Operating Systems

Lecture 25: Common Concurrency Problems

Nipun Batra
Oct 30, 2018

Producer Consumer Problem using Semaphores
Adding mutual exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);    // line p0 (NEW LINE)
    sem_wait(&empty);    // line p1
      put(i);    // line p2
    sem_post(&full);    // line p3
    sem_post(&mutex);    // line p4 (NEW LINE)
  }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);    // line c0 (NEW LINE)
    sem_wait(&full);    // line c1
    int tmp = get();    // line c2
    sem_post(&empty);    // line c3
    sem_post(&mutex);    // line c4 (NEW LINE)
    printf("%d\n", tmp);
  }
}
```

Producer Consumer Problem using Semaphores Adding mutual exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
int i;
for (i = 0; i < loops; i++) {
sem_wait(&mutex);   // line p0 (NEW LINE)
sem_wait(&empty);   // line p1
 put(i);   // line p2
 sem_post(&full);   // line p3
 sem_post(&mutex);   // line p4 (NEW LINE)
 }
}
```

```
void *consumer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&mutex);   // line c0 (NEW LINE
 sem_wait(&full);   // line c1
int tmp = get();   // line c2
 sem_post(&empty);   // line c3
 sem_post(&mutex);   // line c4 (NEW LINE
 printf("%d\n", tmp);
 }
}
```

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

Producer Consumer Problem using Semaphores Adding mutual exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
int i;
for (i = 0; i < loops; i++) {
sem_wait(&mutex);   // line p0 (NEW LINE)
sem_wait(&empty);   // line p1
 put(i);   // line p2
sem_post(&full);   // line p3
sem_post(&mutex);   // line p4 (NEW LINE)
}
}
```

```
void *consumer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&mutex);   // line c0 (NEW LINE)
 sem_wait(&full);   // line c1
int tmp = get();   // line c2
 sem_post(&empty);   // line c3
 sem_post(&mutex);   // line c4 (NEW LINE)
 printf("%d\n", tmp);
 }
}
```

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

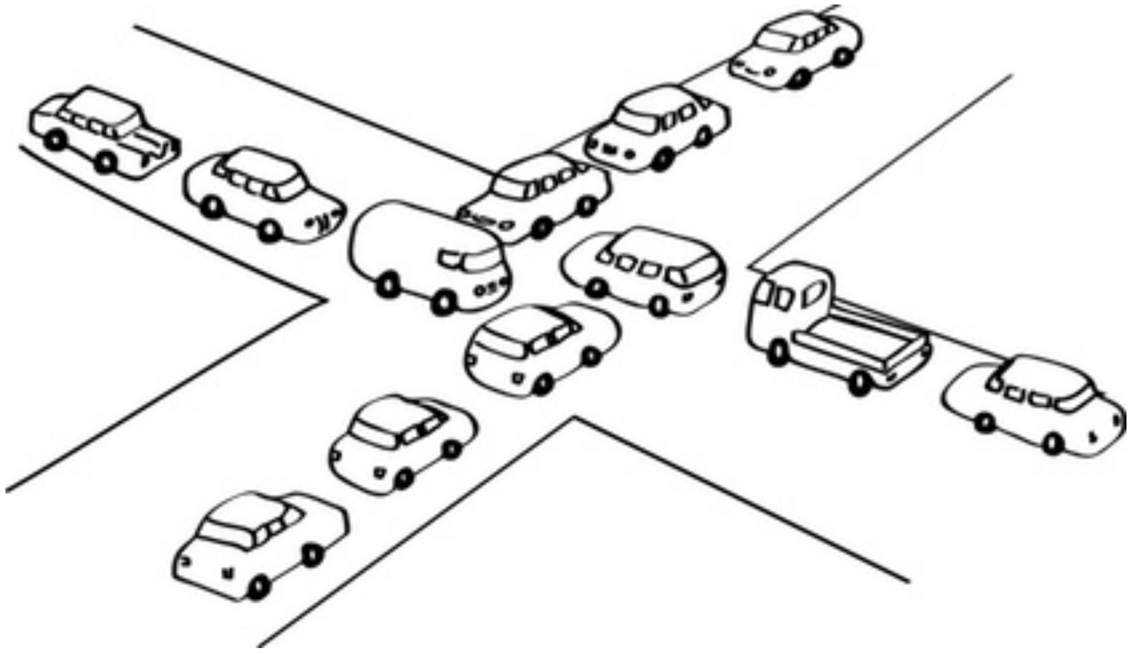
Producer Consumer Problem using Semaphores Adding mutual exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line p0 (NEW LINE)
  sem_wait(&empty);   // line p1
   put(i);   // line p2
   sem_post(&full);   // line p3
   sem_post(&mutex);   // line p4 (NEW LINE)
   }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line c0 (NEW LINE)
  sem_wait(&full);   // line c1
 int tmp = get();   // line c2
  sem_post(&empty);   // line c3
  sem_post(&mutex);   // line c4 (NEW LINE)
  printf("%d\n", tmp);
  }
}
```

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



Producer Consumer Problem using Semaphores Deadlock

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
int i;
for (i = 0; i < loops; i++) {
sem_wait(&mutex);   // line p0 (NEW LINE)
sem_wait(&empty);   // line p1
  put(i);   // line p2
 sem_post(&full);   // line p3
 sem_post(&mutex);   // line p4 (NEW LINE)
 }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line c0 (NEW LINE)
  sem_wait(&full);   // line c1
 int tmp = get();   // line c2
  sem_post(&empty);   // line c3
  sem_post(&mutex);   // line c4 (NEW LINE)
  printf("%d\n", tmp);
  }
}
```

Producer Consumer Problem using Semaphores

Deadlock

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line p0 (NEW LINE)
  sem_wait(&empty);   // line p1
    put(i);   // line p2
   sem_post(&full);   // line p3
   sem_post(&mutex);   // line p4 (NEW LINE)
   }
 }
```

```
void *consumer(void *arg) {
   int i;
   for (i = 0; i < loops; i++) {
   sem_wait(&mutex);   // line c0 (NEW LINE)
   sem_wait(&full);   // line c1
  int tmp = get();   // line c2
   sem_post(&empty);   // line c3
   sem_post(&mutex);   // line c4 (NEW LINE)
   printf("%d\n", tmp);
   }
 }
```

Imagine two threads: one producer and one consumer.

Producer Consumer Problem using Semaphores Deadlock

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line p0 (NEW LINE)
  sem_wait(&empty);   // line p1
    put(i);   // line p2
   sem_post(&full);   // line p3
   sem_post(&mutex);   // line p4 (NEW LINE)
   }
 }
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line c0 (NEW LINE)
  sem_wait(&full);   // line c1
 int tmp = get();   // line c2
  sem_post(&empty);   // line c3
  sem_post(&mutex);   // line c4 (NEW LINE)
  printf("%d\n", tmp);
  }
}
```

Imagine two threads: one producer and one consumer.

- The consumer acquire the mutex (line c0).

Producer Consumer Problem using Semaphores Deadlock

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
int i;
for (i = 0; i < loops; i++) {
sem_wait(&mutex);   // line p0 (NEW LINE)
sem_wait(&empty);   // line p1
  put(i);   // line p2
 sem_post(&full);   // line p3
 sem_post(&mutex);   // line p4 (NEW LINE)
 }
}
```

```
void *consumer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&mutex);   // line c0 (NEW LINE)
 sem_wait(&full);   // line c1
int tmp = get();   // line c2
 sem_post(&empty);   // line c3
 sem_post(&mutex);   // line c4 (NEW LINE)
 printf("%d\n", tmp);
 }
}
```

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

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
int i;
for (i = 0; i < loops; i++) {
sem_wait(&mutex);   // line p0 (NEW LINE)
sem_wait(&empty);   // line p1
 put(i);   // line p2
sem_post(&full);   // line p3
sem_post(&mutex);   // line p4 (NEW LINE)
}
}
```

```
void *consumer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&mutex);   // line c0 (NEW LINE)
 sem_wait(&full);   // line c1
int tmp = get();   // line c2
 sem_post(&empty);   // line c3
 sem_post(&mutex);   // line c4 (NEW LINE)
 printf("%d\n", tmp);
 }
}
```

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

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
int i;
for (i = 0; i < loops; i++) {
sem_wait(&mutex);   // line p0 (NEW LINE)
sem_wait(&empty);   // line p1
 put(i);   // line p2
sem_post(&full);   // line p3
sem_post(&mutex);   // line p4 (NEW LINE)
}
}
```

```
void *consumer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&mutex);   // line c0 (NEW LINE)
 sem_wait(&full);   // line c1
int tmp = get();   // line c2
 sem_post(&empty);   // line c3
 sem_post(&mutex);   // line c4 (NEW LINE)
 printf("%d\n", tmp);
 }
}
```

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

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);   // line p0 (NEW LINE)
    sem_wait(&empty);   // line p1
     put(i);   // line p2
    sem_post(&full);   // line p3
    sem_post(&mutex);   // line p4 (NEW LINE)
  }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);   // line c0 (NEW LINE)
    sem_wait(&full);   // line c1
   int tmp = get();   // line c2
    sem_post(&empty);   // line c3
    sem_post(&mutex);   // line c4 (NEW LINE)
    printf("%d\n", tmp);
  }
}
```

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 <u>still holds the mutex</u>!
- The producer calls sem_wait() on the binary mutex semaphore (line p0).

Producer Consumer Problem using Semaphores Deadlock

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line p0 (NEW LINE)
  sem_wait(&empty);   // line p1
   put(i);   // line p2
   sem_post(&full);   // line p3
   sem_post(&mutex);   // line p4 (NEW LINE)
   }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&mutex);   // line c0 (NEW LINE)
  sem_wait(&full);   // line c1
 int tmp = get();   // line c2
  sem_post(&empty);   // line c3
  sem_post(&mutex);   // line c4 (NEW LINE)
  printf("%d\n", tmp);
  }
}
```

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

```
sem_t empty;
sem_t full;
sem_t mutex;
```

```
void *producer(void *arg) {
int i;
for (i = 0; i < loops; i++) {
sem_wait(&mutex);   // line p0 (NEW LINE)
sem_wait(&empty);   // line p1
 put(i);   // line p2
 sem_post(&full);   // line p3
 sem_post(&mutex);   // line p4 (NEW LINE)
 }
}
```

```
void *consumer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&mutex);   // line c0 (NEW LINE)
 sem_wait(&full);   // line c1
int tmp = get();   // line c2
 sem_post(&empty);   // line c3
 sem_post(&mutex);   // line c4 (NEW LINE)
 printf("%d\n", tmp);
 }
}
```

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.

