

## Semaphores

As we know now<sup>1</sup>, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems. One of the first people to realize this years ago was **Edsger Dijkstra** (though it is hard to know the exact history [GR92]), known among other things for his famous “shortest paths” algorithm in graph theory [D59], an early polemic on structured programming entitled “Goto Statements Considered Harmful” [D68a] (what a great title!), and, in the case we will study here, the introduction of a powerful and flexible synchronization primitive known as the **semaphore** [D68b,D72]. Indeed, he invented this general semaphore as a single primitive for all things related to synchronization; as you will see, one can use semaphores as both locks and condition variables.

### THE CRUX: HOW TO USE SEMAPHORES

How can we use semaphores instead of locks and condition variables? What is the definition of a semaphore? What is a binary semaphore? Is it straightforward to build a semaphore out of locks and condition variables? What about building locks and condition variables out of semaphores?

---

<sup>1</sup>That is, assuming “we”, which is actually “you”, have been reading and paying attention to the previous chapters. And even remembering a few things. You have, right?!

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Figure 30.1: Initializing A Semaphore

```
int sem_wait(sem_t *s) {
    wait until value of semaphore s is greater than 0
    decrement the value of semaphore s by 1
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by 1
    if there are 1 or more threads waiting, wake 1
}
```

Figure 30.2: Semaphore: Definitions of Wait and Post

### 30.1 Semaphores: A Definition

A semaphore is as an object with an integer value that we can manipulate with two routines (which we will call `sem_wait()` and `sem_post()` to follow the POSIX standard). 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, as the code in Figure 30.1 does.

In the figure, we declare a semaphore `s` and initialize it to the value of 1. You can ignore the second argument to `sem_init()` for now; read the man page for details.

After a semaphore is initialized, we can call one of two functions to interact with it, `sem_wait()` or `sem_post()`<sup>2</sup>. The behavior of these two functions is seen in Figure 30.2.

For now, we are not concerned with the implementation of these routines, which clearly requires some care; with multiple threads calling into `sem_wait()` and `sem_post()`, there is the obvious need for managing these critical sections with locks and queues similar to how we previously built locks. We will now focus on how to *use* these primitives; later we may discuss how they are built.

A couple of notes. First, we can see that `sem_wait()` will ei-

---

<sup>2</sup>Historically, `sem_wait()` was first called P() by Dijkstra (for the Dutch word “to probe”) and `sem_post()` was called V() (for the Dutch word “to test”). Sometimes, people call them down and up, too. Use the Dutch versions to impress your friends.

then return right away (because the value of the semaphore was 1 or higher when we called `sem_wait()`), or it will cause the caller to suspend execution waiting for a subsequent post. Of course, multiple calling threads may call into `sem_wait()`, and thus all be queued waiting to be woken. Once woken, the waiting thread will then decrement the value of the semaphore and return to the user.

Second, we can see that `sem_post()` does not ever suspend the caller. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes 1 of them up.

You should not worry here about the seeming race conditions possible within the semaphore; assume that the modifications they make to the state of the semaphore are all performed atomically (we will soon use locks and condition variables to do just this).

## 30.2 Binary Semaphores (Locks)

We are now ready to use a semaphore. Our first use will be one with which we are already familiar: using a semaphore as a lock. Here is a code snippet:

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

Figure 30.3: A Binary Semaphore, a.k.a. a Lock

To build a lock, we simply surround the critical section of interest with a `sem_wait()/sem_post()` pair. Critical to making this work, though, is the initial value of the semaphore. What should it be?

If we look back at the definition of the `sem_wait()` and `sem_post()` routines above, we can see that the initial value of the semaphore should be 1. Imagine the first thread (thread 0) calling `sem_wait()`; it will first wait for the value of the semaphore to be greater than 0, which it is (the semaphore's value is 1). It will thus not wait at all and decrement the value to 0 before returning to the caller. That thread 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

```

void *
child(void *arg) {
    printf("child\n");
    // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(c, NULL, child, NULL);
    // wait here for child
    printf("parent: end\n");
    return 0;
}

```

Figure 30.4: A Parent Waiting for its Child

to 1 (and not wake any waiting thread, because there are no waiting threads).

