Operating Systems

Lecture 24: Semaphores + Deadlocks

Nipun Batra
Oct 26, 2018

Exercise: Build a lock using semaphores

```
sem_t m;
sem_init(&m, 0, 1);

sem_wait(&m);
//critical section here
sem_post(&m);
```

Refresher Notes

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

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

join() using CVs

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
     Pthread_create(&c, NULL, child, NULL); // create child
     thread_join()
     printf("parent: end\n");
     return 0; }
```

join() using CVs

```
void thread_exit {
    mutex_lock(&m)
    Done  = 1
    cond_signal(&c)
    mutex_unlock(&m)
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
     Pthread_create(&c, NULL, child, NULL); // create child
     thread_join()
     printf("parent: end\n");
     return 0; }
```

join() using CVs

```
void thread_exit {
    mutex_lock(&m)
    Done  = 1
    cond_signal(&c)
    mutex_unlock(&m)
```

```
void thread_join {
    mutex_lock(&m)              //w
    while (done==0)             //x
        cond_wait(&c, &m)       //y
    mutex_unlock(&m) }          //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
     Pthread_create(&c, NULL, child, NULL); // create child
     thread_join()
     printf("parent: end\n");
     return 0; }
```

join() using semaphores

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    sem_init(&s, 0, X);
      Pthread_create(&c, NULL, child, NULL);
      thread_join()
      printf("parent: end\n");
      return 0; }
```

Refresher Notes

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

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

join() using semaphores

```
void thread_exit {

}
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    sem_init(&s, 0, X);
      Pthread_create(&c, NULL, child, NULL);
      thread_join()
      printf("parent: end\n");
      return 0; }
```

Refresher Notes

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

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

join() using semaphores

```
void thread_exit {

}
```

```
void thread_join {

}
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    sem_init(&s, 0, X);
      Pthread_create(&c, NULL, child, NULL);
      thread_join()
      printf("parent: end\n");
      return 0; }
```

Refresher Notes

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

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

join() using semaphores

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
    sem_wait(&s);
}
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    sem_init(&s, 0, 0);
     Pthread_create(&c, NULL, child, NULL);
     thread_join()
     printf("parent: end\n");
     return 0; }
```

Refresher Notes

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

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

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready

Thread Trace: Parent Waiting for Child Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running
0		sleeping	increment sem	Running

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	Interrupt; Switch→Parent	Ready

Thread Trace: Parent Waiting for Child
Parent starts before child

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	Interrupt; Switch→Parent	Ready
0	sem_wait() retruns	Running		Ready

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready

Thread Trace: Parent Waiting for Child Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready
1	call sem_wait()	Running		Ready

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready
0	(sem<0)→awake	Running		Ready

Thread Trace: Parent Waiting for Child
Child starts immediately executing

```
void thread_exit {
    sem_post(&s);
}
```

```
void thread_join {
sem_wait(&s)
    }
```

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready
0	(sem<0)→awake	Running		Ready
0	sem_wait() retruns	Running		Ready

Producer Consumer Problem using Semaphores

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
  buffer[fill] = value;   // line f1
  fill = (fill + 1) % MAX;   // line f2
}

int get() {
  int tmp = buffer[use];   // line g1
  use = (use + 1) % MAX;   // line g2
  return tmp;
}
```

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);     // line C1
  tmp = get();         // line C2
  sem_post(&empty);    // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty,0, ?);
    sem_init(&full, 0, ?);
    // ...
}
```

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);     // line C1
  tmp = get();         // line C2
  sem_post(&empty);    // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty,0, MAX);
    sem_init(&full, 0, 0);
    // ...
}
```

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);     // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty,0, MAX);
    sem_init(&full, 0, 0);
    // ...
}
```

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);     // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty,0, MAX);
    sem_init(&full, 0, 0);
    // ...
}
```

- For Max = 1, does it work for 1 consumer and 1 producer?

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);       // line C1
  tmp = get();           // line C2
  sem_post(&empty);  // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty,0, MAX);
    sem_init(&full, 0, 0);
    // ...
}
```

- For Max = 1, does it work for 1 consumer and 1 producer?
- For Max = 1, does it work for many consumers and producers?

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);     // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty,0, MAX);
    sem_init(&full, 0, 0);
    // ...
}
```

- For Max = 1, does it work for 1 consumer and 1 producer?
- For Max = 1, does it work for many consumers and producers?
- For Max = 20, does it work for multiple consumers and producers?

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
}

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);     // line C1
  tmp = get();         // line C2
  sem_post(&empty);    // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
void put(int value) {
  buffer[fill] = value;    // line f1
  fill = (fill + 1) % MAX;   // line f2
}
```

Producer Consumer Problem using Semaphores

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);     // line C3
  printf("%d\n", tmp);
  }
}
...
```

```c
void put(int value) {
  buffer[fill] = value;   // line f1
  fill = (fill + 1) % MAX;   // line f2
}
```

- Producer 1 (Pa) is on P2

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);     // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
void put(int value) {
  buffer[fill] = value;    // line f1
  fill = (fill + 1) % MAX;    // line f2
}
```

- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

 void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);    // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
void put(int value) {
  buffer[fill] = value;   // line f1
  fill = (fill + 1) % MAX;   // line f2
}
```

- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time
- Let’s assume fill is 10

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);  // line P1
 put(i);            // line P2
 sem_post(&full);   // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);     // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
void put(int value) {
  buffer[fill] = value;    // line f1
  fill = (fill + 1) % MAX;    // line f2
}
```

- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time
- Let's assume fill is 10
- Pa wants to write 20, Pb wants to write 40

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
}

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);     // line C1
  tmp = get();         // line C2
  sem_post(&empty);    // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
void put(int value) {
  buffer[fill] = value;    // line f1
  fill = (fill + 1) % MAX;   // line f2
}
```

- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time
- Let’s assume fill is 10
- Pa wants to write 20, Pb wants to write 40
- Pa executes line F1, Buffer[10] is now 20

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);       // line C1
  tmp = get();           // line C2
  sem_post(&empty);   // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
void put(int value) {
  buffer[fill] = value;   // line f1
  fill = (fill + 1) % MAX;   // line f2
}
```

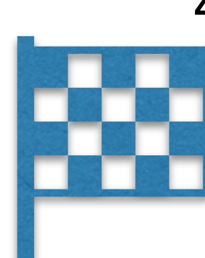
- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time
- Let's assume fill is 10
- Pa wants to write 20, Pb wants to write 40
- Pa executes line F1, Buffer[10] is now 20
- Before Pa executes F2, Pb executes F1. Buffer[10] is now 40

Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
 }

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);       // line C1
  tmp = get();           // line C2
  sem_post(&empty);  // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
5  void put(int value) {
6    buffer[fill] = value;   // line f1
7    fill = (fill + 1) % MAX;   // line f2
8  }
```

- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time
- Let's assume fill is 10
- Pa wants to write 20, Pb wants to write 40
- Pa executes line F1, Buffer[10] is now 20
- Before Pa executes F2, Pb executes F1. Buffer[10] is now 40

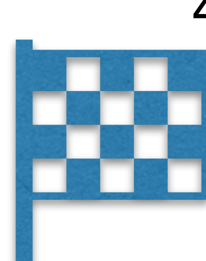


Producer Consumer Problem using Semaphores

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
 sem_wait(&empty);   // line P1
 put(i);             // line P2
 sem_post(&full);    // line P3
  }
}

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
  sem_wait(&full);      // line C1
  tmp = get();          // line C2
  sem_post(&empty);     // line C3
  printf("%d\n", tmp);
  }
}
...
```

```
void put(int value) {
  buffer[fill] = value;    // line f1
  fill = (fill + 1) % MAX;    // line f2
}
```

- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time
- Let's assume fill is 10
- Pa wants to write 20, Pb wants to write 40
- Pa executes line F1, Buffer[10] is now 20
- Before Pa executes F2, Pb executes F1. Buffer[10] is now 40



Race condition

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

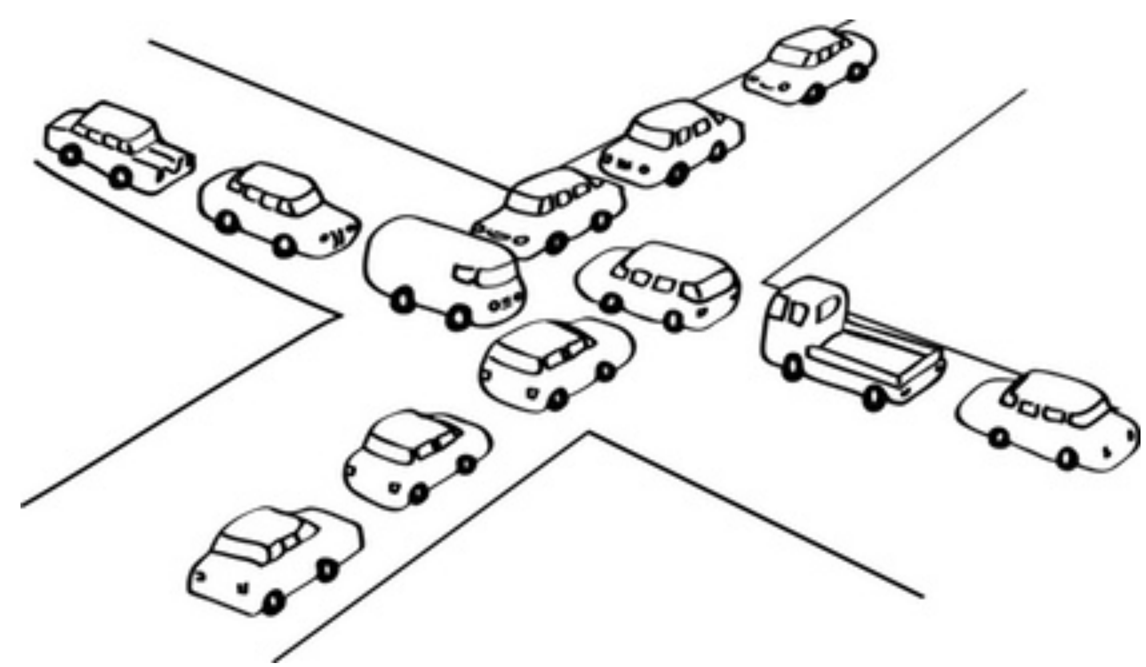
Producer Consumer Problem using Semaphores Adding mutual exclusion

```
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 <u>still holds the mutex</u>!

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).

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

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

```c
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)
  }
}
```

```c
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.

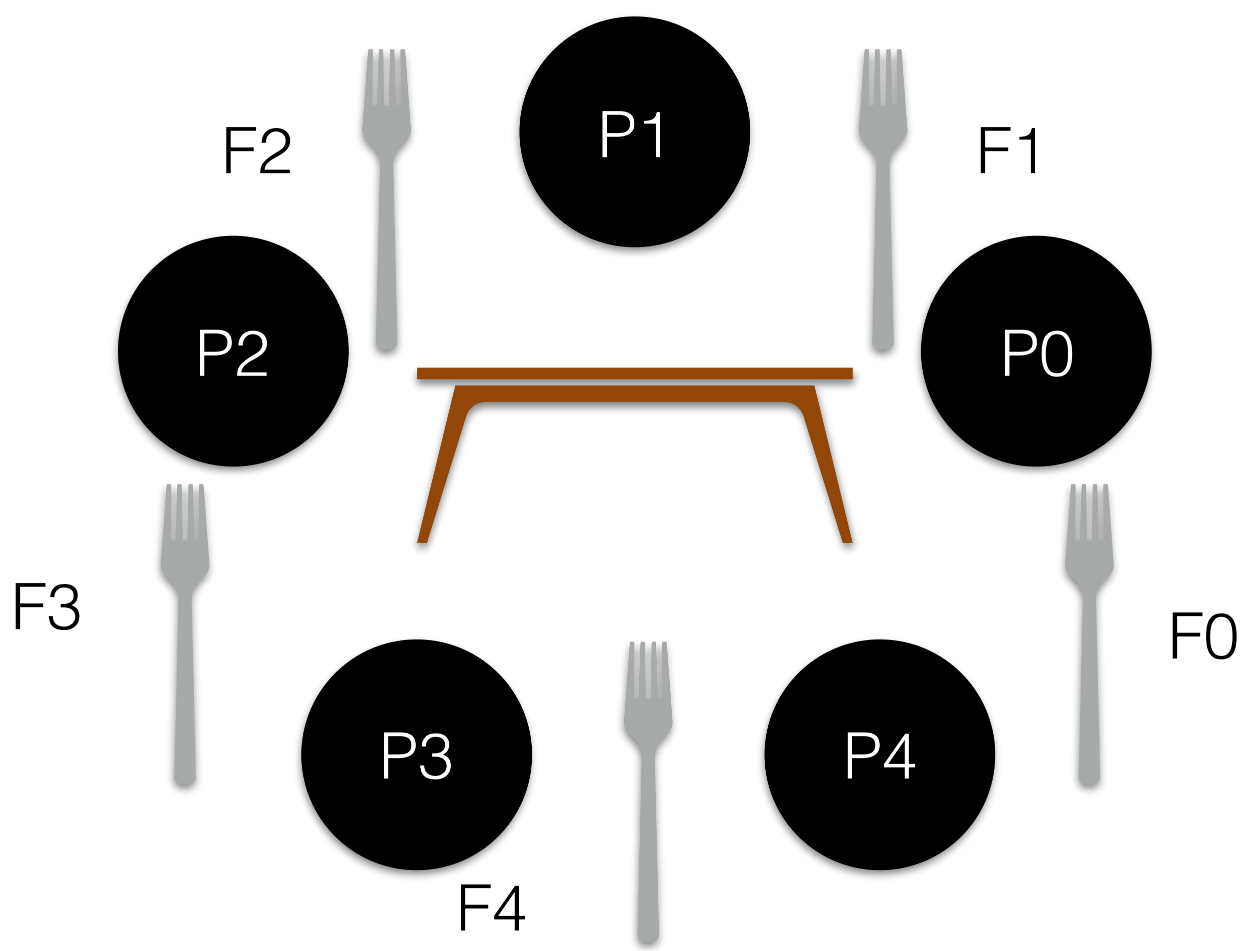
Producer Consumer Problem using Semaphores Correct Solution

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

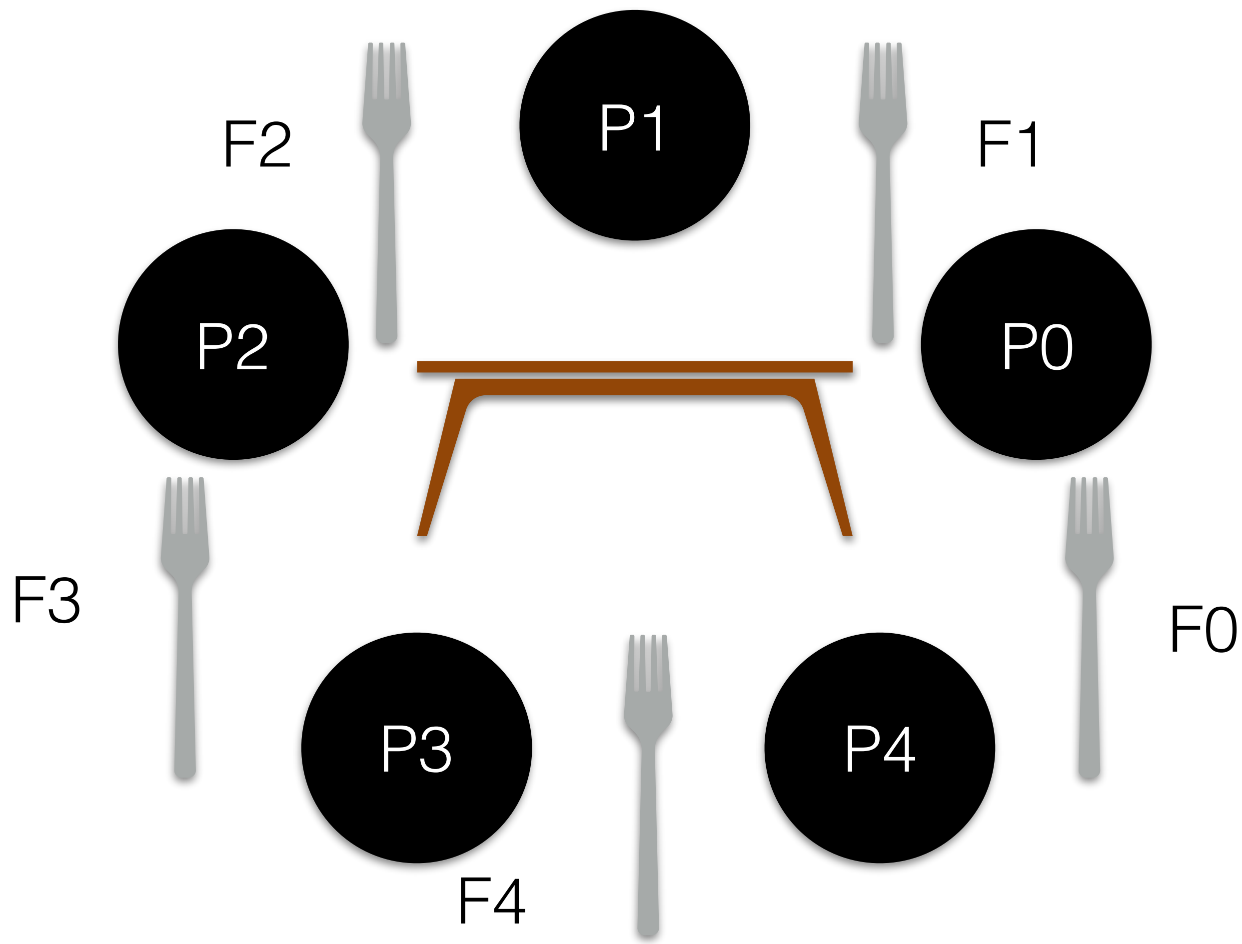
```
void *producer(void *arg) {
 int i;
 for (i = 0; i < loops; i++) {
  sem_wait(&empty);   // line p1
  sem_wait(&mutex);
   put(i);   // line p2
   sem_post(&mutex);
   sem_post(&full);   // line p3
   }
 }

```