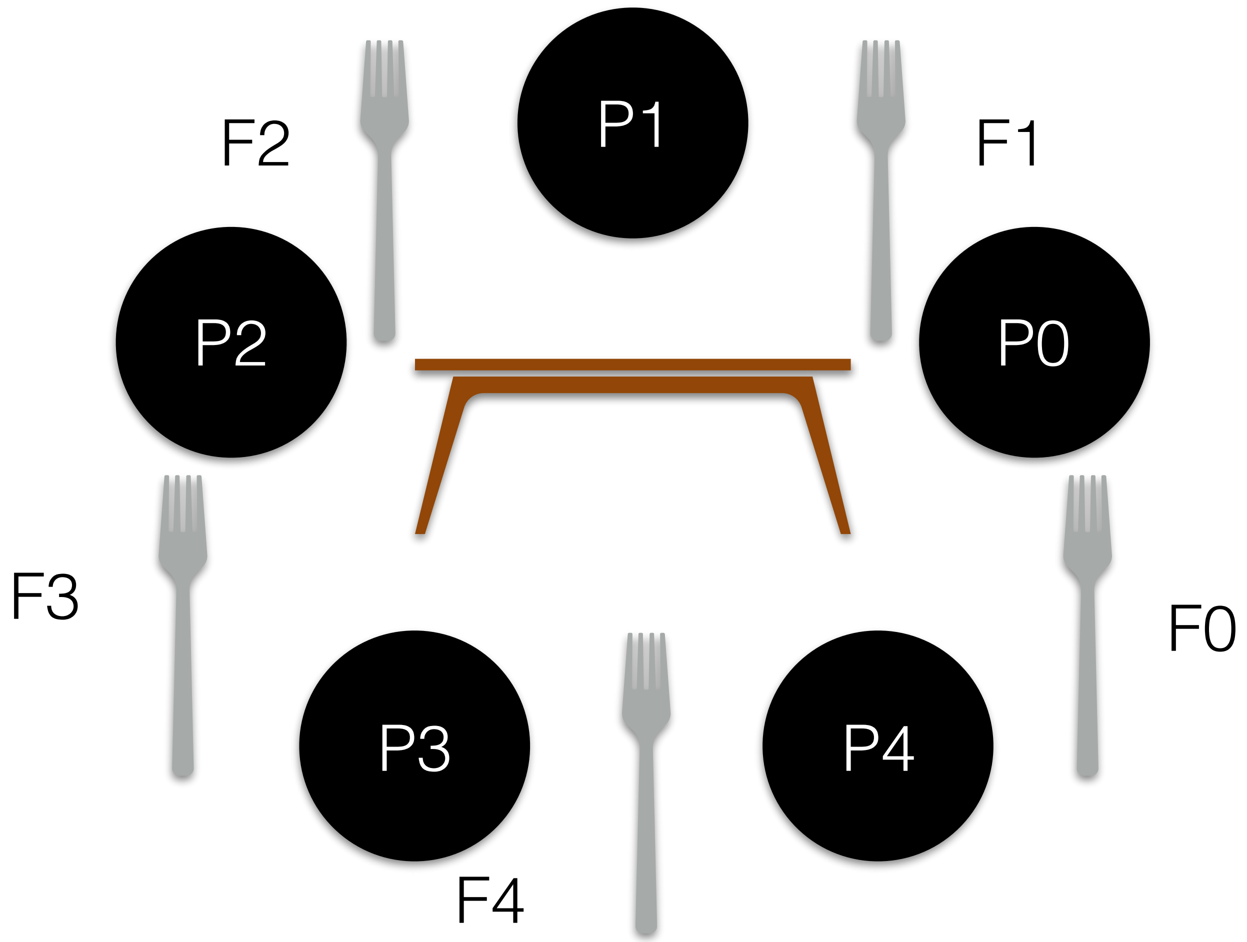
Producer Consumer Problem using Semaphores Correct Solution

```
sem_t empty;
sem_t full;
sem_t mutex;
```

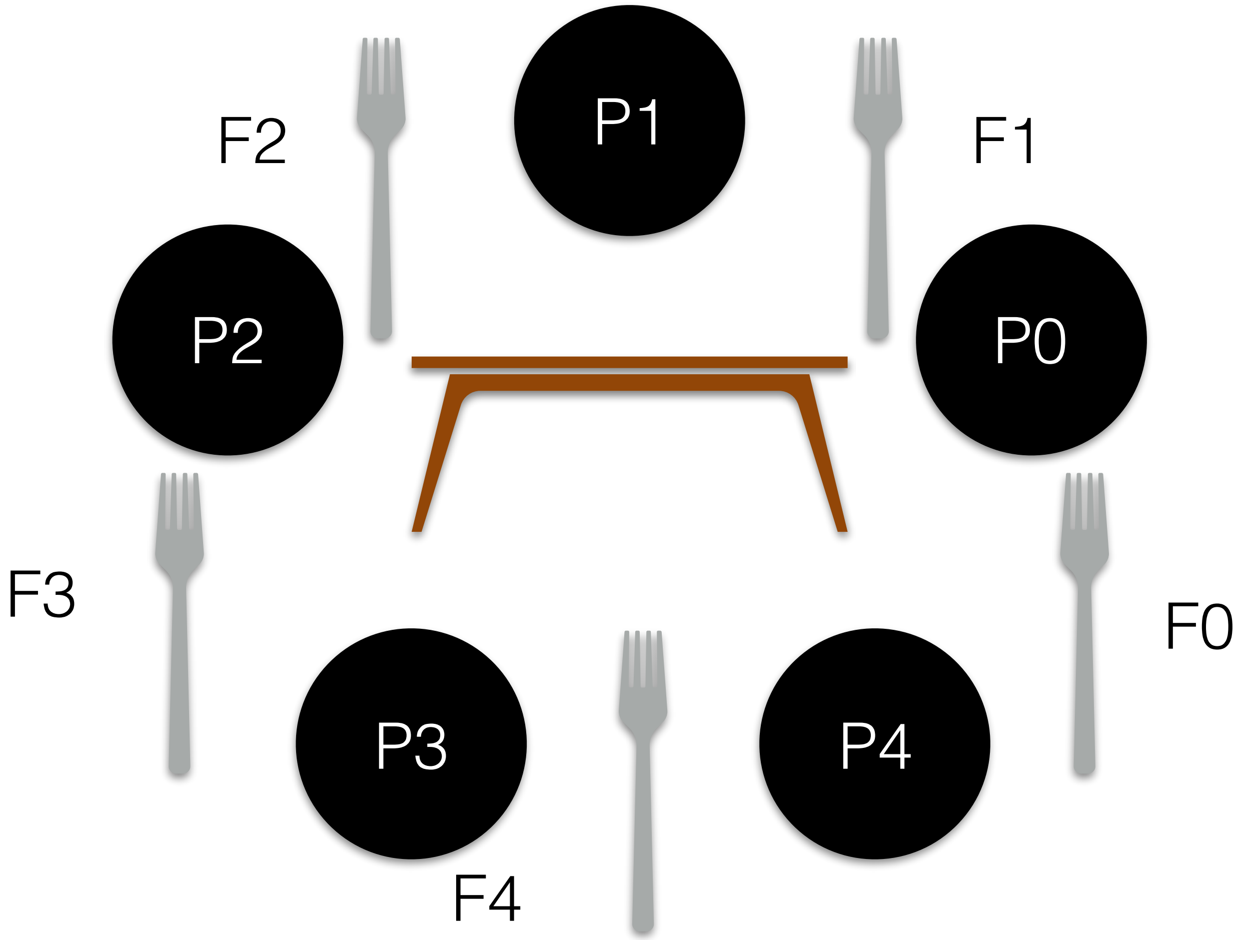
```
void *producer(void *arg) {
  int i;
 for (i = 0; i < loops; i++) {
  sem_wait(&empty);   // line p1
  sem_wait(&mutex);
    put(i);   // line p2
    sem_post(&mutex);
    sem_post(&full);   // line p3
    }
  }

```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
  sem_wait(&full);   // line c1
  sem_wait(&mutex);
  int tmp = get();   // line c2
  sem_post(&mutex);   //
sem_post(&empty);   // line c3
printf(“%d\n”, tmp);
}
}
```

Dining Philosopher's Problem

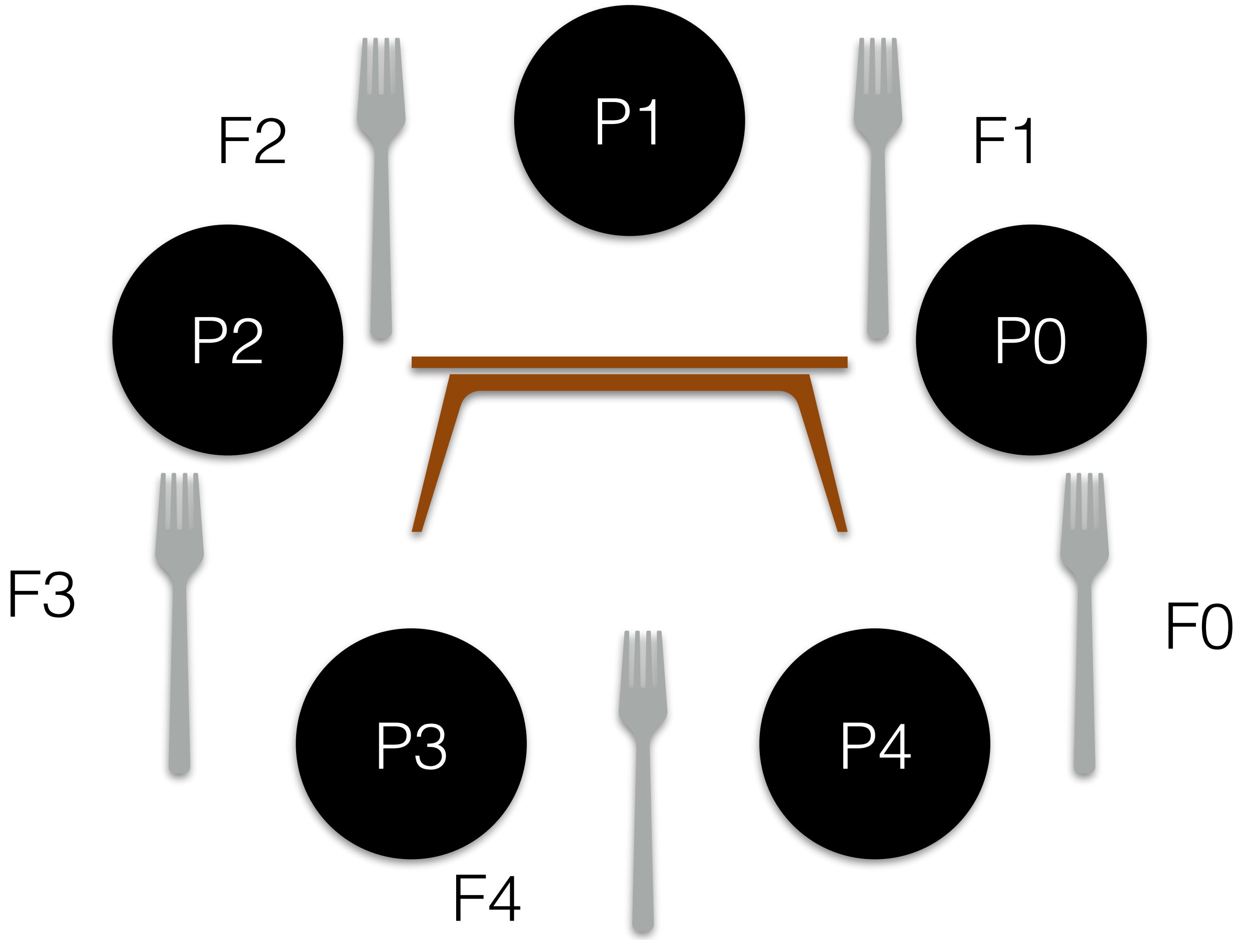


Dining Philosopher's Problem



