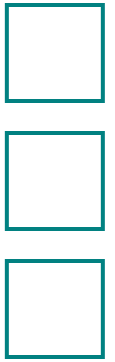
ECS150 Discussion Section

Sophie Engle

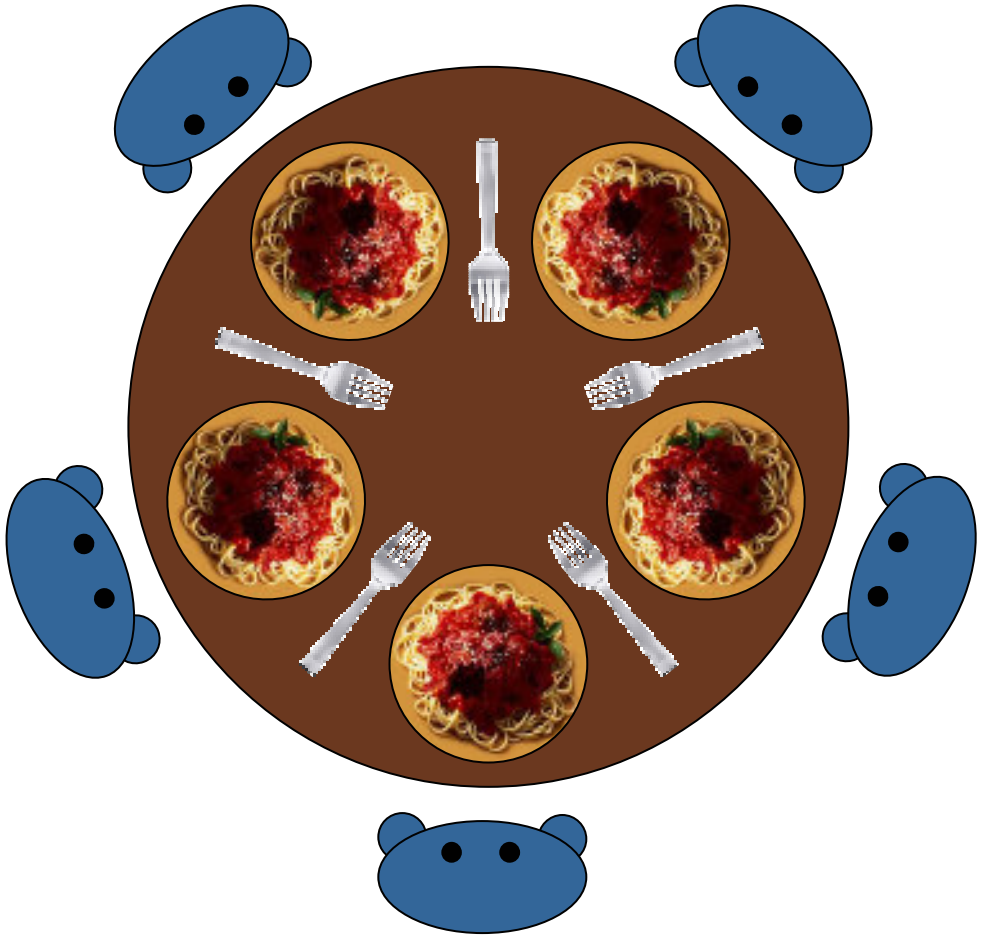
(*February 18/20 2004*)

Agenda

- Dining Philosophers Problem
  - Solve using mutual exclusion
  - Solve using semaphores
  - Solve using monitors
- Resources:
  - Section 2.2 in book (Tanenbaum)

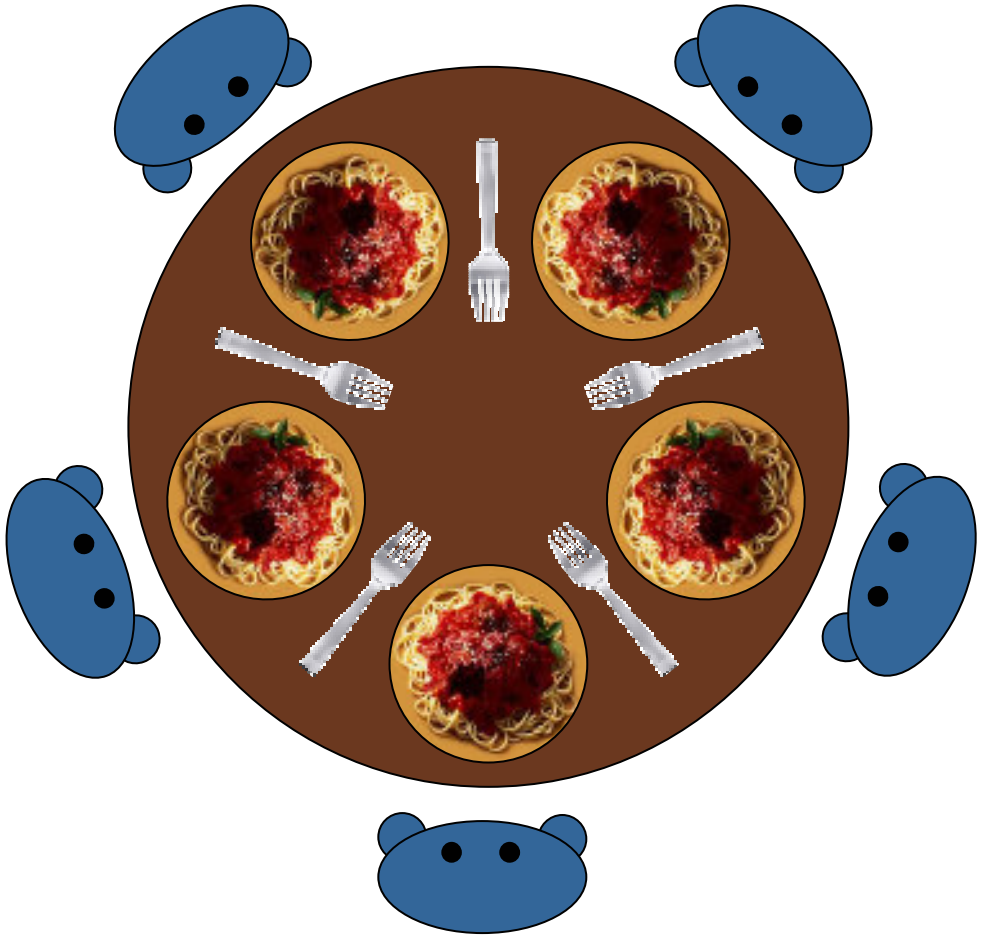


Dining Philosophers



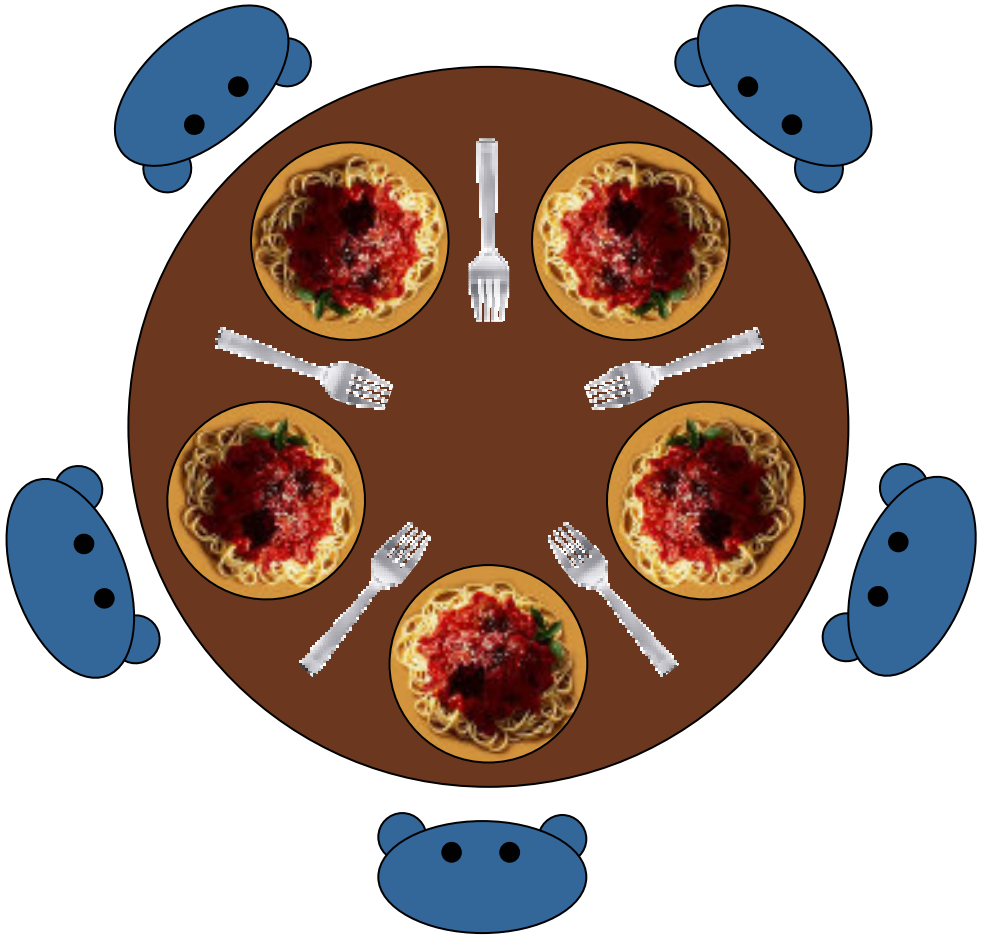
- Situation:
  - Five philosophers at table
  - Want to eat spaghetti
  - Must have 2 forks to eat
  - Only 5 forks at the table

Dining Philosophers



- Each philosopher:
  - Must grab fork one at a time
  - Can act at any time (concurrently with other philosophers)
  - Is either thinking, hungry, or eating

Spaghetti Situation 1

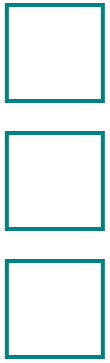


- Everyone follows:
  - Grab left fork
  - Grab right fork
  - Eat spaghetti
- If fork not available, wait until available
- Release fork when done eating
- What if all grab left fork at same time?