```
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

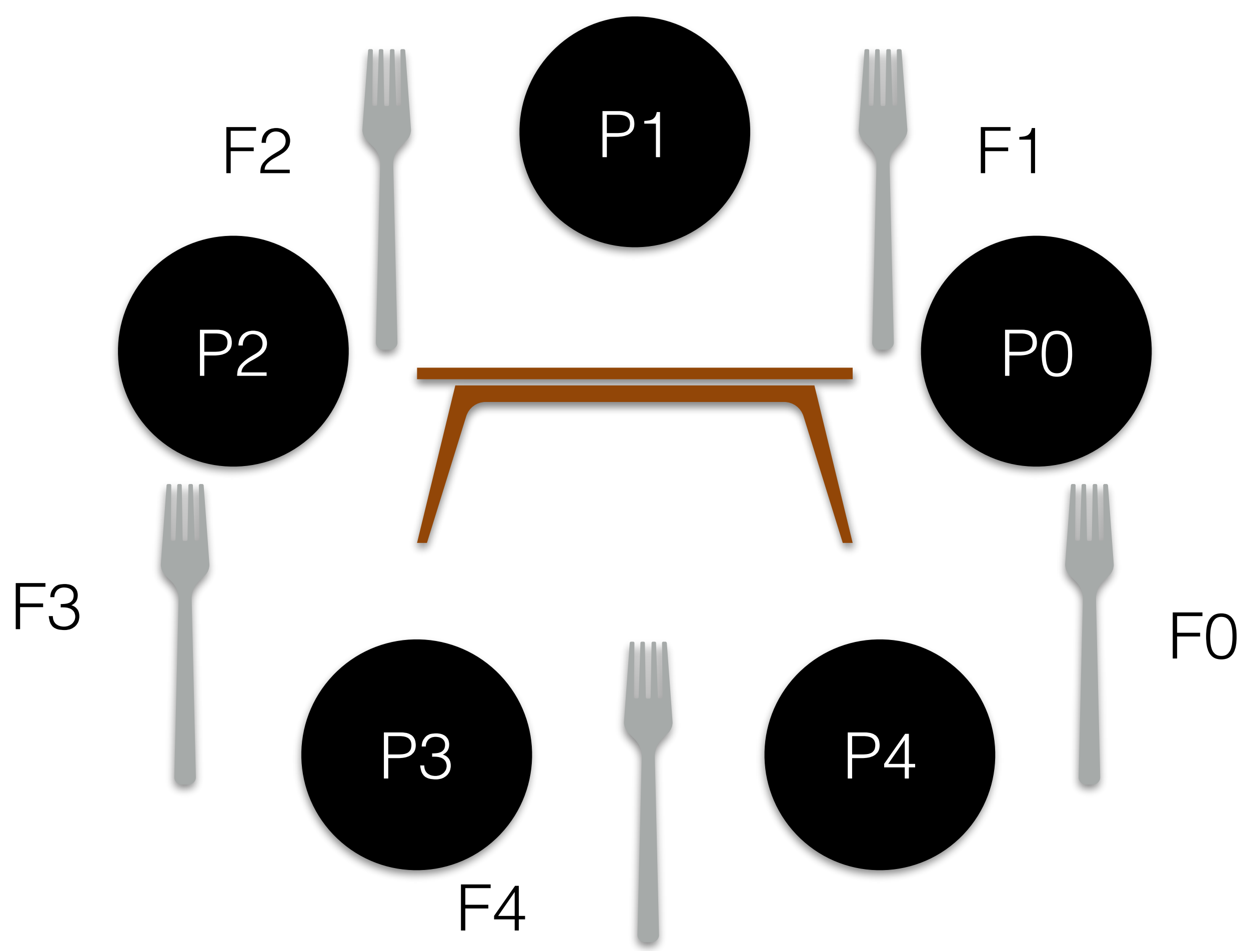


Dining Philosopher's Problem



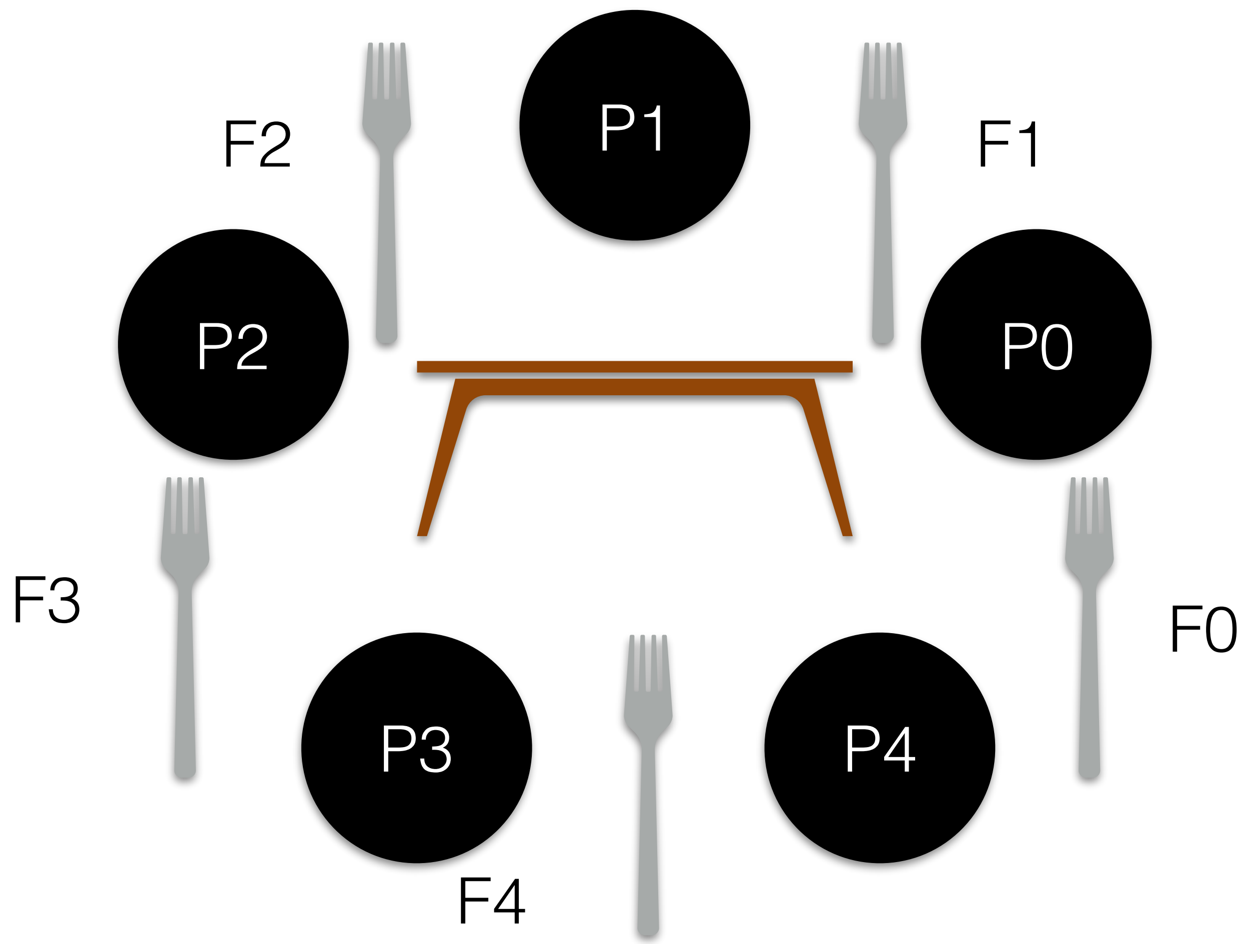
- 5 philosophers sitting around a table

Dining Philosopher's Problem



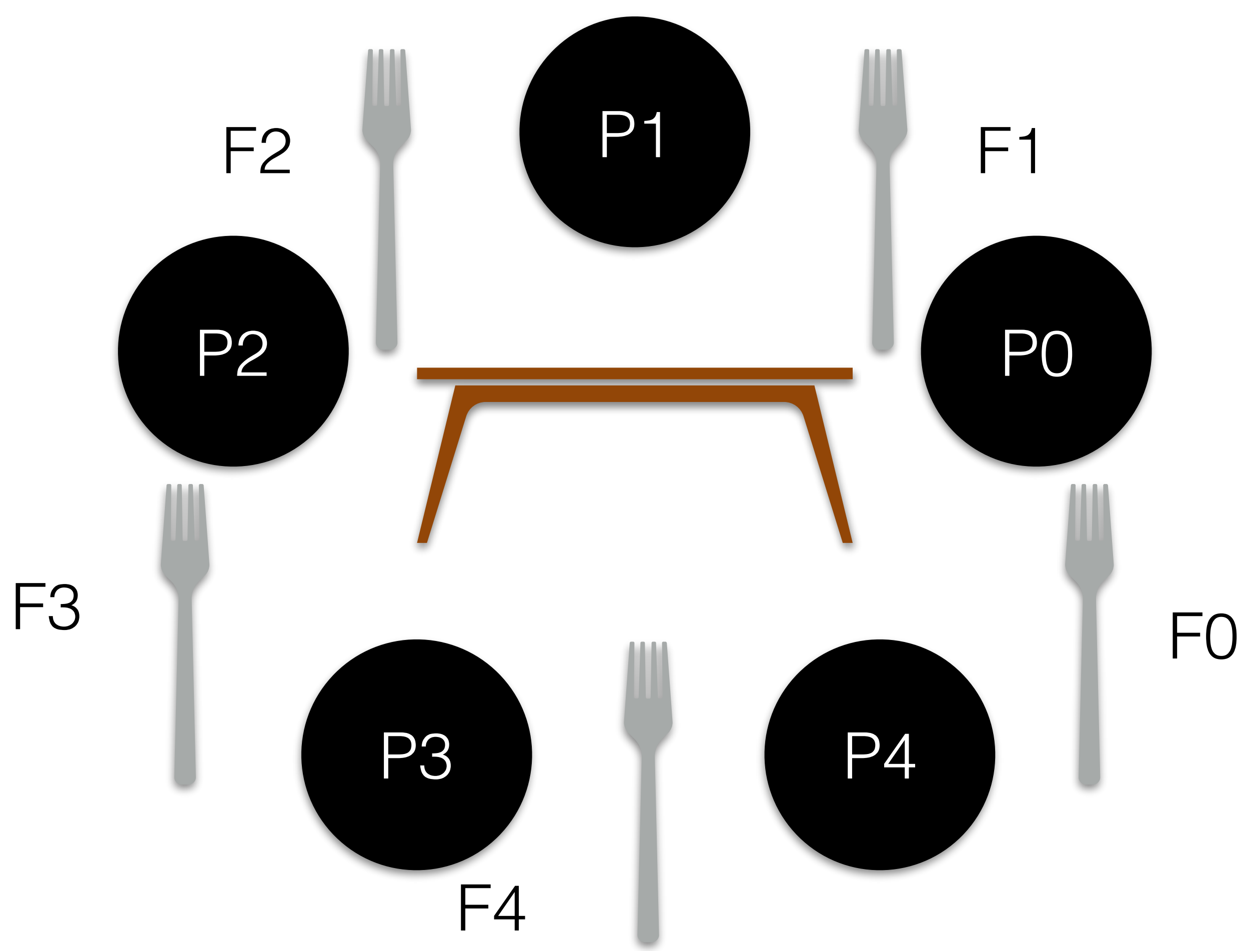
- 5 philosophers sitting around a table
- A fork between a pair of philosophers

