EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Recap

- How to leverage hardware support to implement high-level synchronization primitives?

- For mutual exclusion inside critical section:
  - Disable interrupts to prevent context switches
  - test_and_set(guard) for atomicity across CPUs
- To wait inside critical section:
  - Add self to waiting queue and switch to next ready thread or suspend CPU

Switch invariant

- Before switching to another thread
  - Disable interrupts and acquire guard
- When call to swapcontext returns, can assume
  - Interrupts disabled and guard acquired
- Before returning to user-level code
  - Release guard and enable interrupts

Lock implementation #4

```
//guard is initialized to 0
lock() {
       while (test_and_set(guard)) {}
       disable interrupts

       if (status == FREE) {
              status = BUSY
       } else {
              add thread to queue of threads waiting for lock
              switch to next ready thread
       }
       guard = 0
       enable interrupts
}
```

Project 2

- Work on the project incrementally
- CPU and thread
  - 1 CPU, no interrupts
  - 1 CPU + interrupts
- Implement mutex and cv
- Add support for multiple CPUs

- Due in 12 days!

Constraining schedules

- So far, we have made programs correct by constraining schedules
  - Allow only correct orderings
  - Maximize concurrency

- But, also possible to over-constrain schedules
  - A must happen before B
  - B must happen before A
  - **Deadlock** is a common result of over-constraint

Deadlock

- Resources
  - Things needed by a thread that it waits for
  - Examples: locks, disk space, memory, CPU
- Deadlock
  - Cyclical waiting for resources which prevents progress
  - Results in starvation: threads wait forever
- Example: Swapping classes
  - Alice is in 482, Bob is in 484, and they want to switch

Class example

- Resources are seats in class

- Both Alice and Bob wait forever
  - Deadlock always leads to starvation
  - Not all starvation is deadlock (e.g., R/W lock)

- Not all threads are starved
  - Other students can add/drop other classes

Deadlock example

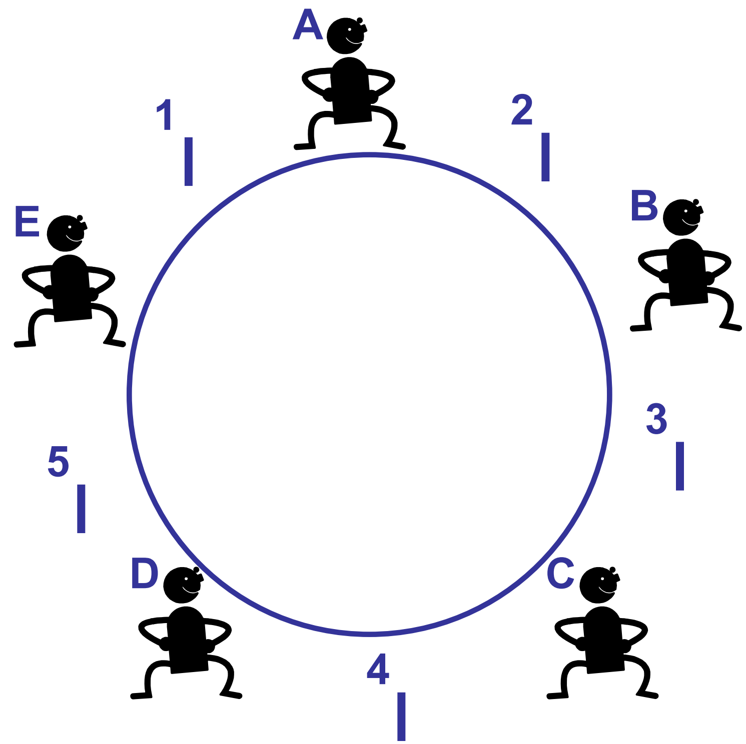
Thread A	Thread B
x.lock	y.lock
y.lock	x.lock
...	...
y.unlock	x.unlock
x.unlock	y.unlock

- Will a deadlock always occur?