```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

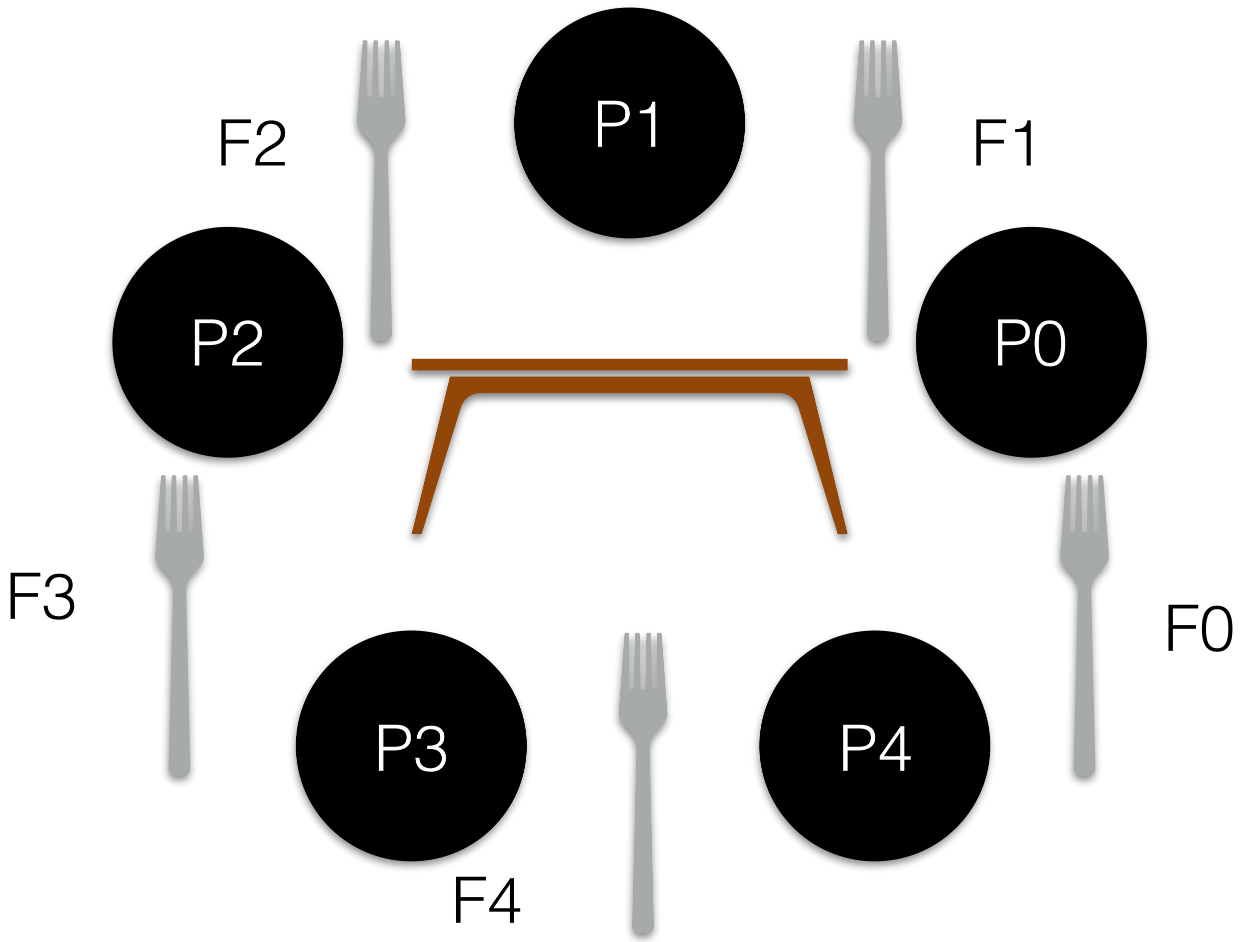
Dining Philosopher's Problem



```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

```
void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

Dining Philosopher’s Problem

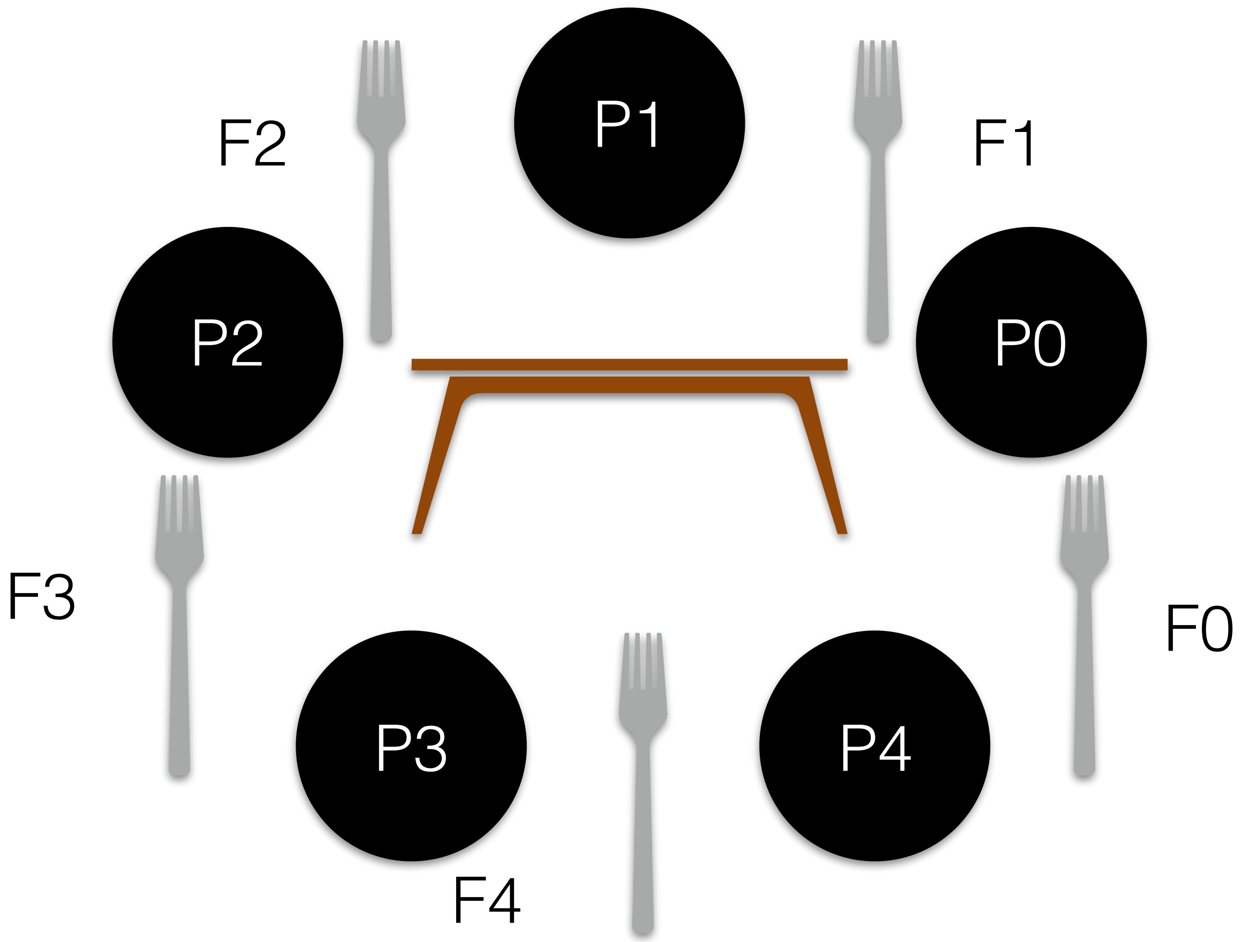


```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

