Deadlock

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

Deadlock

Starvation vs. Deadlock

Starvation vs. Deadlock

- With locks, what are we trying to eliminate?
  - No races
  - No corruption of data
- Even with locks that are paired with an appropriate release, 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 Example

Process X (Priority 4)	Process Y (Priority 2)	Process Z (Priority 5)

`fork1(pr=2) #Y`
`fork1(pr=5) #Z`
`while(join()) {`
`  fork1(pr=2) #Y`
`}`
	`gain(x)`
	`CRITICAL`
	`SECTION`
	`release(x)`
		`gain(x)`
		`CRITICAL`
		`SECTION`
		`release(x)`
		
		Starved Process

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 theoretically 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!

Starvation vs. Deadlock

Deadlock is when some processes have reached a state where it is *impossible* for any of them to make any more progress

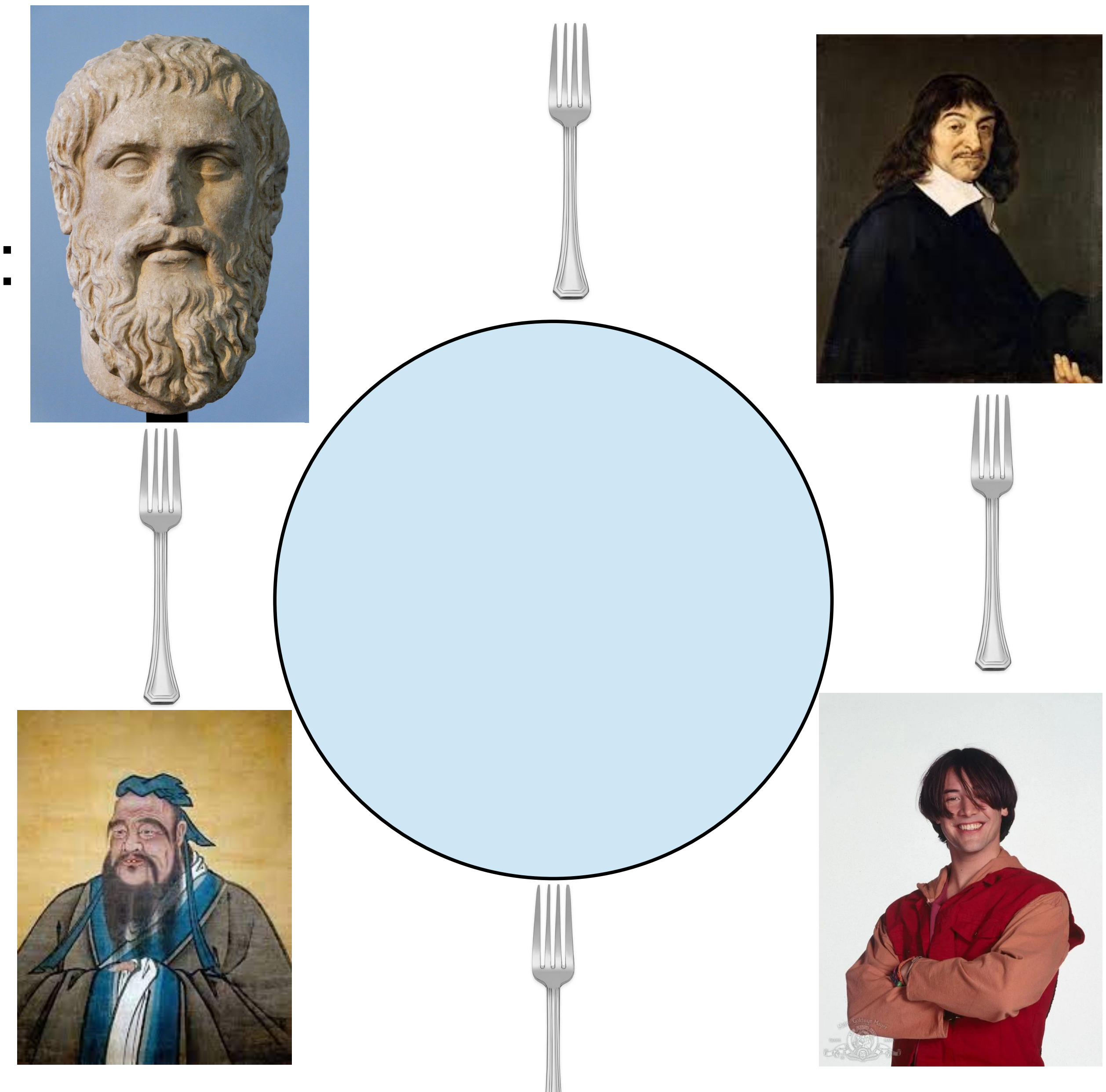
Discuss with your neighbor, can you come up with an example of this?

