

Operating Systems

Common Concurrency Problems

Nipun Batra

Producer Consumer Problem using Semaphores

Adding mutual exclusion

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;

5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&mutex); // line p0 (NEW LINE)
9         sem_wait(&empty); // line p1
10        put(i); // line p2
11        sem_post(&full); // line p3
12        sem_post(&mutex); // line p4 (NEW LINE)
13    }
14 }
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex); // line c0 (NEW LINE)
20         sem_wait(&full); // line c1
21         int tmp = get(); // line c2
22         sem_post(&empty); // line c3
23         sem_post(&mutex); // line c4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
```

Producer Consumer Problem using Semaphores

Adding mutual exclusion

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;

5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&mutex); // line p0 (NEW LINE)
9         sem_wait(&empty); // line p1
10        put(i); // line p2
11        sem_post(&full); // line p3
12        sem_post(&mutex); // line p4 (NEW LINE)
13    }
14 }
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex); // line c0 (NEW LINE)
20         sem_wait(&full); // line c1
21         int tmp = get(); // line c2
22         sem_post(&empty); // line c3
23         sem_post(&mutex); // line c4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
```

- Unfortunately, this program also has a problem — find it out

Producer Consumer Problem using Semaphores

Adding mutual exclusion

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;

5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&mutex); // line p0 (NEW LINE)
9         sem_wait(&empty); // line p1
10        put(i); // line p2
11        sem_post(&full); // line p3
12        sem_post(&mutex); // line p4 (NEW LINE)
13    }
14 }
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex); // line c0 (NEW LINE)
20         sem_wait(&full); // line c1
21         int tmp = get(); // line c2
22         sem_post(&empty); // line c3
23         sem_post(&mutex); // line c4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
```

- Unfortunately, this program also has a problem — find it out
- Hint, the problem is called deadlock

Producer Consumer Problem using Semaphores

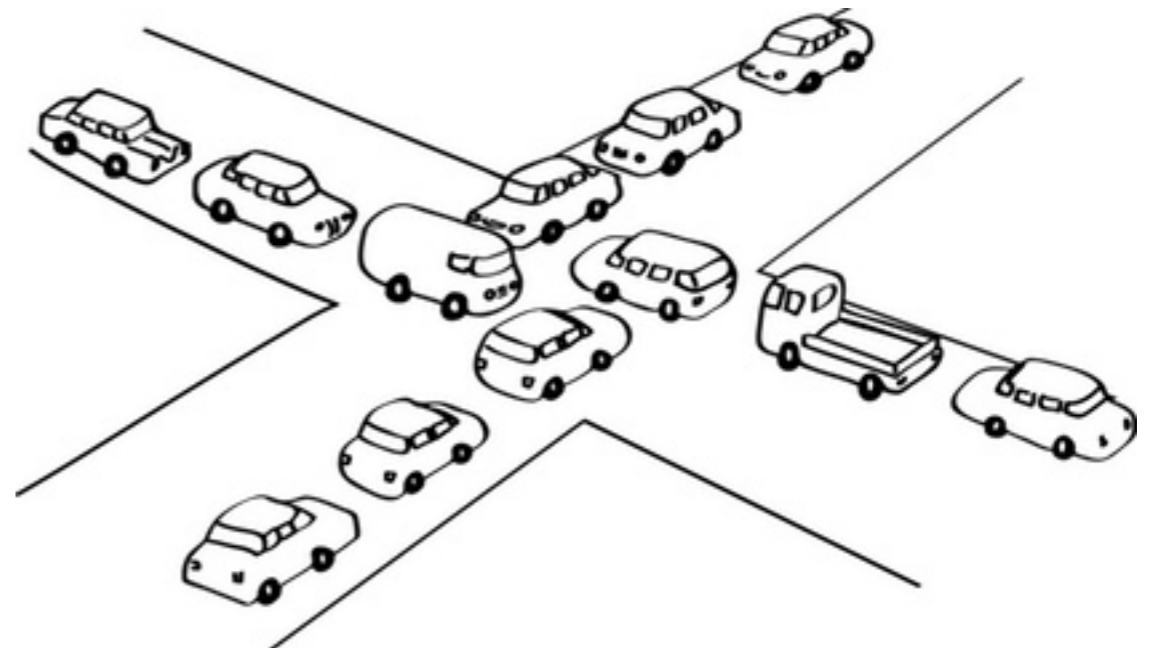
Adding mutual exclusion

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;

5 void *producer(void *arg) {
6 int i;
7 for (i = 0; i < loops; i++) {
8 sem_wait(&mutex); // line p0 (NEW LINE)
9 sem_wait(&empty); // line p1
10 put(i); // line p2
11 sem_post(&full); // line p3
12 sem_post(&mutex); // line p4 (NEW LINE)
13 }
14 }
```

```
16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

- Unfortunately, this program also has a problem — find it out
- Hint, the problem is called deadlock



Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10 put(i); // line p2
11 sem_post(&full); // line p3
12 sem_post(&mutex); // line p4 (NEW LINE)
13 }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10  put(i); // line p2
11  sem_post(&full); // line p3
12  sem_post(&mutex); // line p4 (NEW LINE)
13  }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10 put(i); // line p2
11 sem_post(&full); // line p3
12 sem_post(&mutex); // line p4 (NEW LINE)
13 }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10 put(i); // line p2
11 sem_post(&full); // line p3
12 sem_post(&mutex); // line p4 (NEW LINE)
13 }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).
- The consumer calls sem_wait() on the full semaphore (line c1).

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10 put(i); // line p2
11 sem_post(&full); // line p3
12 sem_post(&mutex); // line p4 (NEW LINE)
13 }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).
- The consumer calls sem_wait() on the full semaphore (line c1).
- The consumer is blocked and yield the CPU.

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10  put(i); // line p2
11  sem_post(&full); // line p3
12  sem_post(&mutex); // line p4 (NEW LINE)
13  }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).
- The consumer calls `sem_wait()` on the full semaphore (line c1).
- The consumer is blocked and yield the CPU.
- The consumer still holds the mutex!

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10  put(i); // line p2
11  sem_post(&full); // line p3
12  sem_post(&mutex); // line p4 (NEW LINE)
13  }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).
- The consumer calls `sem_wait()` on the full semaphore (line c1).
- The consumer is blocked and yield the CPU.
- The consumer still holds the mutex!
- The producer calls `sem_wait()` on the binary mutex semaphore (line p0).

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10  put(i); // line p2
11  sem_post(&full); // line p3
12  sem_post(&mutex); // line p4 (NEW LINE)
13  }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).
- The consumer calls `sem_wait()` on the full semaphore (line c1).
- The consumer is blocked and yield the CPU.
- The consumer still holds the mutex!
- The producer calls `sem_wait()` on the binary mutex semaphore (line p0).
- The producer is now stuck waiting too — **a classic deadlock.**

Producer Consumer Problem using Semaphores

Deadlock

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;

5  void *producer(void *arg) {
6  int i;
7  for (i = 0; i < loops; i++) {
8  sem_wait(&mutex); // line p0 (NEW LINE)
9  sem_wait(&empty); // line p1
10  put(i); // line p2
11  sem_post(&full); // line p3
12  sem_post(&mutex); // line p4 (NEW LINE)
13  }
14 }

16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex); // line c0 (NEW LINE)
20 sem_wait(&full); // line c1
21 int tmp = get(); // line c2
22 sem_post(&empty); // line c3
23 sem_post(&mutex); // line c4 (NEW LINE)
24 printf("%d\n", tmp);
25 }
26 }
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).
- The consumer calls `sem_wait()` on the full semaphore (line c1).
- The consumer is blocked and yield the CPU.
- The consumer still holds the mutex!
- The producer calls `sem_wait()` on the binary mutex semaphore (line p0).
- The producer is now stuck waiting too — **a classic deadlock.**

Producer Consumer Problem using Semaphores

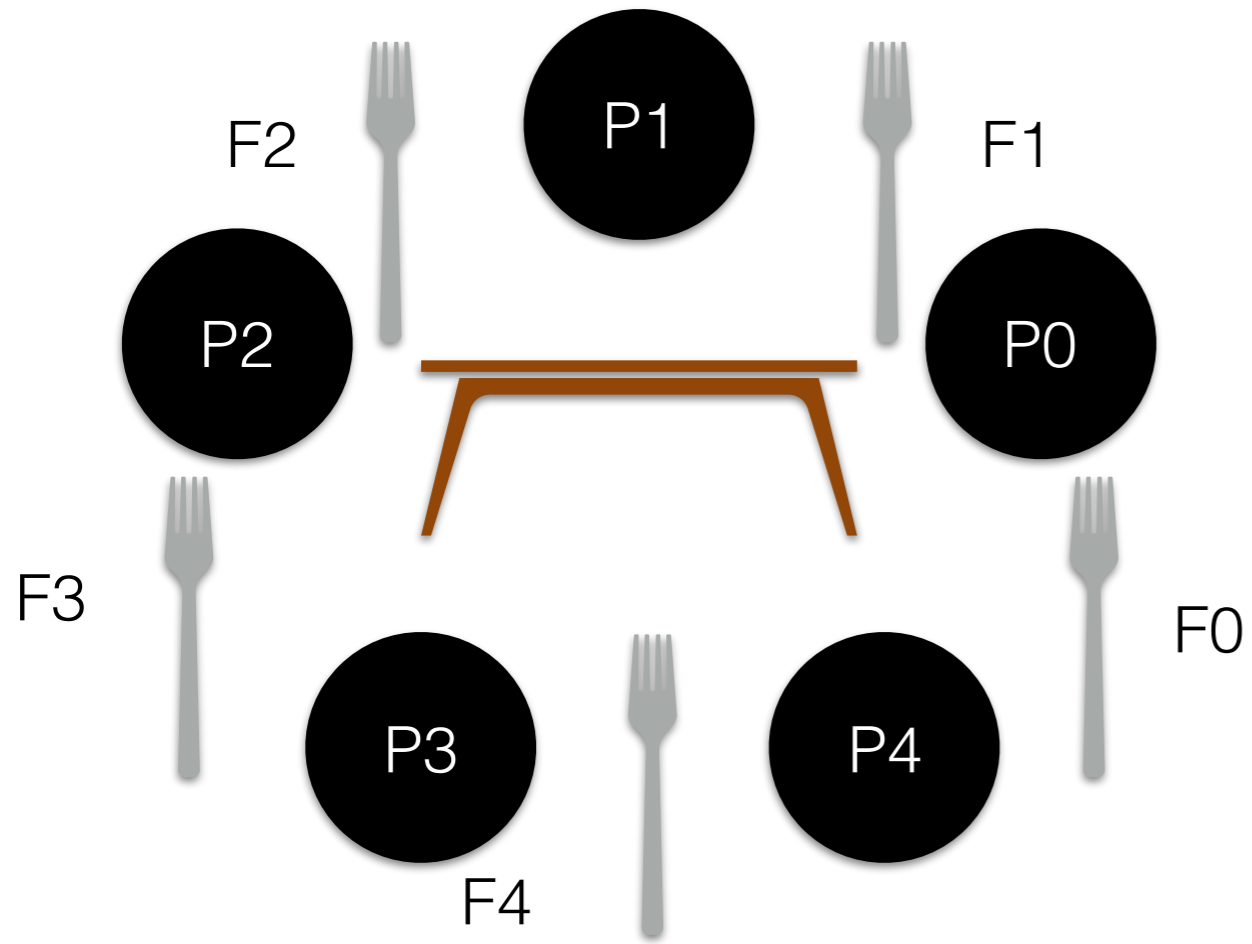
Correct Solution

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;
```

