

CSE 451: Operating Systems, 26au

L10: Semaphores, Deadlock

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head) Mahir Emran Ted Guan

Jai Rungrachakanont Soham Raut Sagar Saby

Andrew Wood

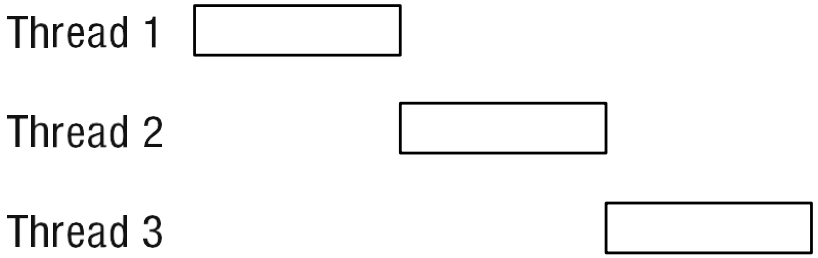
Adapted from slides by Tom Anderson

Administrivia

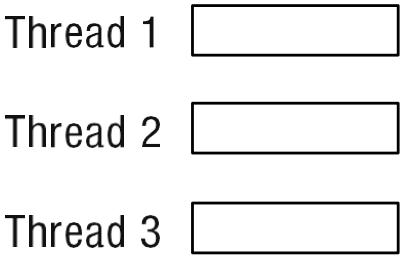
- A note about design docs

Concurrency

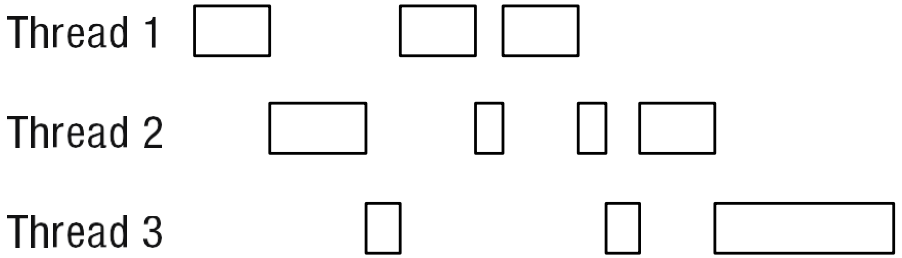
One Execution



Another Execution



Another Execution



An RW lock metaphor



Low-level Synchronization Implementations

(continued from last class)

xk's **kernel spinlock** implementation

- `kernel/spinlock.c`

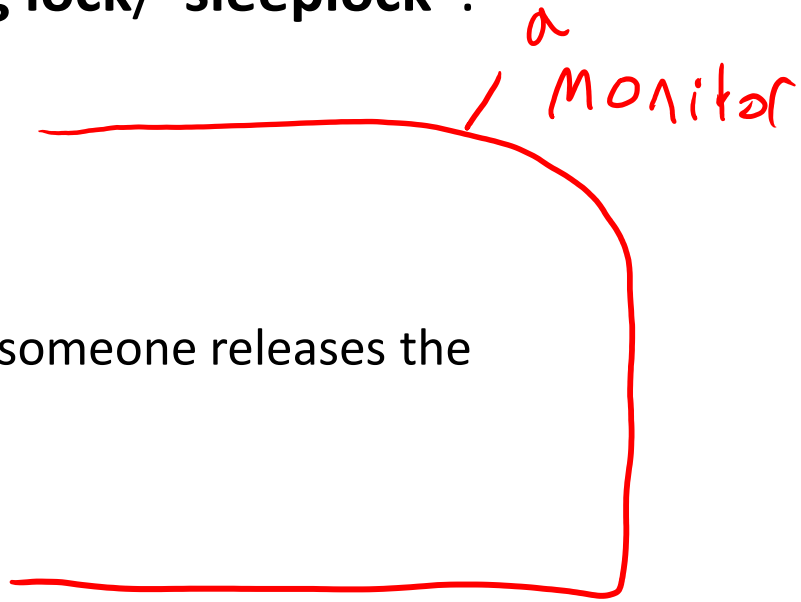


Blocking Locks

- If a thread will be waiting on a lock for a while, we want the scheduler to put it to sleep until the lock is free again. This is a **blocking lock**/"**sleeplock**".

- Common implementation:
  - Use a spinlock to protect the blocking lock struct
  - Acquire spinlock, see if sleeplock is free
  - If not, go to sleep and tell scheduler to not wake us up until someone releases the sleeplock
  - Once we wake up, try to acquire the sleeplock again

- Note: the process list is also protected by its own spinlock
  - Need to manage hand-off carefully
  - Performance bottleneck



xk's **kernel blocking lock** implementation

- `kernel/sleeplock.c`
- `kernel/proc.c`
  - `sleep()`
  - `wakeup()`



Userspace locks

- Our implementation of spinlocks requires us to disable interrupts
  - Disabling interrupts is a ⛔ **PRIVILEGED INSTRUCTION** ⛔
  - Hmm…
- Most real-life userspace locks are “blocking” and involve multiple kernel<->user context switches
  - Small optimization: spin for a little bit waiting for shared memory to be released, and then trip into the kernel if it takes too long
- Xk doesn’t provide userspace synchronization

WDLD (what does Linux do  ?)

