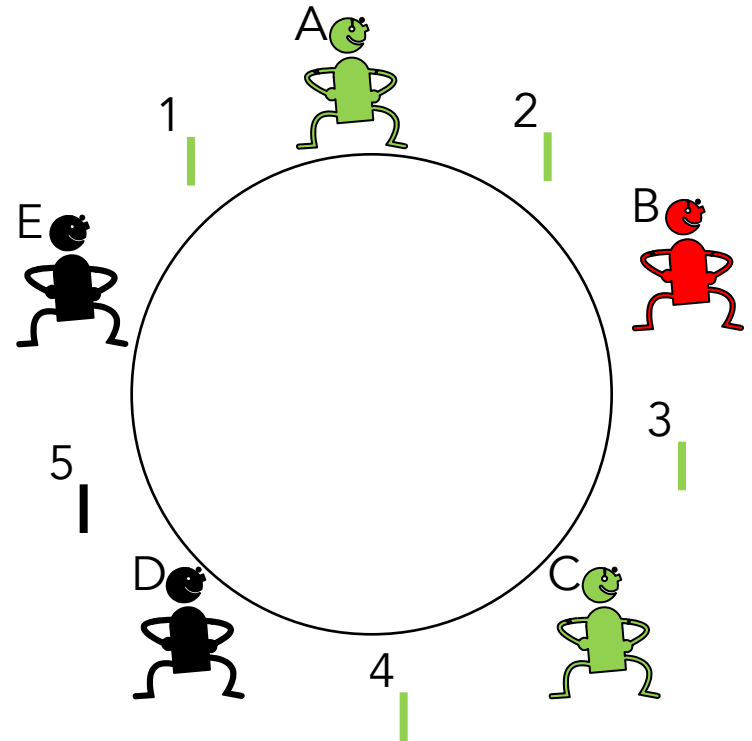
A starts to eat

C finishes

C starts to eat

A finishes

A starts to eat



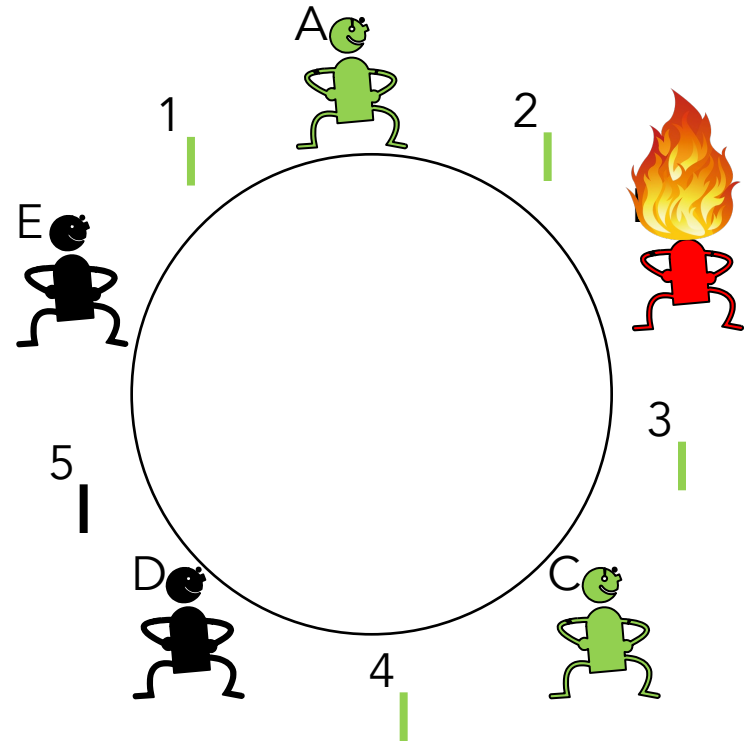
Dining philosophers

A starts to eat

B waits

while (1)

- C starts to eat
- A finishes
- A starts to eat
- C finishes



How to eliminate hold and wait?

Second step

- Option A: Acquire all needed resources **atomically**
- Option B: Release and retry if you encounter a busy resource

phase 1

```
acquire all needed resources
while (!done) {
    work
}
```

phase 2

```
release all resources
```

How to eliminate hold and wait?

What's the problem with this style of solution?

phase 1

```
acquire all needed resources
while (!done) {
    work
}
```

phase 2

```
release all resources
```

How to eliminate cycle of hold-and-wait?

