31

Semaphores

As we know now, 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 synchronization primitive called the **semaphore** [D68b,D72]. Indeed, Dijkstra and colleagues invented the 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? To build locks and condition variables out of 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()`[1]. 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 31.1 does.

---

[1]Historically, `sem_wait()` was called P() by Dijkstra and `sem_post()` called V(). P() comes from "prolaag", a contraction of "probeer" (Dutch for "try") and "verlaag" ("decrease"); V() comes from the Dutch word "verhoog" which means "increase" (thanks to Mart Oskamp for this information). Sometimes, people call them down and up. Use the Dutch versions to impress your friends, or confuse them, or both.

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

Figure 31.1: **Initializing A Semaphore**

In the figure, we declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument to `sem_init()` will be set to 0 in all of the examples we'll see; this indicates that the semaphore is shared between threads in the same process. See the man page for details on other usages of semaphores (namely, how they can be used to synchronize access across *different* processes), which require a different value for that second argument.

After a semaphore is initialized, we can call one of two functions to interact with it, `sem_wait()` or `sem_post()`. The behavior of these two functions is seen in Figure 31.2.

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. We will now focus on how to *use* these primitives; later we may discuss how they are built.

We should discuss a few salient aspects of the interfaces here. 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()`), 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.

Second, we can see that `sem_post()` does not wait for some particular condition to hold like `sem_wait()` does. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

Third, the value of the semaphore, when negative, is equal to the number of waiting threads [D68b]. Though the value generally isn't seen by users of the semaphores, this invariant is worth knowing and perhaps can help you remember how a semaphore functions.

Don't worry (yet) about the seeming race conditions possible within the semaphore; assume that the actions they make are performed atomically. We will soon use locks and condition variables to do just this.

```
1   int sem_wait(sem_t *s) {
2       decrement the value of semaphore s by one
3       wait if value of semaphore s is negative
4   }
5
6   int sem_post(sem_t *s) {
7       increment the value of semaphore s by one
8       if there are one or more threads waiting, wake one
9   }
```

Figure 31.2: **Semaphore: Definitions Of Wait And Post**

```
1   sem_t m;
2   sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4   sem_wait(&m);
5   // critical section here
6   sem_post(&m);
```

Figure 31.3: **A Binary Semaphore (That Is, A Lock)**

31.2 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. See Figure 31.3 for a code snippet; therein, you'll see that 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 m (initialized to X in the figure). What should X be?

... (Try thinking about it before going on) ...

Looking back at definition of the sem_wait() and sem_post() routines above, we can see that the initial value 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 (and not wake a waiting thread, because there are none). Figure 31.4 shows a trace of this scenario.

A 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) tries to enter the critical section by calling sem_wait(). In this case, Thread 1 will decrement the value of the semaphore to -1, 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 zero, and then wake the waiting thread (Thread 1), which will then be able to acquire the lock for itself. When Thread 1 finishes, it will again increment the value of the semaphore, restoring it to 1 again.

Value of Semaphore	Thread 0	Thread 1
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Figure 31.4: **Thread Trace: Single Thread Using A Semaphore**

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect:  begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect:  end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	`sem_wait()` returns	Running
0		Ready	`(crit sect)`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`sem_post()` returns	Running

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

Figure 31.5 shows a trace of this example. In addition to thread actions, the figure shows the **scheduler state** of each thread: Running, Ready (i.e., runnable but not running), and Sleeping. Note in particular that Thread 1 goes into the sleeping state when it tries to acquire the already-held lock; only when Thread 0 runs again can Thread 1 be awoken and potentially run again.

If you want to work through your own example, try a scenario where multiple threads queue up waiting for a lock. What would the value of the semaphore be during such a trace?

Thus we are able to use semaphores as locks. Because locks only have two states (held and not held), we sometimes call a semaphore used as a lock a **binary semaphore**. Note that if you are using a semaphore only in this binary fashion, it could be implemented in a simpler manner than the generalized semaphores we present here.

31.3 Semaphores For Ordering

Semaphores are also useful to order events in a concurrent program. For example, a thread may wish to wait for a list to become non-empty, so it can delete an element from it. In this pattern of usage, 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 (similar to our use of **condition variables** earlier).

```
1   sem_t s;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
6       sem_post(&s); // signal here: child is done
7       return NULL;
8   }
9
10  int
11  main(int argc, char *argv[]) {
12      sem_init(&s, 0, X); // what should X be?
13      printf("parent: begin\n");
14      pthread_t c;
15      Pthread_create(&c, NULL, child, NULL);
16      sem_wait(&s); // wait here for child
17      printf("parent: end\n");
18      return 0;
19  }
```

Figure 31.6: **A Parent Waiting For Its Child**

A simple example is as follows. Imagine a thread creates another thread and then wants to wait for it to complete its execution (Figure 31.6). When this program runs, we would like to see the following:

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

The question, then, is how to use a semaphore to achieve this effect; as it turns out, the answer is relatively easy to understand. 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?

