Chapter 5c: Semaphores: Communicating Between Threads

Gene Cooperman

Contents

THIS IS A WORK IN PROGRESS.

1 Semaphores in Multithreaded Programming

1.1 The Three Multithreaded Primitives

You have already seen how threads communicate through **shared memory**, and how we protect shared data using **mutexes**. Let's step back and note that there are three commonly used synchronization primitives for multithreaded programming:

1. **Mutex** (mutual exclusion) – creates a critical section. *(Already covered.)*
2. **Semaphore** – the subject of this lecture.
3. **Condition Variable** – to be discussed in a later lecture.

1.2 Programmer's Model: Mutex vs. Semaphore

Recall that the "programmer's model" for a mutex is simply a location in shared memory holding a single bit (locked or unlocked):

```
+------------+
|   MUTEX    | <--- a single bit (0 or 1)
+------------+
```

The programmer's model for a **semaphore** is similarly a location in shared memory, but instead of a single bit, it holds an **integer**. That integer is called the **count**.

```
+------------+
| SEMAPHORE  | <--- a single int ("count")
+------------+
```

That's it. At the hardware level, a semaphore is just a memory location holding an integer. All of the interesting behavior comes from the API that manipulates this count.

1.3 The Semaphore API

There are four things you need to know:

```c
#include <semaphore.h>

sem_t mysem;                    // (i)   Declare the semaphore in global memory

sem_init(&mysem, 0, count);    // (ii)  Initialize: set the semaphore's count
                                //       (The second argument, 0, means
                                //        "shared between threads, not processes.")

sem_wait(&mysem);               // (iii) Potentially block (wait) on the semaphore

sem_post(&mysem);               // (iv)  Signal (post to) the semaphore
```

1.4 The Meaning of "count"

The integer count inside a semaphore has a dual interpretation:

- **If count is negative:** The absolute value |count| is the number of threads currently *blocked* (waiting) on this semaphore.
- **If count is positive:** The count is the number of *free passes* – that is, the number of times a thread can call `sem_wait()` and proceed immediately, without actually blocking.
- **If count is zero:** There are no free passes, and no threads are currently waiting. The next thread to call `sem_wait()` will block.

1.5 How Semaphores Differ from Mutexes

With mutexes, `pthread_mutex_lock()` and `pthread_mutex_unlock()` are always called **in pairs by the same thread**. A thread locks the mutex, does its work inside the critical section, and then unlocks it.

Semaphores are more general. In a typical application, one thread calls `sem_wait()` and may block, and then a **different** thread calls `sem_post()` to wake up the first thread. The "wait" and the "post" happen in different threads. This is the key conceptual difference from mutexes.

However, semaphores can also be used as a **generalization of mutexes**. In this case, the *same* thread calls `sem_wait()` followed by `sem_post()`, much like `pthread_mutex_lock()` and `pthread_mutex_unlock()`. We will see examples of both uses.

1.6 sem_wait() and sem_post() in Detail

If you understand the meaning of count, then the operations are intuitive:

- Calling `sem_wait()` means: "I may have to wait." So we **decrement** the count.
- Calling `sem_post()` means: "I'm signaling that one more thread can proceed." So we **increment** the count.

1.6.1 Pseudocode for `sem_wait(&mysem)`:

```
count--;
if (count < 0) {
    BLOCK;         // This thread goes to sleep.
} else {
    CONTINUE;      // This thread proceeds without waiting.
}
```

After decrementing, if count has gone negative, it means there are no free passes left, and this thread must block. If count is still non-negative, a free pass was available, and the thread continues.

1.6.2 Pseudocode for `sem_post(&mysem)`:

```
count++;
if (count <= 0) {
    WAKE UP one blocked thread;   // Some thread was waiting; wake it.
} else {
    CONTINUE;                     // No one was waiting; the free pass is stored.
}
```

After incrementing, if count is still zero or negative, that means there are threads blocked on this semaphore (recall that a negative count indicates waiting threads). So we wake one of them up. If count has become positive, no one was waiting, and the incremented count simply becomes a stored free pass for a future `sem_wait()`.

