CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 27: Concurrency

Instructors: Dan Garcia, Justin Yokota

Agenda

- Race Conditions
- Locks and critical segments

Respond at **pollev.com/jy314**

Text **JY314** to **22333** once to join, then **A, B, C, or D**

Get a 1:2:3:4 ratio of votes

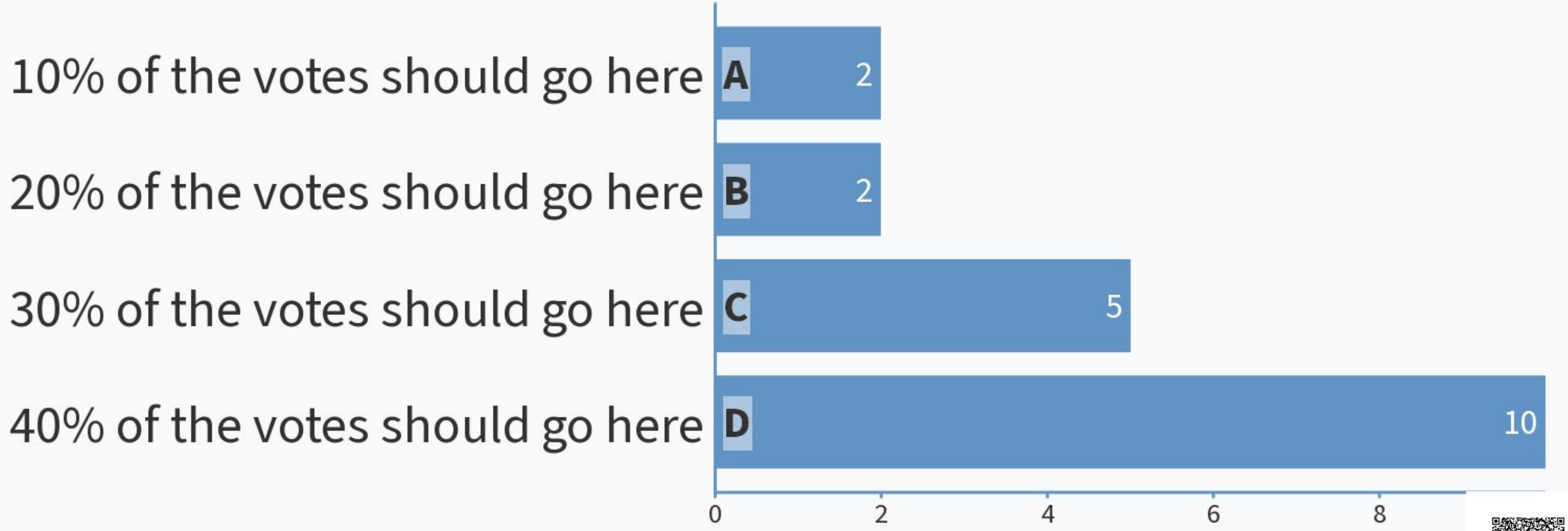
10% of the votes should go here	**A**
20% of the votes should go here	**B**
30% of the votes should go here	**C**
40% of the votes should go here	**D**

