Semaphores

Glenn Bruns
CSUMB

Most of this material on these slides comes directly from *The Little Book of Semaphores*, by Allen B. Downey.

Lecture Objectives

After this lecture, you should be able to:

- explain how semaphores work
- write simple code using semaphores
- list the pros and cons of semaphores vs. other synchronization variables

Should we care about semaphores?

- In our OSTEP text, the authors discuss locks, condition variables, and semaphores
- Semaphore is a "super" synchronization variable: it replaces locks and cond. variables
- On the other hand, Anderson & Dahlin say it's better to not use semaphores
- The pthreads library supports semaphores
- They are used a lot in OS kernels

Semaphore coverage in our class

- We'll use the "Little Book of Semaphores" by Allen Downey
- We'll solve a bunch of small programming puzzles
- We won't write pthreads code that uses semaphores

The Little Book of Semaphores is free!

http://greenteapress.com/semaphores/

Synchronization

- Synchronization is a time relationship between events
- Example synchronization constraints:
  - serialization: one event happens before another
  - mutual exclusion: events can’t happen at the same time

Execution paths

thread A

```
a1   x = 5
a2   print x
```

thread B

```
b1   x = 7
```

The value that gets printed depends on the "execution path".

We can write an execution path like this:

A	B
x=5	
	x=7
print x	

or like this: a1 < b1 < a2

- What path gives output 5 and final value 5?
- What path gives output 7 and final value 7?
- Is there a path that gives output 7 and final value 5?

Semaphores (from Downey)

A semaphore is like a variable, but with three differences:

1. when you create a semaphore, you can initialize it to any value, but afterward there are only two operations you can perform
   - increment
   - decrement
2. when a thread decrements the semaphore, if the result is negative, the thread blocks itself and can't continue until another thread increments the semaphore
3. when a thread increments the semaphore, if there are threads waiting, one of them gets unblocked

Semaphore syntax

```
sem = Semaphore(1)
sem.signal()   // i.e. increment
sem.wait()     // i.e. decrement
```

Points to remember:

- before a wait, we don't know if a thread will block
- after a signal, both threads will run concurrently – no way to know which will run right away
- when a signal occurs, no threads might be waiting

Example: the “signal” pattern

Assume semaphore ‘sem’ has initial value 0.

thread A

```
1   statement a1
2   sem.signal()
```

thread B

```
1   sem.wait()
2   statement b2
```

What happens?

All possible execution paths:

- a1 < a2 < b1 < b2
- a1 < b1 < a2 < b2
- b1 < a1 < a2 < b2

Does a1 always happen before b1?

Puzzle: how to “rendezvous”?

thread A

```
1   statement a1
2   statement a2
```

thread B

```
1   statement b1
2   statement b2
```

Exercise: add semaphore(s) and wait/signal calls to guarantee a1 happens before b2 and b1 happens before a2.

Be sure to specify the names and initial values of your semaphore(s).

Syntax reminder:

```
sem = Semaphore(n)      // create
sem.wait()              // dec
sem.signal()            // inc
```

Hint and Solution

hint:

```
sem1 = Semaphore(0)
sem2 = Semaphore(0)
```

a solution:

thread A

```
1  a1
2  sem2.signal()
3  sem1.wait()
4  a2
```

thread B

```
1  b1
2  sem1.signal()
3  sem2.wait()
4  b2
```

alternative:

thread A

```
1  a1
2  sem2.signal()
3  sem1.wait()
4  a2
```

thread B

```
1  sem2.wait()
2  b1
3  b2
4  sem1.signal()
```

Two execution paths

initialization

```
sem1 = Semaphore(0)
sem2 = Semaphore(0)
```

thread A

```
a1
sem1.signal()
sem2.wait()
a2
```

thread B

```
b1
sem2.signal()
sem1.wait()
b2
```

A	B
a1	
sem1.signal	
	b1
sem2.wait	
	sem2.signal
	sem1.wait
a2	
	b2

A	B
	b1
a1	
sem1.signal	
	sem2.signal
	sem1.wait
	b2
sem2.wait	
a2	

Puzzle: how to do mutual exclusion?

thread A

```
count = count + 1
```

thread B

```
count = count + 1
```

Exercise: write code that will ensure mutually exclusive access to shared variable count

Hint and solution

hint: `sem = Semaphore(1)`

thread A

```
sem.wait()
   // critical section
   count = count + 1
sem.signal()
```

thread B

```
sem.wait()
   // critical section
   count = count + 1
sem.signal()
```

A “symmetric” solution

Puzzle: multiplex

thread A1

```
count = count + 1
```

thread A2

```
count = count + 1
```

...

Exercise: write code that will ensure at most n threads can be in the critical section at the same time.

Hint and solution

hint: `sem = Semaphore(n)`

```
sem.wait()
   // critical section
   count = count + 1
sem.signal()
```

(every thread looks like this)

You can think of the semaphore as a set of tokens.

When a thread calls wait, it picks up one of the tokens.

When a thread calls signal, it releases one.

Puzzle: rendezvous with n threads?

thread

```
1   rendezvous
2   critical point
```

Exercise: write code that will guarantee that no thread executes critical point until all threads have executed rendezvous.

This is called a **barrier**.

Assume each thread can access a constant ***n***, which is the number of threads.

Hint: you can introduce variables, and use statements besides sem and wait

Hint 1

hint:

```
1   n = the number of threads
2   count = 0
3   mutex = Semaphore(1)
4   barrier = Semaphore(0)
```

count – how many threads are at the barrier

mutex – provides mutually exclusive access to count

barrier – locked until all threads arrive

thread

```
1   rendezvous
2   critical point
```

Hint 2

hint:

```
1    rendezvous
2
3    mutex.wait()
4       count = count + 1
5    mutex.signal()
6
7    if count == n: barrier.signal()
8
9    barrier.wait()
10
11   critical point
```

this *almost* works

what is wrong?

how to fix it?

Solution

hint:

```
1    rendezvous
2
3    mutex.wait()
4       count = count + 1
5    mutex.signal()
6
7    if count == n: barrier.signal()
8
9    barrier.wait()
10   barrier.signal()
11
12   critical point
```

As each thread passes through the barrier, it signals that another thread can pass.

A wait followed by a signal is called a turnstile.

Arguments against semaphore

The argument of Anderson and Dahlin:

Semaphores tend to be used for two different things: locking, and waiting

1. If locks and CVs are used, it's clearer that you're locking, or waiting.
2. With a lock and CV, it's easy to wait on shared state, and the condition you're waiting on is clear.

Alternative definition of decrement

If the value of the semaphore is > 0, it is decremented and the operation returns right away.

If the value of the semaphore is 0, then the operation blocks until it becomes possible to perform the decrement.

This is the original definition (by Dijkstra).

The Downey book and our textbook use the other definition. The Wiki page uses yet another.

Summary

- A semaphore has operations wait() and signal().
- Semaphores can do the work of locks and condition variables combined.
- They are supported in pthreads, but some argue against coding with them