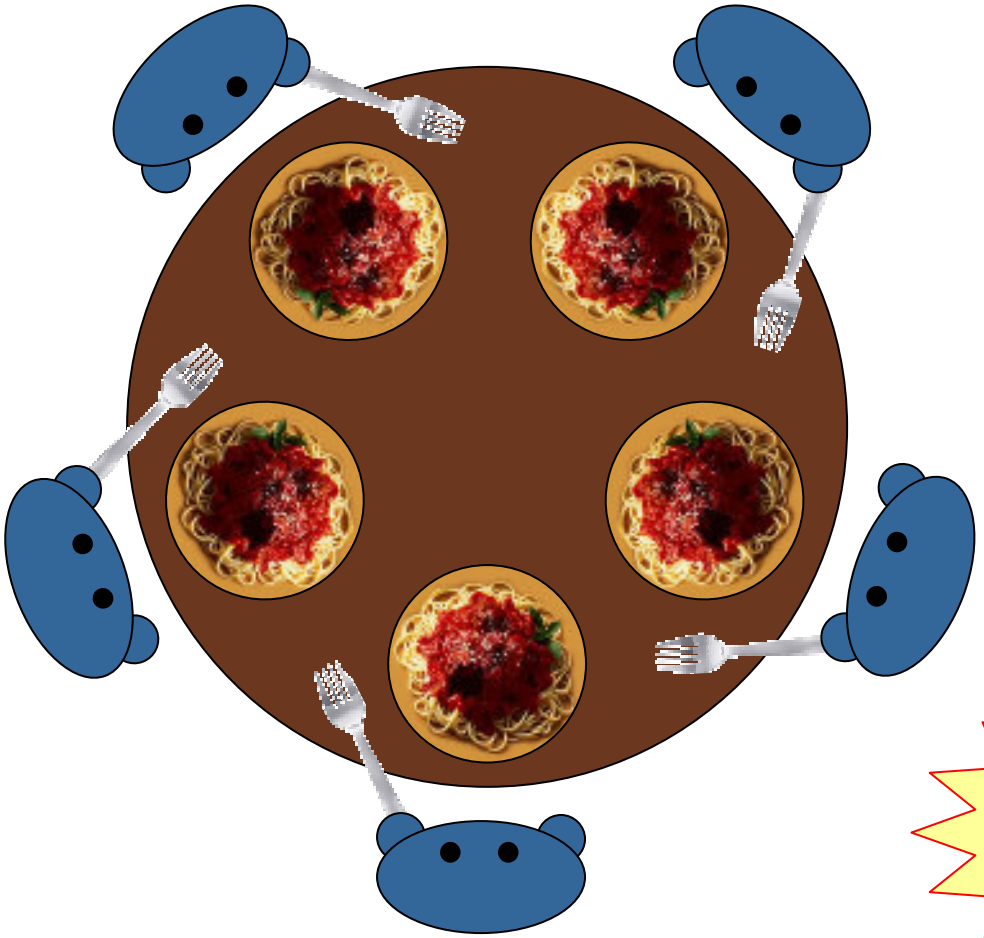
Spaghetti Situation 1

```
#define N 5

void philosopher( int i ) {
      while( TRUE ) {
            think();
            take_fork( i );                  // left fork
            take_fork( (i+1) % N );          // right fork
            eat();
            put_fork(i);
            put_fork( (i+1) % N );
      }
}
```

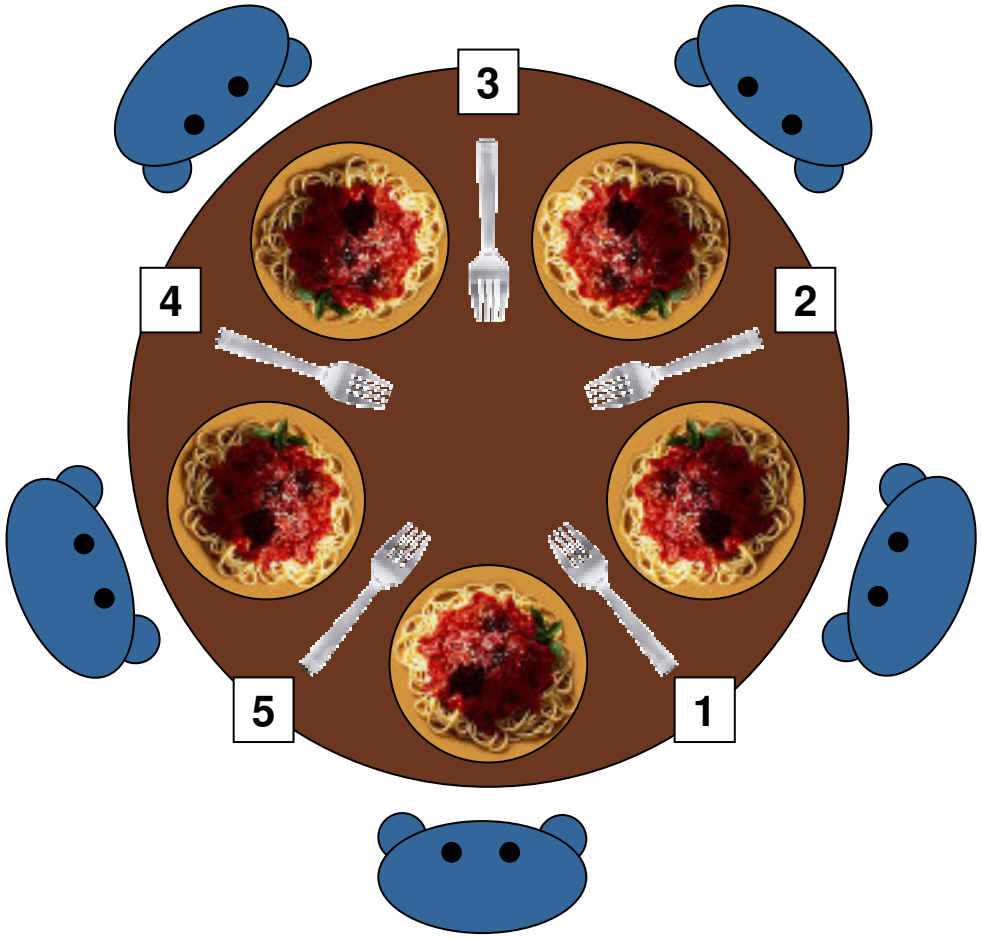


Spaghetti Situation 1



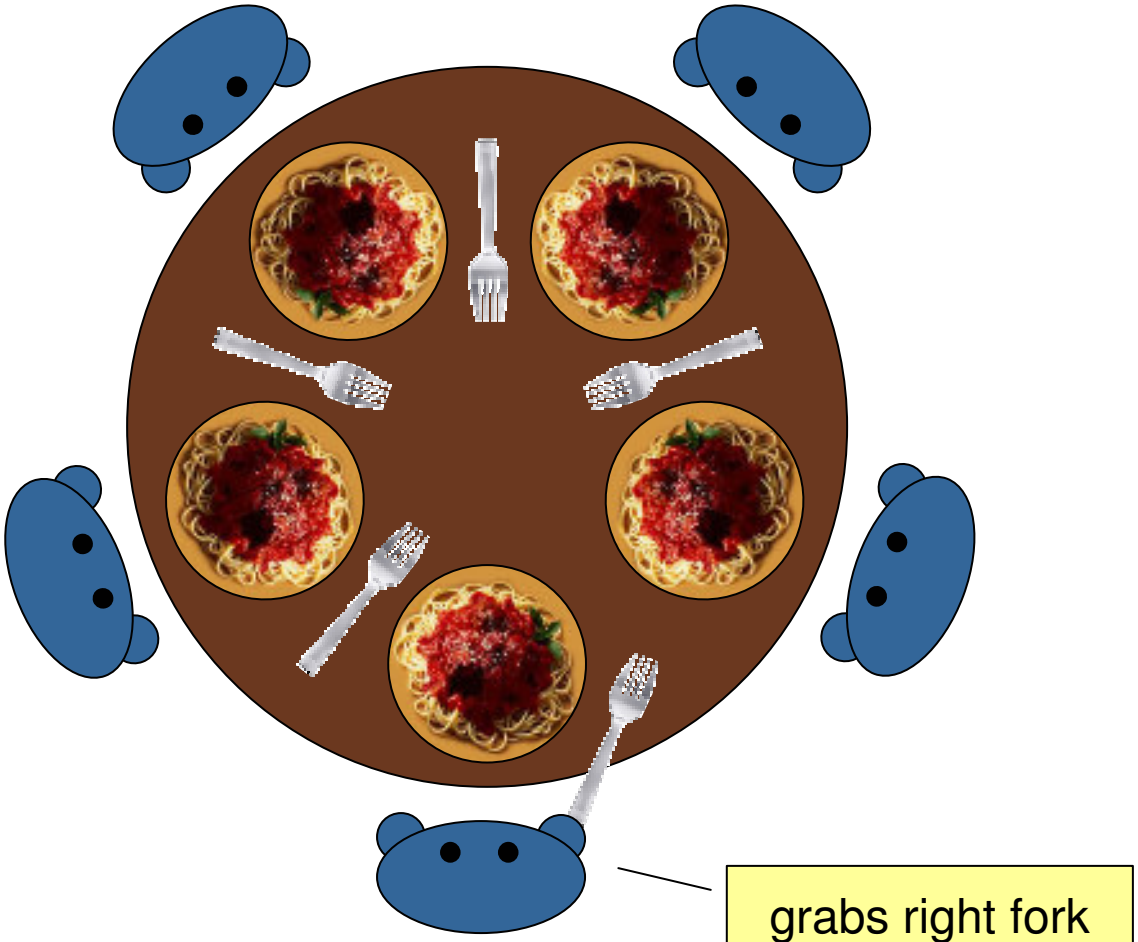
DEADLOCK!

