

# Operating Systems

---

## Semaphores

# Semaphore: A definition

- An object **with an integer value**

- ◆ We can manipulate with two routines; `sem_wait()` and `sem_post()`.
- ◆ Initialization

```
1  #include <semaphore.h>
2  sem_t s;
3  sem_init(&s, 0, 1); // initialize s to the value 1
```

- Declare a semaphore `s` and initialize it to the value 1
- The second argument, 0, indicates that the semaphore is shared between *threads in the same process*.

# Semaphore: Interact with semaphore

## ▣ `sem_wait()`

```
1  int sem_wait(sem_t *s) {  
2      decrement the value of semaphore s by one  
3      wait if value of semaphore s is negative  
4  }
```

- ◆ If the value of the semaphore was *zero* or *higher* when called `sem_wait()`, **return right away**.
- ◆ When negative, it will cause the caller to suspend execution waiting for a subsequent post.
- ◆ When negative, the value of the semaphore is equal to the number of waiting threads.

# Semaphore: Interact with semaphore (Cont.)

## ▣ `sem_post()`

```
1  int sem_post(sem_t *s) {  
2      increment the value of semaphore s by one  
3      if there are one or more threads waiting, wake one  
4  }
```

- ◆ Simply **increments** the value of the semaphore.
- ◆ If there is a thread waiting to be woken, **wakes** one of them up.

# Binary Semaphores (Locks)

## □ What should **x** be?

- ◆ The initial value should be **1**.

```
1  sem_t m;  
2  sem_init(&m, 0, X); // initialize semaphore to X; what should X be?  
3  
4  sem_wait(&m);  
5  //critical section here  
6  sem_post(&m);
```

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

# Thread Trace: Two Threads Using A Semaphore

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	<i>Interrupt; Switch → T1</i>	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0) → sleep	sleeping
-1		Running	<i>Switch → T0</i>	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	<i>Interrupt; Switch → T1</i>	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

# The Producer/Consumer (Bounded-Buffer) Problem

## □ **Producer:** `put()` interface

- ◆ Wait for a buffer to become *empty* in order to put data into it.

## □ **Consumer:** `get()` interface

- ◆ Wait for a buffer to become *filled* before using it.

```
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 }
```

# The Producer/Consumer (Bounded-Buffer) Problem

```
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 ...
```

**First Attempt: Adding the Full and Empty Conditions**



# The Producer/Consumer (Bounded-Buffer) Problem

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

## First Attempt: Adding the Full and Empty Conditions (Cont.)

- ◆ Imagine that `MAX` is greater than 1 .
  - If there are multiple producers, **race condition** can happen at line *f1*.
  - It means that the old data there is overwritten.
- ◆ We've forgotten here is **mutual exclusion**.
  - The filling of a buffer and incrementing of the index into the buffer is a **critical section**.

# A Solution: Adding Mutual Exclusion

```
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 (new line)
10         put(i);                     // line p2
11         sem_post(&mutex);           // line p2.5 (new line)
12         sem_post(&full);            // line p3
13     }
14 }
15
```

**Adding Mutual Exclusion (Correctly)**

# A Working Solution

```
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 (new line)
21         int tmp = get();           // line c2
22         sem_post(&mutex);          // line c2.5 (new line)
23         sem_post(&empty);          // line c3
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with ...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33     // ...
34 }
35     ...
```

**Adding Mutual Exclusion (Correctly)**

# Reader-Writer Locks

- ▣ Imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
  - ◆ **insert:**
    - Change the state of the list
    - A traditional critical section makes sense.
  - ◆ **lookup:**
    - Simply *read* the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed **concurrently**.