The more interesting case arises when thread 0 holds the lock (i.e., it has called `sem_wait()` but not yet called `sem_post()`), and another thread (thread 1, say) tries to enter the critical section by calling `sem_wait()`. In this case, thread 1 will find that the value of the semaphore is 0, and thus wait (putting itself to sleep and relinquishing the processor). When thread 0 runs again, it will eventually call `sem_post()`, incrementing the value of the semaphore back to 1, and then wake the waiting thread 1, which will then be able to acquire the lock for itself.

In this basic way, we are able to use semaphores as locks. Because the value of the semaphore simply alternates between 1 and 0, this usage is sometimes known as a **binary semaphore**. Yes, this is something you just have to remember; life is unfair that way sometimes.

### 30.3 Semaphores As Condition Variables

Semaphores are also useful when a thread wants to halt its own progress waiting for something to change. For example, a thread may wish to wait for a list to become non-empty, so that it can take an element off of the list. In this pattern of usage, we often find a thread *waiting* for something to happen, and a different thread mak-

ing that something happen and then *signaling* that it has indeed happened, thus waking the waiting thread. Because the waiting thread (or threads, really) is waiting for some **condition** in the program to change, we are using the semaphore as a **condition variable**.

A simple example is as follows. Imagine a thread creates another thread and then wants to wait for it to complete its execution (Figure 30.4). When this program runs, what we would like to see here is the following output:

```
parent: begin
child
parent: end
```

The question, then, is how to use a semaphore to achieve this effect, and as it turns out, it is relatively easy to understand (Figure 30.5). As you can see in the code, 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. However, this raises the question: what should the initial value of this semaphore be? (think about it here, instead of reading ahead)

The answer, of course, is that the value of the semaphore should be set to be the number 0. There are two cases to consider. First, let us assume that the parent creates the child but the child has not run yet (i.e., it is sitting in a ready queue but not running). In this case, the parent will call `sem_wait()` before the child has called `sem_post()`, and thus we'd like the parent to wait for the child to run. The only way this will happen is if the value of the semaphore is not greater than 0; hence, 0 as the initial value makes sense. When the child finally runs, it will call `sem_post()`, incrementing the value to 1 and waking the parent, which will then return from `sem_wait()` and complete the program.

The second case occurs when the child runs to completion before the parent gets a chance to call `sem_wait()`. In this case, the child will first call `sem_post()`, thus incrementing the value of the semaphore from 0 to 1. When the parent then gets a chance to run, it will call `sem_wait()` and find the value of the semaphore to be 1; the parent will thus decrement the value and return from `sem_wait()` without waiting, also achieving the desired effect.

```

sem_t s;

void *
child(void *arg) {
    printf("child\n");
    // signal here: child is done
    sem_post(&s);
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(c, NULL, child, NULL);
    // wait here for child
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}

```

Figure 30.5: Parent Waiting for its Child with Semaphores

## 30.4 The Producer/Consumer (Bounded-Buffer) Problem

The next problem we will confront in this chapter is known as the **producer/consumer** problem, or sometimes as the **bounded buffer** problem [D72]. This problem is described in detail in the previous chapter on condition variables; see there for details.

### First Attempt

Our first attempt at solving the problem introduces two semaphores, `empty` and `full`, which the threads will use to indicate when a buffer entry has been emptied or filled, respectively. The code for the `put` and `get` routines is in Figure 30.6, and our attempt at solving the producer and consumer problem is in Figure 30.7.

In this example, the producer first waits for a buffer to become empty in order to put data into it, and the consumer similarly waits for a buffer to become filled before using it. Let us first imagine that `MAX=1` (there is only one buffer in the array), and see if this works.

