Chapter 11  Semaphores in C

Semaphores are a good way to learn about synchronization, but they are not as widely used, in practice, as mutexes and condition variables.

Nevertheless, there are some syncrhonization problems that can be solved simply with semaphores, yielding solutions that are more demonstrably correct.

This chapter presents a C API for working with semaphores and my code for making it easier to work with. And it presents a final challenge: can you write an implementation of a semaphore using mutexes and condition variables?

The code for this chapter is in directory semaphore in the repository for this book (see Section 0.1).

11.1  POSIX Semaphores

A semaphore is a data structure used to help threads work together without interfering with each other.

The POSIX standard specifies an interface for semaphores; it is not part of Pthreads, but most UNIXes that implement Pthreads also provide semaphores.

POSIX semaphores have type sem\_t. This type behaves like a structure, so if you assign it to a variable you are making a copy of the contents. Copying a semaphore is almost certainly a bad idea; in POSIX, the behavior of the copy is undefined.

Fortunately, it is easy to put a wrapper around sem\_t to make it safer and easier to use. My wrapper API is in sem.h:

typedef sem\_t Semaphore;

Semaphore \*make\_semaphore(int value);

void semaphore\_wait(Semaphore \*sem);

void semaphore\_signal(Semaphore \*sem);

Semaphore is a synonym for sem\_t, but I find it more readable, and the capital letter reminds me to treat it like an object and pass it by pointer.

The implementation of these functions is in sem.c:

Semaphore \*make\_semaphore(int value)

{

Semaphore \*sem = check\_malloc(sizeof(Semaphore));

int n = sem\_init(sem, 0, value);

if (n != 0) perror\_exit("sem\_init failed");

return sem;

}

make\_semaphore takes the initial value of the semaphore as a parameter. It allocates space for a Semaphore, initializes it, and returns a pointer to Semaphore.

sem\_init returns 0 if it succeeds and -1 if anything goes wrong. One nice thing about using wrapper functions is that you can encapsulate the error-checking code, which makes the code that uses these functions more readable.

Here is the implementation of semaphore\_wait:

void semaphore\_wait(Semaphore \*sem)

{

int n = sem\_wait(sem);

if (n != 0) perror\_exit("sem\_wait failed");

}

And here is semaphore\_signal:

void semaphore\_signal(Semaphore \*sem)

{

int n = sem\_post(sem);

if (n != 0) perror\_exit("sem\_post failed");

}

I prefer to call this operation “signal” rather than “post”, although both terms are common.

Here’s an example that shows how to use a semaphore as a mutex:

Semaphore \*mutex = make\_semaphore(1);

semaphore\_wait(mutex);

// protected code goes here

semaphore\_signal(mutex);

When you use a semaphore as a mutex, you usually initialize it to 1 to indicate that the mutex is unlocked; that is, one thread can pass the semaphore without blocking.

Here I am using the variable name mutex to indicate that the semaphore is being used as a mutex. But remember that the behavior of a semaphore is not the same as a Pthread mutex.

11.2  Producers and consumers with semaphores

Using these semaphore wrapper functions, we can write a solution to the Producer-Consumer problem from Section 10.2. The code in this section is in queue\_sem.c.

Here’s the new definition of Queue, replacing the mutex and condition variables with semaphores:

typedef struct {

int \*array;

int length;

int next\_in;

int next\_out;

Semaphore \*mutex; //-- new

Semaphore \*items; //-- new

Semaphore \*spaces; //-- new

} Queue;

And here’s the new version of make\_queue:

Queue \*make\_queue(int length)

{

Queue \*queue = (Queue \*) malloc(sizeof(Queue));

queue->length = length;

queue->array = (int \*) malloc(length \* sizeof(int));

queue->next\_in = 0;

queue->next\_out = 0;

queue->mutex = make\_semaphore(1);

queue->items = make\_semaphore(0);

queue->spaces = make\_semaphore(length-1);

return queue;

}

mutex is used to guarantee exclusive access to the queue; the initial value is 1, so the mutex is initially unlocked.

items is the number of items in the queue, which is also the number of consumer threads that can execute queue\_pop without blocking. Initially there are no items in the queue.

spaces is the number of empty spaces in the queue, which is the number of producer threads that can execute queue\_push without blocking. Initially the number of spaces is the capacity of the queue, which is length-1, as explained in Section 10.1.

