Introduction

To handle a wide range of important and interesting concurrency problems, we now need both locks and condition variables. The semaphore was created as a single primitive for all things synchronization-related; as you'll see, semaphores can be used as both locks and condition variables.

How can we replace locks and condition variables with semaphores? What is a semaphore's definition? What is a binary semaphore, and how does it work? Is it simple to construct a semaphore using locks and condition variables? To use semaphores to create locks and condition variables?

What is a Semaphore?

A semaphore is an object with an integer value that we can handle with two routines: sem wait() and sem post()1 in the POSIX standard. Because its initial value determines the semaphore's behavior, we must first initialize it to a value, as shown in Figure 31.1, before running any other method to interact with it.

We declare a semaphore s in the diagram and set its value to 1 by supplying 1 as the third input. In all of the examples we'll see, the second argument to sem init() will be set to 0; this indicates that the semaphore is shared amongst threads in the same process. However, other uses of semaphores (such as how they might be used to synchronize access across separate processes) necessitate a different value for the second parameter; see the man page for further information.

After a semaphore has been initialized, we can interact with it using one of two functions: sem wait() or sem post() (). Figure 31.2 depicts the behavior of these two functions.

For the time being, we're not concerned with implementing these routines, which necessitates some attention; with many threads executing sem wait() and sem post(), it's clear that these critical parts must be managed. So we'll concentrate on using these primitives for now, and we'll talk about how they're made later.

We should talk about a few critical aspects of the interfaces now. First, we can see that sem wait() will either return immediately (since the semaphore value was one or greater when we used sem wait()), or it will cause the caller's execution to be suspended while waiting for a later post. Second, of course, many calling threads may call sem wait() simultaneously, putting them all in a queue to be woken up.

Second, unlike sem wait(), we can observe that sem post() does not wait for a specific condition to hold. Instead, it simply increases the semaphore value and then wakes up one of the threads waiting to be woken up. Third, the semaphore value is equal to the number of waiting threads when it is negative. Even though users of semaphores aren't aware of the value, this invariant is worth understanding and may help you recall how a semaphore works.

Don't be concerned (yet) about the possible race situations within the semaphore; assume that the activities they do are atomic. We'll employ locks and condition variables to accomplish this shortly.

Binary Semaphores (Locks)

We're all set to use a semaphore now. Our first use will employ a semaphore as a lock, which we are already familiar with. Figure 31.3 shows a code sample where we simply use a sem wait()/sem post() combination to surround the critical section of interest. However, the starting value of the semaphore m is critical to make this function (initialized to X in the figure). What should X stand for?

We can see from the sem wait() and sem post() definitions that the initial value should be 1.

Consider a situation with two threads to illustrate this point. The first thread (Thread 0) runs sem wait(), which decrements the semaphore's value to 0 first. Then, if the value is not larger than or equal to 0, it will wait. Finally, Sem wait() will just return 0, and the calling thread will continue; Thread 0 is now free to enter the critical region. If no other thread tries to acquire the lock while Thread 0 is in the crucial region, it will simply restore the semaphore's value to 1 when it runs sem post() (and not wake a waiting thread because there are none). This scenario is depicted in Figure 31.4.

When Thread 0 "holds the lock" (that is, it has called sem wait() but not yet called sem post()), and another thread (Thread 1) tries to access the crucial region by calling sem wait(), a more intriguing circumstance develops (). Thread 1 will decrement the value of the semaphore to -1 in this scenario and hence wait (putting itself to sleep and relinquishing the processor). When Thread 0 resumes execution, it will ultimately call sem post(), resetting the semaphore's value to zero, and then wake the waiting thread (Thread 1), allowing it to reclaim the lock. When Thread 1 completes, the semaphore's value will be incremented one more before reset to 1.

A trace of this example is shown in Figure 31.5. In addition to thread activities, the graphic depicts each thread's scheduler state: Run (the thread is running), Ready (the thread is runnable but not running), and Sleep (the thread is not running) (the thread is blocked). For example, when Thread 1 tries to acquire the already-held lock, it enters a sleeping state; Thread 1 can only be awoken and potentially run again when Thread 0 runs again.

Try a scenario where numerous threads wait for a lock if you wish to go through your own example. What would the semaphore's value be in such a trace?

As a result, semaphores can be used as locks. Because locks have just two states (held and unheld), a semaphore employed as a lock is frequently referred to as a binary semaphore. It's worth noting that if you're simply

utilizing a semaphore in this binary method, it might be written in a far more straightforward way than the generalized semaphores we've shown here.

Semaphores For Ordering

In a concurrent program, semaphores can also be used to order events.