- https://docs.kernel.org/locking/locktypes.html

Semaphores

Semaphores

- A more general synchronization primitive (can implement both locks and condition variables with them)
  - Mathematical purity
  - Write once/cry once

- The data: **an atomically modifiable non-negative integer**
- The (atomic) operations:
  - **Up / "Signal" / "V":** Increment the integer
  - **Down / "Wait" / "P"**: Wait until the integer is positive, and then decrement it

Semaphore varietals

- **Mutex** / **binary** semaphore init to 1
  - Used to represents single resource (eg, a critical section of code)
  - up( ) sets value to 1

- **Counting / general** semaphore init to N
  - Used to represent a resource with many units, or a resource that allows concurrent access (e.g., reading)
  - Multiple threads can up/down the semaphore



Semaphore design patterns (1/2)

- Mutual exclusion:

```
semaphore sem(1); // initialize with value of 1
thread_func() {
        sem.down()
        // critical section
        sem.up()
}
```

Semaphore design patterns (2/2)

- Ordering constraints:
  - Example: thread A wants to wait for thread B to finish

```
semaphore sem(0); // initialize with value of 0
```

```
thread_A() {
        // some work…
        sem.down() // hold up!!
        // more work…
}
```

```
thread_B() {
        // some work…
        sem.up() // all good!!
}
```

Semaphore Bounded Buffer

```
// Sempahores: mutex(1), emptySlots(MAX), fullSlots(0)
```

```
get() {
    fullSlots.down();
    mutex.down();
    item = buf[front % MAX];
    front++;
    mutex.up();
    emptySlots.up();
    return item;
}
```

```
put(item) {
    emptySlots.down();
    mutex.down();
    buf[last % MAX] = item;
    last++;
    mutex.up();
    fullSlots.up();
}
```

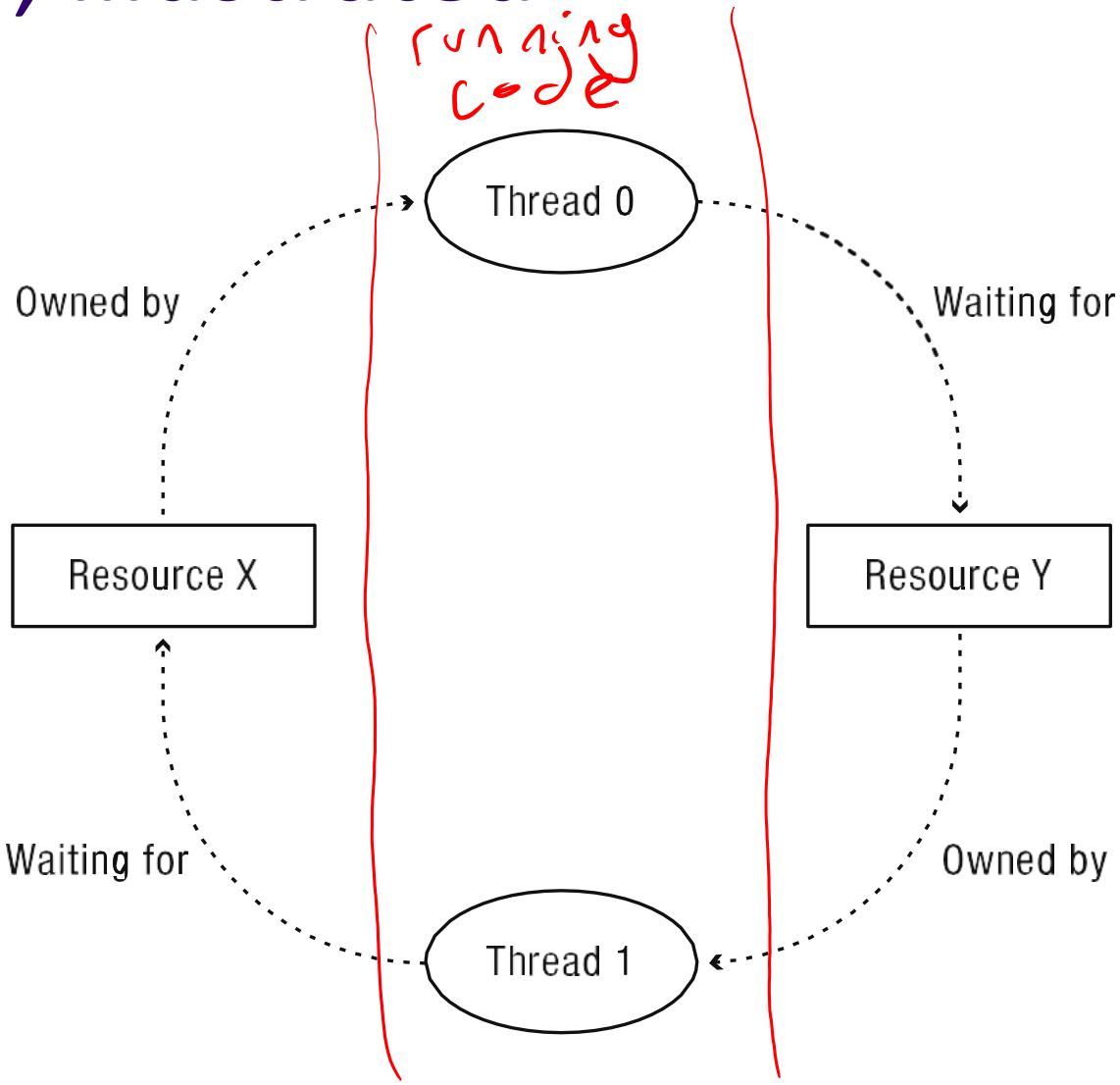
Deadlock

Multi-Object Programs

- Trouble starts to emerge in large programs when we're acquiring multiple locks at once
  - We'll often need to work with multiple shared data structures, each with its own lock and condition variables

