Operating Systems
Semaphores + Deadlocks

Nipun Batra

Exercise: Build a lock using semaphores

```
1   sem_t m;
2   sem_init(&m, 0, 1);
3
4   sem_wait(&m);
5   //critical section here
6   sem_post(&m);
```

Refresher Notes

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

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

join() using CVs

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

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

join() using CVs

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

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

7   int main(int argc, char *argv[]) {
8       printf("parent: begin\n");
9       pthread_t c;
10        Pthread_create(&c, NULL, child, NULL); // create child
11        thread_join()
12        printf("parent: end\n");
13        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
```

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

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

join() using semaphores

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

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

Refresher Notes

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

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

join() using semaphores

```
void thread_exit {

}
```

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

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

Refresher Notes

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

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

join() using semaphores

```
void thread_exit {

}
```

```
void thread_join {

}
```

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

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

Refresher Notes

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

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

join() using semaphores

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

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

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

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

Refresher Notes

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

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

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

```
1   int buffer[MAX];
2   int fill = 0;
3   int use = 0;
4
5   void put(int value) {
6     buffer[fill] = value;   // line f1
7     fill = (fill + 1) % MAX;   // line f2
8   }
9
10  int get() {
11    int tmp = buffer[use];   // line g1
12    use = (use + 1) % MAX;   // line g2
13    return tmp;
14  }
```

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);                    // line P2
9    sem_post(&full);       // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);          // line C1
17    tmp = get();                // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
21  int main(int argc, char *argv[]) {
22      // ...
23      sem_init(&empty,0, ?);
24      sem_init(&full, 0, ?);
25      // ...
26  }
```

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);                    // line P2
9    sem_post(&full);       // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);          // line C1
17    tmp = get();                // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
21  int main(int argc, char *argv[]) {
22      // ...
23      sem_init(&empty,0, MAX);
24      sem_init(&full, 0, 0);
25      // ...
26  }
```

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);                    // line P2
9    sem_post(&full);       // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);          // line C1
17    tmp = get();                // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
21  int main(int argc, char *argv[]) {
22      // ...
23      sem_init(&empty,0, MAX);
24      sem_init(&full, 0, 0);
25      // ...
26  }
```

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);                    // line P2
9    sem_post(&full);       // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);          // line C1
17    tmp = get();                // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
21  int main(int argc, char *argv[]) {
22      // ...
23      sem_init(&empty,0, MAX);
24      sem_init(&full, 0, 0);
25      // ...
26  }
```

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

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);                    // line P2
9    sem_post(&full);       // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);          // line C1
17    tmp = get();                // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
21  int main(int argc, char *argv[]) {
22      // ...
23      sem_init(&empty,0, MAX);
24      sem_init(&full, 0, 0);
25      // ...
26  }
```

- 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

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
21  int main(int argc, char *argv[]) {
22      // ...
23      sem_init(&empty,0, MAX);
24      sem_init(&full, 0, 0);
25      // ...
26  }
```

- 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

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

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

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
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 Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
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

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
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

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
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

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
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

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

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

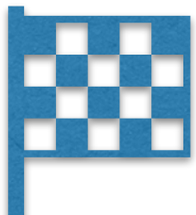
- Producer 1 (Pa) is on P2
- Producer 2 (Pb) is on P2 almost at the same time
- Let's assume fill is 10
- Pa wants to write 20, Pb wants to write 40
- Pa executes line F1, Buffer[10] is now 20
- Before Pa executes F2, Pb executes F1. Buffer[10] is now 40

Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21  }
22  ...
```

```
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

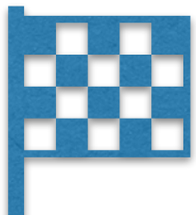


Producer Consumer Problem using Semaphores

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5    int i;
6    for (i = 0; i < loops; i++) {
7    sem_wait(&empty);   // line P1
8    put(i);             // line P2
9    sem_post(&full);    // line P3
10    }
11  }
12
13  void *consumer(void *arg) {
14    int i, tmp = 0;
15    while (tmp != -1) {
16    sem_wait(&full);        // line C1
17    tmp = get();            // line C2
18    sem_post(&empty);   // line C3
19    printf("%d\n", tmp);
20    }
21 }
22 ...
```

```
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



Race condition

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

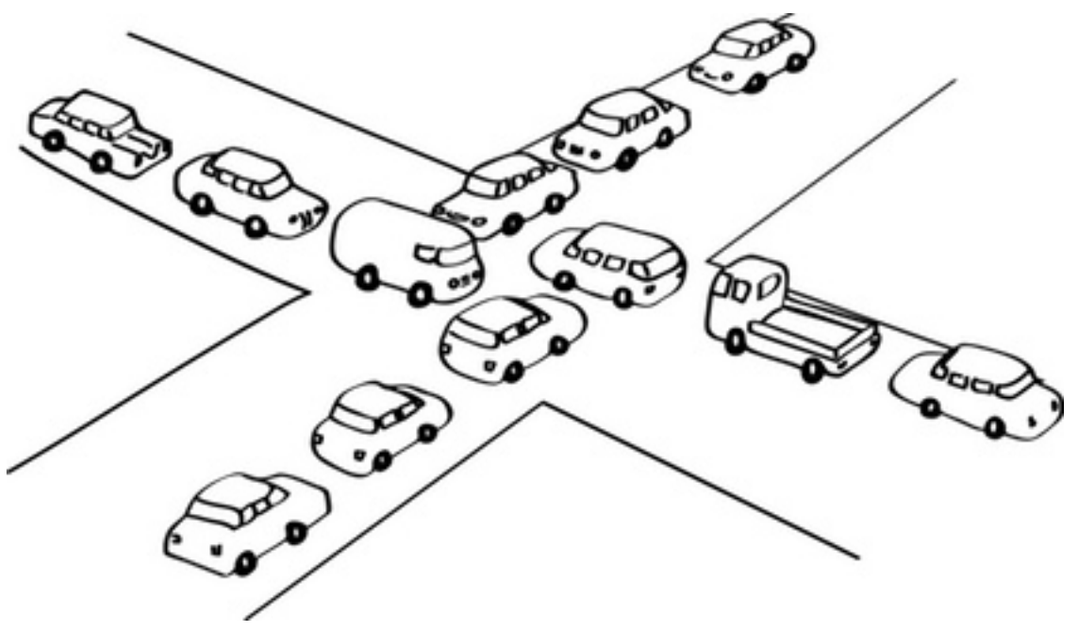
Producer Consumer Problem using Semaphores Adding mutual exclusion

```
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.

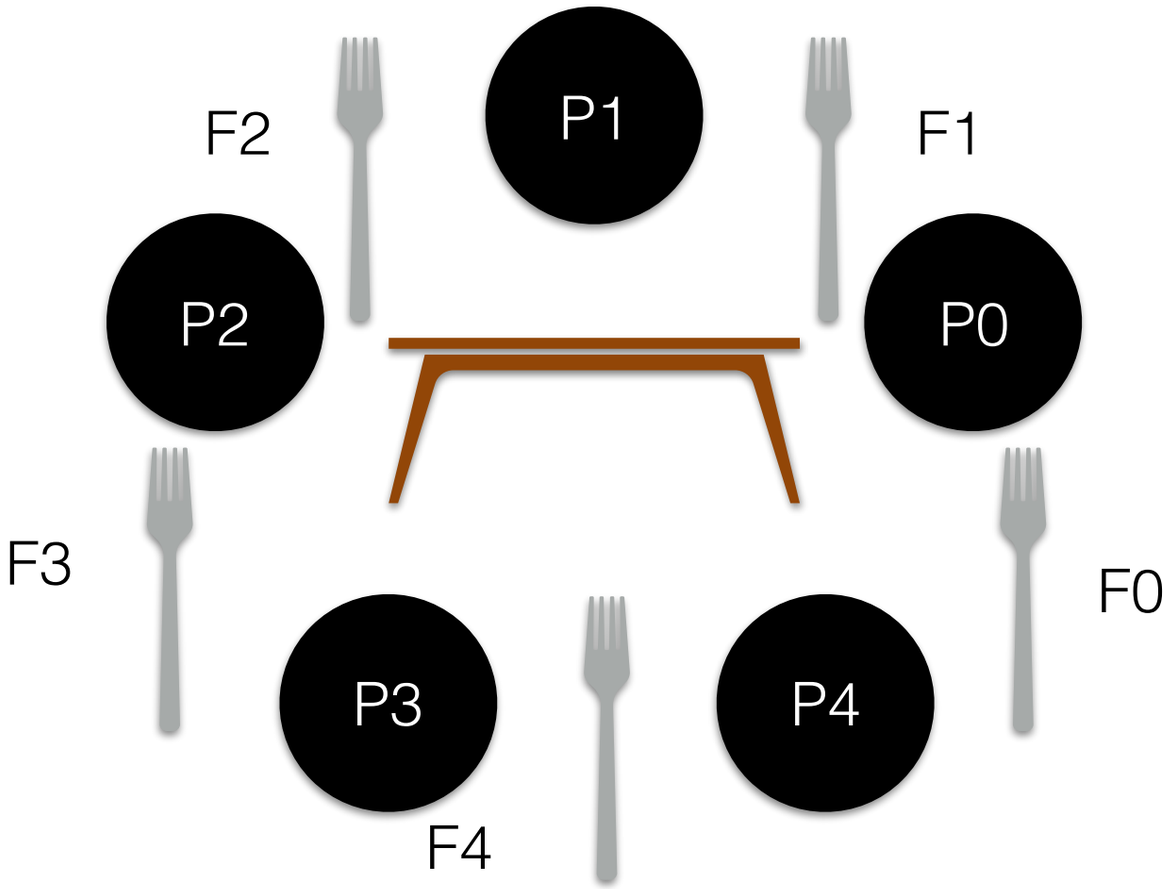
Producer Consumer Problem using Semaphores Correct Solution