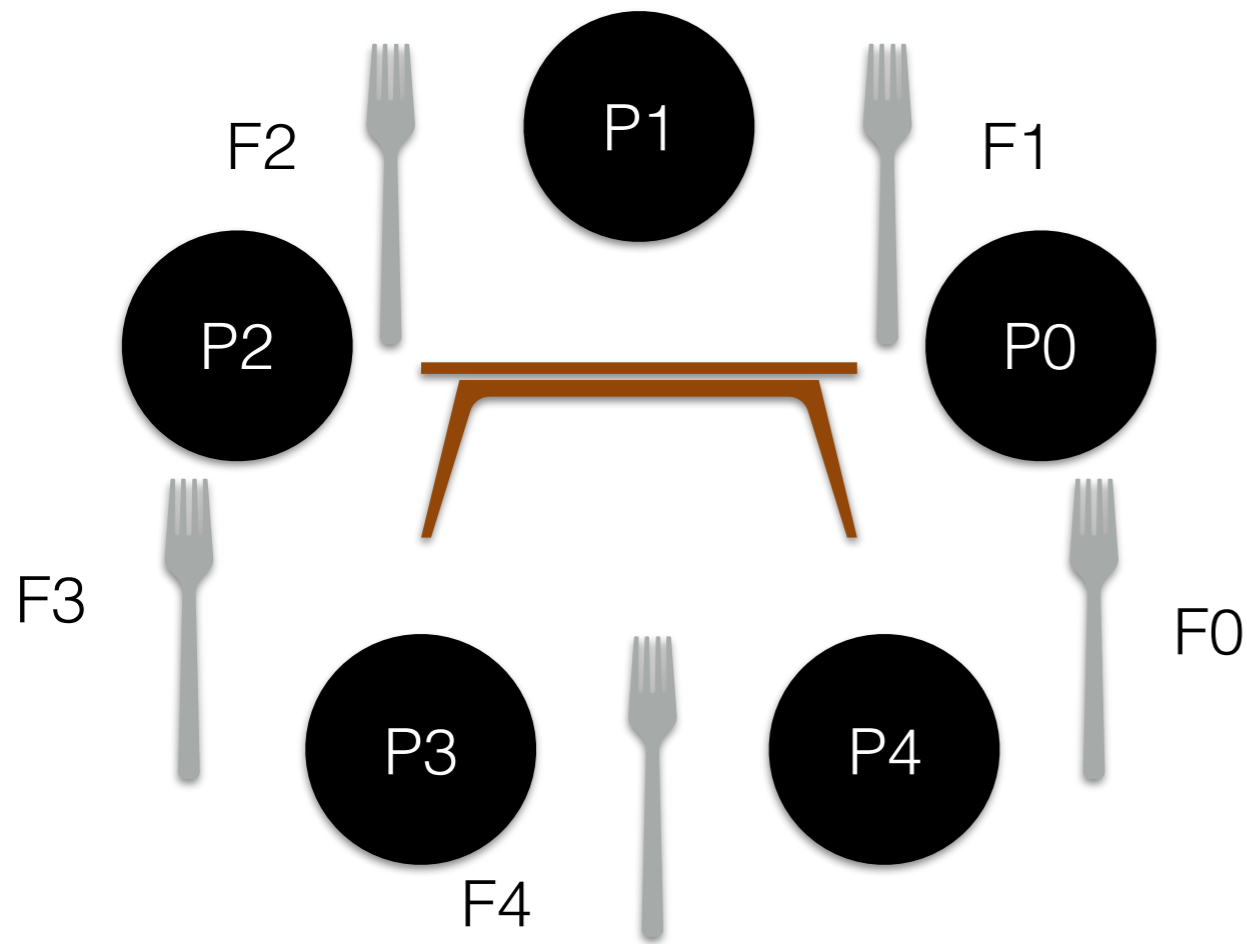
```
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&empty); // line p1
9         sem_wait(&mutex);
10        put(i); // line p2
11        sem_post(&mutex);
12        sem_post(&full); // line p3
13    }
14 }
15
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full); // line c1
20        sem_wait(&mutex);
21        int tmp = get(); // line c2
22        sem_post(&mutex); //
23        sem_post(&empty); // line c3
24        printf("%d\n", tmp);
25    }
26 }
```

Dining Philosopher's Problem

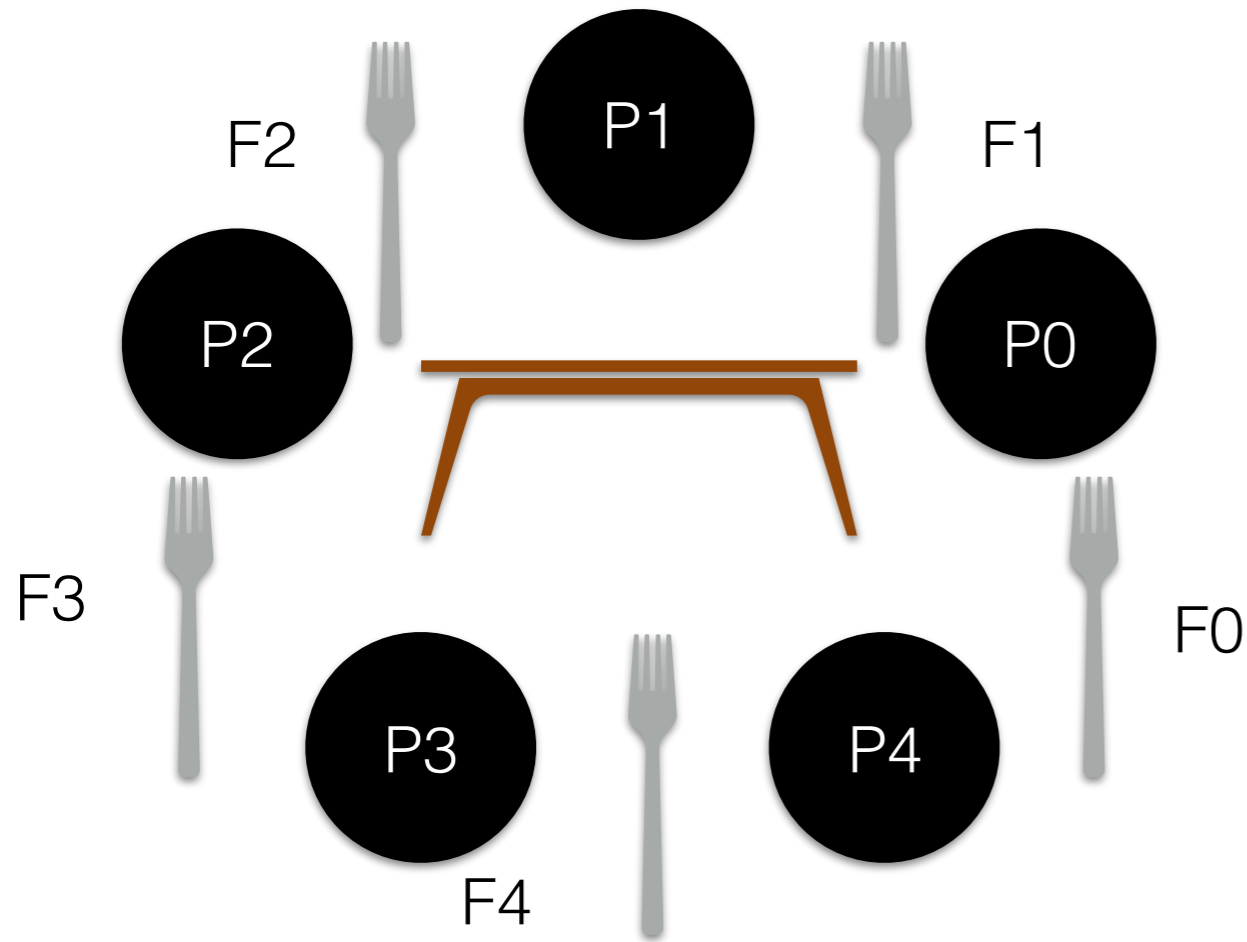


Dining Philosopher's Problem



```
1 void getforks() {  
2   sem_wait(forks[left(p)]);  
3   sem_wait(forks[right(p)]);  
4 }
```

