Introducing Semaphores

A **semaphore**, like a mutex, is a **synchronization primitive** used to control access to a common resource by multiple processes or threads. However, a semaphore offers a more generalized way to manage access than a simple mutex.

---

1. Semaphores: Communicating Between Threads

Semaphores are a fundamental **synchronization primitive** in operating systems and concurrent programming. They provide a mechanism for two or more independent threads or processes to **communicate** with each other. This communication serves two main purposes: **mutual exclusion** (protecting a critical section of code) and **synchronization** (enforcing a specific order of execution). Unlike simple flags or shared variables, a semaphore's operations are guaranteed to be **atomic** (uninterruptible), making them safe to use in multi-threaded environments.

1.1. The Core Concept: The Count

The fundamental difference between a semaphore and a mutex lies in its **state**.

- A **mutex** is like a binary lock: it's either **locked** or **unlocked**. It protects *one* critical section.
- A **semaphore** is conceptually represented by an integer value, which we call the **count**.

This **count** determines how many threads can access a resource *simultaneously*.

- When the count is **positive** (> 0), it represents the **number of threads** that can still acquire the semaphore without blocking.
- When the count is **zero** ($= 0$), no more threads can acquire the semaphore; the resource is **fully in use**.
- When the count is **negative** (< 0), its absolute value ($|\text{count}|$) represents the **number of threads currently blocked** (sleeping) on this semaphore.

2. The `sem_wait()` Operation (The "Acquire" or "Down" Operation)

When a thread wants to use the protected resource, it calls `sem_wait()`.

1. The `sem_wait()` function **decrements the count** of the semaphore.
2. Crucially, this is done **atomically** (as a single, uninterruptible operation).
3. The thread then checks the new value of the count:
    - If the new count ≥ 0, the thread **continues executing** (it successfully acquired a "permit").

- If the new count < 0, the thread **atomically transitions to the SLEEPING state** (it has to wait because all available permits are taken).

3. The `sem_post()` Operation (The "Release" or "Up" Operation)

When a thread is finished using the protected resource (or signaling an event), it calls `sem_post()`.

1. The `sem_post()` function **increments the count** of the semaphore (also **atomically**).
2. The thread then checks the new value of the count:
   - If the new count > 0, the thread simply exits the function.
   - If the new count ≤ 0, it means there are threads waiting (sleeping). The system will then **wake up (unblock) one** of the threads sleeping on this semaphore, which can now proceed.

Semaphore vs. Mutex: The Key Difference

Feature	Mutex (Mutual Exclusion)	Semaphore (Counting/General)
Concept	A **binary lock** (0 or 1).	An **integer counter** (≥ 0).
Purpose	Ensures that **only one** thread can be in the critical section at a time.	Controls access to a resource that has a **limited number** of identical units available, or signals events.
Initial Value	Unlocked (**0**)	Any non-negative integer N (the max number of users).
Use Case	Protecting shared variables or a critical code segment.	Limiting concurrent threads, managing a fixed-size buffer (like a producer-consumer problem), or simply as a mutex (if initialized to 1).

Takeaway: A **mutex is a special case of a semaphore** where the initial count is set to **1** (a **binary semaphore**). The power of a general semaphore

is that it allows you to control **multiple permits** for a resource and to **signal** events between unrelated threads.

---

2. The Two Usage Scenarios: Ownership

Semaphores are incredibly versatile because they **do not require the thread that calls `sem_wait()` to be the same thread that calls `sem_post()`**. This flexibility leads to two primary scenarios for semaphore usage:

Scenario A: Self-Releasing (Resource Limiting or Mutex)

In this scenario, a single thread executes both operations:

- **Thread 1** calls → **`sem_wait()`** (Acquires a permit/lock)
- **Thread 1** calls → **`sem_post()`** (Releases the permit/lock)

This is the standard pattern when using a semaphore for:

1. **Mutual Exclusion:** When the initial count is 1 (a binary semaphore), it acts exactly like a mutex.
2. **Resource Limiting (Thread Pool):** The semaphore's count limits the number of threads that are **active** or **working** simultaneously.

   Example (Dynamic Thread Limiting): A system might use a semaphore initialized to 10 to limit a thread pool to 10 active threads. If the computer is running slowly, a supervisor thread may deliberately call **`sem_wait()`** an **extra time** outside of the worker threads' loop, permanently decrementing the count to 9. This **allows fewer threads** to be active. Conversely, if the computer has more idle capacity, the supervisor thread can call **`sem_post()`** an **extra time** to increment the count back to 10, **allowing more threads** to become active.