Implementation note: When a thread executes BLOCK, the operating system changes that thread's state from RUNNING to SLEEPING. The thread consumes no CPU while sleeping. When a corresponding `sem_post()` wakes it up, the OS moves it back to READY (and eventually RUNNING).

1.7 Three Applications of Semaphores (Use Cases)

In practice, semaphores are used in two main patterns.

1.7.1 APPLICATION #1: Ordering (Event Signaling)

Sometimes one thread must wait until another thread has completed a particular task. A semaphore initialized to **0** achieves this.

Example: A parent thread creates a child thread. The child must initialize a shared data structure before the parent can use it.

```
sem_t mysem;

// ---- Parent Thread (Thread A) ----
sem_init(&mysem, 0, 0);              // count = 0: no free passes

pthread_create(&tid, NULL, child_fn, NULL);

sem_wait(&mysem);                    // Block here until the child signals.
// At this point, the shared data structure is ready.
// The parent can now share work with the child.

// ---- Child Thread (Thread B) ----
void *child_fn(void *arg) {
    initialize_shared_data_structure();

    sem_post(&mysem);                // Signal the parent: "Data is ready."

    // Continue doing other work ...
}
```

Why this works: The semaphore starts at count = 0. When the parent calls `sem_wait()`, count becomes -1 (negative), so the parent blocks. Later, when the child calls `sem_post()`, count goes from -1 back to 0, and the blocked parent is woken up.

Even if the child runs first and calls `sem_post()` before the parent calls `sem_wait()`, it still works correctly: count goes from 0 to 1 (a stored free pass), so when the parent eventually calls `sem_wait()`, count goes from 1 to 0, and the parent continues without blocking.

1.7.2 APPLICATION #2: Shared Resources (Counting)

A semaphore can track a finite number of identical resources. The count is initialized to the **number of available resources**. Each `sem_wait()` claims one resource; each `sem_post()` releases one.

Example: Array entry pool. Suppose you have an array of 5 available slots:

```c
#define NUM_SLOTS 5
sem_t slot_sem;
sem_init(&slot_sem, 0, NUM_SLOTS);   // count = 5: five resources available

// Any thread that needs a slot:
sem_wait(&slot_sem);      // Claim a slot (count decremented).
// ... use the slot ...
sem_post(&slot_sem);      // Release the slot (count incremented).
```

If all 5 slots are in use (count = 0) and a sixth thread calls `sem_wait()`, it will block until some other thread releases a slot via `sem_post()`.

Example: E-commerce server sockets. A web server has a fixed number of CPU cores, and each core can handle one customer transaction at a time. The semaphore count is initialized to the number of cores. When a new customer connection arrives, the handler calls `sem_wait()` to claim a core. When the transaction is complete, it calls `sem_post()` to release the core. If all cores are busy, new customers wait automatically until a core is freed.

1.7.3 The Producer-Consumer Problem

Applications #1 and #2 come together in one of the most important patterns in concurrent programming: **producer-consumer**.

Motivation: Imagine a pipeline where one or more *producer* threads generate data items (e.g., incoming network requests, file chunks, tasks to process), and one or more *consumer* threads process those items. The producers and consumers share a **bounded buffer** – a fixed-size array of slots.

Extended example – an e-commerce server: Consider a server with multiple CPU cores. The server maintains a bounded buffer whose size equals the number of CPU cores. A *producer* is a thread handling an incoming client connection – when a customer submits an e-commerce transaction (e.g., a purchase), the producer thread deposits that transaction into the buffer. A *consumer* is a thread bound to one of the server's CPU cores – it pulls a transaction from the buffer and processes it against a back-end database (updating inventory, charging the customer's account, etc.). If customers are submitting transactions faster than the cores can process them, producers must wait until a core finishes and frees a buffer slot. If the cores are idle and the buffer is empty, consumers must wait until a new transaction arrives.