Spaghetti Situation 2

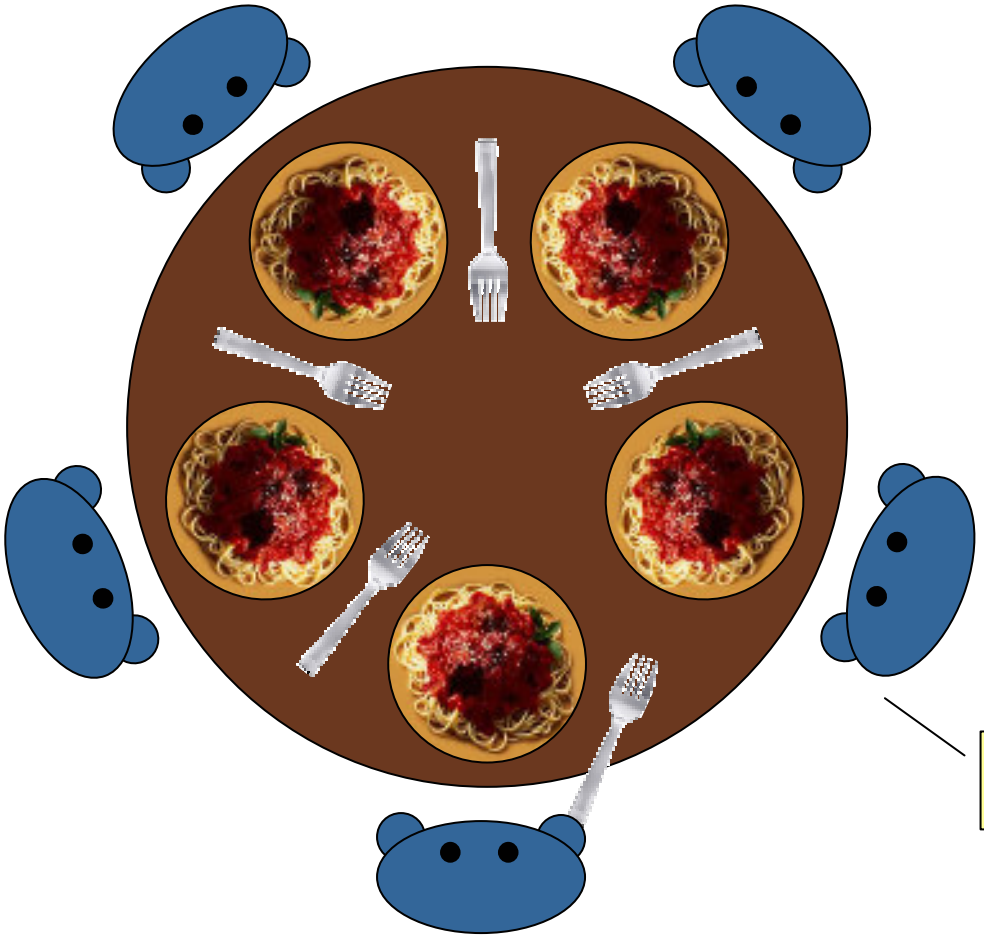


- One person follows:
  - Grab right fork
  - Grab left fork
  - Eat spaghetti
- Everyone else follows:
  - Grab left fork
  - Grab right fork
  - Eat spaghetti
- Avoids deadlock!
  - Uses hierarchical allocation
  - Must grab lower numbered fork first!