Scenario B: Cross-Thread Signaling (Synchronization)

In this scenario, one thread waits for an event, and a second, unrelated thread signals that the event has occurred:

- **Thread 1** calls → **`sem_wait()`** (Sleeps, waiting for an event)
- **Thread 2** calls → **`sem_post()`** (Signals the event and wakes Thread 1)

This pattern is used exclusively for **synchronization**—coordinating the execution order between different threads.

---

3. Scenario A Example: Thread Pool

The following C code (`dosem.c`) initializes a thread pool and uses a semaphore to explicitly limit the number of threads that can execute the `do_work()` function concurrently.

```c
#include <pthread.h>
#include <sem.h>
#define NUM_THREADS 5
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 negative
    do_work(mytask);
    sem_post(&mysem); // Increment count
    return NULL;
}
int main() {
    pthread_t thread_ids[NUM_THREADS];
    // Initialize count to 3; At most 3 threads may call do_work.
    sem_init(&mysem, 0, 3);
    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;
}
```

Explanation

This code is an example of **Scenario A (Self-Releasing)** because **each thread** that calls **`sem_wait()`** on `mysem` **is responsible for calling `sem_post()`** on `mysem` later.

1. **Initialization:** The semaphore `mysem` is initialized with a count of **3**

in `main()`: `sem_init(&mysem, 0, 3);`. This sets the limit: **at most three threads** can pass the `sem_wait()` call and enter the critical section (`do_work()`) simultaneously.
2. **Acquisition:** Five threads are created (`NUM_THREADS = 5`). The first three threads will call `sem_wait(&mysem)`, decrement the count to 2, 1, and 0 respectively, and proceed to `do_work()`.
3. **Blocking:** The fourth and fifth threads will call `sem_wait(&mysem)`, decrement the count to -1 and -2, and then **block (sleep)**, waiting for a permit.
4. **Release:** When one of the three active threads finishes `do_work()`, it calls `sem_post(&mysem)`, incrementing the count. If the count becomes ≤ 0 (e.g., from 0 to 1, or from -1 to 0), a waiting thread (the 4th or 5th) is woken up and granted the permit.

The semaphore thus acts as a **gatekeeper** to limit the resource consumption of the pool of threads.

---

4. Scenario B Example: Cross-Thread Signaling (Synchronization)

This scenario, where one thread waits on a semaphore and a *different* thread posts to it, is the true power of general semaphores. It is used exclusively to **signal events** and **enforce execution order** between threads. We will examine several classic synchronization problems that utilize this scenario.

4.1. Example: Enforcing Order Execution

This is the simplest application of Scenario B, often used to guarantee that a parent thread does not proceed until a setup task initiated in a child thread is complete.

- **Goal:** The parent thread must wait for the child thread to finish initializing a shared resource (a table) before the parent can access it.
- **Semaphore:** `init_done_sem` (initialized to **0**). The parent must block immediately, so the initial count must be less than 1.

Parent Thread (Main Execution)	Child Thread (Initialization Task)
`sem_init(&init_done_sem, 0, 0);`	`ChildStartFnc(...) {`
`pthread_create(..., childStartFnc, ...);`	`initialize_child();`
`// Parent blocks until child posts`	`add_child_entry_in_shared_table();`
`sem_wait(&init_done_sem);`	`// Post that child added to table`
`// The table is now safe to use`	`sem_post(&init_done_sem);`
`access_shared_table();`	`return;`
	`}`

Why `init_done_sem` is needed:

The `init_done_sem` semaphore is critical here because it enforces **ordering** between the parent and child threads. Since the parent thread initializes the semaphore's count to **0** and immediately calls `sem_wait(&init_done_sem);`, it is guaranteed to **block (sleep)**. It will remain blocked until the child thread has completed its necessary setup steps (`initialize_child()` and `add_child_entry_in_shared_table()`) and then calls `sem_post(&init_done_sem);`. This ensures that the parent thread's `access_shared_table()` call only executes *after* the child has successfully finished the shared resource preparation, preventing a **race condition** where the parent might access incomplete or uninitialized data.

4.2. Simplified Producer-Consumer: One-to-One Signaling

The **Producer-Consumer Problem** is a classic example in which there are two types of threads: **producer** and **consumer** threads. The producer generates data, and the consumer processes it. To start simply, we assume just **one producer** and **one consumer**, and they communicate via a single global integer variable, `task`.

We need two semaphores to coordinate their execution:

- `producer_sem`: Initialized to **1** (The producer can run first to create the initial task).
- `consumer_sem`: Initialized to **0** (The consumer must wait until a task is ready).

