ECE 570/670

David Irwin

Lecture 10

Administrative Details

- Should be working on Assignment 1
  - Assignment 1 disk scheduler (1d)
    - How to use concurrency primitives
  - Assignment 1 thread library (1t)
    - How concurrency primitives actually work
  - **Midterm coming up (3/29)**
    - Review in class on 3/28
- Piazza questions are great
- Will post videos in advance of mid-term

Administrative Details

- ECE670 Project Proposals (due Friday, 3/24)
  - Spoke with most groups
  - Other groups should drop by office hours

Lock implementation #3

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Putting lock thread on lock wait queue, switch must be atomic. Must call switch with interrupts off.

How is switch returned to?

- Recall swapcontext
- Think of switch as three phases
  - Save current location (SP, PC)
  - Point processor to another stack (SP')
  - Jump to another instruction (PC')
- Only way to get back to a switch?
  - Have **another** thread call switch

Lock implementation #3

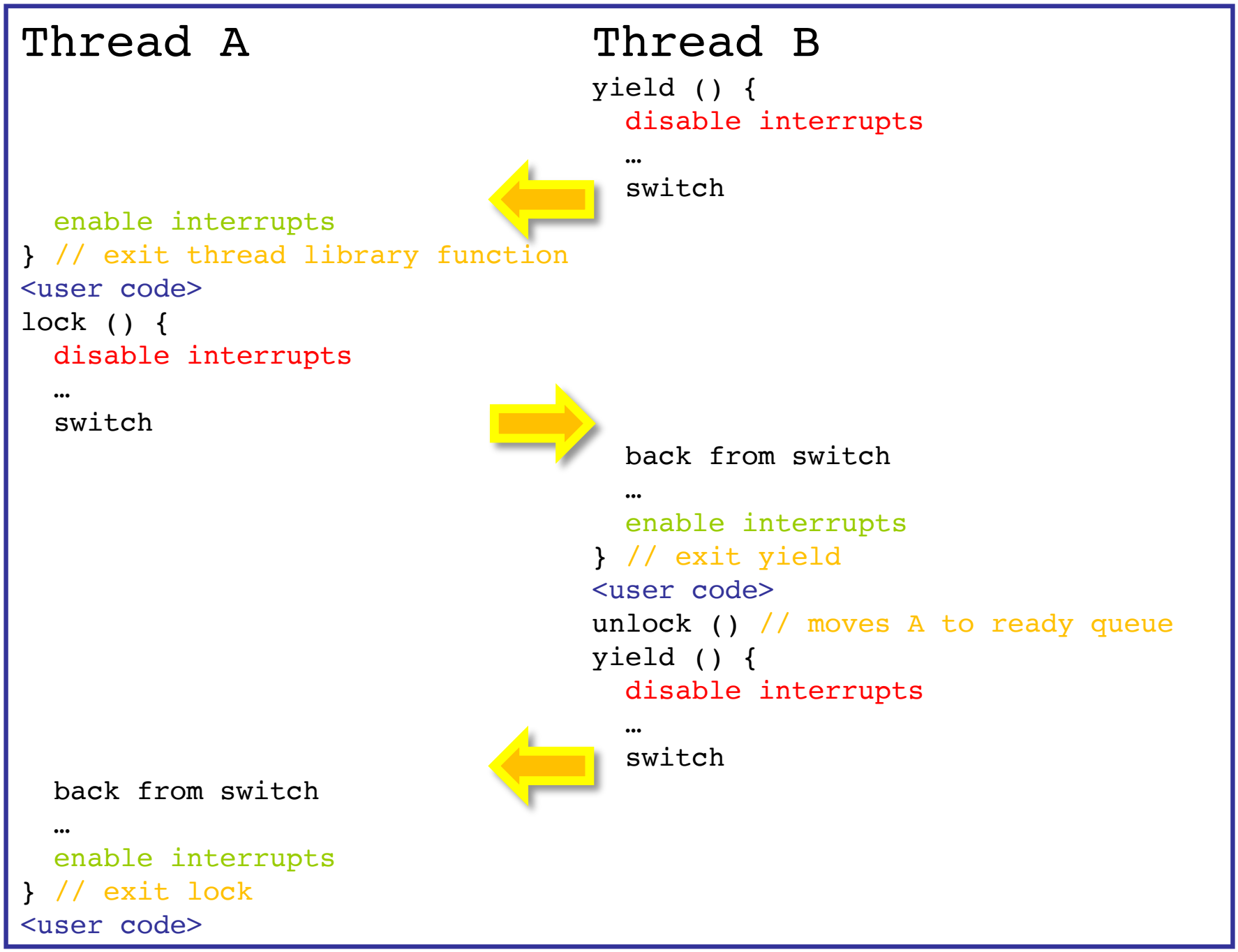
```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

What is lock() assuming about the state of interrupts after switch returns?

Interrupts and returning to switch

- Lock() can assume that switch
  - Is always called with interrupts disabled
- On return from switch
  - Previous thread must have disabled interrupts
- Next thread to run
  - Becomes responsible for re-enabling interrupts
- **Invariants**: threads promise to
  - Disable interrupts before switch is called
  - Re-enable interrupts after returning from switch


Thread A
Thread B
yield () {
disable interrupts
…
switch
enable interrupts
} // exit thread library function
<user code>
lock () {
disable interrupts
…
switch
back from switch
…
enable interrupts
} // exit yield
<user code>
unlock () // moves A to ready queue
yield () {
disable interrupts
…
switch
back from switch
…
enable interrupts
} // exit lock
<user code>

Lock implementation #4

- Test&set, minimal busy-waiting