Two problems must be solved simultaneously:

1. **A consumer must wait if the buffer is empty** – there is nothing to consume. *(This is an ordering problem – Application #1.)*
2. **A producer must wait if the buffer is full** – there is no room to produce. *(This is a shared-resource problem – Application #2.)*

We solve this with **two semaphores**:

```c
#define BUFFER_SIZE 10

sem_t prod_empty_slots;  // Shared resource for producers: empty slots.
sem_t cons_full_slots;   // Shared resource for consumers: full slots.
pthread_mutex_t mutex; // Protects the shared buffer itself.

int buffer[BUFFER_SIZE];
int in = 0, out = 0;
```

```c
void init() {
    sem_init(&prod_empty_slots, 0, BUFFER_SIZE);  // Producers see BUFFER_SIZE empty slots.
    sem_init(&cons_full_slots,  0, 0);             // Consumers see 0 full slots.
    pthread_mutex_init(&mutex, NULL);
}
```

1.7.3.1 Producer:

```c
void *producer(void *arg) {
    while (1) {
        int item = produce_item();

        sem_wait(&prod_empty_slots);      // Claim one empty slot resource.
        pthread_mutex_lock(&mutex);        // Enter critical section.

        buffer[in] = item;                 // Place item in buffer.
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);      // Leave critical section.
        sem_post(&cons_full_slots);        // Create one full-slot resource for consumers.
    }
}
```

1.7.3.2 Consumer:

```c
void *consumer(void *arg) {
    while (1) {
        sem_wait(&cons_full_slots);        // Claim one full-slot resource.
        pthread_mutex_lock(&mutex);        // Enter critical section.

        int item = buffer[out];            // Remove item from buffer.
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);      // Leave critical section.
        sem_post(&prod_empty_slots);       // Create one empty-slot resource for producers.

        consume_item(item);
    }
}
```

1.7.3.3 Why this works – thinking in terms of shared resources:

Both semaphores follow **Application #2 (Shared Resources)**. The `sem_init` calls establish the initial resource counts that each type of thread sees.

`prod_empty_slots` is initialized to `BUFFER_SIZE`: From a producer's perspective, the shared resources are *empty slots* in the buffer. Initially, all `BUFFER_SIZE` slots are empty, so a producer has `BUFFER_SIZE` free passes. A producer can call `sem_wait(&prod_empty_slots)` up to `BUFFER_SIZE` times – each time claiming one empty slot and filling it – before the count reaches 0 and the next `sem_wait` forces it to block. At that point, the buffer is full, and the producer must wait until a consumer frees up an empty slot.

`cons_full_slots` is initialized to 0: From a consumer's perspective, the shared resources are *full slots* – slots containing items ready to be consumed. Initially, no items have been produced yet, so there are zero full-slot resources available. This means that any consumer calling `sem_wait(&cons_full_slots)` will immediately block, because the count goes from 0 to -1. The consumer has no choice but to wait until a producer makes a full slot available.

The flow of resources between producers and consumers: When a producer finishes placing an item in the buffer, it calls `sem_post(&cons_full_slots)`. This *creates a new full-slot resource* for the consumers. The count of `cons_full_slots` is incremented, and if any consumer was blocked, it is woken up to claim that resource. Symmetrically, when a consumer finishes removing an item from the buffer, it calls `sem_post(&prod_empty_slots)`. This *creates a new empty-slot resource* for the producers. The count of `prod_empty_slots` is incremented, and if any producer was blocked waiting for space, it is woken up.

In other words, the total number of resources is conserved: every empty slot that a producer consumes (via `sem_wait`) is transformed into a full slot that it donates to consumers (via `sem_post`), and vice versa. The two semaphores track the two pools of shared resources, and the `sem_wait`/`sem_post` calls transfer resources back and forth between them.

The **mutex** plays a separate role. It protects the buffer array and the index variables (`in`, `out`) from race conditions when multiple producers or consumers access the buffer concurrently. The mutex provides a critical section, just as you learned before.

1.7.4 APPLICATION #3: Semaphores as a Generalization of Mutexes: Thread Pools

In all of the previous examples, one thread called `sem_wait()` and a *different* thread called `sem_post()`. Now we look at a pattern where the **same thread** calls both `sem_wait()` and later `sem_post()`. This makes the semaphore behave like a generalization of `pthread_mutex_lock()`/`pthread_mutex_unlock()`.