Tot .9

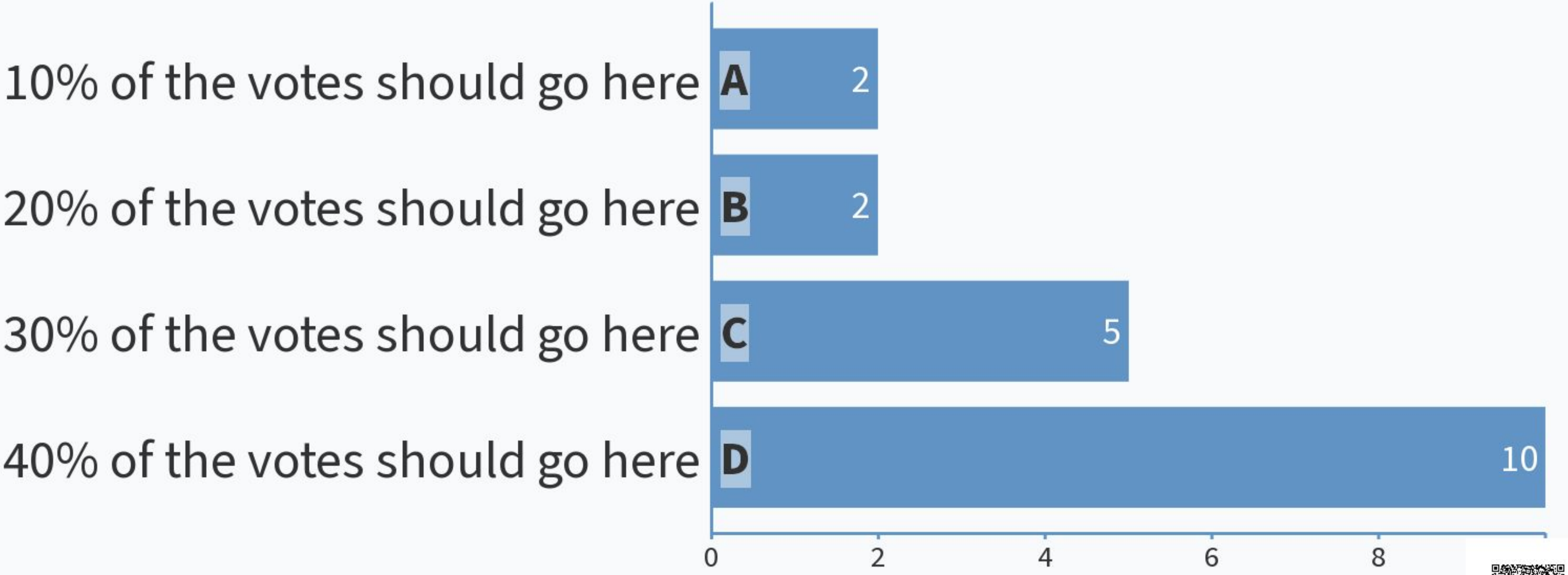
Respond at **pollev.com/jy314**

Text **JY314** to **22333** once to join, then **A, B, C, or D**

Get a 1:2:3:4 ratio of votes



Get a 1:2:3:4 ratio of votes



Data Races/Race Conditions

- Note that when we ran Hello World parallel, we ended up with the threads running in random order
  - In fact, every time we run Hello World, we get a different order!
  - The x values stayed largely in-order, but didn't always strictly increase
- Recall the OS can choose whichever threads it wants to run, and change threads at any time
- This is one of the biggest downsides to multithreading: A multithreaded program is no longer deterministic, and will have a random execution order every time we run the program.
- Formally, a multithreaded program is only considered correct if ANY interlacing of threads yield the same result.

Data Race: Example

- If we run this code on 4 threads, what possible values could x be at the end?

```
int x = 0;
#pragma omp parallel {
    x = x + 1;
}
```

If we run this code on 4 threads, how many different possible values could x be at the end?

- 1
- 2
- 3
- 4
- 5
- 6

To 0

If we run this code on 4 threads, how many different possible values could x be at the end?

1

2

3

4

5

6

If we run this code on 4 threads, how many different possible values could x be at the end?

1

2

3

4

5

6

Data Race: Example

- To analyze this, we need to see the equivalent assembly code
  - C will compile to x86, but we can still do a correct analysis by compiling to RISC-V, since we're mainly trying to reduce the code to atomic instructions. We can assume that no two atomic instructions happen simultaneously.
- Only the loads and stores affect shared memory, so we only need to consider the different ways we can order the loads and stores
- Even with this, there are $8!/(2!)^4=2520$ different possible orders
  - Can use the fact that all the threads are identical to reduce this to 105 orders, but still too many to check manually

```
sw x0 0(sp)
```