A thread, for example, could want to wait for a list to become non-empty before deleting an entry from it. We frequently find one thread waiting for something to happen and another thread causing that event and then signaling that it has, thereby waking the waiting thread. As a result, the semaphore is used as an ordering primitive (similar to our use of condition variables earlier).

The following is a simple example. Consider a thread that spawns another thread and then wants to wait for it to finish (Figure 31.6). We'd like to observe the following when this software runs:

The challenge then becomes how to use a semaphore to achieve this effect, which turns out to be relatively simple. As you can see in the code, the parent simply runs sem wait(), and the child does sem post() to wait for the child's condition to become true when it completes its execution. First, however, this poses the question of what this semaphore's starting value should be.
(Rather of reading ahead, think about it now.)

The answer is, of course, that the semaphore's value should be set to 0. Consider the following two scenarios. First, assume that the parent creates the child, but it has not yet run (i.e., sitting in a ready queue but not running). The parent will call sem wait() before the child has called sem post() in this scenario (Figure 31.7, page 6); we want the parent to wait for the child to run. This may happen only if the semaphore's value is less than 0; thus, 0 is the starting value. Next, the parent runs, then waits while the semaphore is decremented (to -1). (sleeping). When the kid is finished, it will call sem post(), set the semaphore value to 0, and wake the parent, who will return from sem wait() and complete the program.

The second scenario (Figure 31.8) occurs when the child completes the task before the parent can call sem wait (). In this example, the child will run sem post() first, increasing the semaphore's value from 0 to 1. Then, when the parent is ready to start, it will call sem wait() and discover that the value of the semaphore is 1; the parent will then decrease the value (to 0) and exit sem wait() without waiting, producing the desired result.

The Producer/Consumer (Bounded Buffer) Problem

The producer/consumer problem, also known as the bounded buffer problem, is the next issue we'll tackle in this chapter. This problem is detailed in the previous chapter on condition variables; consult that chapter for more information.

We've now seen two different ways to start a semaphore. If we set the value to 1 for the first case, the semaphore will act as a lock, and if we set it to 0 for the second case, the semaphore will act as an ordering mechanism. So, what is the general semaphore startup rule?

A straightforward method is to consider how many resources you are willing to give away right after activation. For example, it was 1 with the lock because you were willing to have it locked (given away) right after startup. It was 0 in the ordering example because there was nothing to give away at the start; the resource is produced only when the child thread is finished, at which point the value is incremented to 1. Try this approach to future semaphore situations to see whether it works.

Attempt No. 1

The threads will employ two semaphores, empty and full, to signify when a buffer entry has been emptied or filled, respectively, in our first attempt to solve the problem. Figure 31.9 shows the code for the put and get procedures, and Figure 31.10 shows our attempt to solve the producer and consumer dilemma (page 8).

In this example, the producer waits for a buffer to become empty before filling it with data, while the consumer waits for a buffer to full before utilizing it. Let's pretend MAX=1 (there's just one buffer in the array) and see whether that works.

Consider that there are two threads, one for the manufacturer and one for the consumer. Let's look at a specific instance on a single processor. Assume that the customer is the one who gets to run first. As a result, the consumer will call sem wait on Line C1 in Figure 31.10. (&full). The call will decrement full (to -1), stall the consumer, and wait for another thread to perform sem post() on full, as desired because full was initialized to 0.

Assume the producer then exits the room. It will call the sem wait(&empty) routine when it reaches Line P1. Because empty was initialized to the value MAX, unlike the consumer, the producer will proceed through this line (in this case, 1). As a result, empty will be set to 0, and the processor will insert a data value into the buffer's first entry (Line P2). The processor will then proceed to P3 and call sem post(&full), which will change the full semaphore's value from -1 to 0 and wake the consumer (e.g., move it from blocked to ready).

One of two things could happen in this situation. First, if the producer keeps running, it will circle back around and hit Line P1 once more. This time, though, it would block since the value of the empty semaphore is 0. Second, if the consumer started running after the producer was interrupted, it would return from sem wait(&full) (Line C1), find that the buffer was full, and consume it. In each scenario, the intended behavior is achieved.

You can repeat this exercise with more threads (e.g., multiple producers and multiple consumers). It should still be functional.

Assume MAX is more than 1 (for example, MAX=10). Let's pretend that there are several producers and consumers in this scenario. We now have a race condition to deal with. Do you know where it happens? (spend some time looking for it) If you can't notice it, here's a hint: look at the put() and get() code more closely.