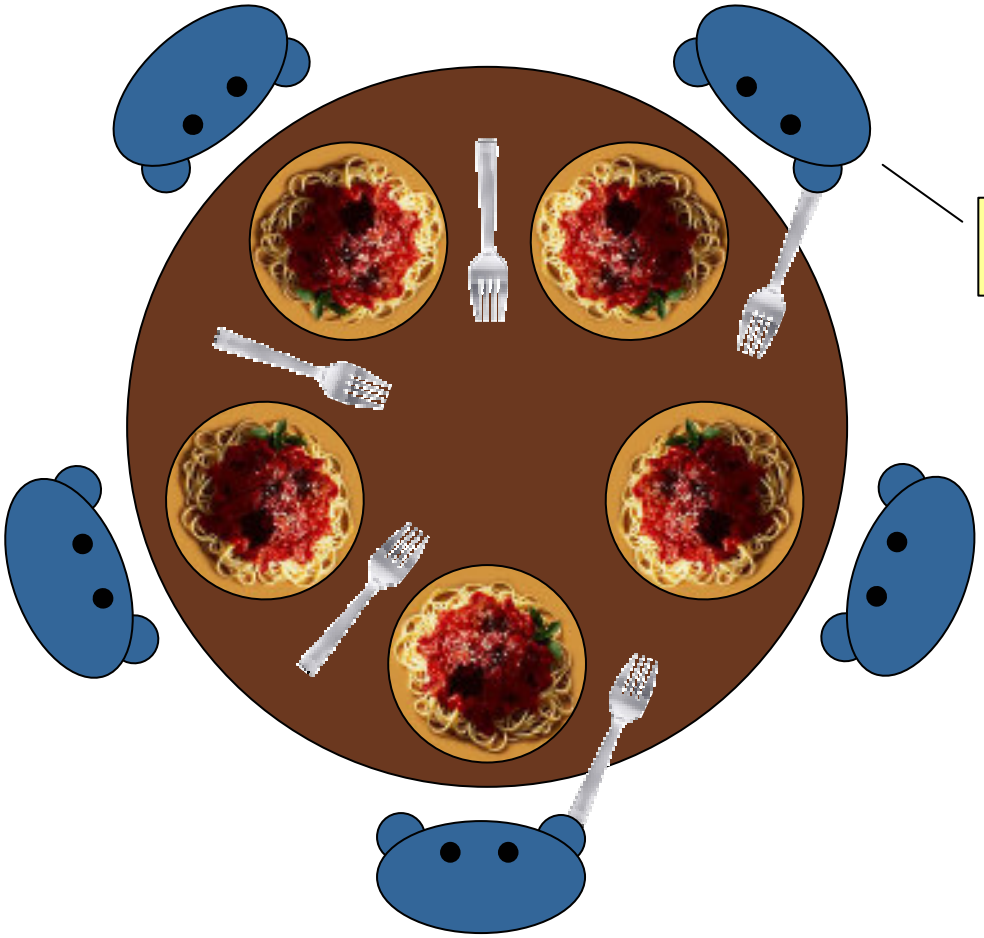
Spaghetti Situation 2



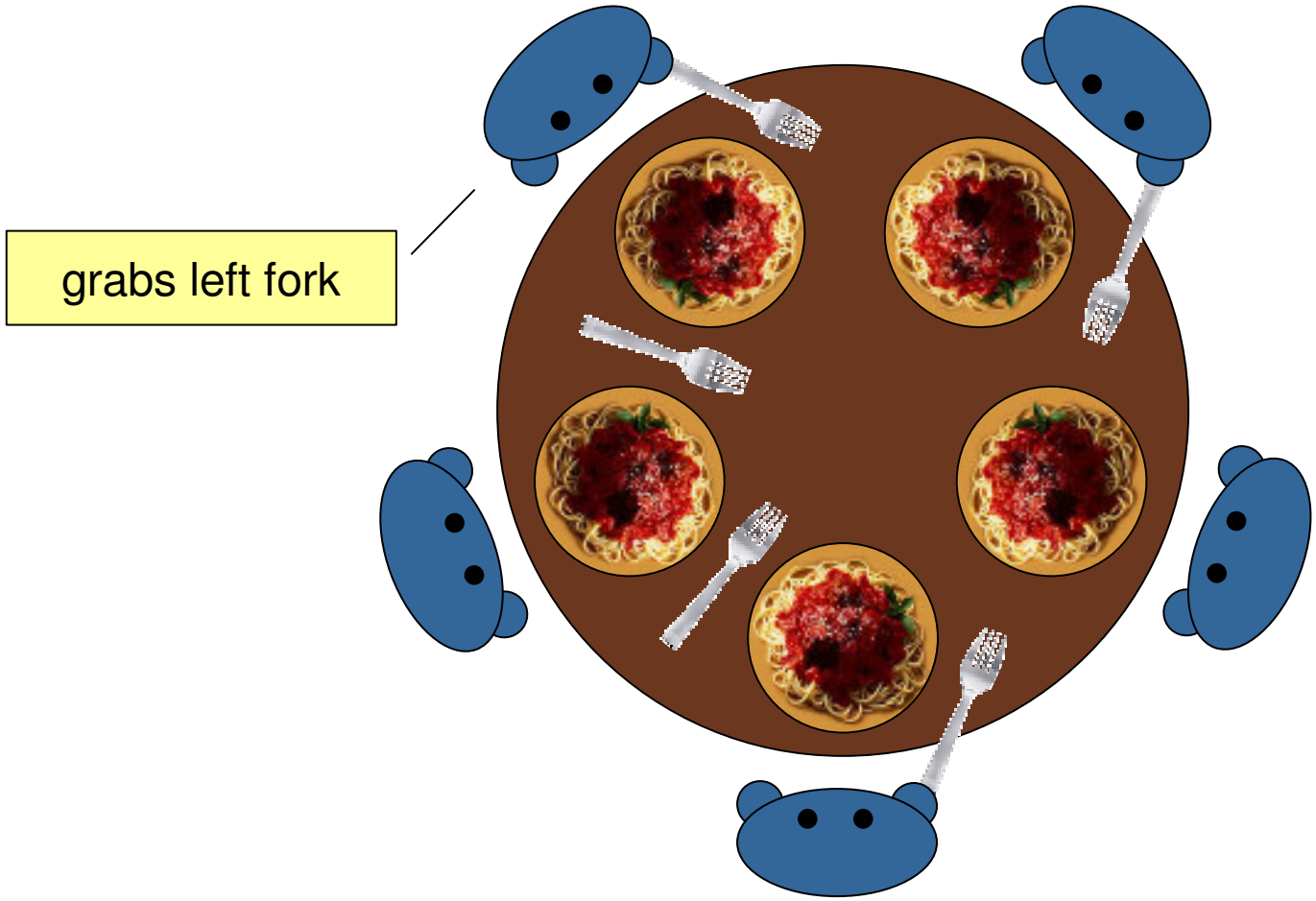
Spaghetti Situation 2



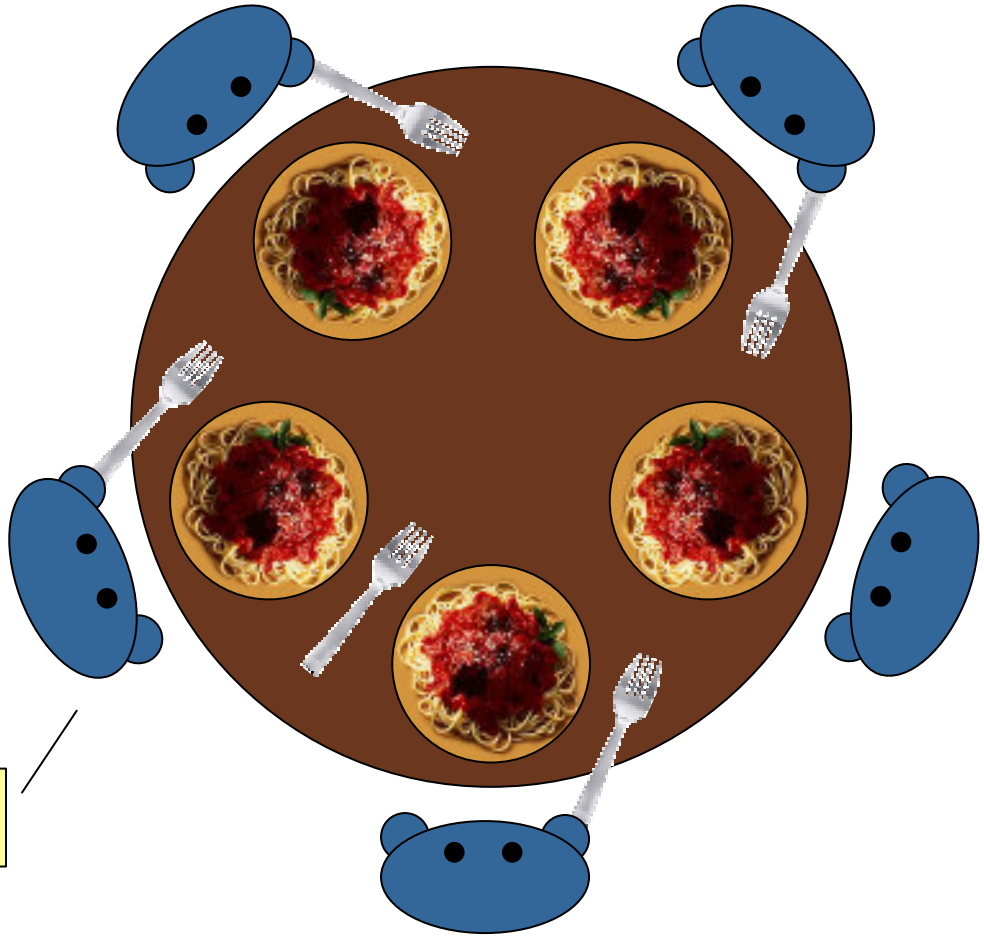
Spaghetti Situation 2



Spaghetti Situation 2



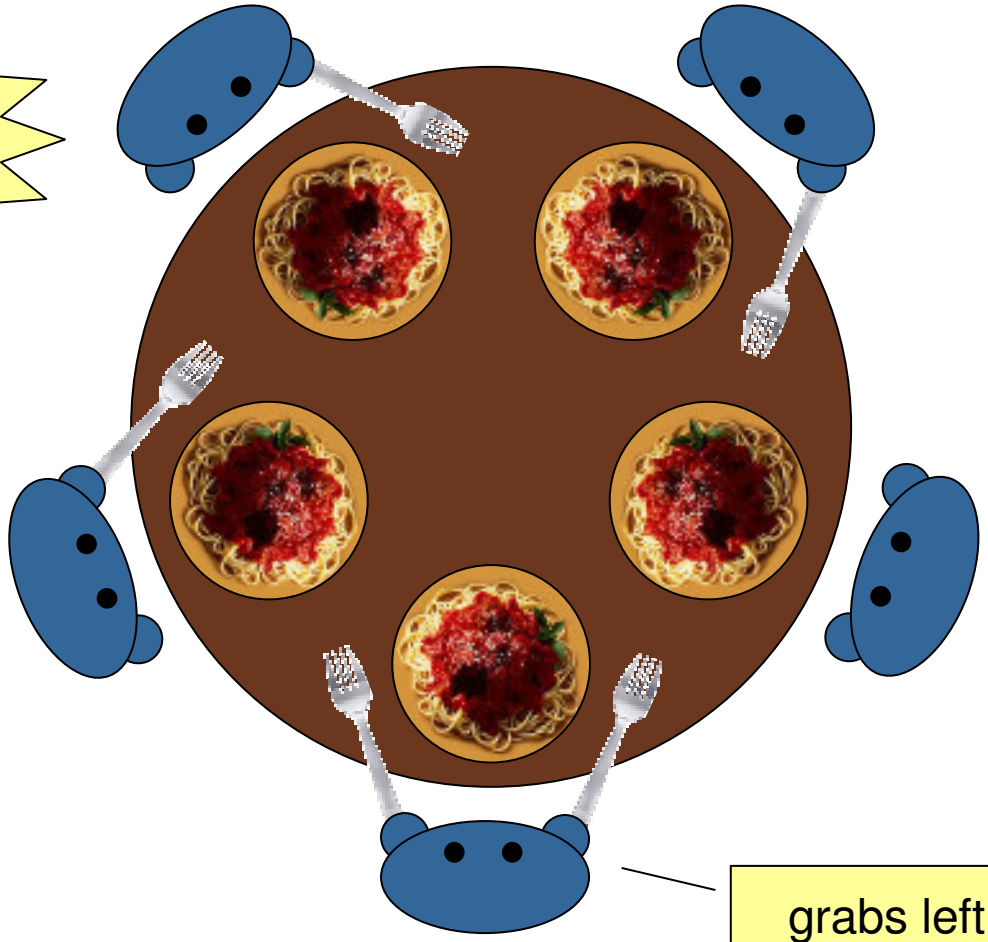
Spaghetti Situation 2



grabs left fork

Spaghetti Situation 2

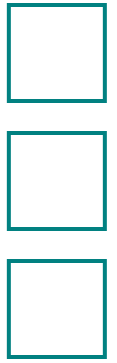
DEADLOCK AVOIDED



grabs left fork

Solving Spaghetti Problem

- Must avoid:
  - Deadlock
  - Starvation
- Some tools:
  - Mutual exclusion
  - Semaphores
  - Monitors

