# Classical Synchronization Problems

# Reader-Writer Problem

- Readers read data
- Writers write data
- Rules
  - Multiple readers may read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot access data simultaneously
- Locking table
  - Whether any two can be in the critical section simultaneously

| | Reader | Writer |
|---|---|---|
| Reader | OK | No |
| Writer | No | No |

# Sleeping Barber

- Customers
  - N chairs for waiting
- Barber
  - Can cut one customer's hair at any time
  - No waiting customer => barber sleeps
- Customer enters
  - If all waiting chairs full, customer leaves
  - If barber asleep, wake up barber and get hair cut
  - Otherwise (barber is busy), wait in a chair

# Sleeping Barber

```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting
```

```
barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

```
customer {
    mutexLock(lock);
    if (waiting < chairs) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    else {
        mutexUnlock(lock);
}
```

What is the shared data?
What part protects the shared data?

# Sleeping Barber

```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting
```

```
barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

```
customer {
    mutexLock(lock);
    if (waiting < chairs) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    else {
        mutexUnlock(lock);
}
```

What guarantees that not too many customer are waiting?

# Sleeping Barber

```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting
```

```
barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

What guarantees that there is only one customer in the chair?

```
customer {
    mutexLock(lock);
    if (waiting < chairs) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    else {
        mutexUnlock(lock);
}
```

# Sleeping Barber

```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting
```

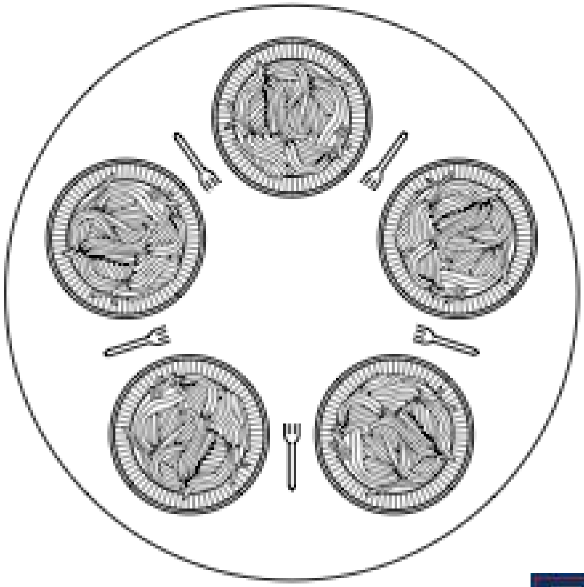
```
barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

What guarantees that the barber doesn't miss a customer?