Imagine again there are two threads, a producer and a consumer. Let us examine a specific scenario on a single CPU. Assume the con-

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
    buffer[fill] = value;    // line f1
    fill = (fill + 1) % MAX; // line f2
}

int get() {
    int tmp = buffer[use];    // line g1
    use = (use + 1) % MAX;    // line g2
    return tmp;
}
```

Figure 30.6: The Put and Get Routines

sumer gets to run first. Thus, the consumer will hit line c1 in the figure above, calling `sem_wait(&full)`. Because `full` was initialized to the value 0, the call will block the consumer and wait for another thread to call `sem_post()` on the semaphore, as desired.

Assume the producer then runs. It will hit line P1, thus calling `sem_wait(&empty)`. Unlike the consumer, the producer will continue through this line, because `empty` was initialized to the value `MAX` (in this case, 1). Thus, `empty` will be decremented to 0 and the producer will put a data value into the first entry of `buffer` (line P2). The producer will then continue on to P3 and call `sem_post(&full)`, changing the value of the `full` semaphore from 0 to 1 and waking the consumer (e.g., move it from blocked to ready).

In this case, one of two things could happen. If the producer continues to run, it will loop around and hit line P1 again. This time, however, it would block, as the `empty` semaphore's value is 0. If the producer instead was interrupted and the consumer began to run, it would call `sem_wait(&full)` (line c1) and find that the `buffer` was indeed full and thus consume it. In either case, we achieve the desired behavior.

You can try this same example with more threads (e.g., multiple producers, and multiple consumers). It should still work, or it is time to go to sleep.

Let us now imagine that `MAX` is greater than 1 (say `MAX = 10`). For this example, let us assume that there are multiple producers and multiple consumers. We now have a problem: a race condition. Do you see where it occurs? (take some time and look for it) If you can't see it, here's a hint: look more closely at the `put()` and `get()` code.

```

sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line p1
        put(i);                     // line p2
        sem_post(&full);            // line p3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);            // line c1
        tmp = get();                // line c2
        sem_post(&empty);           // line c3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    // ...
}

```

Figure 30.7: Adding the Full and Empty Conditions

OK, let's understand the issue. Imagine two producers (Pa and Pb) both calling into `put()` at roughly the same time. Assume producer Pa gets to run first, and just starts to fill the first buffer entry (fill = 0 at line f1). Before Pa gets a chance to increment the fill counter to 1, it is interrupted. Producer Pb starts to run, and at line f1 it also puts its data into the 0th element of buffer, which means that the old data there is overwritten! This is a no-no; we don't want any data generated by a producer to be lost.

### A Solution: Adding Mutual Exclusion

As you can see, what we've forgotten here is *mutual exclusion*. The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded carefully. So let's use our friend the binary semaphore and add some locks. Figure 30.8 shows our attempt.

```

sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // line p0 (NEW LINE)
        sem_wait(&empty);           // line p1
        put(i);                     // line p2
        sem_post(&full);            // line p3
        sem_post(&mutex);           // line p4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // line c0 (NEW LINE)
        sem_wait(&full);            // line c1
        int tmp = get();            // line c2
        sem_post(&empty);           // line c3
        sem_post(&mutex);           // line c4 (NEW LINE)
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
    // ...
}

```

Figure 30.8: Adding Mutual Exclusion (Incorrectly)

Now we've added some locks around the entire `put()/get()` parts of the code, as indicated by the `NEW LINE` comments. That seems like the right idea, but it also doesn't work. Why? Deadlock. Why does deadlock occur? Take a moment to consider it; try to find a case where deadlock arises. What sequence of steps must happen for the program to deadlock?

## Avoiding Deadlock

OK, now that you figured it out, here is the answer. Imagine two threads, one producer and one consumer. The consumer gets to run first. It acquires the mutex (line c0), and then calls `sem_wait()` on