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

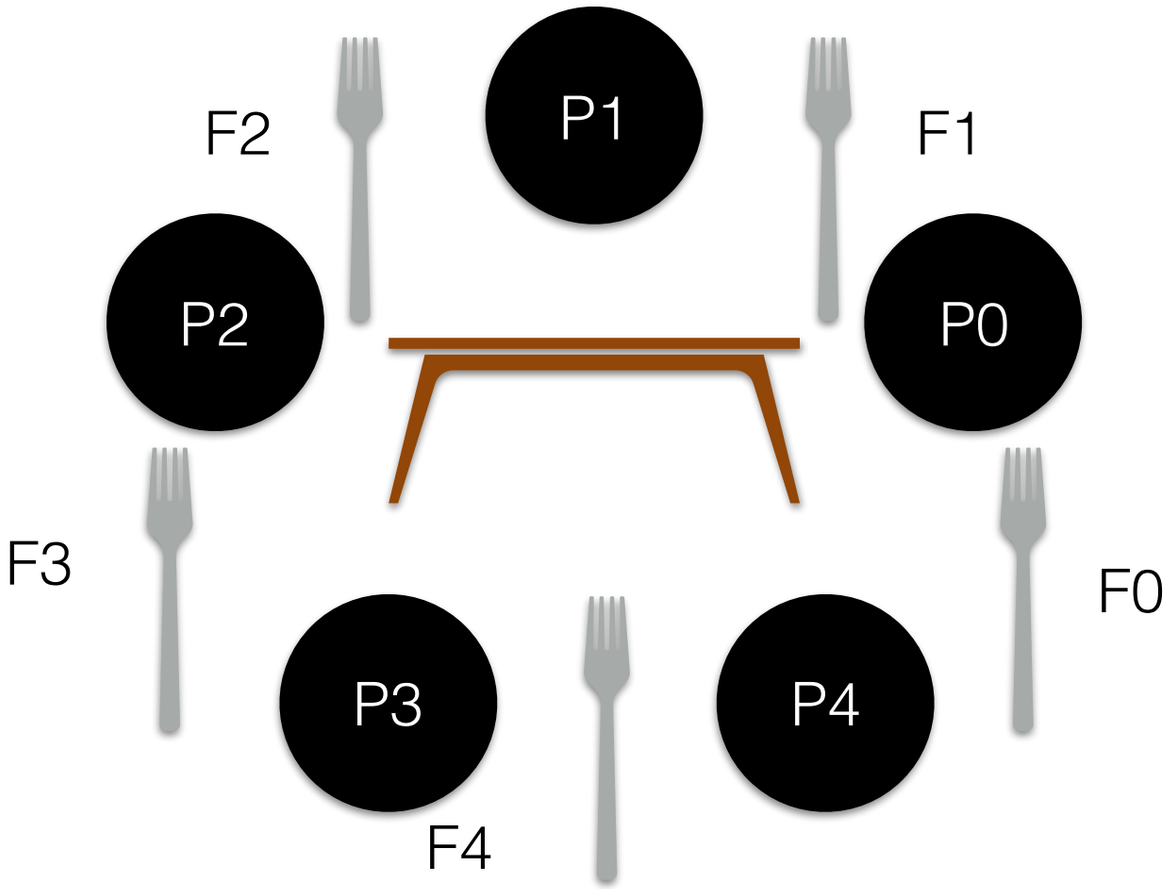
```
5   void *producer(void *arg) {
6     int i;
7    for (i = 0; i < loops; i++) {
8    sem_wait(&empty);   // line p1
9    sem_wait(&mutex);
10     put(i);   // line p2
11     sem_post(&mutex);
12     sem_post(&full);   // line p3
13     }
14  }
15
```

```
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

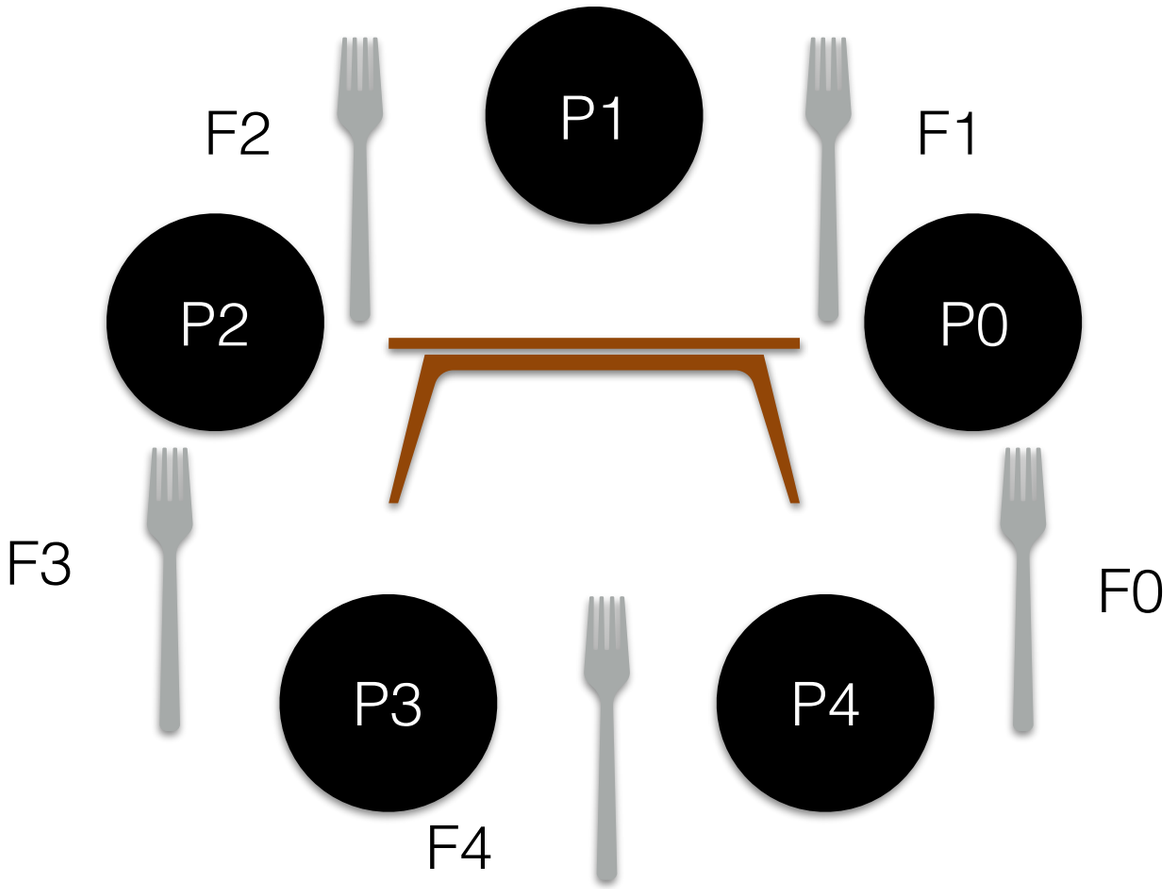


Dining Philosopher's Problem



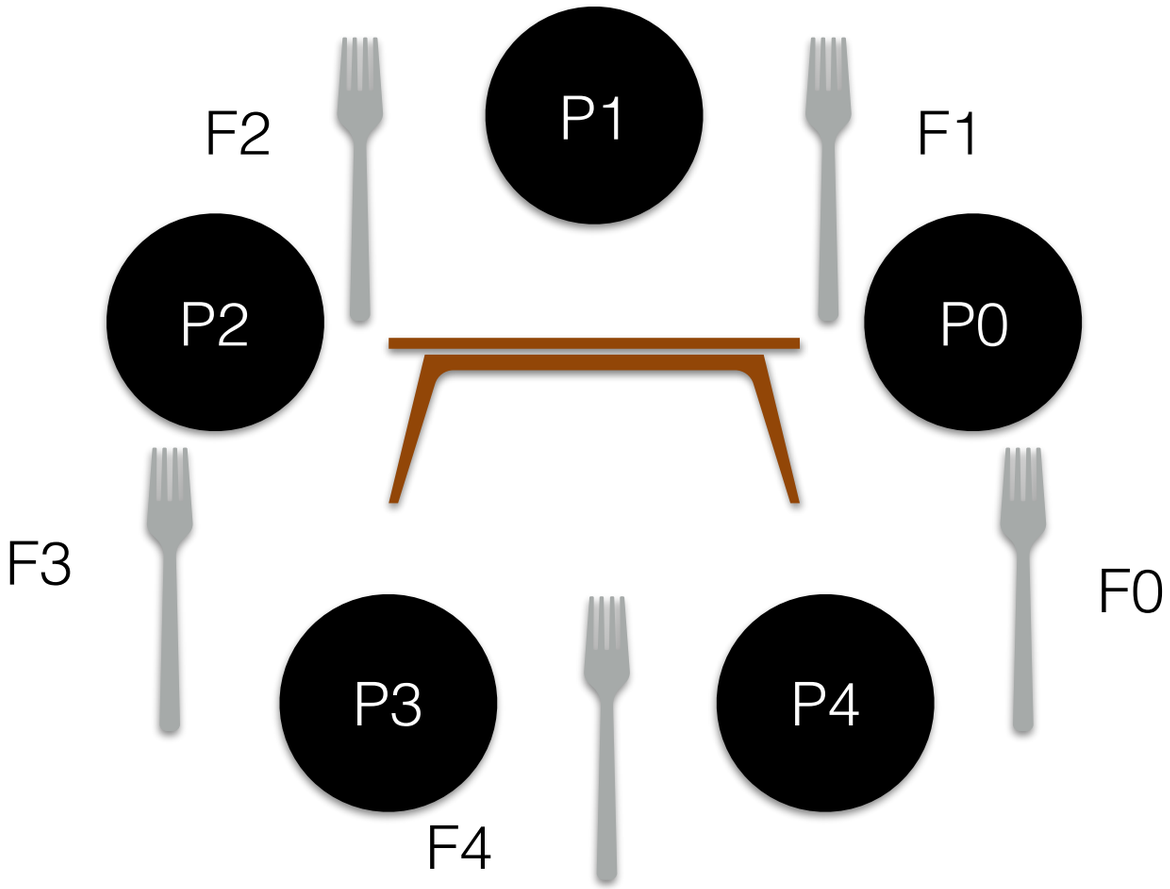
- 5 philosophers sitting around a table

Dining Philosopher's Problem



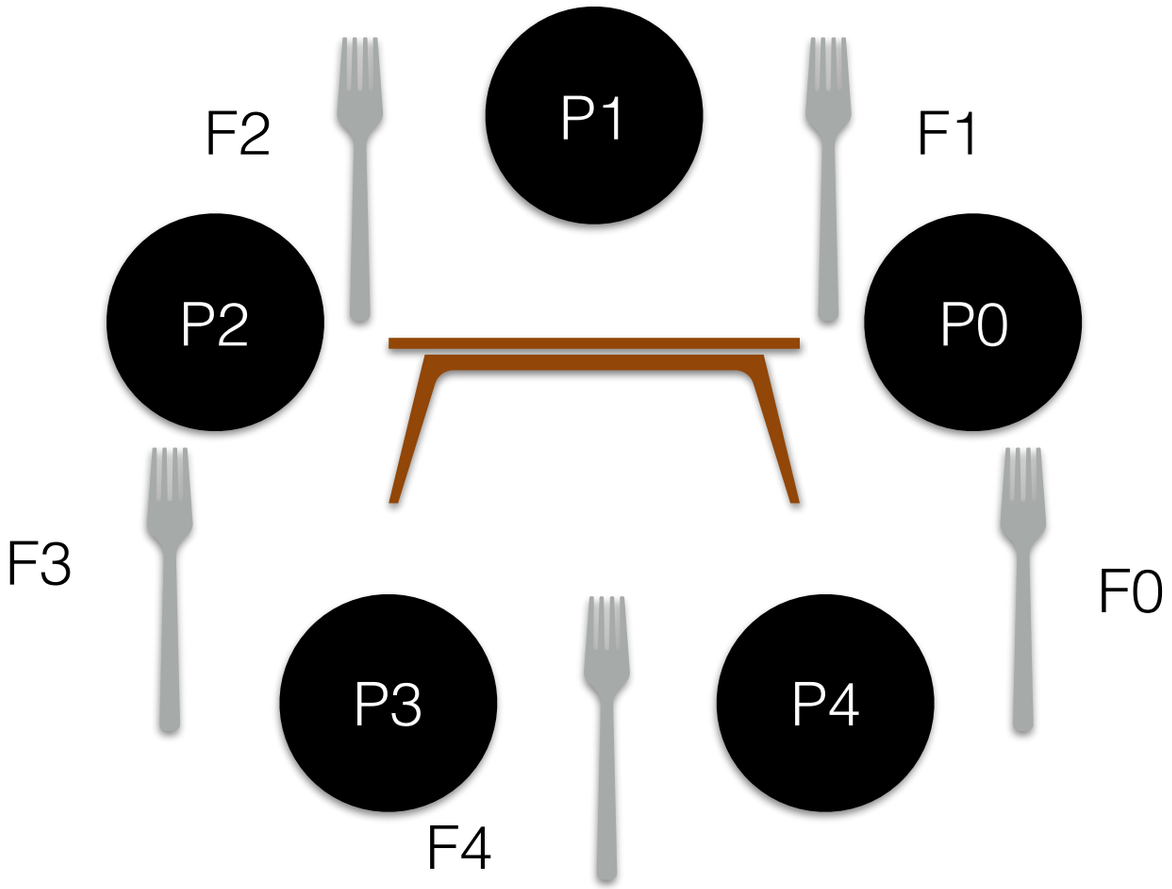
- 5 philosophers sitting around a table
- A fork between a pair of philosophers