Define a **global** order over all resources

- All threads **must** follow this order when acquiring resources
- Guarantees that some thread can always make progress

Thread A

`x.lock()`

`y.lock()`

`...`

`y.unlock()`

`x.unlock()`

Thread B

`y.lock()`

`x.lock()`

`...`

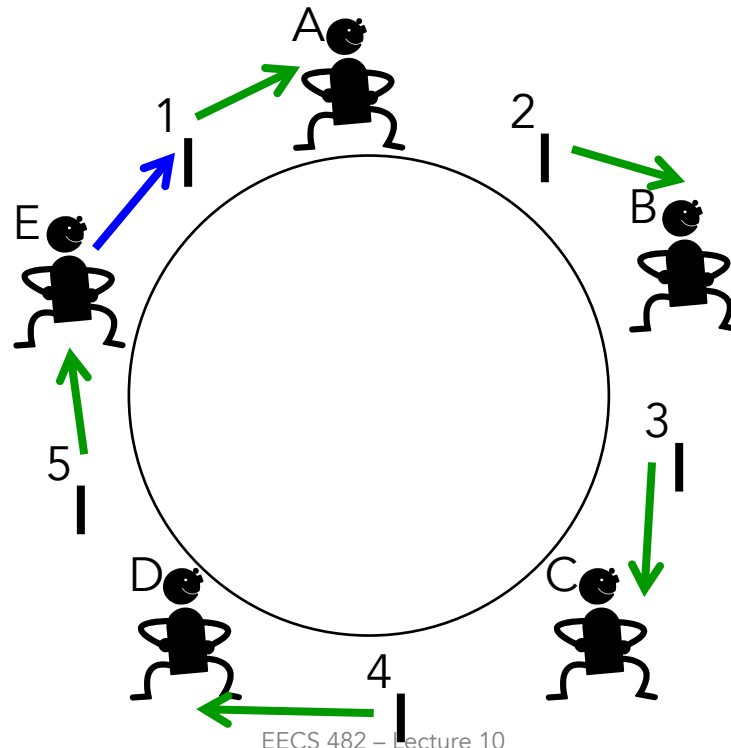
`x.unlock()`

`y.unlock()`

Dining philosophers

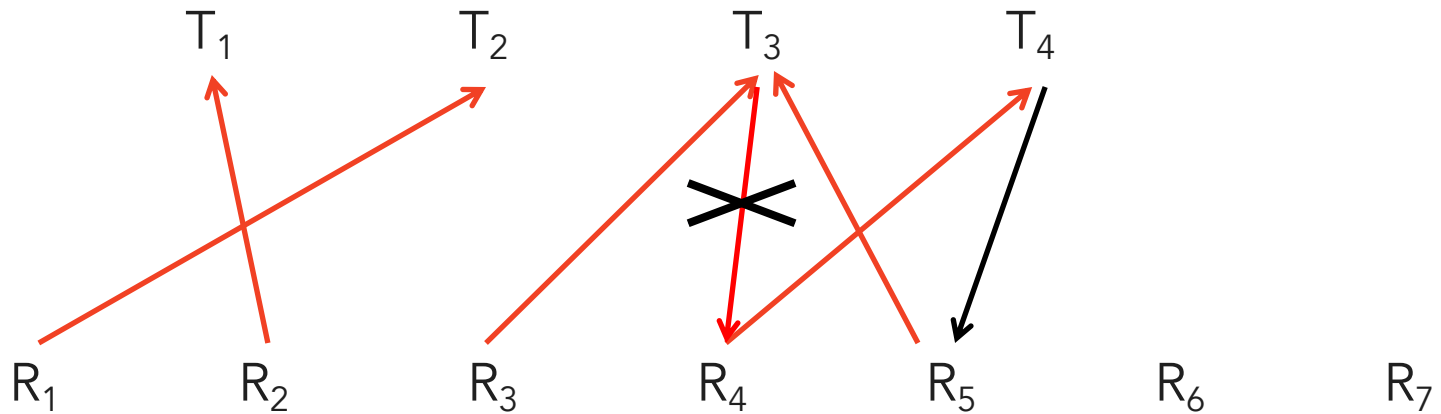
Pick up lower # chopstick first

Pick up higher # chopstick second



Global ordering of resources

How can we be sure that *some* thread can make progress?



Which thread can make progress?