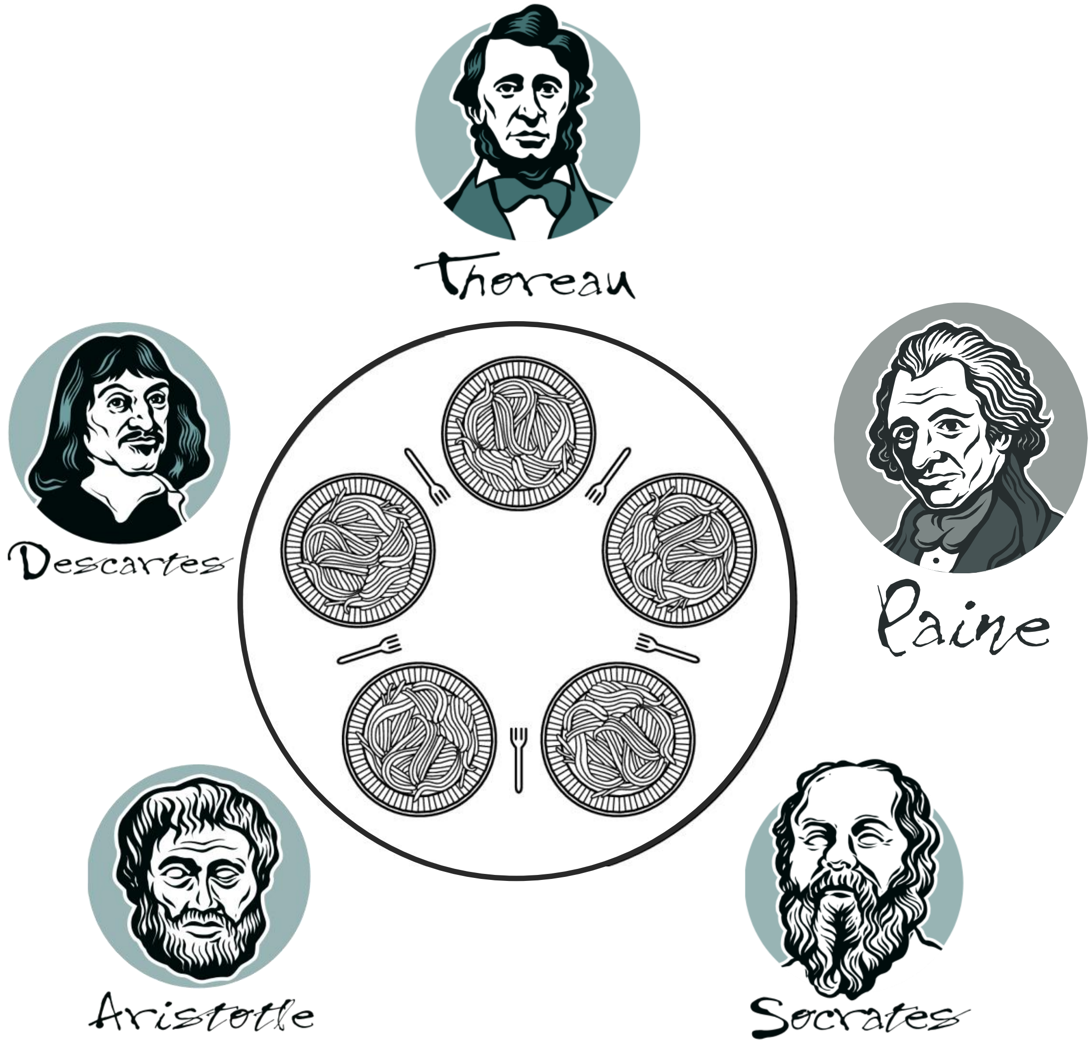
```
customer {
    mutexLock(lock);
    if (waiting < chairs) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    else {
        mutexUnlock(lock);
}
```

# Dining Philosophers

- N philosophers and N forks
- Philosophers eat/think
- Eating needs 2 forks
- Pick up one fork at a time



# Dining Philosophers



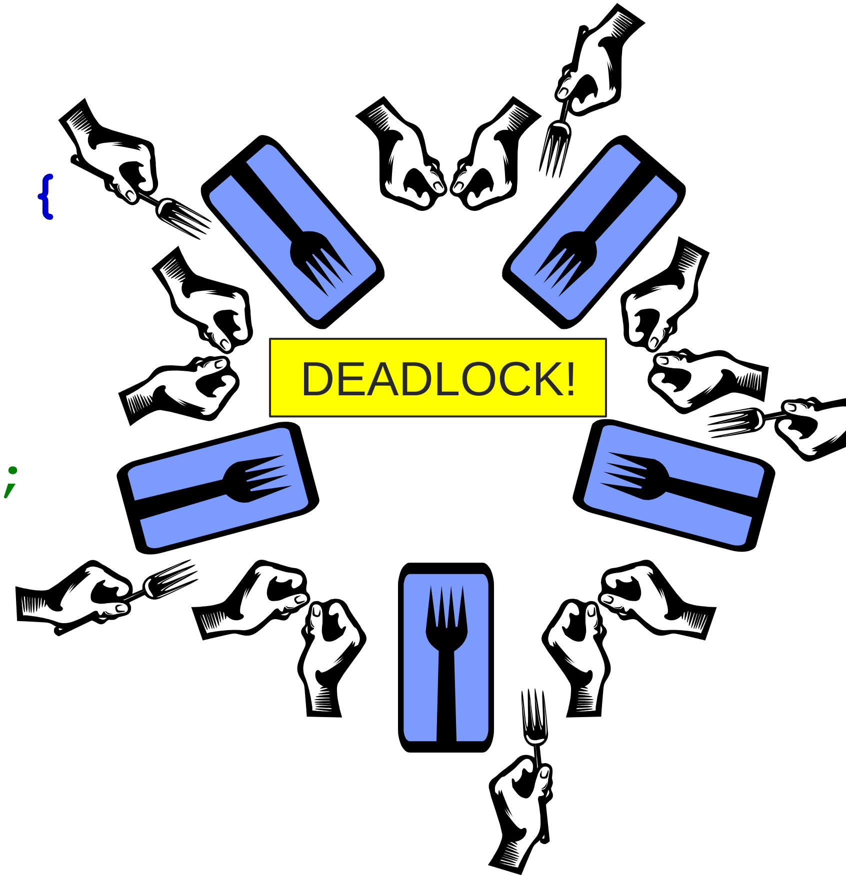
# Dining Philosophers: Take 1

```
# define N 5

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

Does this work?

# Dining Philosophers: Take 1