`lw t0 0(sp)`	`lw t0 0(sp)`	`lw t0 0(sp)`	`lw t0 0(sp)`
`addi t0 t0 1`	`addi t0 t0 1`	`addi t0 t0 1`	`addi t0 t0 1`
`sw t0 0(sp)`	`sw t0 0(sp)`	`sw t0 0(sp)`	`sw t0 0(sp)`

Data Race: Example

- Case 1: All the threads run one at a time
  - Purple thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread reads x = 1
  - Brown thread stores x = 2
  - Red thread reads x = 2
  - Red thread stores x = 3
  - Blue thread reads x = 3
  - Blue thread stores x = 4
- Final value: 4

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
```

Data Race: Example

- Case 2: The threads are perfectly interleaved
  - Purple thread reads x = 0
  - Red thread reads x = 0
  - Brown thread reads x = 0
  - Blue thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread stores x = 1
  - Red thread stores x = 1
  - Blue thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Case 3: Same as case 1, except purple's store happens last
  - Purple thread reads x = 0
  - Brown thread reads x = 0
  - Brown thread stores x = 1
  - Red thread reads x = 1
  - Red thread stores x = 2
  - Blue thread reads x = 2
  - Blue thread stores x = 3
  - Purple thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Some other ordering?
  - We can find orderings that give x=2,3
- Can we do any more/less?
- Can't go above 4
  - Only 4 "+1s" overall, so can't increase to 5 or more
- Can't go below 1
  - The smallest value that can be loaded by a thread is 0, so the smallest value that can be stored is 1. Therefore, the last store must be at least 1.
- Therefore, we can get any value between 1 and 4

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Data Race Example: Retrospective

- In practice, most times you run this code, you get 4
  - Each thread is small enough that it's unlikely to get interrupted
- Empirical tests suggest an error rate around 0.01%.
- In summation, race condition bugs are:
  - Rare
  - Nondeterministic
  - Silent-failing (you get the wrong answer instead of crashing the program)
- Very difficult to debug. You have been warned…

Avoiding Data Races

- Formally, a multithreaded program is only considered correct if ANY interlacing of threads yield the same result.
- If you make sure that each thread works on independent data (no two threads write to the same value, or read a value that another thread wrote to), you can guarantee correctness
- The hardest part of multithreading is maintaining correctness while also speeding up the code, but this ends up being a fairly transferable skill to management
  - If you can coordinate a group of threads to perform a task, you can coordinate a group of people to perform a task more easily.
- How to handle cases where coordination is mandatory?

Resolving Data Races: Buying Milk

- Alice and Bob want to buy milk. How do they do this without communicating?
- Assumption: Must decide procedure beforehand
- Assumption: Both people get the same instructions (though we can refer to one person with names)

Buying Milk: Attempt 1


```
If milk not in fridge:
    Go to store and buy milk
    Put milk in fridge
```

Buying Milk: Problem with Attempt 1


```
If milk not in fridge: #No milk in fridge, so True
If milk not in fridge: #No milk in fridge, so True
    Go to store and buy milk #Bob bought milk
    Go to store and buy milk #Alice bought milk
    Put milk in fridge #1 milk in fridge
    Put milk in fridge #2 milks in fridge
```

Maybe we should have put a note on the fridge…

Buying Milk: Attempt 2


```
If note not on fridge:
    If milk not in fridge:
        Put note on fridge
        Go to store and buy milk
        Put milk in fridge
        Take note off fridge
```

Buying Milk: Problem with Attempt 2


```
If note not on fridge: #No note on fridge
    If milk not in fridge: #No milk in fridge
If note not on fridge: #No note on fridge
        Put note on fridge #Now there's a note on the fridge…
    If milk not in fridge: #Still no milk in fridge
        Put note on fridge #Two notes on fridge
        Go to store and buy milk
        Put milk in fridge #1 milk in fridge
        Take note off fridge #1 note on fridge
        Go to store and buy milk
        Put milk in fridge #2 milks in fridge
        Take note off fridge #0 notes on fridge
```

Buying Milk: Attempt 3