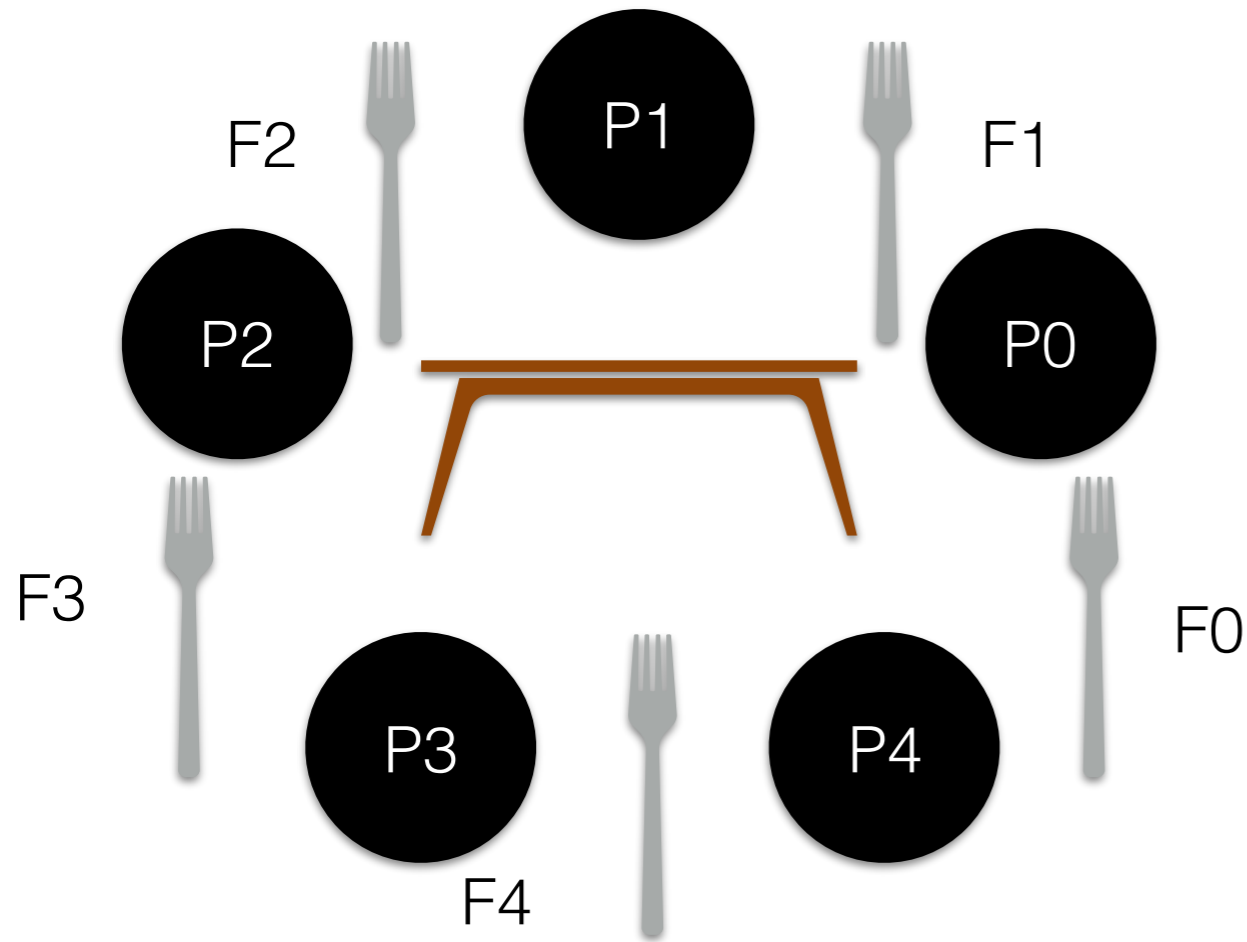
Dining Philosopher's Problem



```
1 void getforks() {  
2  sem_wait(forks[left(p)]);  
3  sem_wait(forks[right(p)]);  
4 }
```

```
1 void putforks() {  
2  sem_post(forks[left(p)]);  
3  sem_post(forks[right(p)]);  
4 }
```

Dining Philosopher's Problem

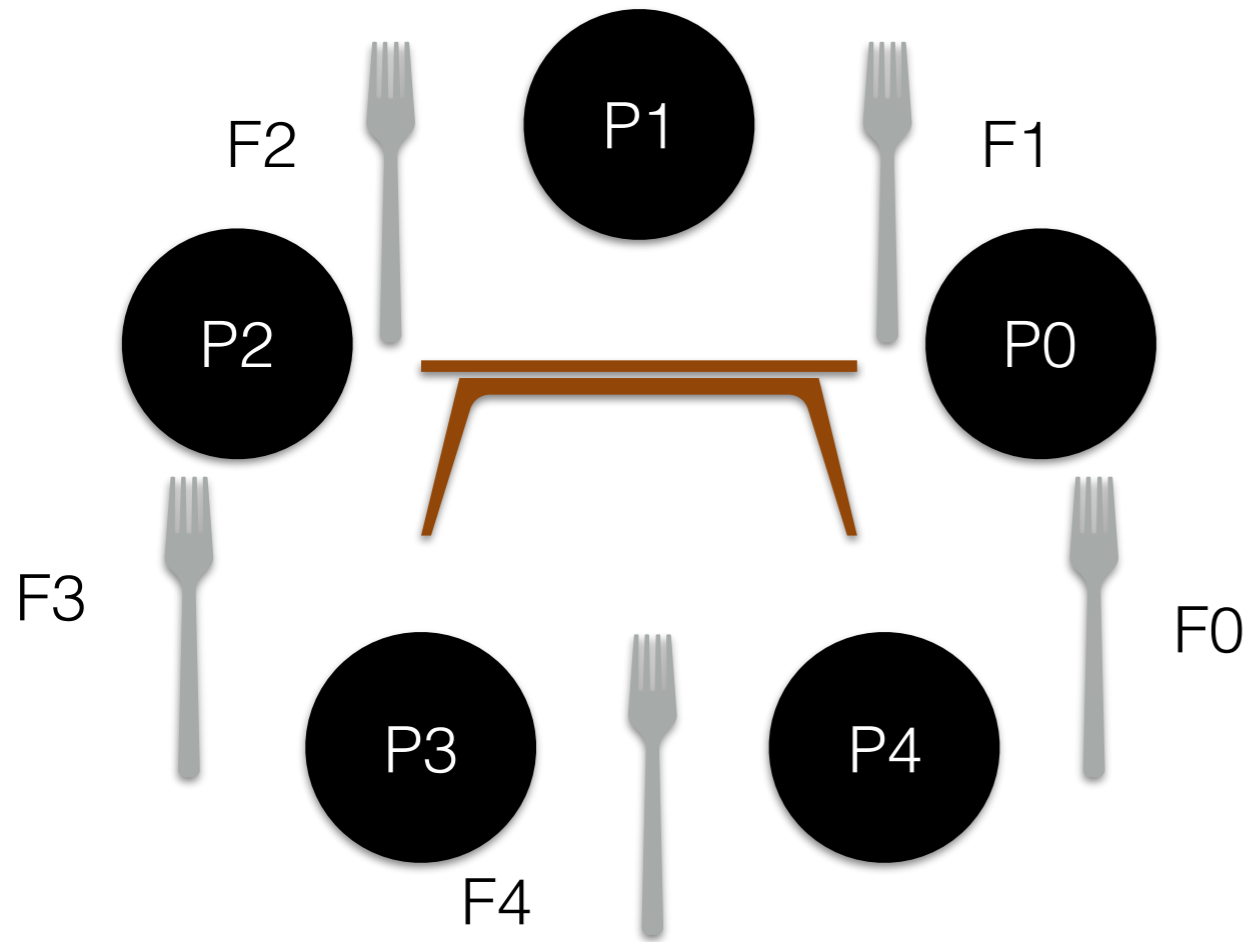


```
1 void getforks() {  
2   sem_wait(forks[left(p)]);  
3   sem_wait(forks[right(p)]);  
4 }
```

```
1 void putforks() {  
2   sem_post(forks[left(p)]);  
3   sem_post(forks[right(p)]);  
4 }
```

- P0 picks F0; P1 picks F1; ..., P4 picks F4

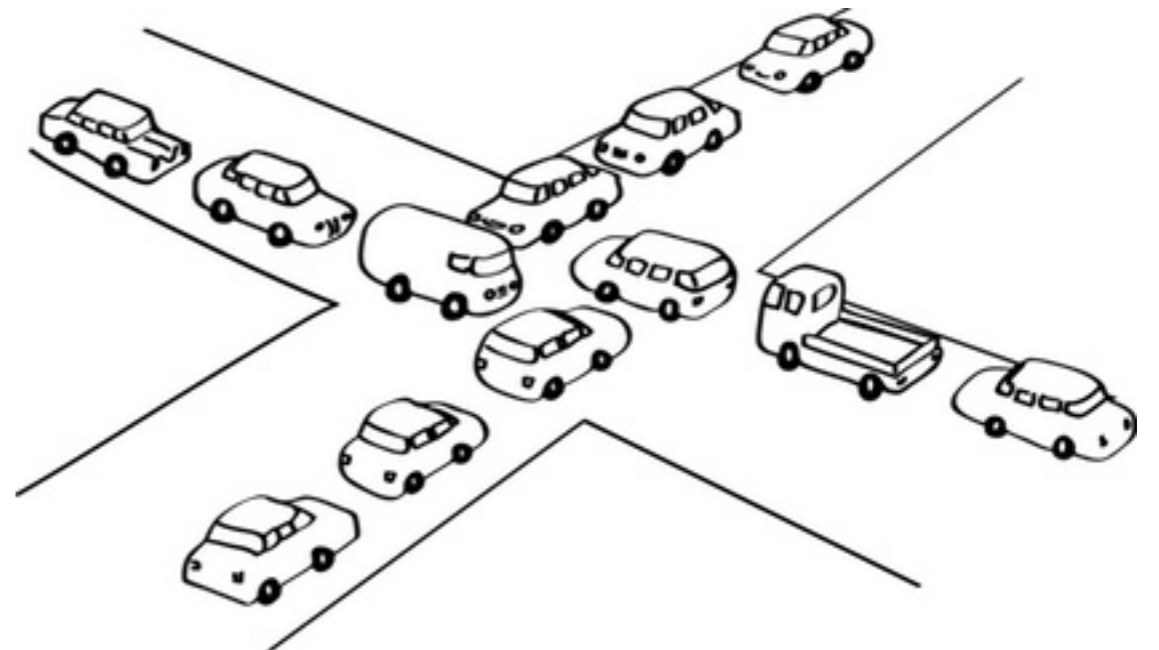
Dining Philosopher's Problem



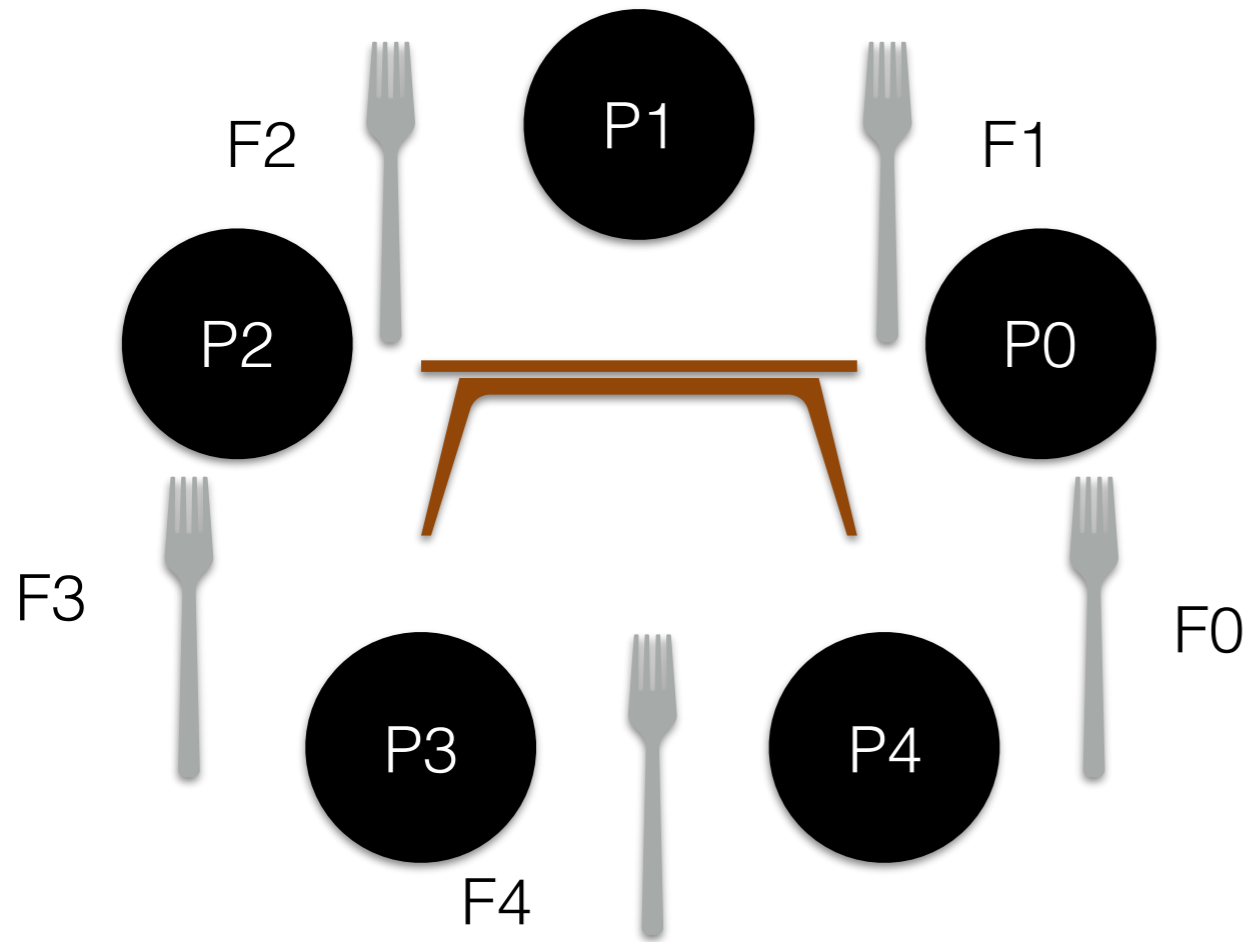
- P0 picks F0; P1 picks F1; ..., P4 picks F4

```
1 void getforks() {  
2   sem_wait(forks[left(p)]);  
3   sem_wait(forks[right(p)]);  
4 }
```

```
1 void putforks() {  
2   sem_post(forks[left(p)]);  
3   sem_post(forks[right(p)]);  
4 }
```