Dining Philosopher's Problem



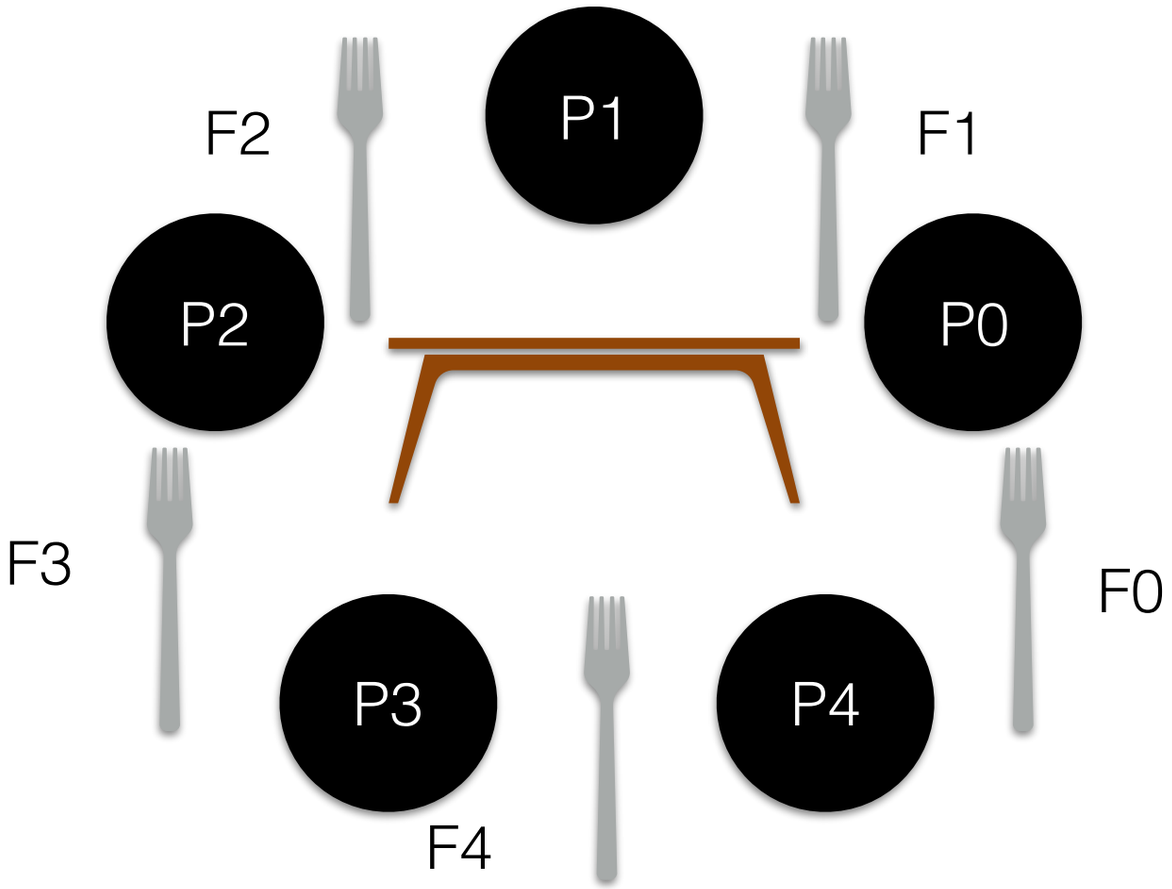
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:

Dining Philosopher's Problem



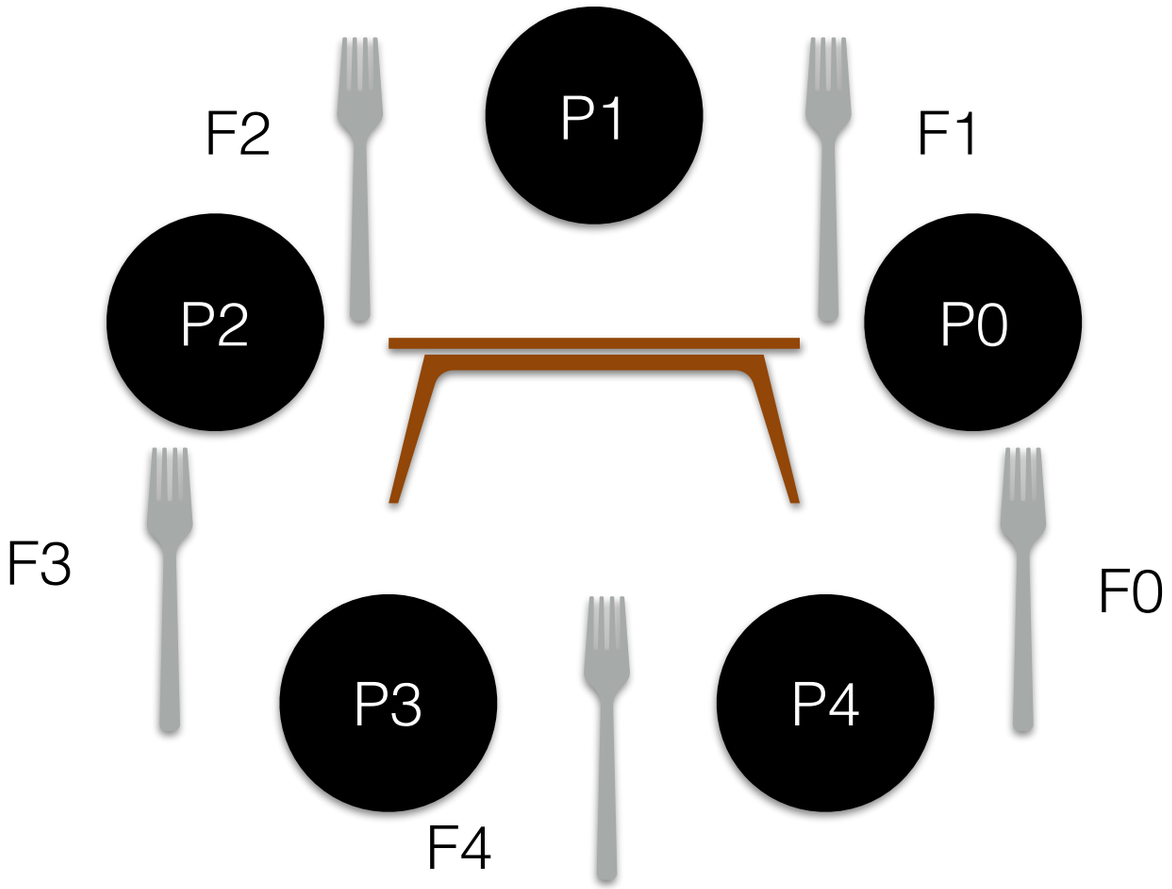
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork

Dining Philosopher's Problem



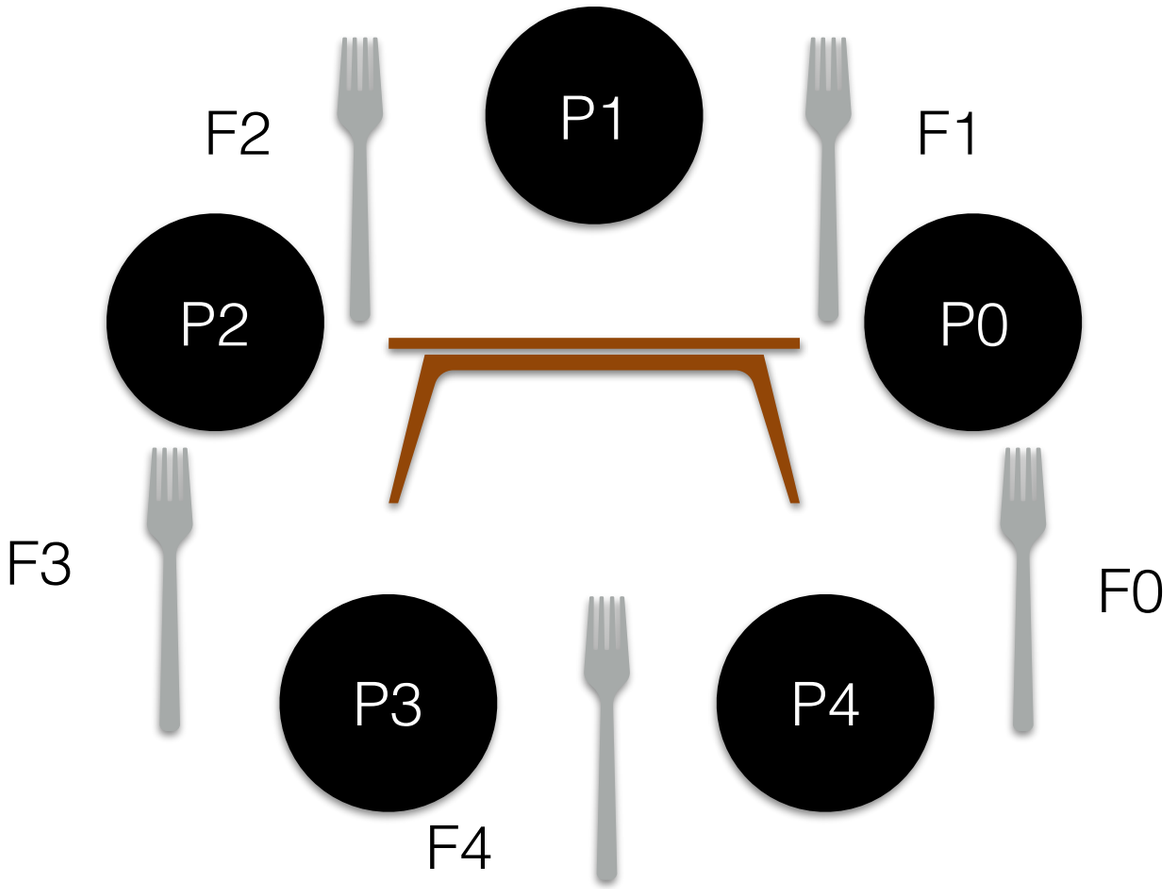
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right

Dining Philosopher's Problem



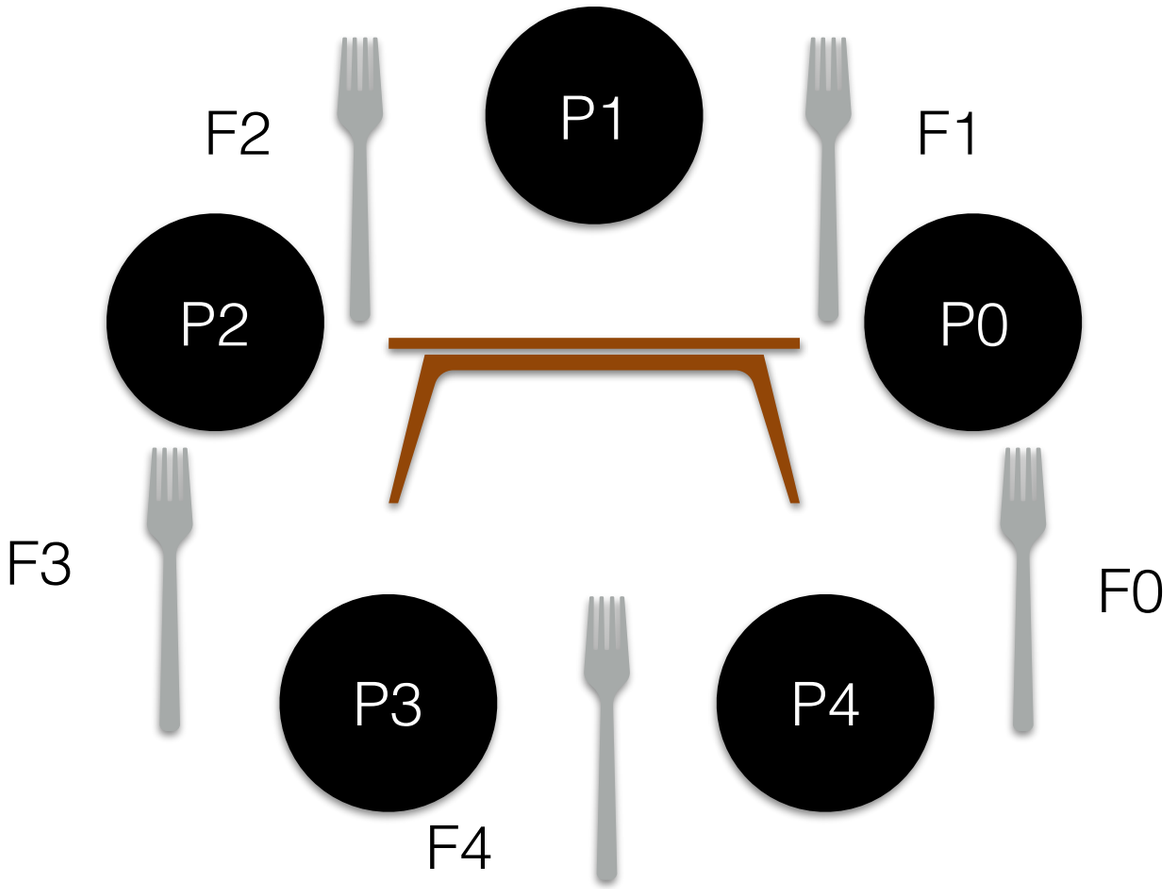
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
- Forks have contention

Dining Philosopher's Problem



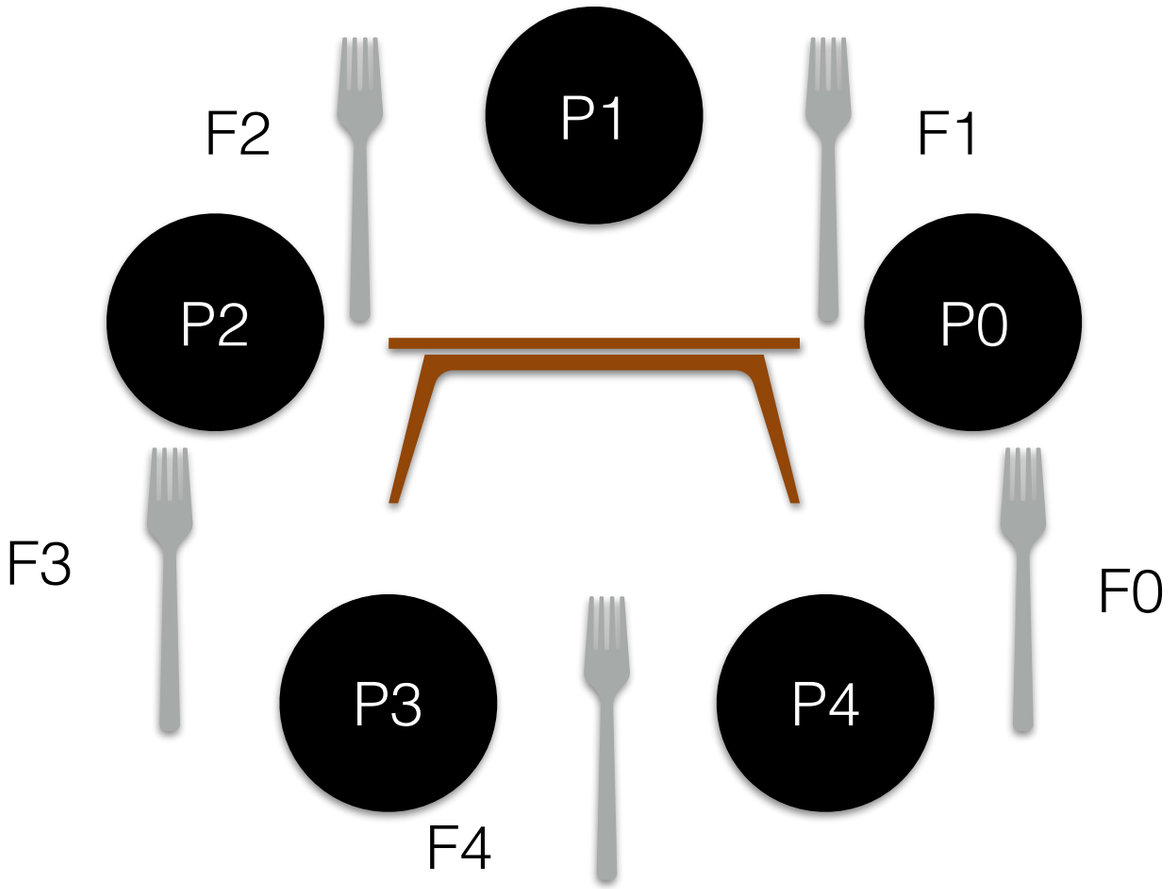
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:

Dining Philosopher's Problem



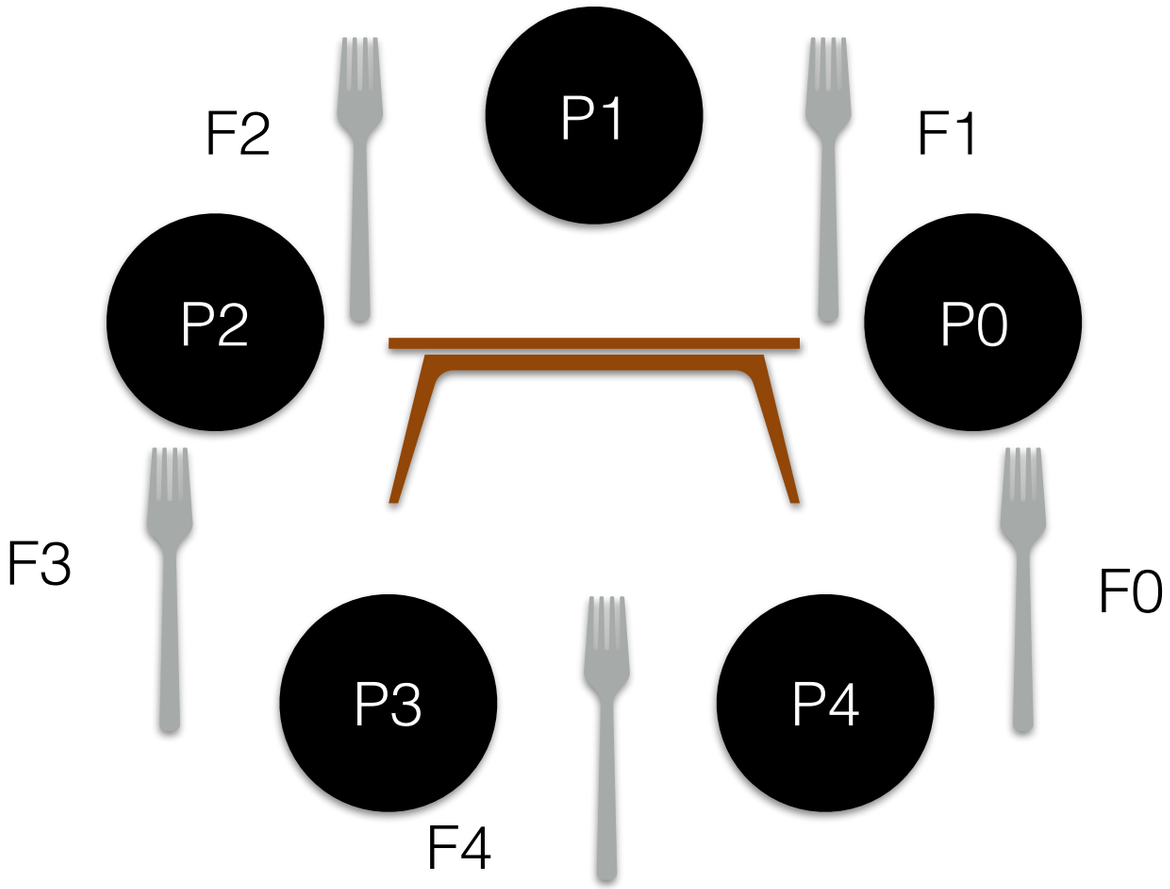
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:
  - No deadlock

Dining Philosopher's Problem



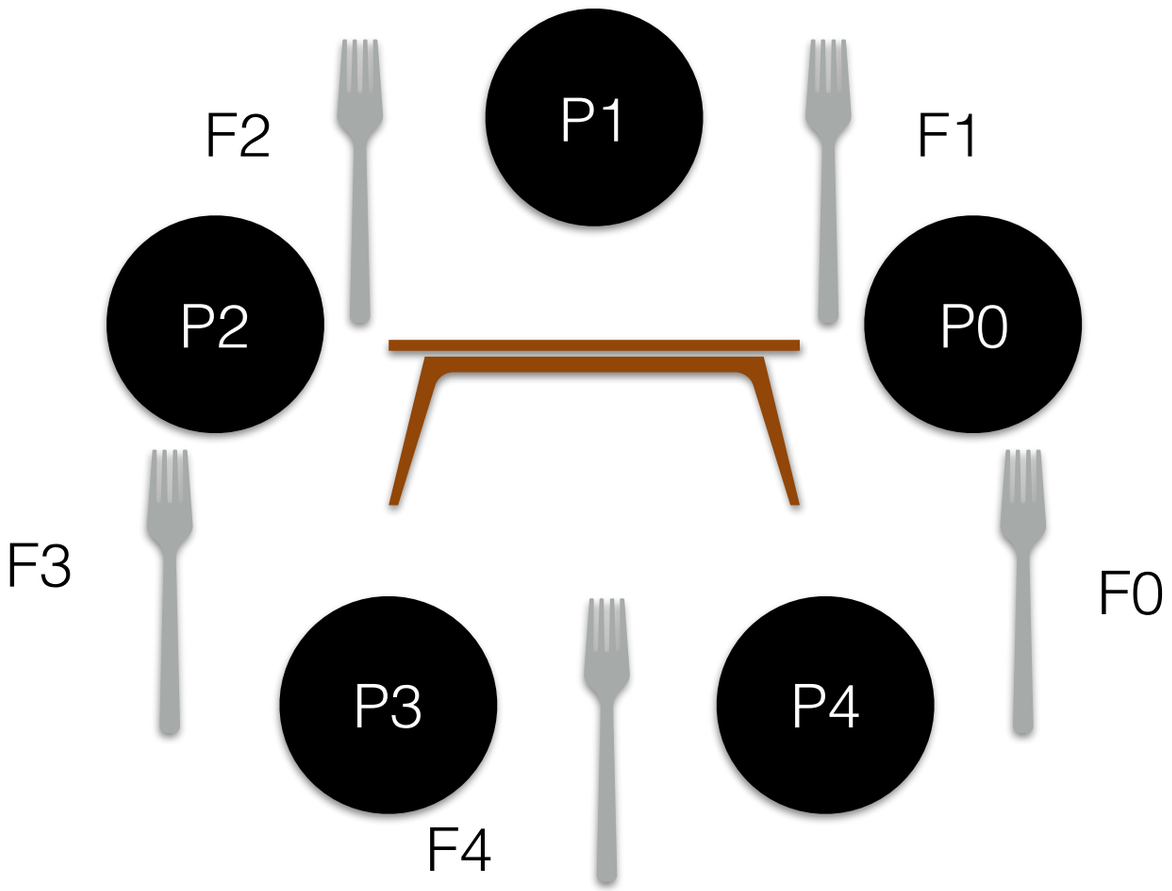
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:
  - No deadlock
  - No one starves

Dining Philosopher's Problem



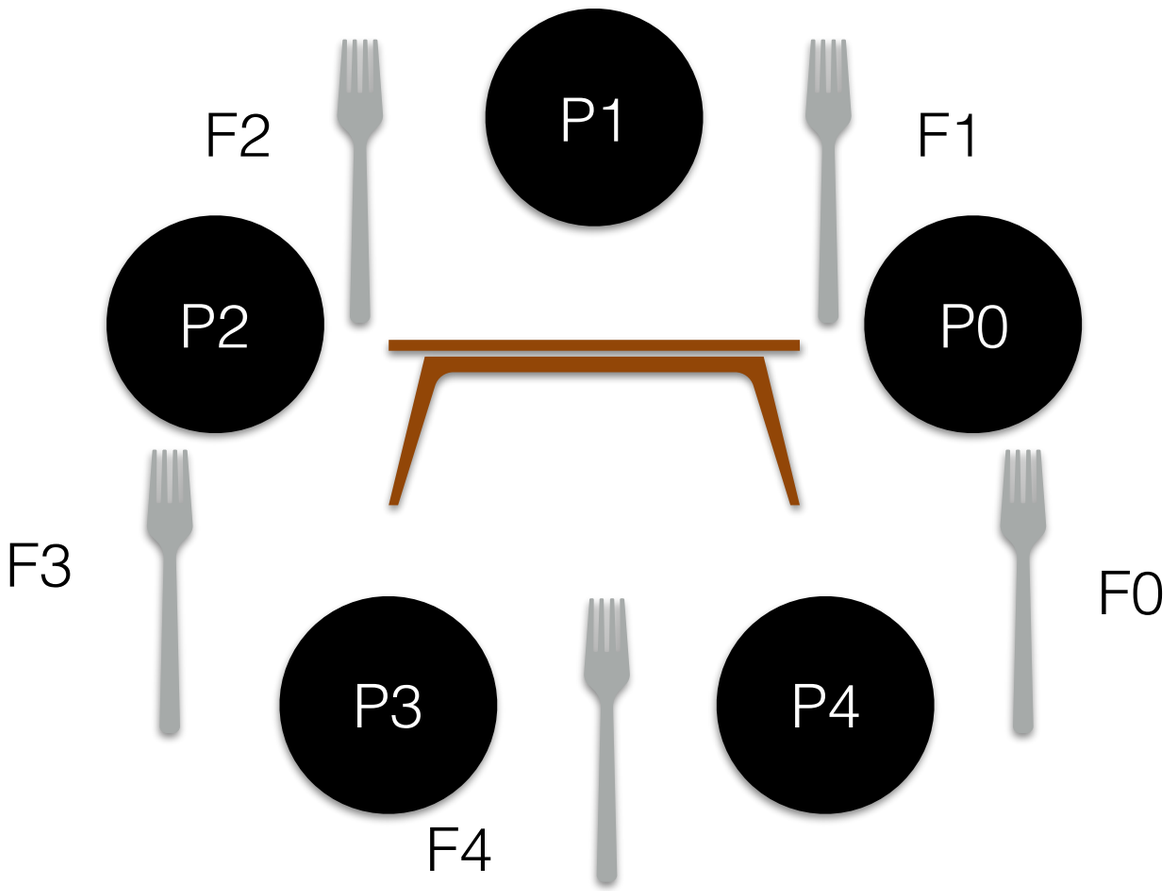
- 5 philosophers sitting around a table
- A fork between a pair of philosophers
- Philosopher's activities:
  - Think — don't need fork
  - Eat — need fork on left and right
  - Forks have contention
- Challenges:
  - No deadlock
  - No one starves
  - High Concurrency

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;
}
```