```
void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

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

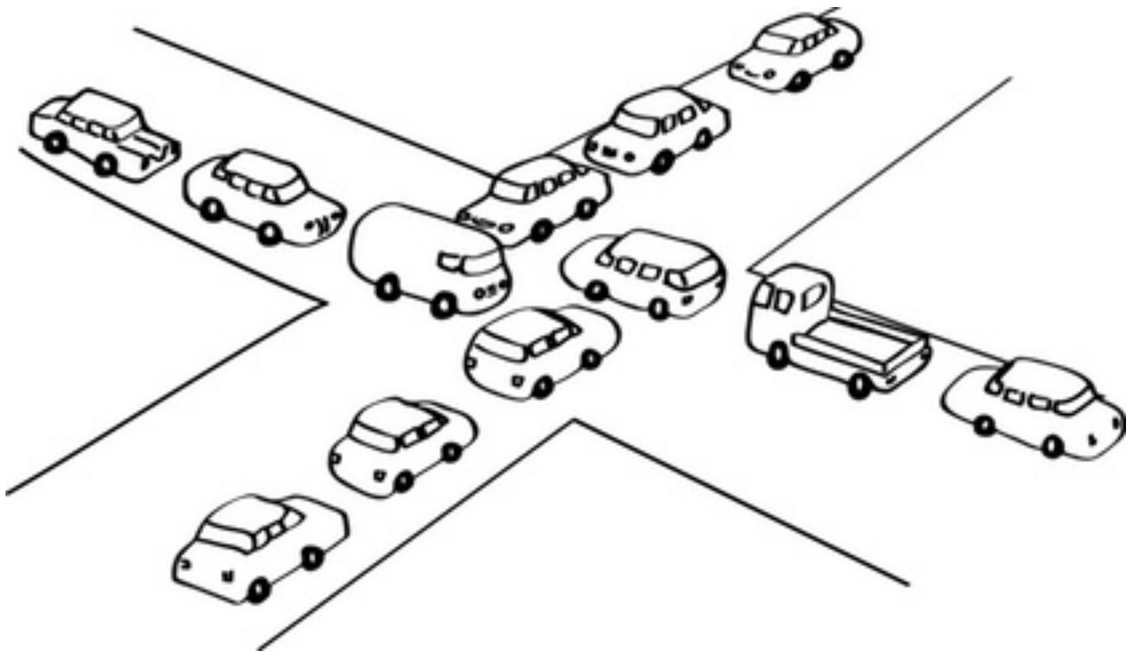
Dining Philosopher's Problem



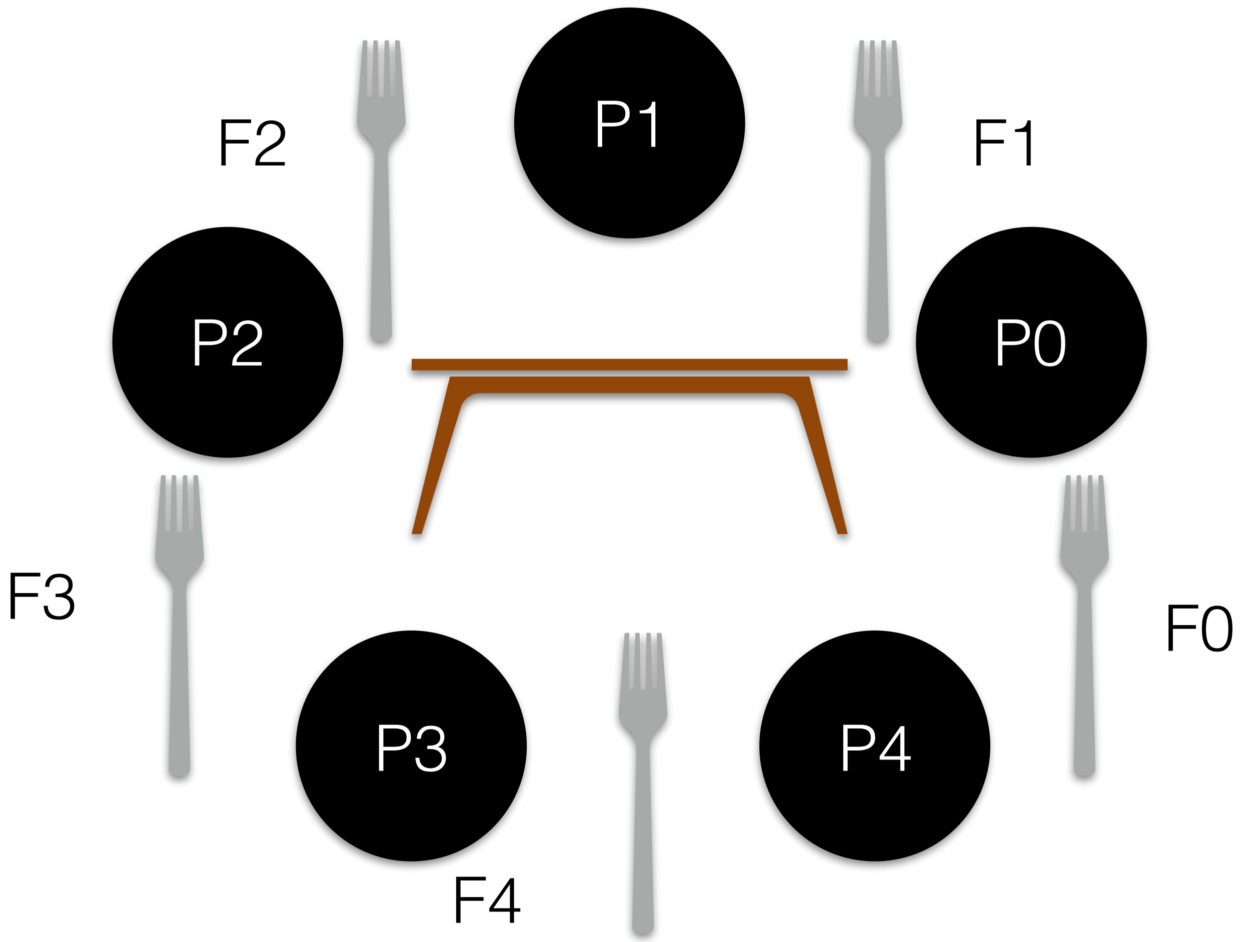
```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

```
void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

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



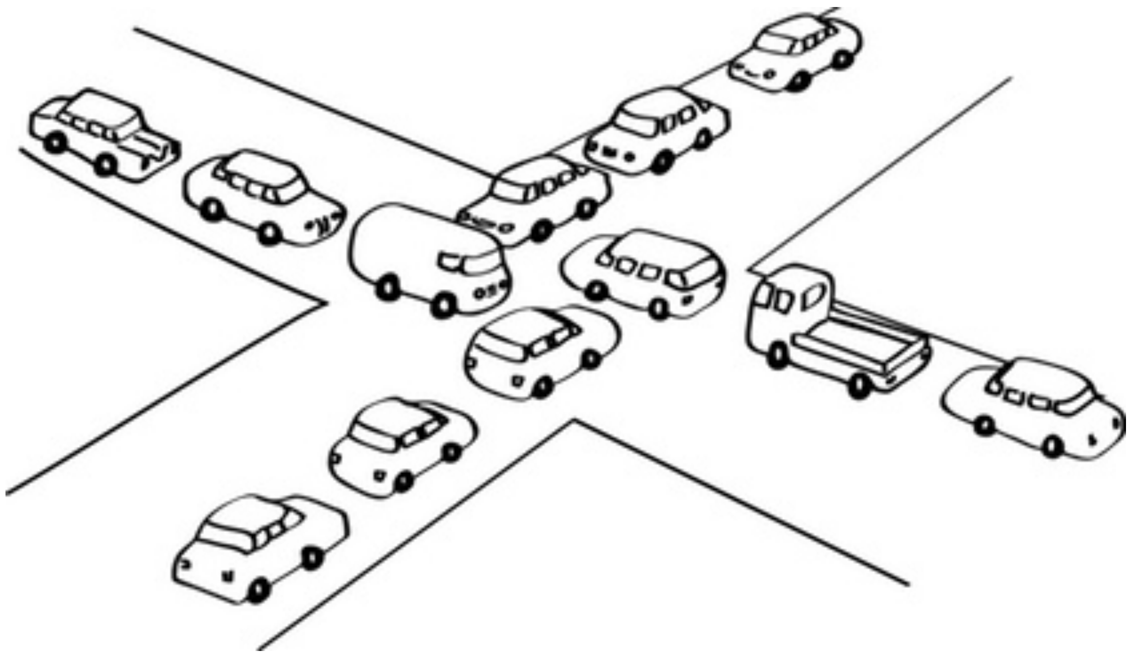
Dining Philosopher's Problem



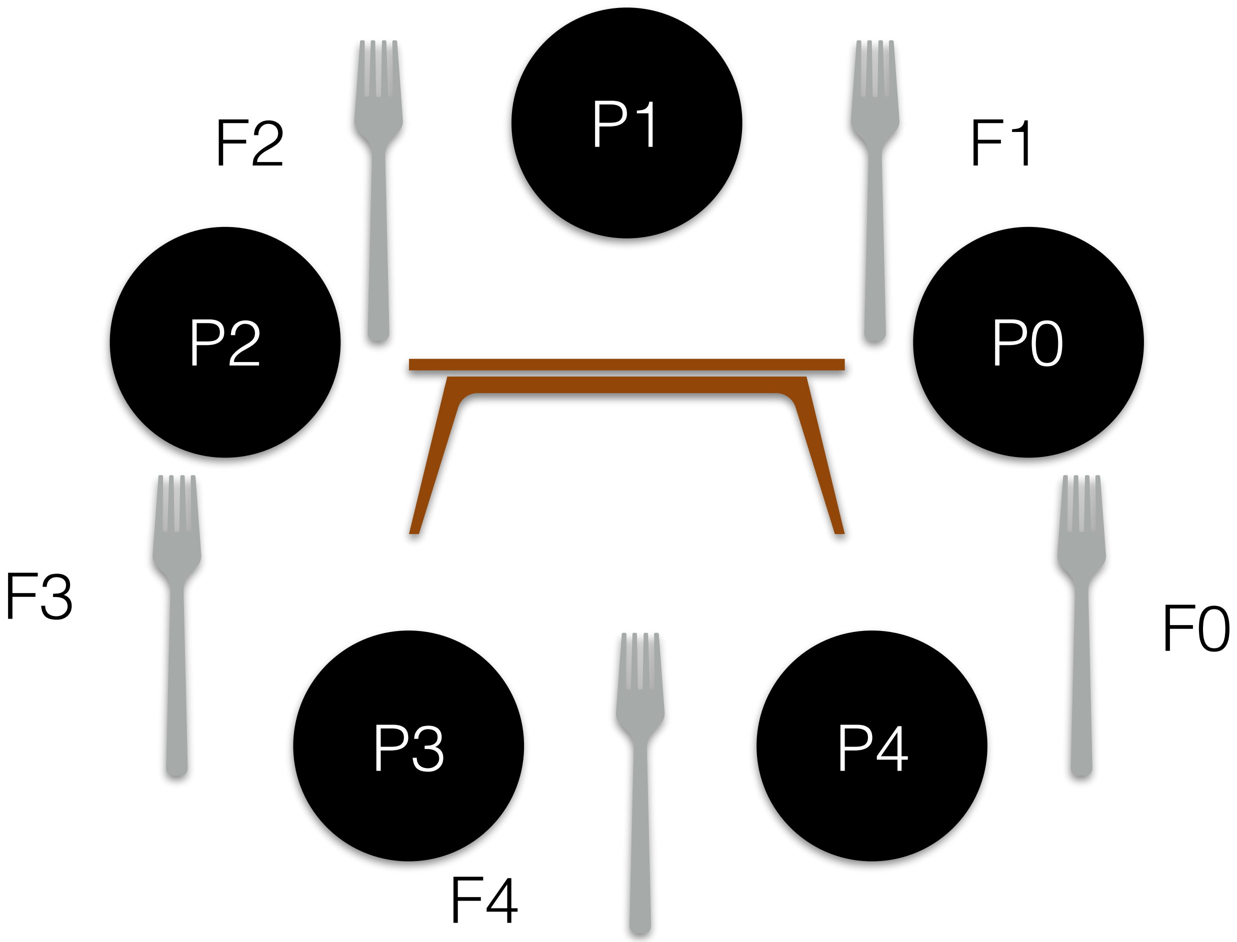
```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