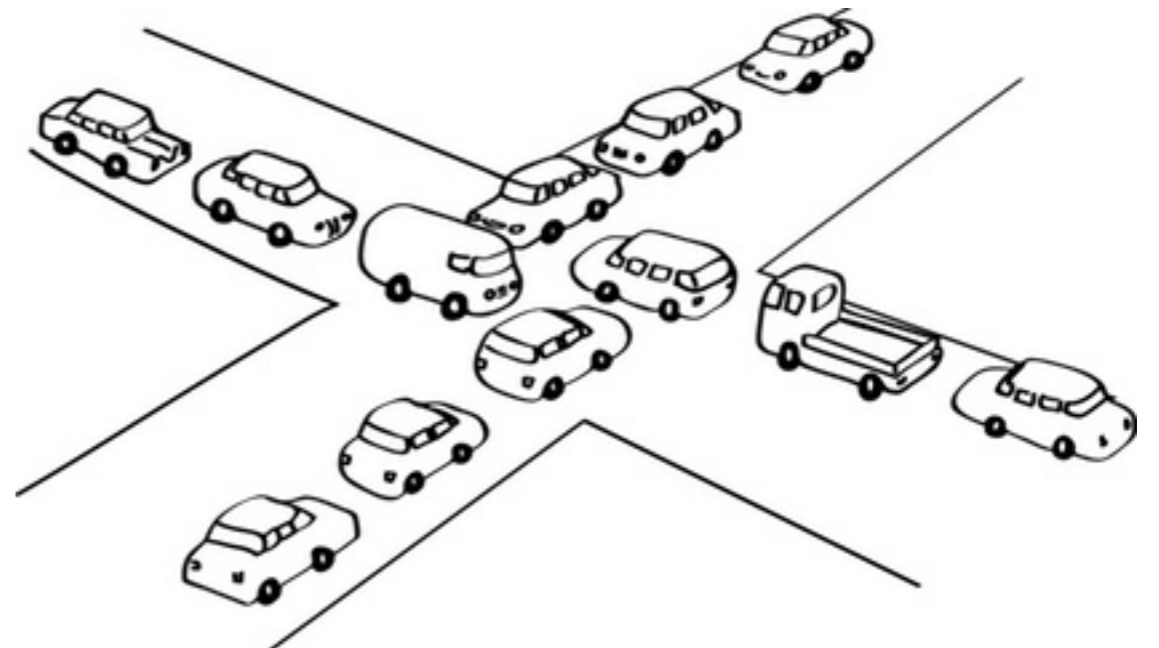
Dining Philosopher's Problem



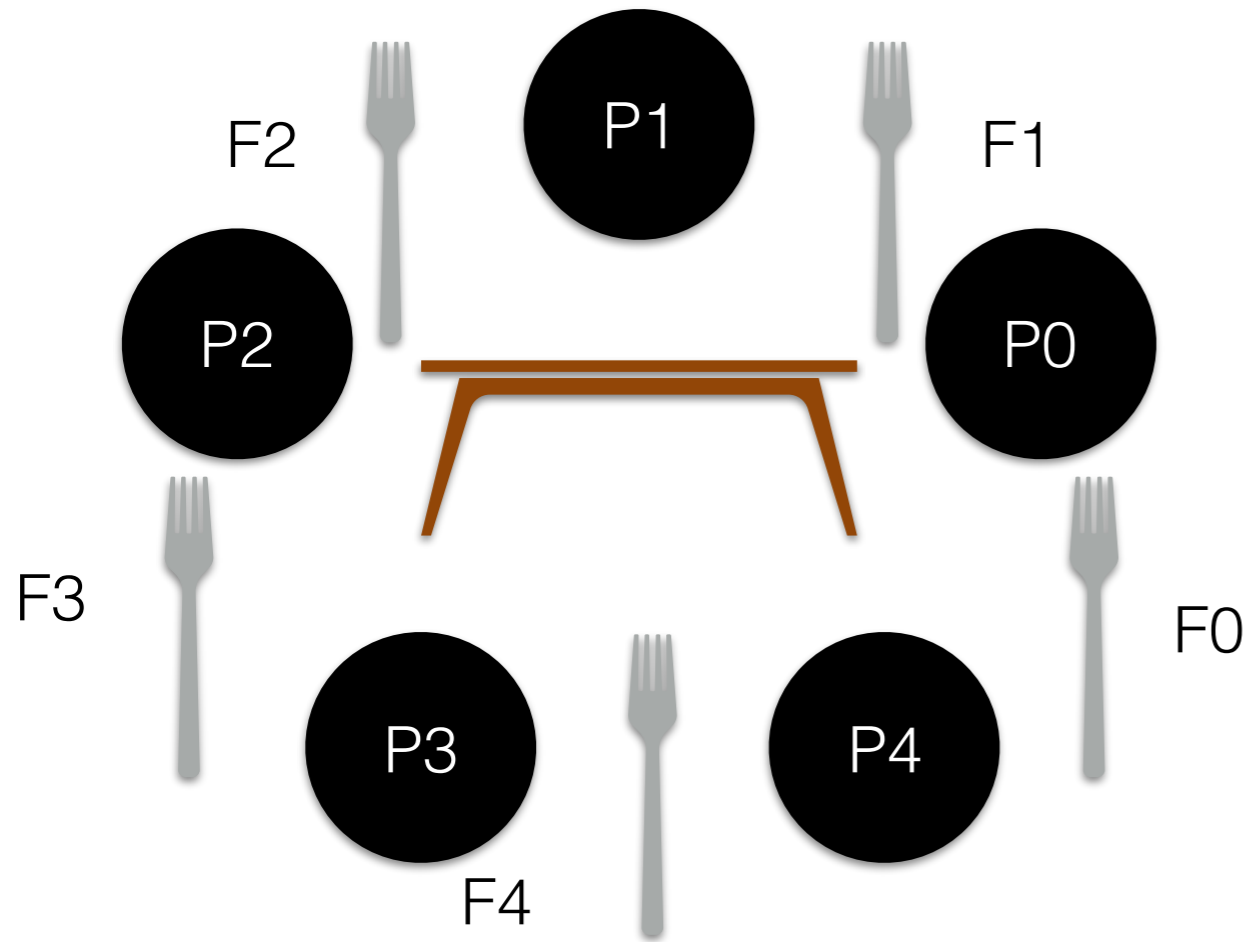
```
1 void getforks() {  
2   sem_wait(forks[left(p)]);  
3   sem_wait(forks[right(p)]);  
4 }
```

```
1 void putforks() {  
2   sem_post(forks[left(p)]);  
3   sem_post(forks[right(p)]);  
4 }
```

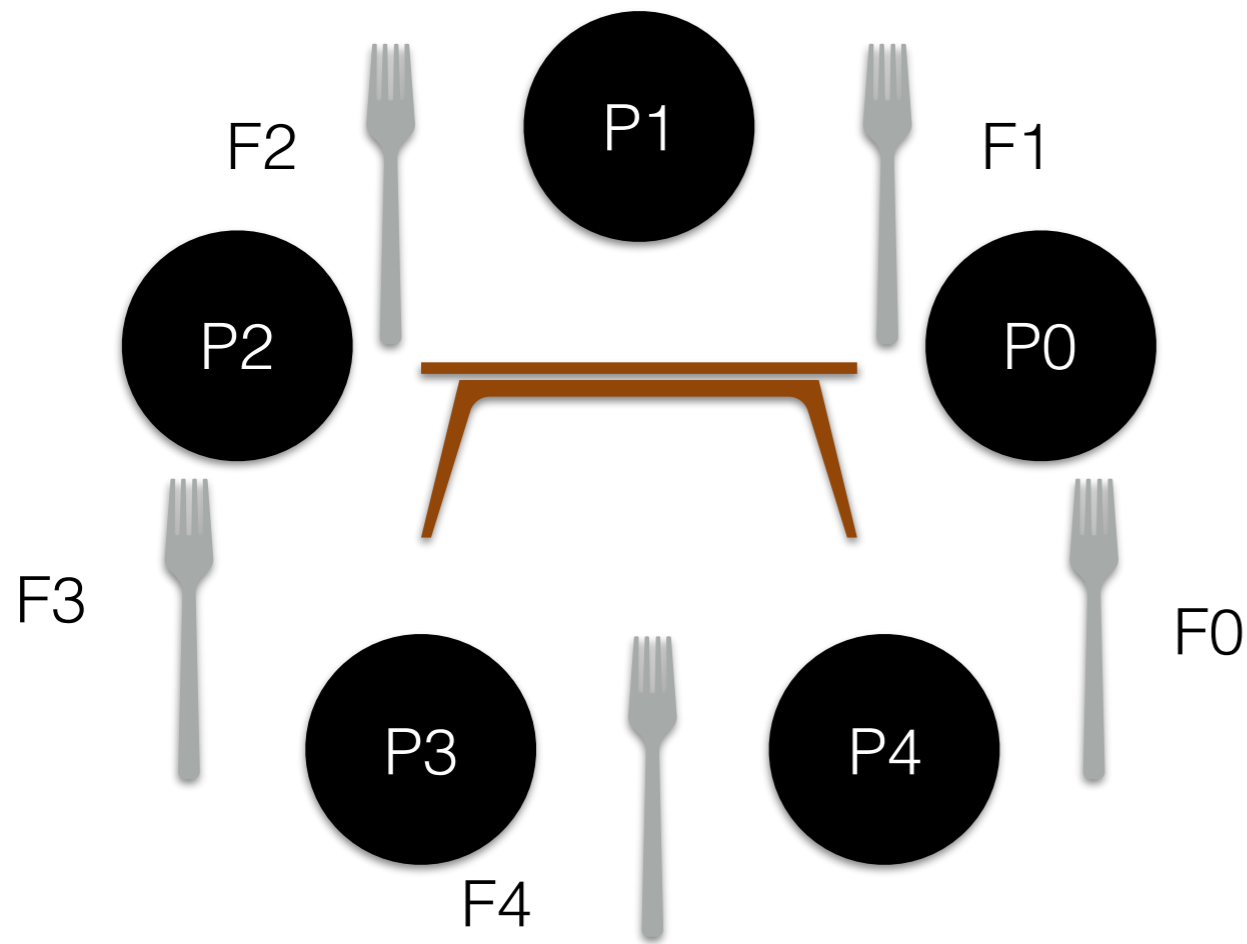
- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?