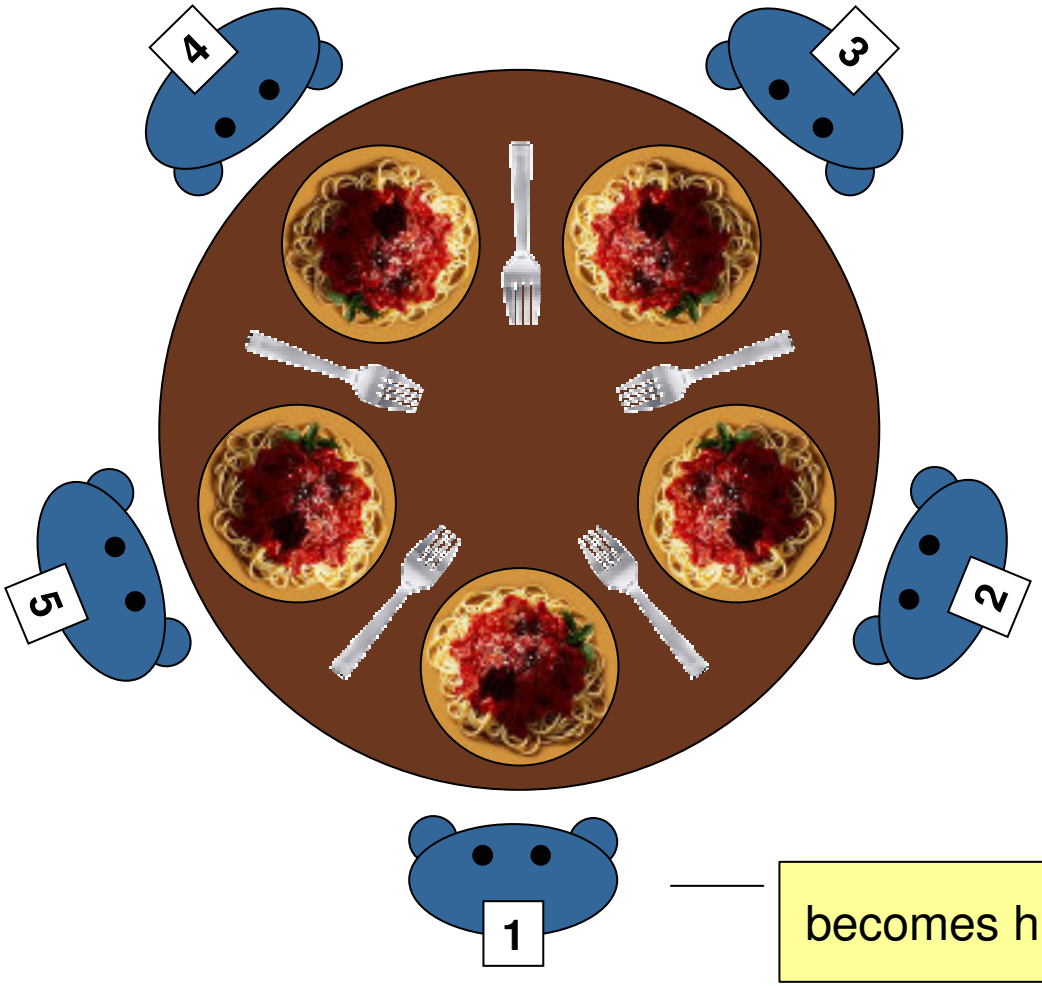


Semaphore: Example 1

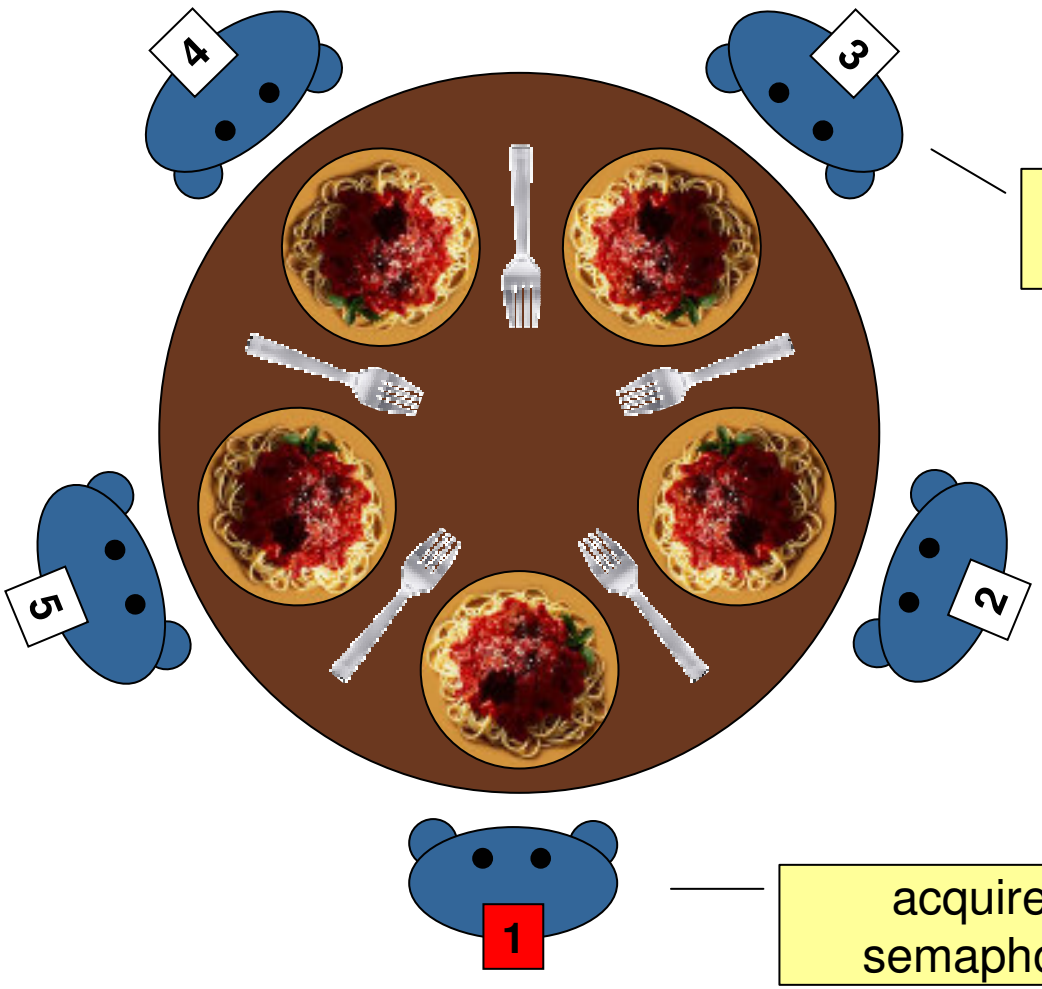
```
void philosopher( int i ) {
        semaphore mutex = 1;

        while( TRUE ) {
                think();
                down( &mutex );     // enter critical region
                take_fork( i );
                take_fork( (i+1) % N );
                eat();
                put_fork(i);
                put_fork( (i+1) % N );
                up( &mutex );       // exit critical region
        }
}
```