Dining Philosopher's Problem



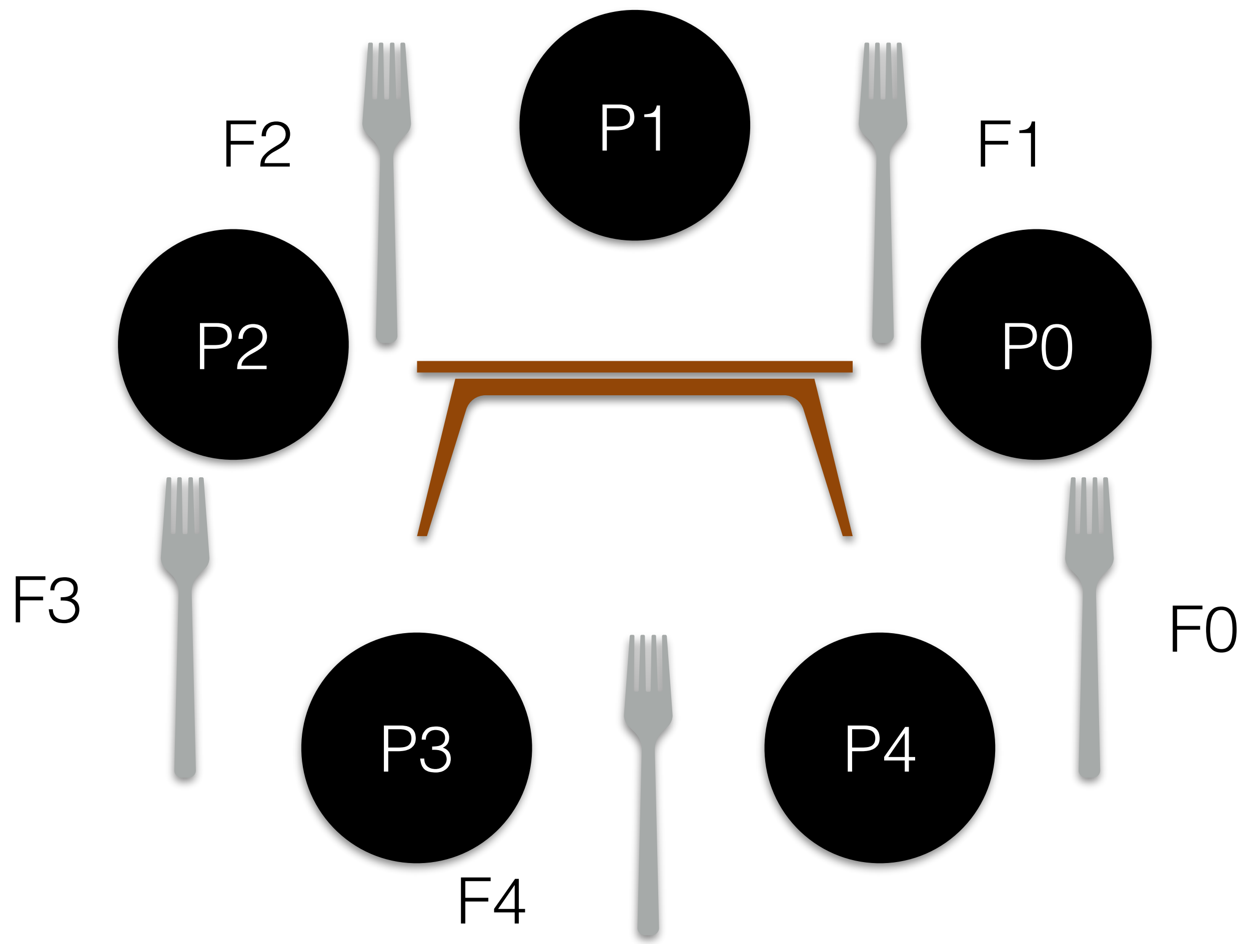
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:

Dining Philosopher's Problem



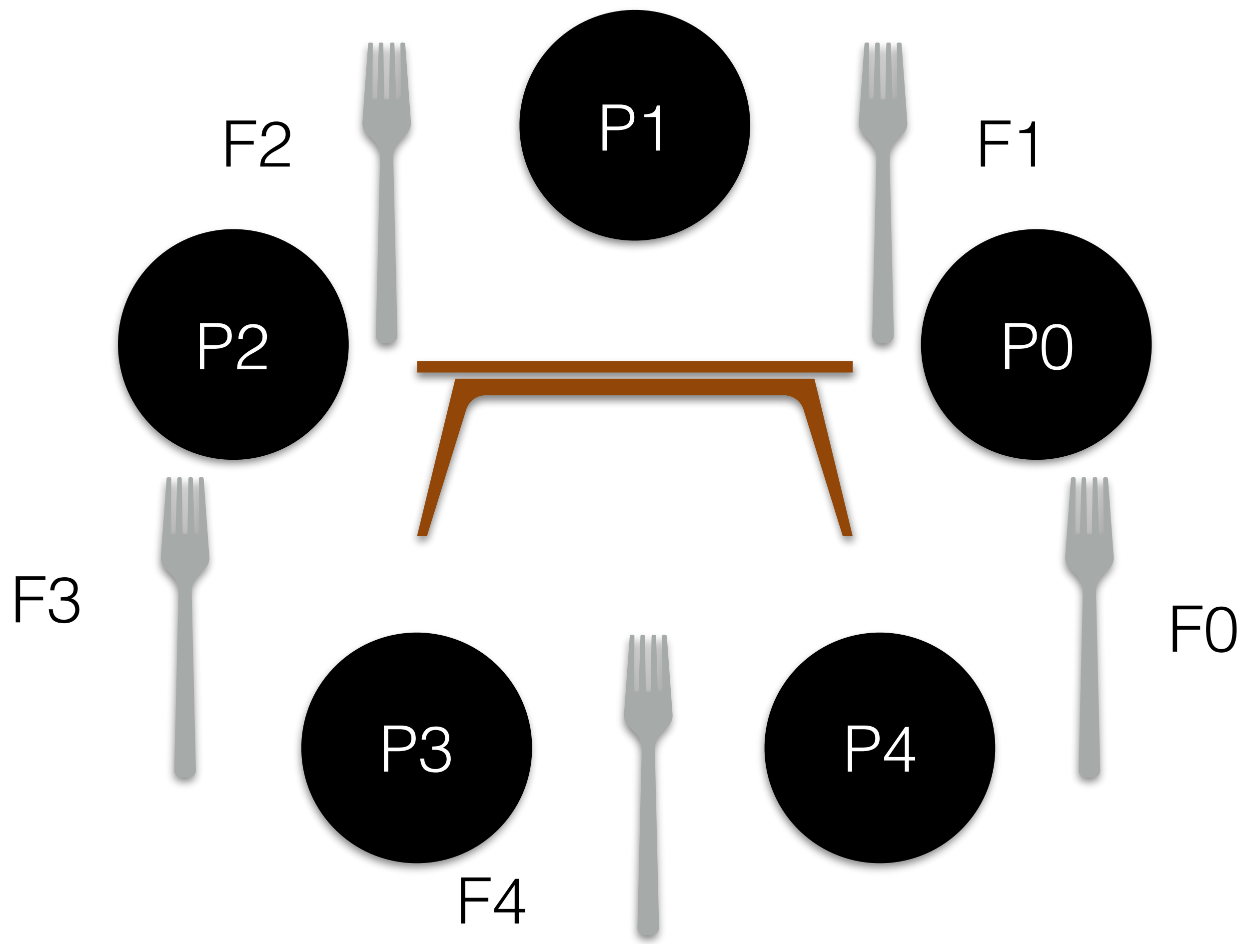
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork

Dining Philosopher's Problem



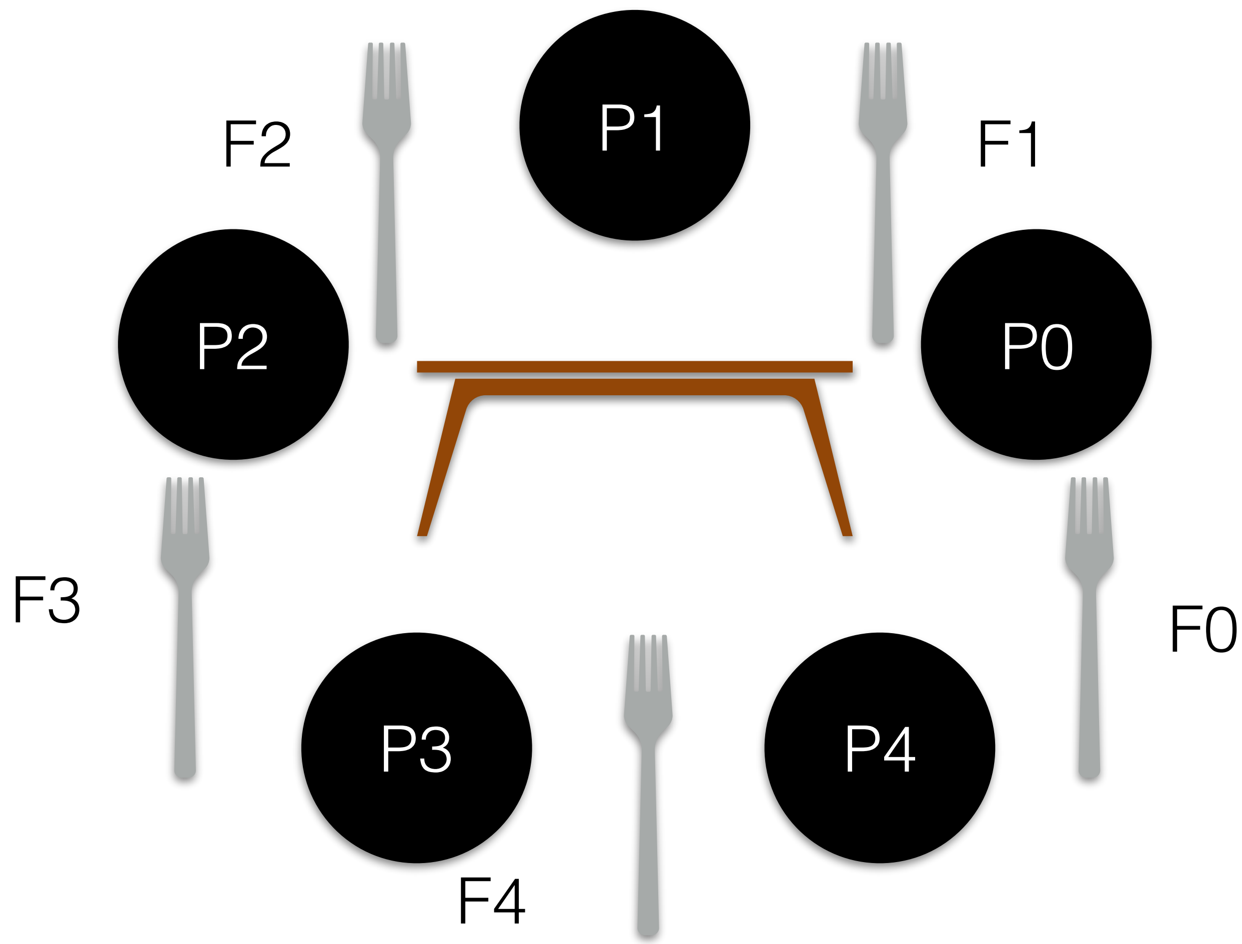
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right

Dining Philosopher's Problem



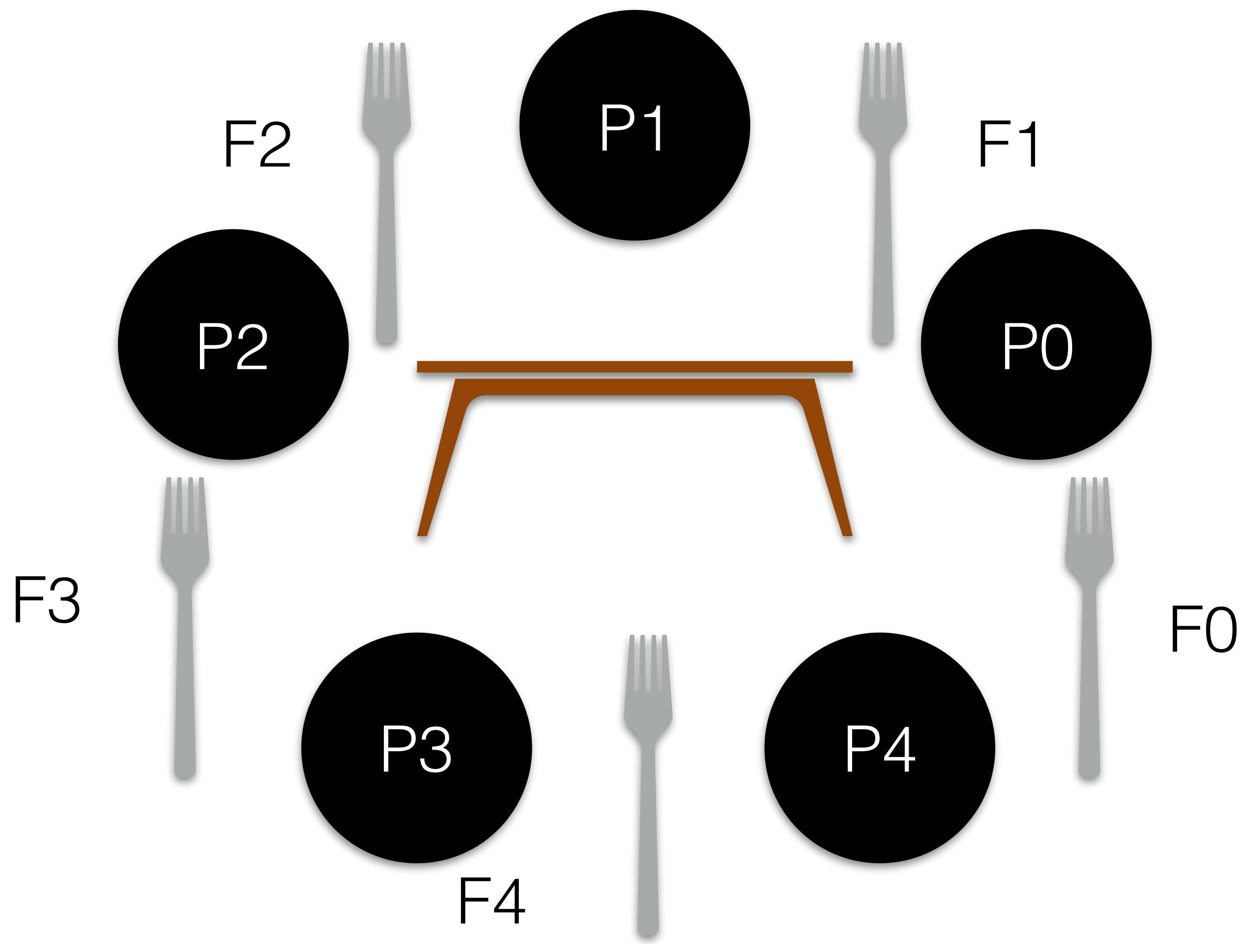
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
    - Think — don't need fork
    - Eat — need fork on left and right
    - Forks have contention