Deadlock

Dining Philosophers

Dining Philosophers

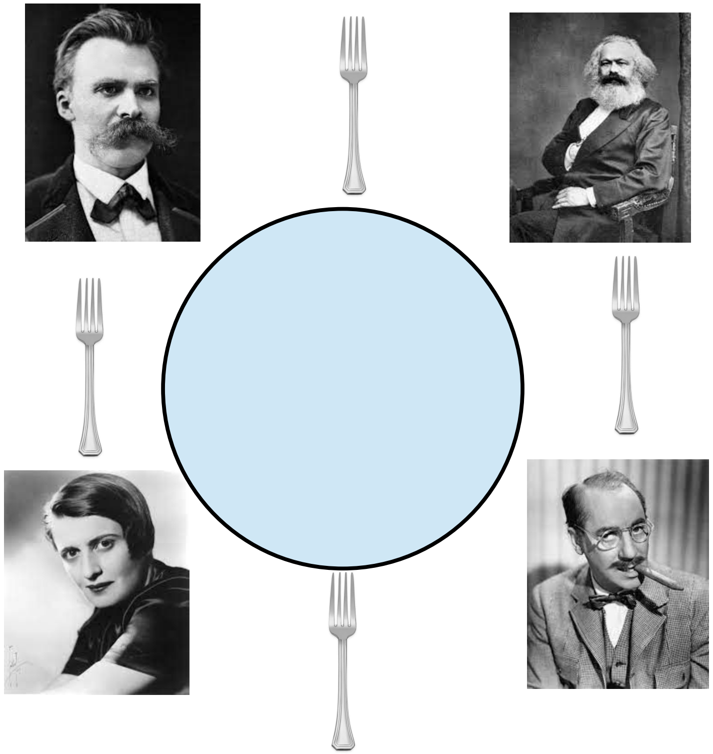
- N philosophers
- Each alternates:
  - Think
  - Eat
- Each needs
  2 forks to eat



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: 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

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?

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 blocked waiting for another

- If you own nothing, you cannot cause deadlock
- If you never block, you cannot cause deadlock
  - Spin-lock counts as blocking, since you never end!

Conditions for Deadlock

- **No preemption** means that no one can take away a resource, once it's been promised
- This is usually how locks work
  - Hard to write a program otherwise!
- ***Question:*** Could something like this be implemented with Dining Philosophers?

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

ICA: Deadlock

The four conditions of deadlock:

Mutual Exclusion

Hold and Wait

No Preemption

Circular Wait

Can a single process cause a deadlock situation? Why or why not?

Deadlock Recovery

Deadlock Recovery

- Is it possible to ***break*** deadlock once it has happened?
  - Generally, no – unless you ***kill*** a process
- Could also try to intelligently take away resource to break. Not easy to do right!

Deadlock Recovery

- Is it possible to ***break*** deadlock once it has happened?
  - Generally, no – unless you ***kill*** a process
- Could also try to intelligently take away resource to break. Not easy to do right!

Can you explain to your neighbor how to do this with Dining Philosophers, why it would help?