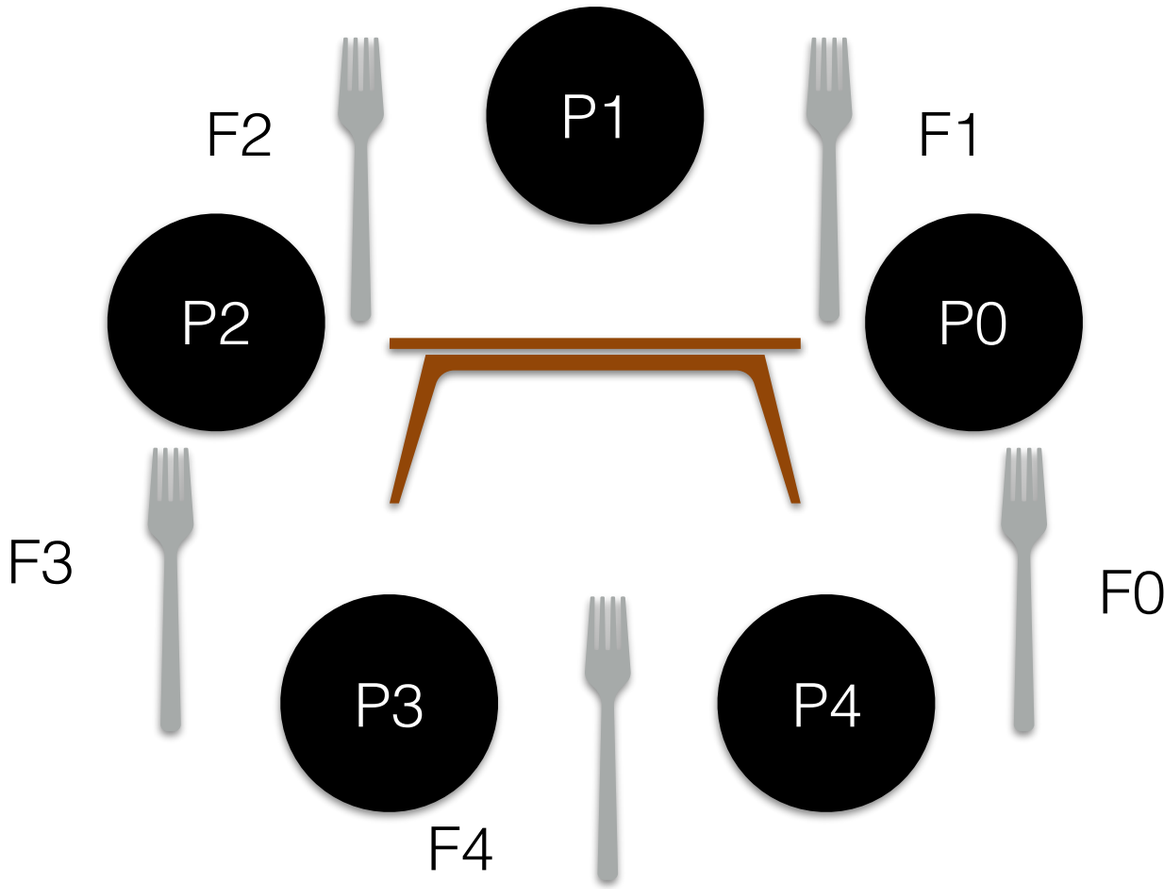
Dining Philosopher's Problem



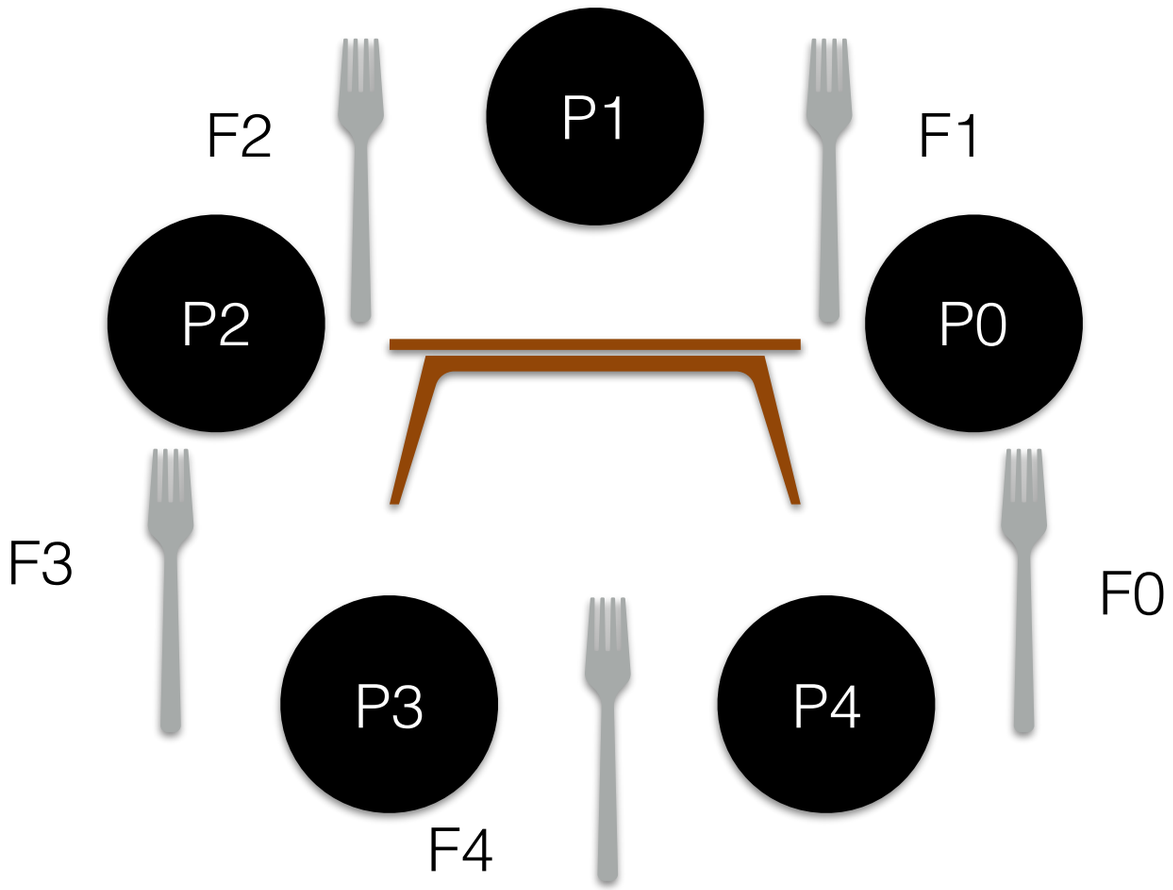

```
while (1) {
  think();
  getforks();
  eat();
  putforks();
}

// 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

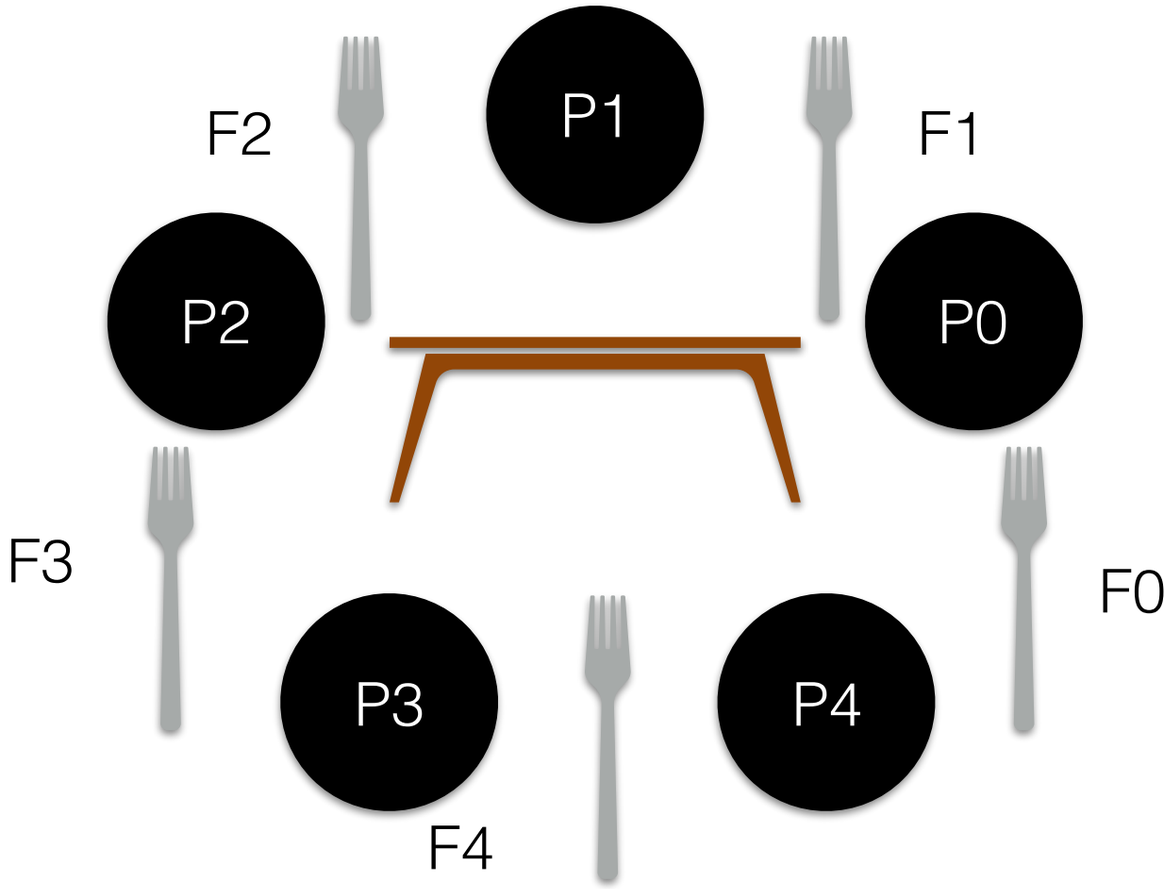


Dining Philosopher's Problem




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

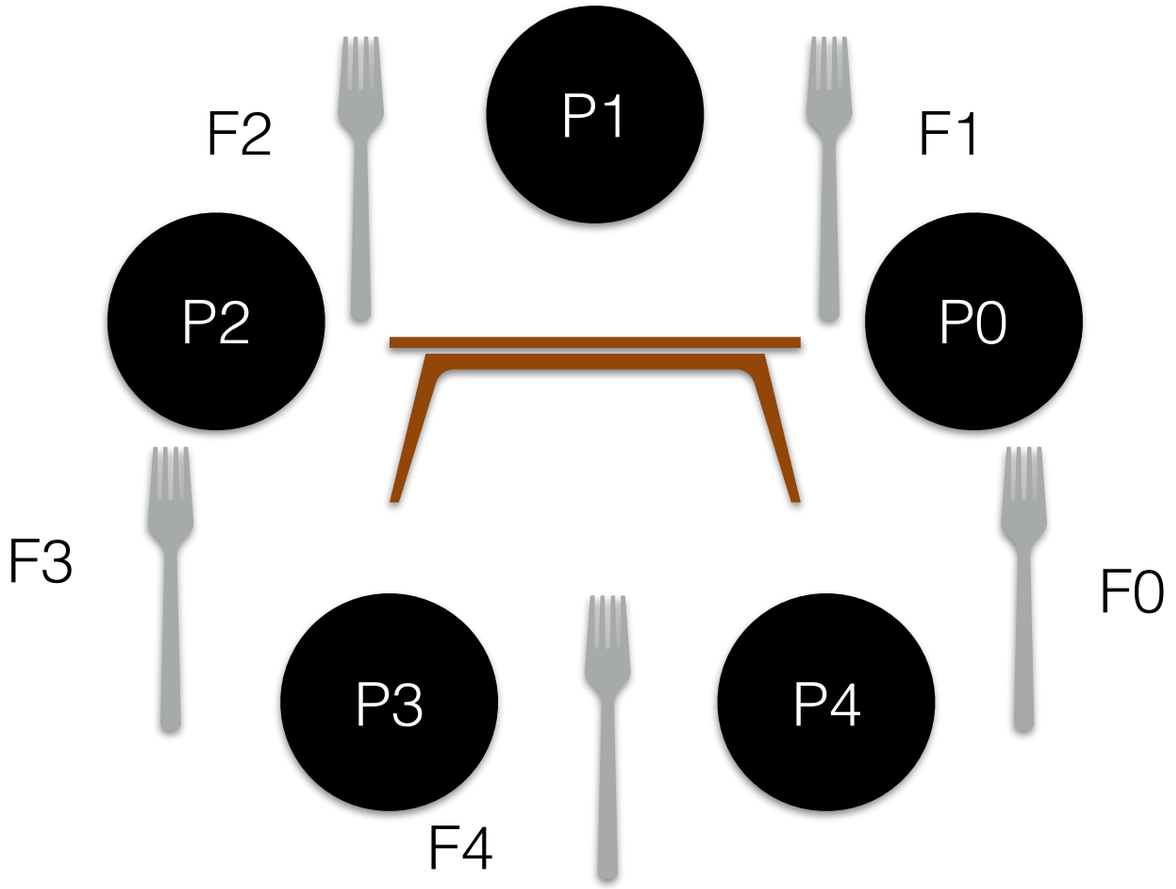
Dining Philosopher's Problem




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

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

Dining Philosopher's Problem

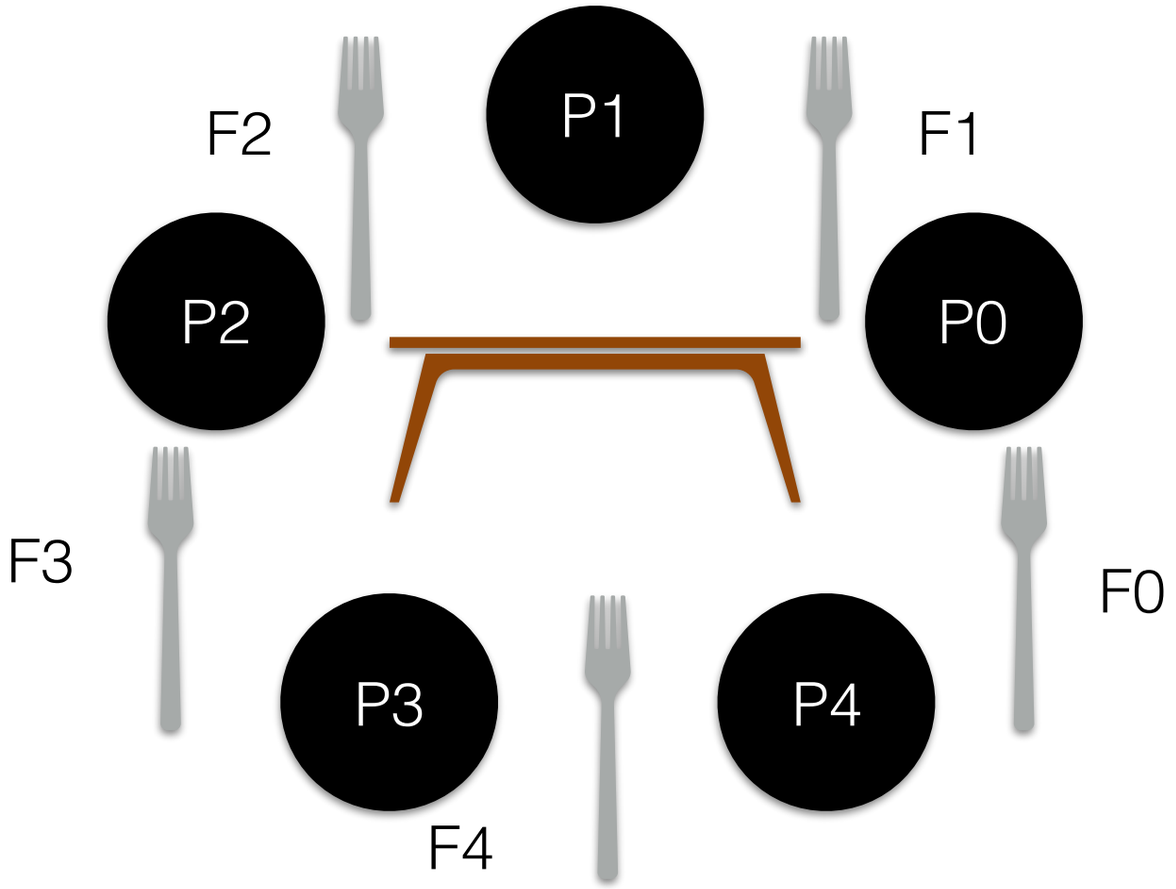