```
lock () {
  while (test&set (guard)) {} // like interrupt disable
  if (value == FREE) {
    value = BUSY
  } else {
    put on queue of threads waiting for lock
    switch to another thread // don't add to ready queue
  }
  guard = 0  // like interrupt enable
}
```

```
unlock () {
  while (test&set (guard)) {} // like interrupt disable
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  guard = 0  // like interrupt enable
}
```

Lock implementation #2

- Use test&set
- Initially, value = 0

```
lock () {
  while (test&set(value) == 1) {
  }
}
```

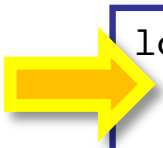
```
unlock () {
  value = 0
}
```

What happens if value = 1?

What happens if value = 0?

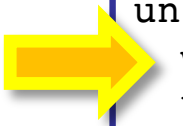
Lock implementation #4

Why is this better than implementation #2?



Only busy-wait while another thread is in lock or unlock

```
lock () {
  while (test&set (guard)) {} // like interrupt disable
  if (value == FREE) {
    value = BUSY
  } else {
    put on queue of threads waiting for lock
    switch to another thread // don't add to ready queue
  }
  guard = 0  // like interrupt enable
}
```



Before, we busy-waited while lock was held

```
unlock () {
  while (test&set (guard)) {} // like interrupt disable
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  guard = 0  // like interrupt enable
}
```

Lock implementation #4

What is the switch invariant?

Threads promise to call switch with guard set to 1.

```
lock () {
  while (test&set (guard)) {} // like interrupt disable
  if (value == FREE) {
    value = BUSY
  } else {
    put on queue of threads waiting for lock
    switch to another thread // don't add to ready queue
  }
  guard = 0  // like interrupt enable
}
```

