Little Book of Semaphores, Chapter 3

Geoffrey Matthews
Western Washington University

January 20, 2016

Signaling

Initialization

```
sem = Semaphore(0)
```

Thread A

```
a1
sem.signal()
```

Thread B

```
sem.wait()
b1
```

Rendezvous

Thread A
```
a1
a2
```

Thread B
```
b1
b2
```

- A has to wait for b1 to finish before a2
- B has to wait for a1 to finish before b2
- Ideas?

Rendezvous

Thread A

```
a1
a2
```

Thread B

```
b1
b2
```

- A has to wait for b1 to finish before a2
- B has to wait for a1 to finish before b2
- Ideas?
- Hint: use `aArrived` and `bArrived`

Rendezvous

Initialization

```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```

Thread A

```
a1
aArrived.signal()
bArrived.wait()
a2
```

Thread B

```
b1
bArrived.signal()
aArrived.wait()
b2
```

Rendezvous 2

Thread A
```
a1
bArrived.wait()
aArrived.signal()
a2
```

Thread B
```
b1
bArrived.signal()
aArrived.wait()
b2
```

- Will it still work?

Rendezvous 2

Thread A

```
a1
bArrived.wait()
aArrived.signal()
a2
```

Thread B

```
b1
bArrived.signal()
aArrived.wait()
b2
```

- Will it still work?
- Try to limit context switches.
- Do everything you can before waiting.

Rendezvous 3

Thread A

```
a1
bArrived.wait()
aArrived.signal()
a2
```

Thread B

```
b1
aArrived.wait()
bArrived.signal()
b2
```

- Will it still work?

Rendezvous 3

Thread A
```
a1
bArrived.wait()
aArrived.signal()
a2
```

Thread B
```
b1
aArrived.wait()
bArrived.signal()
b2
```

- Will it still work?
- **Deadlock!**

Mutex

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

- Any problems?

Mutex

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

- Any problems?
- Cannot assume operations are atomic.
- Even if we write:

  `count++`

Mutex

Thread A
```
count = count + 1
```

Thread B
```
count = count + 1
```

- Can't let both threads operate at the same time.
- Don't care which goes first.
- Ideas?

Mutex

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

- Can't let both threads operate at the same time.
- Don't care which goes first.
- Ideas?
- Hint: create a semaphore `mutex` initialized to 1.
- The rock in the box!

Mutex

Initialization

```
mutex = Semaphore(1)
```

Thread A

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

Mutex

Initialization

```
mutex = Semaphore(1)
```

Thread A

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

- A **symmetric** solution.
- Symmetric solutions are easy to generalize.

Mutex

Initialization

```
mutex = Semaphore(1)
```

Thread A

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

- A **symmetric** solution.
- Symmetric solutions are easy to generalize.
- Metaphorically we can look at this as a **token**:
  the rock in the box, the talking stick

Mutex

Initialization

```
mutex = Semaphore(1)
```

Thread A

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

- A **symmetric** solution.
- Symmetric solutions are easy to generalize.
- Metaphorically we can look at this as a **token**:
  the rock in the box, the talking stick
- Another metaphor is a **lock**
- Sometimes called "getting" and "releasing" a lock.

Multiplex

- Generalize the mutex so that at most `n` threads can access the critical section at a time.
- Ideas?

Multiplex

- Generalize the mutex so that at most `n` threads can access the critical section at a time.
- Ideas?
- Initialize the mutex to `n`.

Initialization

```
mutex = Semaphore(n)
```

Thread i

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

Thread j

```
mutex.wait()
  # critical section
  count = count + 1
mutex.signal()
```

Barrier

Initialization

```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```

Thread A

```
a1
aArrived.signal()
bArrived.wait()
a2
```

Thread B

```
b1
bArrived.signal()
aArrived.wait()
b2
```

- Recall the rendezvous, above.
- Is there a way to generalize this to `n` threads?

Barrier

Thread i

```
rendezvous
critical point
```

Thread j

```
rendezvous
critical point
```

- We want all tasks to finish `rendezvous` before beginning `critical point`
- When the first $n - 1$ threads arrive, they should block until the nth thread arrives, when all should proceed.
- Ideas?

Barrier Hint

Initialization

```
n = number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

- `count` keeps track of how many threads have arrived
- `mutex` provides atomic increment of `count`
- `barrier` is locked until all threads arrive

Barrier non-solution

Thread i

```
#rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == n:    barrier.signal()

barrier.wait()

#critical point
```

- What's wrong?

Barrier non-solution

```
Thread i
#rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == n:    barrier.signal()

barrier.wait()

#critical point
```

- What's wrong?
- Deadlock!

Barrier non-solution

Thread i

```
#rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == n:    barrier.signal()

barrier.wait()

#critical point
```

- What's wrong?
- Deadlock!
- Does it *always* deadlock?

Barrier working solution

Thread i

```
#rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == n:  barrier.signal()

barrier.wait()
barrier.signal()

#critical point
```

- `wait` then `signal` is called a **turnstile**
- It blocks all threads until one thread signals, then lets all through.
- An “open” turnstyle can be *locked* by calling `wait` and *unlocked* by calling `signal`.

Barrier working solution

Thread i