BANCO REAL
11455

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 philosopher:

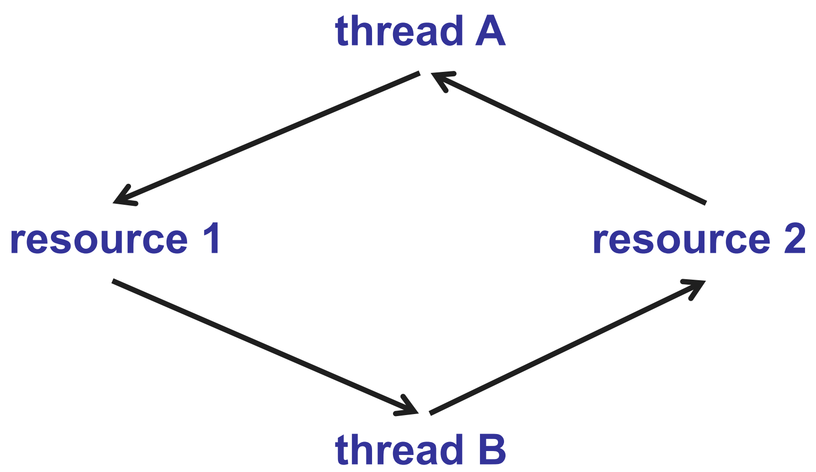
```
1. wait for chopstick on right to be free
2. pick up chopstick on right
3. wait for chopstick on left to be free
4. pick up chopstick on left
5. put both chopsticks down
```

- Can this deadlock?

Generic example of multi-threaded program

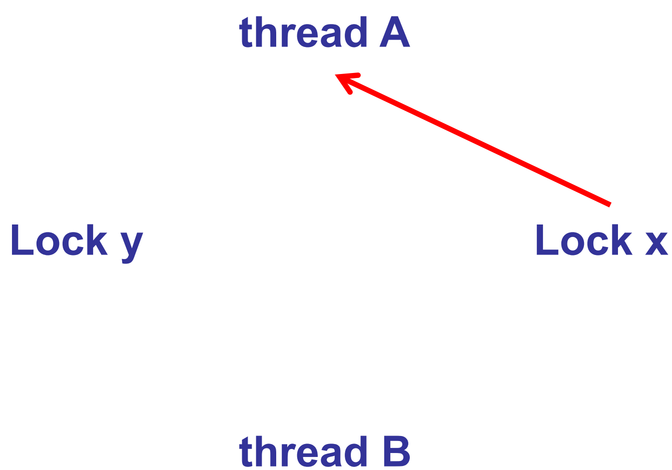
```
phase 1:
while (!done) {
      acquire some resource
      work
}

phase 2:
release all resources
```

Waits-for graph



- Cycle represents a deadlock

Waits-for graph



Cycle represents a deadlock

<u>Thread A</u>

x.lock()

y.lock()

...

y.unlock()

x.unlock()

<u>Thread B</u>

y.lock()

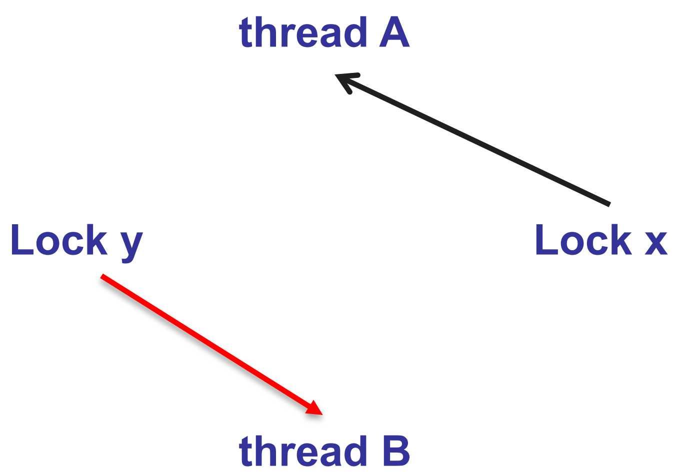
x.lock()

...

x.unlock()

y.unlock()

Waits-for graph



Cycle represents a deadlock

Thread A

x.lock()

y.lock()

...

y.unlock()

x.unlock()

Thread B

y.lock()

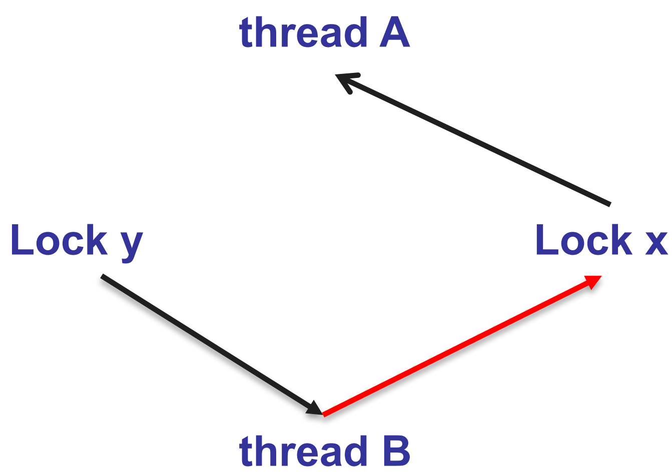
x.lock()