```
unlock () {
  while (test&set (guard)) {} // like interrupt disable
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  guard = 0  // like interrupt enable
}
```

Summary of implementing locks

- Synchronization code needs atomicity
- Three options
  - **Atomic load-store**
    - Lots of busy-waiting
  - **Interrupt disable-enable**
    - No busy-waiting
    - Breaks on a multi-processor machine
  - **Atomic test-set**
    - Minimal busy-waiting
    - Works on multi-processor machines

Assignment 1 (Part 2)

- Write the thread library used in Part 1
  - Use Linux calls provided for building user-level thread libraries
    - getcontext(); makecontext(); swapcontext();
    - Operate on ucontext structs (i.e., the TCB)
    - Example of creating new threads on website
    - Also, see man pages
  - Given libinterrupt library
    - Allows you to enable and disable interrupts in your thread library
    - Only disable interrupts in your thread library code (not in applications!)
  - Look at pseudocode from class
    - Implement ready queue, wait queue for lock, and wait queue for signal
    - Locks should be acquired in order they are requested (FIFO)

Assignment 1 (Part 2)

- Write the thread library used in Part 1
  - getcontext();
    - Initialization of a context, e.g., TCB
  - makecontext();
    - Takes function associated with the context as argument
    - Should have this point to a function **startFunction** *in your thread library*
      - This function takes the user function as an argument
    - **startFunction** will call the user's thread function
      - When user function returns the thread is finished executing
  - swapcontext();
    - As in the slides

Assignment 1 (Part 2)

- Error Handling
  - Must detect library misuse
    - E.g., calling thread function before calling thread_libinit, calling thread_libinit multiple times, etc.
  - Must detect if functions it calls return an error
    - Return -1 from thread function on errors
  - Use "asserts" throughout thread library to catch logic errors in your thread library

- Getting started
  - Start with implementing thread_libinit, thread_create, and thread_yield
  - After those working, then work on the monitor functions

Assignment 1 (Part 2)

- You will also need to work on thread library test suite
- For each of your test cases
  - Compare output of your library with thread.o
- Writing test cases
  - Read through spec
    - Write test cases to stress each required part
      - E.g. lock() blocks if lock is already held
    - Use yield() to create the right inter-leavings
  - Read through your code
    - Write test cases to exercise all lines of code
    - E.g. each clause of an if/else statement

Assignment 1 (Part 2)

