Deadlock

- Starvation vs. Deadlock
- Dining Philosophers
- Conditions for Deadlock
- Deadlock Recovery & Avoidance
- Deadlock Prevention

Starvation vs. Deadlock

Starvation vs. Deadlock

- With locks, the most important thing is *correctness:*
  - No races
  - No corruption of data
- But there are other ways to fail
  - Starvation / Unfair
  - Deadlock

Starvation vs. Deadlock

- **Starvation** is when the system continues to make progress, but one or more processes are blocked endlessly
- Not *formally* starved forever, but sometimes there's no end to the wait, when the load is high
  - Example: Turning left into heavy traffic

Starvation vs. Deadlock

- Many systems are susceptible to starvation, in worst-case scenarios
  - Example: Unable to write to a lock variable, too much contention for the cache line

- Sometimes we live with it, if it’s rare
- But design code to avoid it
  - “Under reasonable load, our system is starvation-free...”

Starvation vs. Deadlock

- **Starvation:** must be possible for the condition to end

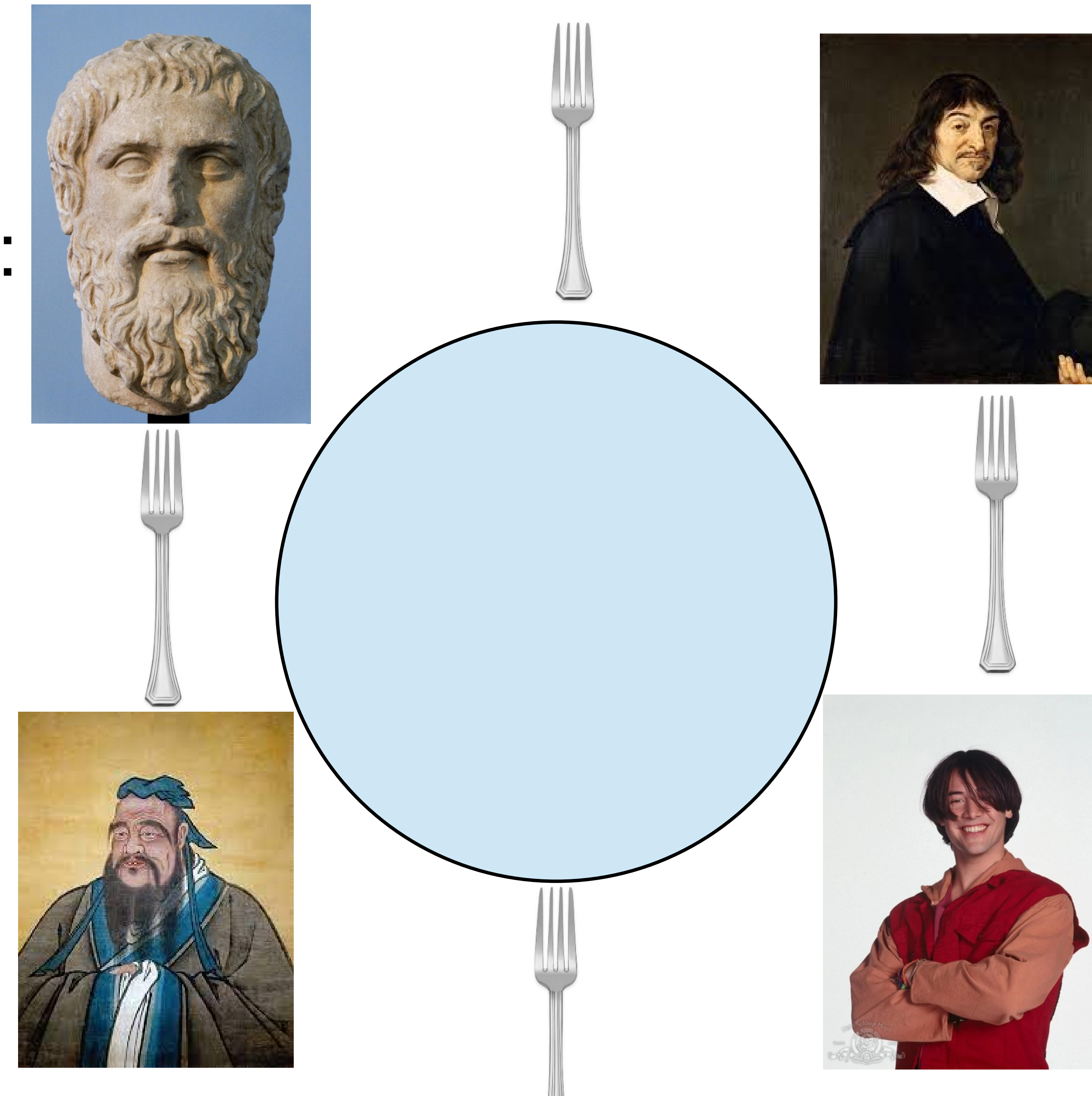
- **Deadlock** is when some processes have reached a state where it is *impossible* for any of them to make any more progress
  - Some processes may still be running OK
    - Though often, not for long!