```
1 void getforks() {
2   sem_wait(forks[left(p)]);
3   sem_wait(forks[right(p)]);
4 }
```

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

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

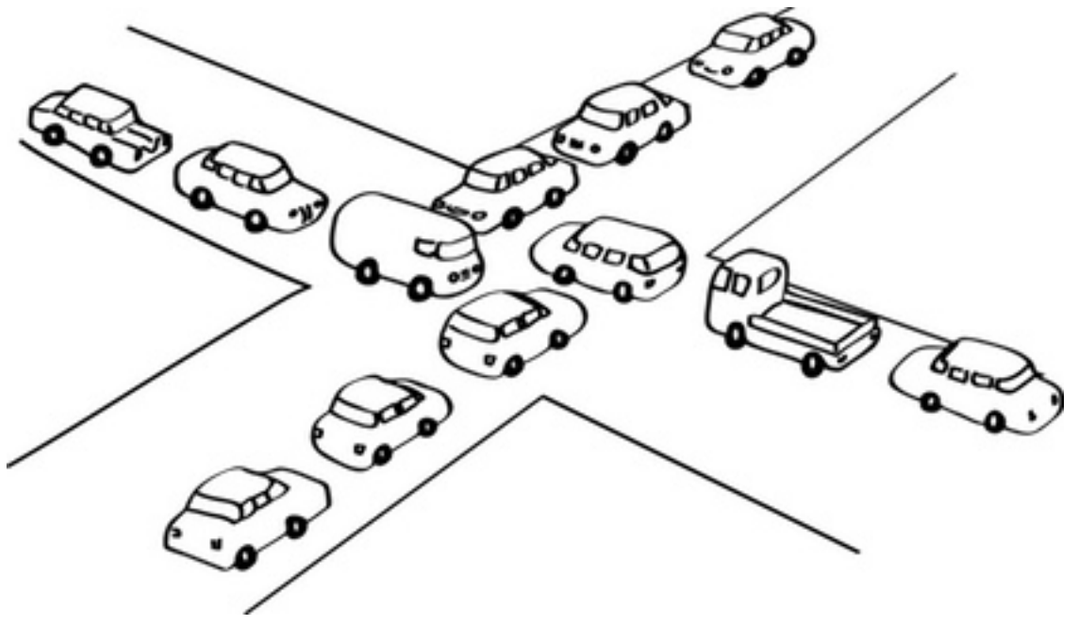
Dining Philosopher's Problem



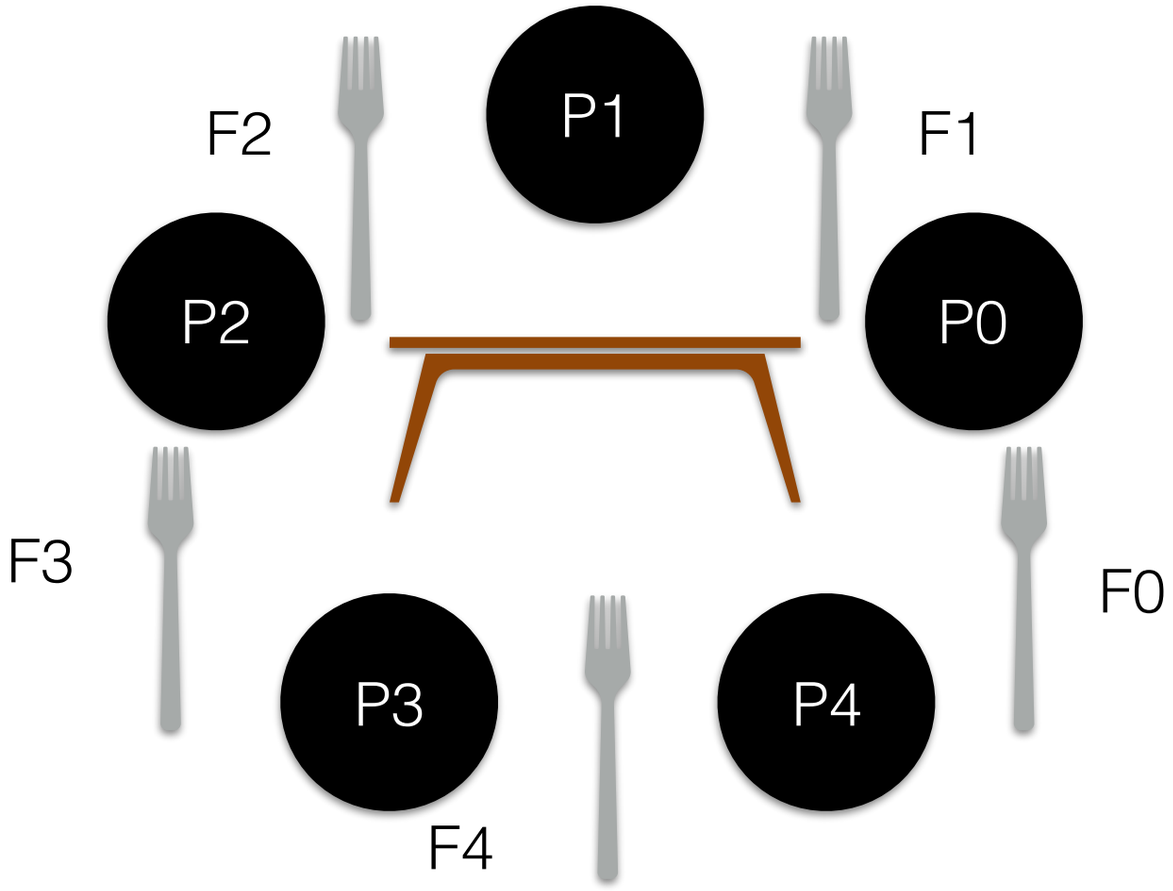

```
1 void getforks() {
2   sem_wait(forks[left(p)]);
3   sem_wait(forks[right(p)]);
4 }
```

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

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



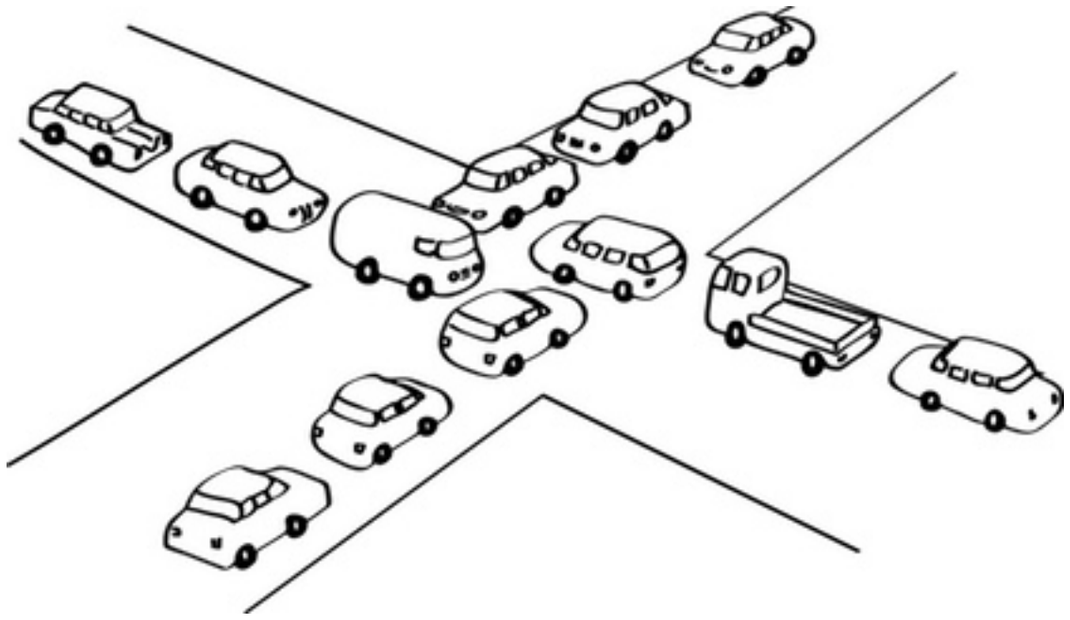
Dining Philosopher's Problem



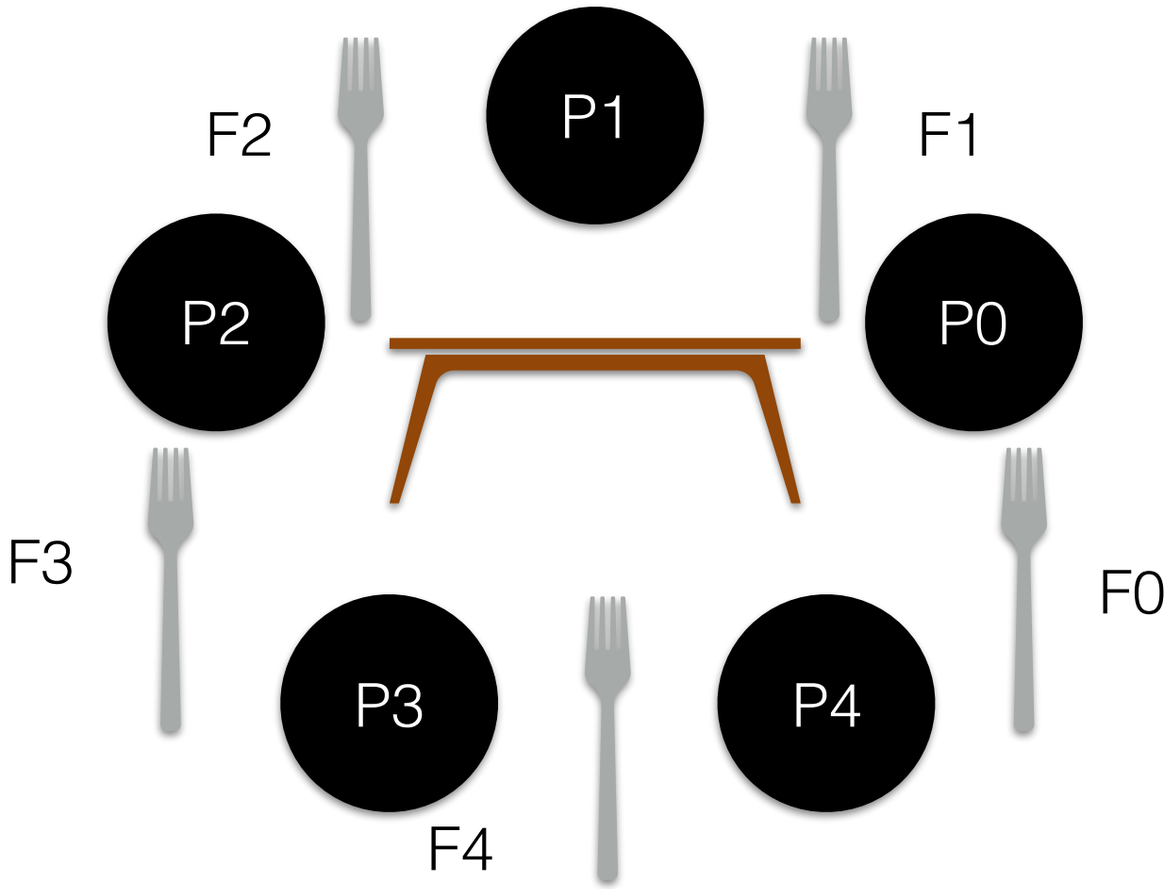