Here is the new version of queue\_push, which is run by producer threads:

void queue\_push(Queue \*queue, int item) {

semaphore\_wait(queue->spaces);

semaphore\_wait(queue->mutex);

queue->array[queue->next\_in] = item;

queue->next\_in = queue\_incr(queue, queue->next\_in);

semaphore\_signal(queue->mutex);

semaphore\_signal(queue->items);

}

Notice that queue\_push doesn’t have to call queue\_full any more; instead, the semaphore keeps track of how many spaces are available and blocks producers if the queue is full.

Here is the new version of queue\_pop:

int queue\_pop(Queue \*queue) {

semaphore\_wait(queue->items);

semaphore\_wait(queue->mutex);

int item = queue->array[queue->next\_out];

queue->next\_out = queue\_incr(queue, queue->next\_out);

semaphore\_signal(queue->mutex);

semaphore\_signal(queue->spaces);

return item;

}

This solution is explained, using pseudo-code, in Chapter 4 of The Little Book of Semaphores.

Using the code in the repo for this book, you should be able to compile and run this solution. You should be able to run:

$ make queue\_sem

$ ./queue\_sem

11.3  Make your own semaphores

Any problem that can be solved with semaphores can also be solved with condition variables and mutexes. One way to prove that’s true is to implement a semaphore using condition variables and mutexes.

Before you go on, you might want to try this as an exercise: write functions that implement the semaphore API in sem.h using using condition variables and mutexes. You can put your solution in mysem.c and mysem.h in the repo for this book, and you’ll find my solution in mysem\_soln.c and mysem\_soln.h.

If you have trouble getting started, you can use the following structure definition, from my solution, as a hint:

typedef struct {

int value, wakeups;

Mutex \*mutex;

Cond \*cond;

} Semaphore;

value is the value of the semaphore. wakeups counts the number of pending signals; that is, the number of threads that have been woken but have not yet resumed execution. The reason for wakeups is to make sure that our semaphores have Property 3, described in The Little Book of Semaphores.

mutex provides exclusive access to value and wakeups; cond is the condition variable threads wait on if they wait on the semaphore.

Here is the initialization code for this structure:

Semaphore \*make\_semaphore(int value)

{

Semaphore \*semaphore = check\_malloc(sizeof(Semaphore));

semaphore->value = value;

semaphore->wakeups = 0;

semaphore->mutex = make\_mutex();

semaphore->cond = make\_cond();

return semaphore;

}

11.3.1  Semaphore implementation

Here is my implementation of semaphores using POSIX mutexes and condition variables:

void semaphore\_wait(Semaphore \*semaphore)

{

mutex\_lock(semaphore->mutex);

semaphore->value--;

if (semaphore->value < 0) {

do {

cond\_wait(semaphore->cond, semaphore->mutex);

} while (semaphore->wakeups < 1);

semaphore->wakeups--;

}

mutex\_unlock(semaphore->mutex);

}

When a thread waits on the semaphore, it has to lock the mutex before it decrements value. If the value of the semaphore becomes negative, the thread blocks until a “wakeup” is available. While it is blocked, the mutex is unlocked, so another thread can signal.

Here is the code for semaphore\_signal:

void semaphore\_signal(Semaphore \*semaphore)

{

mutex\_lock(semaphore->mutex);

semaphore->value++;

if (semaphore->value <= 0) {

semaphore->wakeups++;

cond\_signal(semaphore->cond);

}

mutex\_unlock(semaphore->mutex);

}

Again, a thread has to lock the mutex before it increments value. If the semaphore was negative, that means threads are waiting, so the signalling thread increments wakeups and signals the condition variable.

At this point one of the waiting threads might wake up, but the mutex is still locked until the signalling thread unlocks it.

At that point, one of the waiting threads returns from cond\_wait and checks whether a wakeup is still available. If not, it loops and waits on the condition variable again. If so, it decrements wakeups, unlocks the mutex, and exits.

One thing about this solution that might not be obvious is the use of a do...while loop. Can you figure out why it is not a more conventional while loop? What would go wrong?

The problem is that with a while loop this implementation would not have Property 3. It would be possible for a thread to signal and then run around and catch its own signal.

With the do...while loop, it is guaranteed1 that when a thread signals, one of the waiting threads will get the signal, even if the signalling thread runs around and gets the mutex before one of the waiting threads resumes.