Dining Philosopher's Problem

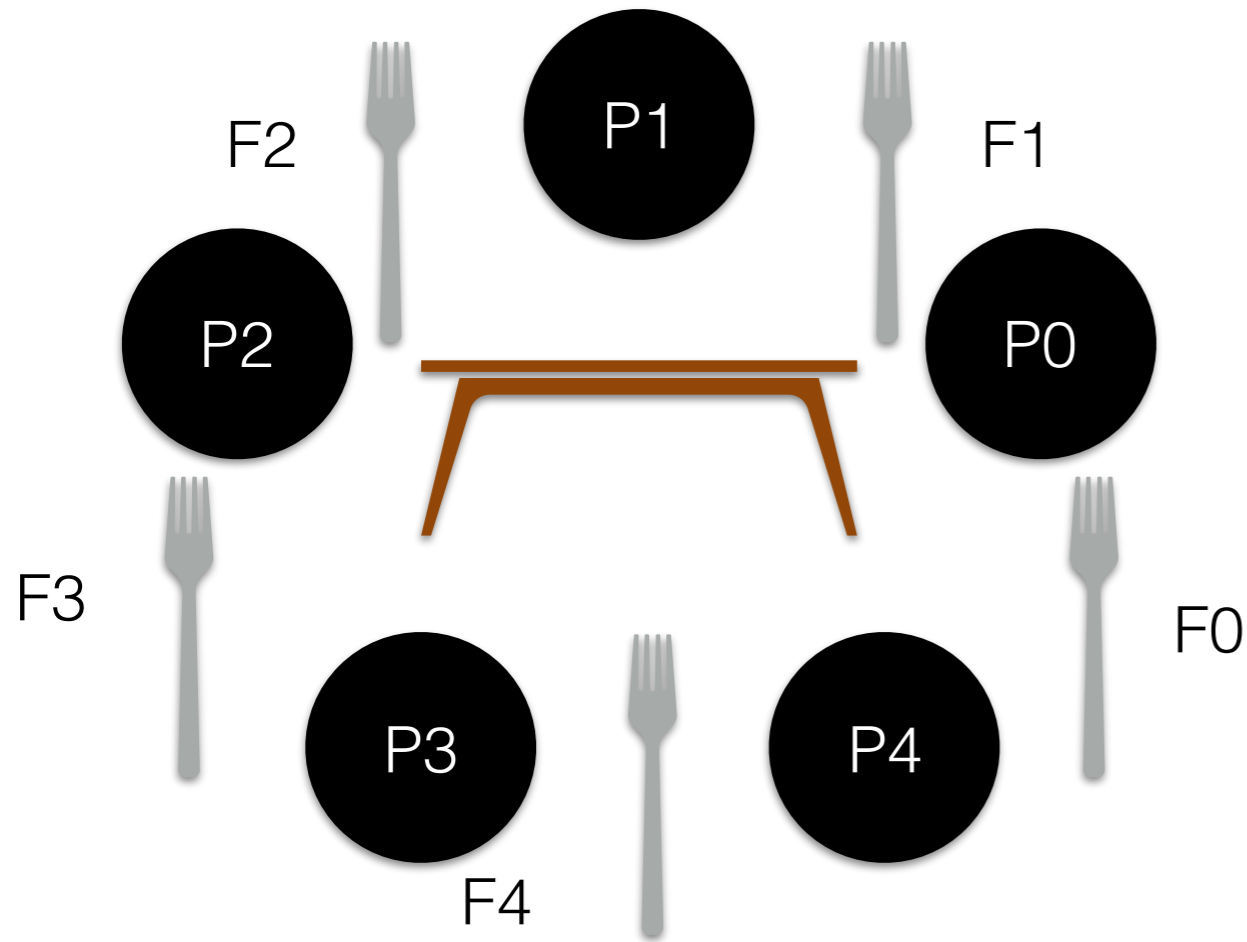


Dining Philosopher's Problem



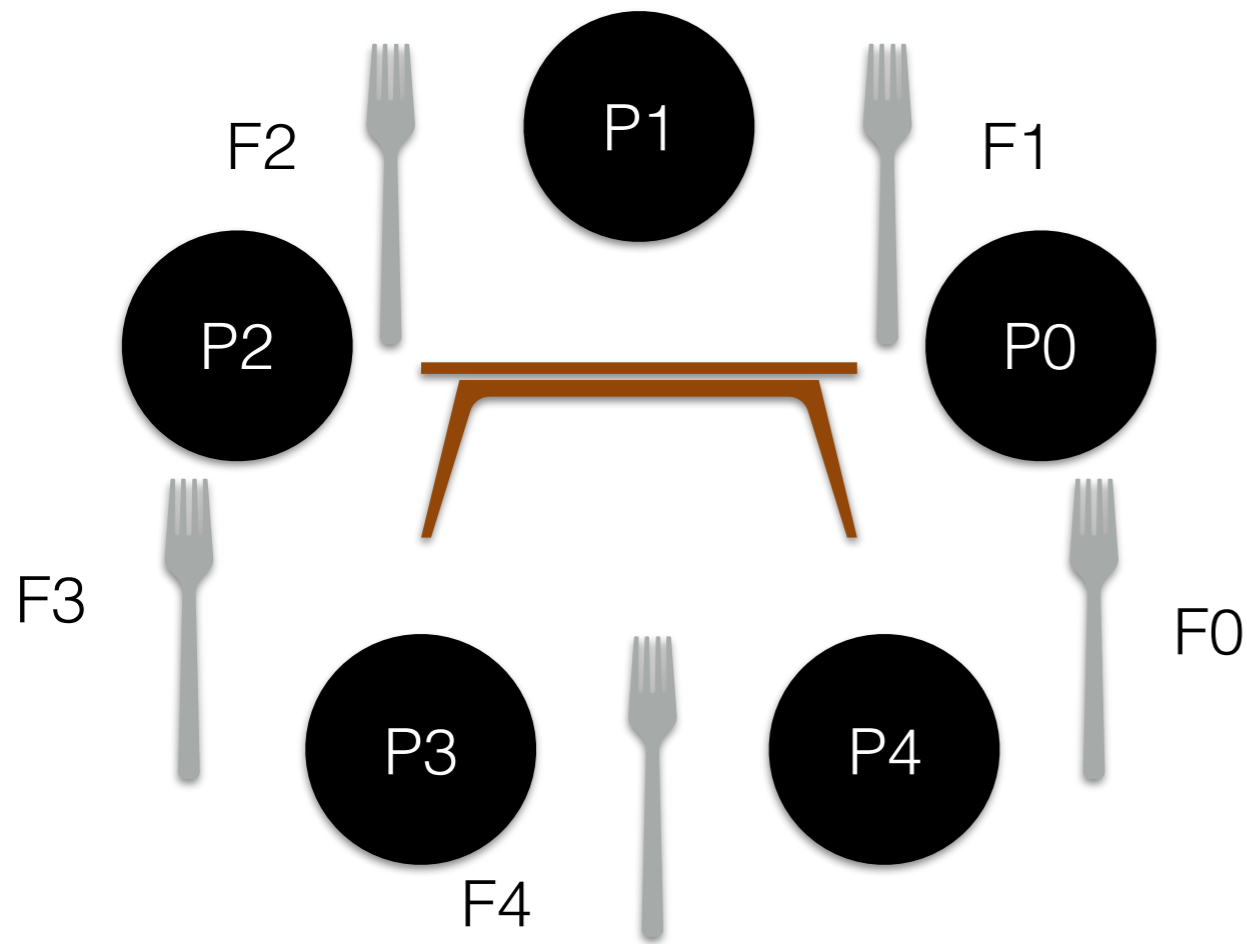
- P0 picks F0; P1 picks F1; ..., P4 picks F4

Dining Philosopher's Problem



- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?