- What is there to worry about?
  - Performance
  - Semantics/correctness
  - Deadlock

Deadlocks, illustrated



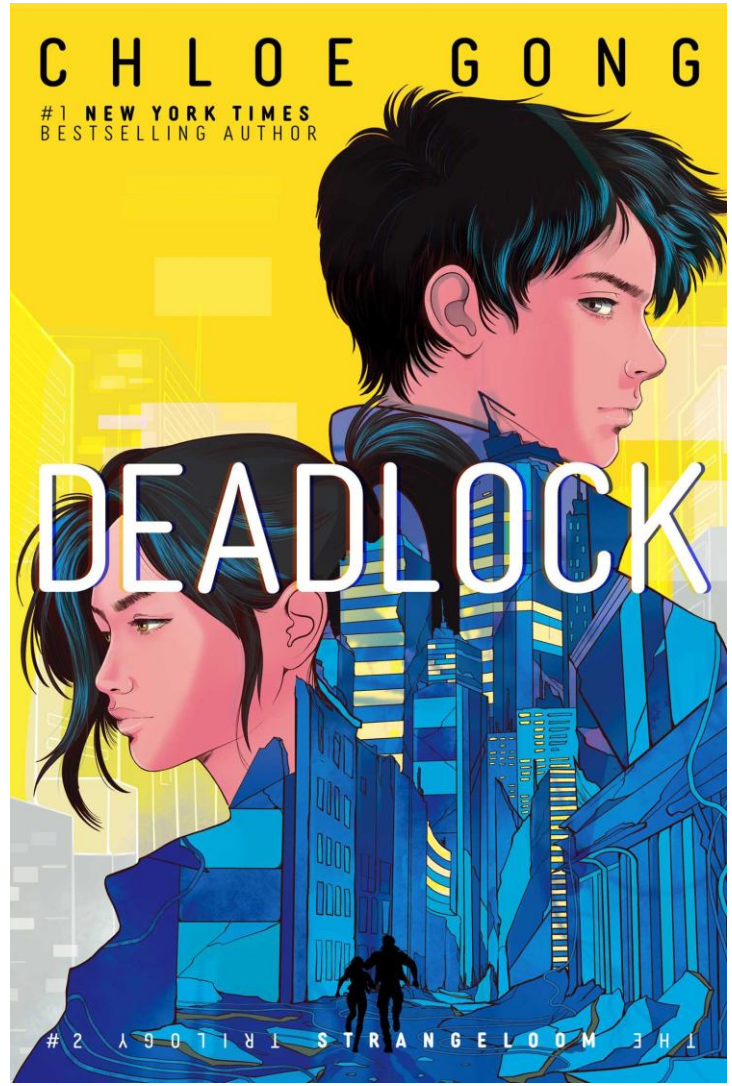
Deadlocks, classicly

Thread A

```
lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();
```

Thread B

```
lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
```


CHLOE GONG
#1 NEW YORK TIMES
BESTSELLING AUTHOR
DEADLOCK
#2 TRILOGY THE WOOLOOMS

Deadlocks, with condition variables

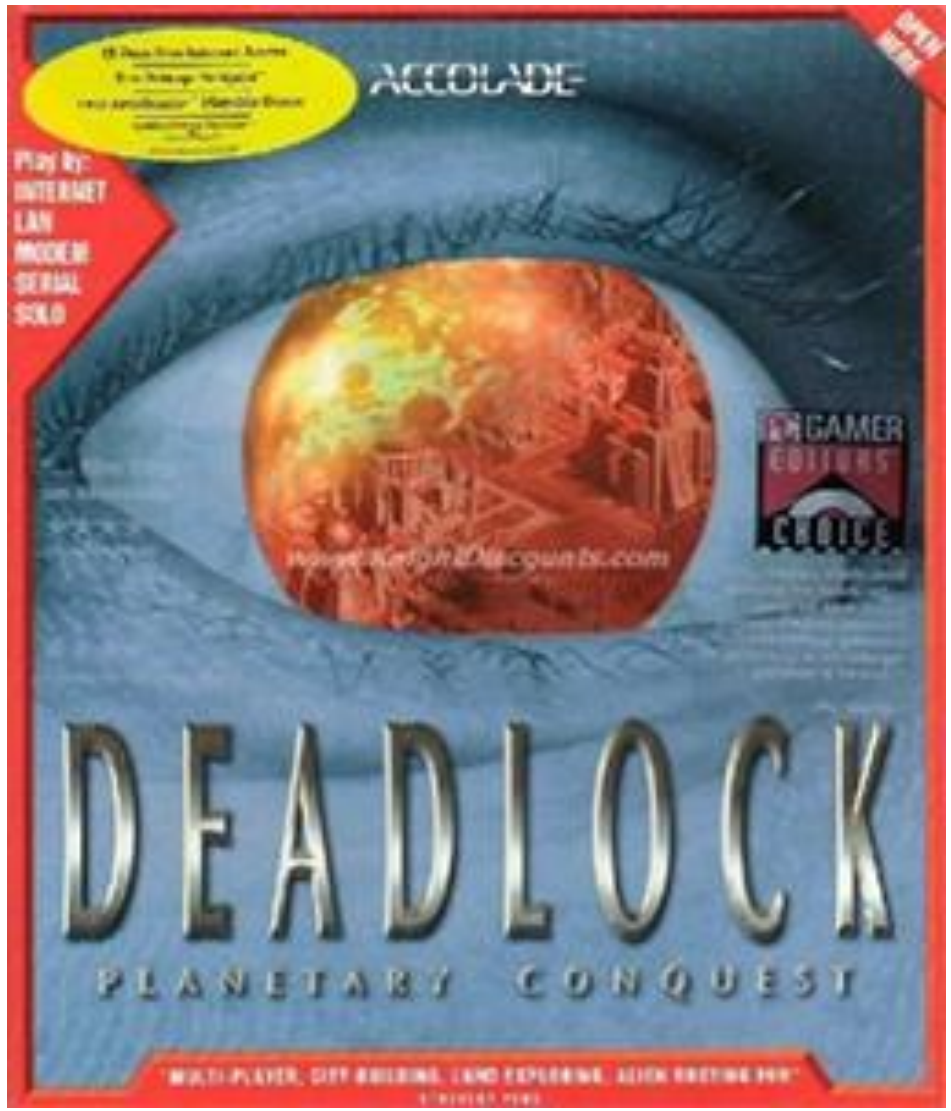
Thread A

```
lock1.acquire();
…
lock2.acquire();
while (need to wait) {
  condition.wait(lock2);
}
lock2.release();
…
lock1.release();
```