...

x.unlock()

y.unlock()

Waits-for graph



Cycle represents a deadlock

Thread A

x.lock()

y.lock()

...

y.unlock()

x.unlock()

Thread B

y.lock()

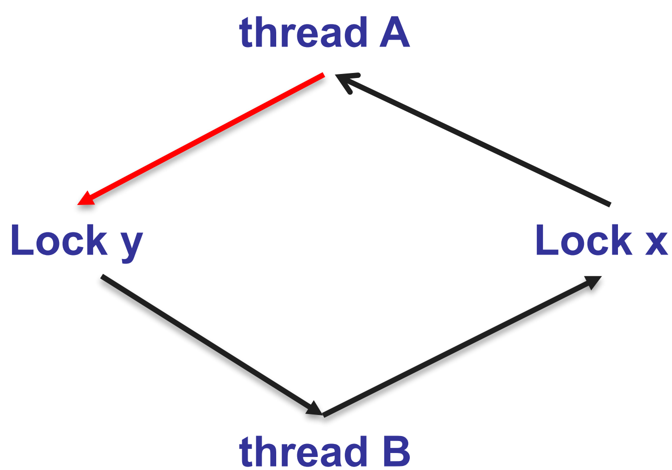
x.lock()

...

x.unlock()

y.unlock()

Waits-for graph



Cycle represents a deadlock

Thread A

x.lock()

y.lock()

...

y.unlock()

x.unlock()

Thread B

y.lock()

x.lock()

...

x.unlock()

y.unlock()

Coping with deadlocks

- Ignore
  - Typical OS strategy for application deadlocks
  - Do deadlocked apps consume CPU?

Coping with deadlocks

- Ignore
- Detect and fix
  - Use waits-for graph to detect
  - How to fix?
  - Could kill threads but not always safe to do so
    - » Invariants can be broken while thread hold lock
  - Databases often rollback work done
    - » General purpose rollback is costly, difficult
- Prevent

Four necessary conditions for deadlock

- Limited resources
  - Not enough to serve all threads simultaneously
- No preemption
  - Can’t force threads to give up resources
- Hold and wait
  - Hold resources while waiting to acquire others
- Cyclical chain of requests

Preventing deadlock

- How to prevent limited resources?
  - Could increase # of resources
  - E.g., buy more machines
  - Not always feasible, e.g., increase # of locks
- How to prevent no preemption?
  - Some resources can be preempted, e.g., CPU
    - » Ensure interrupts enabled
  - Others (e.g., locks) are not preemptable

Midterm exam

- We will have covered all material for midterm by end of this lecture
- Two sample exams posted on web page
  - Review session on Feb 18th or 19th
  - Take a crack in exam setting before then

Four necessary conditions for deadlock

- Limited resources
  - Not enough to serve all threads simultaneously
- No preemption
  - Can’t force threads to give up resources
- Hold and wait
  - Hold resources while waiting to acquire others
- Cyclical chain of requests

Eliminating hold-and-wait

- Two ways to avoid hold and wait:
  - Wait for all resources to be free; grab all atomically
  - If cannot get a resource, release all and start over
- Move resource acquisition to beginning

```
Phase 1a: acquire all resources
Phase 1b: while (!done) {
              work
          }
Phase 2: release all resources
```