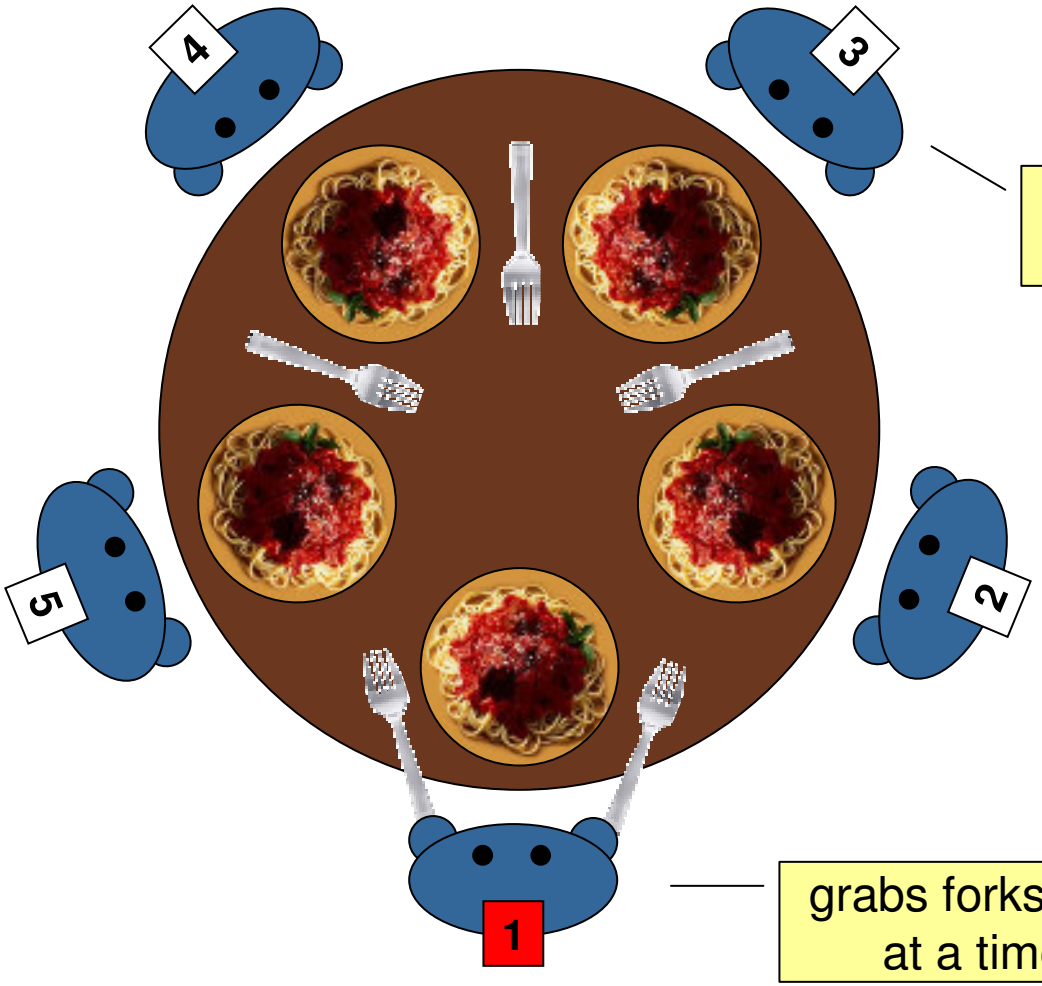
Semaphore: Example 1



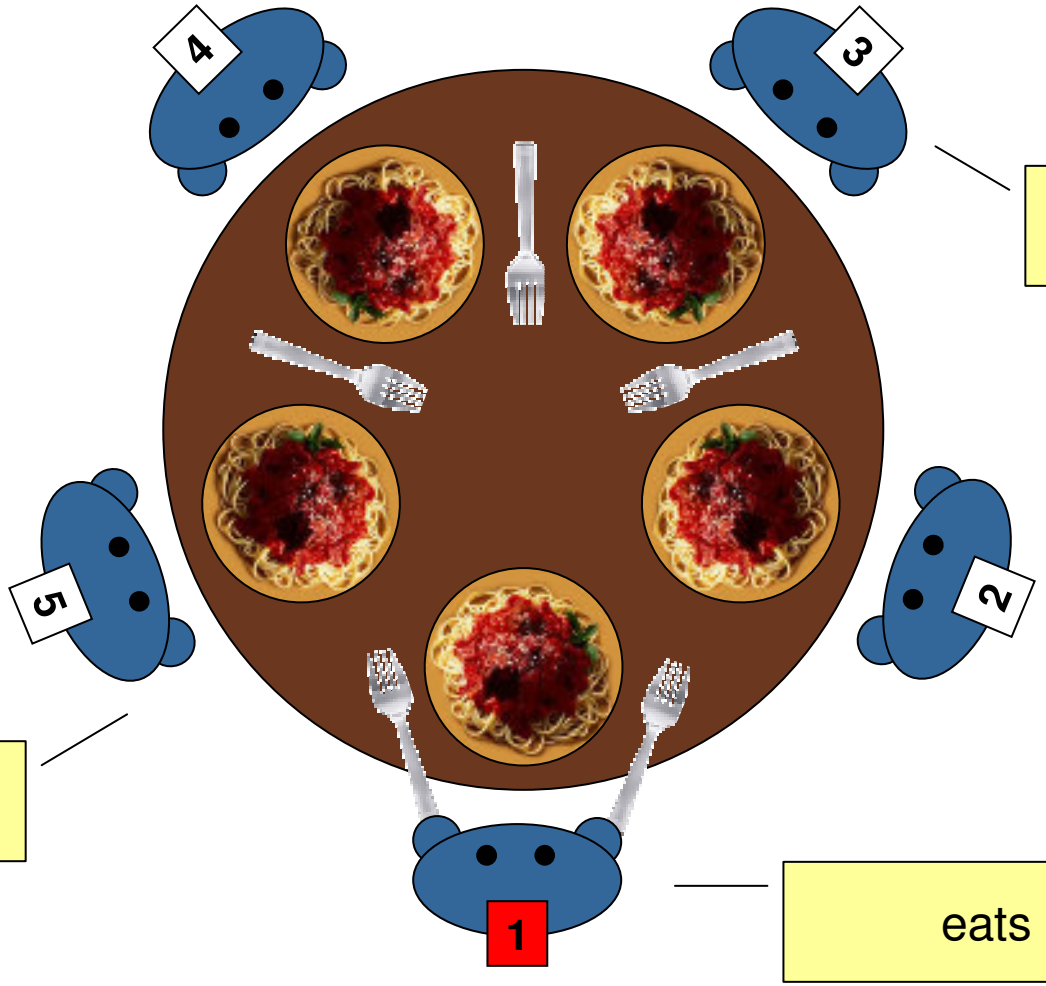
Semaphore: Example 1



Semaphore: Example 1



Semaphore: Example 1

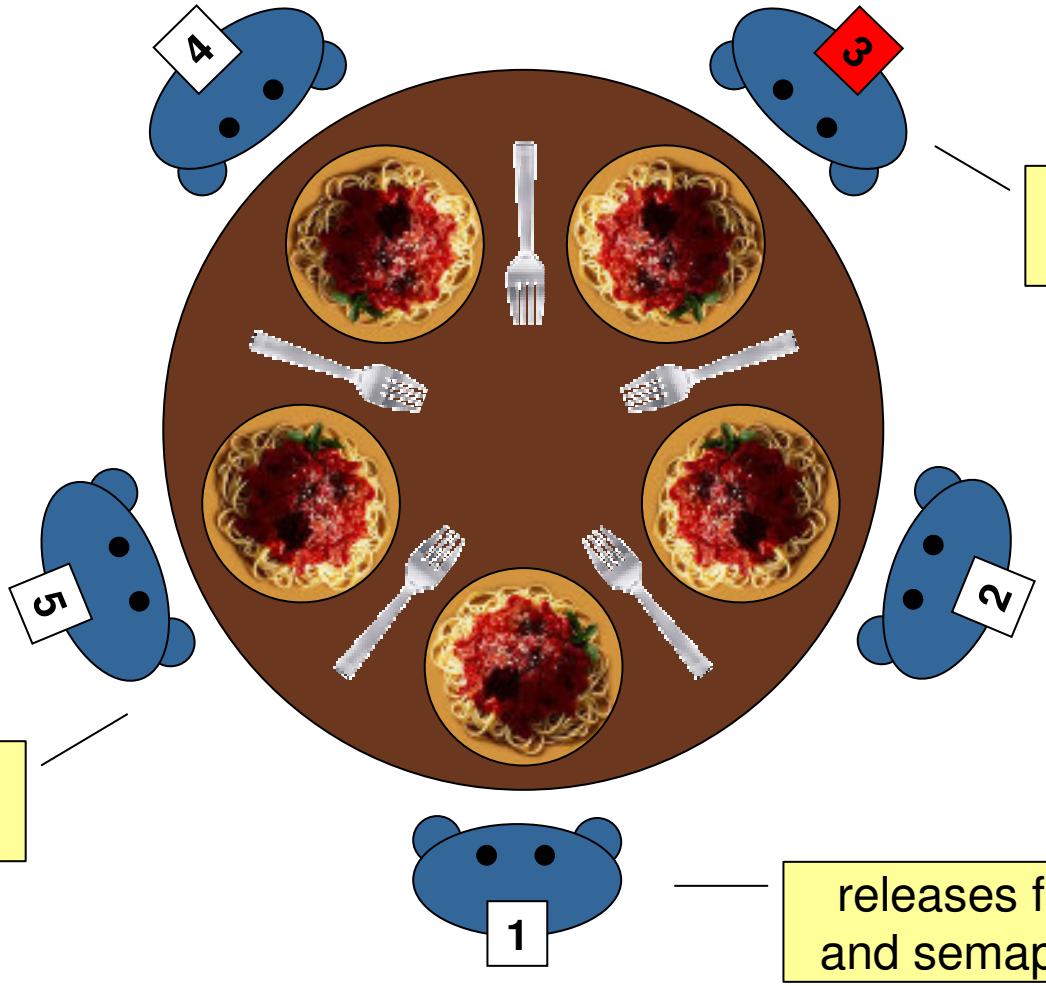


waits for semaphore

becomes hungry

eats

Semaphore: Example 1



acquires semaphore

waits for semaphore

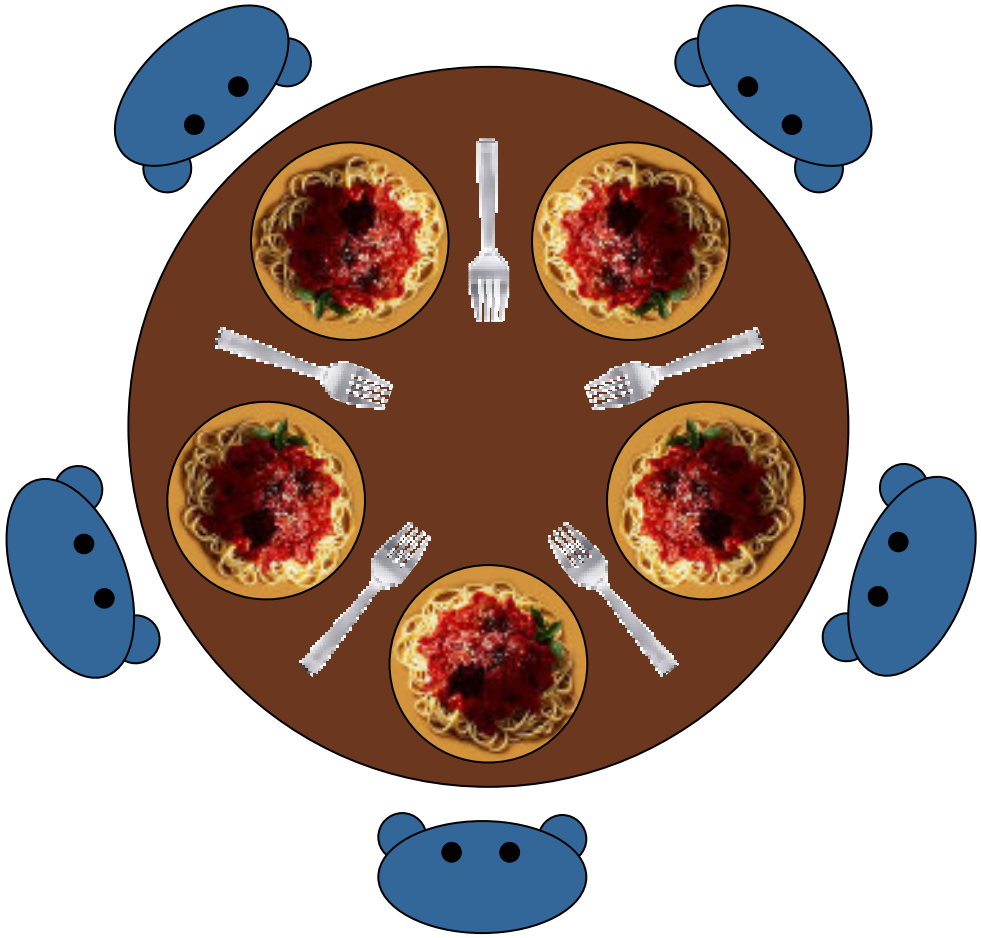
releases forks and semaphore