```
If note not on fridge:
    If milk not in fridge:
        Put note on fridge
        If two notes on fridge:
            goto End
        Go to store and buy milk
        Put milk in fridge
End:    Take note off fridge
```

Buying Milk: Problem with Attempt 3


```
If note not on fridge: #No note on fridge
If note not on fridge: #No note on fridge
    If milk not in fridge: #No milk in fridge
    If milk not in fridge: #No milk in fridge
        Put note on fridge #One note on fridge
        Put note on fridge #Two notes on fridge
        If two notes on fridge: #Two notes on fridge
        If two notes on fridge: #Two notes on fridge
            goto End #Go to End
            goto End #Go to End
        Go to store and buy milk
        Go to store and buy milk
        Put milk in fridge
        Put milk in fridge
End:  Take note off fridge #1 note on fridge
End:  Take note off fridge #0 notes on fridge, 0 milk in fridge
```

Buying Milk: Problems with all our attempts

- Regardless of how we do it, this doesn't work if the instructions happen to be perfectly interleaved
- Why?
  - Every branch will have the same result, since no instruction checks a value AND writes to a memory location at the same time.
  - If both people follow the same code path, we either get no milk, or two milk.
- Well, there is one strategy that works

Buying Milk: Attempt 4


```
If name is Alice:
    If milk not in fridge:
        Go to store and buy milk
        Put milk in fridge
```

Buying Milk: Problem with Attempt 4

- If we give the task to Alice, that works; we'll get exactly 1 milk, guaranteed
- Problem: What if Alice is really busy and can't check the fridge for a few days?
- We end up getting milk later than we want to.

Atomic Operations

- Regardless of how we do it, this doesn't work if the instructions happen to be perfectly interleaved
- Why?
  - Every branch will have the same result, since no instruction checks a value AND writes to a memory location at the same time.
  - If both people follow the same code path, we either get no milk, or two milk.
- Solution: Create an instruction that checks a value AND writes to a memory location at the same time.
- Known as "atomic" instructions because they do two things but are indivisible.
- RISC-V atomic extension example:
  - `amoswap.w rd rs2 (rs1)`: rd = 0(rs1), 0(rs1) = rs2 atomically
  - A bit hard to use directly, but using this, we can make synchronization primitives

Locks

- A lock is an object which helps with synchronization
- Essentially, each thread can try to "acquire" a lock, but only one thread can have the lock at a given time.
  - Think bathroom stall lock; only one person can use the stall at a time, and we use atomics to make sure two people don't go into the same stall at the same time
- Formally, has two operations
  - acquire: Tries to acquire the lock. If successful, keep going. Otherwise, wait a bit and try again later.
  - release: Unlocks the lock and continues. Only works if we had the lock to start with.
  - Optional but common: try-acquire: Same as acquire, but if the lock is being used, return false and let the thread continue running
- Code surrounded by a lock is called a "critical section", because only one thread is allowed to run that section at a time.

Buying Milk: Attempt 5


```
Acquire fridgelock
    If milk not in fridge:
        Go to store and buy milk
        Put milk in fridge
Release fridgelock
```

Buying Milk: Attempt 5

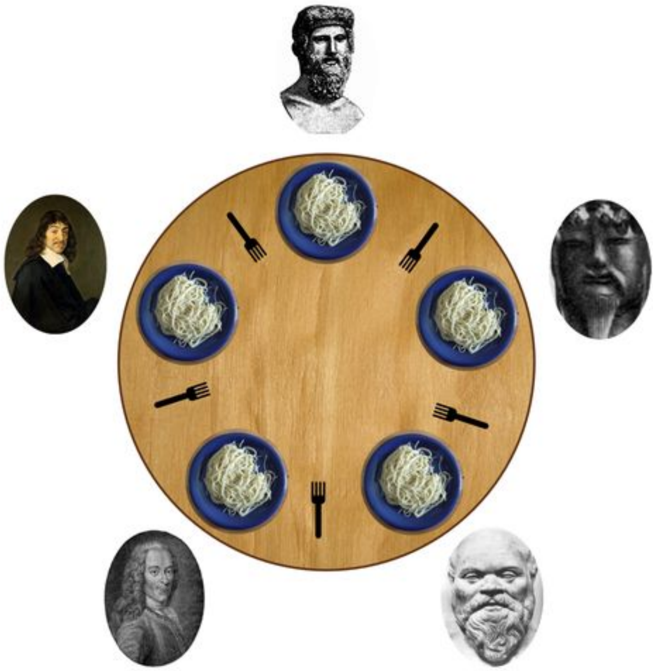