- Micro-tests better for debugging than macro-tests
  - One test case usually uncovers many bugs
  - Use disk scheduler and example program as templates for creating test cases

- Test cases will be graded on how thoroughly they exercise your thread library
  - Will run them against thread libraries with various common bugs and see if you test case exposes those bugs

Assignment 1 (Part 2)

- Compiling and running
  - Must be in a single file thread.cc
  - **g++ thread.cc app.cc libinterrupt.a -ldl -o app**
  - Thread library only generates one output line
    - "Thread library exiting."
  - **submit570 1t thread.cc test1.cc test2.cc**

Hints on Part 2

- Warnings are probably ok; errors are not ok
- Make sure you implement hand-off locks, as discussed in class
- Make sure you can create lots of threads without running out of memory
- Make sure you delete threads that are finished
  - How do you know a thread finishes?
- Make sure you enable and disable interrupts appropriately
  - Follow invariants discussed in class
- Test cases: not just for the autograder, mostly for your own code!

Concurrency so far

- Mostly tried to constrain orderings
  - Locks, monitors, semaphores
- Note that it is possible to over-constrain too
  - A must run before B
  - B must run before A
- This can lead to **deadlock**

Definitions

- **Resource**
  - "Thing" needed by a thread to do its job
  - Threads wait for resources
  - "Things" can be locks, disk space, memory, CPU
- **Deadlock**
  - Circular waiting for resources
  - Leaves all waiting threads unable to make progress

Example deadlock

- Both CS187 and ECE241 are full
  - You want to switch from 187 to 241
  - Someone else wants to switch from 187 to 241
- Algorithm for switching
  - Wait for spot in new class to open up
  - Add new class
  - Drop old class
- Problem: must add before dropping

Deadlock and starvation

- Deadlock → starvation
  - Starvation = one process waits forever

Thread A	**Thread B**
`lock (x)`	`lock (y)`
`lock (y)`	`lock (x)`
…	…
`unlock (y)`	`unlock(x)`
`unlock (x)`	`unlock(y)`

Can deadlock occur?

Deadlock and starvation

- Deadlock → starvation
  - Starvation = one process waits forever

Thread A	**Thread B**
`lock (x)`	
	`lock (y)`
`lock (y) // wait for B`	
	`lock (x) // wait for A`

Will deadlock always occur?

Can deadlock always occur?

Common thread work pattern

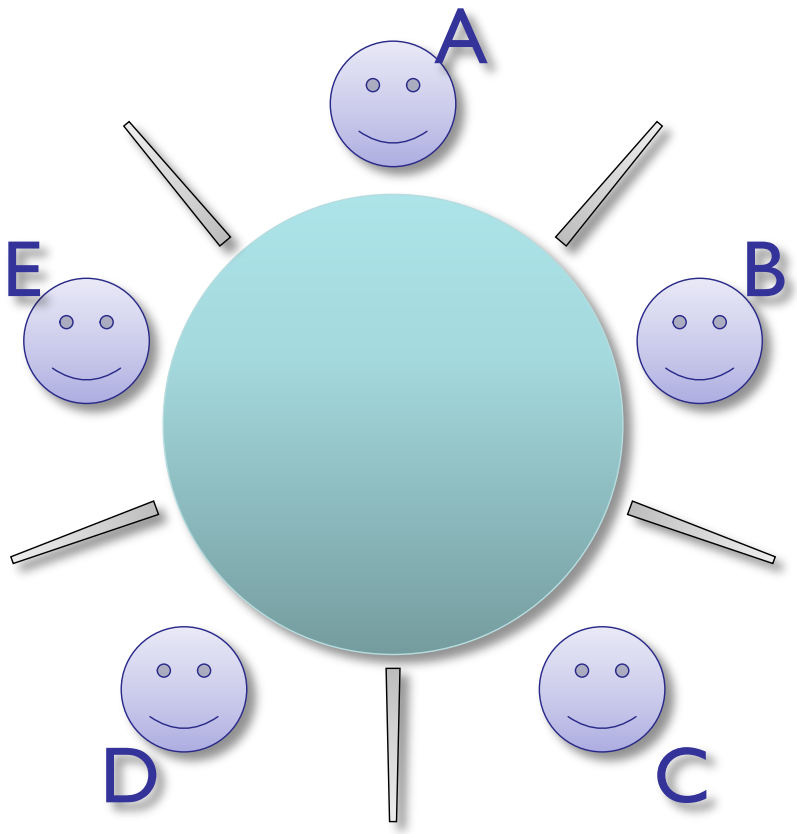
```
Phase 1.  while (not done) {
            get some resources (block if necessary)
            work  // assume finite
          }
Phase 2.  release all resources
```

- This is called "two-phase" locking
  - Common in databases, but…
  - …deadlock-prone

Dining philosophers

Philosophers alternate between eating (noodles) and thinking

