# 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)

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

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

```c
sem_t s;

void *
child(void *arg) {
   printf("child\n");
   sem_post(&s); //child is done
   return NULL;
}

int
main(int argc, char *argv[]) {
   sem_init(&s, 0, X); //what should X be?
   printf("parent: begin\n");
   pthread_t c;
   Pthread_create(c, NULL, child, NULL);
   sem_wait(&s); //wait here for child
   printf("parent: end\n");
   return 0;
}
```

# 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

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

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

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

| 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 }
```

Works only when a producer and a consumer!

# Adding Mutual Exclusion but Deadlock

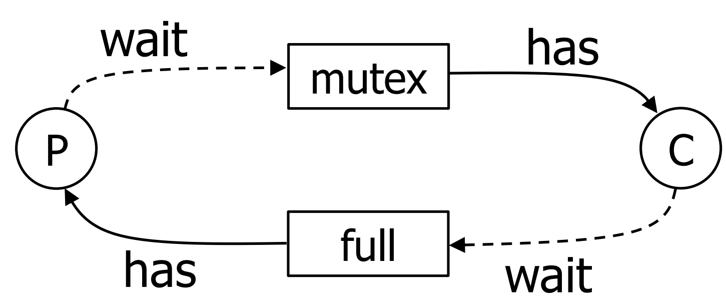
The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded with a lock.

```
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

```
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); // line p1.5
      put(i);            // line p2
      sem_post(&mutex); // line p2.5
      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); // line c1.5
      int tmp = get();   // line c2
      sem_post(&mutex); // line c2.5
      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);
  sem_init(&mutex, 0, 1); // mutex=1
  // ...
}
```

# Reader-Writer Locks

```
typedef struct _rwlock_t {
  sem_t lock; // binary semaphore
  sem_t writelock; // RW lock
  int readers; // count of readers
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

```

```
void rwlock_acquire_readlock(rwlock_t *rw) {
     sem_wait(&rw->lock);
     rw->readers++;
     if (rw->readers == 1)
         sem_wait(&rw->writelock);//first reader
     sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
     sem_wait(&rw->lock);
     rw->readers--;
     if (rw->readers == 0)
         sem_post(&rw->writelock); //last reader
     sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
     sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
     sem_post(&rw->writelock);
}
```

Only one writer can enter CS.

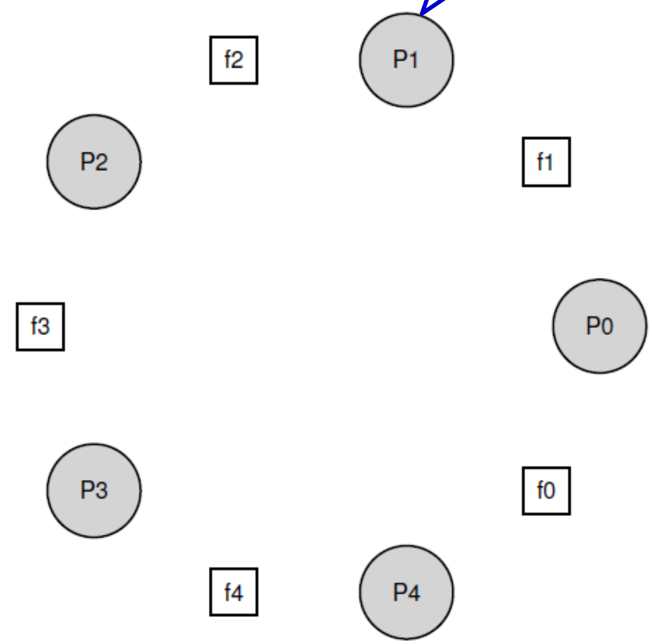
The first reader acquires writelock. (open)
The last reader acquires writelock. (close)
The other readers can read w/o writelock.
➔ multiple readers can enter CS.

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

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

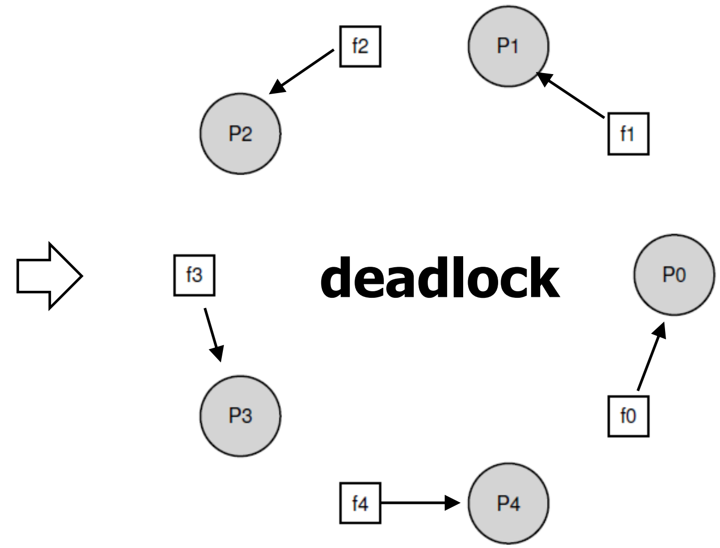


# Dining Philosopher's Problem

- **Broken Solution**

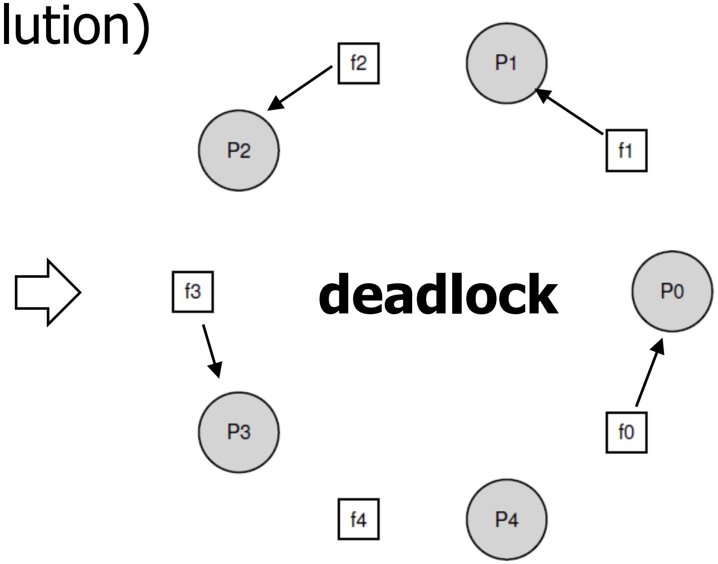
```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

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




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

```
void getforks() {
    if (p == 4) {
        sem_wait(forks[right(p)]);
        sem_wait(forks[left(p)]);
    } else {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
    }
}
```



# 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
}
```

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

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

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

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