Based on slides by Harsha V. Madhyastha

# EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 9:  Deadlock

Nicole Hamilton

https://web.eecs.umich.edu/~nham/

nham@umich.edu

## Midterm exam

Online using *Crabster.org* Wed Jun 24 3:00 to 5:00 pm EDT.

If you need an accommodation, please let us know soon.

We will have covered all material for midterm by end of this lecture.

Material for midterm:

1. All the lecture topics from start until end of deadlock.
2. All the labs on these topics.
3. Projects 1 and 2.

## Interrupt enable/disable pattern

Adding thread to lock wait queue + switching must be atomic.

Swapcontext invariants for uniprocessors:

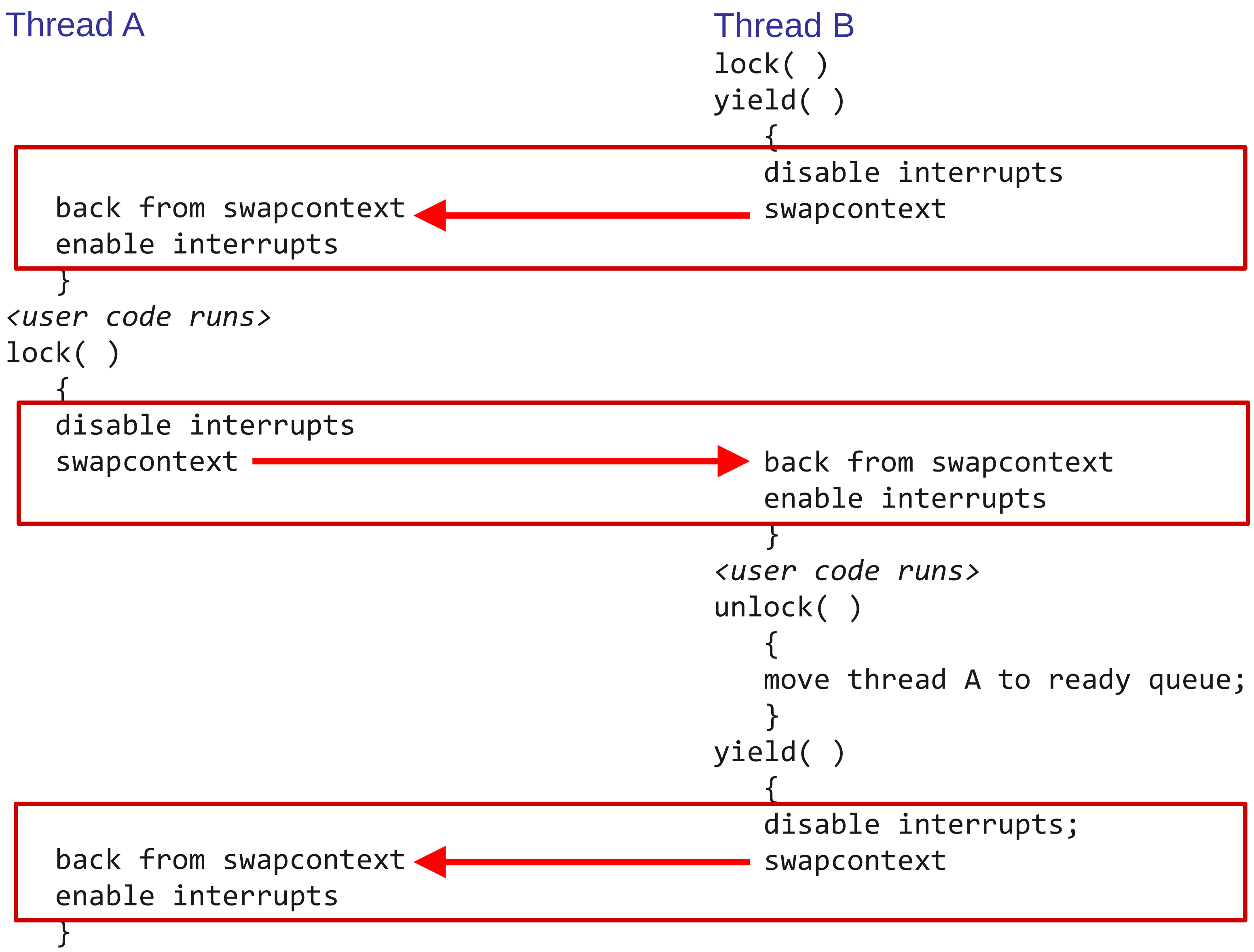
1. Thread must leave interrupts disabled when calling swapcontext.
2. All threads assume interrupts are disabled when returning from swapcontext.
3. Must re-enable interrupts before returning to user code.