Deadlock

Dining Philosophers

Dining Philosophers

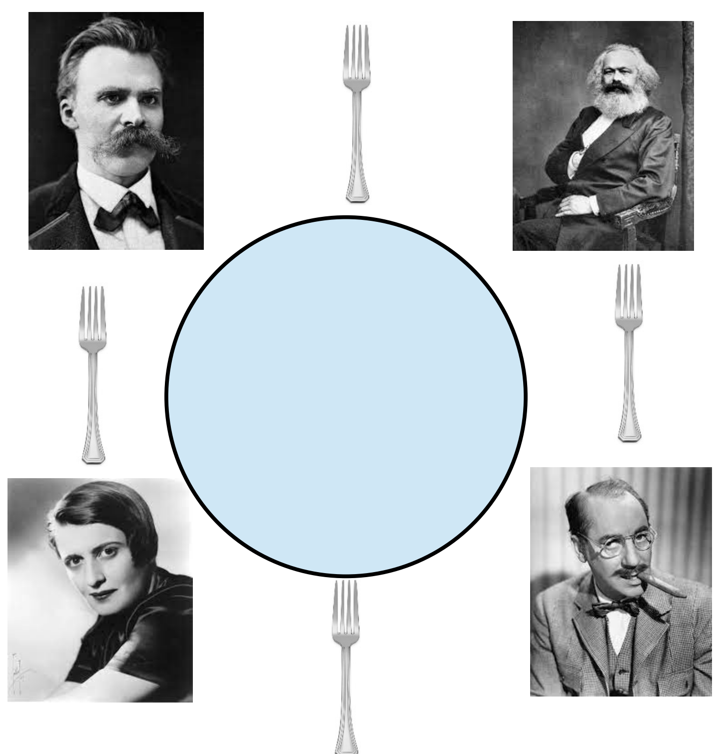
- N philosophers
- Each alternates:
  - Think
  - Eat
- Each needs 2 forks to eat



Dining Philosophers

```
def philosopher(n):
  l_fork = Fork( n-1)
  r_fork = Fork((n+1) % count)

  while True:
    think()
    l_fork.grab()
    r_fork.grab()
    eat()
    l_fork.drop()
    r_fork.drop()
```



Dining Philosophers

```
def philosopher(n):
  l_fork = Fork( n-1)
  r_fork = Fork((n+1) % count)

  while True:
    think()
    l_fork.grab()
    r_fork.grab()
    eat()
    l_fork.drop()
    r_fork.drop()
```

Q: How long will this run without any problems?

Dining Philosophers

```
def philosopher(n):
  l_fork = Fork( n-1)
  r_fork = Fork((n+1) % count)

  while True:
    think()
    l_fork.grab()
    r_fork.grab()
    eat()
    l_fork.drop()
    r_fork.drop()
```

Q: How long will this run without any problems?

A: Impossible to tell; it's a race! How long are `think()` and `eat()` ? How many philosophers?