```
void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

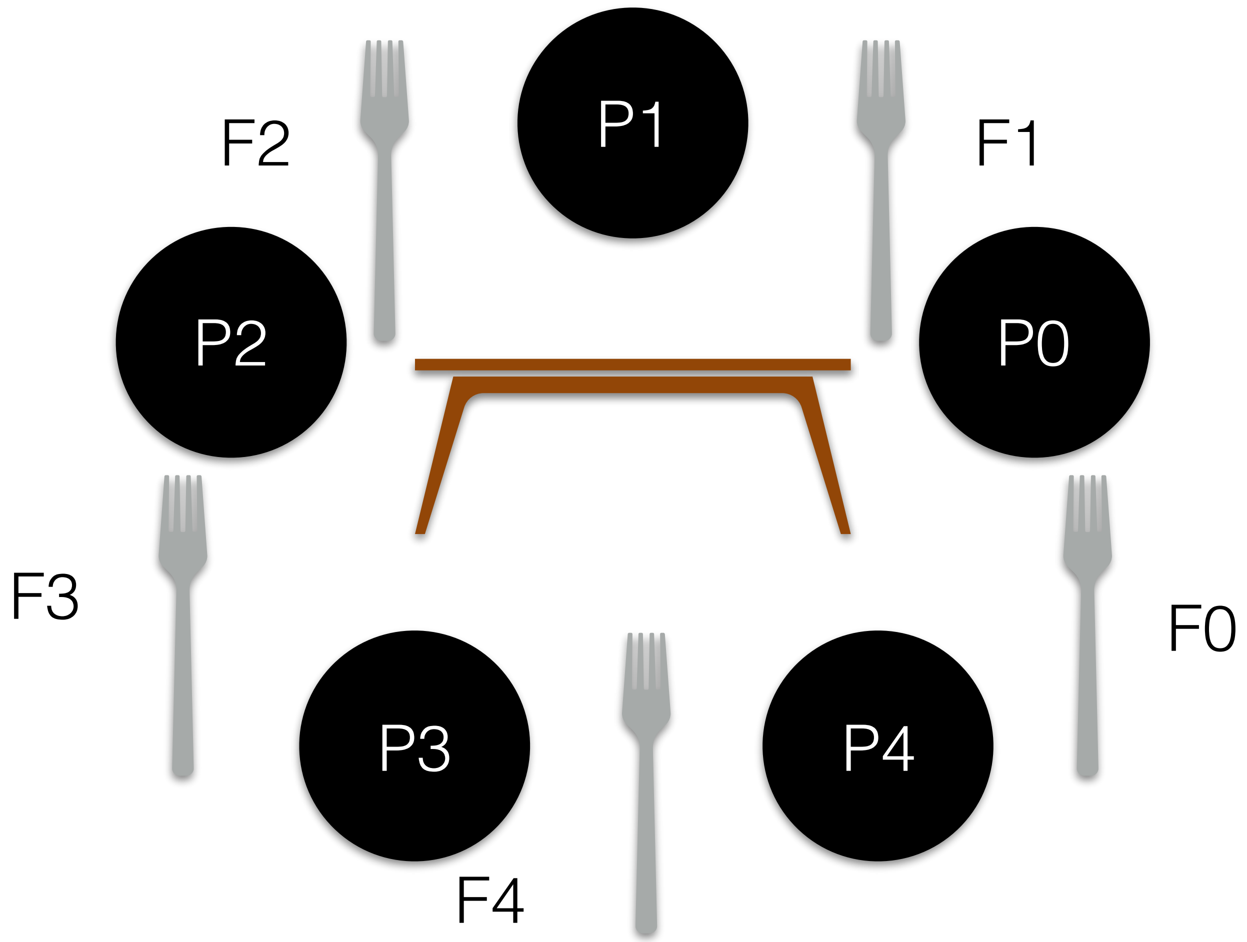
- P0 picks F0; P1 picks F1; …, P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?



