EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

Recap

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

Lock implementation #1

```
lock() {
        disable interrupts
        while (status != FREE) {
                enable interrupts
                disable interrupts
        }
        status = BUSY
        enable interrupts
}
```

```
unlock() {
        disable interrupts
        status = FREE
        enable interrupts
}
```

Lock implementation #2

```
// status=0 means lock is free
lock() {
      while (test_and_set(status) == 1) {
      }
}

unlock() {
      status = 0
}
```

Lock implementation #3

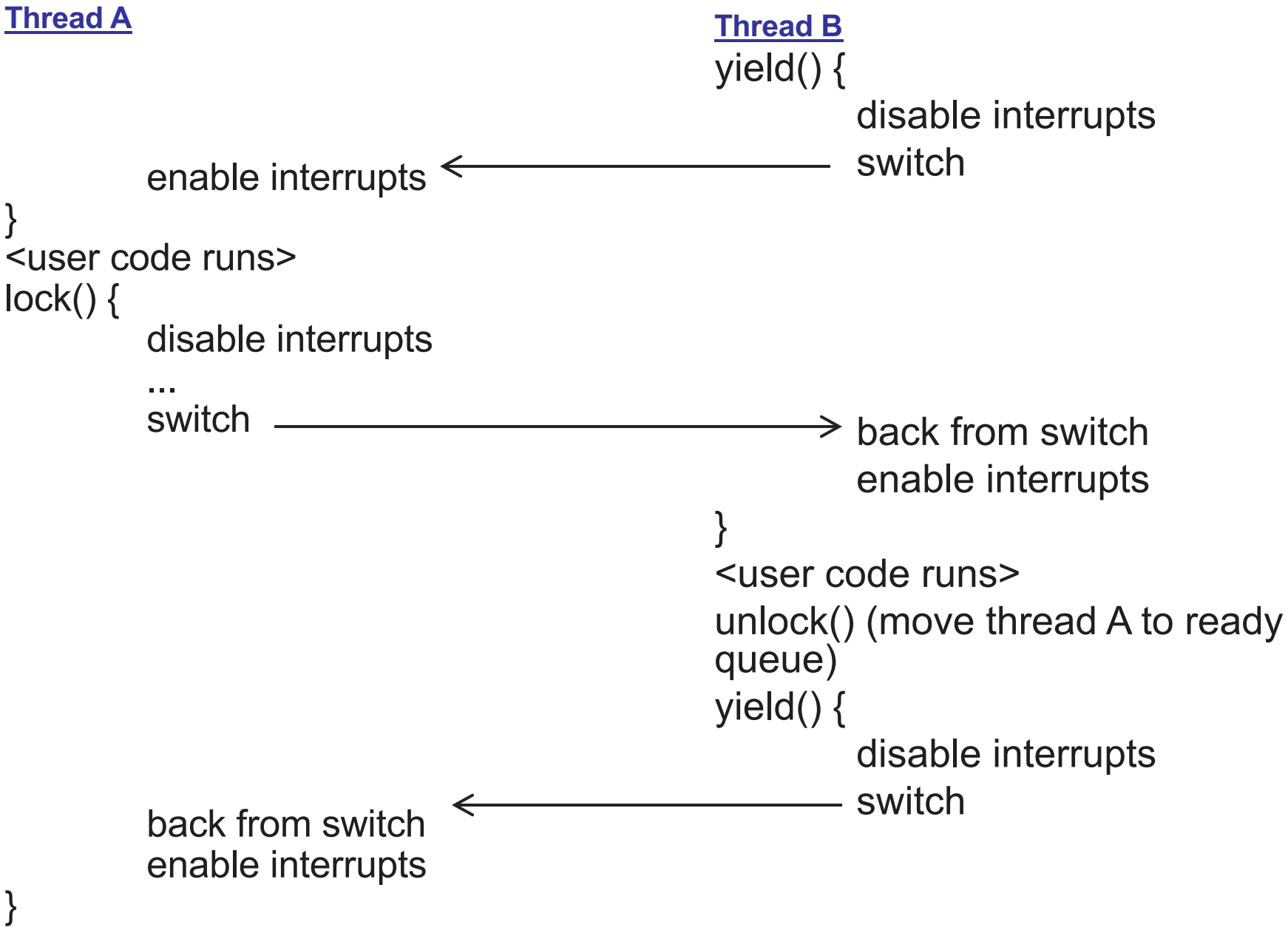
```
lock() {
        disable interrupts
        if (status == FREE) {
                status = BUSY
        } else {
                add thread to queue of threads waiting for lock
                switch to next ready thread
        }
        enable interrupts
}
unlock() {
        disable interrupts
        status = FREE
        if (any thread is waiting for this lock) {
                move waiting thread to ready queue
                status = BUSY
        }
        enable interrupts
}
```