monitor

Thread B

```
lock1.acquire();
…
lock2.acquire();
…
condition.signal(lock2);
…
lock2.release();
…
lock1.release();
```


ACCOLADE
Play By:
INTERNET
LAN
MODEM
SERIAL
SOLO
GAMER
DEADLOCK
PLANETARY CONQUEST

Deadlocks, without the locks

Suppose we have two ring buffers that are both almost full (only one empty slot left):

Thread 1	Thread 2
`buffer1.put(data);`	`buffer2.put(data);`
`buffer1.put(data);`	`buffer2.put(data);`
`buffer2.get(data);`	`buffer1.get(data);`
`buffer2.get(data);`	`buffer1.get(data);`


prime
FROM THE WICKEDLY FUNNY MINDS OF
KATE McCARTNEY & KATE McLENNAN
DEADLOCH
JUNE 2
New Series

Class concluded here on 10/21 and will resume on 10/23



Deadlocks, realistically

- **Low priority thread A** acquires lock
- Preempted by **high priority thread B**
- **Thread B** tries to acquire lock; sleep
- Switch back to low priority **thread A**
- Preempted by **medium priority thread C**
- **Thread C** prevents high priority **thread B** from running



Deadlocks, formalized

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

The 🎶 Necessary Conditions for Deadlock 🎶

1. An exhaustible resource
   - Eg: a limited number of threads can use the resource
2. No preemption
   - Eg: nobody else can revoke a thread's access to the resource
3. Multiple independent requests
   - Eg: A thread holds one resource while waiting for another
4. Circular chain of requests
   - Eg: There is a set of waiting threads such that each is waiting for something held by another