```

sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line p1
        sem_wait(&mutex);           // line p1.5 (MOVED MUTEX HERE...)
        put(i);                     // line p2
        sem_post(&mutex);           // line p2.5 (... AND HERE)
        sem_post(&full);            // line p3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);            // line c1
        sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE...)
        int tmp = get();            // line c2
        sem_post(&mutex);           // line c2.5 (... AND HERE)
        sem_post(&empty);           // line c3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // ...
}

```

Figure 30.9: Adding Mutual Exclusion (Correctly)

the full semaphore (line c1); because there is no data yet, this call causes the consumer to block and thus yield the CPU; importantly, though, the consumer still holds the lock.

A producer then runs. It has data to produce and if it were able to run, it would be able to wake the consumer thread and all would be good. Unfortunately, the first thing it does is call `sem_wait()` on the binary mutex semaphore (line p0). The lock is already held. Hence, the producer is now stuck waiting too.

There is a simple cycle here. The consumer *holds* the mutex and is *waiting* for the someone to signal full. The producer could *signal* full but is *waiting* for the mutex. Thus, the producer and consumer are each stuck waiting for each other: a classic deadlock.

## Finally, A Working Solution

To solve this problem, we simply must reduce the scope of the lock. Figure 30.9 shows the final working solution. As you can see, we simply move the mutex acquire and release to be just around the critical section; the full and empty wait and signal code is left outside. The result is a simple and working bounded buffer, a commonly-used pattern in multithreaded programs. Understand it now; use it later. You will thank us for years to come. Or at least, you will thank us when the same question is asked on the final exam.

### 30.5 Reader-Writer Locks

Another classic problem stems from the desire for a more flexible locking primitive that admits that different data structure accesses might require different kinds of locking. For example, imagine a number of concurrent list operations, including inserts and simple lookups. While inserts change the state of the list (and thus a traditional critical section makes sense), 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. The special type of lock we will now develop to support this type of operation is known as a **reader-writer lock** [CHP71]. The code for such a lock is available in Figure 30.10.

The code is pretty simple. 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. Internally, these simply use the `writelock` semaphore to ensure that only a single writer can acquire the lock and thus enter the critical section to update the data structure in question.

More interesting is the pair of routines to acquire and release read locks. 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 finally releasing the lock by calling `sem_post()`.

```

typedef struct _rwlock_t {
    sem_t lock;        // binary semaphore (basic lock)
    sem_t writelock;   // used to allow ONE writer or MANY readers
    int    readers;     // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock); // first reader acquires writelock
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock); // last reader releases writelock
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock);
}

```

Figure 30.10: A Simple Reader-Writer Lock

Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until *all* readers are finished; the last one to exit the critical section calls `sem_post()` on “writelock” and thus enables a waiting writer to acquire the lock.

This approach works (as desired), but does have some negatives, especially when it comes to fairness. In particular, it would be relatively easy for readers to starve writers. More sophisticated solutions to this problem exist; perhaps you can think of a better implementation? Hint: think about what you would need to do to prevent more readers from entering the lock once a writer is waiting.

Finally, it should be noted that reader-writer locks should be used with some caution. They often add more overhead (especially with more sophisticated implementations), and thus do not end up speeding up performance as compared to just using simple and fast locking primitives [CB08]. Either way, they showcase once again how we can use semaphores in an interesting and useful way.

**TIP: SIMPLE AND DUMB CAN BE BETTER (HILL'S LAW)**

You should never underestimate the notion that the simple and dumb approach can be the best one. With locking, sometimes a simple spin lock works best, because it is easy to implement and fast. Although something like reader/writer locks sounds cool, they are complex, and complex can mean slow. Thus, always try the simple and dumb approach first.

This idea, of appealing to simplicity, is found in many places. One early source is Mark Hill's dissertation [H87], which studied how to design caches for CPUs. Hill found that simple direct-mapped caches worked better than fancy set-associative designs (one reason is that in caching, simpler designs enable faster lookups). As Hill succinctly summarized his work: "Big and dumb is better." And thus we call this similar advice **Hill's Law**.

## 30.6 The Dining Philosophers

One of the most famous concurrency problems posed, and solved, by Dijkstra, is known as the **dining philosopher's problem** [DH071]. The problem is famous because it is fun and somewhat intellectually interesting; however, its practical utility is low. However, its fame forces its inclusion here; indeed, you might be asked about it on some interview, and you'd really hate your OS professor if you miss that question and don't get the job. Conversely, if you get the job, please feel free to send your OS professor a nice note, or some stock options.

The basic setup for the problem is this (as shown in Figure 30.11): assume there are five "philosophers" sitting around a table. Between each pair of philosophers is a single fork (and thus, five total). The philosophers each have times where they think, and don't need any forks, and times where they eat. In order to eat, a philosopher needs

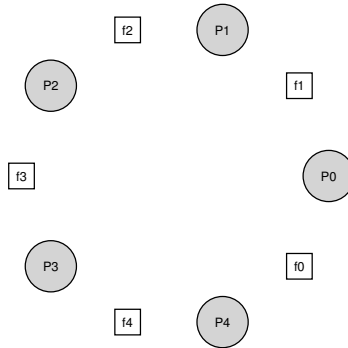


Figure 30.11: The Dining Philosophers

two forks, both the one on their left and the one on their right. The contention for these forks, and the synchronization problems that ensue, are what makes this a problem we study in concurrent programming. Here is the basic loop of each philosopher:

```
while (1) {
    think();
    getforks();
    eat();
    putforks();
}
```

The key challenge, then, is to write the routines `getforks()` and `putforks()` such that there is no deadlock, no philosopher starves and never gets to eat, and concurrency is high (i.e., as many philosophers can eat at the same time as possible).

Following Downey's solutions [D08], we'll use a few helper functions to get us towards a solution. They are:

```
int left(int p) { return p; }
int right(int p) { return (p + 1) % 5; }
```

```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

void putforks() {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

Figure 30.12: The `getforks()` and `putforks()` Routines

When philosopher `p` wishes to refer to the fork on their left, they simply call `left(p)`. Similarly, the fork on the right of a philosopher `p` is referred to by calling `right(p)`; the modulo operator therein handles the one case where the last philosopher (`p=4`) tries to grab the fork on their right, which is fork 0.

We'll also need some semaphores to solve this problem. Let us assume we have five, one for each fork: `sem_t forks[5]`.

## Broken Solution

We can now attempt our first solution to the problem. Assume we initialize each semaphore (in the `forks` array) to have an initial value of 1. Assume also that each philosopher knows its own number, which we refer to as `p`. We can thus write the `getforks()` and `putforks()` routine as follows (Figure 30.12).

The intuition behind this (broken) solution is as follows. 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. Simple, no? Unfortunately, in this case, simple means broken. Can you see the problem that arises? Think about it.

The problem is **deadlock**. If each philosopher happens to grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another, forever. Specifically, philosopher 0 grabs fork 0, philosopher 1 grabs fork 1, philosopher 2 grabs fork 2, philosopher 3 grabs fork 3, and philosopher 4 grabs fork 4; all the forks are acquired, and all the philosophers are stuck waiting for a fork that another philosopher possesses. We'll study deadlock in more detail soon; for now, it is safe to say that this is not a working solution.

### A Solution: Breaking The Dependency

The simplest way to attack this problem is to change how forks are acquired by at least one of the philosophers; indeed, this is how Dijkstra himself solved the problem. Specifically, let's assume that philosopher 4 (the highest numbered one) acquires the forks in a *different* order. The code to do so is as follows:

```
void getforks() {
    if (p == 4) {
        sem_wait(forks[right(p)]);
        sem_wait(forks[left(p)]);
    } else {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
    }
}
```

Because the last philosopher tries to grab the right fork before the left, you can never create a situation where each philosopher grabs one fork and is stuck waiting for another; thus, the cycle of waiting is broken. Think through the ramifications of this simple solution, and convince yourself that it works.

There are other “famous” problems like this one, e.g., the **cigarette smoker’s problem** or the **sleeping barber problem**. Most of them are just excuses to think about concurrency; look them up if you are interested in learning more [D08].

## 30.7 How To Implement Semaphores

Finally, let's use our low-level synchronization primitives, locks and condition variables, to build semaphores. It is fairly straightforward (see Figure 30.13).

As you can see from the figure, it is pretty simple. Just one lock and one condition variable, plus a state variable to track the value of the semaphore, is all you need. Study the code for yourself until you really understand it. Do it!