Dining Philosophers

```
def philosopher(n):
  l_fork = Fork( n-1)
  r_fork = Fork((n+1) % count)

  while True:
    think()
    l_fork.grab()
    r_fork.grab()
    eat()
    l_fork.drop()
    r_fork.drop()
```

Q: Give an example of deadlock

Dining Philosophers

```
def philosopher(n):
  l_fork = Fork( n-1)
  r_fork = Fork((n+1) % count)

  while True:
    think()
    l_fork.grab()
    r_fork.grab()
    eat()
    l_fork.drop()
    r_fork.drop()
```

Q: Give an example of deadlock

A: All philosophers grab their left fork before any grab their right

Deadlock

- Dining philosophers deadlocks when each philosopher is waiting on the next to release a fork
  - Circular
  - Blocks forever

- But if *all* processes block, then *none* can make progress

Deadlock

Conditions for Deadlock

Conditions for Deadlock

- Deadlock requires **four conditions:**
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait

Conditions for Deadlock

- **Mutual exclusion** means that it's not possible for two processes to possess the same resource at the same time
- If you use locks, this is just how they work
- But this can apply to any system that allocates resources
  - Routes through a train network
  - Seats on a plane
  - Enrollment in a class

Conditions for Deadlock

- **Hold and wait** means that each process that is involved in the deadlock both (a) holds at least one resource; and (b) is block waiting for another

- If you own nothing, you cannot cause deadlock
- If you never block, you cannot cause deadlock

Conditions for Deadlock

- **No preemption** means that no one can take away a resource, once it's been promised

- This is usually how things work
  - Hard to write a program otherwise!
- People have experimented with breaking this rule – it's ugly

Conditions for Deadlock

- **Circular wait** means that the set of blocked processes have to form a cycle

- If no cycle, then the process at one end will eventually finish its work
- Then, the next process, and the next
- But if a cycle, then no process ever finishes

Deadlock

Deadlock Recovery & Avoidance

Deadlock Recovery & Avoidance

- Is it possible to ***break*** deadlock once it has happened?
  - Generally, no – unless you ***kill*** a process
- Some people have tried it, usually by taking away a resource
  - What does your program do next? Start over? From where???

- Which deadlock condition does this prevent?

Deadlock Recovery & Avoidance

- Is it possible to ***avoid*** deadlock in the first place?
  - Yes, if you make locking more complex
- **Banker's Algorithm** (not Baker's Algorithm!)
  - Pre-declare which locks you want
  - Or, at least, the ***max*** that you want
  - Block until all are available, gain none until then
  - Which deadlock condition does this prevent?

Banker’s Algorithm

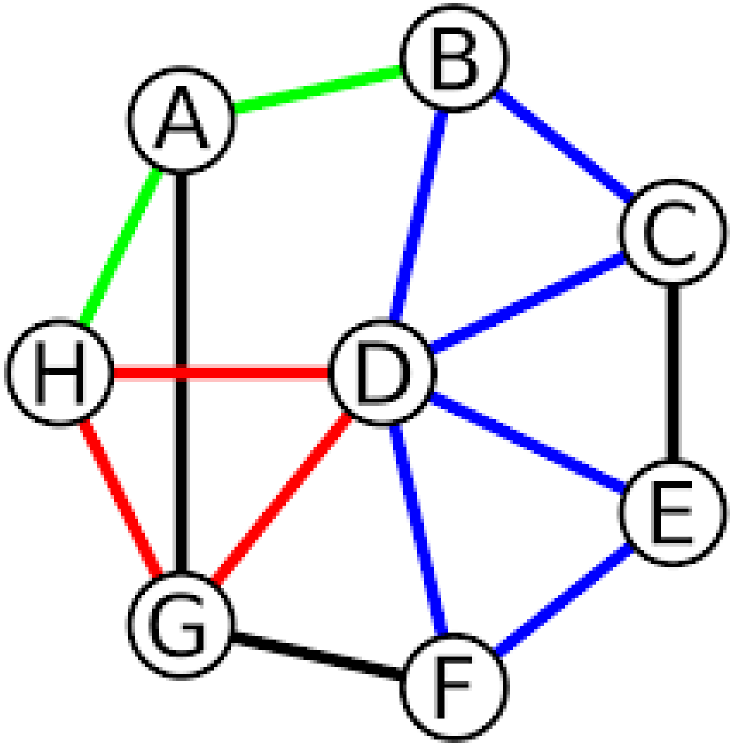
- Banker’s Algorithm prevents Hold-and-Wait
  - Thus, deadlock is impossible!