```
Philosopher algorithm
1) Wait for right chopstick
2) Pick up right chopstick
3) Wait for left chopstick
4) Pick up left chopstick
5) Eat
6) Put both chopsticks down
```



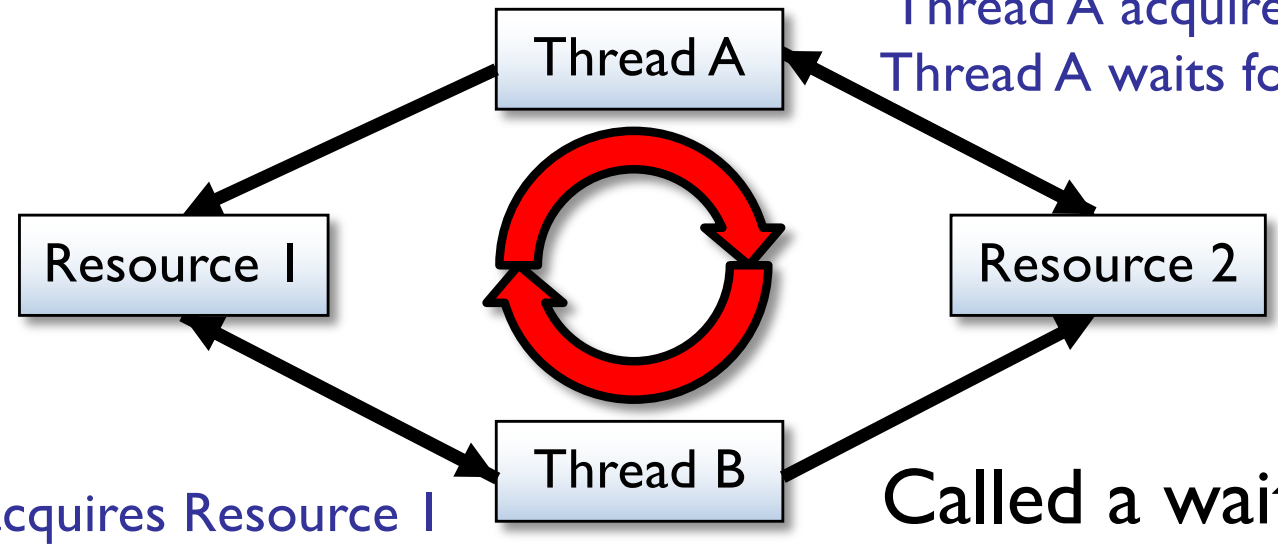
How can deadlock occur?

Conditions for deadlock

1. Limited resource
   - Not enough for all threads simultaneously
2. Hold-and-wait
   - Hold one resource, while waiting for another
3. No pre-emption
   - Cannot force threads to give up resources
4. Circular chain of requests

Circular chain of requests

- Arrows
  - Thread → resource it's waiting for
  - Resource → thread that's holding it



Thread A acquires Resource 2
Thread A waits for Resource 1

Thread B acquires Resource 1
Thread B waits for Resource 2

Called a wait-for graph

Deadlock

- As the OS, what should we do about deadlocks?

1. Ignore them
   - Common OS-level solution for programs
   - (OS has no control over user programs)
2. Detect-and-fix
3. Prevent

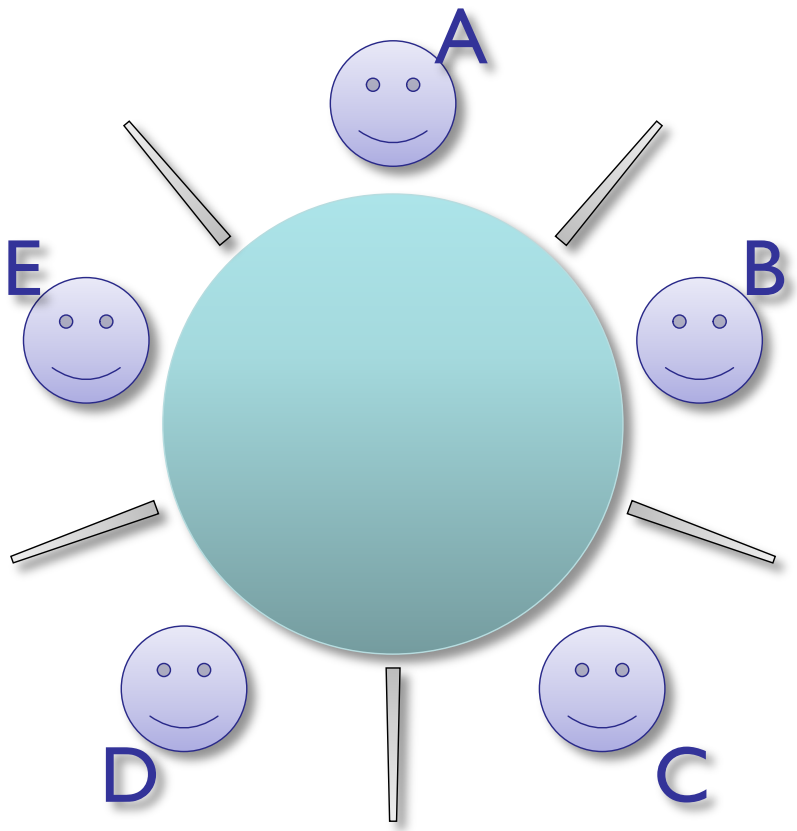
Detect-and-fix

- First part: **detect**
  - This is easy; just scan the wait-for graph
- Second part: **fix**
  1. Kill first thread, take back resources by force
     - Why might this be unsafe?