```
1 void getforks() {
2   sem_wait(forks[left(p)]);
3   sem_wait(forks[right(p)]);
4 }
```

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

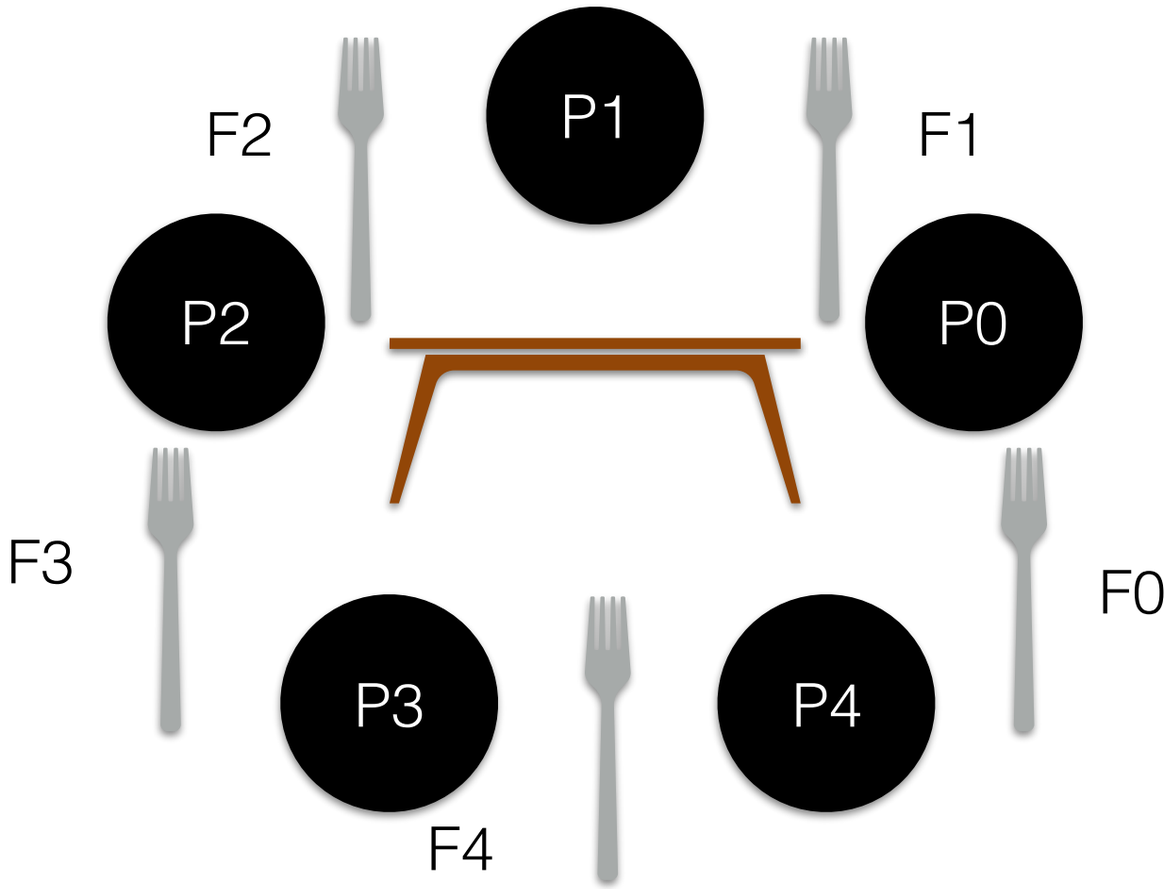
- P0 picks F0; P1 picks F1; …, P4 picks F4
- Change something in above code to avoid deadlock. Hint: Maybe some philosopher should break the order of picking up forks?



Dining Philosopher's Problem

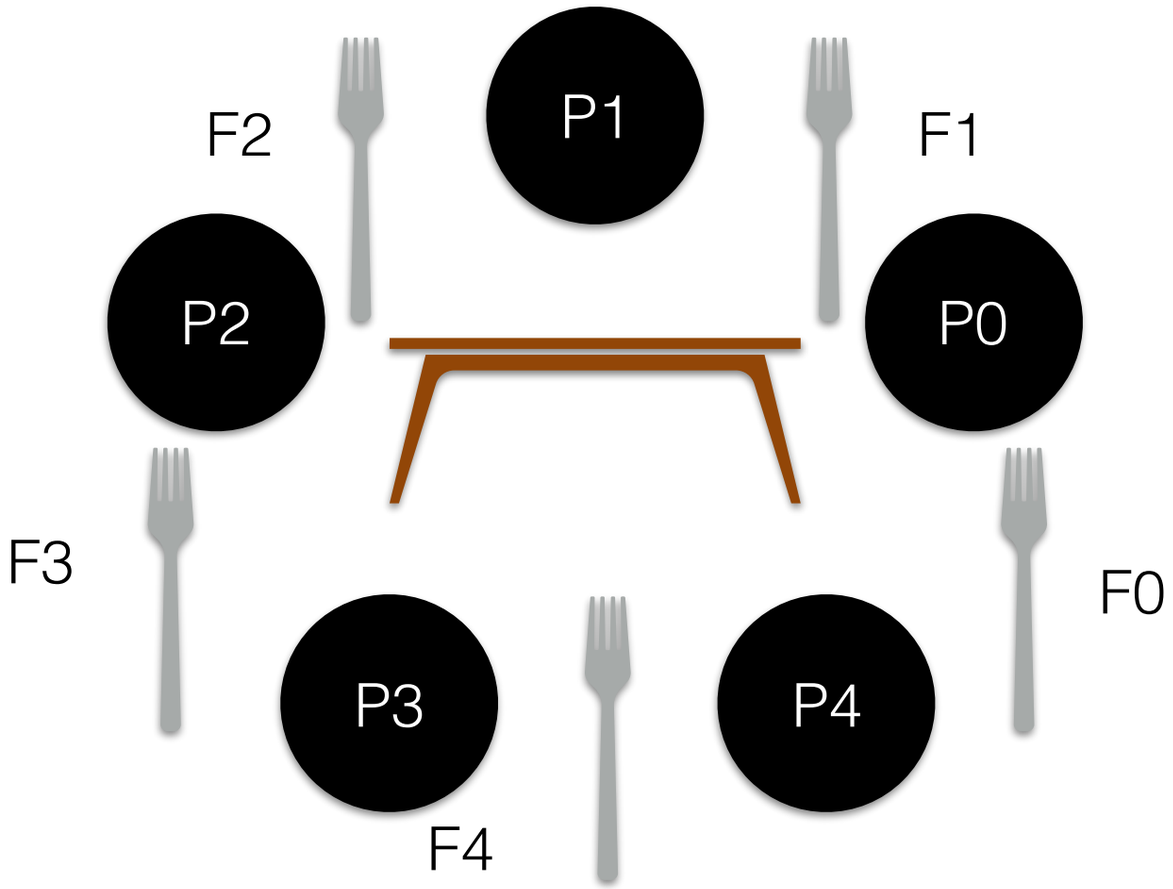


Dining Philosopher's Problem



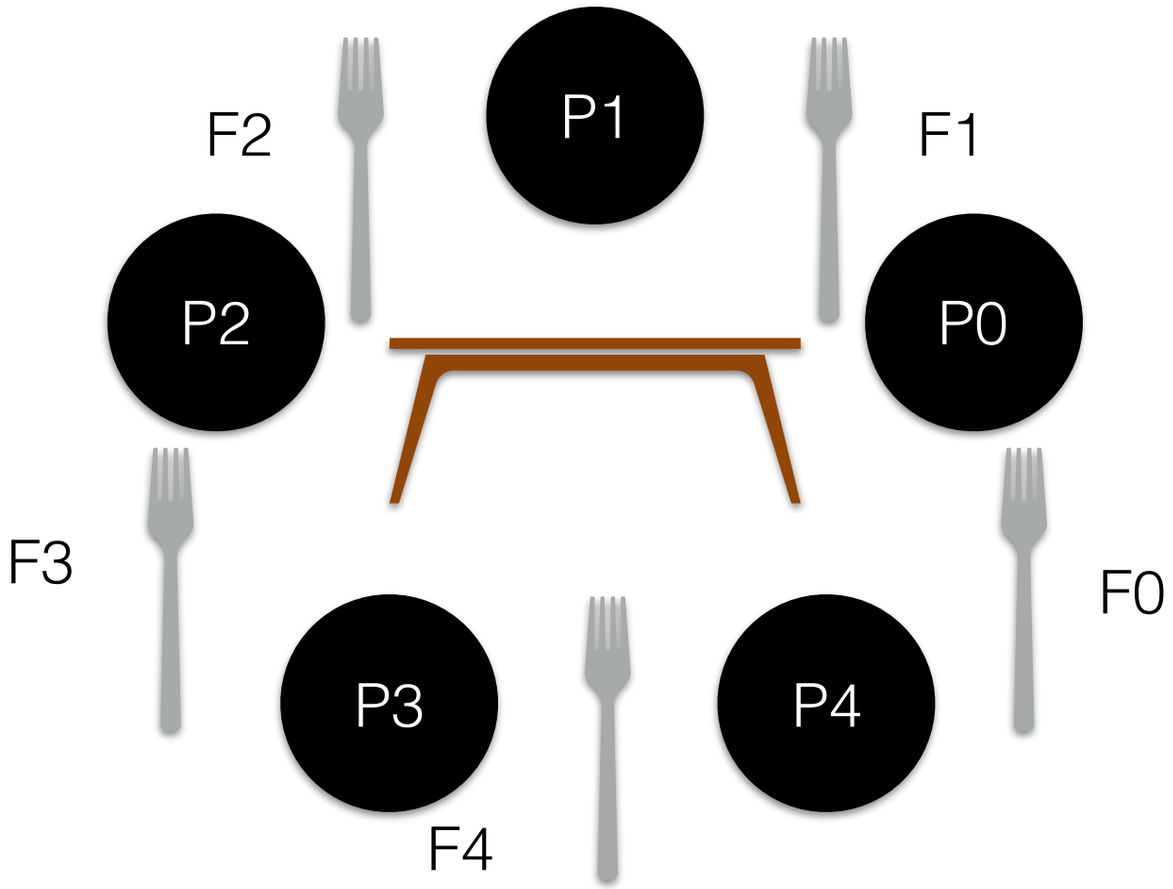
- P0 picks F0; P1 picks F1; …, P4 picks F4

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

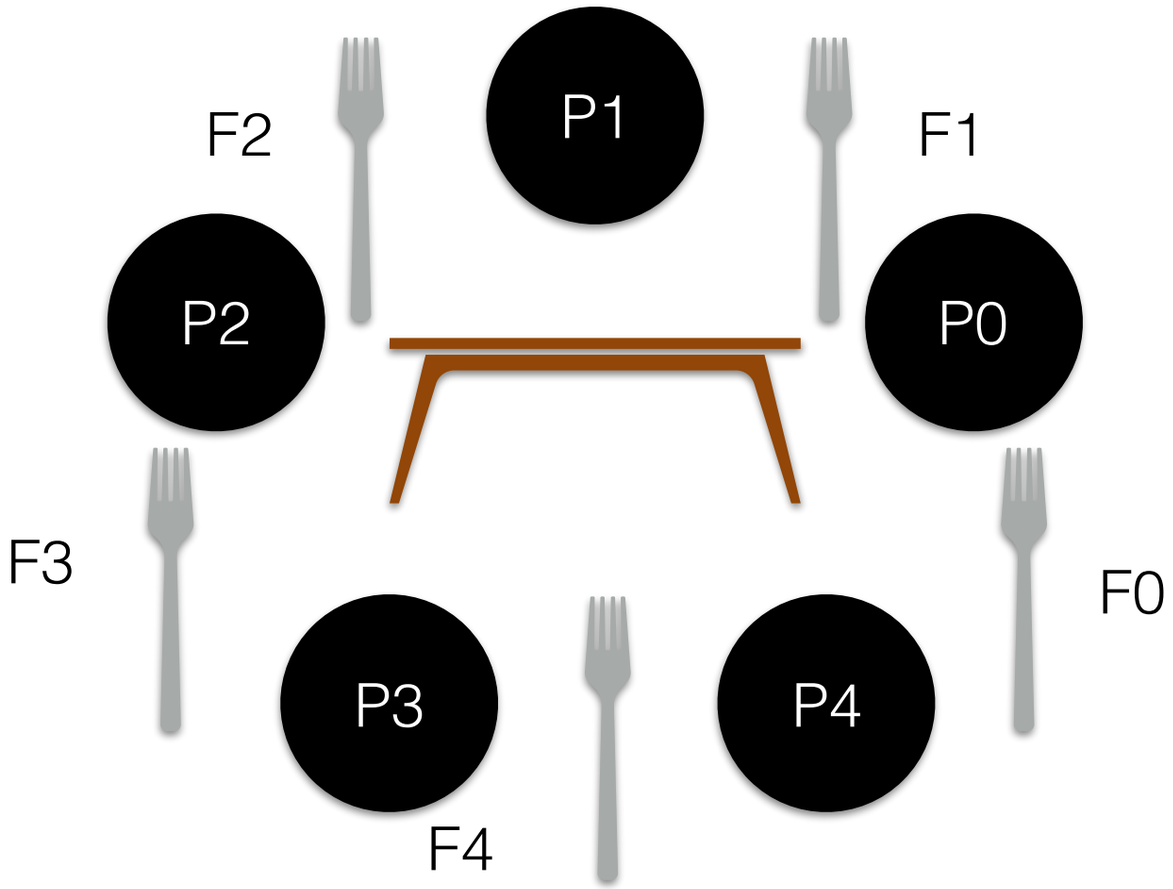


- If P ==4:

- 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

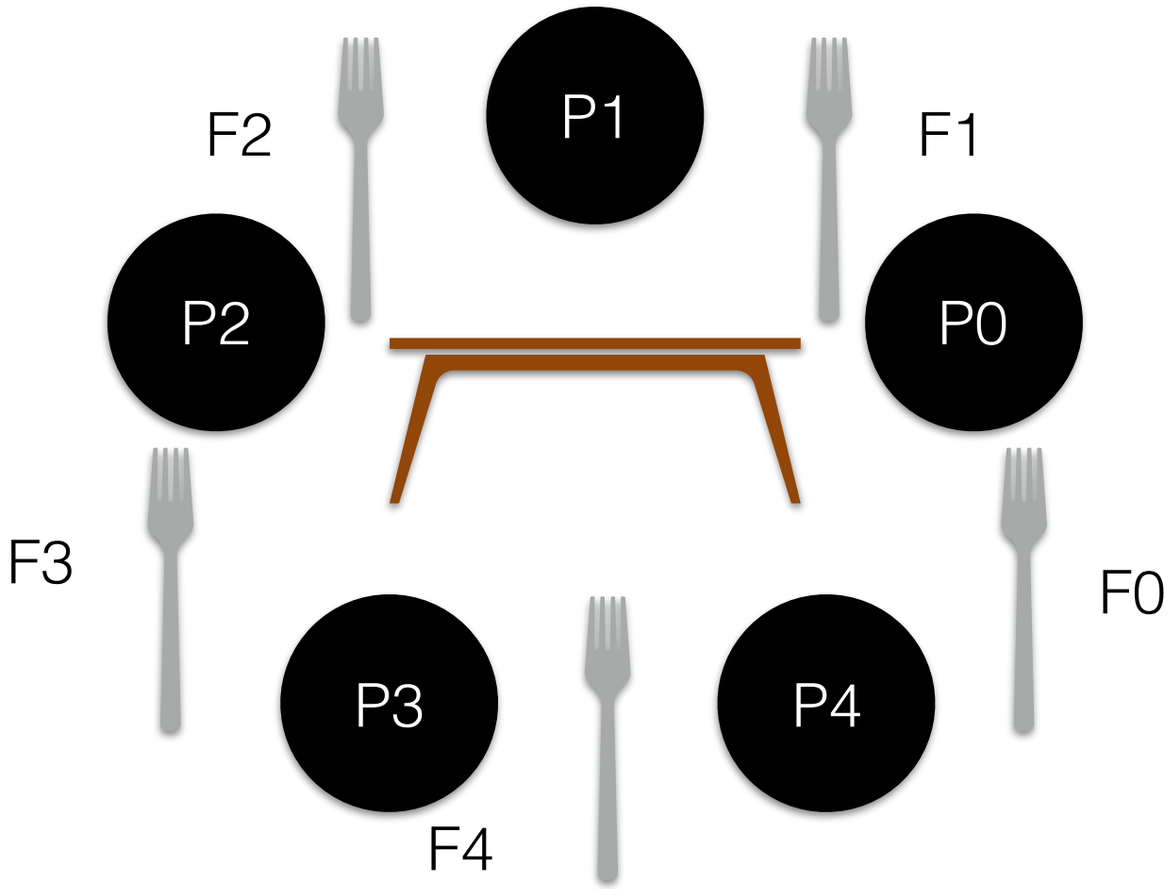


- If P ==4:
  - Wait on Right

- 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

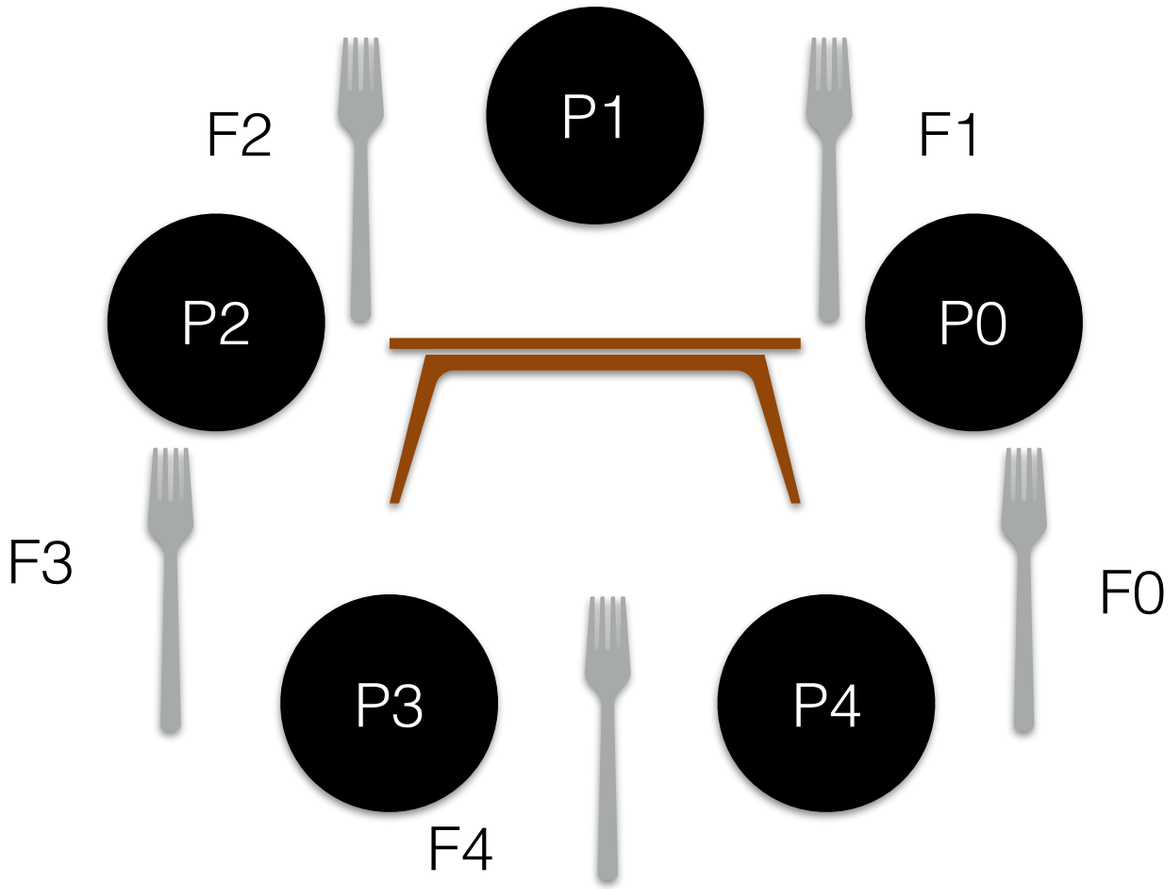


- If P ==4:
  - Wait on Right
  - Want on Left

- 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

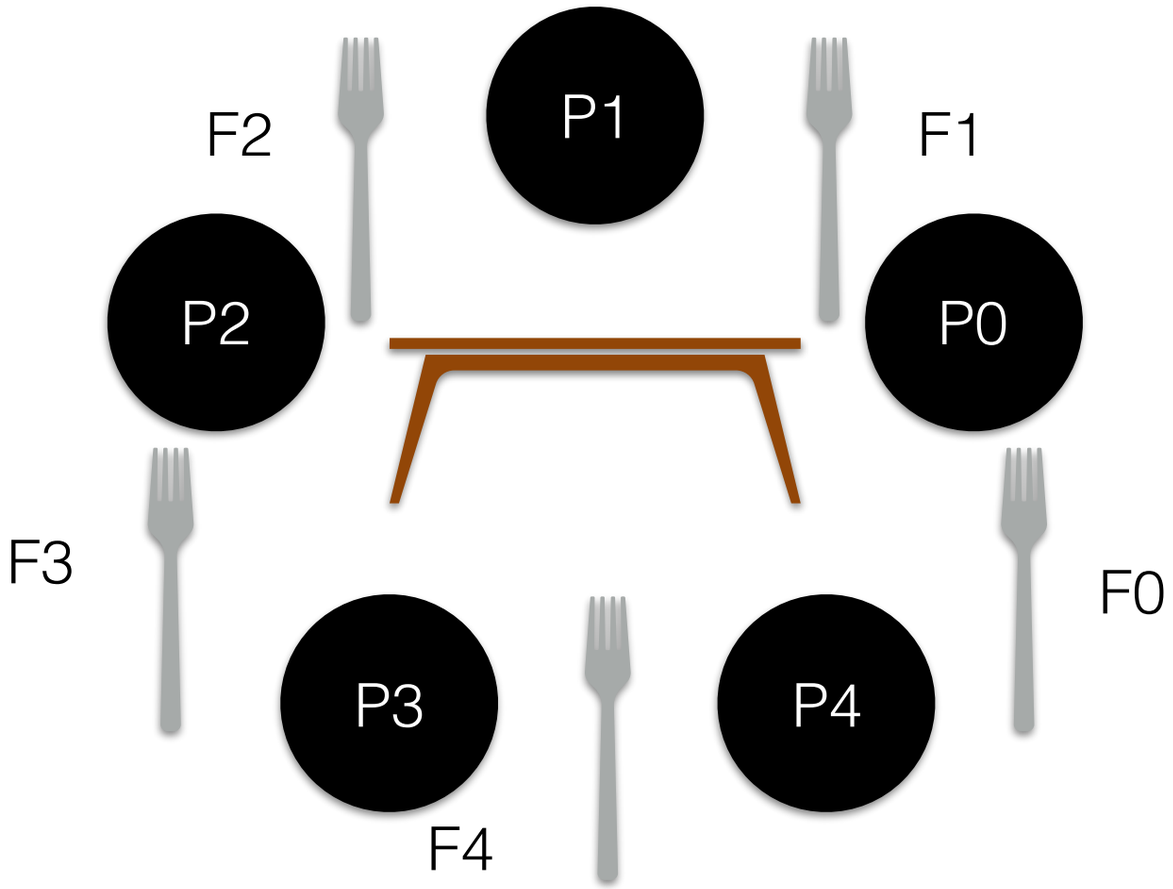


- If P ==4:
  - Wait on Right
  - Want on Left
- Else:

- 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

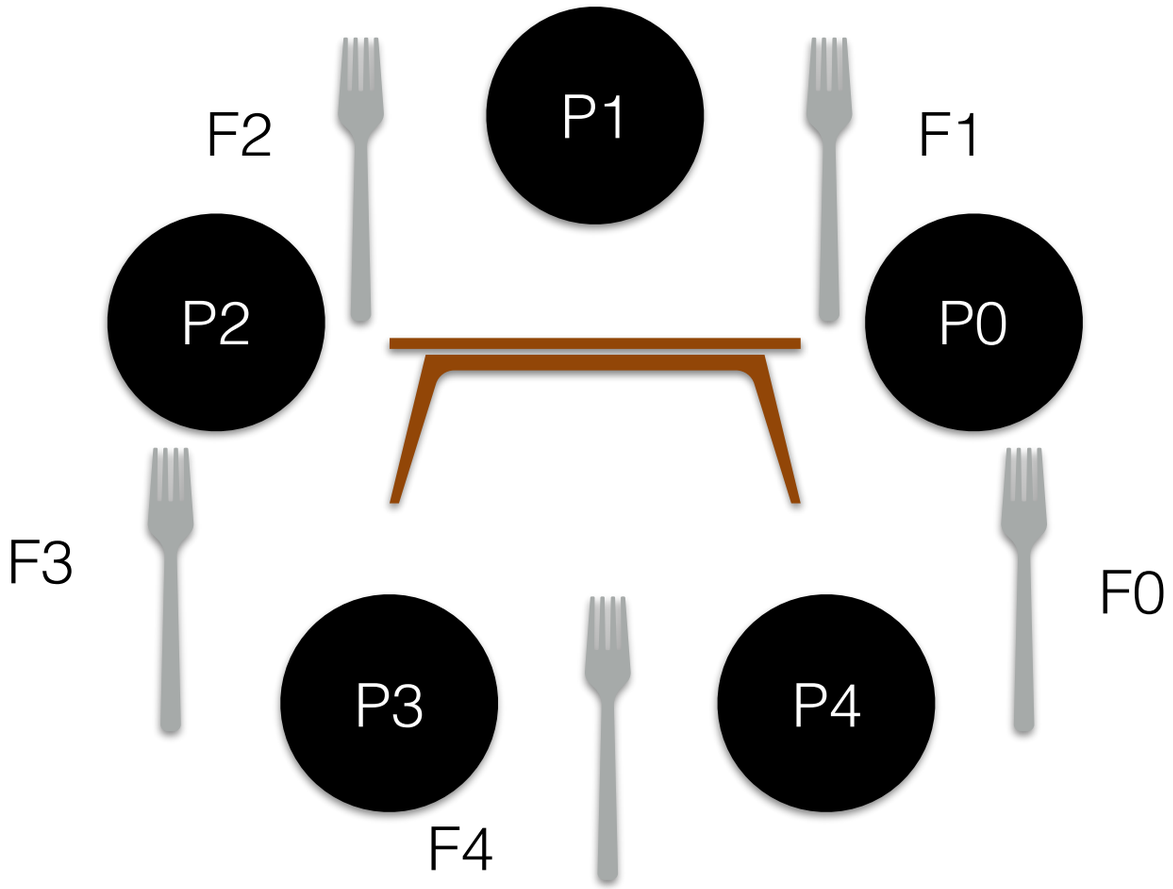


- If P ==4:
  - Wait on Right
  - Want on Left
- Else:
  - Wait on Left

- 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

Lock(L1);

Lock(L2);

Thread 2

Lock(L2);

Lock(L1);