**This special type of lock is known as a **reader-write lock**.**

# A Reader-Writer Locks

- ▣ Only a **single writer** can acquire the lock.
- ▣ Once a reader has acquired a **read lock**,
  - ◆ **More readers** will be allowed to acquire the read lock too.
  - ◆ A writer will have to wait until all readers are finished.

```
1  typedef struct _rwlock_t {
2      sem_t lock;           // binary semaphore (basic lock)
3      sem_t writelock;      // used to allow ONE writer or MANY readers
4      int readers;          // count of readers reading in critical section
5  } rwlock_t;
```

```
1. void rwlock_init(rwlock_t *rw) {
2.     rw->readers = 0;
3.     sem_init(&rw->lock, 0, 1);
4.     sem_init(&rw->writelock, 0, 1);
5. }
```

# A Reader-Writer Locks

```
1. void rwlock_acquire_readlock(rwlock_t *rw) {
2.     sem_wait(&rw->lock);
3.     rw->readers++;
4.     if (rw->readers == 1)
5.         sem_wait(&rw->writelock); // first reader acquires writelock
6.     sem_post(&rw->lock);
7. }
```

```
1. void rwlock_release_readlock(rwlock_t *rw) {
2.     sem_wait(&rw->lock);
3.     rw->readers--;
4.     if (rw->readers == 0)
5.         sem_post(&rw->writelock); // last reader releases writelock
6.     sem_post(&rw->lock);
7. }
```

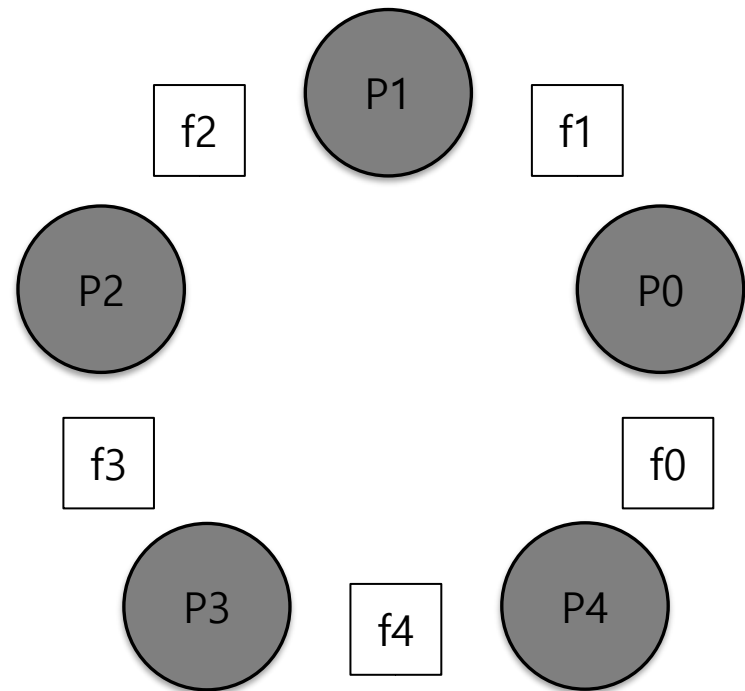
## A Reader-Writer Locks (Cont.)

```
1. void rwlock_acquire_writelock(rwlock_t *rw) {  
2.     sem_wait(&rw->writelock);  
3. }
```

```
1. void rwlock_release_writelock(rwlock_t *rw) {  
2.     sem_post(&rw->writelock);  
3. }
```

# The Dining Philosophers

- ▣ Assume there are five “**philosophers**” sitting around a table.
  - ◆ Between each pair of philosophers is a single fork (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.**





# The Dining Philosophers (Cont.)

## □ Key challenge

- ◆ There is **no deadlock**.
- ◆ **No** philosopher **starves** and never gets to eat.
- ◆ **Concurrency** is high.

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

**Basic loop of each philosopher**

- Philosopher  $p$  wishes to refer to the fork on their left  $\rightarrow$  call `left(p)`.
- Philosopher  $p$  wishes to refer to the fork on their right  $\rightarrow$  call `right(p)`.

# The Dining Philosophers (Cont.)

- We need some **semaphore**, one for each fork: `sem_t forks[5]`.

```
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  }
```

The `getforks()` and `putforks()` Routines (Broken Solution)

- ◆ **Deadlock** occur!
  - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right.
  - Each will be stuck *holding one fork* and waiting for another, *forever*.

# A Solution: Breaking The Dependency

- ▣ Change how forks are acquired.

- ◆ Let's assume that philosopher 4 acquire the forks in a *different order*.

```
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  }
```

- There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken.**