1.7.4.1 Motivation: Preventing Thrashing

Suppose a server has `NUM_CORES` CPU cores and needs to process many tasks. We could create a thread for every task – but if we create far more threads than cores, the operating system must constantly switch between them. This **thrashing** (excessive context switching) wastes CPU time and degrades performance. Instead, we want to ensure that at most `NUM_CORES` threads are actively doing work at any given time, even if more threads have been created. This is the idea behind a **thread pool**.

1.7.4.2 The Code

```
#include <pthread.h>
#include <semaphore.h>
#define NUM_THREADS 5
#define NUM_CORES 3
int task_num = 0;
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
sem_t mysem;

// The "start function" for a new thread is do_task.
void *do_task(void *arg) {
    pthread_mutex_lock(&mymutex);
    int mytask = task_num++;
```

```
    pthread_mutex_unlock(&mymutex);

    sem_wait(&mysem); // Decrement count; wait if count went negative.
    do_work(mytask);
    sem_post(&mysem); // Increment count; wake a waiting thread if any.
    return NULL;
}

int main() {
    pthread_t thread_ids[NUM_THREADS];
    // Initialize count to NUM_CORES: at most NUM_CORES threads may
    // call do_work concurrently.
    sem_init(&mysem, 0, NUM_CORES);
    for (int i = 0; i < NUM_THREADS; i++) {
        // Create a new thread with the start function, do_task.
        pthread_create(&thread_ids[i], NULL, do_task, NULL);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        // Let the thread exit when the start function finishes.
        pthread_join(thread_ids[i], NULL);
    }
    return 0;
}
```

1.7.4.3 How it works

The semaphore `mysem` is initialized to `NUM_CORES` (in this example, 3). Think of the count as the number of available cores – i.e., the number of threads that are allowed to be inside `do_work()` at the same time.

Each thread calls `sem_wait(&mysem)` before entering `do_work()`. The first `NUM_CORES` threads each decrement the count (from 3 to 2, then to 1, then to 0) and proceed without blocking. When the next thread calls `sem_wait()`, the count goes to -1, and that thread blocks. It must wait until one of the active threads finishes `do_work()` and calls `sem_post(&mysem)`, which increments the count and wakes up the blocked thread.

Notice the pattern: the **same thread** calls `sem_wait()` before the work and `sem_post()` after the work. This is exactly how `pthread_mutex_lock()` and `pthread_mutex_unlock()` are used – in pairs, by the same thread. The difference is that a mutex allows only **one** thread into the critical section, while this semaphore allows up to `NUM_CORES` threads.

1.7.4.4 Connection to mutexes

If we changed the initialization to `sem_init(&mysem, 0, 1)`, then only one thread at a time could be inside `do_work()`. This would behave identically to wrapping `do_work()` in `pthread_mutex_lock()`/`pthread_mutex_unlock()`. In other words, a mutex is a semaphore with an initial count of 1. The semaphore generalizes the mutex by allowing the count to be any positive integer, letting up to N threads proceed concurrently instead of just one.

1.8 FUTURE NOTES (Semaphores and Resources)

Explain that a resource can be:

1. *single memory address (count) varying between 0 and -1*
2. *each array element*
3. *each CPU core (explain thrashing creates context switches, and we assume we just want to execute FIFO, without each new thread interrupting an old thread in a context switch)*

1.9 Summary

Concept	Key Idea
Semaphore	A shared integer ("count") with atomic wait/post operations.
`sem_wait()`	Decrement count. If count < 0, block.
`sem_post()`	Increment count. If count $<= 0$, wake a blocked thread.
Application #1	**Ordering:** Initialize count to 0. One thread waits; another posts.
Application #2	**Shared resources:** Initialize count to N (number of resources).
Producer-Consumer	Combines both applications with two semaphores and a mutex.
Thread Pool	Same thread calls `sem_wait`/`sem_post`; generalizes a mutex to allow N concurrent threads.