```
#rendezvous
mutex.wait()
  count = count + 1
mutex.signal()

if count == n:  barrier.signal()

barrier.wait()
barrier.signal()

#critical point
```

- After the *n*th thread, what state is the turnstile in?
- Is the barrier reusable? Can we put this code in a loop?

Another barrier non-solution

Thread i

```
#rendezvous

mutex.wait()
  count = count + 1
  if count == n:  barrier.signal()

  barrier.wait()
  barrier.signal()
mutex.signal()

#critical point
```

- Deadlock again. Why?

Another barrier non-solution

Thread i

```
#rendezvous

mutex.wait()
  count = count + 1
  if count == n:  barrier.signal()

  barrier.wait()
  barrier.signal()
mutex.signal()

#critical point
```

- Deadlock again. Why?
- Common source of deadlocks: blocking on a semaphore while holding a mutex.

Reusable barrier non-solution #1

Thread i

```
#rendezvous
mutex.wait();  count += 1;  mutex.signal()
if count == n: turnstile.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();  count -= 1;  mutex.signal()
if count == 0: turnstile.wait()
```

- What's wrong?

Reusable barrier non-solution #1

Thread i

```
#rendezvous
mutex.wait();  count += 1;  mutex.signal()
if count == n: turnstile.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();  count -= 1;  mutex.signal()
if count == 0: turnstile.wait()
```

- What's wrong?
- If we interrupt a process *just before* evaluating the first conditional?

Reusable barrier non-solution #1

Thread i

```
#rendezvous
mutex.wait();  count += 1;  mutex.signal()
if count == n: turnstile.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();  count -= 1;  mutex.signal()
if count == 0: turnstile.wait()
```

- What's wrong?
- If we interrupt a process *just before* evaluating the first conditional?
- It is possible that *all* the threads will see `count == n` and signal the turnstile.

Reusable barrier non-solution #1

Thread i

```
#rendezvous
mutex.wait();  count += 1;  mutex.signal()
if count == n: turnstile.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();  count -= 1;  mutex.signal()
if count == 0: turnstile.wait()
```

- What's wrong?
- If we interrupt a process *just before* evaluating the first conditional?
- It is possible that *all* the threads will see `count == n` and signal the turnstile.
- If the second conditional is interrupted?

Reusable barrier non-solution #1

Thread i

```
#rendezvous
mutex.wait();  count += 1;  mutex.signal()
if count == n: turnstile.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();  count -= 1;  mutex.signal()
if count == 0: turnstile.wait()
```

- What's wrong?
- If we interrupt a process *just before* evaluating the first conditional?
- It is possible that *all* the threads will see `count == n` and signal the turnstile.
- If the second conditional is interrupted?
- Deadlock!

Reusable barrier non-solution #2

Thread i

```
#rendezvous
mutex.wait();
  count += 1;
  if count == n: turnstile.signal()
mutex.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();
  count -= 1;
  if count == 0: turnstile.wait()
mutex.signal()
```

- Still doesn't work. Why?

Reusable barrier non-solution #2

Thread i

```
#rendezvous
mutex.wait();
  count += 1;
  if count == n: turnstile.signal()
mutex.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();
  count -= 1;
  if count == 0: turnstile.wait()
mutex.signal()
```

- Still doesn't work. Why?
- Hint: this is meant to be in a loop.

Reusable barrier non-solution #2

Thread i

```
#rendezvous
mutex.wait();
  count += 1;
  if count == n: turnstile.signal()
mutex.signal()
turnstile.wait()
turnstile.signal()
#critical point
mutex.wait();
  count -= 1;
  if count == 0: turnstile.wait()
mutex.signal()
```

- Still doesn't work. Why?
- Hint: this is meant to be in a loop.
- One thread could go around and through the turnstile again while the others sit there, getting one lap ahead.
- What to do?

Reusable barrier solution

Thread i

```
#rendezvous
mutex.wait();
  count += 1;
  if count == n:
    turnstile2.wait()         # lock the second
    turnstile.signal()        # unlock the first
mutex.signal()
turnstile.wait()              # first turnstile
turnstile.signal()
#critical point
mutex.wait();
  count -= 1;
  if count == 0:
    turnstile.wait()          # lock the first
    turnstile2.signal()       # unlock the second
mutex.signal()
turnstile2.wait()             # second turnstile
turnstile2.signal()
```

Reusable barrier solution

- Called a **two-phase barrier**
- Forces all threads to wait twice: once for all to arrive, and again for all threads to execute the critical section.

Reusable barrier solution

- Called a **two-phase barrier**
- Forces all threads to wait twice: once for all to arrive, and again for all threads to execute the critical section.
- Typical of semaphores—complex and difficult to understand.
- Can we prove it correct?

Reusable barrier solution

- Called a **two-phase barrier**
- Forces all threads to wait twice: once for all to arrive, and again for all threads to execute the critical section.
- Typical of semaphores—complex and difficult to understand.
- Can we prove it correct?
- Only the *n*th thread can unlock turnstiles.
- Before a thread can unlock the first turnstile, it has to close the second, and vice versa. It is therefore impossible for one thread to get ahead of the others by more than one turnstile.

Preloaded turnstile