Dining Philosopher's Problem



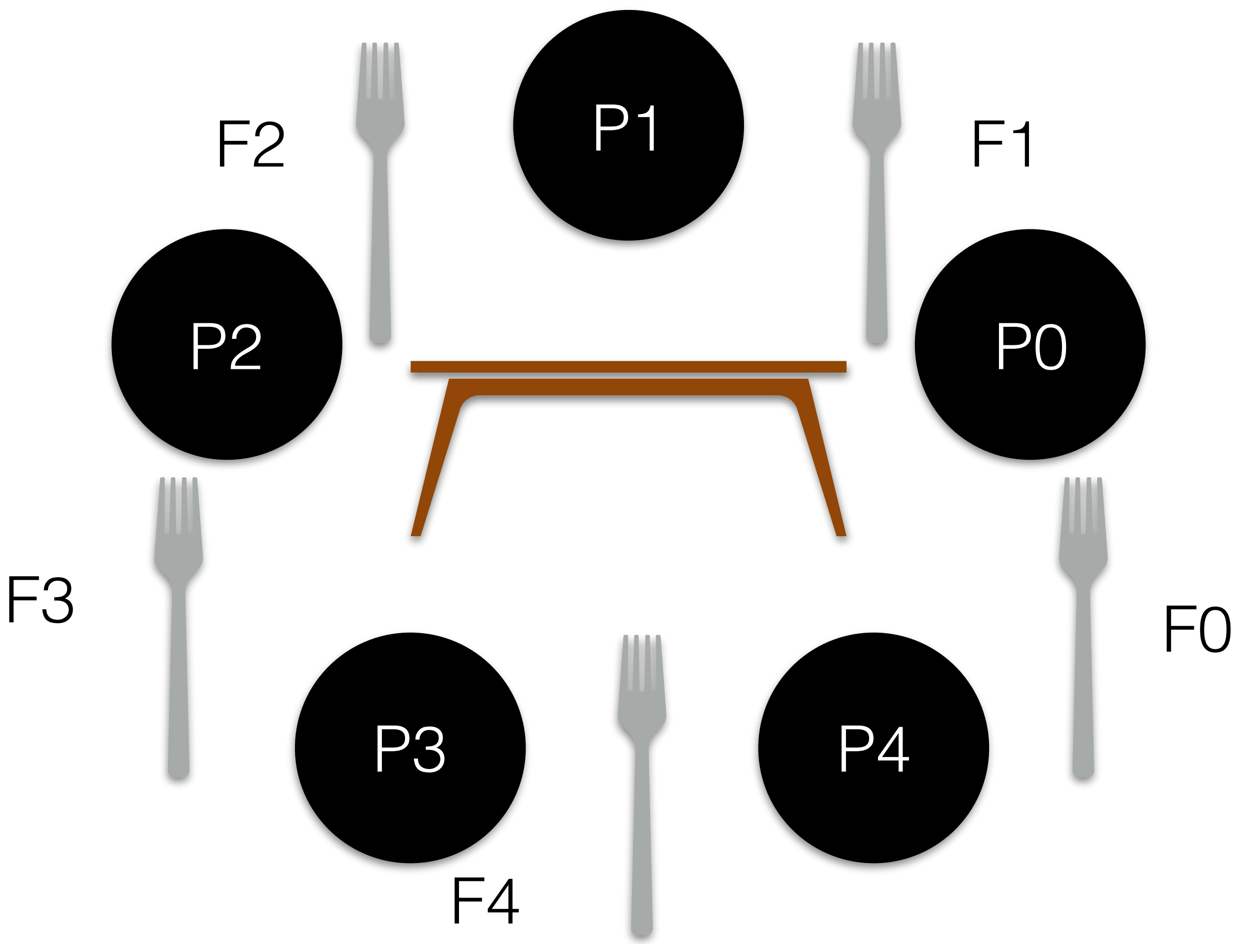
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:

Dining Philosopher's Problem



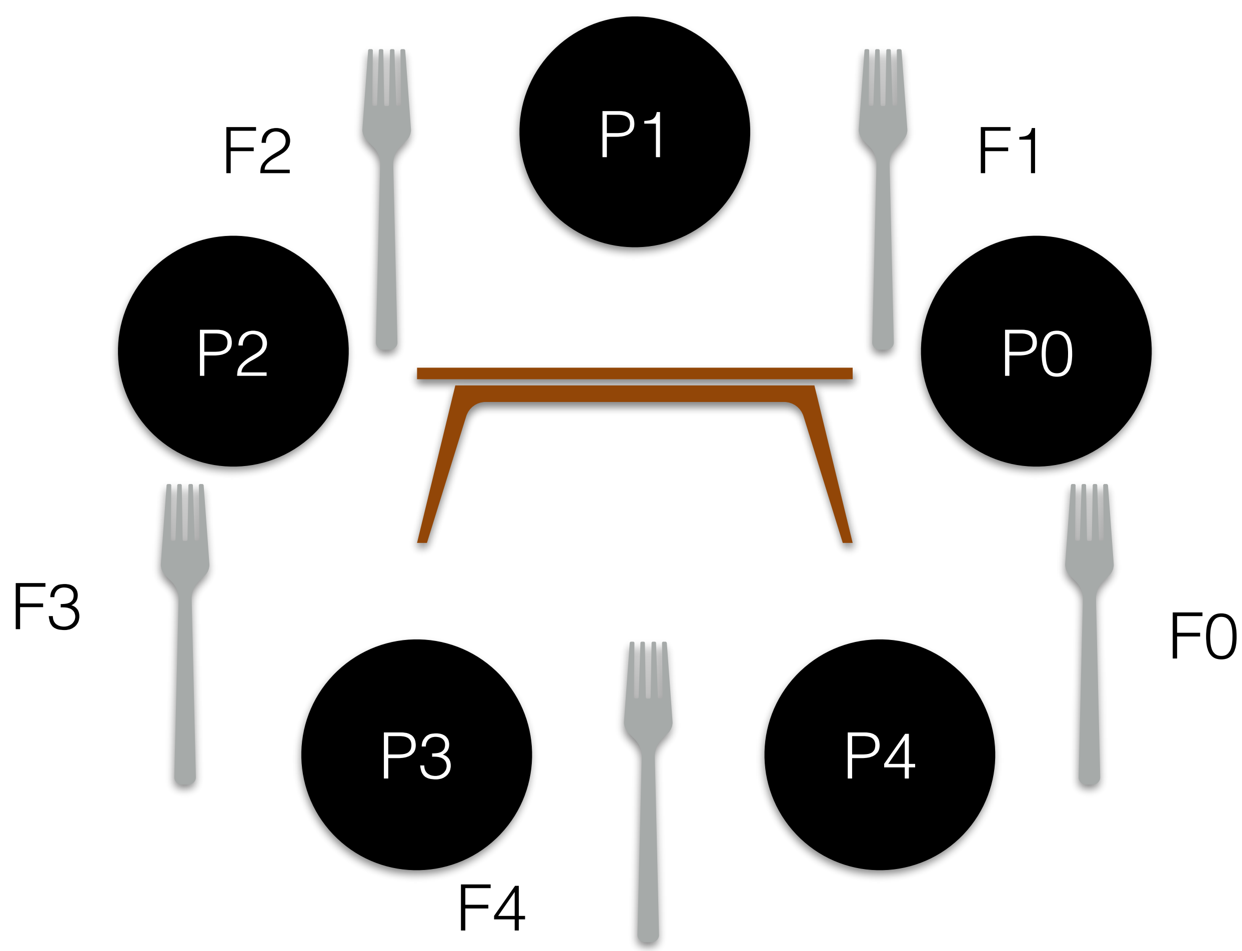
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:
  - No deadlock

Dining Philosopher's Problem



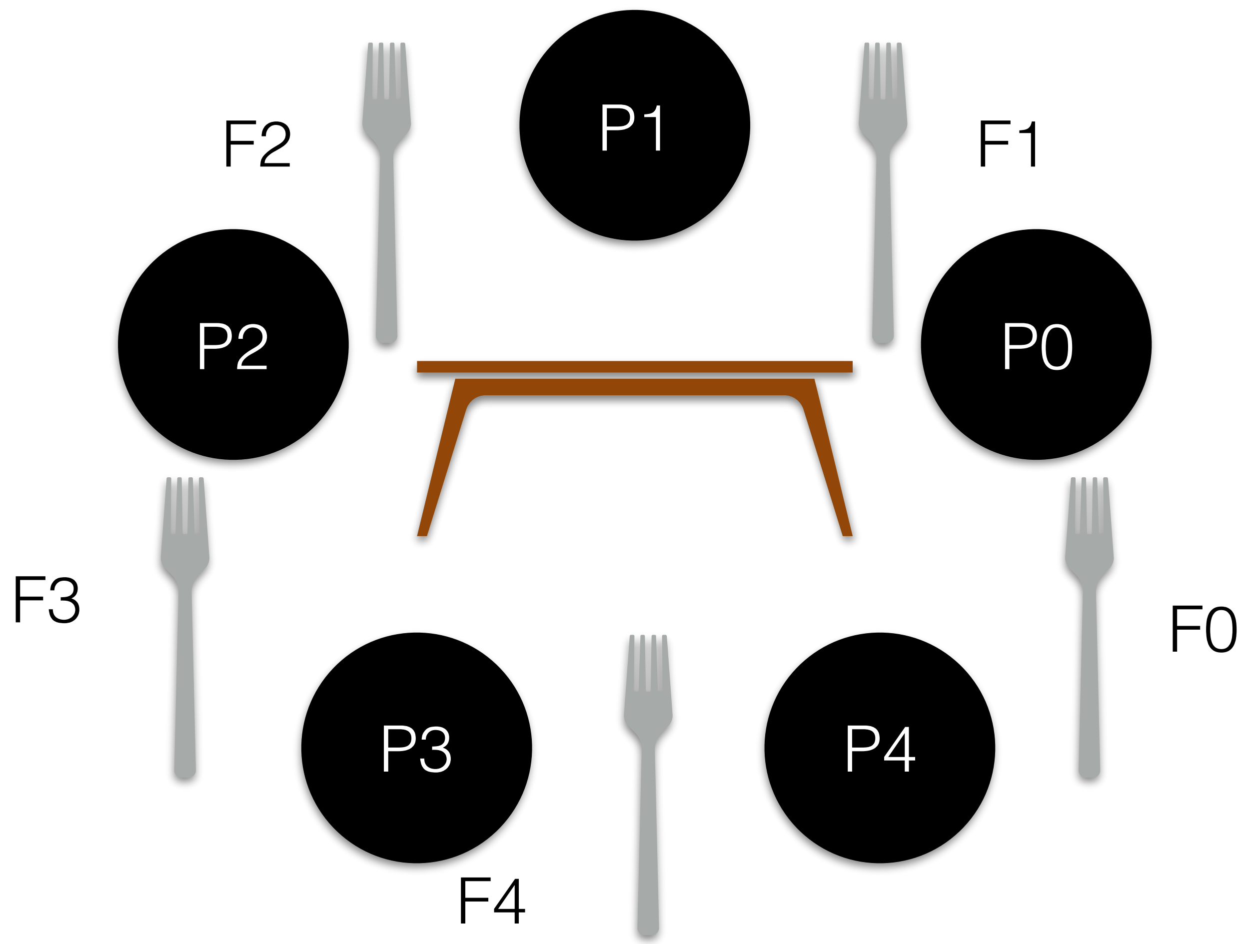
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:
  - No deadlock
  - No one starves

Dining Philosopher's Problem



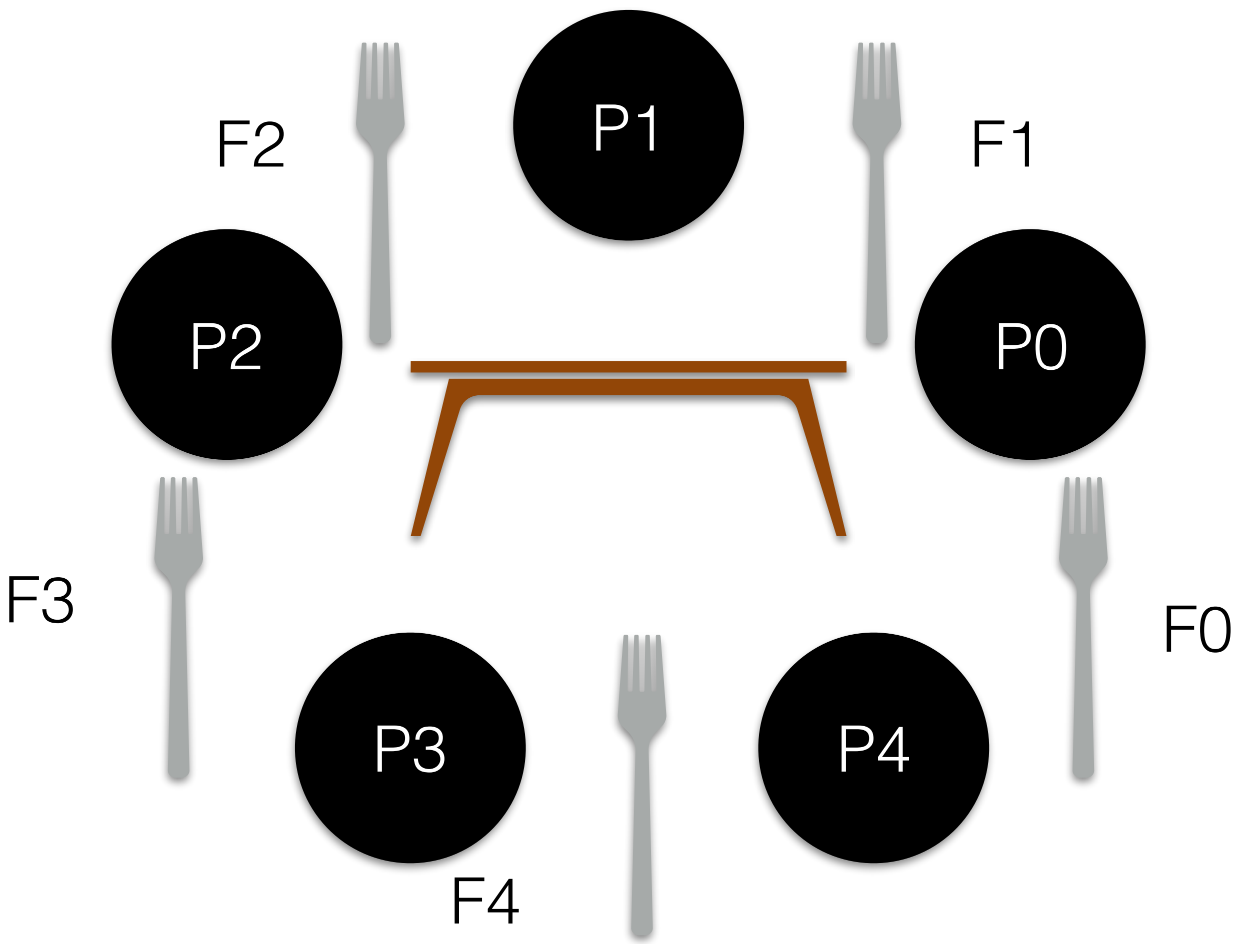
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:
  - No deadlock
  - No one starves
  - High Concurrency

Dining Philosopher's Problem




```
while (1) {
  think();
  getforks();
  eat();
  putforks();
}

// helper functions
int left(int p) { return p; }

int right(int p) {
  return (p + 1) % 5;
}
```