Consumer Thread (Processes data)	Producer Thread (Produces data)

while (true) {
`sem_wait(consumer_sem);`
`do_work(task);`
`sem_post(producer_sem);`
}
	while (true) {
	`sem_wait(producer_sem);`
	`task = produce_new_data();`
	`sem_post(consumer_sem);`
	}

Why Two Semaphores? Ideally, we might think that we only need `consumer_sem`, and the producer could simply signal to the consumer after adding a new task:

Producer (Simple Attempt)	Consumer

while (true) {
`task = produce_new_data();`
`sem_post(consumer_sem);`
	while (true) {
	`sem_wait(consumer_sem);`
	`do_work(task);`

Producer (Simple Attempt)	Consumer
}	}

The problem with this simpler idea is that the Producer might quickly produce a second task and **overwrite** the global `task` variable **before** the Consumer thread has even begun to execute the first instance of `do_work(task)`. The `producer_sem` semaphore is necessary to force the Producer to wait until the Consumer signals that the previous task has been processed.

4.3. Semaphore Example: The General Producer-Consumer Problem

The full Producer-Consumer Problem involves multiple threads and a fixed-size **buffer**. This problem expands on the simple example above by using a third semaphore (a mutex) and coordinating access to a finite resource.

- A **Producer** thread generates data and puts it into the buffer.
- A **Consumer** thread takes data out of the buffer and consumes it.
- The **Buffer** has a fixed, finite size (e.g., N slots).

Semaphores are used here for two distinct purposes: **mutual exclusion** and **synchronization**.

The Three Semaphores

We need three semaphores to correctly manage this problem:

Semaphore	Initial Value	Purpose	Scenario
`mutex`	**1**	**Mutual Exclusion:** Ensures only one thread accesses the buffer at a time.	A (Self-Releasing)
`producer_sem`	N (Buffer Size)	**Synchronization:** Counts the number of **empty** slots in the buffer.	B (Cross-Thread Signaling)

Semaphore	Initial Value	Purpose	Scenario
`consumer_sem`	**0**	**Synchronization:** Counts the number of **full** slots (items) in the buffer.	B (Cross-Thread Signaling)

A Buffer, Protected by the Mutex

The pseudo-code below now requires a mutex because a shared buffer data structure (the 'task' variable in the previous subsection (Section 4.2) is passed between producers and consumers. We must not allow two producers or two consumer to update the shared buffer simultaneously. This is true reglardless of whether the shared buffer is a simple int (the 'task of Section 4.2) or a circular buffer with'N' slots.

1. The Producer's Logic

The Producer's goal is to insert an item into the buffer. It must wait until there is an **empty** slot.

```
Producer() {
  while (true) {
    // 1. Produce an item
    // **Wait for an empty slot**
    sem_wait(producer_sem);
    // **Acquire exclusive access to the buffer**
    sem_wait(mutex);
    // 2. Insert item into buffer
    // **Release exclusive access**
    sem_post(mutex);
    // **Signal that a slot is now full**
    sem_post(consumer_sem);
  }
}
```

2. The Consumer's Logic

The Consumer's goal is to remove an item from the buffer. It must wait until there is a **full** slot.

```
Consumer() {
  while (true) {
```

```
    // **Wait for a full slot (an item to consume)**
    sem_wait(consumer_sem);
    // **Acquire exclusive access to the buffer**
    sem_wait(mutex);
    // 1. Remove item from buffer
    // **Release exclusive access**
    sem_post(mutex);
    // **Signal that a slot is now empty**
    sem_post(producer_sem);
    // 2. Consume the item
  }
}
```

Why This Works

The `consumer_sem` and `producer_sem` semaphores manage **synchronization** (making sure they wait for each other), while the `mutex` semaphore manages **mutual exclusion** (making sure they don't corrupt the data structure).

It is important to note that the above **Producer's Logic** and **Consumer's Logic** provide only the *synchronization blueprint* using semaphores, and **do not include the actual implementation of the buffer data structure**. The buffer, denoted abstractly as N slots, must be implemented separately. If the buffer were merely a single integer (like the `task` in the one-to-one signaling example in Section 4.2), the synchronization structure would implicitly prevent a second producer from generating input and returning to production until a consumer had processed the first task. For the *general* problem with $N > 1$ (multiple slots), the buffer is typically implemented as a **circular buffer** or **ring buffer** with N slots, which requires pointers or indices to track where the next item should be inserted and removed.