- But what are the downsides, in practical code?
  - Need strict plan of all resources
  - What if you call library functions?
  - What if the set of resources is hard to discover?

Banker's Algorithm

Example:

- Algorithm starts at one node
- Reads neighbors
- Removes one link
- Needs lock of 1st node to read neighbors
- Needs lock of both nodes to remove the link



Deadlock

Deadlock Prevention

Deadlock Prevention

Inspiration:

- Circular Wait is impossible if we have a global order for all locks, and gain them in order
  - If you block, you always block on an ***earlier*** lock

- How does this change Dining Philosophers?

Deadlock Prevention

```
def philosopher(n):
  l_fork = Fork( n-1)
  r_fork = Fork((n+1) % count)

  if (l_fork > r_fork):
    (l_fork,r_fork) = (r_fork,l_fork)

  while True:
    ...
```

Deadlock Prevention

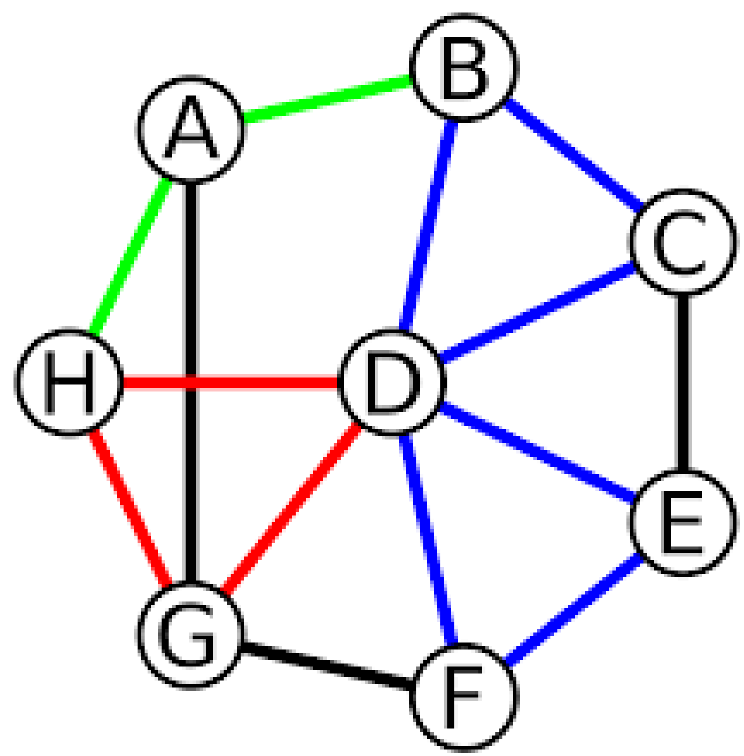
- We fix Dining Philosophers such that each philosopher gains their fork in the same order
  - In practice, this means that one process grabs “right,left” instead of “left,right”

- ***Assymetry:*** We have a special philosopher, different than all the rest
  - Does this introduce starvation? Worth considering!

Deadlock Prevention

- Ordering easy to detect on a graph
- But may be harder to gain locks in order
  - Start at B, go to A; stuck!

- Can we make this smarter?



Deadlock Prevention

Inspiration:

- Hold and Wait is impossible if we ***use non-blocking operations*** when attempting a lock, while we already own one
  - OK to block on first lock
  - OK to gain out-of-order

- Combined: use non-blocking when violating the order rules

Deadlock Prevention

- `trylock()` is a function which attempts to gain a lock
  - If it succeeds, it's exactly like `lock()`
  - If it fails, return an error code

Deadlock Prevention

- What do do if a `trylock()` fails?

- ***Must not*** just spin, waiting for it to succeed
  - Why?
- Instead, must unlock everything & start again
  - Could simply repeat steps, or could try in a new order
  - Don't need `trylock()` if gaining in-order

Dining Philosophers

```
l_fork.grab()


if l_fork < r_fork:
  r_fork.grab()           # BLOCKING!
else:
  r_fork.try_grab()       # NOT
  if FAIL:
    l_fork.drop()         # RELEASE
    r_fork.grab()         # REVERSE ORDER
    l_fork.grab()
```