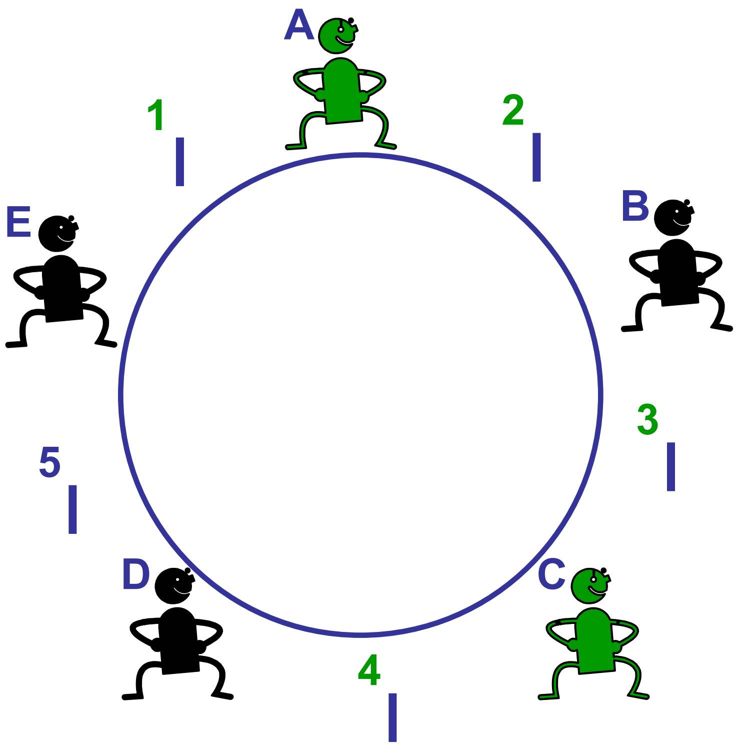
Atomic acquisition

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

Any problems with this solution?