       - Can expose inconsistent state – invariants might be broken
  2. Roll-back actions of 1 or more thread, retry
     - Often used in databases
     - Not always possible (some actions can't be undone)
       - E.g. can't unsend a network message

Detect-and-fix

- Retrying during fix phase can be tricky
  - If holding R and waiting for L, drop R and try again.

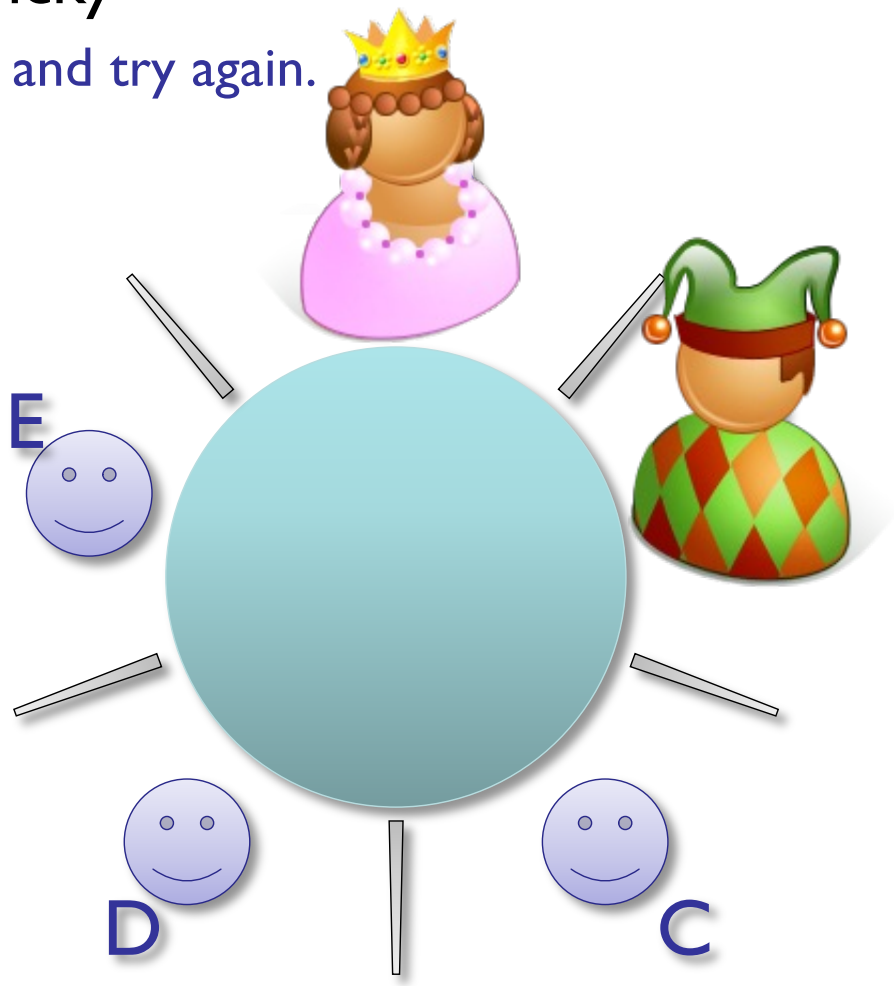
What could happen?
Everyone picks up R
Everyone drops R
Everyone picks up R
Everyone drops R
(and so on)
This is called "livelock."



Detect-and-fix

- Retrying during fix phase can be tricky
  - If holding R and waiting for L, drop R and try again.

How to prevent livelock?
  Choose a winner.
How to choose a winner?
  First to start/pick up?
What if we have priorities?
  (called "priority inversion")
  e.g. Queen waiting for joker
Should Queen always win?
  Depends, could starve the joker



Deadlock prevention

- Idea: remove one of four pre-conditions

1. Make resources unlimited
   - Best solution, but often impossible
   - Can still increase number, reduce likelihood
   - (larger 187 and 241 classes)
   - This doesn't work for locks

Deadlock prevention

2. Eliminate hold-and-wait

- Idea: move resource acquisition to beginning