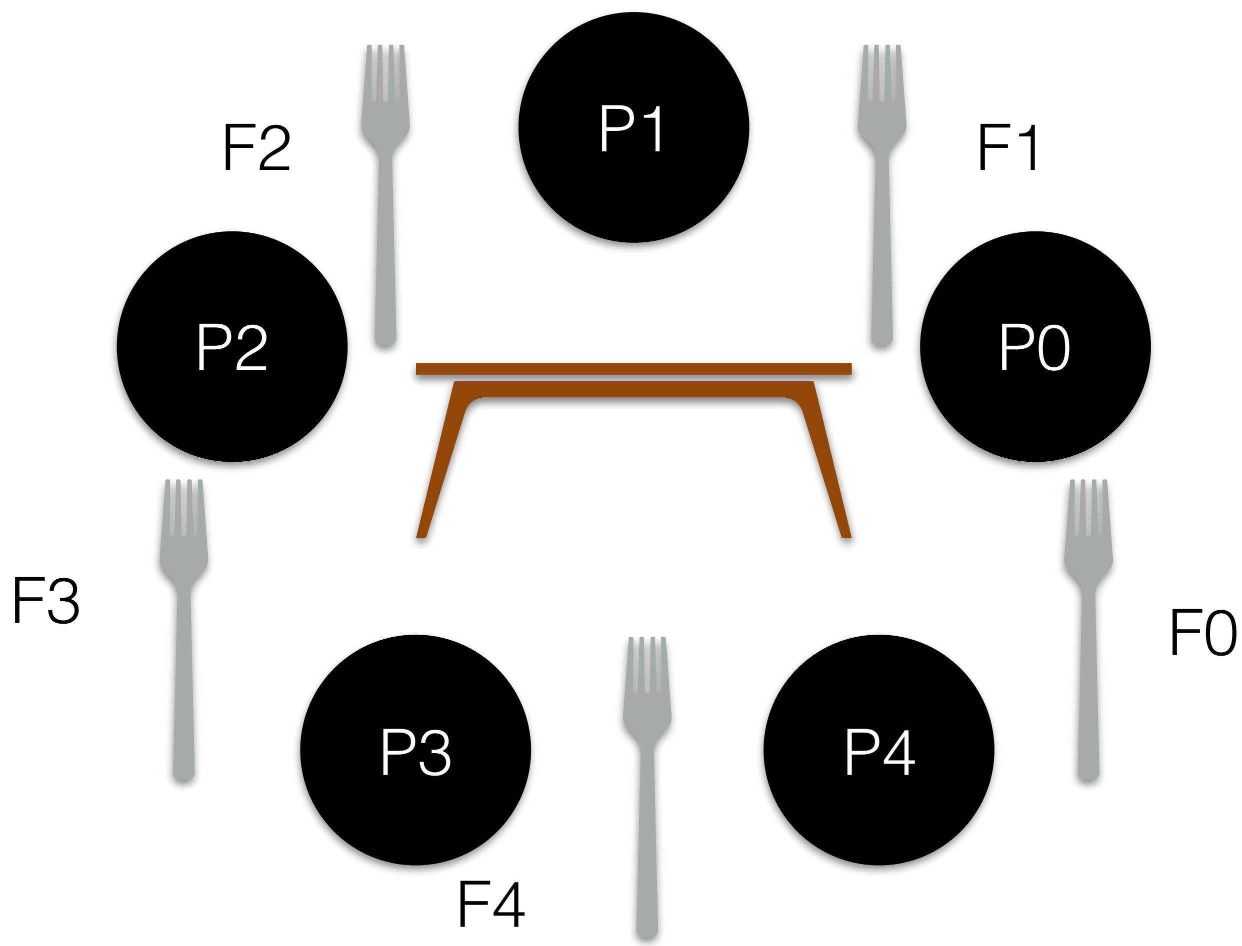
Dining Philosopher's Problem



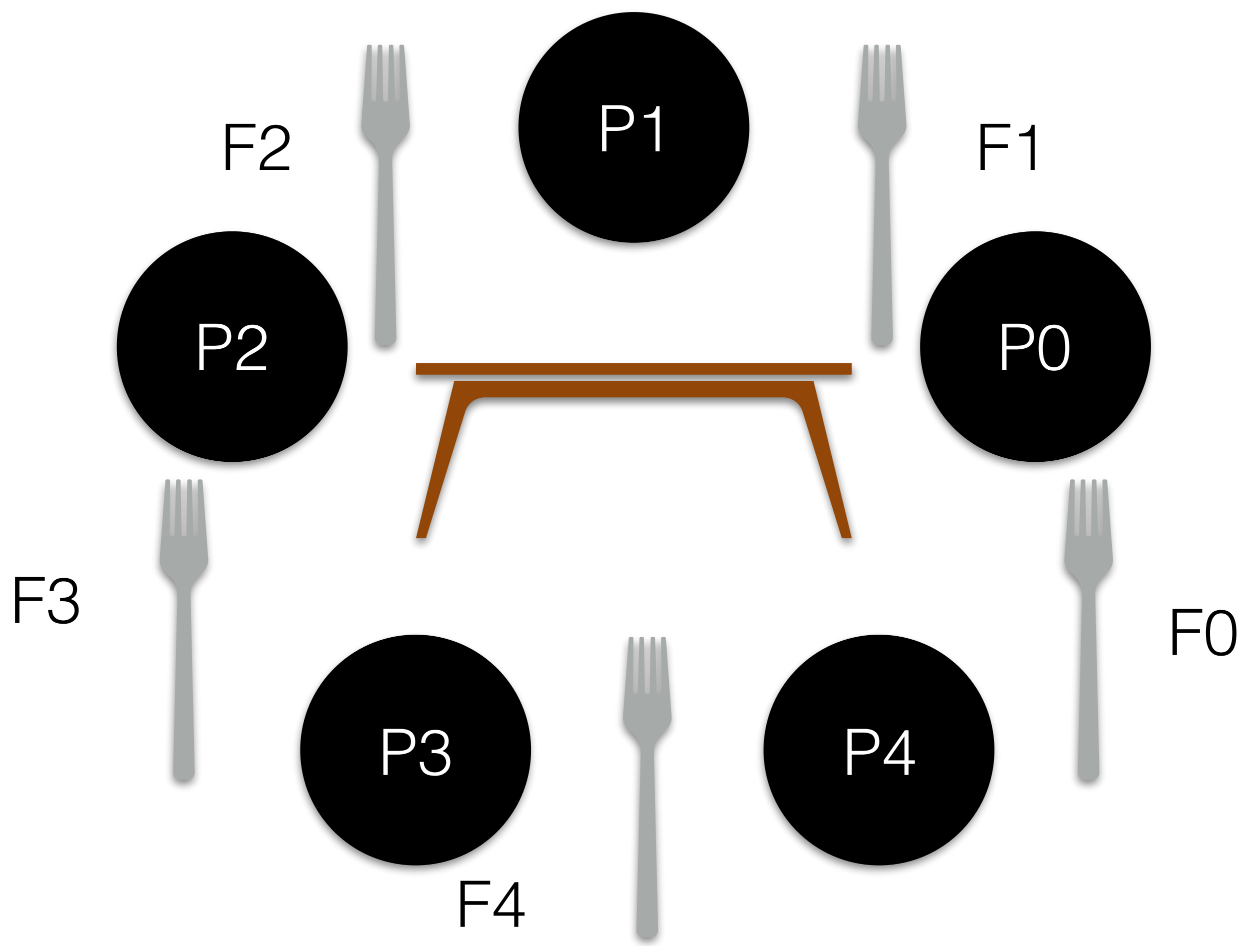

```
while (1) {
  think();
  getforks();
  eat();
  putforks();
}

// helper functions
int left(int p) { return p; }

int right(int p) {
  return (p + 1) % 5;
}
```

1. Using the provided routines write a simple working solution without concurrency
2. Now with concurrency

Dining Philosopher's Problem

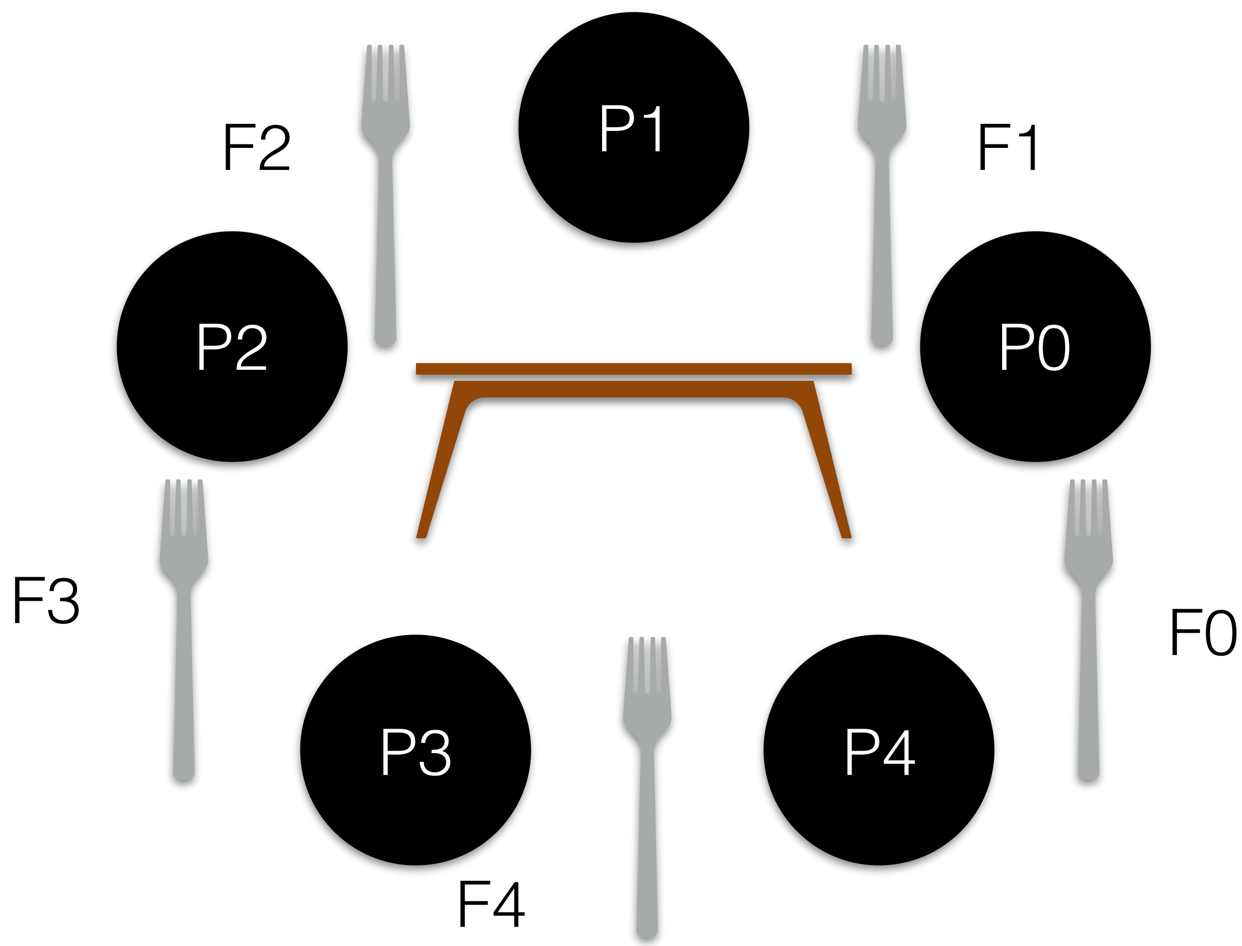


Dining Philosopher's Problem




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

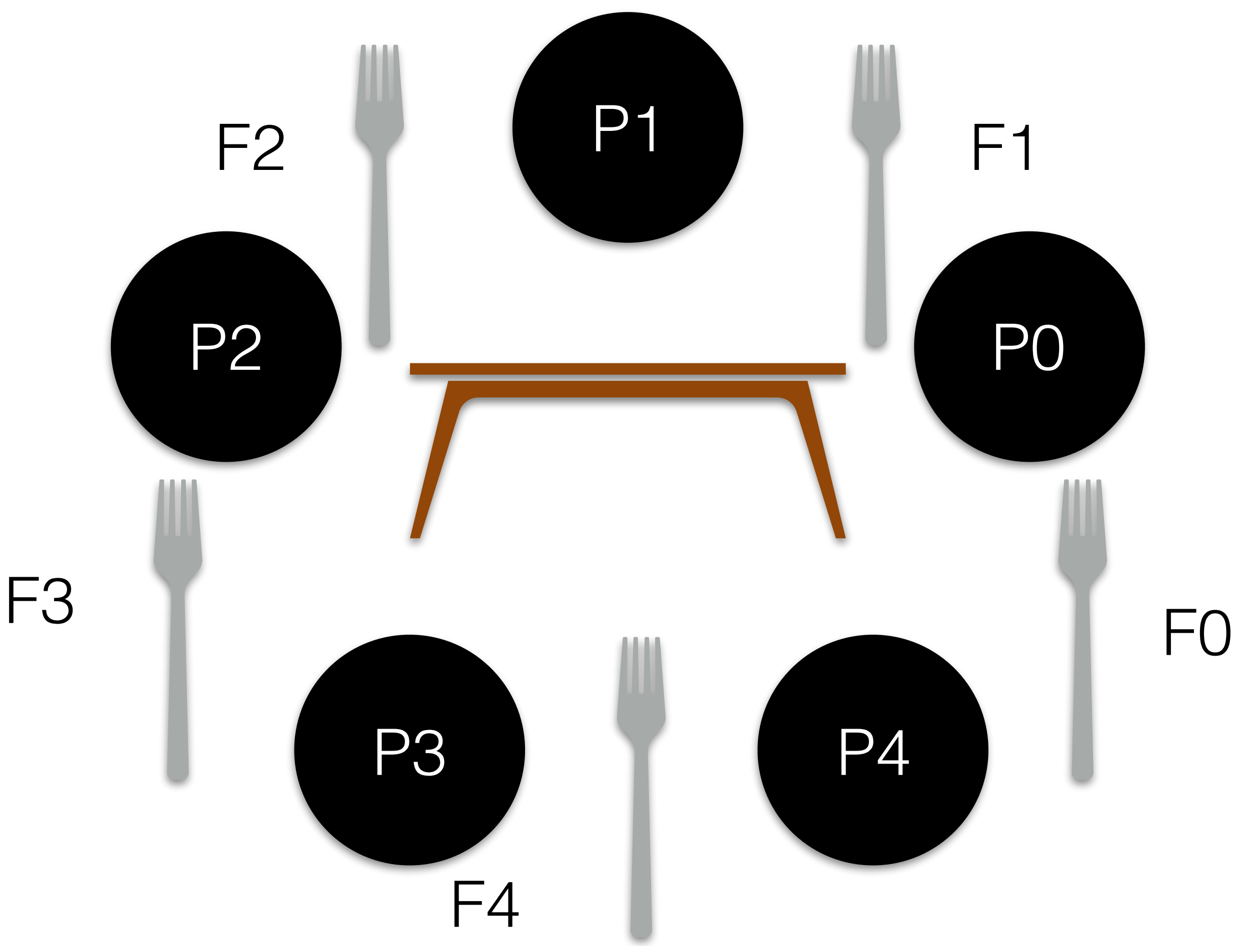
Dining Philosopher's Problem




```
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

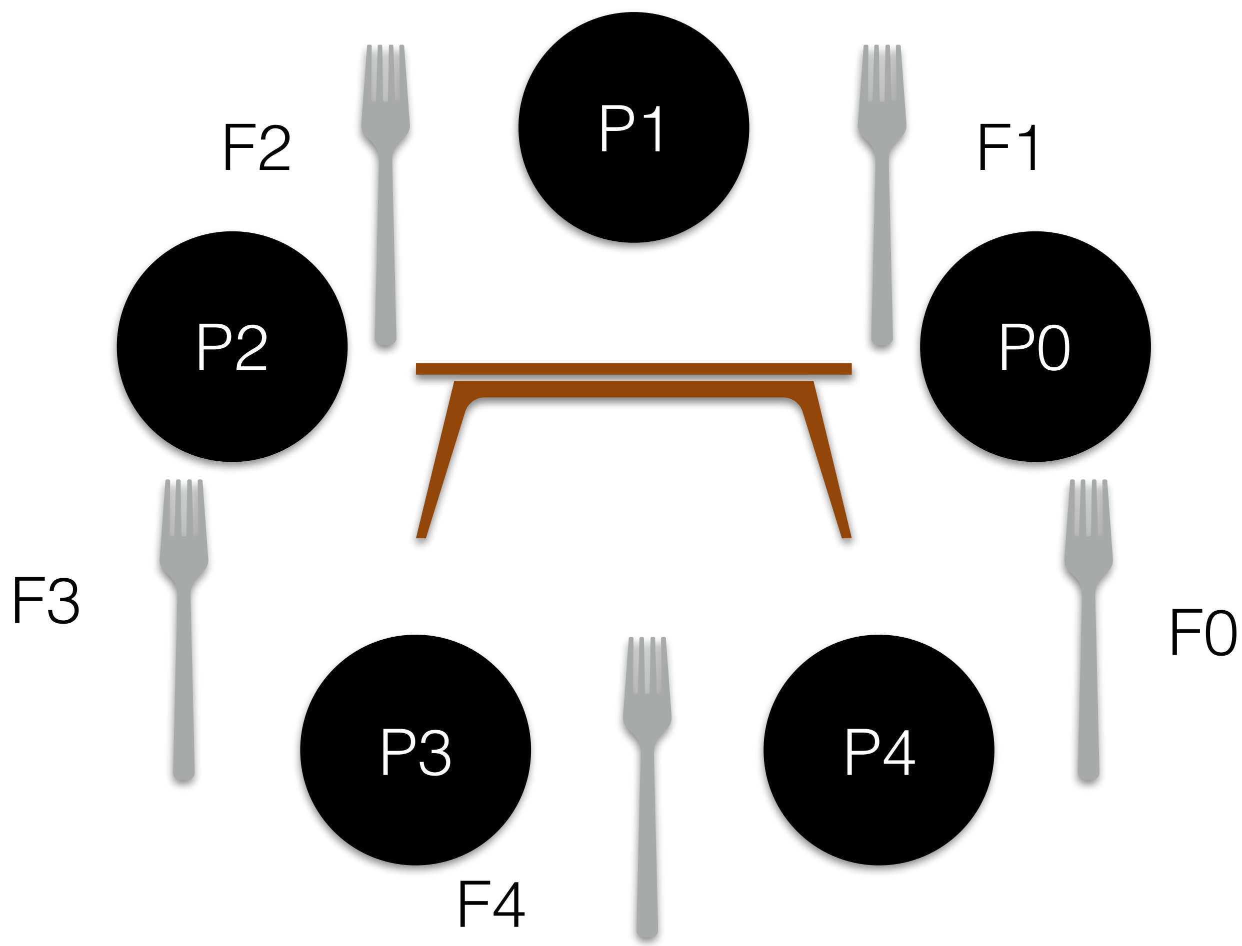