A messier example still

Thread 1

1. Acquire A

3. Acquire C

5. If (maybe) Wait for B

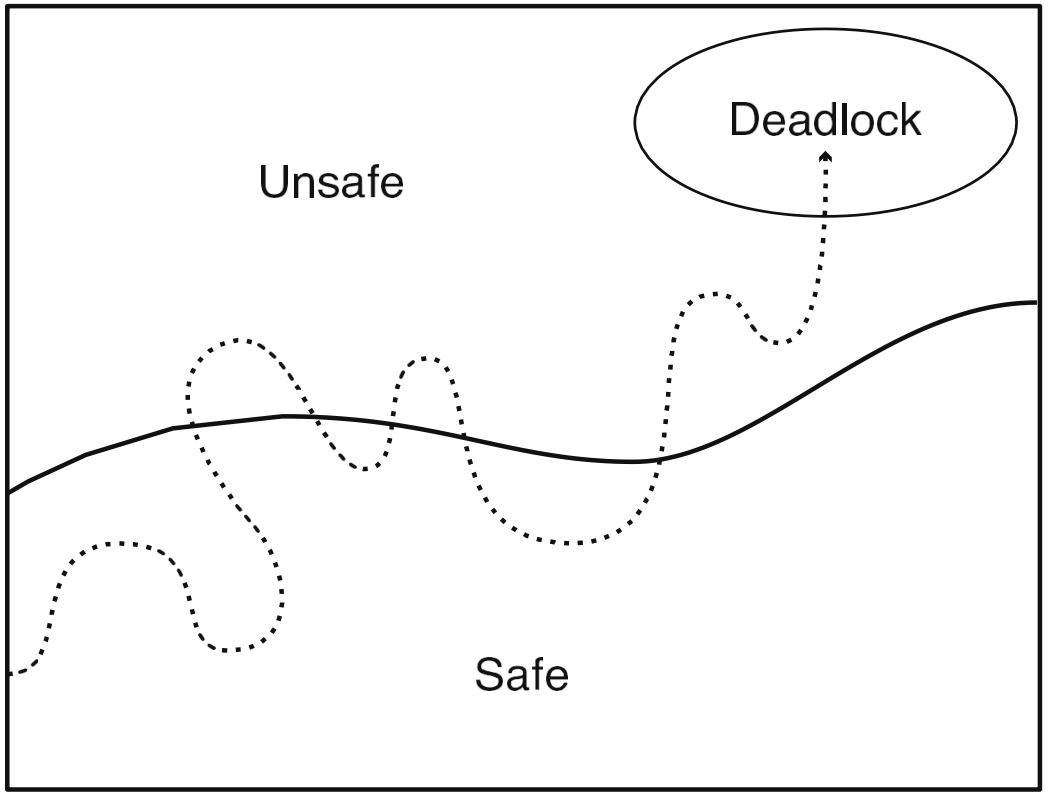
Thread 2

2. Acquire B

4. Wait for A

Deadlock Dynamics

- Safe state:
  - For any possible sequence of future resource requests, it is possible to eventually grant all requests
  - May require waiting even when resources are available!
- Unsafe state:
  - Some sequence of resource requests can result in deadlock
- Doomed state:
  - All possible computations lead to deadlock

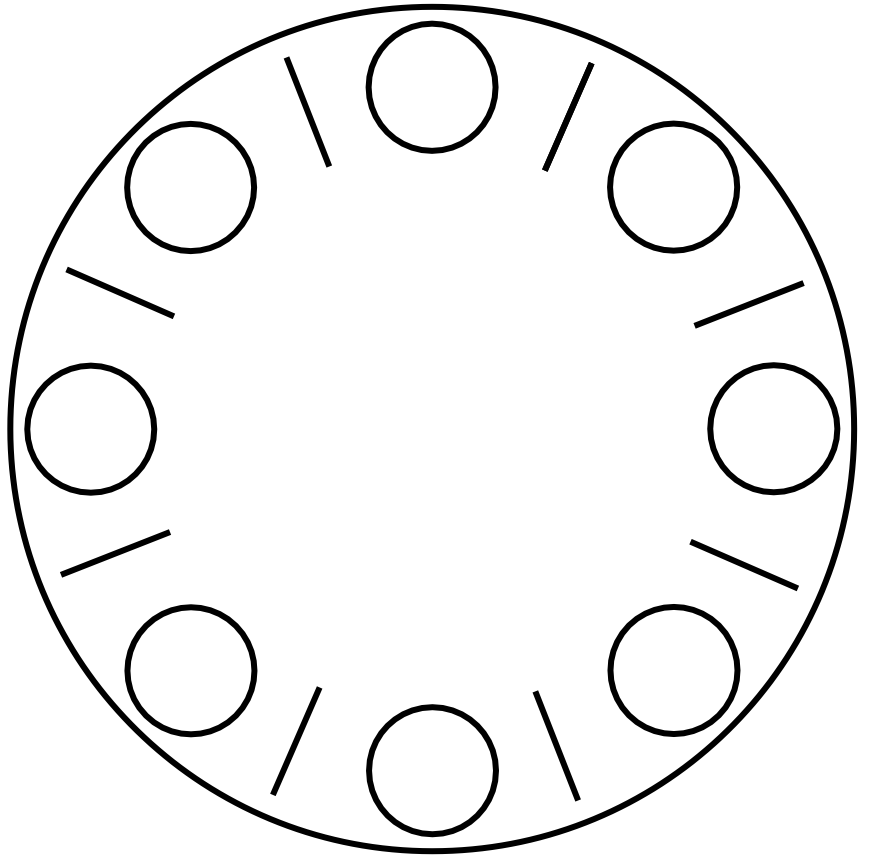


The Dining Lawyers

N lawyers. Each lawyer needs two chopsticks to eat.

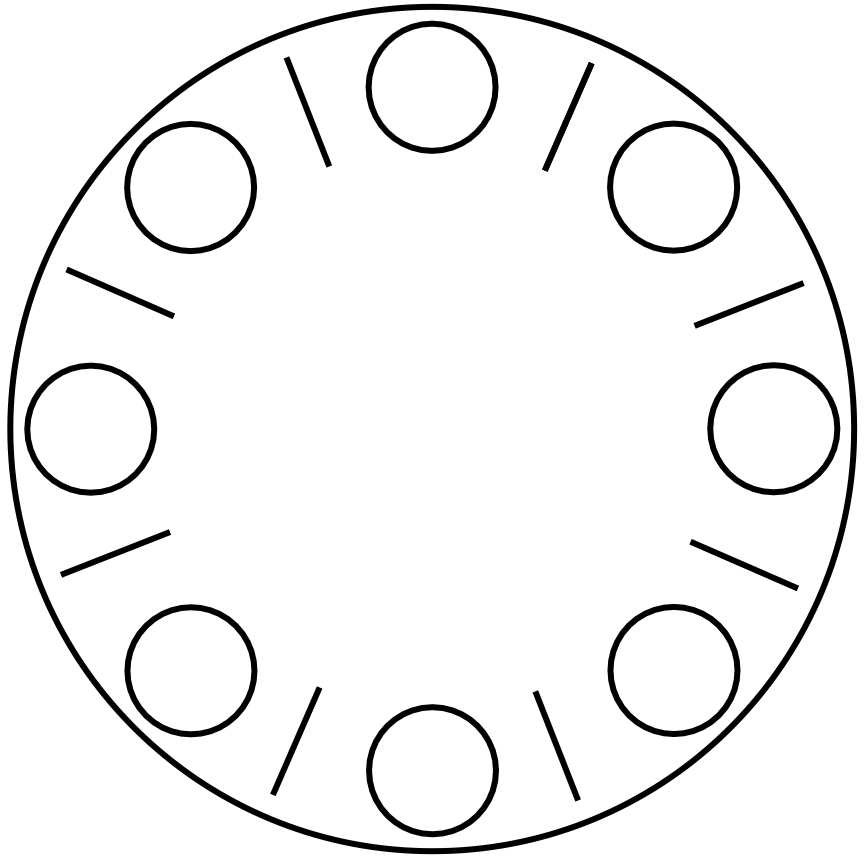
Each lawyer grabs chopstick on their right, then the one on their left.

```
lawyer(idx) {
    chopsticks[(idx + 1) % N].acquire()
    chopsticks[idx].acquire()
    eat();
    chopsticks[idx].release()
    chopsticks[(idx + 1) % N].release()
}
```