## Correct pattern on uniprocessor

```
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;
   }
```

When we switch context, interrupts must be disabled. So, they're still disabled when we're switched back and we must immediately re-enable them. (We can only swap back to somewhere that swapped context.)



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

## Locks on multiprocessors

On uniprocessor, disabling interrupts prevents current thread from being switched out.

But this doesn’t work on a multiprocessor:

1. Other processors are still running threads.
2. Not acceptable to stop all other CPUs from executing.

Solution is an atomic *TestAndSet* in a *spin lock*.

# Atomic Read-Modify-Write:  Test-And-Set

Semantics of test-and-set are to atomically write 1 to a memory location and return the old value.

In Project 2, use *exchange* in `std::atomic`

```
TestAndSet( X )
   {
   old = X;
   X = 1;
   return old;
   }
```

Atomic

## TestAndSet usage

If you are able to change the status from 0 to 1, it means you successfully took the lock.

TestAndSet is atomic, so only one thread will see transition from 0 to 1.

```
// lock is initially free.

int status = 0;

SpinLock( )
   {
   while ( TestAndSet( status ) )
      ;
   }

ReleaseSpinLock( )
   {
   status = 0;
   }
```

# Correct pattern on a multiprocessor

```
int guard = 0;

lock( )
   {
   disable interrupts;
   while ( TestAndSet( 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;
   }
```

```
unlock( )
   {
   disable interrupts;
   while ( TestAndSet( guard ) )
      ;

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

   guard = 0;
   enable interrupts;
   }
```

## Would this work?

```
int guard = 0;

lock( )
   {
   // disable interrupts;
   while ( TestAndSet( 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;
   }
```

```
unlock( )
   {
   disable interrupts;
   while ( TestAndSet( guard ) )
      ;

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

   guard = 0;
   enable interrupts;
   }
```

No, we could get a timer interrupt holding the guard that might want to move us to the ready queue but it wouldn't be able to acquire the guard to do that.

## Would this work?

```
int guard = 0;

lock( )
   {
   while ( TestAndSet( 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;
   }
```

```
unlock( )
   {
   disable interrupts;
   while ( TestAndSet( guard ) )
      ;

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

   guard = 0;
   enable interrupts;
   }
```

No, we could be switched out, holding the guard, locking out the other threads in this spin lock.

# Multi-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
  - Enable interrupts and release guard

## Summary of lock solution

High-level idea:

Atomically add thread to a waiting list and go to sleep.

How did we achieve this?

1. Disable interrupts and TestAndSet( guard ) to protect the critical section.
2. Switch to another thread and hand off the task of enabling interrupts and resetting the guard.

What if no other thread to run?

Atomically suspend CPU with interrupts enabled.

## 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

## Starvation

Starvation is a condition where a thread is perpetually denied resources needed to make progress.

We’ve encountered notion of starvation in discussion of the RW lock in lecture 5, where we were concerned that if lots of new readers kept coming in, a waiting writer might a long, long time, maybe forever.

So, we modified the lock algorithm so that if there were any waiting writers, new readers would wait.

# Avoiding writer starvation

```
void ReadLock ( )
   {
   rwLock.lock( )
   while ( writers > 0 ||
         waitingWriters > 0 )
      waitingReaders.wait( );
   readers++;
   rwLock.unlock( );
   }
```

```
void WriteLock( )
   {
   rwLock.lock( );
   while ( readers > 0 ||
        writers > 0 )
     waitingWriters.wait( );
   writers++;
   rwLock.unlock( );
   }
```

```
void ReadUnlock( )
   {
   rwLock.lock( )
   if ( readers == 1 )
      waitingWriters.signal( );
   readers--;
   rwLock.unlock( )
   }
```

```
void WriteUnlock( )
   {
   rwLock.lock( );
   writers--;
   waitingReaders.broadcast( );
   waitingWriters.signal( );
   rwLock.unlock( );
   }
```

## 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
- The extreme case of starvation where threads wait will wait not just possibly a long time, they will wait forever.