Semaphore: Example 1

- Prevents deadlock
- Only allows one philosopher to eat at a time

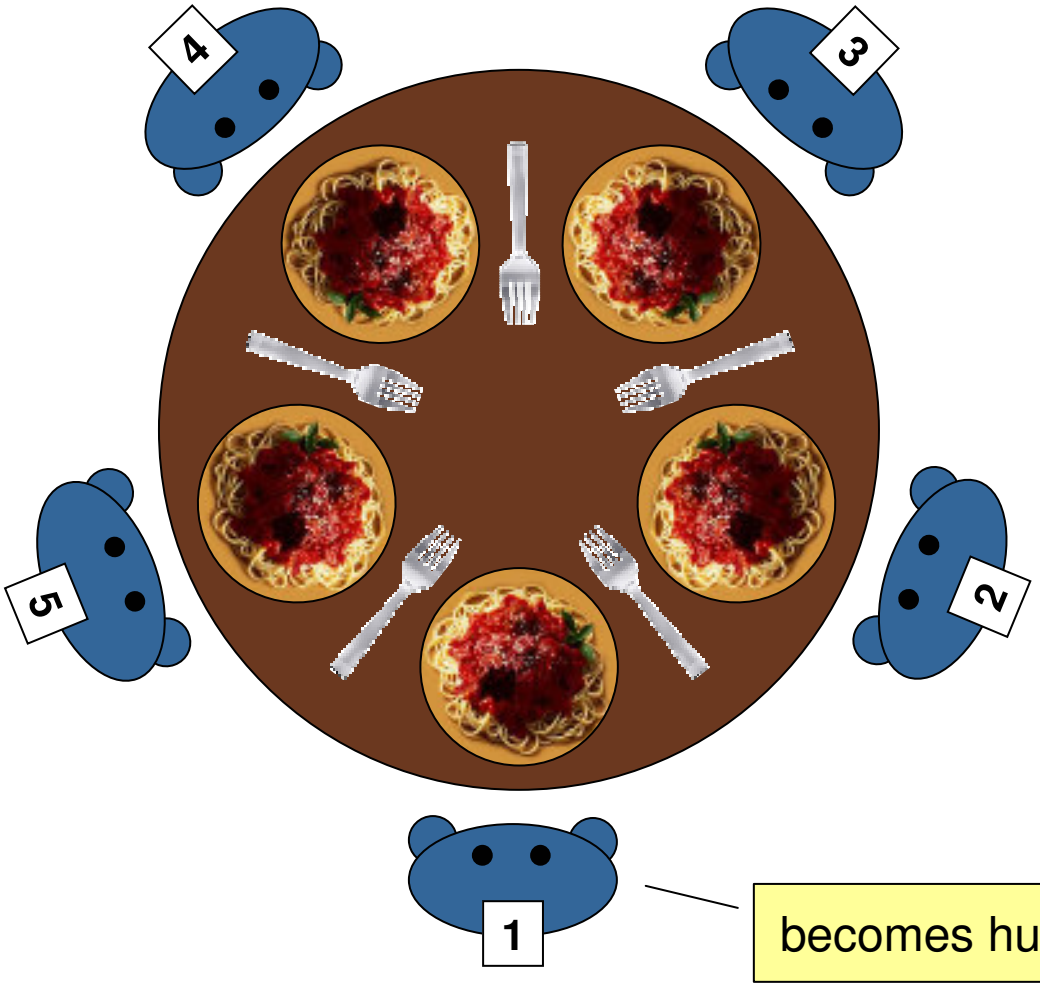
- Can improve efficiency?

Semaphore: Example 2



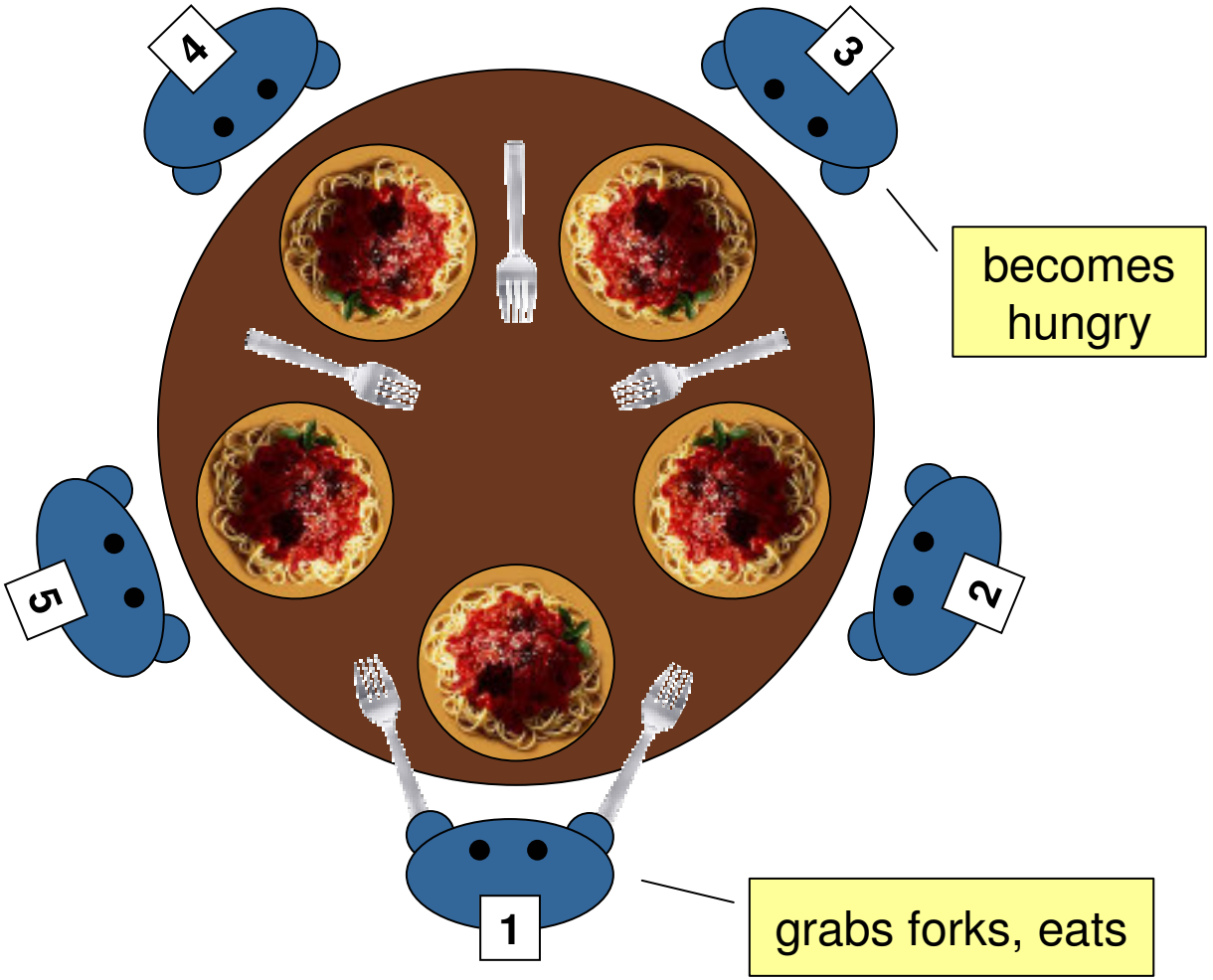
- Each person is either:
  - Thinking
  - Hungry -or-
  - Eating
- Can eat only if neither neighbor is eating
  - Uses semaphores
  - See book p78 for code