Paralysis Analysis

- Which 🎶 necessary conditions for deadlock 🎶 do the Dining Lawyers meet?
  - Limited access to resources
  - No preemption
  - Multiple independent requests (wait while holding)
  - Circular chain of requests

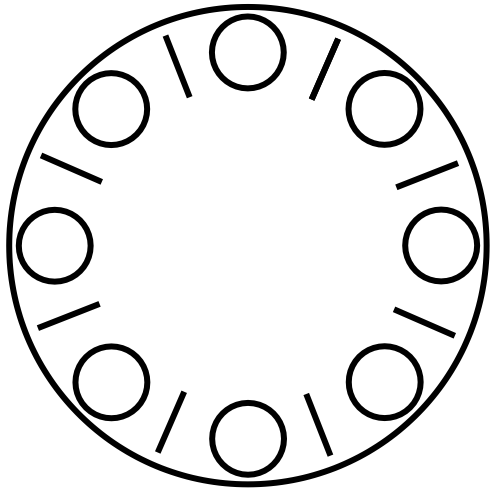


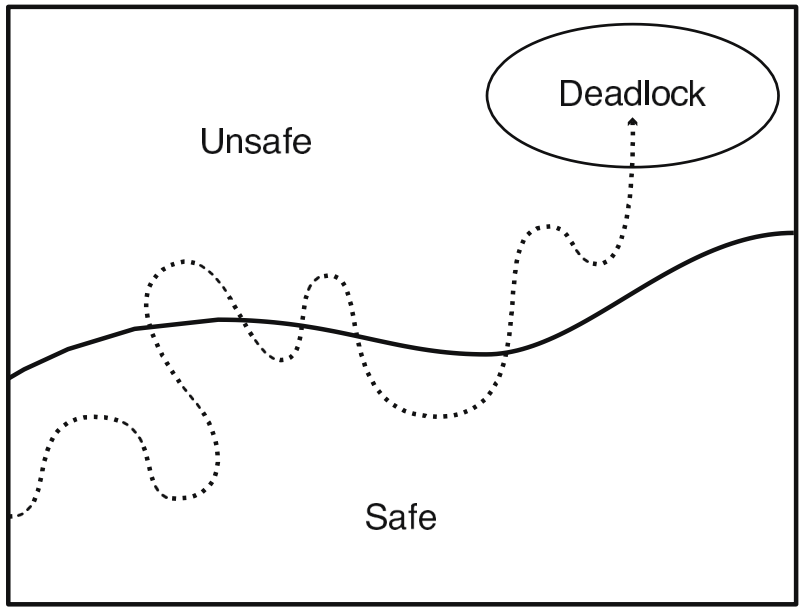
Paralysis Analysis (2)

- What are the doomed states for Dining Lawyers?

- What are the unsafe states?

- What are the safe states?





Deadlock Mitigation

Preventing Deadlock

- Exploit or limit program behavior
  - Limit program from doing anything that might lead to deadlock
- Predict the future
  - If we know what program will do, we can tell if granting a resource might lead to deadlock
- Detect and recover
  - If we can rollback a thread, we can fix a deadlock once it occurs

Exploit or Limit Behavior: resources

- Provide enough resources
  - Not always possible to increase resource count (eg, mutexes)
  - Not always obvious which resources are causing the problem

Exploit or Limit Behavior: waiting

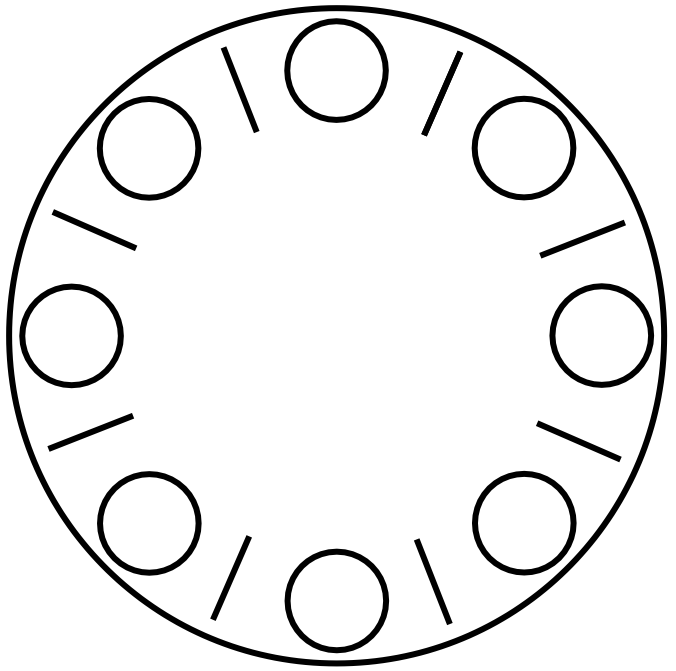
- Eliminate wait while holding
  - Release lock when calling out of module
  - Telephone circuit setup gives busy signal