Interrupt enable/disable pattern

- Atomically add thread to lock wait queue and switch
- Thread leaves interrupts disabled when calling switch

- Who will enable interrupts?
- When will a thread return from a switch?

- Switch invariant
  - All threads promise to disable interrupts when calling switch
  - All threads assume interrupts are disabled when returning from switch


Thread A
Thread B
yield() {
disable interrupts
switch
enable interrupts
}
<user code runs>
lock() {
disable interrupts
...
switch
back from switch
enable interrupts
}
<user code runs>
unlock() (move thread A to ready queue)
yield() {
disable interrupts
switch
back from switch
enable interrupts
}

Locks on multiprocessors

- Disabling interrupts insufficient to ensure atomicity if there are multiple CPUs
- Need to extend lock implementation #3 to use *test_and_set*

Lock implementation #4

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

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

Lock implementation #4

```
unlock() {
        disable interrupts
        while (test_and_set(guard)) {}

        status = FREE
        if (any thread is waiting for this lock) {
                move waiting thread to ready queue
                status = BUSY
        }

        guard = 0
        enable interrupts
}
```

What's the switch invariant for multiprocessors?

Summary of lock implementations

- High-level takeaways:
  - Disable/enable interrupts and test_and_set(guard) to protect critical section inside synchronization code
  - Atomically add thread to a waiting list and sleep
- How did we achieve this?
  - Switch to another thread and hand off task of enabling interrupts and resetting guard
- What if no other thread to run?
  - Atomically suspend CPU with interrupts enabled

Project 2

- Covered everything you need to know to implement all of the project
- Use assertions liberally
  - Assert any property that you expect to be true
  - Enables catching errors closer to where they occur
- Example:
  - Any thread is executing on at most one CPU

Deadlock

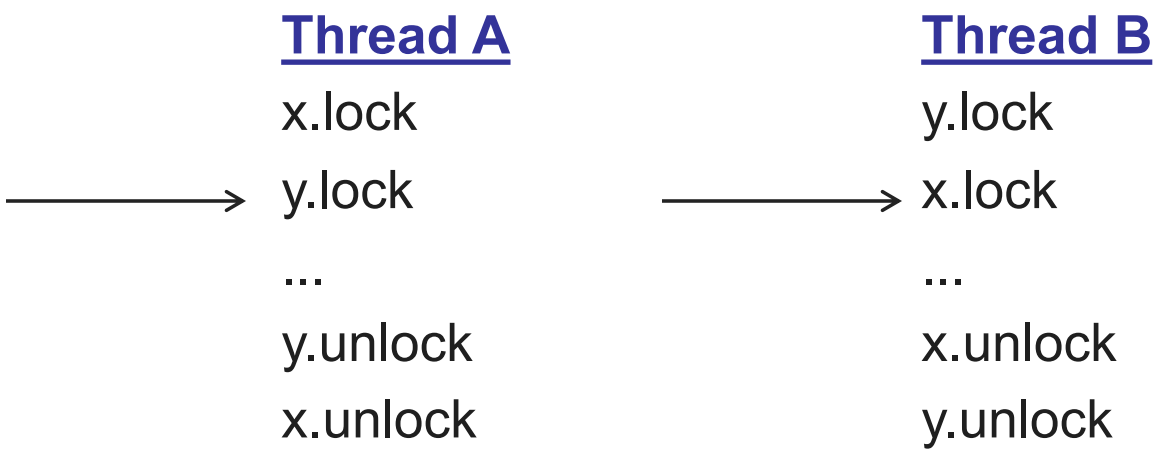
- Synchronization is about constraining executions
- Deadlock is what happens when an execution is over-constrained
  - A must happen before B, B must happen before A
- Example: Swapping classes
  - Alice is in 482, Bob is in 485, and they want to switch
    - » wait for spot to open
    - » add new class
    - » drop old class