Deadlock Avoidance & Prevention

Deadlock Avoidance & Prevention

- 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 the max number of each resource / lock you want
  - Block until all are available

Banker's Algorithm

Processes (allocated): The number of resources each process currently has allocated	Processes (maximum): The max number of each resource each process could allocate
Available system resources: The total amount available of each resource on the system	Need = (maximum - allocated) The remaining amount of each resource a process may request in order to complete

Banker's Algorithm

```
Processes (allocated):

    A B C D
P1  1 2 2 1
P2  1 0 3 3
P3  1 2 1 0
```

```
Processes (maximum):

    A B C D
P1  3 3 2 2
P2  1 2 3 4
P3  1 3 5 0
```

```
Available system resources:

      A B C D
Free  3 1 1 2
```

```
Need = (maximum - allocated)

    A B C D
P1  2 1 0 1
P2  0 2 0 1
P3  0 1 4 0
```

Example Starting State

Banker’s Algorithm

Processes (allocated):

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Processes (maximum):

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

Available system resources:

	A	B	C	D
Free	3	1	1	2

Need = (maximum - allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Process 3 requests 1 unit of resource C

Banker's Algorithm

```
Processes (allocated):

    A B C D
P1  1 2 2 1
P2  1 0 3 3
P3  1 2 2 0
```

```
Processes (maximum):

    A B C D
P1  3 3 2 2
P2  1 2 3 4
P3  1 3 5 0
```

```
Available system resources:

     A B C D
Free 3 1 0 2
```

```
Need = (maximum - allocated)

    A B C D
P1  2 1 0 1
P2  0 2 0 1
P3  0 1 3 0
```

Process 3 requests 1 unit of resource C

Banker's Algorithm

```
Processes (allocated):

    A B C D
P1  1 2 2 1
P2  1 0 3 3
P3  1 2 1 0
```

```
Processes (maximum):

    A B C D
P1  3 3 2 2
P2  1 2 3 4
P3  1 3 5 0
```

```
Available system resources:

      A B C D
Free  3 1 1 2
```

```
Need = (maximum - allocated)

    A B C D
P1  2 1 0 1
P2  0 2 0 1
P3  0 1 4 0
```

Process 2 requests 1 unit of resource B

Banker's Algorithm

```
Processes (allocated):

    A B C D
P1  1 2 2 1
P2  1 1 3 3
P3  1 2 1 0
```

```
Processes (maximum):

    A B C D
P1  3 3 2 2
P2  1 2 3 4
P3  1 3 5 0
```

```
Available system resources:

     A B C D
Free 3 0 1 2
```

```
Need = (maximum - allocated)

    A B C D
P1  2 1 0 1
P2  0 1 0 1
P3  0 1 4 0
```

Process 2 requests 1 unit of resource B

Banker's Algorithm

```
Processes (allocated):

    A B C D
P1  1 2 2 1
P2  1 1 3 3
P3  1 2 1 0
```

```
Processes (maximum):

    A B C D
P1  3 3 2 2
P2  1 2 3 4
P3  1 3 5 0
```

```
Available system resources:

      A B C D
Free  3 0 1 2
```

```
Need = (maximum - allocated)

    A B C D
P1  2 1 0 1
P2  0 1 0 1
P3  0 1 4 0
```

Each process could need an additional B resource in order to continue!

Deadlock Avoidance & Prevention

- 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 the max number of each resource / lock you want
  - Block until all are available

Explain how this solves D.P. to your neighbor

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 the set of resources is hard to discover?

Deadlock Prevention

- Can also use **Global Ordering**
- 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
- **T,P,S:** How does this fix 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 a global 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

- 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

- **T,P,S:** How would this fix Dining Philosophers?

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
  - Never blocks

Deadlock Prevention

- What to 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

```
while true:
  THINK
  l_fork.grab()
  FAIL = r_fork.try_grab()
  if FAIL:
    l_fork.drop()       # RELEASE
    blockMe(1)          # TRY AGAIN LATER
  else:
    EAT # HAVE BOTH FORKS
    r_fork.drop()
    l_fork.drop()
```

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()
```

ICA: Deadlock Prevention

Come up with a new technique for preventing deadlock in dining philosophers (or in general)

Talk with your neighbor!