Exploit or Limit Behavior: cycles

- Eliminate circular waiting
  - Lock ordering: **always acquire locks in a fixed order**
  - Example: move file from one directory to another

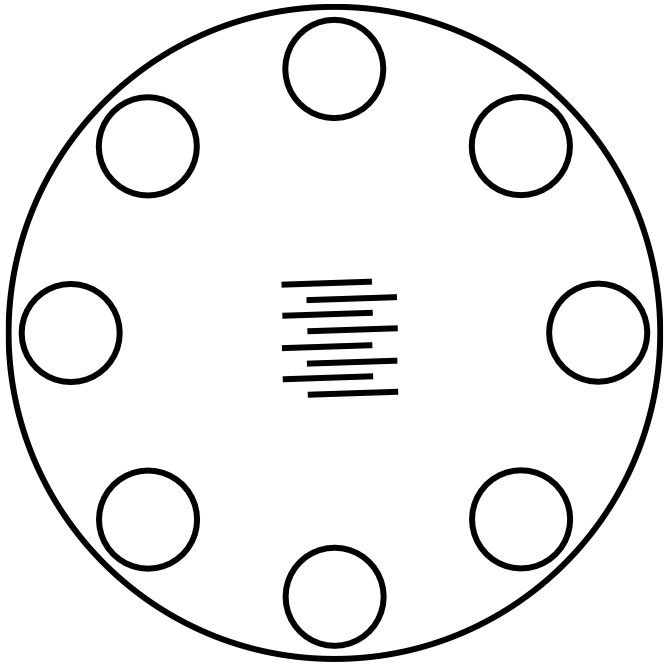
Lawyers again

- Can we use increase resource limits to eliminate deadlock?
- Can we use resource ordering to eliminate deadlock?
- How else can we be clever?



Communal Dining Lawyers

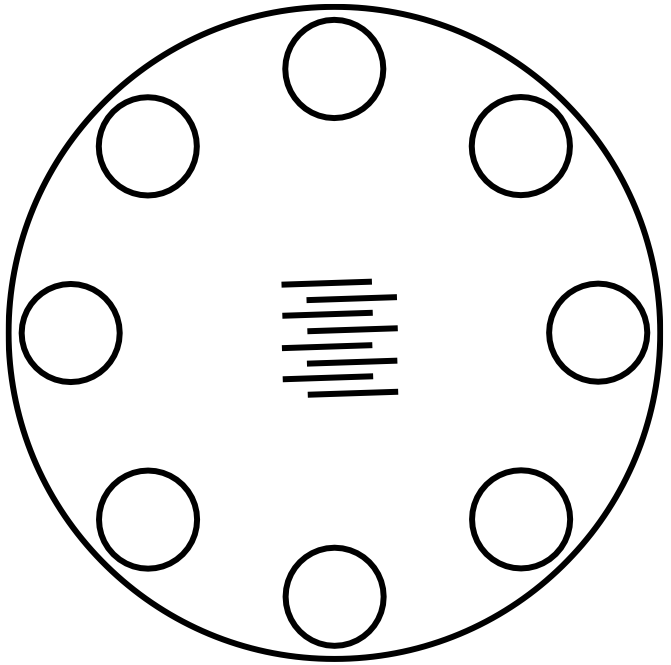
- **N** lawyers
- **N** chopsticks in **middle** of table
- each needs two chopsticks but can only take one at a time



- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Dining Lawyers

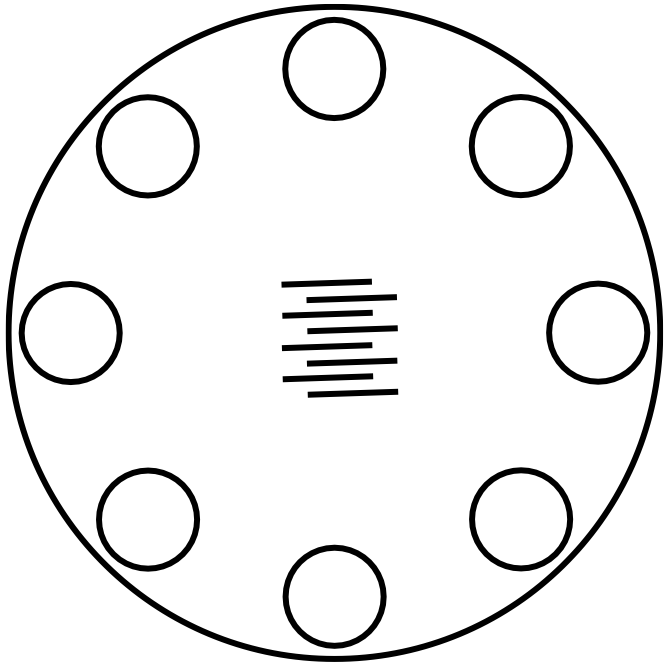
- **N** lawyers
- **N** chopsticks in **middle** of table
- each needs **k** chopsticks but can only take one at a time, for **k** > 1



- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Dining Lawyers **With Butterfingers**

- **N** lawyers
- **N** chopsticks in **middle** of table
- each needs **k** chopsticks but can only take one at a time, for **k** > 1
  - Sometimes a lawyer has a slippery grip and drops a chopstick, so they need **k+1** chopsticks
- What are the safe states?
- What are the unsafe states?
- What are the doomed states?