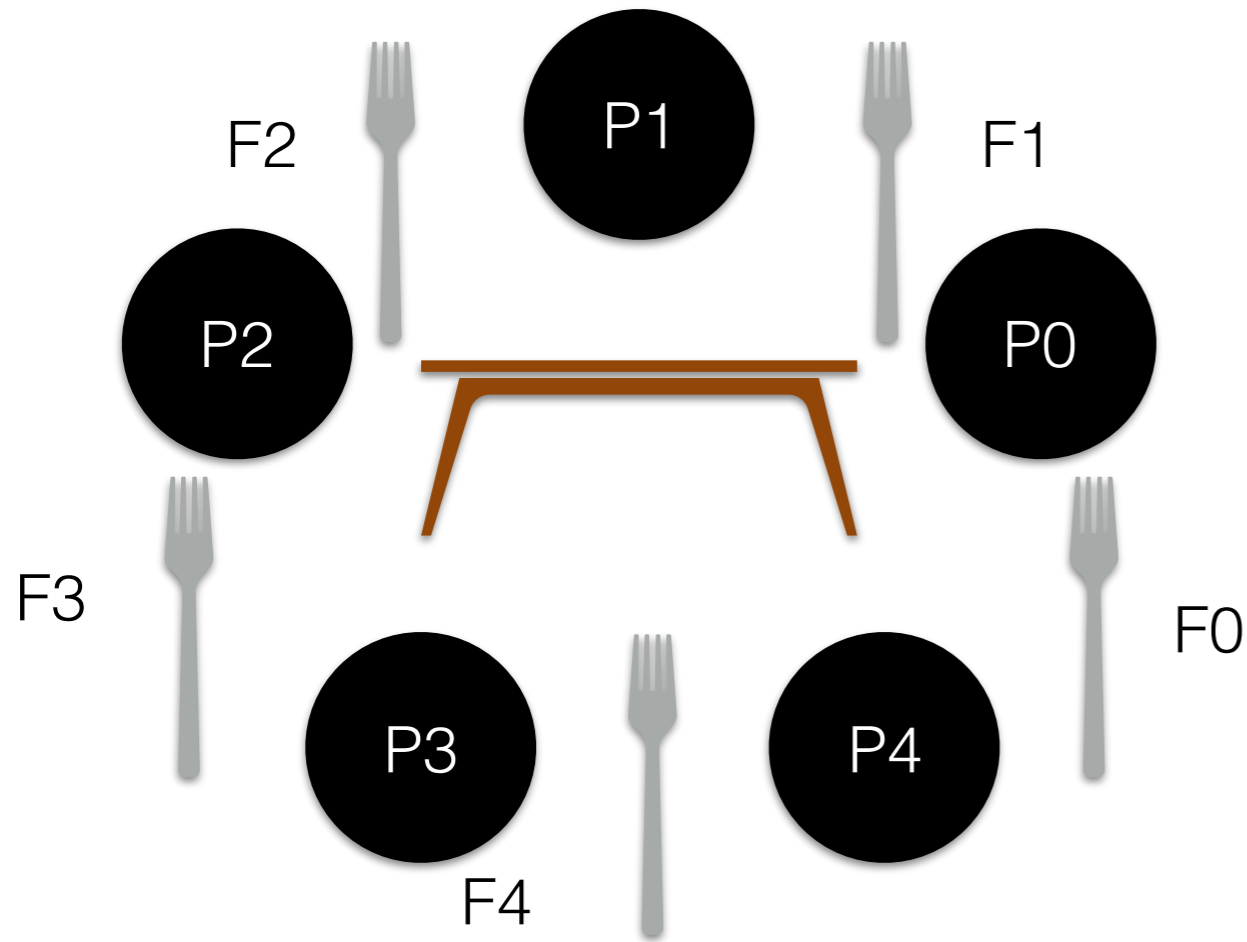
Dining Philosopher's Problem



- If $P == 4$:

- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?

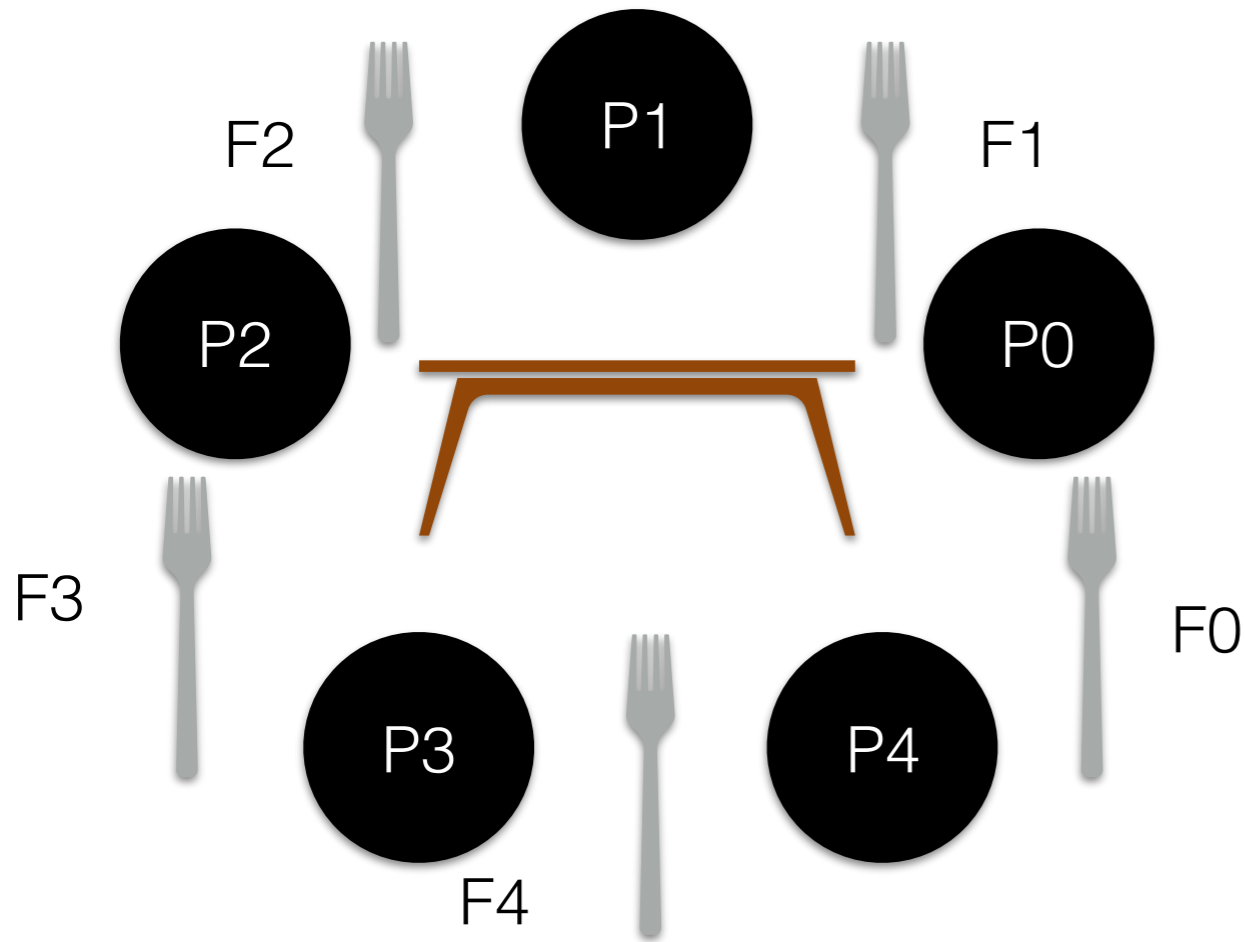
Dining Philosopher's Problem



- If $P == 4$:
 - Wait on Right

- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?

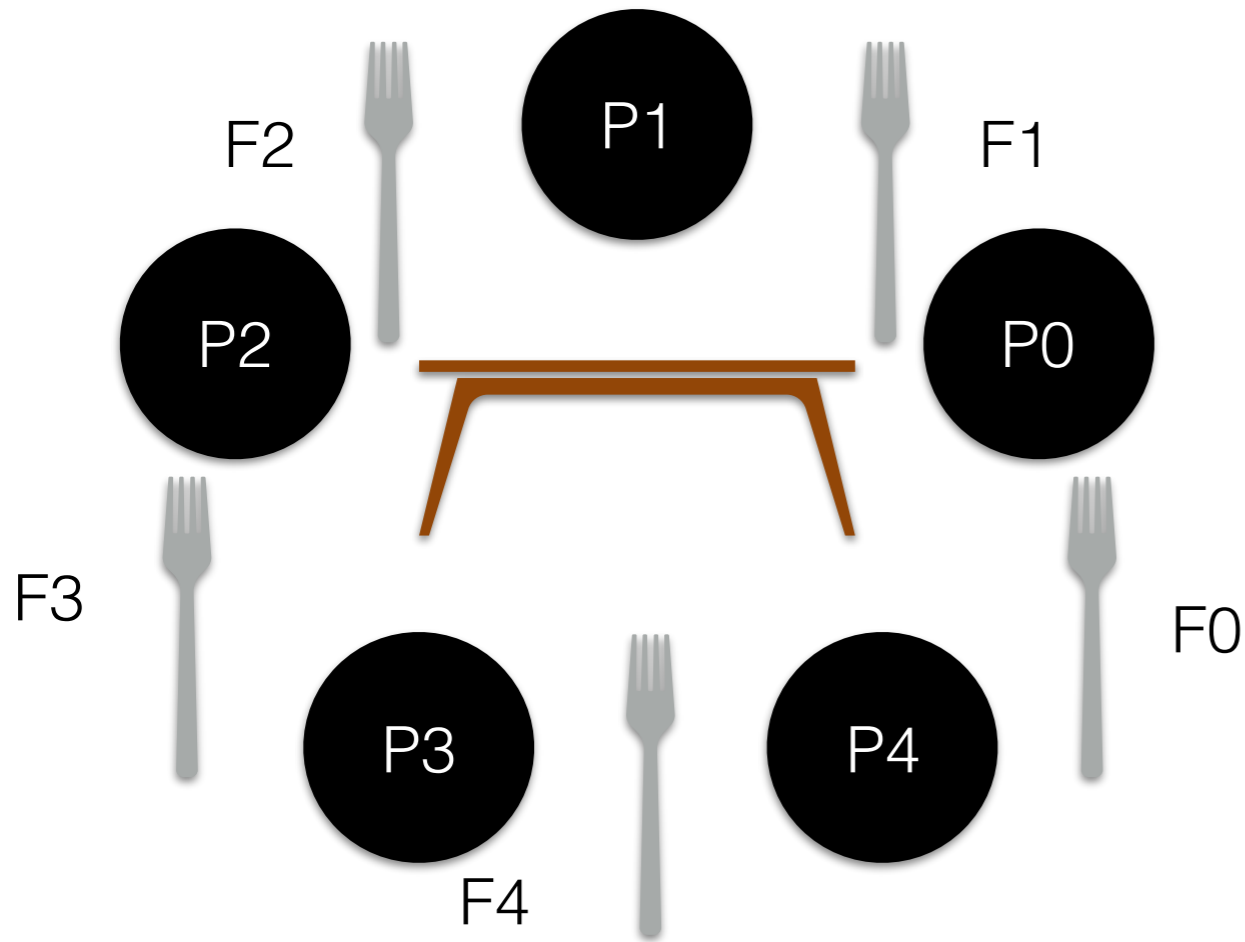
Dining Philosopher's Problem



- If $P == 4$:
 - Wait on Right
 - Want on Left

- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?