```
# define N 5

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



# What is deadlock?

- Necessary and sufficient conditions for deadlock
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

- Which properties does our solution to dining philosophers have?

# Conditions for Deadlock

- Mutual exclusion
  - Exclusive use of chopsticks
- Hold and wait
  - Hold 1 chopstick, wait for next
- No preemption
  - Cannot force another to release held resource
- Circular wait
  - Each waits for next neighbor to put down chopstick

# Dining Philosophers: Take 1

```
# define N 5

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

take_forks(i);

put_forks(i);

How can we fix this?

# Dining Philosophers: Take 2

```
#define N           5
#define THINKING    0
#define HUNGRY      1
#define EATING      2
#define LEFT        (i – 1)%N
#define RIGHT       (i + 1)%N
```

```
int state[N];
mutex lock;
semaphore sem[N];

void philosopher (int i) {
   while (TRUE) {
      think();
      take_forks(i);
      eat(); /* yummy */
      put_forks(i);
   }
}
```

# Dining Philosophers: Take 2

```
void take_forks(int i) {
    mutexLock(lock);
    state[i] = HUNGRY;
    test(i);
    mutexUnlock(lock);
    semWait(sem[i]);
}

void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

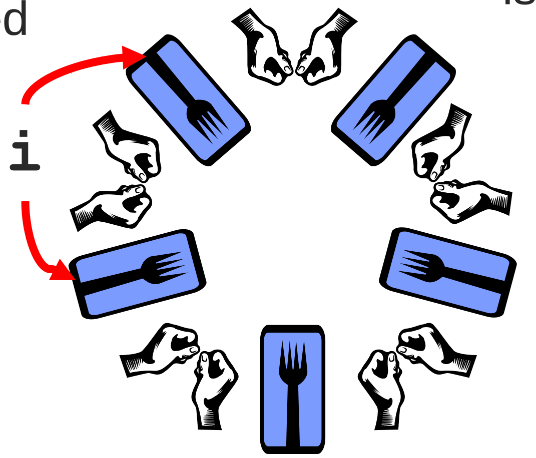
```
/* only called with lock set!
   */

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i] = EATING;
    semSignal(sem[i]);
  }
}
```

# Dining Philosophers: Take 2

Try to get 2 forks

Block if forks not acquired

```
void take_forks(int i) {
    mutexLock(lock);
    state[i] = HUNGRY;
    test(i);
    mutexUnlock(lock);
    semWait(sem[i]);
}
```