```
Acquire fridgelock #Alice has the lock
Acquire fridgelock #Bob can't get the lock, so needs to wait
    If milk not in fridge: #No milk in fridge
        Go to store and buy milk
        Put milk in fridge #1 milk in fridge
Release fridgelock #Alice releases lock. Bob can now get the lock
    If milk not in fridge: #1 milk in fridge
        Go to store and buy milk
        Put milk in fridge
Release fridgelock #Bob releases lock
```

Locks: Downsides

- Using a lock inherently means you need to pause one thread while waiting for another thread to run the critical segment
- Ends up making some parts of your code serial
  - Amdahl's Law strikes again!
- Is it possible for all threads to get stuck?

Thought Experiment: The Dining Philosophers Problem

- Five pre-COVID philosophers are sitting at a table eating spaghetti
- The philosophers will alternate between eating and thinking
- Between each philosopher is one chopstick
- To eat spaghetti, a philosopher must pick up two chopsticks (the one immediately to their left, and the one immediately to their right). A philosopher will take a bite of spaghetti, put the chopsticks back down, and resume thinking.
- Philosophers can't speak or coordinate
- Goal: prevent the philosophers from starving



The Dining Philosophers Problem: Naive Solution

While True:

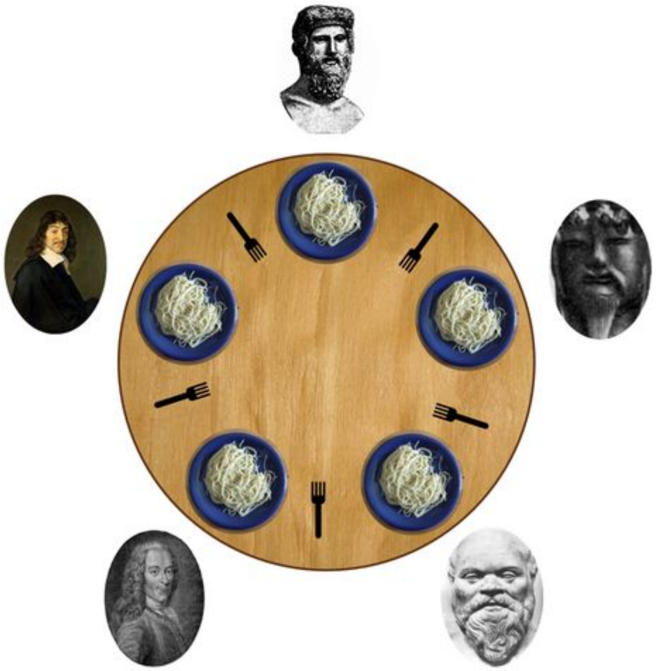
Acquire left chopstick; think until it's available

Acquire right chopstick; think until it's available

Take a bite of spaghetti

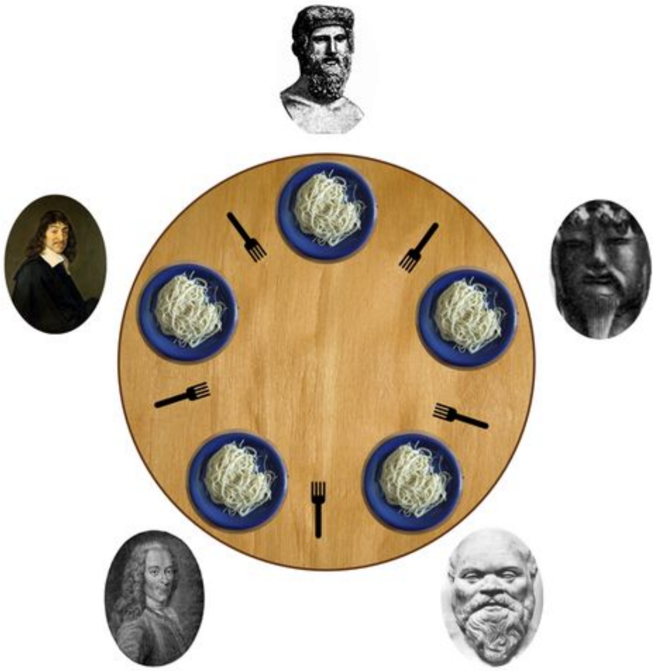
Put down left chopstick

Put down right chopstick



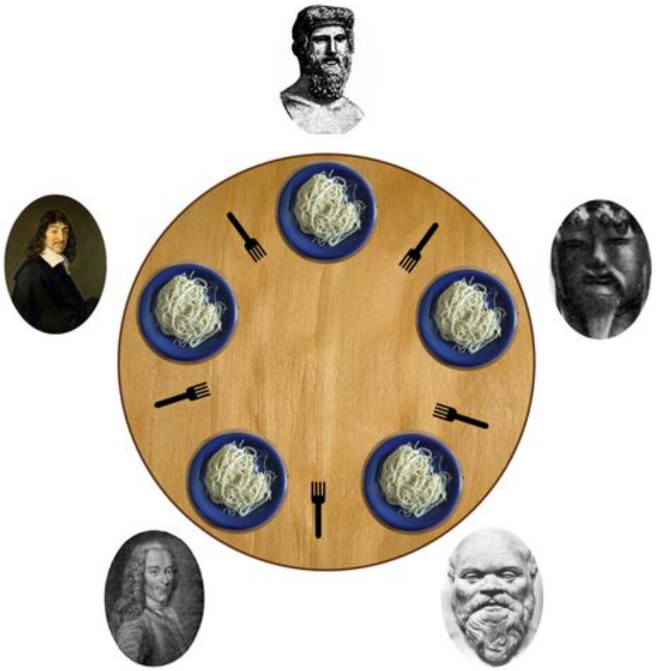
The Dining Philosophers Problem: Naive Solution Problem

- What happens?
- Imagine all philosophers grab the left chopstick at the same time