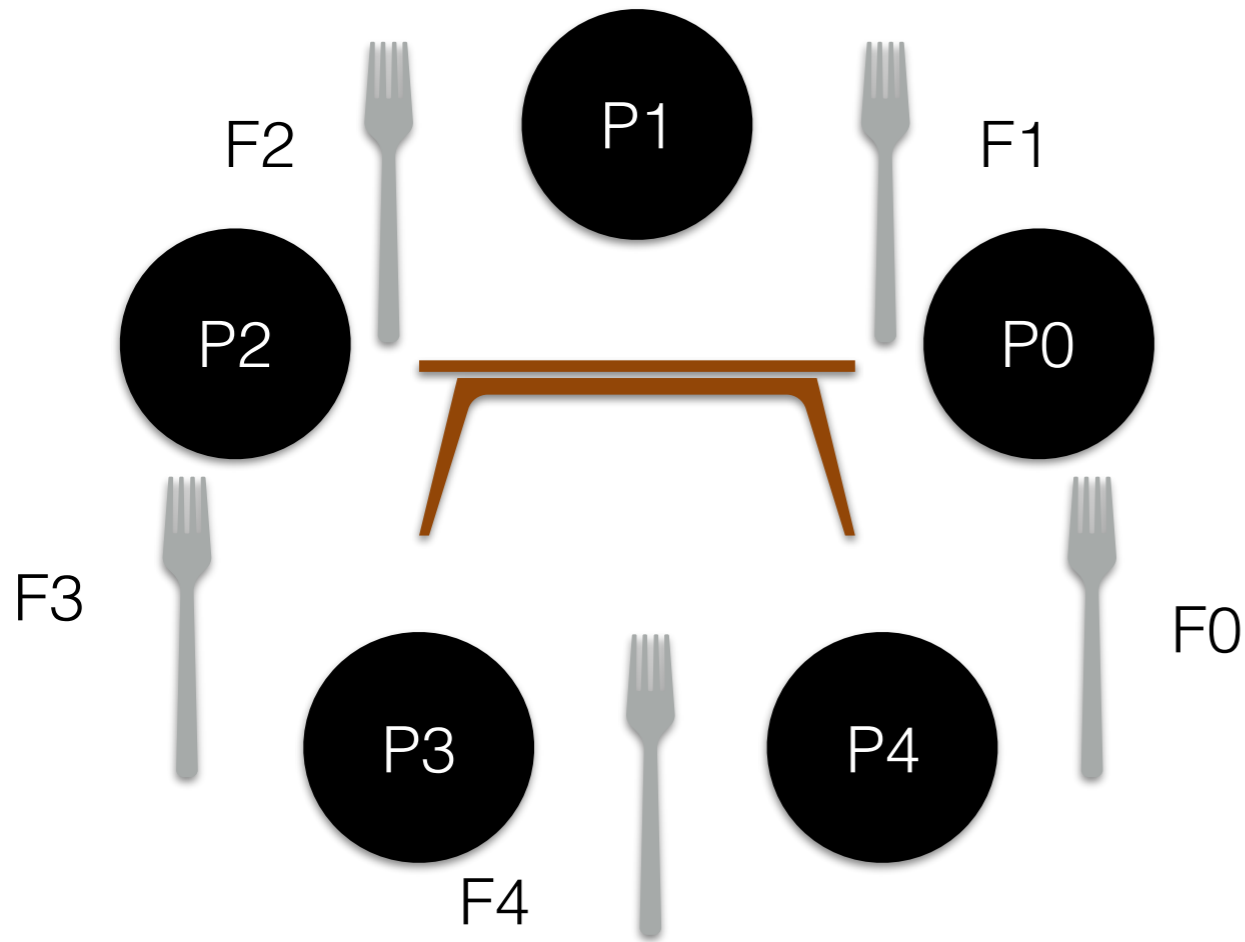
Dining Philosopher's Problem



- If $P == 4$:
 - Wait on Right
 - Want on Left
- Else:

- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?

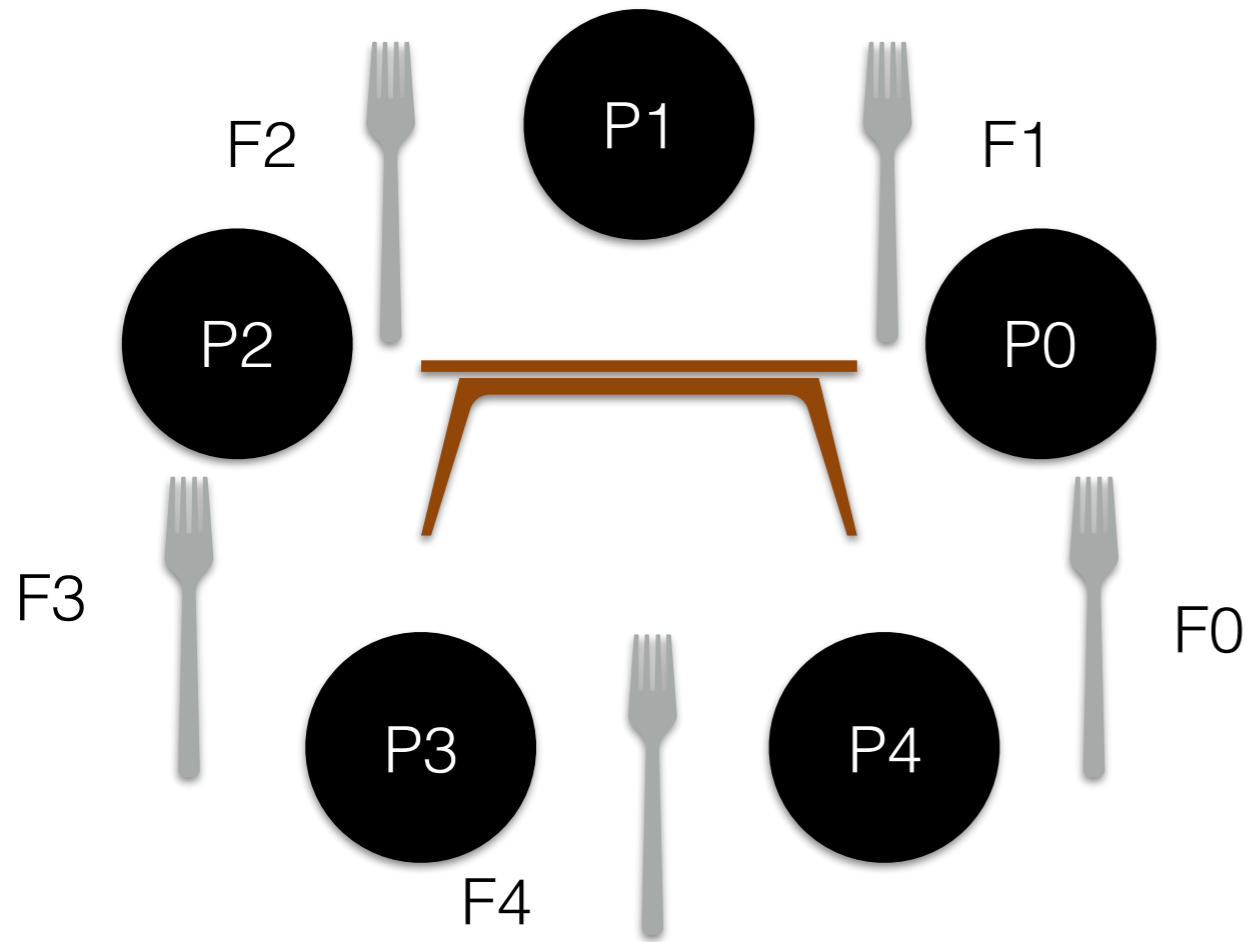
Dining Philosopher's Problem



- If $P == 4$:
 - Wait on Right
 - Wait on Left
- Else:
 - Wait on Left

- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?

Dining Philosopher's Problem



- If $P == 4$:
 - Wait on Right
 - Wait on Left
- Else:
 - Wait on Left
 - Wait on Right

- P0 picks F0; P1 picks F1; ..., P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?

Condition Variables

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
 - Assumes lock is held when wait() is called

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
 - Assumes lock is held when wait() is called
 - Puts caller to sleep + atomically releases lock

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
 - Assumes lock is held when wait() is called
 - Puts caller to sleep + atomically releases lock
 - When awoken, reacquires lock before returning

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
 - Assumes lock is held when wait() is called
 - Puts caller to sleep + atomically releases lock
 - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
 - Assumes lock is held when wait() is called
 - Puts caller to sleep + atomically releases lock
 - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**
 - Wake a single waiting thread

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
 - Assumes lock is held when wait() is called
 - Puts caller to sleep + atomically releases lock
 - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**
 - Wake a single waiting thread
 - If there is no waiting thread, just return, do nothing

Semaphores Implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
1 #include <semaphore.h>
2 sem_t s;
```

Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
1 #include <semaphore.h>
2 sem_t s;
```

Our implementation

```
1 typedef struct __Zem_t {
2     int value;
3     pthread_cond_t cond;
4     pthread_mutex_t lock;
5 } Zem_t;
6
```


Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

API	
-----	--

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
1 int sem_init(sem_t *s,  
int init_val) {  
2   s->value=init_val;  
3 }
```

Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
1 int sem_init(sem_t *s,  
int init_val) {  
2     s->value=init_val;  
3 }
```

Our implementation

```
1 // only one thread can call this  
2 void Zem_init(Zem_t *s, int value) {  
3     s->value = value;  
4     Cond_init(&s->cond);  
5     Mutex_init(&s->lock);  
6 }
```

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

API	
-----	--

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Our implementation

```
1 int sem_wait(sem_t *s) {  
2     s->value -= 1  
3     wait if s->value < 0  
4 }
```

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
1 int sem_wait(sem_t *s) {  
2   s->value -= 1  
3   wait if s->value < 0  
4 }
```

Our implementation

Atomic operation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
1 int sem_wait(sem_t *s) {  
2   s->value -= 1  
3   wait if s->value < 0  
4 }
```

Atomic operation

Our implementation

```
void Zem_wait(Zem_t *s) {  
1   Mutex_lock(&s->lock);  
2   while (s->value <= 0)  
3     Cond_wait(&s->cond, &s->lock);  
4   s->value--;  
5   Mutex_unlock(&s->lock);  
6 }
```

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

API	
-----	--

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API	Our implementation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Our implementation

```
1 int sem_post(sem_t *s) {  
2     s->value += 1  
3     wake one waiting thread if  
any  
4 }
```

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

Our implementation

```
1 int sem_post(sem_t *s) {  
2     s->value += 1  
3     wake one waiting thread if  
any  
4 }
```

Atomic operation

Semaphores Implementation

- Build semaphores using **locks** and **condition variables**
- Any critical section should require **locking**
- **Signal** and **Wait** on condition
- Don't maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

API

```
1 int sem_post(sem_t *s) {  
2     s->value += 1  
3     wake one waiting thread if  
any  
4 }
```

Our implementation

```
void Zem_post(Zem_t *s) {  
23 Mutex_lock(&s->lock);  
24 s->value++;  
25 Cond_signal(&s->cond);  
26 Mutex_unlock(&s->lock);  
27 }
```

Atomic operation

Readers-writer lock

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

- Insert:

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

- Insert:
 - Changes state of the list

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

- Insert:
 - Changes state of the list
 - Traditional critical section needed

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

- Insert:
 - Changes state of the list
 - Traditional critical section needed
- Lookup

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

- Insert:
 - Changes state of the list
 - Traditional critical section needed
- Lookup
 - Read the data structure, no modification

Readers-writer lock

Imagine a number of concurrent list operations, including inserts and simple lookups.