```
Phase 1a. get all resources
Phase 1b. while (not done) {
            work  // assume finite
          }
Phase 2.  release all resources
```

- Two ways to avoid holding while waiting

Eliminating hold-and-wait

1. Wait for everything to be free, then grab them all at once

```
Philosopher algorithm
  lock
  while (left chopstick busy ||
         right chopstick busy) {
    wait
  }
  pick up right chopstick
  pick up left chopstick
  unlock
  eat
  lock
  drop chopsticks
  unlock
```

Any potential problems?

Could induce starvation (neighbors alternate eating, one starves).

Eliminating hold-and-wait

2. If you find a resource busy, drop everything, and try again

```
Philosopher algorithm
  lock
  while (1) {
    while (right chopstick busy) {
      wait
    }
    pick up right chopstick
    if (left chopstick busy) {
      drop right chopstick
    } else {
      pick up left chopstick
      break
    }
  }
  unlock
  eat
  lock
  drop chopsticks
  unlock
```

Issues?

Overconstrains concurrency

Hold reservations longer than needed.

Deadlock prevention

3. Enable pre-emption (force threads to give up resources)

- Can pre-empt the CPU (context switch)
- Can pre-empt memory (swap to disk)
- Not everything can be pre-empted
- Can locks be pre-empted?
  - Probably not
  - Must ensure that data is in consistent state

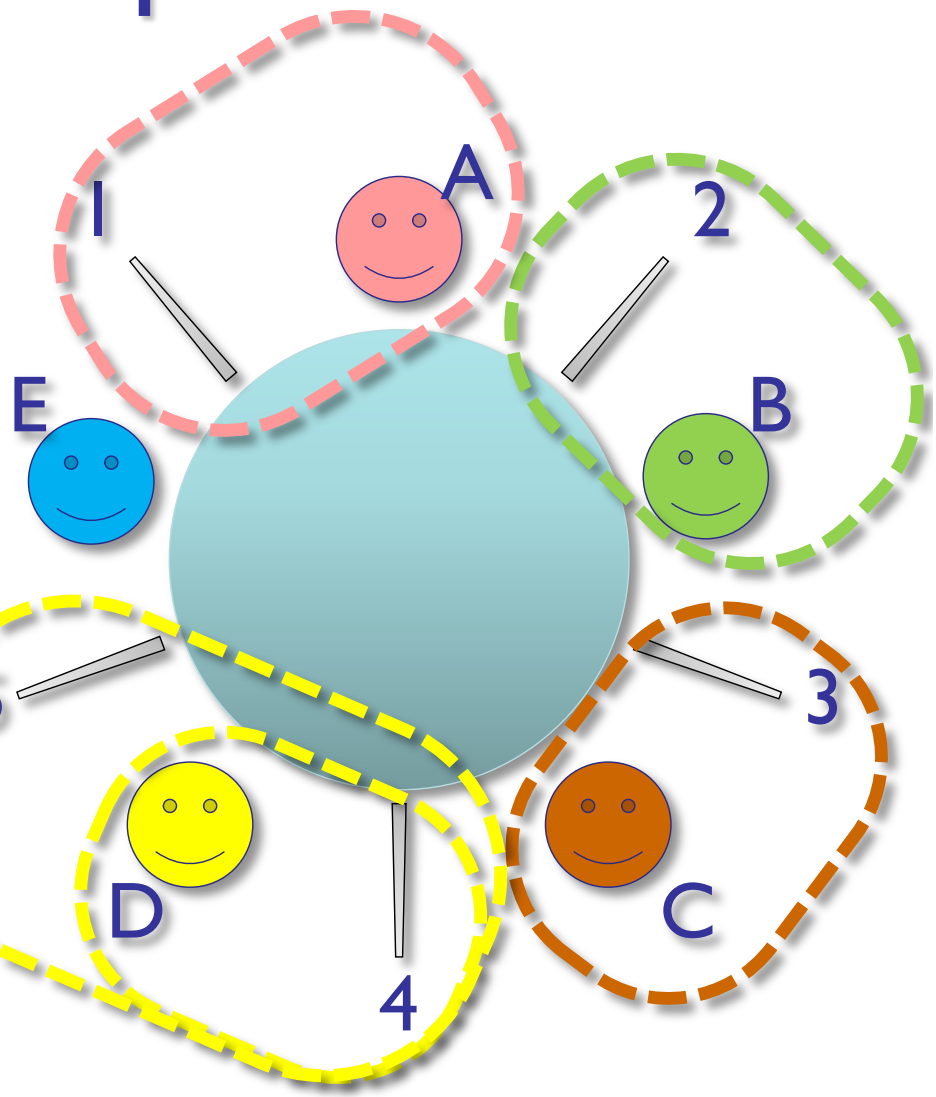
Deadlock prevention

4. Eliminate circular chain of requests

- How can we get rid of these cycles?
  - Impose an ordering on resource acquisition
  - E.g. must always acquire A before B
- In dining philosophers?
  - Number chopsticks
  - Pick up lower-numbered chopstick first

Dining philosophers

```
Philosopher algorithm
1) Wait for lower chopstick
2) Pick up lower chopstick
3) Wait for higher chopstick
4) Pick up higher chopstick
5) Eat
6) Put both chopsticks down
```



Try to create a deadlock

Universal ordering

- Why does this work?
- At some point in time
  - T holds highest-numbered acquired resource
  - T is guaranteed to make progress. Why?
    - If T needs a higher-numbered resource, it must be free
    - If T needs a lower-numbered resource, it already has it
  - If T can make progress, it will eventually release all of its resources
  - What if another thread acquires a higher-numbered resource?
    - They just become T
    - (in which case, the same reasoning as above holds)