- All the philosophers wait for the right chopstick to be available, and get stuck thinking forever
- The five philosophers starve to death waiting for chopsticks
- This is known as a deadlock; if this happens in your program, every thread gets stuck waiting for some other thread to finish work, so the program freezes.



The Dining Philosophers Problem: Resolutions

- An active topic in concurrency programming
- Several common solutions; goal is to get rid of the symmetry of the problem:
  - Only let four philosophers in to the dining room at any given time
  - Have a "manager" that assigns both chopsticks at once
  - Choose one philosopher to pick up the right chopstick first before the left chopstick



Critical sections

- Fortunately, OpenMP gives you some commands that let you use critical segments completely safely
- #pragma omp barrier
  - Forces all threads to wait until all threads have hit the barrier
- #pragma omp critical
  - Creates a critical segment in parallel code; only one thread can run a critical segment at a time.
- Using these guarantees you avoid deadlocks, so we'll recommend using these exclusively in this class.
- Locks get used significantly more in CS 162 (Operating Systems)

Parallel Hello World with Critical Segments


```
#include <stdio.h>
#include <omp.h>
int main () {
    int x = 0; //Shared variable
     #pragma omp parallel
     {
          int tid = omp_get_thread_num(); //Private variable
          #pragma omp critical
          {
               x++;
               printf("Hello World from thread %d, x = %d\n", tid, x);
          }
          #pragma omp barrier
          if(tid==0) {
               printf("Number of threads = %d\n", omp_get_num_threads());
          }
     }
     printf("Done with parallel segment\n");
}
```