- Insert:
 - Changes state of the list
 - Traditional critical section needed
- Lookup
 - Read the data structure, no modification
 - As long as we guarantee no insertion, multiple readers can read concurrently

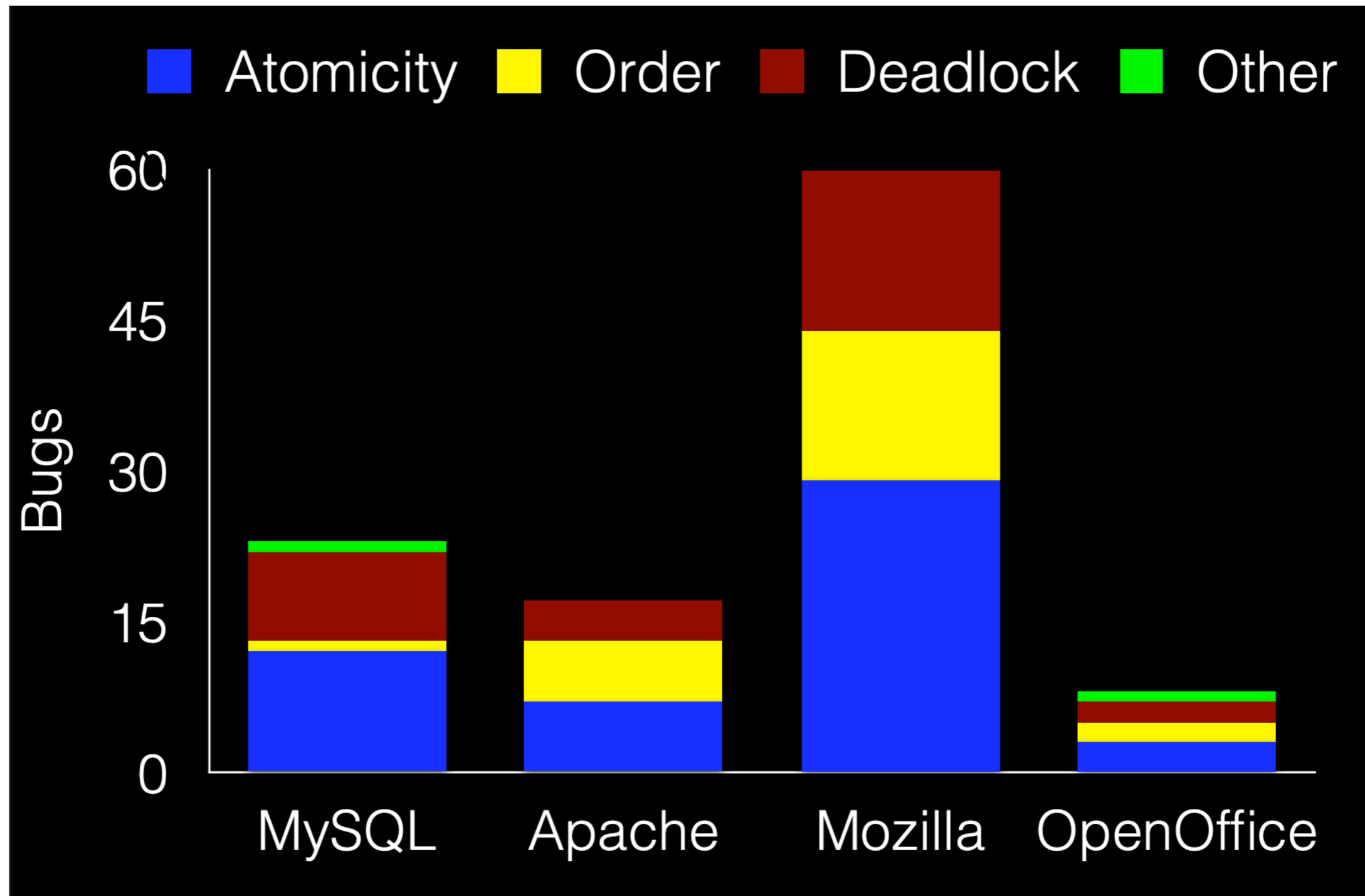
Concurrency Bugs

Spot the murderer ...



<https://www.youtube.com/watch?v=izGSOsAGIVQ>

Concurrency Bugs



Types of bugs in 4 major projects from 500K bug reports

Concurrency Bugs — Atomicity

Concurrency Bugs — Atomicity

MySQL bug ...

Concurrency Bugs — Atomicity

MySQL bug ...

1 Thread1::

```
2  if(thd->proc_info){
3    ...
4    fputs(thd->proc_info , ...);
5  ...
6  }
```

Concurrency Bugs — Atomicity

MySQL bug ...

1 Thread1::

```
2  if(thd->proc_info){
3      ...
4      fputs(thd->proc_info , ...);
5  ...
6  }
```

8 Thread2::

```
9  thd->proc_info = NULL;
```

Concurrency Bugs — Atomicity

MySQL bug ...

1 Thread1::

```
2  if(thd->proc_info){
3      ...
4      fputs(thd->proc_info , ...);
5  ...
6  }
```

8 Thread2::

```
9  thd->proc_info = NULL;
```

- Is this problematic?

Concurrency Bugs — Atomicity

MySQL bug ...

1 Thread1::

```
2  if(thd->proc_info){
3      ...
4      fputs(thd->proc_info , ...);
5  ...
6  }
```

8 Thread2::

```
9  thd->proc_info = NULL;
```

- Is this problematic?
 - Yes, else we wouldn't be discussing ...

Concurrency Bugs — Atomicity

MySQL bug ...

1 Thread1::

```
2  if(thd->proc_info){
3      ...
4      fputs(thd->proc_info , ...);
5  ...
6  }
```

8 Thread2::

```
9  thd->proc_info = NULL;
```

- Is this problematic?
 - Yes, else we wouldn't be discussing ...
 - How?

Concurrency Bugs — Atomicity

```
1 pthread_mutex_t lock =  
PTHREAD_MUTEX_INITIALIZER;  
2
```

3 Thread1::

```
4 pthread_mutex_lock(&lock);  
5 if(thd->proc_info){  
6   ...  
7   fputs(thd->proc_info , ...);  
8   ...  
9 }  
10 pthread_mutex_unlock(&lock);
```

1 Thread2::

```
2 pthread_mutex_lock(&lock);  
3 thd->proc_info = NULL;  
4 pthread_mutex_unlock(&lock);
```

Concurrency Bugs — Atomicity

Simple Solution

```
1 pthread_mutex_t lock =  
PTHREAD_MUTEX_INITIALIZER;  
2
```

3 Thread1::

```
4 pthread_mutex_lock(&lock);  
5 if(thd->proc_info){  
6   ...  
7   fputs(thd->proc_info , ...);  
8   ...  
9 }  
10 pthread_mutex_unlock(&lock);
```

1 Thread2::

```
2 pthread_mutex_lock(&lock);  
3 thd->proc_info = NULL;  
4 pthread_mutex_unlock(&lock);
```


Concurrency Bugs — Order Violation

1 Thread1::

```
2 void init(){
3   mThread =
  PR_CreateThread(mMain, ...);
4 }
5
```

6 Thread2::

```
7 void mMain(...){
8   mState = mThread->State
9 }
```

Concurrency Bugs — Order Violation

Mozilla bug ...

1 Thread1::

```
2 void init(){  
3   mThread =  
PR_CreateThread(mMain, ...);  
4 }  
5
```

6 Thread2::

```
7 void mMain(...){  
8   mState = mThread->State  
9 }
```

Concurrency Bugs — Order Violation

Mozilla bug ...

1 Thread1::

```
2 void init(){
3   mThread =
4   PR_CreateThread(mMain, ...);
5 }
```

6 Thread2::

```
7 void mMain(...){
8   mState = mThread->State
9 }
```

- Is this problematic?

Concurrency Bugs — Order Violation

Mozilla bug ...

1 Thread1::

```
2 void init(){  
3   mThread =  
PR_CreateThread(mMain, ...);  
4 }  
5
```

6 Thread2::

```
7 void mMain(...){  
8   mState = mThread->State  
9 }
```

- Is this problematic?
 - Yes, else we wouldn't be discussing ...

Concurrency Bugs — Order Violation

Mozilla bug ...

1 Thread1::

```
2 void init(){
3   mThread =
PR_CreateThread(mMain, ...);
4 }
5
```

6 Thread2::

```
7 void mMain(...){
8   mState = mThread->State
9 }
```

- Is this problematic?
 - Yes, else we wouldn't be discussing ...
 - How?

Concurrency Bugs — Order Violation

Concurrency Bugs — Order Violation

```
1 pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;  
2 pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;  
3 int mtInit = 0;
```

Concurrency Bugs — Order Violation

```
1 pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3 int mtInit = 0;
```

1 Thread 1::

```
2 void init(){
3   ...
4   mThread = PR_CreateThread(mMain,...);
5
6   // signal that the thread has been created.
7   pthread_mutex_lock(&mtLock);
8   mtInit = 1;
9   pthread_cond_signal(&mtCond);
10  pthread_mutex_unlock(&mtLock);
11  ...
12 }
```


Concurrency Bugs — Order Violation

```
1 pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3 int mtInit = 0;
```

1 Thread 1::

```
2 void init(){
3   ...
4   mThread = PR_CreateThread(mMain,...);
5
6   // signal that the thread has been created.
7   pthread_mutex_lock(&mtLock);
8   mtInit = 1;
9   pthread_cond_signal(&mtCond);
10  pthread_mutex_unlock(&mtLock);
11  ...
12 }
```

20 Thread2::

```
21 void mMain(...){
    // wait for the thread to be initialized
    ...
22  pthread_mutex_lock(&mtLock);
23  while(mtInit == 0)
24    pthread_cond_wait(&mtCond,
    &mtLock);
25
26  pthread_mutex_unlock(&mtLock);
27  mState = mThread->State;
28 }
```

Concurrency Bugs — Deadlock

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section
- Context Switch

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section
- Context Switch
- Thread T2 gets Lock L2 and Lock L1

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Thread T1 gets Lock L2
- Thread T1 completes critical section
- Context Switch
- Thread T2 gets Lock L2 and Lock L1
- Works :)

Concurrency Bugs — Deadlock

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

- Thread T1 gets Lock L1
- Context Switch

Thread 2

Lock(L2);

Lock(L1);

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2

Thread 2

Lock(L2);

Lock(L1);

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch

Thread 2

Lock(L2);

Lock(L1);

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn't have Lock 2

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn't have Lock 2
- Context Switch

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn't have Lock 2
- Context Switch
- Thread t2 waits since it doesn't have Lock 1

Concurrency Bugs — Deadlock

Thread 1

Lock(L1);

Lock(L2);

- Thread T1 gets Lock L1
- Context Switch
- Thread T2 gets Lock L2
- Context Switch
- Thread T1 waits since it doesn't have Lock 2
- Context Switch
- Thread t2 waits since it doesn't have Lock 1

Thread 2

Lock(L2);

Lock(L1);