Example: Swapping classes

- Barack is in 482, Michelle is in 485, and they want to switch, but neither wants to give up what they already have.

## Class example

Resources are seats in class

Both Barack and Michelle 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
①  x.lock( );        ②  y.lock( );
③  y.lock( );        ④  x.lock( );
    ...                   ...
    y.unlock( );          x.unlock( );
    x.unlock( );          y.unlock( );
```

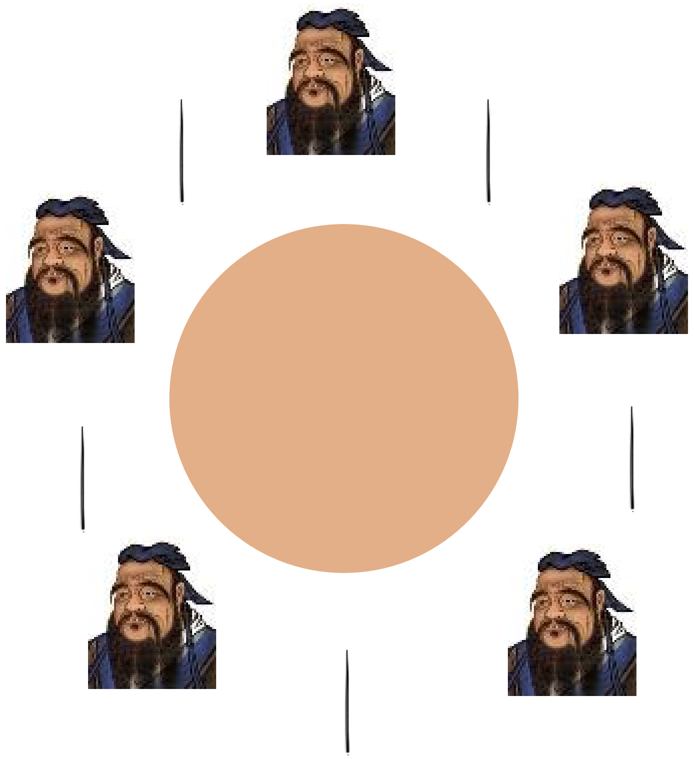
Will a deadlock always occur?

## Dining philosophers

5 philosophers sit at round table.

One chopstick between each pair of philosophers.

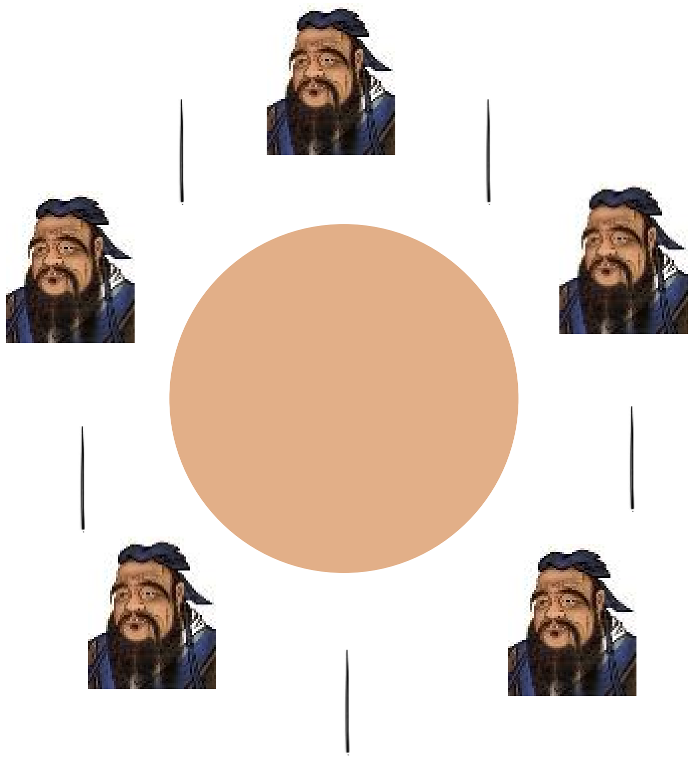
Each philosopher needs two chopsticks to eat.



## Dining philosophers