```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

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

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

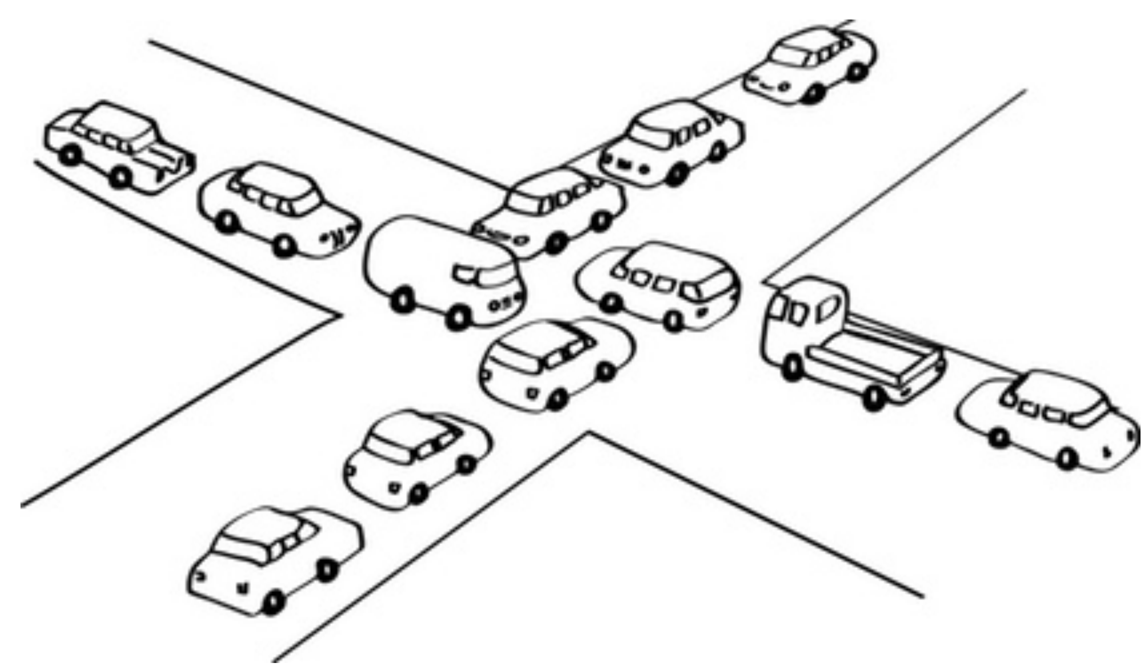
Dining Philosopher's Problem



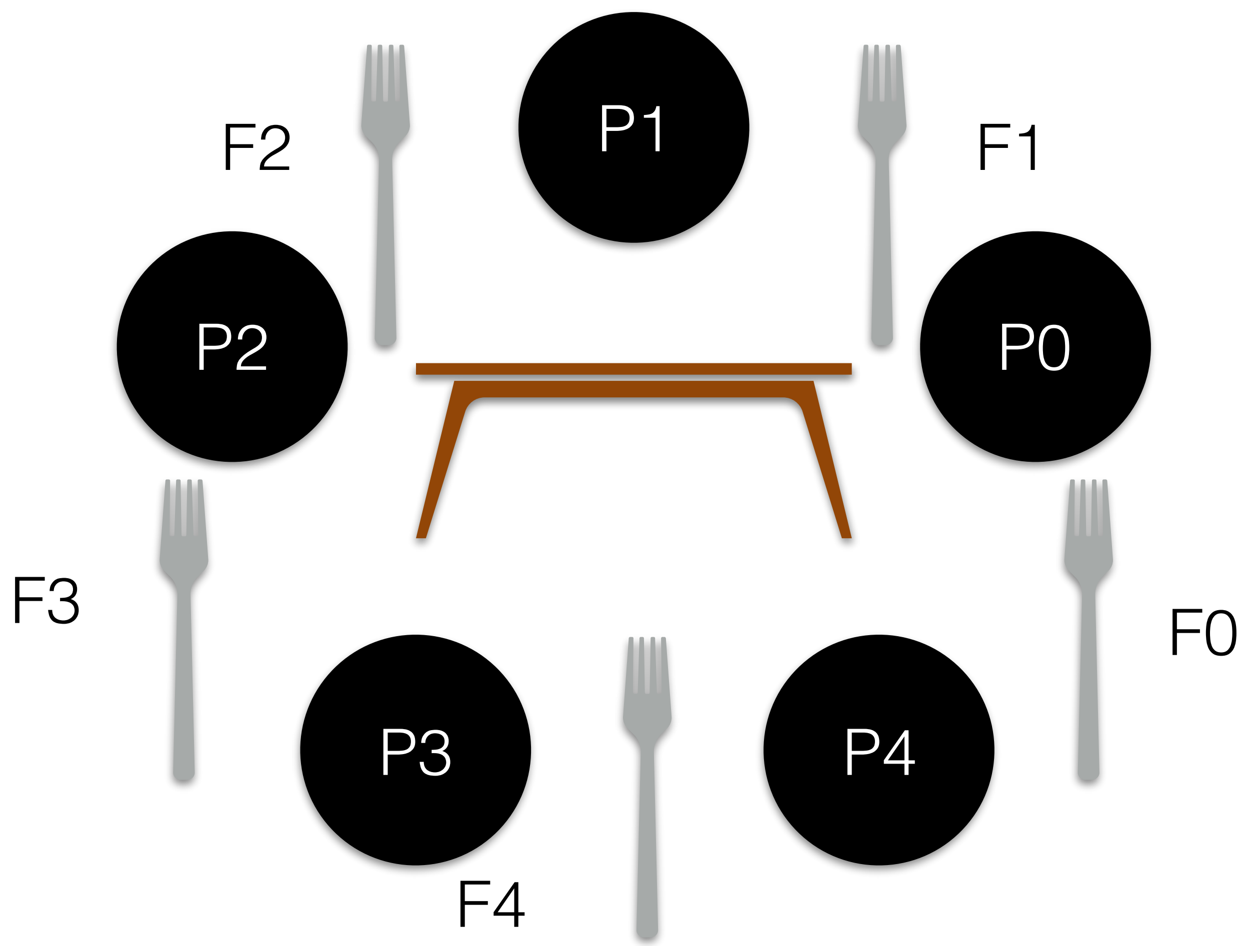

```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

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

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



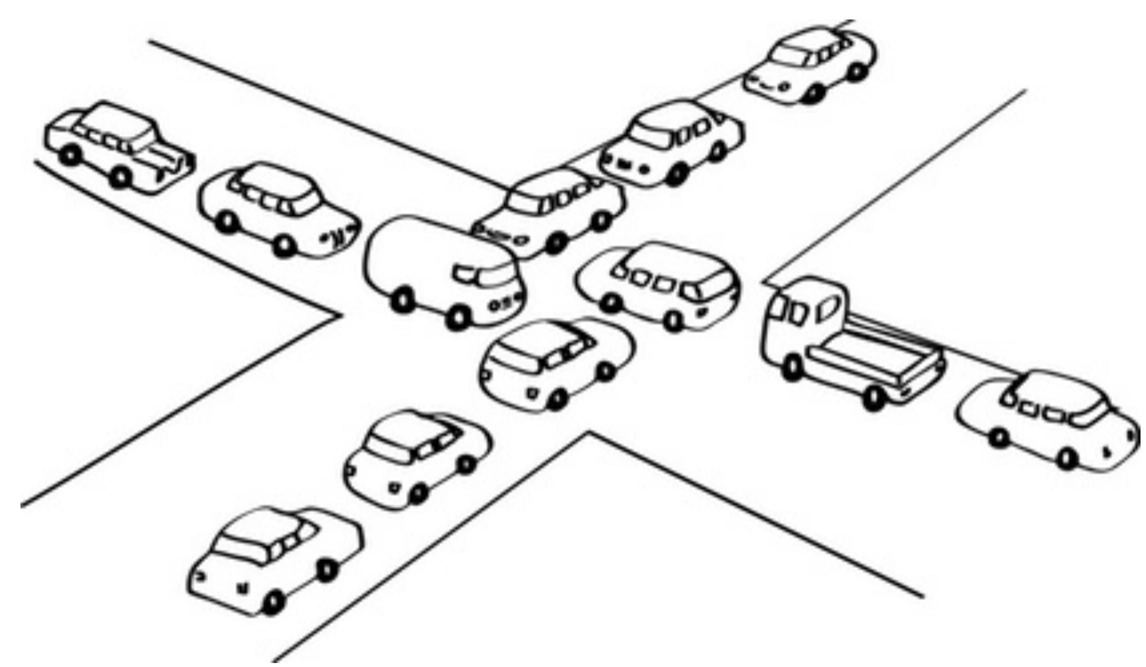
Dining Philosopher's Problem



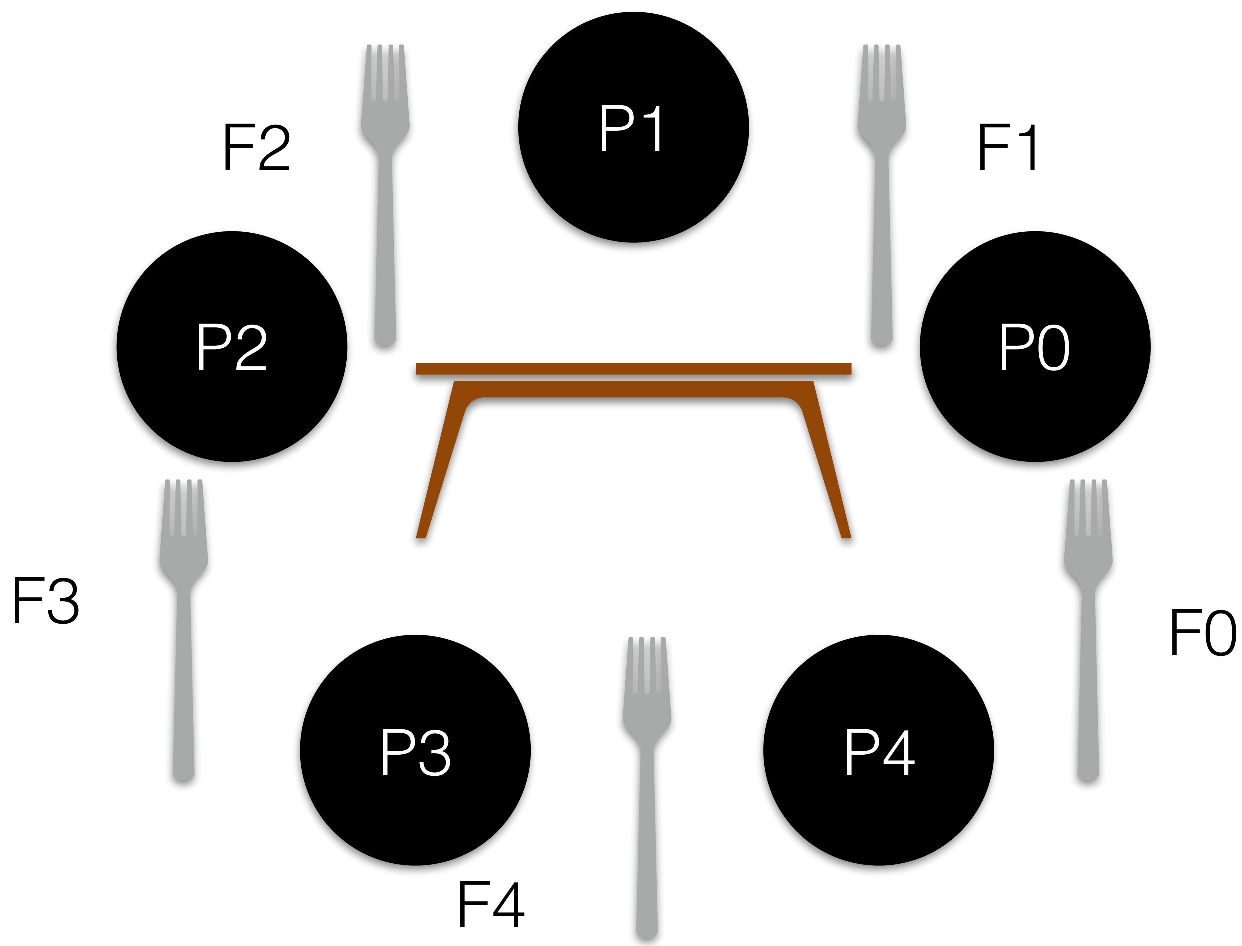