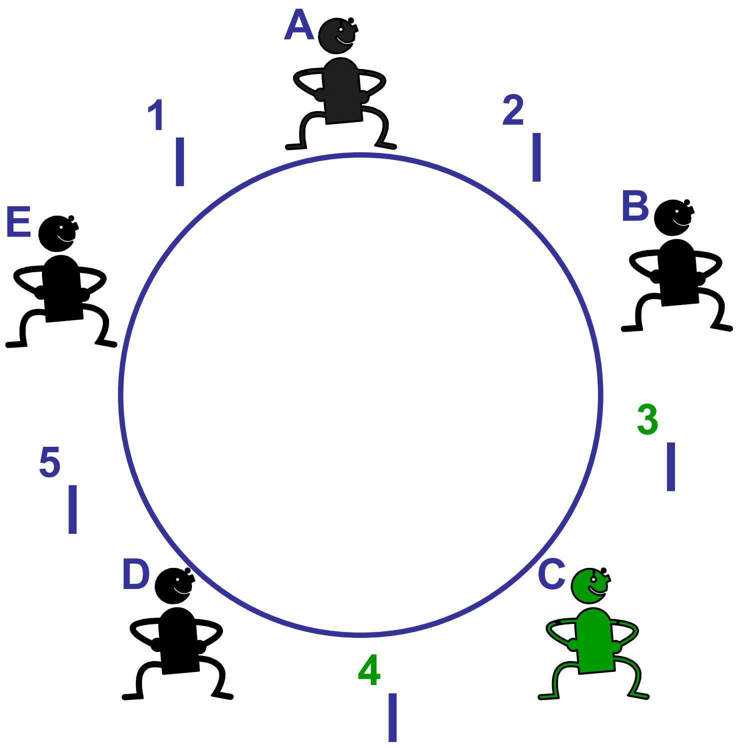
Dining philosophers

- A and C eat



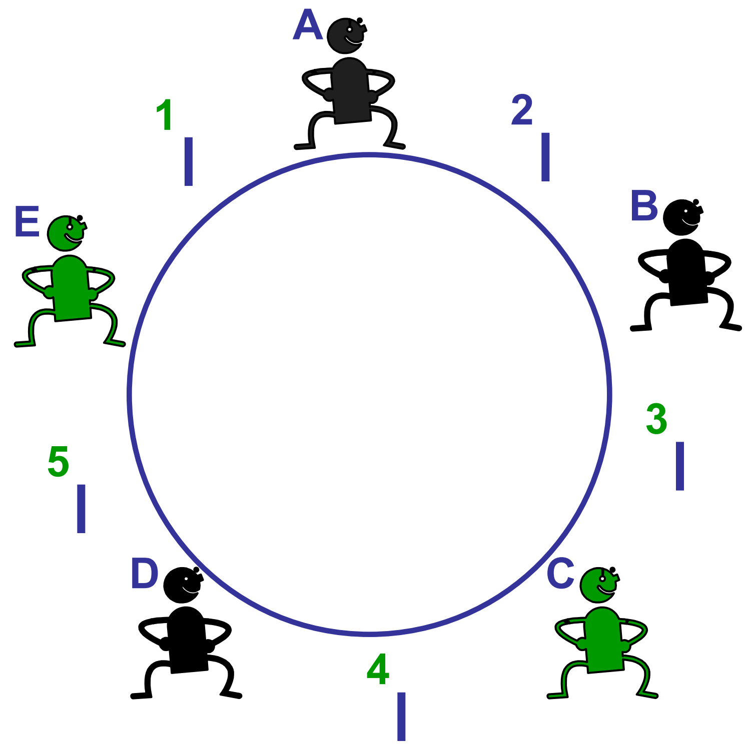
Dining philosophers

- A finishes



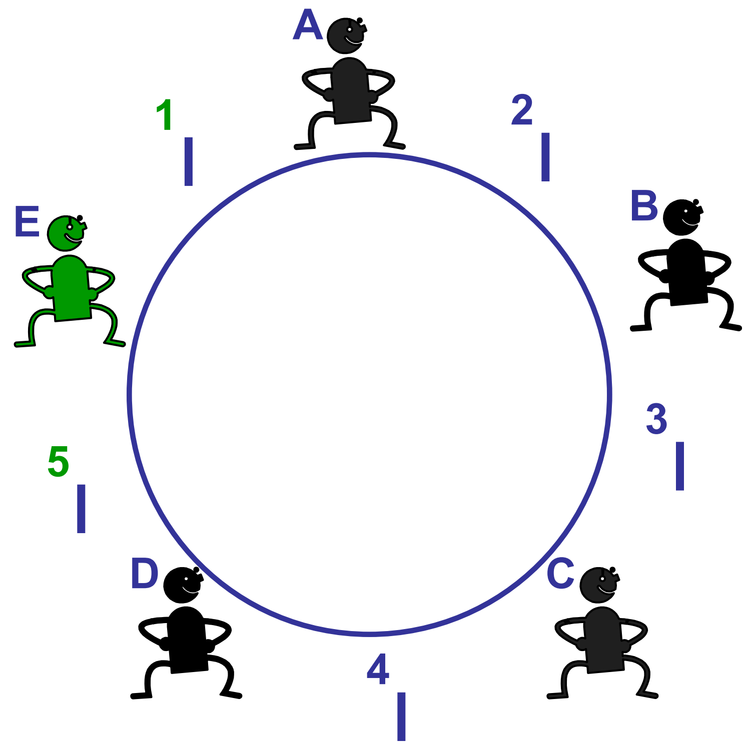
Dining philosophers

- E eats



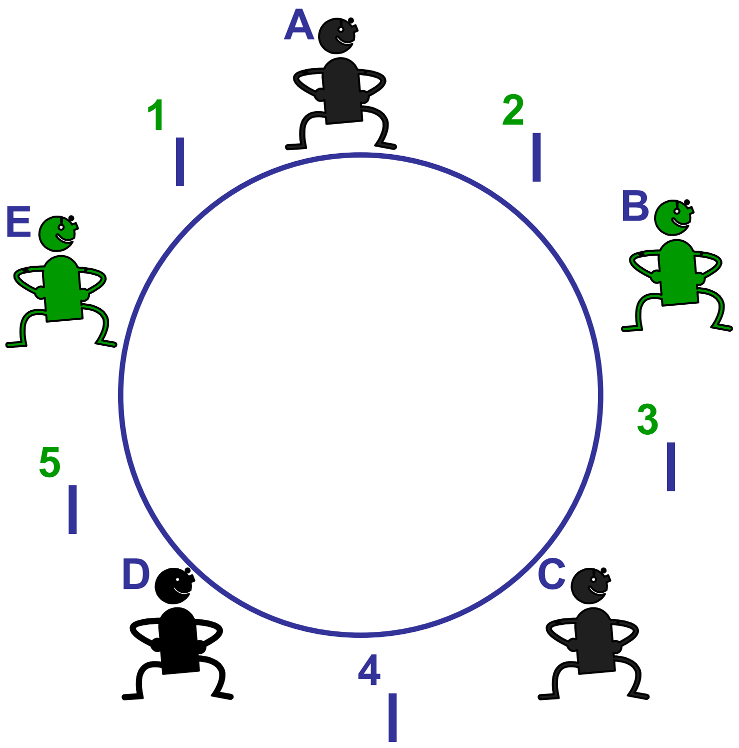
Dining philosophers