Curiously, building locks and condition variables out of standard semaphores is a much trickier proposition. Some highly experienced concurrent programmers tried to do this in the Windows environment, and many different bugs ensued [B04]. Try it yourself, and see if you can figure out why it seems harder to do.

```
typedef struct __Sem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Sem_t;

// only one thread can call this
void Sem_init(Sem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Sem_wait(Sem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Sem_post(Sem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Figure 30.13: Implementing Semaphores with Locks and CVs

## 30.8 Summary

Semaphores are a powerful and flexible primitive for writing concurrent programs. Some programmers use them exclusively, shunning locks and condition variables, due to their simplicity and utility.

In this chapter, we have presented just a few classic problems and solutions. If you are interested in finding out more, there are many other materials you can reference. One great (and free reference) is Allen Downey's book on concurrency and programming with semaphores [D08]. This book has lots of puzzles you can work on to improve your understanding of both semaphores in specific and concurrency in general. Becoming a real concurrency expert takes years of effort; going beyond what you learn in this class is undoubtedly the key to mastering such a topic.

## References

[B04] “Implementing Condition Variables with Semaphores”

Andrew Birrell  
December 2004

*An interesting read on how difficult implementing CVs on top of semaphores really is, and the mistakes the author and co-workers made along the way. Particularly relevant because the group had done a ton of concurrent programming; Birrell, for example, is known for (among other things) writing various thread-programming guides.*

[CB08] “Real-world Concurrency”

Bryan Cantrill and Jeff Bonwick  
ACM Queue. Volume 6, No. 5. September 2008

*A nice article by some kernel hackers from a company formerly known as Sun on the real problems faced in concurrent code.*

[CHP71] “Concurrent Control with Readers and Writers”

P.J. Courtois, F. Heymans, D.L. Parnas  
Communications of the ACM, 14:10, October 1971

*The introduction of the reader-writer problem, and a simple solution. Later work introduced more complex solutions, skipped here because, well, they are pretty complex.*

[D59] “A Note on Two Problems in Connexion with Graphs”

E. W. Dijkstra  
Numerische Mathematik 1, 269271, 1959

Available: <http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf>  
*Can you believe people worked on algorithms in 1959? We can't. Even before computers were any fun to use, these people had a sense that they would transform the world...*

[D68a] “Go-to Statement Considered Harmful”

E.W. Dijkstra  
Communications of the ACM, volume 11(3): pages 147148, March 1968  
Available: <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>  
*Sometimes thought as the beginning of the field of software engineering.*

[D68b] “The Structure of the THE Multiprogramming System”

E.W. Dijkstra  
Communications of the ACM, volume 11(5), pages 341346, 1968  
*One of the earliest papers to point out that systems work in computer science is an engaging intellectual endeavor. Also argues strongly for modularity in the form of layered systems.*

[D72] "Information Streams Sharing a Finite Buffer"

E.W. Dijkstra

Information Processing Letters 1: 179180, 1972

Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF>

*Did Dijkstra invent everything? No, but maybe close. He certainly was the first to clearly write down what the problems were in concurrent code. However, it is true that practitioners in operating system design knew of many of the problems described by Dijkstra, so perhaps giving him too much credit would be a misrepresentation of history.*

[D08] "The Little Book of Semaphores"

A.B. Downey

Available: <http://greenteapress.com/semaphores/>

*A nice (and free!) book about semaphores. Lots of fun problems to solve, if you like that sort of thing.*

[DHO71] "Hierarchical ordering of sequential processes"

E.W. Dijkstra

Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF>

*Presents numerous concurrency problems, including the Dining Philosophers. The wikipedia page about this problem is also quite informative.*

[GR92] "Transaction Processing: Concepts and Techniques"

Jim Gray and Andreas Reuter

Morgan Kaufmann, September 1992

*The exact quote that we find particularly humorous is found on page 485, at the top of Section 8.8: "The first multiprocessors, circa 1960, had test and set instructions ... presumably the OS implementors worked out the appropriate algorithms, although Dijkstra is generally credited with inventing semaphores many years later."*

[H87] "Aspects of Cache Memory and Instruction Buffer Performance"

Mark D. Hill

Ph.D. Dissertation, U.C. Berkeley, 1987

*Hill's dissertation work, for those obsessed with caching in early systems. A great example of a quantitative dissertation.*