```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}
```

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

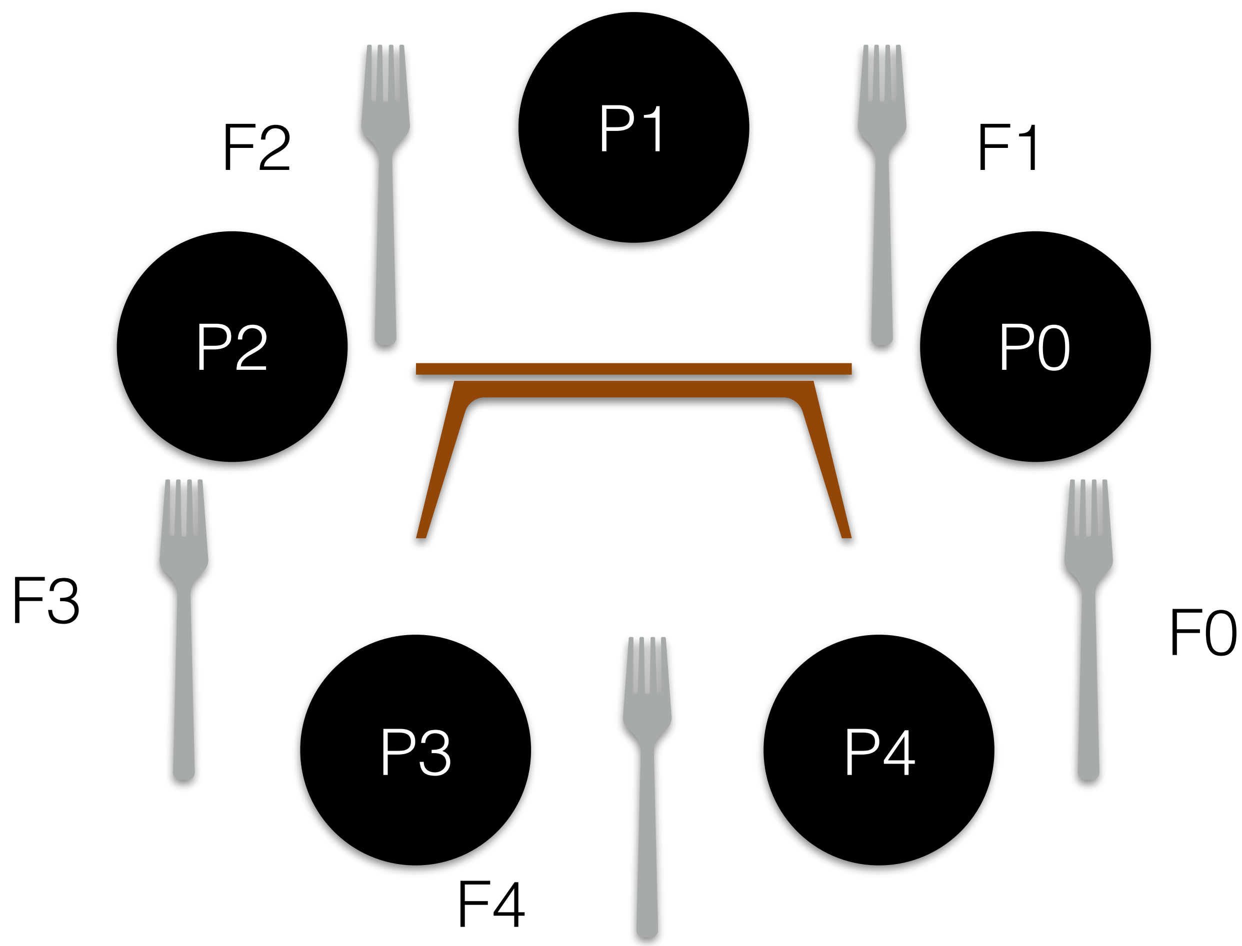
- P0 picks F0; P1 picks F1; …, P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?



Dining Philosopher's Problem

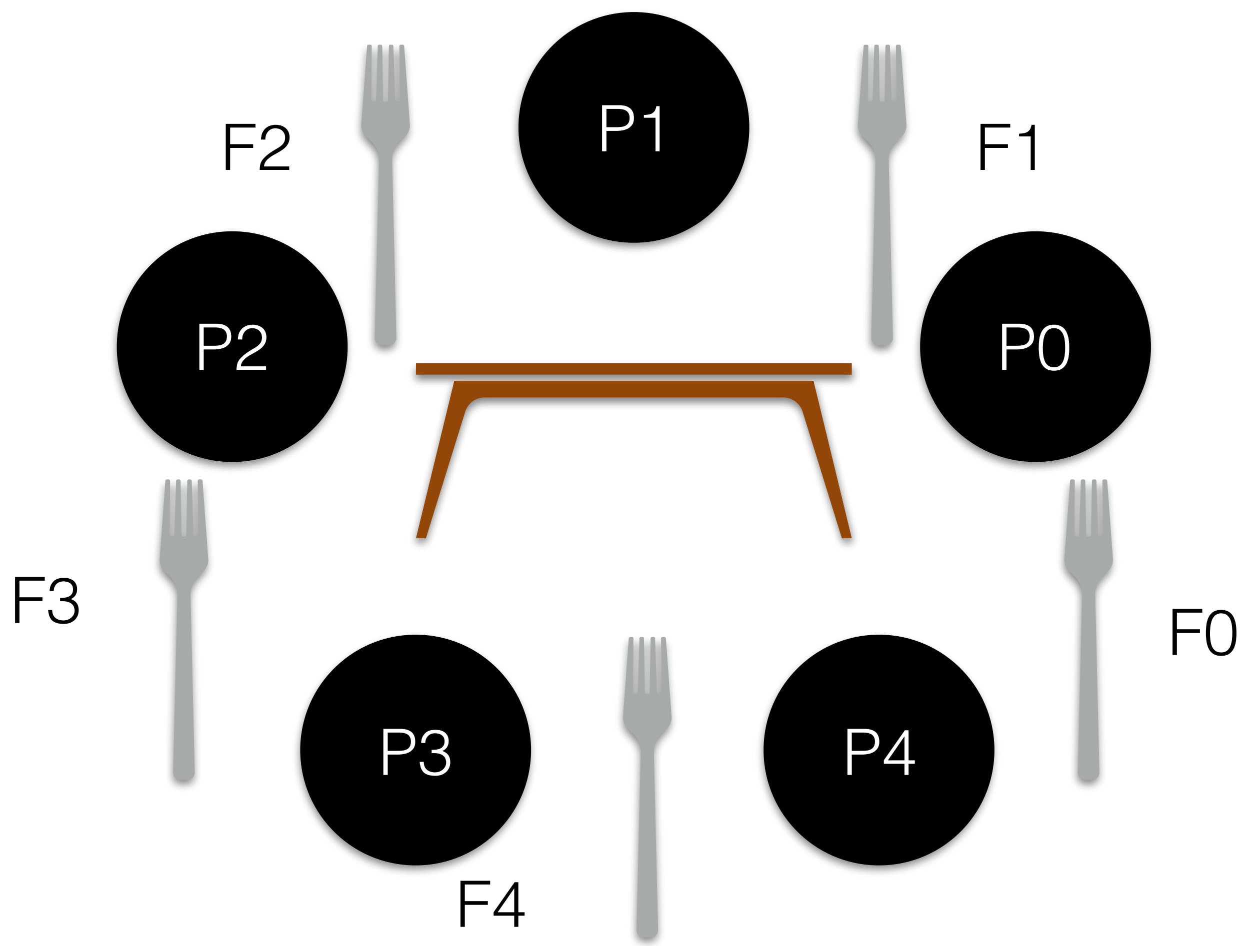


Dining Philosopher's Problem



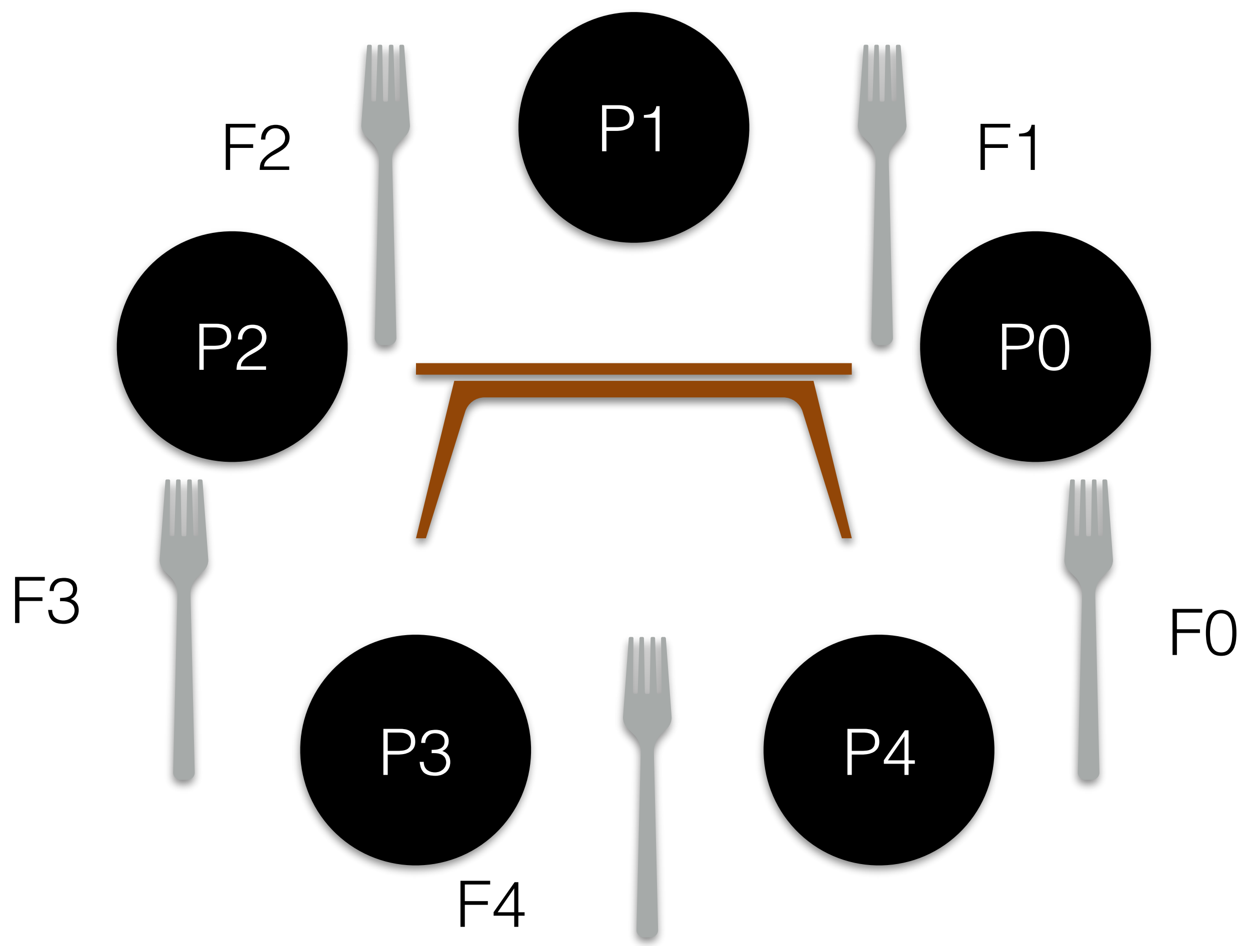
- P0 picks F0; P1 picks F1; …, P4 picks F4

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

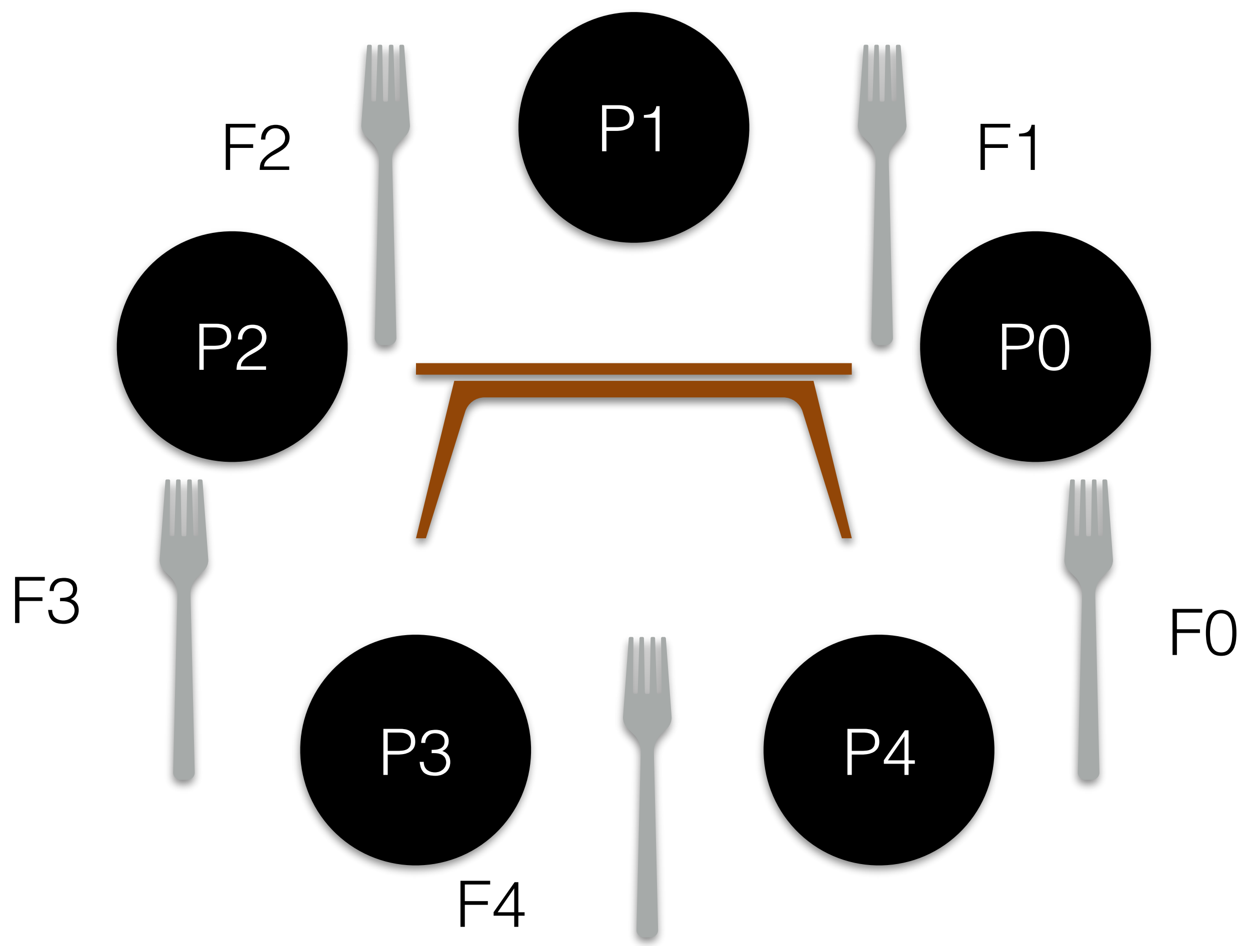


- If P ==4:

- 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?