(Again, think about it here, instead of reading ahead)

The answer, of course, is that the value of the semaphore should be set to is 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 (Figure 31.7, page 6), the parent will call sem_wait() before the child has called sem_post(); 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 is the initial value. The parent runs, decrements the semaphore (to -1), then waits (sleeping). When the child finally runs, it will call sem_post(), increment the value of the semaphore to 0, and wake the parent, which will then return from sem_wait() and finish the program.

The second case (Figure 31.8, page 6) 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 (to 0) and return from sem_wait() without waiting, also achieving the desired effect.

Value	Parent	State	Child	State
0	create(Child)	Running	*(Child exists; is runnable)*	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem<0)→sleep	Sleeping		Ready
-1	*Switch→Child*	Sleeping	child runs	Running
-1		Sleeping	call sem_post()	Running
0		Sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	sem_wait() returns	Running		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Value	Parent	State	Child	State
0	create(Child)	Running	*(Child exists; is runnable)*	Ready
0	*Interrupt; Switch→Child*	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	*Interrupt; Switch→Parent*	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready
0	(sem≥0)→awake	Running		Ready
0	sem_wait() returns	Running		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

31.4 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 31.9, and our attempt at solving the producer and consumer problem is in Figure 31.10.

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 consumer gets to run first. Thus, the consumer will hit Line C1 in Figure 31.10, calling sem_wait(&full). Because full was initialized to the value 0,

```
1   int buffer[MAX];
2   int fill = 0;
3   int use  = 0;
4
5   void put(int value) {
6       buffer[fill] = value;    // Line F1
7       fill = (fill + 1) % MAX; // Line F2
8   }
9
10  int get() {
11      int tmp = buffer[use];   // Line G1
12      use = (use + 1) % MAX;   // Line G2
13      return tmp;
14  }
```

Figure 31.9: **The Put And Get Routines**

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           sem_wait(&empty);          // Line P1
8           put(i);                    // Line P2
9           sem_post(&full);           // Line P3
10      }
11  }
12
13  void *consumer(void *arg) {
14      int i, tmp = 0;
15      while (tmp != -1) {
16          sem_wait(&full);           // Line C1
17          tmp = get();               // Line C2
18          sem_post(&empty);          // Line C3
19          printf("%d\n", tmp);
20      }
21  }
22
23  int main(int argc, char *argv[]) {
24      // ...
25      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26      sem_init(&full, 0, 0);    // ... and 0 are full
27      // ...
28  }
```

Figure 31.10: **Adding The Full And Empty Conditions**

the call will decrement full (to -1), block the consumer, and wait for another thread to call sem_post() on full, as desired.

Assume the producer then runs. It will hit Line P1, thus calling the sem_wait(&empty) routine. 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 -1 to 0 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.

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.

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 from the 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 31.11 shows our attempt.

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 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.

```
1    sem_t empty;
2    sem_t full;
3    sem_t mutex;
4
5    void *producer(void *arg) {
6        int i;
7        for (i = 0; i < loops; i++) {
8            sem_wait(&mutex);           // Line P0 (NEW LINE)
9            sem_wait(&empty);           // Line P1
10           put(i);                     // Line P2
11           sem_post(&full);            // Line P3
12           sem_post(&mutex);           // Line P4 (NEW LINE)
13       }
14   }
15
16   void *consumer(void *arg) {
17       int i;
18       for (i = 0; i < loops; i++) {
19           sem_wait(&mutex);           // Line C0 (NEW LINE)
20           sem_wait(&full);            // Line C1
21           int tmp = get();            // Line C2
22           sem_post(&empty);           // Line C3
23           sem_post(&mutex);           // Line C4 (NEW LINE)
24           printf("%d\n", tmp);
25       }
26   }
27
28   int main(int argc, char *argv[]) {
29       // ...
30       sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31       sem_init(&full, 0, 0);    // ... and 0 are full
32       sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33       // ...
34   }
```

Figure 31.11: **Adding Mutual Exclusion (Incorrectly)**

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.

At Last, A Working Solution

To solve this problem, we simply must reduce the scope of the lock. Figure 31.12 shows the correct 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 multi-threaded 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.

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&empty);         // Line P1
9           sem_wait(&mutex);         // Line P1.5 (MOVED MUTEX HERE...)
10          put(i);                   // Line P2
11          sem_post(&mutex);         // Line P2.5 (... AND HERE)
12          sem_post(&full);          // Line P3
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&full);          // Line C1
20          sem_wait(&mutex);         // Line C1.5 (MOVED MUTEX HERE...)
21          int tmp = get();          // Line C2
22          sem_post(&mutex);         // Line C2.5 (... AND HERE)
23          sem_post(&empty);         // Line C3
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33      // ...
34  }
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

31.5 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 31.13.

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 ac-