Semaphore: Example 2



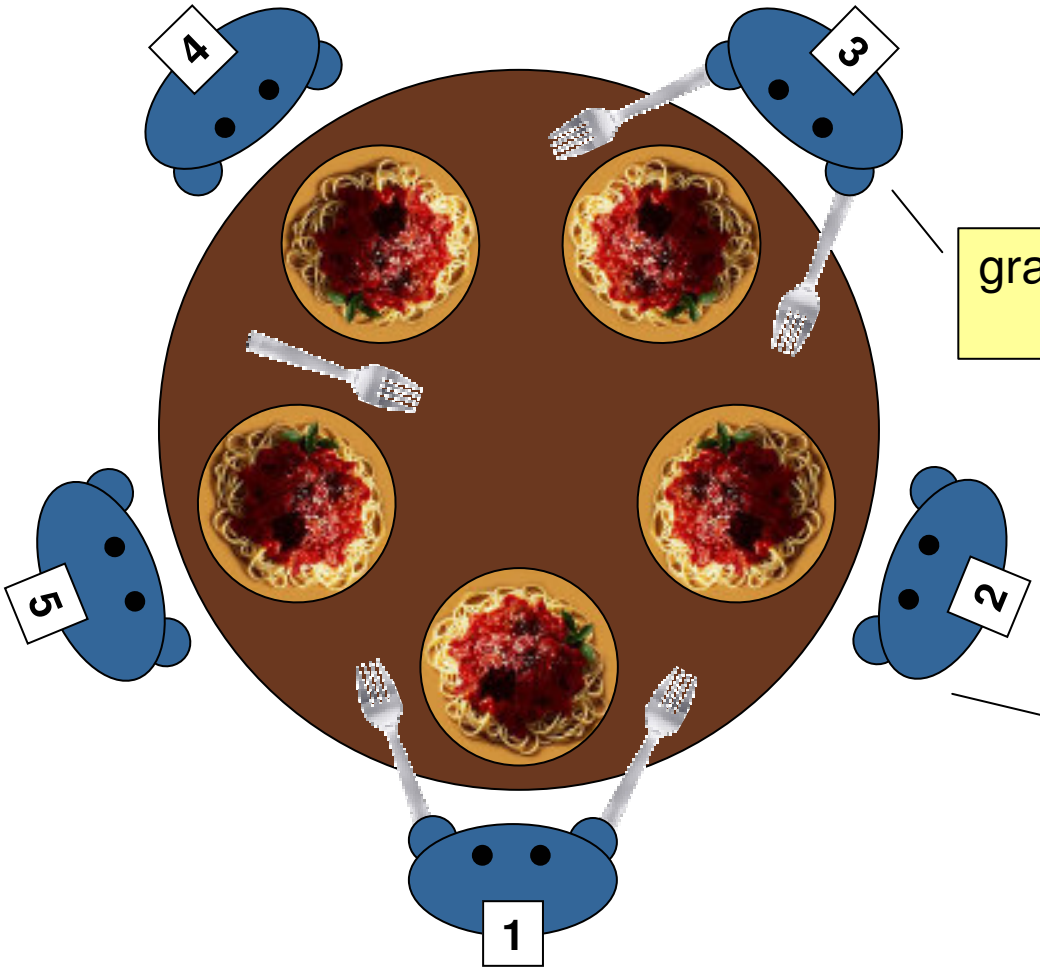
#	state
1	hungry
2	thinking
3	thinking
4	thinking
5	thinking

Semaphore: Example 2



#	state
1	eating
2	thinking
3	hungry
4	thinking
5	thinking

Semaphore: Example 2

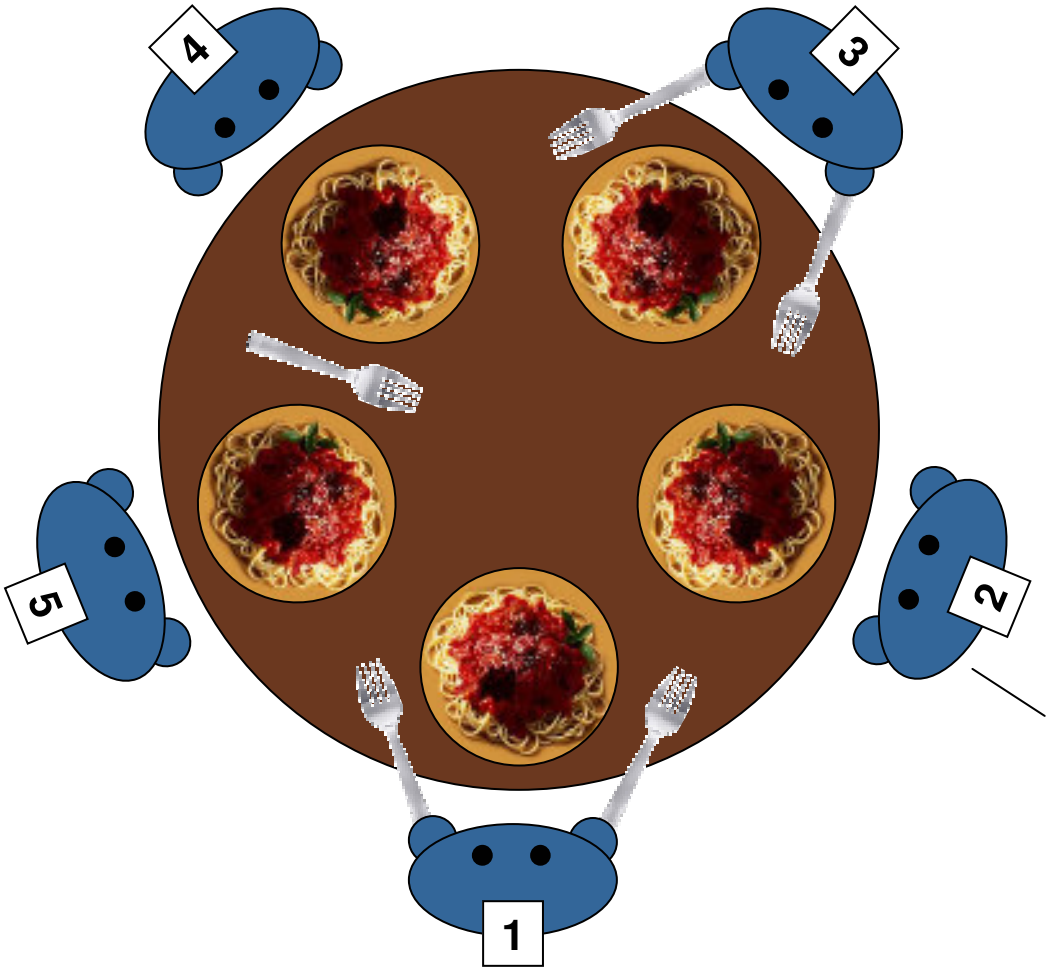


grabs forks, eats

becomes hungry

#	state
1	eating
2	hungry
3	eating
4	thinking
5	thinking

Semaphore: Example 2



#	state
1	eating
2	hungry
3	eating
4	thinking
5	thinking

waits until neighbors done

Monitor: Example 1

```
cobegin( i:= 0 to N - 1 )
    do true →
        # think
            ...
        # get forks
        dp.getforks( i );

        #eat
            ...
        # release forks
        dp.relforks( i );
    od
coend
```



Monitor: Example 1

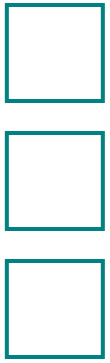
```
monitor dp
    # N - 1 forks, initialized to 2
    var forks[ 0 : N - 1 ] : int := ( [N] 2 )

    # condition variable (indicates both forks free)
    var both_free[ 0 : N - 1 ] : condition

    # right[i] is philosopher i's right neighbor
    var right[ 0 : N - 1 ] : int := ( N - 1, 0, 1, ..., N - 2 )

    # left[i] is philosopher i's left neighbor
    var left[ 0 : N - 1 ]  : int := ( 1, ..., N - 1, 0 )

    ...
end
```



Monitor: Example 1

```
monitor dp
    ...

    # grabs both forks, or waits until both free
    proc getforks(  i : int )
        if forks[i] < 2 → wait( both_free[i] ) fi
        forks[ right[i] ]--
        forks[ left[i]  ]--
    end

    ...
end
```

Monitor: Example 1

```
monitor dp
    ...

    # release both forks, signals if both free
    proc relforks(  i : int )
        forks[ right[i] ]++
        forks[ left[i]  ]++

        if forks[ right[i] ] = 2
            → signal( both_free[ right[i] ] ) fi

        if forks[ left[i] ] = 2
            → signal( both_free[ left[i] ] ) fi
    end
end
```



Monitor: Example 1

- Starvation still possible!
- Rest on chalkboard…