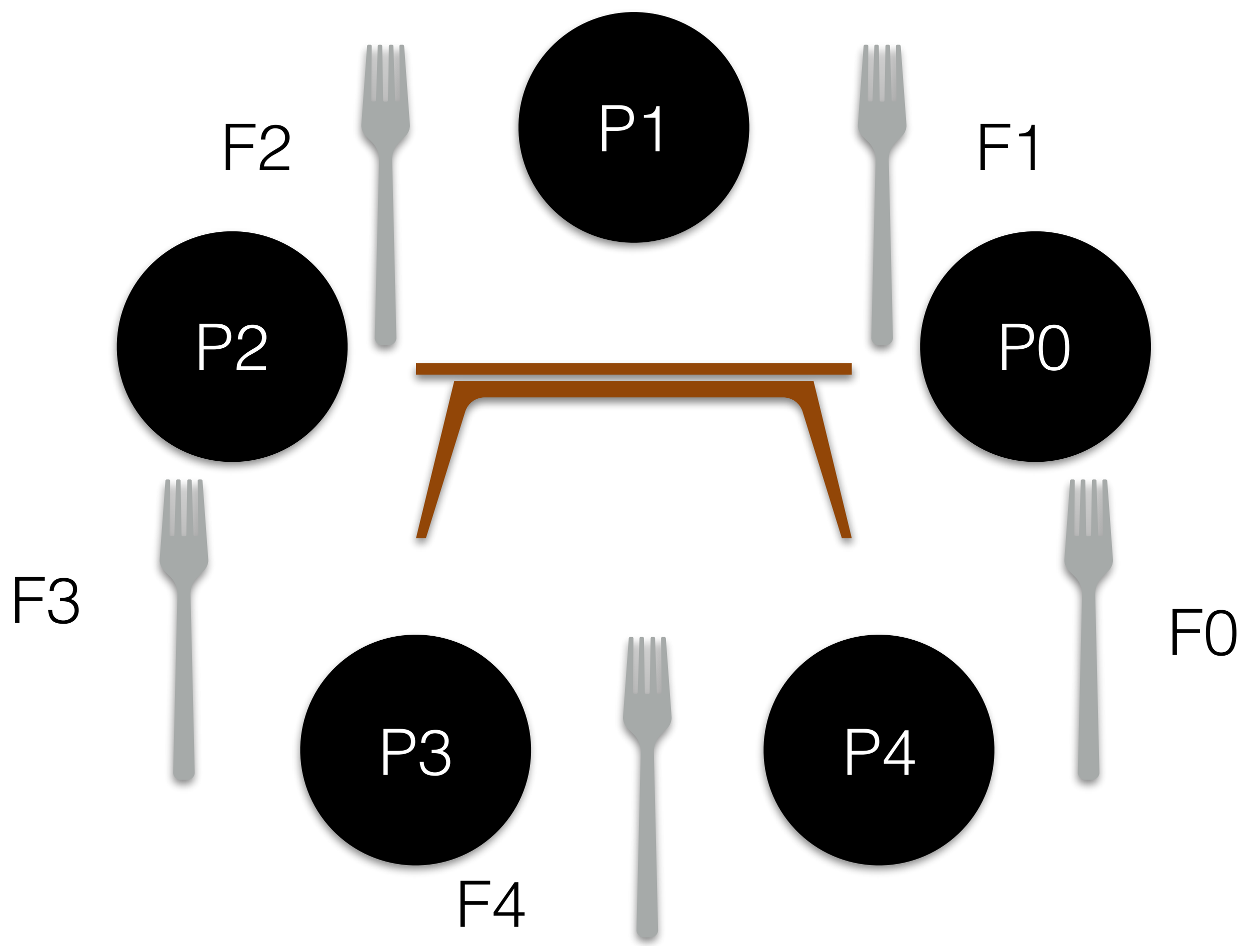
Dining Philosopher's Problem



- 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?

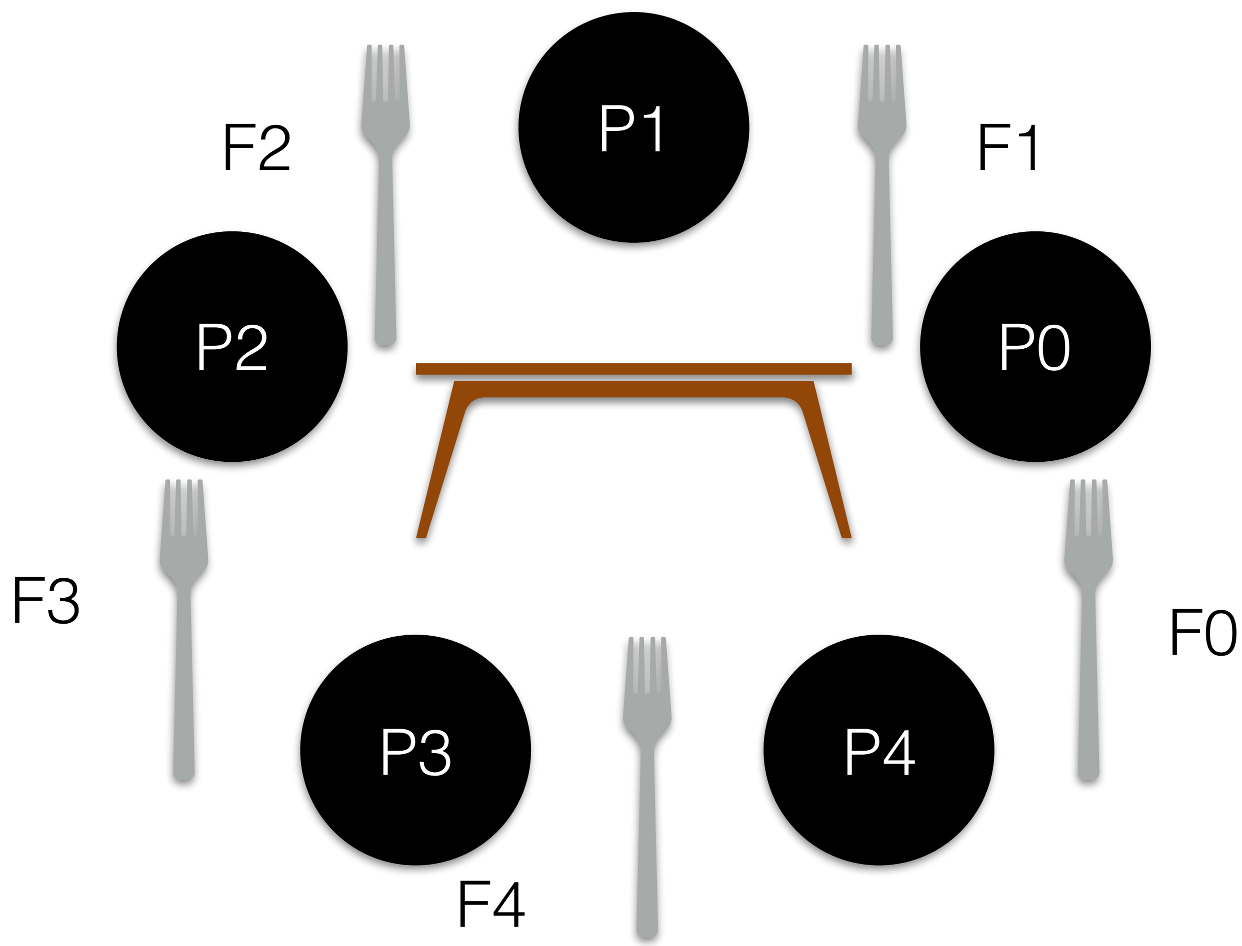
Dining Philosopher's Problem



- 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?

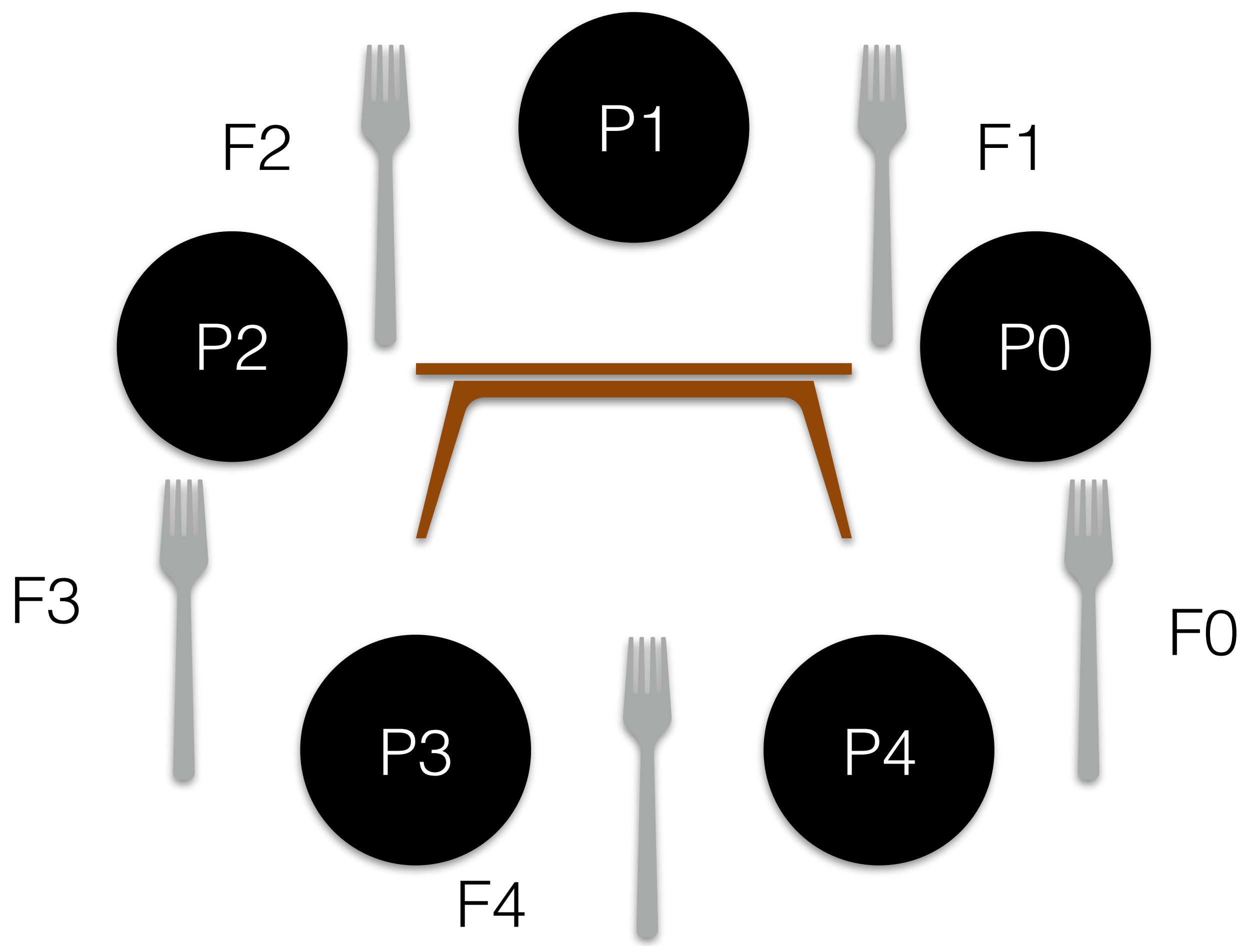
Dining Philosopher's Problem



- 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?

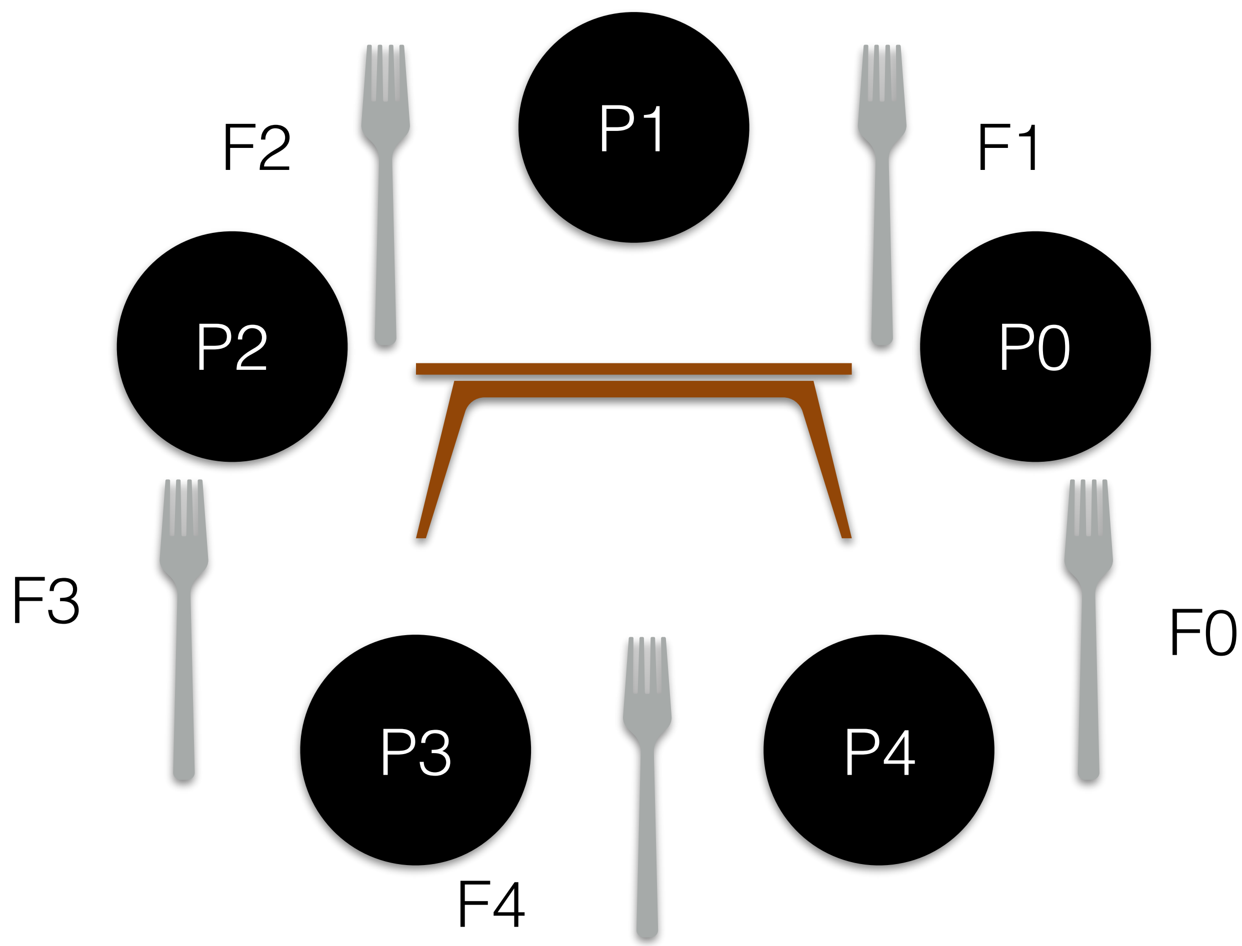
Dining Philosopher’s Problem



- 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?

Concurrency Bugs — Deadlock Dependency Graphs

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