```
// Proposed algorithm

EatMeal( )
   {
   Wait for right chopstick
      to be free;
   Pick up right chopstick;
   Wait for left chopstick
      to be free;
   Pick up left chopstick;
   Eat the meal;
   Put both chopsticks down;
   }
```



Can this deadlock?

## Generic pattern of resource acquisition and release

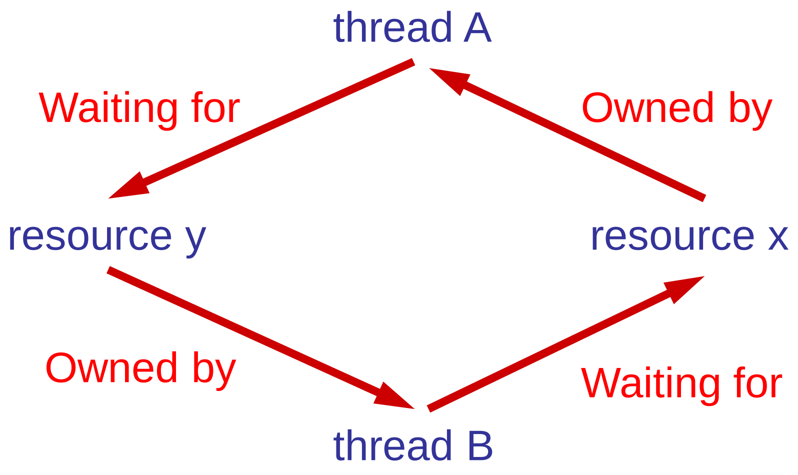
As a program runs, it tends to accumulate resources and then release them at the end.

```
// Phase 1
while ( !done )
   {
   acquire resources;
   do some work;
   }

// Phase 2
release all resources;
```

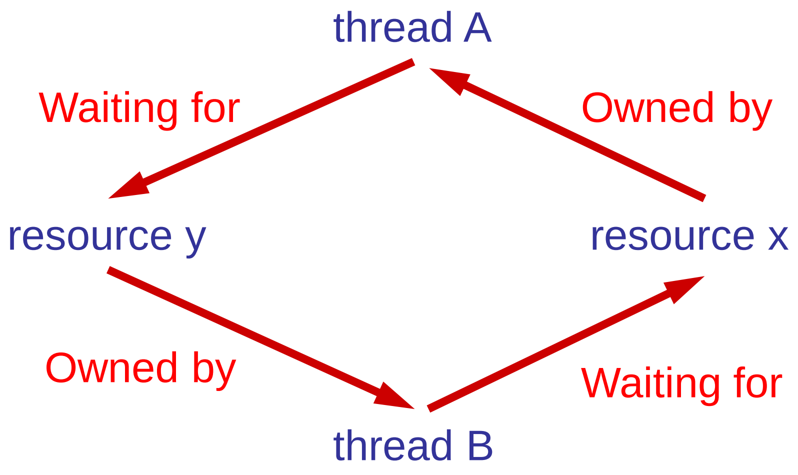
## Waits-for graph

Cycle represents a deadlock.



# Waits-for graph

```
    Thread A                 Thread B
1   x.lock( );           2   y.lock( );
3   y.lock( );           4   x.lock( );
    ...                      ...
    y.unlock( );             x.unlock( );
    x.unlock( );             y.unlock( );
```



# Coping with deadlocks

Alternatives:

1. Ignore.

   Typical OS strategy for application deadlocks.

   Do deadlocked apps consume CPU?

2. 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 a thread holds a lock.

   Databases can rollback to a previous representation invariant state and restart, but general purpose rollback is costly, difficult.

3. Prevent them from occurring.

## Four necessary conditions for deadlock

1. Limited resources: Not enough to serve all threads simultaneously.
2. No preemption: Can’t force threads to give up resources.
3. Hold and wait: Threads hold resources while waiting to acquire others.
4. Cyclical chain of requests.

## Four necessary conditions for deadlock

1. Limited resources: Not enough to serve all threads simultaneously.
2. No preemption: Can’t force threads to give up resources.
3. Hold and wait: Threads hold resources while waiting to acquire others.
4. Cyclical chain of requests.

## Limited resources

Could increase # of resources, e.g., buy more machines, use finer-grained locking.

Not always feasible, e.g., you may only have the number of machines you have, not the number you wish you had, and more locks may introduce new problems.

# Four necessary conditions for deadlock

1. Limited resources: Not enough to serve all threads simultaneously.
2. No preemption: Can’t force threads to give up resources.
3. Hold and wait: Threads hold resources while waiting to acquire others.
4. Cyclical chain of requests.

## No preemption

Some resources can be preempted, e.g., the CPU, if interrupts are enabled.

Others (e.g., locks) are not preemptable.

# Four necessary conditions for deadlock

1. Limited resources:  Not enough to serve all threads simultaneously.
2. No preemption:  Can’t force threads to give up resources.
3. Hold and wait:  Threads hold resources while waiting to acquire others.
4. Cyclical chain of requests.

## Hold-and-wait

Happens because threads acquire resources incrementally.

Two ways to avoid hold and wait:

1. Wait for all resources to be free, then grab all atomically.
2. If cannot get an additional resource, release all and start over.

## Move resource allocation to the beginning.