Let’s have a look at the problem. Consider two producers (Pa and Pb) who both call put() at the same time. Assume that producer Pa is the first to execute and begins by filling the first buffer entry (fill=0 at Line F1). Pa is halted before he has a chance to set the fill counter to 1. Then, producer Pb begins to execute, and at Line F1 also writes its data to the 0th element of buffer, overwriting the previous data! This is a no-no since we don’t want any of the producer’s data to be lost.

Adding Mutual Exclusion as a Solution

As you can see, we've forgotten about mutual exclusion in this case. Filling a buffer and incrementing the index into the buffer is a critical area that must be carefully handled. So, let's add some locks to our binary semaphore pal. Our attempt is depicted in Figure 31.11.

As stated by the NEW LINE remarks, we've now implemented some locks around the entire put()/get() parts of the code. That appears to be a good idea, but it doesn't function. Why? Deadlock. Deadlock occurs for a variety of reasons. Consider it for a moment; try to think of a situation when there is a standstill. What actions must occur for the program to reach a deadlock?

Avoiding Deadlock

Now that we've worked it out, here's the solution. Consider two threads: one for the producer and one for the consumer. The consumer is the one who gets to go first. It first acquires the mutex (Line C0), then calls sem wait() on the full semaphore (Line C1); because no data has yet been received, this call causes the consumer to block and hence yield the CPU; yet, the consumer retains the lock.

After that, a producer takes off. It has data to create, and if it could execute, it could wake the consumer thread, and everything would be OK. Unfortunately, it calls sem wait() on the binary mutex semaphore as the first thing it performs (Line P0). The key is already in the lock. As a result, the producer is now forced to wait as well.

There is a short cycle at work here. The mutex is held by the consumer, who is waiting for someone to signal full. The producer might signal full, but he's holding off until the mutex is released. As a result, both the manufacturer and the customer are locked in a classic impasse.

A Working Solution

To solve this problem, we simply must reduce the scope of the lock. Figure 31.12 shows the correct solution. We simply move the mutex acquire and release around the critical section; the full and empty wait and signal code are left outside.

The result is a working and straightforward bounded buffer, a commonly-used pattern in multithreaded programs.

Reader-Writer Locks

Another common issue is the desire for a more flexible locking primitive that acknowledges that different data structure accesses may necessitate different types of locking. For example, consider a series of concurrent list actions, such as inserts and primary lookups. While inserts modify the state of the list (and so require a typical critical section), lookups just read the data structure; as long as no insert is in progress, we can allow many lookups to run simultaneously. A reader-writer lock is a unique form of lock that we will now design to support this type of action. Figure 31.13 shows the code for this type of lock.

The code is straightforward. If a thread wants to update the data structure in question, it should use the new synchronization procedures rwlock acquire writelock() and rwlock release writelock() to acquire and release a write lock. Internally, these just employ the writelock semaphore to ensure that only one writer may obtain the lock and so enter the vital region of the data structure in question.

The pair of methods for acquiring and releasing read locks is more intriguing. The reader acquires a read lock first, then increments the readers variable to track how many readers are currently within the data structure. When the first reader receives the lock, the reader must also acquire the write lock by executing sem wait() on the writelock semaphore and then releasing the lock by calling sem post().

As a result, other readers will have access to the read lock once the first has done so. On the other hand, if a writer wishes to access the write lock, it must wait until all readers have completed their tasks. The last one to exit the critical section calls sem post() on “writelock,” allowing a waiting writer to acquire the lock.

This strategy works (as expected), but it has certain drawbacks, particularly fairness. It would be relatively easy for readers to starve writers in particular. There are more advanced solutions to this problem; perhaps you can come up with a better implementation? Consider what you’d have to do to prevent other readers from accessing the lock while a writer is waiting.

Finally, it’s important to remember that reader-writer locks should only be utilized with discretion. They frequently add extra complexity (especially with more complex implementations) and do not improve performance compared to quick and straightforward locking primitives. In any case, they demonstrate how we might use semaphores in new and intriguing ways.

Dining philosophers

The dining philosopher's problem is one of the most well-known concurrency problems. The problem is well-known because it is entertaining and intellectually stimulating.

It has a limited practical utility. However, its notoriety necessitates its inclusion here.

The problem's basic setup is as follows (as shown in Figure 31.14): Assume a group of five "philosophers" is gathered around a table. A single fork separates each pair of philosophers (and thus, five total). Each philosopher has times when they think and don't require forks and times when they eat. A philosopher requires two forks, one on their left and one on their right, to eat. We explore this subject in concurrent programming because of the competition for these forks and the synchronization issues that result.

