
Semaphore

Semaphores: A Definition

- Dijkstra invented the semaphore as a single primitive for all things related to synchronization
- We can use semaphores instead of locks and condition variables
- A semaphore is an object with an integer value that we can manipulate with two routines;
 - `sem_wait()` (P)
 - `sem_post()` (V)

Initializing A Semaphore

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

initialize it to the value 1

shared between threads in the **same** process

Wait and Post

```
int sem_wait(sem_t *s) { //performed atomically
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}
```

→ = number of waiting threads

```
int sem_post(sem_t *s) { //performed atomically
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

Binary Semaphores (Locks)

```

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

```

Value of Semaphore	Thread 0	Thread 1
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: begin)	Running		Ready
0	Interrupt; Switch→T1	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem<0)→sleep	Sleeping
-1		Running	Switch→T0	Sleeping
-1	(crit sect: end)	Running		Sleeping
-1	call sem_post()	Running		Sleeping
0	increment sem	Running		Sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	Interrupt; Switch→T1	Ready		Running
0		Ready	sem_wait() returns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

goes into the sleeping state when it tries to acquire the already-held lock

Semaphores For Ordering

- Semaphore as an ordering primitive (similar to condition variables)

```
1 sem_t s;
2
3 void *
4 child(void *arg) {
5     printf("child\n");
6     sem_post(&s); //child is done
7     return NULL;
8 }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); //what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); //wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

Semaphores For Ordering

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 () returns	Running		Ready

When the parent calls sem_wait() before the child has called sem_post().

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 () returns	Running		Ready

When the child runs to completion before the parent calls sem_wait().

Producer/Consumer (Bounded Buffer) Problem

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

Line	empty	full
26	1	0
C1	1	-1 (C blocked)
P1	0	-1
P3	0	0 (C awoken)
P1	-1 (P blocked)	0
C3	0 (P awoken)	0
C1	0	-1 (C blocked)
P3	0	0 (C awoken)

```
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
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); //MAX = 1
26     sem_init(&full, 0, 0);
27     // ...
28 }
```

Adding Mutual Exclusion but Deadlock

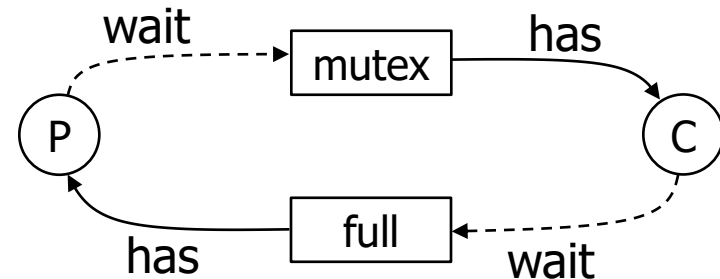
The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded with a lock.

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;
4
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&mutex); // line p0
9         sem_wait(&empty); // line p1
10        put(i);           // line p2
11        sem_post(&full);  // line p3
12        sem_post(&mutex); // line p4
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex); // line c0
20         sem_wait(&full);  // line c1
21         int tmp = get();  // line c2
22         sem_post(&empty); // line c3
23         sem_post(&mutex); // line c4
24         printf("%d\n", tmp);
25     }
26 }
```

```
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX);
31     sem_init(&full, 0, 0);
32     sem_init(&mutex, 0, 1); // mutex=1
33     // ...
34 }
```

Line	mutex	empty	full
32	1	1	0
C0	0	1	0
C1	0	1	-1 (C blocked)
P0	-1 (P blocked)	1	-1

each stuck waiting for each other



deadlock

A Working Solution – Fine-Grained Lock

To solve the deadlock problem, we simply must reduce the scope of the lock

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;
4
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); // line p1.5
10        put(i);           // line p2
11        sem_post(&mutex); // line p2.5
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); // line c1.5
21         int tmp = get(); // line c2
22         sem_post(&mutex); // line c2.5
23         sem_post(&empty); // line c3
24         printf("%d\n", tmp);
25     }
26 }
```

```
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX);
31     sem_init(&full, 0, 0);
32     sem_init(&mutex, 0, 1); // mutex=1
33     // ...
34 }
```