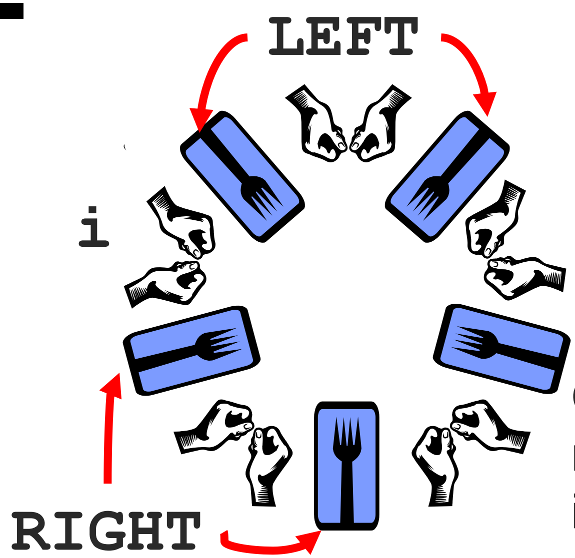



Get both forks iff neither neighbor is hungry

```
/* only called with lock set! */

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i] = EATING;
    semSignal(sem[i]);
  }
}
```

Signal myself

# Dining Philosophers: Take 2




```
void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

Let others get a turn

```
/* only called with lock set!
   */

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i] = EATING;
    semSignal(sem[i]);
  }
}
```

Get both forks iff neither neighbor is hungry

Signal waiting philosopher

# Dining Philosophers: Take 2

```
void take_forks(int i) {
    mutexLock(lock);
    state[i] = HUNGRY;
    test(i);
    mutexUnlock(lock);
    semWait(sem[i]);
}

void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

```
/* only called with lock set!
   */

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
     state[i] = EATING;
     semSignal(sem[i]);
  }
}
```

How do we guarantee that only one philosopher is using a given fork?

# Dining Philosophers: Take 2

```
void take_forks(int i) {
    mutexLock(lock);
    state[i] = HUNGRY;
    test(i);
    mutexUnlock(lock);
    semWait(sem[i]);
}

void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

```
/* only called with lock set!
   */

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
     state[i] = EATING;
     semSignal(sem[i]);
  }
}
```

How do we guarantee that there is no deadlock?

# Dining Philosophers: Take 2

```
void take_forks(int i) {
    mutexLock(lock);
    state[i] = HUNGRY;
    test(i);
    mutexUnlock(lock);
    semWait(sem[i]);
}

void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

```
/* only called with lock set!
   */

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
     state[i] = EATING;
     semSignal(sem[i]);
  }
}
```

How do we guarantee that the solution is fair?

# Dining Philosophers: Take 2

```
void take_forks(int i) {
    mutexLock(lock);
    state[i] = HUNGRY;
    test(i);
    mutexUnlock(lock);
    semWait(sem[i]);
}

void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

```
/* only called with lock set!
   */

void test(int i) {
  if (state[i] == HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING) {
     state[i] = EATING;
     semSignal(sem[i]);
  }
}
```

What do we need to change to solve this with condition variables?

# Dining Philosophers: Take 2

```
void take_forks(int i) {
    mutexLock(lock);
    state[i] = HUNGRY;
    test(i);
    mutexUnlock(lock);
    semWait(sem[i]);
}

void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

```
/* only called with lock set!
   */

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i] = EATING;
    semSignal(sem[i]);
  }
}
```

What do we need to change to solve this with condition variables?

# Dining Philosophers: with Condition Variables

```
void take_forks(int i) {
   mutexLock(lock);
   state[i] = HUNGRY;
   test(i);
   while (state[i]==HUNGRY)
     condWait(cond[i]);
   mutexUnlock(lock);
}

void put_forks(int i) {
    mutexLock(lock);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    mutexUnlock(lock);
}
```

```
int state[N];
mutex lock;
condition cond[N];

void test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
     state[i] = EATING;
     condSignal(cond[i]);
  }
}
```

# What if...

- Picking up both left and right chopsticks is an atomic operation?
  - That works (i.e., prevents deadlock)
  - This is essentially what we just did!
- Or, we have N philosophers & N+1 chopsticks?
  - That works too!
- And we'll see another solution later...