**Semaphores**

**31.1 Semaphores: A Definition**

A **semaphore** is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are sem\_wait() and sem\_post(). Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value:

Text

Description automatically generated with low confidence

we declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument indicates that the semaphore is shared between threads in the same process.

After a semaphore is initialized, we can call one of two functions to interact with it, sem\_wait() or sem\_post().

Text

Description automatically generated

First, we can see that sem\_wait() will either return right away because the value of the semaphore was one or higher when we called sem\_wait().

Second, we can see that sem\_post() does not wait for some particular condition to hold like sem\_wait() does.

Third, the value of the semaphore, when negative, is equal to the number of waiting threads.

**31.2 Binary Semaphores (Locks)**

We will implement a semaphore as a lock

Graphical user interface, text, application, email

Description automatically generated

The value of X should be 1.

To make this clear, let’s imagine a scenario with two threads. The first thread (Thread 0) calls sem\_wait(); it will first decrement the value of the semaphore, changing it to 0. Then, it will wait only if the value is not greater than or equal to 0. Because the value is 0, sem\_wait() will simply return and the calling thread will continue; Thread 0 is now free to enter the critical section. If no other thread tries to acquire the lock while Thread 0 is inside the critical section, when it calls sem\_post(), it will simply restore the value of the semaphore to 1.

Text

Description automatically generated

However, what if thread 1 call sem\_wait() when thread 0 has not called sem\_post()? The behavior is shown as follows:

Table

Description automatically generated

Since the locks only have two states, we sometimes call it **binary semaphore**.

**31.3 Semaphores For Ordering**

We often find one thread waiting for something to happen, and another thread making that something happen and then signaling that it has happened, thus waking the waiting thread. We are thus using the semaphore as an **ordering** primitive. A classic example would be a parent waits for a child to finish. The parent simply calls sem\_wait() and the child sem\_post() to wait for the condition of the child finishing its execution to become true.

In this case, the initial value would be 0 because when the parent call wait, it will decrement the value to -1 and go to sleep. Then, as the child finishes, it increment to 0 and wake up the parent.

If the child runs to completion before the parent gets a chance to wait, then the parent simply wakes up because the value is 1 (the child has already increased it) and then decrement to 0.

**31.4 The Producer/Consumer (Bounded Buffer) Problem**

**First Attempt**

Our first attempt uses two semaphores, **empty** and **full**, which the threads will use to indicate when a buffer entry has been emptied or filled.

Text, letter

Description automatically generated

Text, letter

Description automatically generated

The code should work when MAX=1. However, when MAX is greater than 1, and assuming that we have multiple processors and consumers. Race condition will occur here as two producers might try to fill and the value is overwritten.

**A Solution: Adding Mutual Exclusion**

Table

Description automatically generated

This is a wrong way of doing this because deadlock might occur because the critical section, we only care to fix is put and get. If the critical section includes locking the empty and full semaphore, then if the consumer runs first when there are no resources, then it goes to sleep on waiting for full. The producer, however, cannot do anything because the mutex lock is still held, so everyone stuck.

**A working solution**

Table

Description automatically generated

We simply swap the two locks.

**31.5 Reader-Writer Locks**

Imagine a number of concurrent list operations, including inserts and simple lookups. While inserts change the state of the list, lookups 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 is a case where we use reader-writer lock.

Text

Description automatically generated

If some thread wants to update the data structure in question, it should call the new pair of synchronization operations: rwlock\_acquire\_writelock(), to acquire a write lock, and rwlock\_release\_writelock(), to release it.

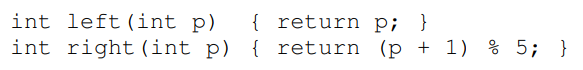
When acquiring a read lock, the reader first acquires lock and then increments the readers variable to track how many readers are currently inside the data structure. The important step then taken within rwlock\_acquire\_readlock() occurs when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling sem\_wait() on the writelock semaphore, and then releasing the lock by calling sem\_post(). Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock.

This approach has problem with fairness as it is easy for readers to starve.

**31.6 The Dining Philosophers**

The problem is that the philosophers sit in circle in a table and between each of them has forks. Each philosopher has to have two fork in order to eat.

The key challenge, then, is to write the routines get\_forks() and put\_forks() such that there is no deadlock, no philosopher starves and never gets to eat, and concurrency is high. The first solution is to have a left and right function.



**Broken Solution**

Text, letter

Description automatically generated

To acquire the forks, we simply grab a “lock” on each one: first the one on the left, and then the one on the right. When we are done eating, we release them.

However, this causes a problem as if every philosopher grabs their left fork, so they will wait for their right fork forever.

**A Solution: Breaking The Dependency**

The idea of this solution is that a philosopher will grab fork in different order (right then left).

**31.7 Thread Throttling**

How can a programmer prevent too many threads from doing something at once and bogging the system down?

Depends on how many threads then we use a semaphore to limit the number of threads concurrently executing the piece of code. We call this approach **throttling** and consider it a form of **admission control**.

Imagine that you create hundreds of threads to work on some problem in parallel. However, in a certain part of the code, each thread acquires a large amount of memory to perform part of the computation. If every thread requires a lot of memory, then we will run out of memory. The machine will start thrashing and the entire computation will be slow.

By initializing the value of the semaphore to the maximum number of threads you wish to enter the memory-intensive region at once, and then putting a sem\_wait() and sem\_post() around the region, a semaphore can naturally throttle the number of threads that are ever concurrently in the dangerous region of the code.

**31.8 How To Implement Semaphores**

One subtle difference between our Zemaphore and pure semaphores as defined by Dijkstra is that we don’t maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads

Text

Description automatically generated