Dining Philosopher’s Problem

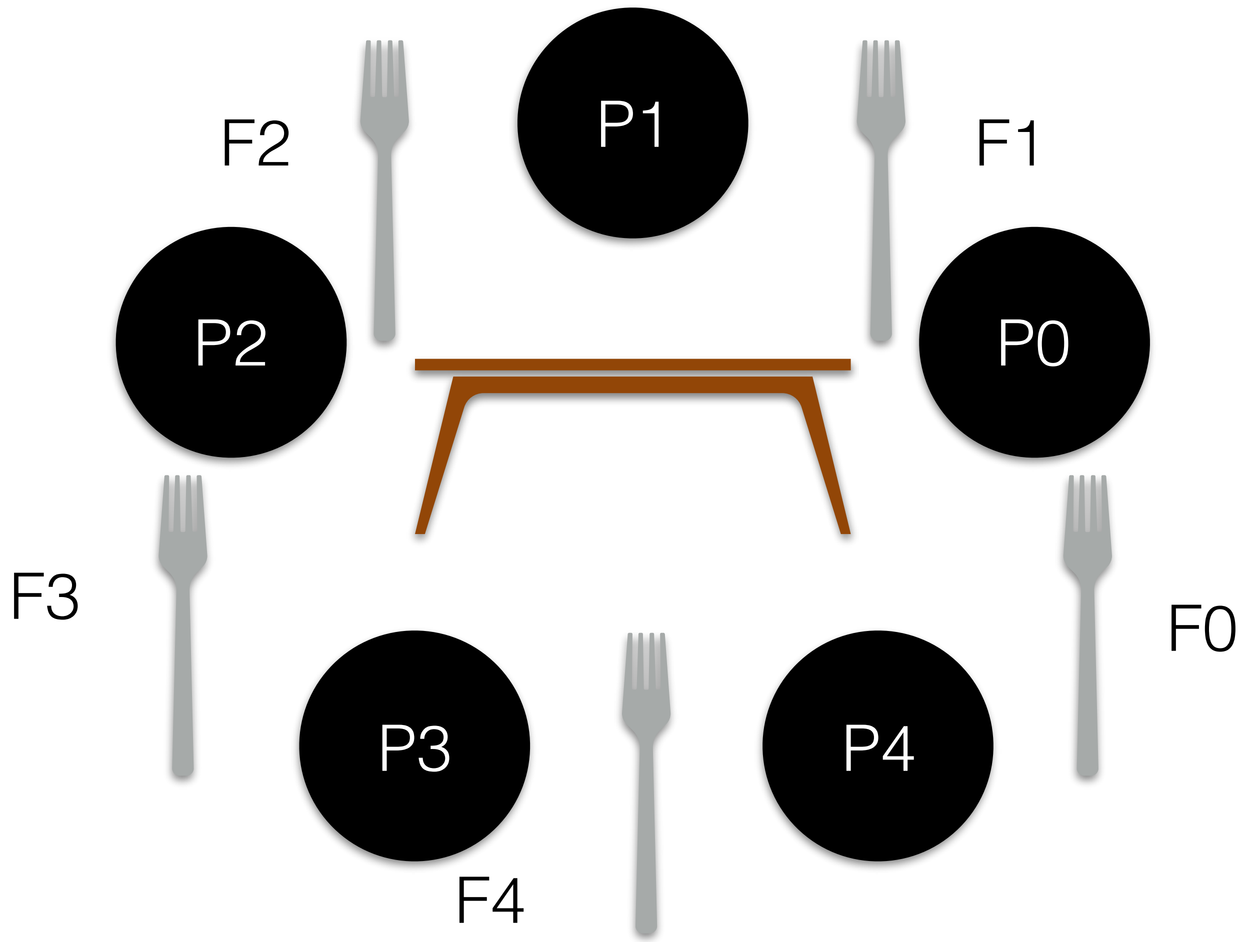


Dining Philosopher’s Problem



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

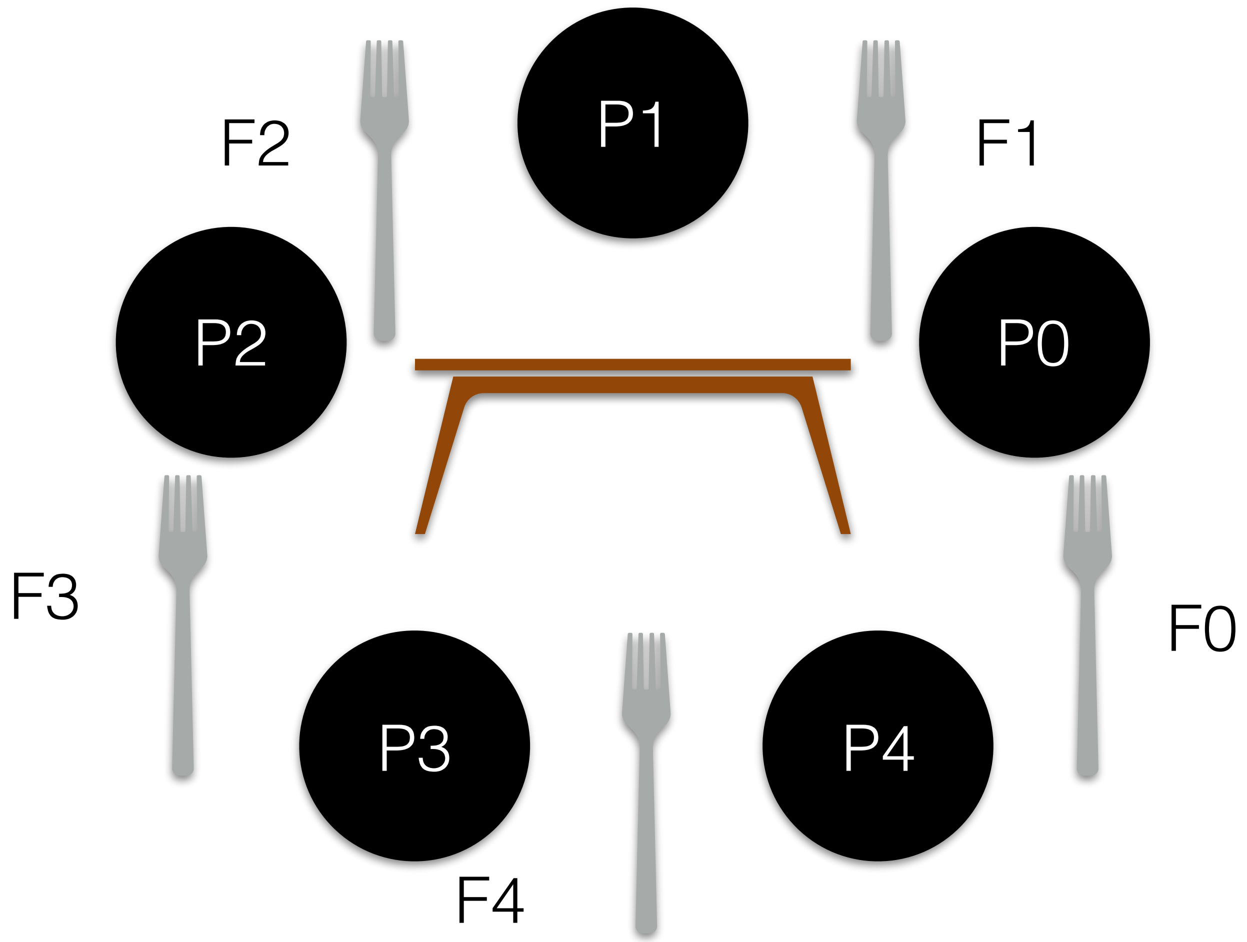
Dining Philosopher’s Problem



- 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

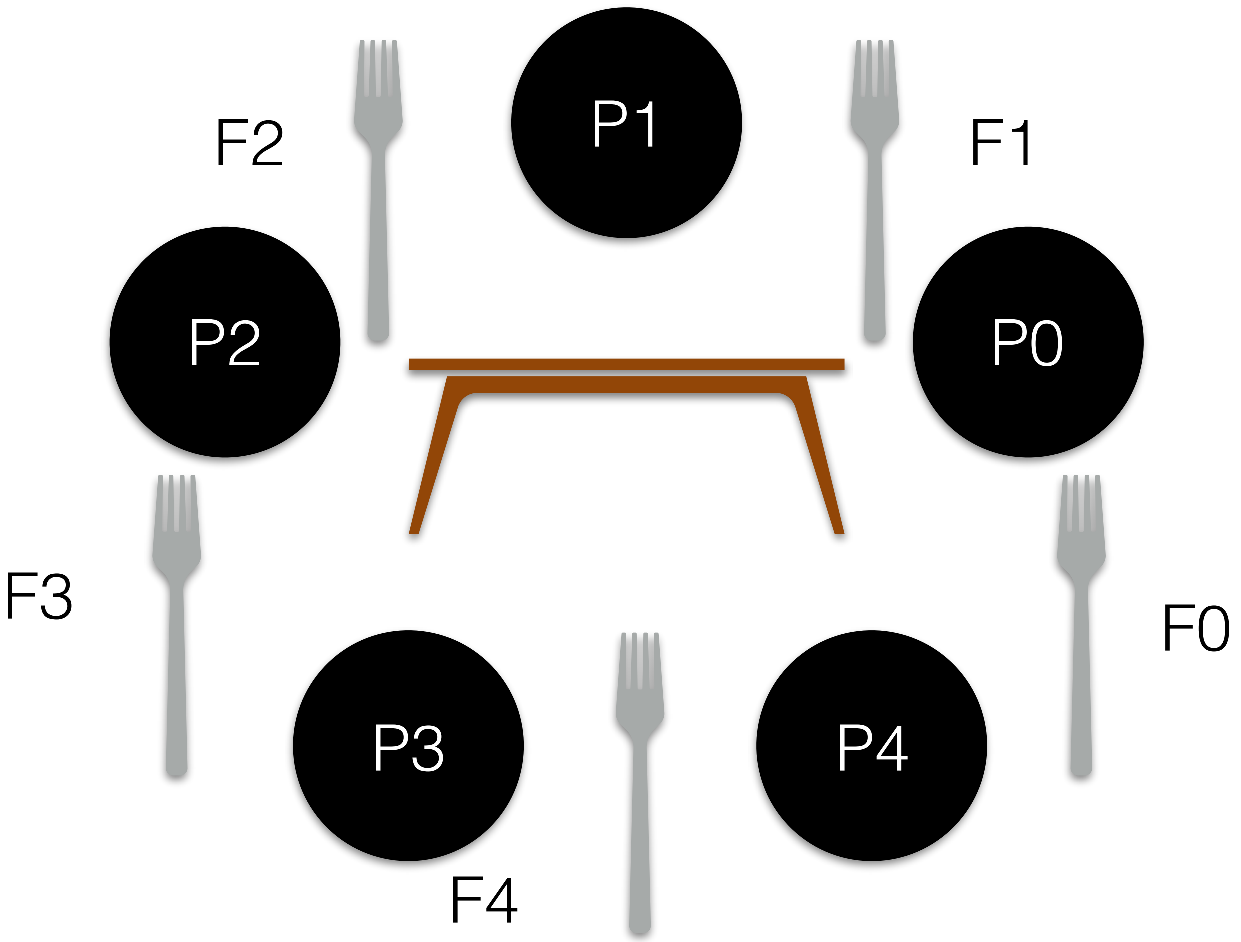


- If P ==4:

- 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?