Banker's algorithm

- "You need to spend money to make money"
  - Banks offer small business loans under the assumption the business will use it to make enough money to pay it back (and then some)

- In an OS:
  - Provide a "gate" function that asks a process to specify its worst-case resource needs before it starts to acquire them
  - Based on global knowledge of all processes, determine if allowing the process to go ahead could *possibly* lead to deadlock
  - If it could, make process wait

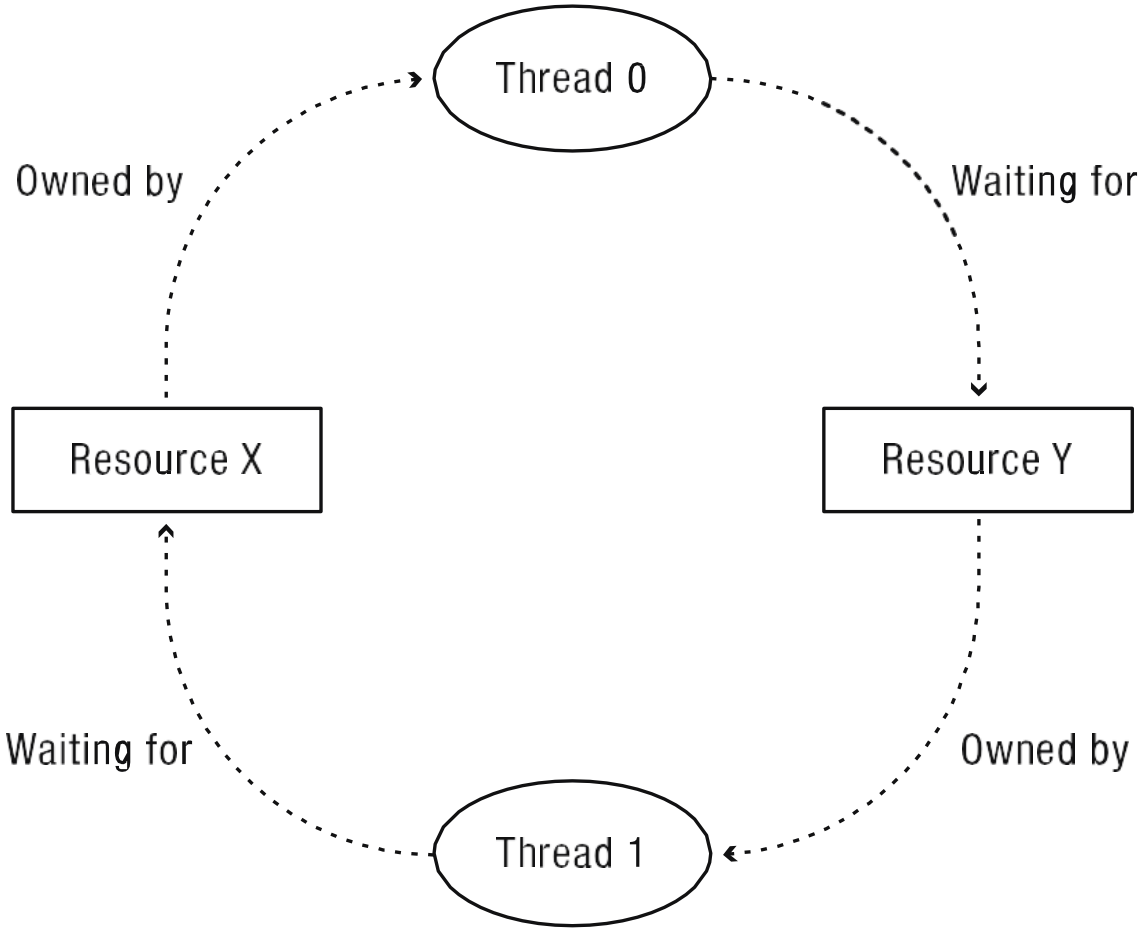
“Could cause deadlock” ?

- Consider how much total resource there is (T), how much is presently allocated elsewhere (A), and the maximum amount a process will need (M)
- If T – A >= M, we golden :)
- If T – A <= M …….well…..maybe
  - If we can find *some* order of giving each process the resources they need, we’re safe from deadlock
  - Finding this order can be difficult

Banker’s algorithm, practically

- Practically useful when our resources are quantifiable (like chopsticks) and non-unitary (unlike a lock/mutex)
  - Computer networking algorithms
  - Heap allocators

Deadlocks, illustrated



Alternative to playing it safe: **“Detect and Repair”**

- Algorithm
  - Examine the “waiting-for” graph
  - Detect cycles
  - Fix cycles…somehow
- Strategy 1: Proceed without the resource
  - Requires robust exception handling code
- Strategy 2: Roll back and retry
  - Transaction: all operations are provisional until have all required resources to complete operation