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


BANCO REAL
11455

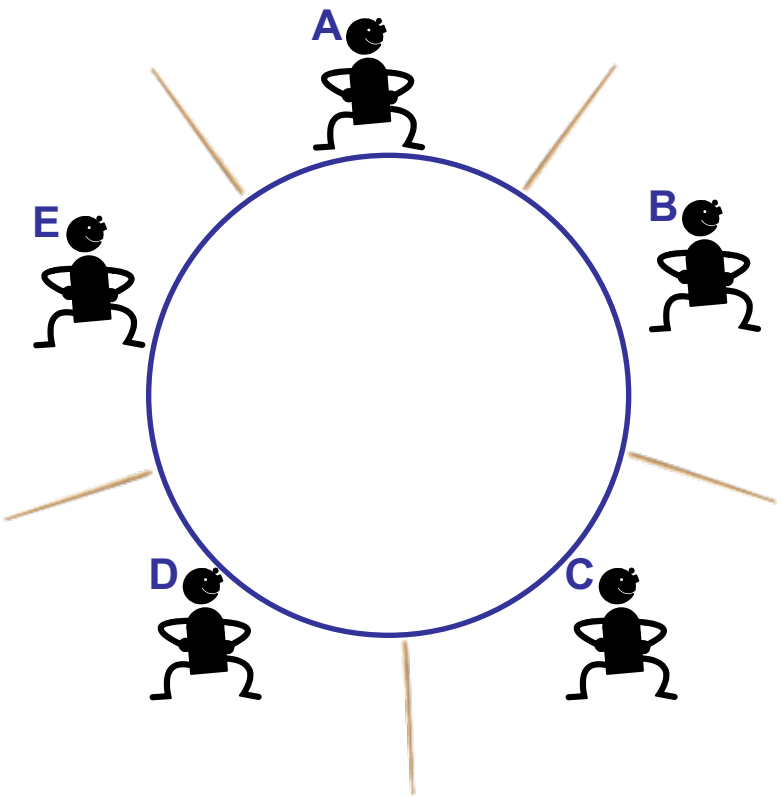
Deadlock example



- Will a deadlock always occur?

Dining philosophers

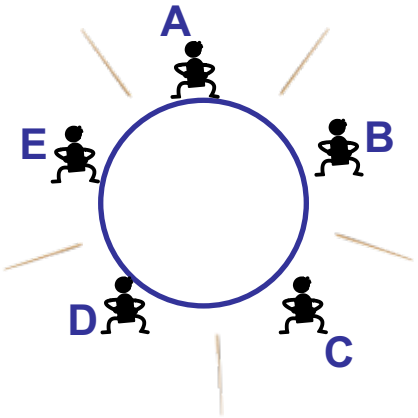
- 5 philosophers sit at round table
- 1 chopstick between each pair of philosophers
- Each philosopher needs 2 chopsticks to eat



Dining philosophers

- Algorithm for philosopher:

```
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
put both chopsticks down
```



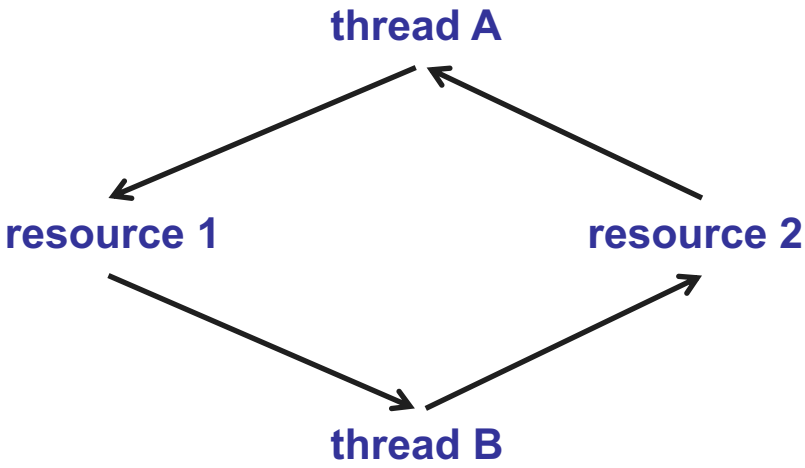
- Can this deadlock?

