

ECE 344: Operating Systems
Lecture 11

Semaphores

1.0.0

Jon Eyolfson
October 3, 2022



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/)

Locks Ensure Mutual Exclusion

Only one thread at a time can be between the `lock` and `unlock` calls

It does not help you ensure ordering between threads

How would we ensure an ordering between two threads?

Problem: Make One Thread Always Print First

Thread 1 (print_first)
printf("This is first\n");

Thread 2 (print_second)
printf("I'm going second\n");

Try executing ./ordered-print and see what happens

Recall: printf is thread safe, which you may need to ensure

Try executing ./safe-print which (oddly) prints using multiple system calls
Practice: ensure ./safe-print behaves the same as ./ordered-print

Semaphores are Used for Signaling

Semaphores have a value that's shared between threads (optionally processes)

Think of value as an integer that is always ≥ 0

It has two fundamental operations wait and post

wait decrements the value atomically

post increments the value atomically

If wait will not return until the value is greater than 0

You can initially set value to whatever you want

That number of wait calls may occur without any post calls

Semaphore API is Similar to pthread Locks

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value)
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

int sem_post(sem_t *sem);
```

All functions return 0 on success

The pshared argument is a boolean, you can set it to 1 for IPC
For IPC the semaphore needs to be in shared memory

Problem: Make One Thread Always Print First

See `ordered-print.c` for the full code

Note: return statements are removed for space

```
void* print_first(void* arg) {  
    printf("This is first\n");  
}  
  
void* print_second(void* arg) {  
    printf("I'm going second\n");  
}  
  
int main(int argc, char *argv[]) {  
    /* Initialize, create, and join threads */  
}
```

This Always Executes print_first Then print_second

```
static sem_t sem;

void* print_first(void* arg) {
    printf("This is first\n");
    sem_post(&sem);
}

void* print_second(void* arg) {
    sem_wait(&sem);
    printf("I'm going second\n");
}

int main(int argc, char *argv[])
{
    sem_init(&sem, 0, 0);
    /* Initialize, create, and join threads */
}
```

No Matter Which Thread Executes First, We Get the Same Order

The value is initially 0

Assume `print_second` executes first

It executes `sem_wait`, which is 0, and doesn't continue

`print_first` doesn't have to wait, it prints first before it increments the value

`print_second` can then execute its print statement

What happens if we initialized the value to 1?

We Can Use a Semaphore as a Mutex

How?

Using a Semaphore as a Mutex, Note the value

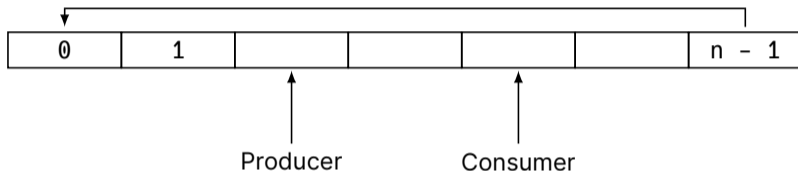
```
static sem_t sem;
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        sem_wait(&sem);
        ++counter;
        sem_post(&sem);
    }
}

int main(int argc, char *argv[]) {
    sem_init(&sem, 0, 1);
    /* Initialize, create, and join multiple threads */
    printf("counter = %i\n", counter);
}
```

Can We Come Up with a Solution for a Producer/Consumer Problem?

Assume you have a circular buffer (each slot is either empty or filled):



The producer should write to the buffer (if the buffer is not full)

The consumer should read from the buffer (if the buffer is not empty)

All consumers share an index and all producers share an index

In both cases the index is initially 0 and increases sequentially

Problem 1: Ensure Producers Never Overwrite Filled Slots

```
static uint32_t buffer_size;

void init_semaphores() {
    sem_init(&empty_slots, 0, /* ? */);
}

void producer() {
    while (/* ... */) {
        /* spend time producing data */
        fill_slot();
    }
}

void consumer() {
    while (/* ... */) {
        empty_slot();
        /* spend time consuming data */
    }
}
```

Use a Semaphore to Track the Number of Empty Slots

```
void init_semaphores() {
    sem_init(&empty_slots, 0, buffer_size);
}
void producer() {
    while (/* ... */) {
        /* spend time producing data */
        sem_wait(&empty_slots);
        fill_slot();
    }
}
void consumer() {
    while (/* ... */) {
        empty_slot();
        sem_post(&empty_slots);
        /* spend time consuming data */
    }
}
```

What is our next problem?

Problem 2: Ensure Consumers Never Consume Empty Slots

```
void init_semaphores() {
    sem_init(&empty_slots, 0, buffer_size);
    sem_init(&filled_slots, 0, /* ? */);
}
void producer() { while (/* ... */) {
    /* spend time producing data */
    sem_wait(&empty_slots);
    fill_slot();
} }
void consumer() { while (/* ... */) {
    empty_slot();
    sem_post(&empty_slots);
    /* spend time consuming data */
} }
```

Two Semaphores Ensure Proper Order for Producers and Consumers

```
void init_semaphores() {
    sem_init(&empty_slots, 0, buffer_size);
    sem_init(&filled_slots, 0, 0);
}

void producer() { while (/* ... */) {
    /* spend time producing data */
    sem_wait(&empty_slots);
    fill_slot();
    sem_post(&filled_slots);
} }

void consumer() { while (/* ... */) {
    sem_wait(&filled_slots);
    empty_slot();
    sem_post(&empty_slots);
    /* spend time consuming data */
} }
```

What Happens If We Initialize Both Semaphore Values to 0?

```
void init_semaphores() {
    sem_init(&empty_slots, 0, 0);
    sem_init(&filled_slots, 0, 0);
}

void producer() { while (/* ... */) {
    /* spend time producing data */
    sem_wait(&empty_slots);
    fill_slot();
    sem_post(&filled_slots);
} }

void consumer() { while (/* ... */) {
    sem_wait(&filled_slots);
    empty_slot();
    sem_post(&empty_slots);
    /* spend time consuming data */
} }
```


We Used Semaphores to Ensure Proper Order

Previously we ensured mutual exclusion, now we can ensure order

- Semaphores contain an initial value you choose
- You can increment the value using post
- You can decrement the value using wait (it blocks if the current value is 0)
- You still need to be prevent data races