The following is each philosopher's primary loop, assuming each has a unique thread identification p ranging from 0 to 4 (inclusive):

The significant difficulty is to construct the get forks() and put forks() procedures so that there is no deadlock, no philosopher goes hungry and never eats. Concurrency is high (i.e., as many philosophers as possible can eat at the same time).

We'll employ a few utility functions to get us closer to a solution as Downey did. They are as follows:

When philosopher p wants to talk about the fork to their left, they just say left (p). Similarly, calling right(p) refers to the fork to a philosopher's right; the modulo operator therein deals with the one instance where the final philosopher (p=4) wants to grasp the fork to their right, fork 0.

To fix this problem, we'll also need some semaphores. For example, assume we have five forks, one for each: sem t forks.

A Broken Solution

We make our first attempt at solving the challenge. Assume we set the value of each semaphore (in the forks array) to 1. Assume that each philosopher has a unique number (p). As a result, the get forks() and put forks() routines can be written (Figure 31.15, page 15).

The following is the reasoning behind this (broken) solution. First, to get the forks, we just grasp a “lock” on each one, starting with the left and working our way to the right. Then, we let them go once we’ve finished eating.

Isn’t it simple? Unfortunately, simplicity means broken in this situation. Are you able to see the issue that arises? Consider that for a moment.

Deadlock is the issue. If they grab their left fork before their right fork, everyone will have to wait for another for the rest of their lives. Philosopher 0 takes fork 0, philosopher 1 takes fork 1, philosopher 2 takes fork 2, philosopher 3 takes fork 3, and philosopher 4 takes fork 4; all forks have been taken, and all philosophers are trapped waiting for a fork that another philosopher has. We’ll look into deadlock in more depth later; for now, it’s reasonable to say that this isn’t a viable option.

Breaking the Dependency

Changing how at least one of the philosophers acquires forks is the simplest way to attack this challenge. Let's pretend that philosopher 4 (the highest-numbered one) receives the forks in a different order than the others (Figure 31.16); the put forks() function remains unchanged.

There is no case where one philosopher grabs one fork and is stuck waiting for another because the final philosopher tries to grasp right before left; the waiting cycle is broken. Consider the implications of this solution and persuade yourself that it is viable.

Other "renowned" difficulties, such as the cigarette smoker's dilemma or the sleeping barber's problem, exist. Most of these are only pretexts for thinking about concurrency; nonetheless, some of them have intriguing titles.

Thread Throttling

Another important use case for semaphores happens from time to time, and we've included it here. The difficulty is this: how can a programmer prevent "too many" threads from running simultaneously and slowing down the system? Answer: set a threshold for "too many" and then use a semaphore to limit the number of threads running the code in questionsimultaneously simultaneously. This method is known as throttling, and it is a type of admission control.

Let's take a look at a more concrete case. Consider creating hundreds of threads to work on a problem concurrently. However, each thread gets a considerable amount of memory to conduct part of the computation in a specific section of the code, which we'll refer to as the memory-intensive region. The aggregate of all memory allocation requests will surpass the amount of physical memory on the machine if all threads access the memory-intensive section simultaneously. As a result, the machine will begin to thrash (i.e., switching pages to and from the disk), slowing down the entire computation.

This can be solved with a simple semaphore. You can configure a semaphore to throttle the number of threads that enter the memory-intensive region by initializing its value to the maximum number of threads you wish to enter the region at once and then put sem wait() sem post() around the region.

Implementing Semaphores

Finally, we'll leverage our low-level synchronization primitives, such as locks and condition variables, to create our version of semaphores, which we'll call Zemaphores. As you can see in Figure 31.17, this task is quite simple (page 17).

Only one lock and one condition variable are used in the code above, together with a state variable to track the semaphore's value. You should study the code until you have a firm grasp of it.

Our Zemaphore differs from pure semaphores. We don't retain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads; in fact, the value will never be lower than zero. This behavior is more straightforward to implement and corresponds to the current Linux implementation.

Constructing condition variables from semaphores, on the other hand, is a far more difficult task. Some highly experienced concurrent programmers attempted to achieve this in the Windows environment, and a slew of issues resulted. Try it for yourself to discover why creating condition variables from semaphores is more complex than it appears.

Summary

For building concurrent applications, semaphores are a solid and flexible primitive. Because of their simplicity and utility, some programmers use them solely, avoiding locks and condition variables.

We've only covered a few classic problems and answers in this chapter. There are plenty of other resources available if you are interested in learning more. For example, Allen Downey's book on concurrency and programming with semaphores is a fantastic (and free) resource. This book has many problems that you can work on to increase your understanding of semaphores in particular and concurrency in general.