- C finishes



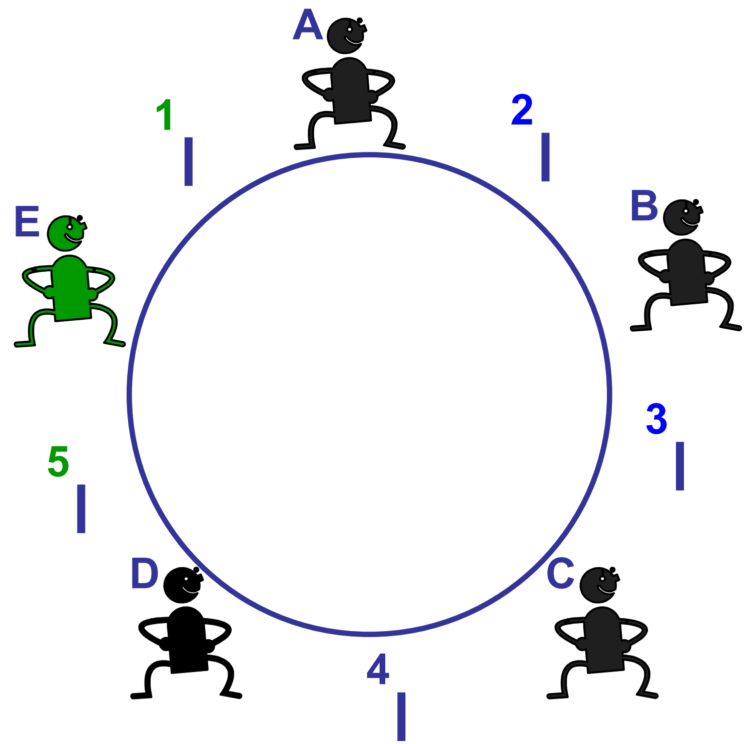
Dining philosophers

- B eats



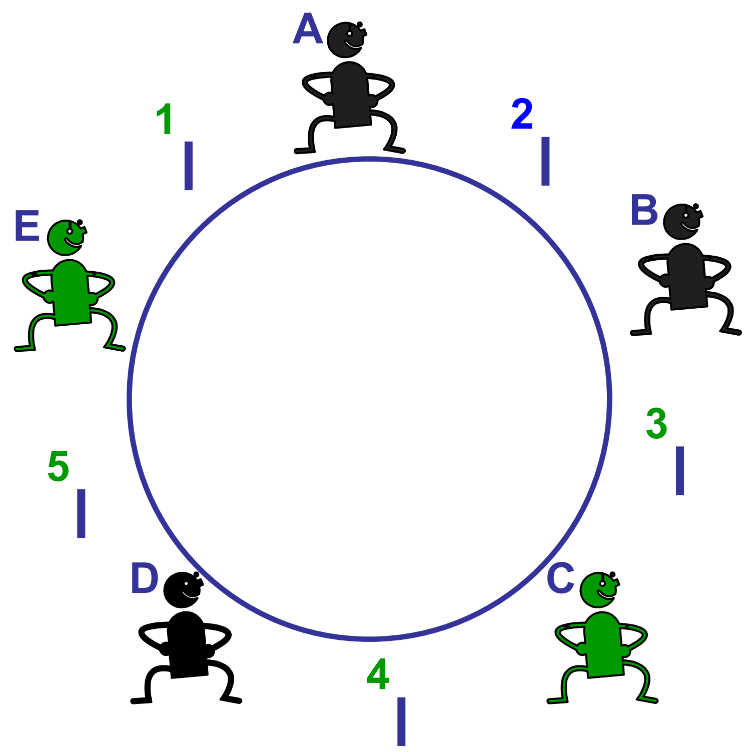
Dining philosophers

- B finishes



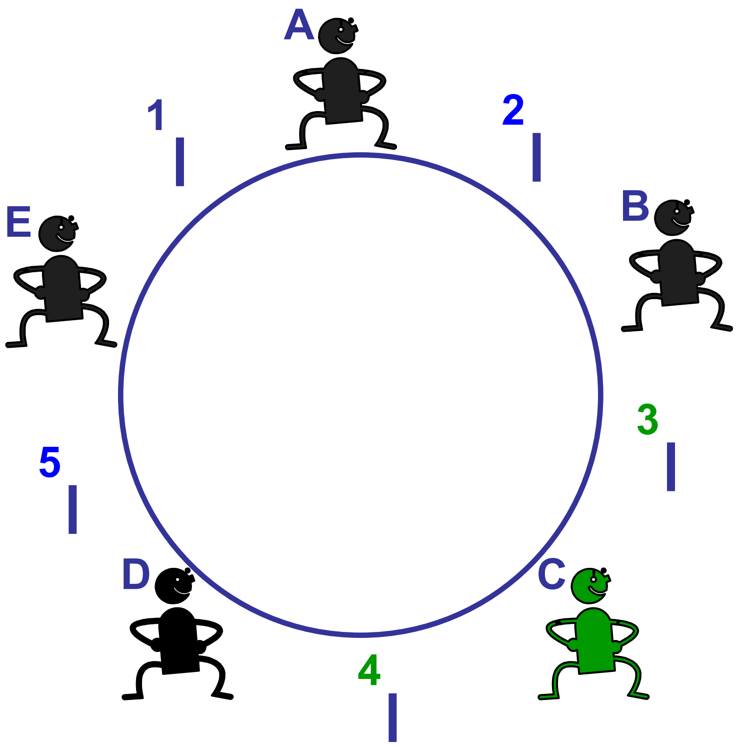
Dining philosophers

- C eats



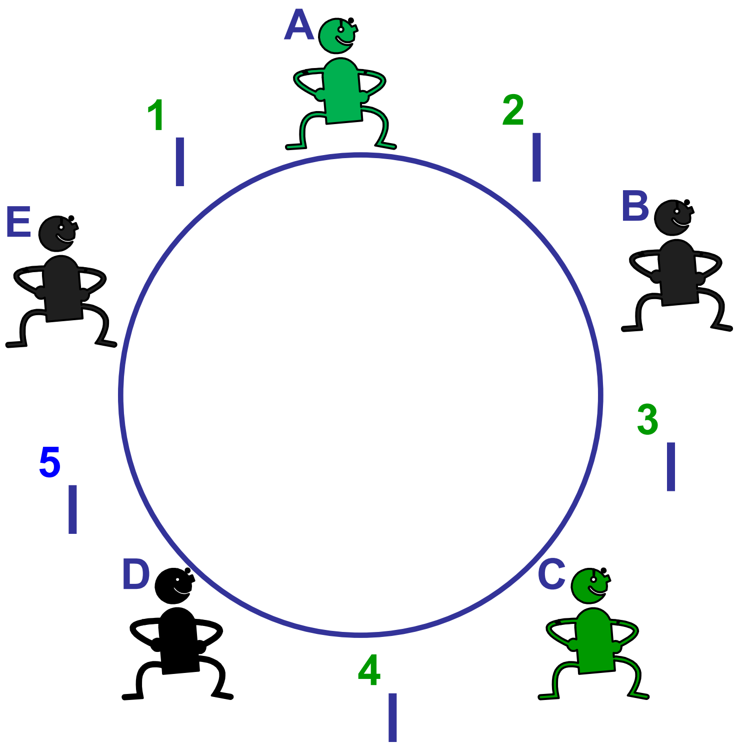
Dining philosophers