Reader-Writer Locks

```
1 typedef struct _rwlock_t {
2     sem_t lock; // binary semaphore
3     sem_t writelock; // RW lock
4     int readers; // count of readers
5 } rwlock_t;
6
7 void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 0, 1);
10    sem_init(&rw->writelock, 0, 1);
11}
12
```

```
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); //first reader
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); //last reader
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

Only one writer can enter CS.

The first reader acquires writelock. (open)

The last reader releases writelock. (close)

The other readers can read w/o writelock.

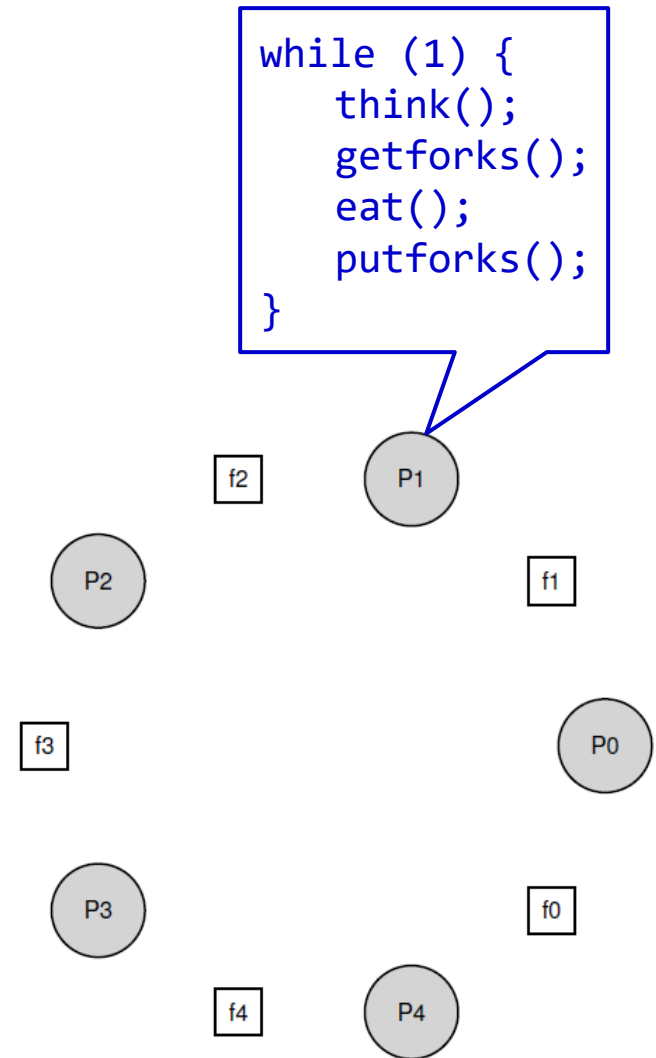
→ multiple readers can enter CS.

Any writer must wait until all readers are finished. → fairness problem: **readers can starve writers.**

→ Sol. prevent more readers from entering the lock once a writer is waiting.

Dining Philosopher's Problem

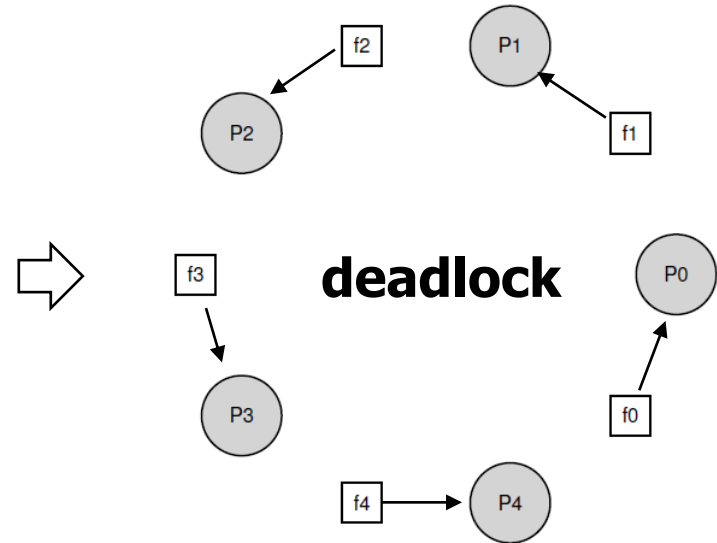
- Five “philosophers” sitting around a table.
- Between each pair of philosophers is a single fork (and thus, five total).
- The philosophers each have times where they think, and don’t need any forks, and times where they eat.
- In order to eat, a philosopher needs two forks, both the one on their left and the one on their right.
- The contention for these forks is a synchronization problem
- Requirements
 - No deadlock
 - No starvation



Dining Philosopher's Problem

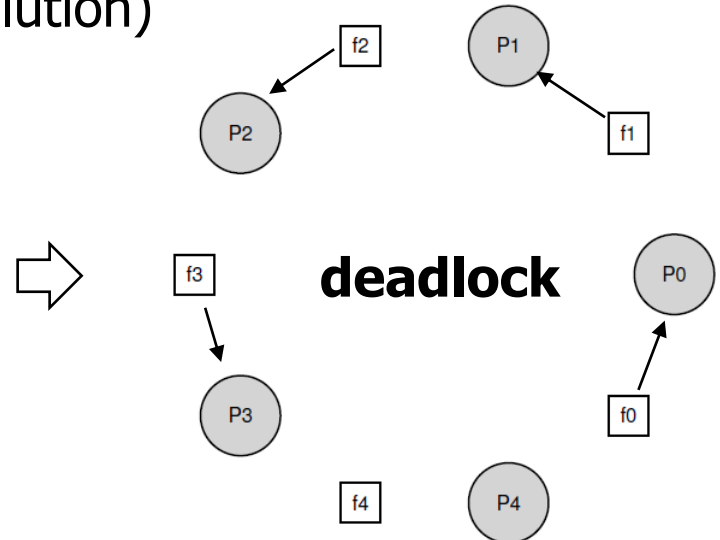
- **Broken Solution**