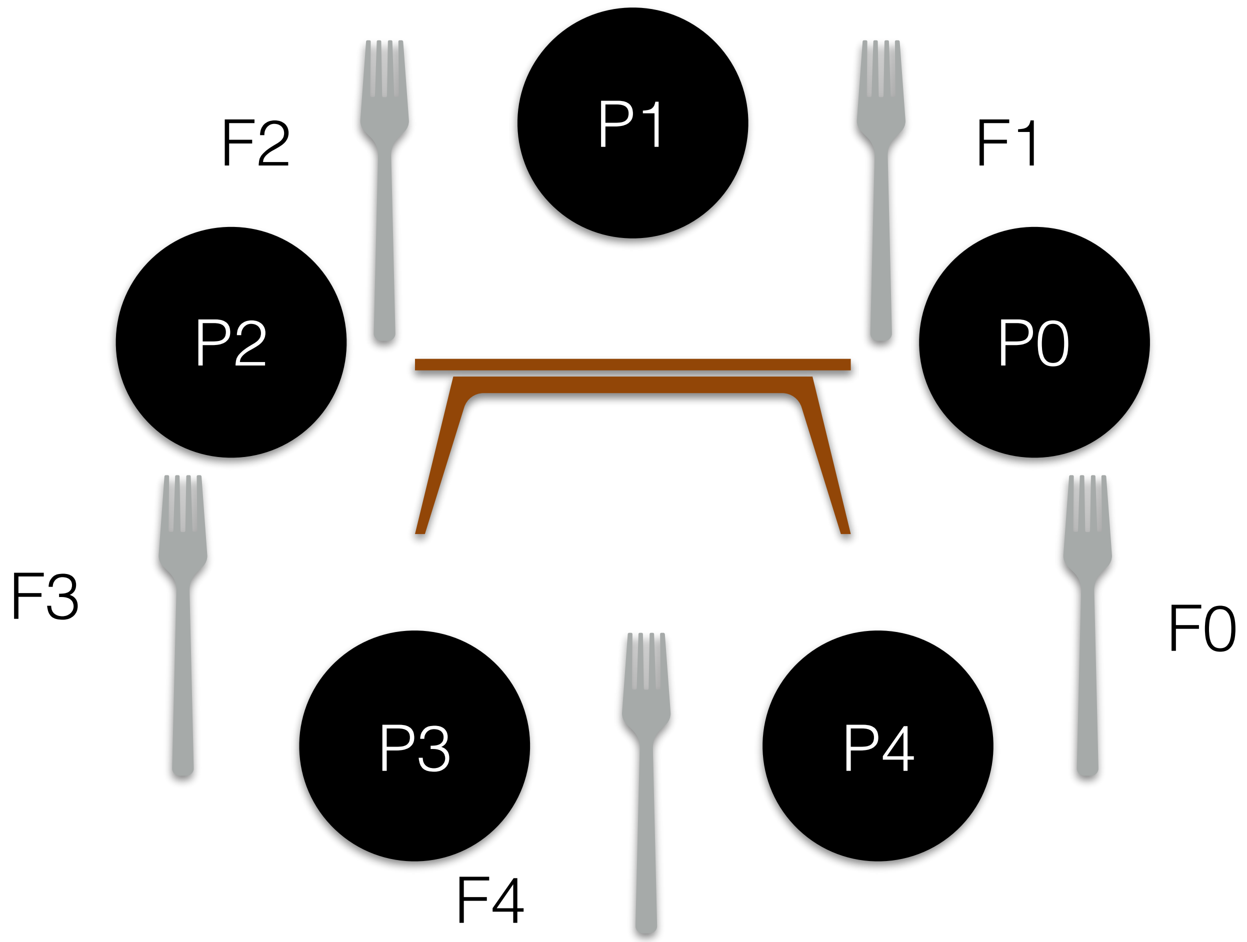
Dining Philosopher’s Problem



- 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

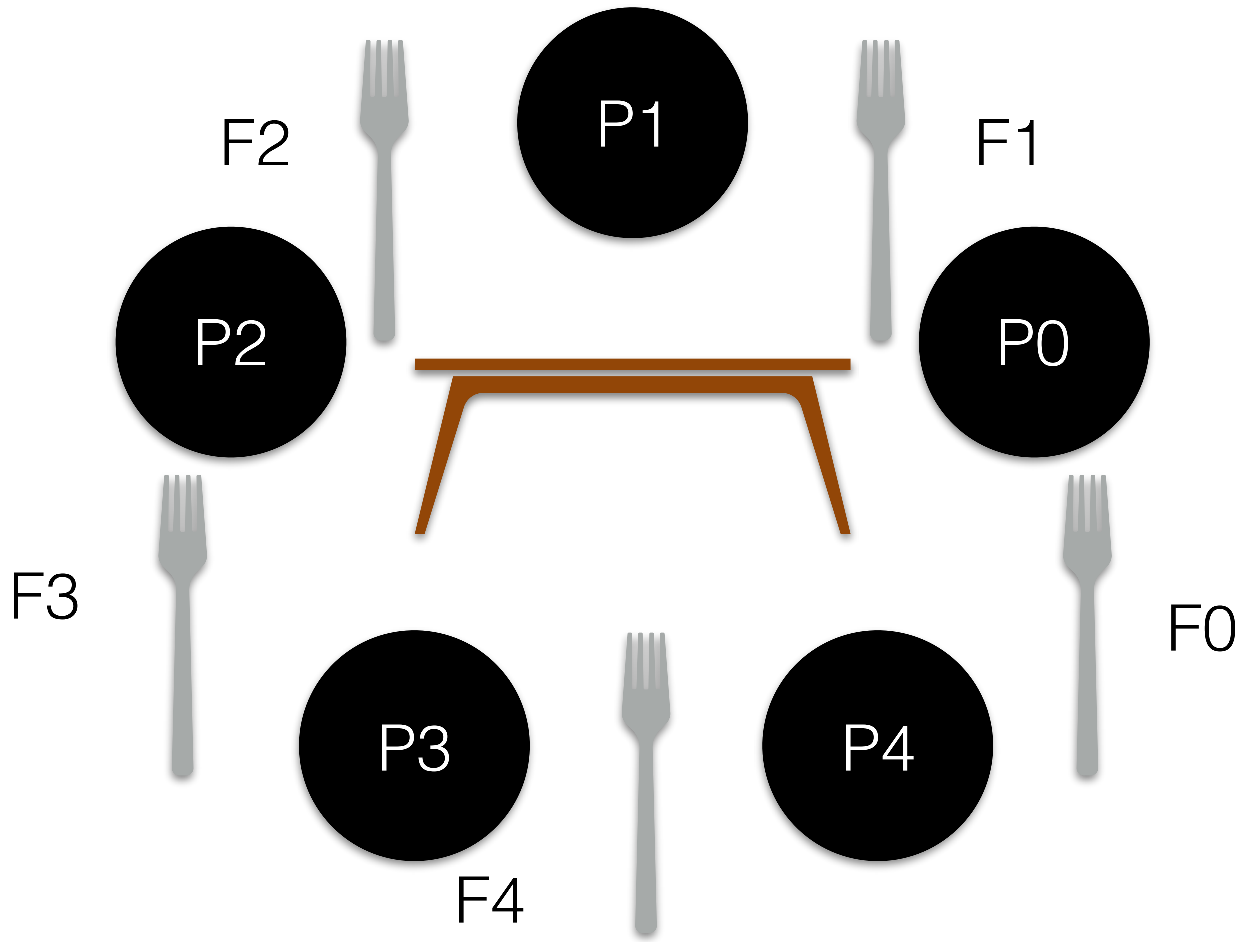


- If P ==4:
  - Wait on Right
  - Want on Left

- 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?

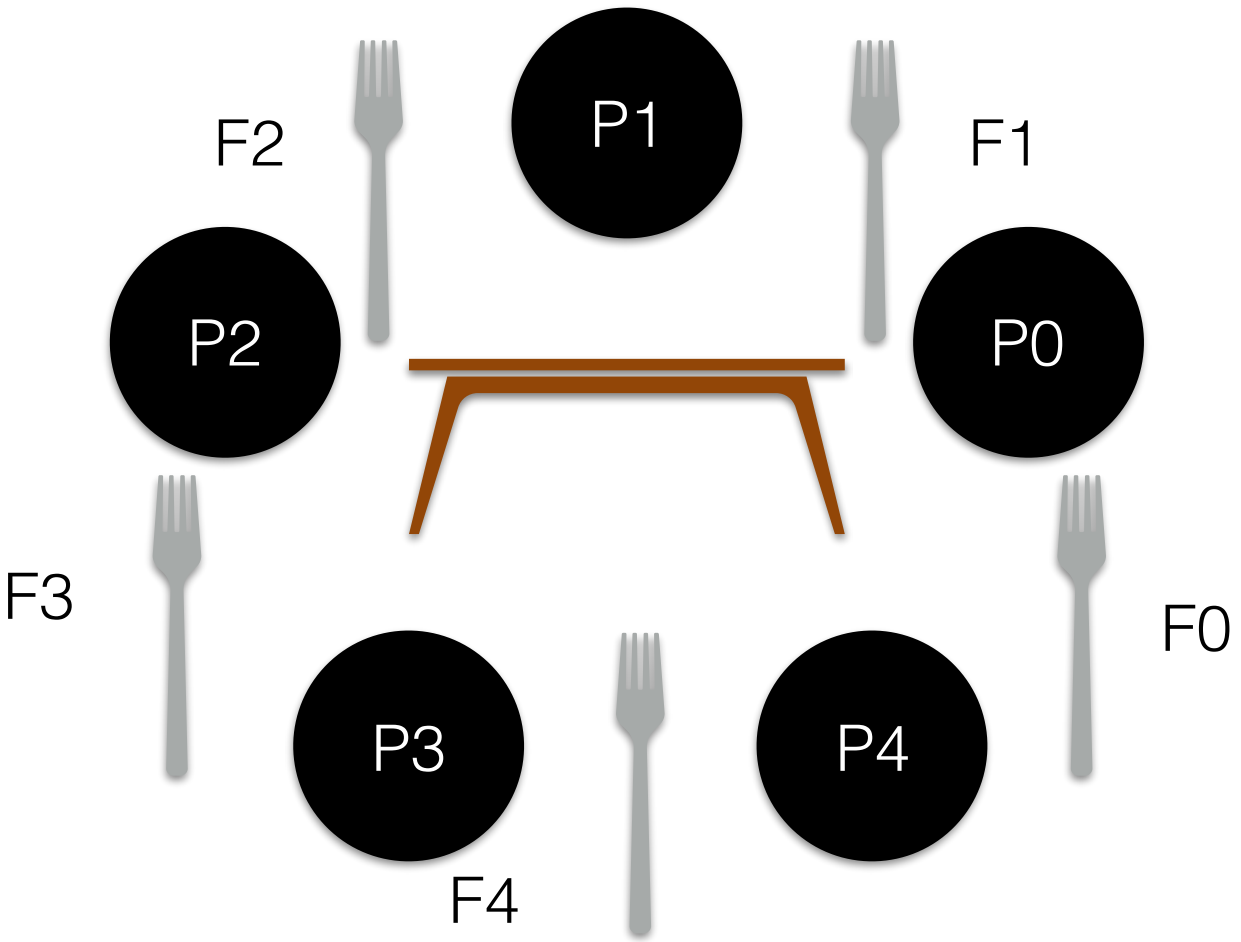
Dining Philosopher's Problem



- 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?

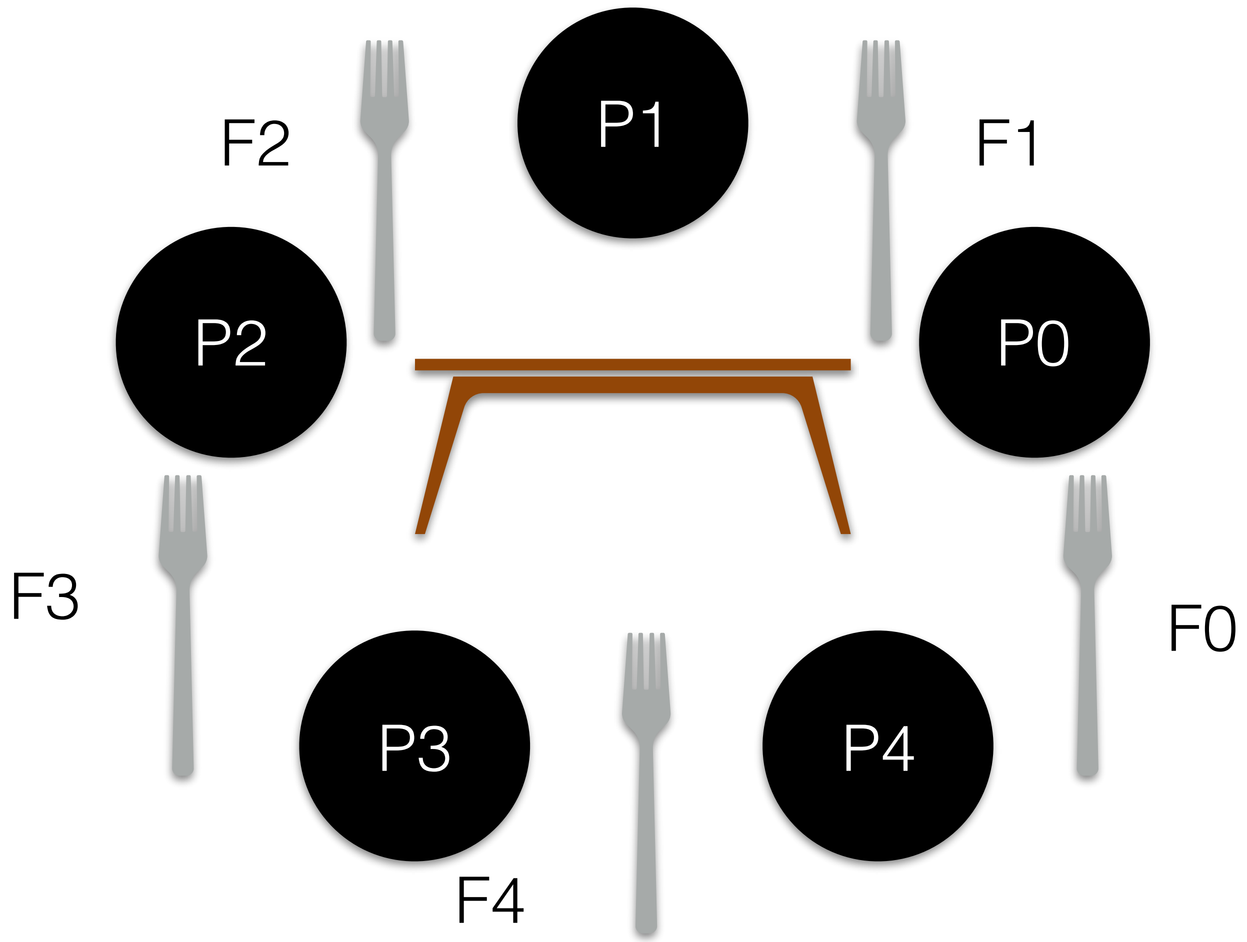
Dining Philosopher’s Problem



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

```
#include <semaphore.h>
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

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

Our implementation