```
1  typedef struct _rwlock_t {
2    sem_t lock;      // binary semaphore (basic lock)
3    sem_t writelock; // used to allow ONE writer or MANY readers
4    int   readers;   // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8    rw->readers = 0;
9    sem_init(&rw->lock, 0, 1);
10   sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14   sem_wait(&rw->lock);
15   rw->readers++;
16   if (rw->readers == 1)
17     sem_wait(&rw->writelock); // first reader acquires writelock
18   sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22   sem_wait(&rw->lock);
23   rw->readers--;
24   if (rw->readers == 0)
25     sem_post(&rw->writelock); // last reader releases writelock
26   sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30   sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34   sem_post(&rw->writelock);
35 }
```

Figure 31.13: **A Simple Reader-Writer Lock**

quire 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 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 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, espe-

> 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**.

cially when it comes to fairness. In particular, it would be relatively easy
for readers to starve writers. More sophisticated solutions to this prob-
lem 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 primi-
tives [CB08]. Either way, they showcase once again how we can use
semaphores in an interesting and useful way.

31.6 The Dining Philosophers

One of the most famous concurrency problems posed, and solved, by
Dijkstra, is known as the **dining philosopher's problem** [D71]. The prob-
lem is famous because it is fun and somewhat intellectually interesting;
however, its practical utility is low. However, its fame forces its inclu-
sion 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 31.14): as-
sume there are five "philosophers" sitting around a table. Between each
pair of philosophers is a single fork (and thus, five total). The philoso-
phers each have times where they think, and don't need any forks, and
times where they eat. In order to eat, a philosopher needs 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
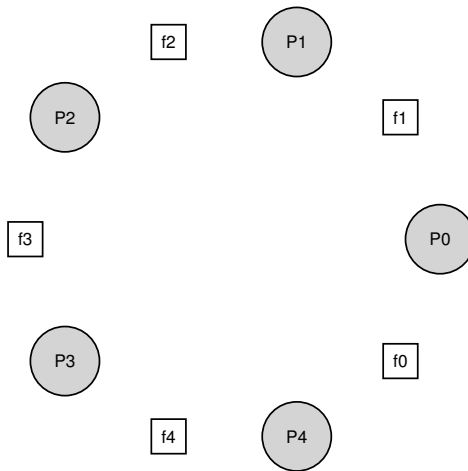a problem we study in concurrent programming.

Figure 31.14: **The Dining Philosophers**

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; }
```

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]`.

```
1   void getforks() {
2     sem_wait(forks[left(p)]);
3     sem_wait(forks[right(p)]);
4   }
5
6   void putforks() {
7     sem_post(forks[left(p)]);
8     sem_post(forks[right(p)]);
9   }
```

Figure 31.15: **The `getforks()` And `putforks()` Routines**

Broken Solution

We attempt our first solution to the problem. Assume we initialize each semaphore (in the `forks` array) to a value of 1. Assume also that each philosopher knows its own number (`p`). We can thus write the `getforks()` and `putforks()` routine as shown in Figure 31.15.

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:

```
1   void getforks() {
2     if (p == 4) {
3       sem_wait(forks[right(p)]);
4       sem_wait(forks[left(p)]);
5     } else {
6       sem_wait(forks[left(p)]);
7       sem_wait(forks[right(p)]);
8     }
9   }
```

Because the last philosopher tries to grab right before left, there is no situation where each philosopher grabs one fork and is stuck waiting for another; the cycle of waiting is broken. Think through the ramifications of this solution, and convince yourself that it works.

```
1   typedef struct __Zem_t {
2       int value;
3       pthread_cond_t cond;
4       pthread_mutex_t lock;
5   } Zem_t;
6
7   // only one thread can call this
8   void Zem_init(Zem_t *s, int value) {
9       s->value = value;
10      Cond_init(&s->cond);
11      Mutex_init(&s->lock);
12  }
13
14  void Zem_wait(Zem_t *s) {
15      Mutex_lock(&s->lock);
16      while (s->value <= 0)
17          Cond_wait(&s->cond, &s->lock);
18      s->value--;
19      Mutex_unlock(&s->lock);
20  }
21
22  void Zem_post(Zem_t *s) {
23      Mutex_lock(&s->lock);
24      s->value++;
25      Cond_signal(&s->cond);
26      Mutex_unlock(&s->lock);
27  }
```

Figure 31.16: **Implementing Zemaphores With Locks And CVs**

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; some of them have fascinating names. Look them up if you are interested in learning more, or just getting more practice thinking in a concurrent manner [D08].