- E finishes



Dining philosophers

- A eats
  - Back where we started
  - D starves!



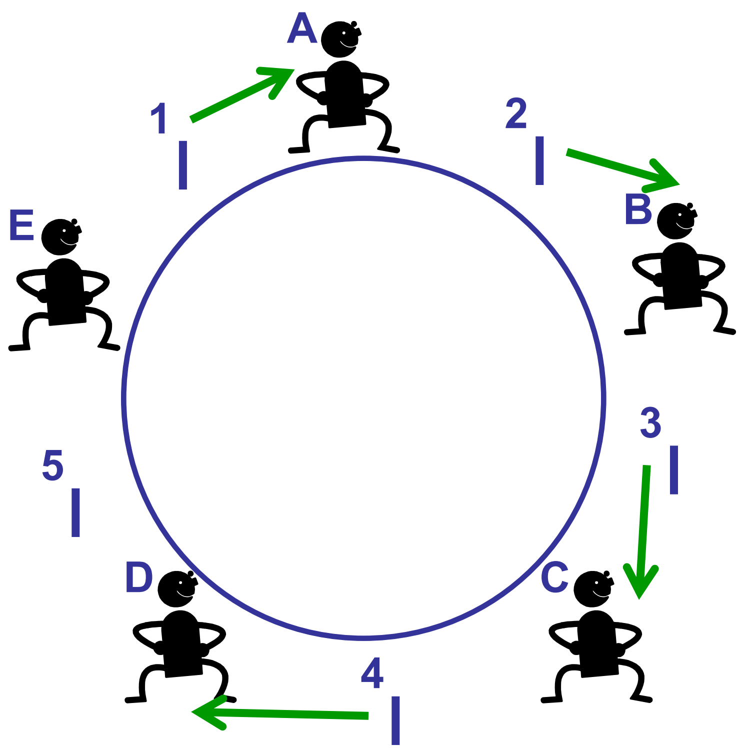
Eliminating circular chain

- Define a global order over all resources
  - All threads acquire resources in this order
  - Thread with highest # resource can make progress