```
typedef struct __Zem_t {
  int value;
  pthread_cond_t cond;
  pthread_mutex_t lock;
} Zem_t;

```

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

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

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

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

Our implementation

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

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

Our implementation

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

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

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

Atomic operation

Our implementation

Semaphores Implementation

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

API

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

Atomic operation

Our implementation

```
void Zem_wait(Zem_t *s) {

Mutex_lock(&s->lock);
while (s->value <= 0)
  Cond_wait(&s->cond, &s->lock);
s->value--;
Mutex_unlock(&s->lock);
}
```

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	Our implementation

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

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

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

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

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

Atomic operation

Our implementation

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

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.

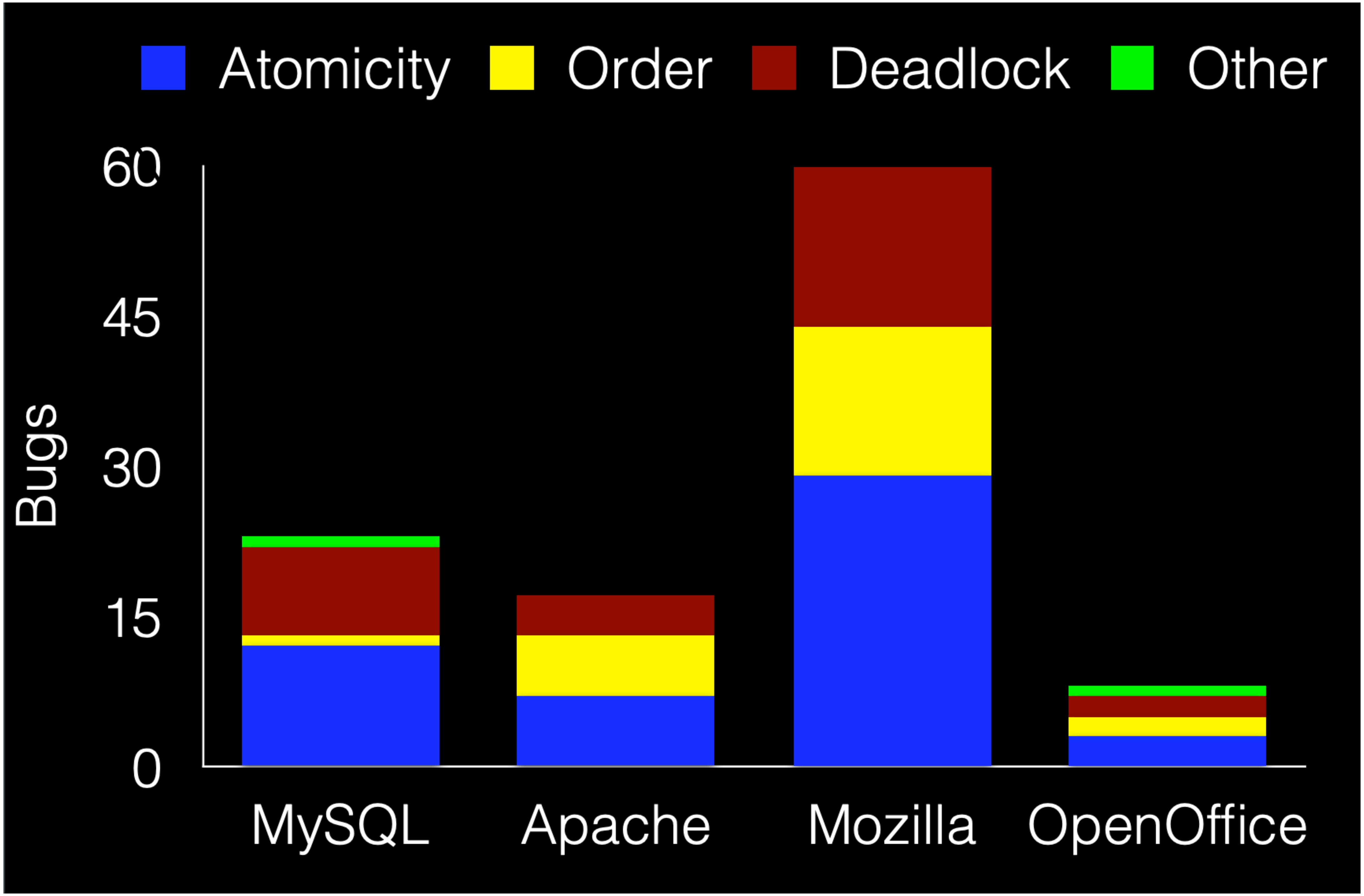
- Insert:
  - Changes state of the list
  - Traditional critical section needed
- Lookup
  - Read the data structure, no modification
  - As long as we guarantee no insertion, multiple readers can read concurrently

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 …

```
  Thread1::
  if(thd->proc_info){
    …
    fputs(thd->proc_info , …);
…
  }
```

Concurrency Bugs — Atomicity

MySQL bug …

```
Thread1::
if(thd->proc_info){
  …
  fputs(thd->proc_info , …);
…
}
```

```
Thread2::
thd->proc_info = NULL;
```

Concurrency Bugs — Atomicity

MySQL bug …

```
Thread1::
if(thd->proc_info){
  …
  fputs(thd->proc_info , …);
…
}
```

```
Thread2::
thd->proc_info = NULL;
```

- Is this problematic?

Concurrency Bugs — Atomicity

MySQL bug …

```
Thread1::
if(thd->proc_info){
  …
  fputs(thd->proc_info , …);
…
}
```

```
Thread2::
thd->proc_info = NULL;
```

- Is this problematic?
    - Yes, else we wouldn’t be discussing …

Concurrency Bugs — Atomicity

MySQL bug …

```
Thread1::
if(thd->proc_info){
  …
  fputs(thd->proc_info , …);
…
}
```

```
Thread2::
thd->proc_info = NULL;
```

- Is this problematic?
    - Yes, else we wouldn’t be discussing …
    - How?

Concurrency Bugs — Atomicity

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1::
pthread_mutex_lock(&lock);
if(thd->proc_info){
  ...
  fputs(thd->proc_info , ...);
  ...
}
pthread_mutex_unlock(&lock);
```

```
Thread2::
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Concurrency Bugs — Atomicity

Simple Solution

```
pthread_mutex_t lock =
PTHREAD_MUTEX_INITIALIZER;

Thread1::
pthread_mutex_lock(&lock);
if(thd->proc_info){
  ...
  fputs(thd->proc_info , ...);
  ...
}
pthread_mutex_unlock(&lock);
```

```
Thread2::
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Concurrency Bugs — Order Violation

```
Thread1::
void init(){
  mThread = PR_CreateThread(mMain, ...);
}

```

```
Thread2::
void mMain(...){
  mState = mThread->State
}
```

Concurrency Bugs — Order Violation

Mozilla bug …

```
Thread1::
void init(){
  mThread = PR_CreateThread(mMain, ...);
}

```

```
Thread2::
void mMain(...){
  mState = mThread->State
}
```

Concurrency Bugs — Order Violation

Mozilla bug …

```
Thread1::
void init(){
  mThread =
PR_CreateThread(mMain, ...);
}

```

```
Thread2::
void mMain(...){
  mState = mThread->State
}
```

- Is this problematic?

Concurrency Bugs — Order Violation

Mozilla bug …

```
Thread1::
void init(){
  mThread =
PR_CreateThread(mMain, ...);
}

```

```
Thread2::
void mMain(...){
  mState = mThread->State
}
```

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

Concurrency Bugs — Order Violation

Mozilla bug …

```
Thread1::
void init(){
  mThread =
PR_CreateThread(mMain, ...);
}

```

```
Thread2::
void mMain(...){
  mState = mThread->State
}
```

- Is this problematic?
  - Yes, else we wouldn’t be discussing …
  - How?

Concurrency Bugs — Order Violation

Concurrency Bugs — Order Violation

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

Concurrency Bugs — Order Violation

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

Thread 1::

```
void init(){
  ...
  mThread = PR_CreateThread(mMain,...);

  // signal that the thread has been created.
  pthread_mutex_lock(&mtLock);
  mtInit = 1;
  pthread_cond_signal(&mtCond);
  pthread_mutex_unlock(&mtLock);
  ...
}
```

Concurrency Bugs — Order Violation

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

```
Thread 1::
void init(){
  ...
  mThread = PR_CreateThread(mMain,...);

  // signal that the thread has been created.
  pthread_mutex_lock(&mtLock);
  mtInit = 1;
  pthread_cond_signal(&mtCond);
  pthread_mutex_unlock(&mtLock);
  ...
}
```

```
Thread2::
void mMain(...){

// wait for the thread to be initialized
...
  pthread_mutex_lock(&mtLock);
  while(mtInit == 0)
  pthread_cond_wait(&mtCond, &mtLock);
  pthread_mutex_unlock(&mtLock);
  mState = mThread->State;
}
```

Concurrency Bugs — Deadlock

Concurrency Bugs — Deadlock

Thread 1

```
Lock(L1);
Lock(L2);
```

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1

Concurrency Bugs — Deadlock

<u>**Thread 1**</u>	<u>**Thread 2**</u>
Lock(L1);	Lock(L2);
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	**Thread 2**
Lock(L1);	Lock(L2);
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	**Thread 2**
Lock(L1);	Lock(L2);
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	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

- Thread T1 gets Lock L1
- Context Switch

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

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

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
Lock(L2);	Lock(L1);

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

Concurrency Bugs — Deadlock

Thread 1	**Thread 2**
Lock(L1);	Lock(L2);
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	**Thread 2**
Lock(L1);	Lock(L2);
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	**Thread 2**
Lock(L1);	Lock(L2);
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	**Thread 2**
Lock(L1);	Lock(L2);
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