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

Strategies for handling deadlock

- Ignore
- Detect and fix
  - Detect cycles in the wait-for graph
  - How to fix once detected?
    - » Kill thread, grab resources
    - » Roll back execution
- 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
  - Threads hold resources while waiting to acquire other resources
- Cyclical chain of requests

Eliminating hold-and-wait

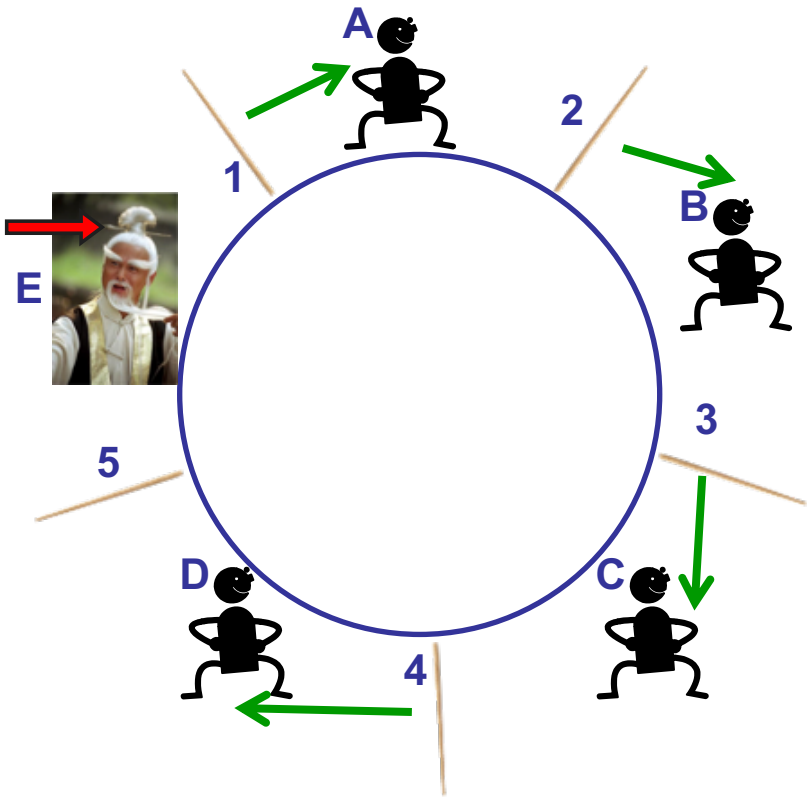
- Two ways to avoid hold and wait:
  - Wait for all resources needed to be free; grab them all atomically
  - If cannot get a resource, release all and start over

- Move resource acquisition to beginning

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

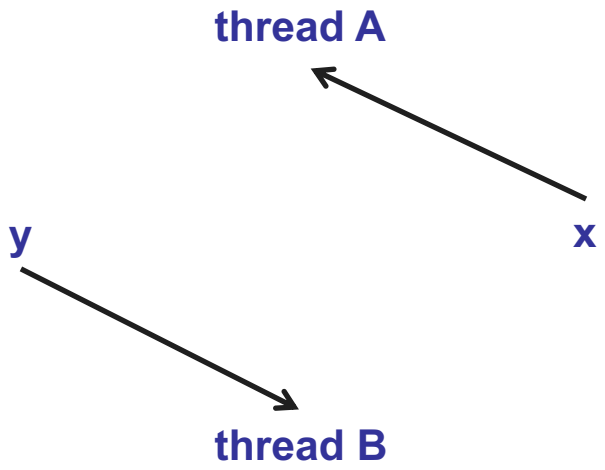
Eliminating circular chain

- Impose global ordering of resources



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