Thread A	Thread B
x.lock()	x.lock()
y.lock()	y.lock()
...	...
y.unlock()	y.unlock()
x.unlock()	x.unlock()

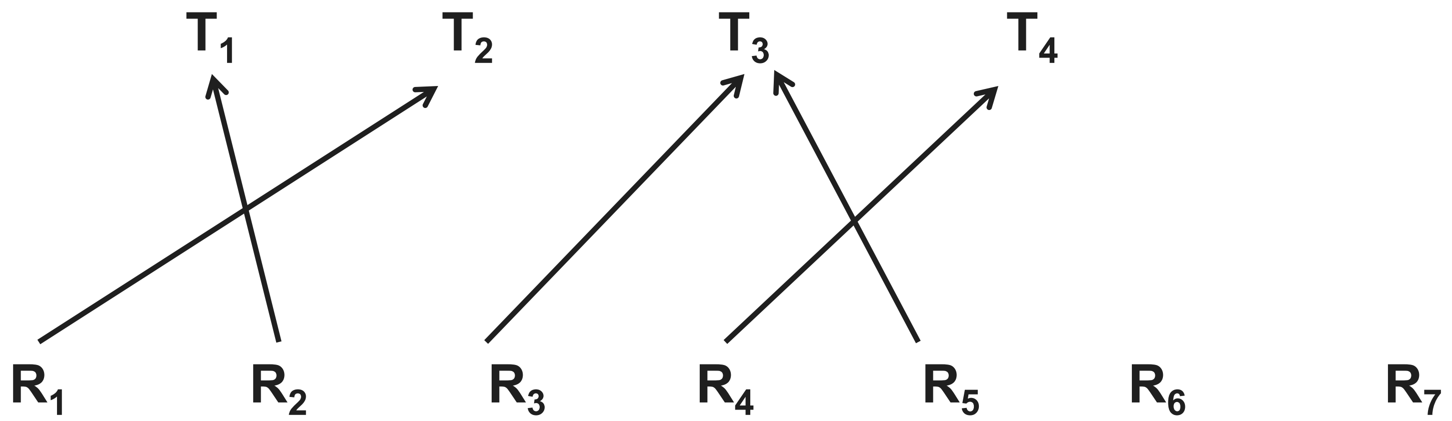
Dining philosophers

- Pick up lower # chopstick first
- Pick up higher # chopstick second



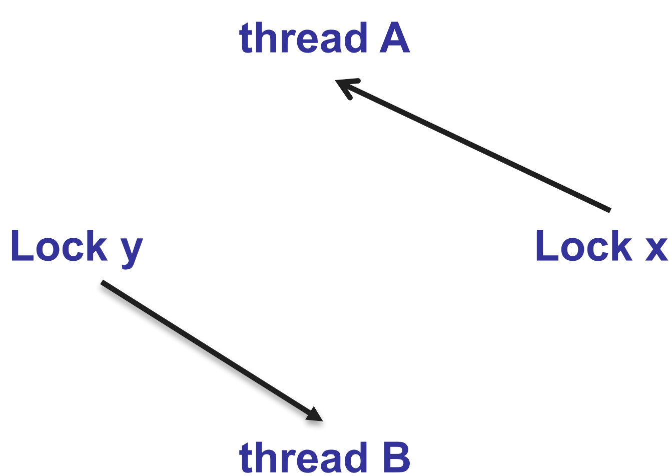
Global ordering of resources

- If every thread acquires resources in order
  - How can we be sure that *some* thread can progress?



Preventing deadlock

- What if we don't grant resources that will lead to cycle in waits-for-graph?



Thread A	Thread B
x.lock()	y.lock()
y.lock()	x.lock()
...	...
y.unlock()	x.unlock()
x.unlock()	y.unlock()

Next time ...

- We’ll move on to how OS abstracts use of memory