```
1 void getforks() {  
2     sem_wait(forks[left(p)]);  
3     sem_wait(forks[right(p)]);  
4 }  
5  
6 void putforks() {  
7     sem_post(forks[left(p)]);  
8     sem_post(forks[right(p)]);  
9 }
```



- **Breaking Dependency (Dijkstra's Solution)**

```
1 void getforks() {  
2     if (p == 4) {  
3         sem_wait(forks[right(p)]);  
4         sem_wait(forks[left(p)]);  
5     } else {  
6         sem_wait(forks[left(p)]);  
7         sem_wait(forks[right(p)]);  
8     }  
9 }
```



Dining Philosopher's Problem

- **Other Solutions**

- Allow at most 4 philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
- Use an asymmetric solution
 - an odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
 - Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

How To Implement Semaphores

Original Definition of Semaphore

```
int sem_wait(sem_t *s) { //atomic
    s->value--;
    if (s->value < 0) wait;
}

int sem_post(sem_t *s) { //atomic
    s->value++;
    if (s->value <= 0)
        wake one waiting thread
}
```

the value will never
be lower than zero

```
1 typedef struct __Zem_t {
2     int value;
3     pthread_cond_t cond;
4     pthread_mutex_t lock;
5 } Zem_t;
6
7 // only one thread can call this
8 void Zem_init(Zem_t *s, int value) {
9     s->value = value;
10    Cond_init(&s->cond);
11    Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```