31.7 How To Implement Semaphores

Finally, let's use our low-level synchronization primitives, locks and condition variables, to build our own version of semaphores called ... *(drum roll here)* ... **Zemaphores**. This task is fairly straightforward, as you can see in Figure 31.16.

As you can see from the figure, we use just one lock and one condition variable, plus a state variable to track the value of the semaphore. Study the code for yourself until you really understand it. Do it!

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; indeed, the value will never be lower than zero. This behavior is easier to implement and matches the current Linux implementation.

TIP: BE CAREFUL WITH GENERALIZATION

The abstract technique of generalization can thus be quite useful in systems design, where one good idea can be made slightly broader and thus solve a larger class of problems. However, be careful when generalizing; as Lampson warns us "Don't generalize; generalizations are generally wrong" [L83].

One could view semaphores as a generalization of locks and condition variables; however, is such a generalization needed? And, given the difficulty of realizing a condition variable on top of a semaphore, perhaps this generalization is not as general as you might think.

Curiously, building condition variables out of 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 building condition variables out of semaphores is more challenging than it might appear.

31.8 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" by 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" by Bryan Cantrill, 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" by 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" by 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" by E.W. Dijkstra. CACM, volume 11(3), March 1968. `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" by E.W. Dijkstra. CACM, volume 11(5), 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" by E.W. Dijkstra. Information Processing Letters 1, 1972. `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" by A.B. Downey. Available at the following site: `http://greenteapress.com/semaphores/`. *A nice (and free!) book about semaphores. Lots of fun problems to solve, if you like that sort of thing.*

[D71] "Hierarchical ordering of sequential processes" by E.W. Dijkstra. Available online here: `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" by Jim Gray, 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." Oh, snap!*

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

[L83] "Hints for Computer Systems Design" by Butler Lampson. ACM Operating Systems Review, 15:5, October 1983. *Lampson, a famous systems researcher, loved using hints in the design of computer systems. A hint is something that is often correct but can be wrong; in this use, a signal() is telling a waiting thread that it changed the condition that the waiter was waiting on, but not to trust that the condition will be in the desired state when the waiting thread wakes up. In this paper about hints for designing systems, one of Lampson's general hints is that you should use hints. It is not as confusing as it sounds.*

Homework (Code)

In this homework, we'll use semaphores to solve some well-known concurrency problems. Many of these are taken from Downey's excellent "Little Book of Semaphores"[2], which does a good job of pulling together a number of classic problems as well as introducing a few new variants; interested readers should check out the Little Book for more fun.

Each of the following questions provides a code skeleton; your job is to fill in the code to make it work given semaphores. On Linux, you will be using native semaphores; on a Mac (where there is no semaphore support), you'll have to first build an implementation (using locks and condition variables, as described in the chapter). Good luck!

Questions

1. The first problem is just to implement and test a solution to the **fork/join problem**, as described in the text. Even though this solution is described in the text, the act of typing it in on your own is worthwhile; even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one. See `fork-join.c` for details. Add the call `sleep(1)` to the child to ensure it is working.

2. Let's now generalize this a bit by investigating the **rendezvous problem**. The problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task, and see `rendezvous.c` for details.

3. Now go one step further by implementing a general solution to **barrier synchronization**. Assume there are two points in a sequential piece of code, called P_1 and P_2. Putting a **barrier** between P_1 and P_2 guarantees that all threads will execute P_1 before any one thread executes P_2. Your task: write the code to implement a `barrier()` function that can be used in this manner. It is safe to assume you know N (the total number of threads in the running program) and that all N threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See `barrier.c` for details.

4. Now let's solve the **reader-writer problem**, also as described in the text. In this first take, don't worry about starvation. See the code in `reader-writer.c` for details. Add `sleep()` calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?

5. Let's look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See `reader-writer-nostarve.c` for details.

6. Use semaphores to build a **no-starve mutex**, in which any thread that tries to acquire the mutex will eventually obtain it. See the code in `mutex-nostarve.c` for more information.

7. Liked these problems? See Downey's free text for more just like them. And don't forget, have fun! But, you always do when you write code, no?

---

[2]Available: `http://greenteapress.com/semaphores/downey08semaphores.pdf`.