```
// Phase 1
while ( !done )
   {
   acquire resources;
   do some work;
   }

// Phase 2
release all resources;
```

```
// Phase 1a
acquire all resources;

// Phase 1b
while ( !done )
   do work;

// Phase 2
release all resources;
```

# Atomic acquisition

```
// Initial algorithm

EatMeal( )
   {
   Wait for right chopstick
      to be free;
   Pick up right chopstick;
   Wait for left chopstick
      to be free;
   Pick up left chopstick;
   Eat the meal;
   Put both chopsticks down;
   }
```

```
// Revised algorithm

mutex m;
cv c;

EatMeal( )
   {
   m.lock( )
   while ( either chopstick busy )
      c.wait( &m );
   Pick up left chopstick;
   Pick up right chopstick;
   m.unlock( );
   Eat the meal;
   m.lock( );
   Put both chopsticks down;
   c.broadcast( );
   m.unlock( );
   }
```

## But what happens if it’s repeated?

```
// Pick up the chopsticks and
// set them down again for
// each bite.

EatMeal( )
   {
   while ( !done )
      TakeBite( );
   }
```

```
mutex m;
cv c;

TakeBite( )
   {
   m.lock( )
   while ( either chopstick busy )
      c.wait( &m );
   Pick up left chopstick;
   Pick up right chopstick;
   m.unlock( );
   Eat one bite;
   m.lock( );
   Put both chopsticks down;
   c.broadcast( );
   m.unlock( );
   }
```

If all necessary resources are acquired atomically and everyone only needs them once, deadlock cannot occur.

But if resource requests keep coming in, starvation can occur because there’s no guarantee who will run next.

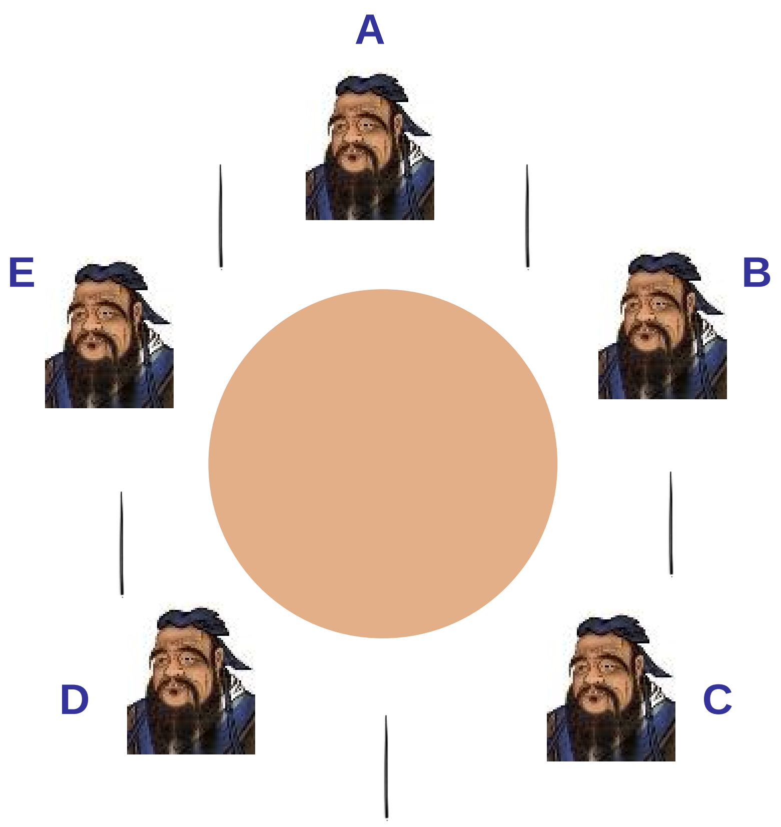
*Scenario:*

A and C take a bite.

B and D take a bite.

Then A and C take another bite.

E starves (at least for a while.)



## Tends to be wasteful

Forces the acquisition of all possible resources that might ever be needed even though most may never actually be needed.

May even create new deadlock patterns.

```
// Phase 1a
acquire all resources;

// Phase 1b
while ( !done )
   do work;

// Phase 2
release all resources;
```

## Four necessary conditions for deadlock

1. Limited resources: Not enough to serve all threads simultaneously.
2. No preemption: Can’t force threads to give up resources.
3. Hold and wait: Threads hold resources while waiting to acquire others.
4. Cyclical chain of requests.

## Eliminating circular chain

Define a global order over all resources by numbering them.

The numbering can be arbitrary but is usually least precious first to most precious last (so you hold the most precious resources the shortest amount of time.)

All threads acquire resources in this order.

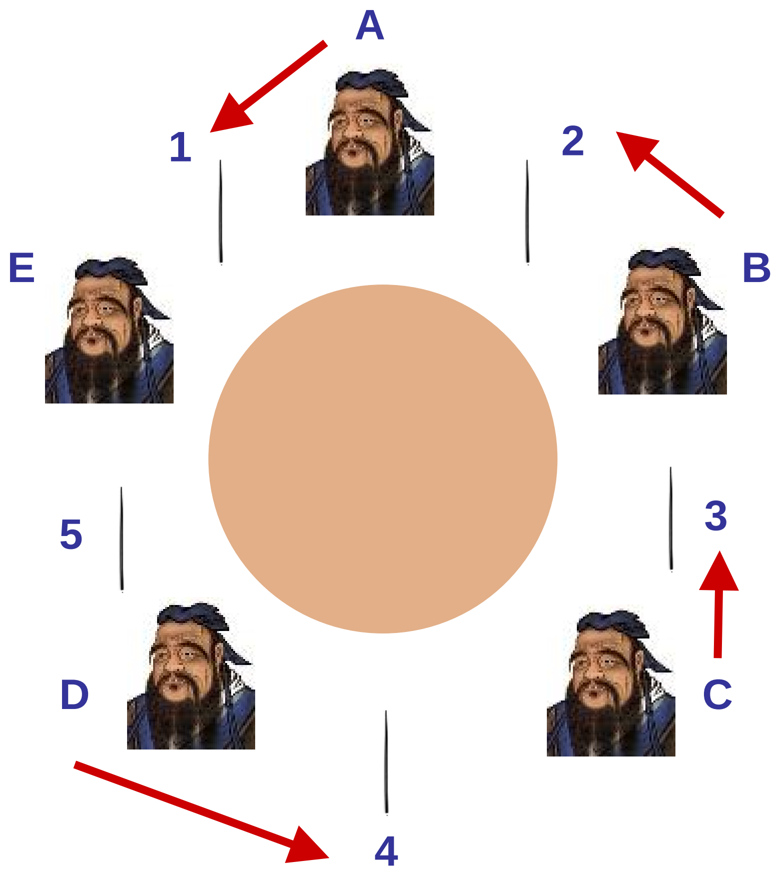
# Dining philosophers

Solution:

Pick up the lower number chopstick first, then pick up the higher number chopstick.

Might get to a situation where E blocks but D proceeds, then C, etc., until A finishes, then E proceeds.

No deadlock.



## Eliminating the circular chain

But what if you already hold the higher number resource but also want the lower number resource?

You must first give up the higher number, then take them both in order.

You must rewrite your code to ensure you follow the pattern. It's a bug and you have to fix it.

Thread A
```
x.lock( );
y.lock( );
...
y.unlock( );
x.unlock( );
```

Thread B
```
y.lock( );
x.lock( );
...
x.unlock( );
y.unlock( );
```

⇨

Thread B
```
y.lock( );
y.unlock( );
x.lock( );
y.lock( );
...
y.unlock( );
x.unlock( );
```

This completes the section on threads and synchronization.

Up next: Virtual memory.