```
# rendezvous
mutex.wait()
  count  += 1
  if count == n:
    turnstile2.wait()     # lock the second
    turnstile.signal(n)   # unlock the first
mutex.signal()
turnstile.wait()          # first turnstile
# critical point
mutex.wait()
  count -= 1
  if count == 0:
    turnstile.wait()      # lock the first
    turnstile2.signal(n)  # unlock the second
mutex.signal()
turnstile2.wait()         # second turnstile
```

- Signals n at a time
- Could be done in a loop?

Barrier Object

```
class Barrier:
  def __init__(self, n):
    self.n = n
    self.count = 0
    self.mutex = Semaphore(1)
    self.turnstile = Semaphore(0)
    self.turnstile2 = Semaphore(0)
  def phase1(self):
    self.mutex.wait()
      self.count += 1
      if self.count == self.n:
        self.turnstile2.wait()
        self.turnstile.signal(self.n)
    self.mutex.signal()
    self.turnstile.wait()
  def phase2(self):
    self.mutex.wait()
      self.count -= 1
      if self.count == 0:
        self.turnstile.wait()
        self.turnstile2.signal(self.n)
    self.mutex.signal()
    self.turnstile2.wait()
```

```
barrier = Barrier(n)
...
# rendezvous
barrier.phase1()
# critical point
barrier.phase2()
```

```
barrier = Barrier(n)
...
# rendezvous
barrier.phase1()
barrier.phase2()
# critical point
```

Multiple barriers

```
class Barrier:
  def __init__(self, n):
    self.n = n
    self.count = 0
    self.mutex = Semaphore(1)
    self.turnstile = Semaphore(0)
    self.turnstile2 = Semaphore(0)
  def phase1(self):
    self.mutex.wait()
      self.count += 1
      if self.count == self.n:
        self.turnstile2.wait()
        self.turnstile.signal(self.n)
    self.mutex.signal()
    self.turnstile.wait()
  def phase2(self):
    self.mutex.wait()
      self.count -= 1
      if self.count == 0:
        self.turnstile.wait()
        self.turnstile2.signal(self.n)
    self.mutex.signal()
    self.turnstile2.wait()
```

```
barrier = Barrier(n)
...
loop:
    # section A
  barrier.phase1()
    # section B
  barrier.phase2()
    # section C
  barrier.phase1()
    # section D
  barrier.phase2()
```

Multiple barriers

```
class Barrier:
  def __init__(self, n):
    self.n = n
    self.count = 0
    self.mutex = Semaphore(1)
    self.turnstile = Semaphore(0)
    self.turnstile2 = Semaphore(0)
  def phase1(self):
    self.mutex.wait()
      self.count += 1
      if self.count == self.n:
        self.turnstile2.wait()
        self.turnstile.signal(self.n)
    self.mutex.signal()
    self.turnstile.wait()
  def phase2(self):
    self.mutex.wait()
      self.count -= 1
      if self.count == 0:
        self.turnstile.wait()
        self.turnstile2.signal(self.n)
    self.mutex.signal()
    self.turnstile2.wait()
```

```
barrier = Barrier(n)
...
loop:
    # section A
  barrier.phase1()
    # section B
  barrier.phase2()
    # section C
  barrier.phase1()
    # section D
  barrier.phase2()
```

- What if we have an odd number of sections?

Queue

- Ballroom dancing.
- If a leader arrives, it checks for a follower, if none available waits, otherwise proceeds.
- If a follower arrives, it checks for a leader, if none available waits, otherwise proceeds.
- Ideas?

Queue

- Ballroom dancing.
- If a leader arrives, it checks for a follower, if none available waits, otherwise proceeds.
- If a follower arrives, it checks for a leader, if none available waits, otherwise proceeds.
- Ideas?
- Hint:

```
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)
```

Queue solution

Leader

```
followerQueue.signal()
leaderQueue.wait()
dance()
```

Follower

```
leaderQueue.signal()
followerQueue.wait()
dance()
```

Queue solution

Leader

```
followerQueue.signal()
leaderQueue.wait()
dance()
```

Follower

```
leaderQueue.signal()
followerQueue.wait()
dance()
```

- Do the leaders and followers proceed together?

Queue solution

Leader
```
followerQueue.signal()
leaderQueue.wait()
dance()
```

Follower
```
leaderQueue.signal()
followerQueue.wait()
dance()
```

- Do the leaders and followers proceed together?
- It is possible for 100 leaders to dance before any followers do.

Queue solution

Leader

```
followerQueue.signal()
leaderQueue.wait()
dance()
```

Follower

```
leaderQueue.signal()
followerQueue.wait()
dance()
```

- Do the leaders and followers proceed together?
- It is possible for 100 leaders to dance before any followers do.
- Add constraint that only one leader and one follower can dance concurrently.
- Ideas?

Exclusive Queue Hint

```
leaders = followers = 0
mutex = Semaphore(1)
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)
rendezvous = Semaphore(0)
```

Exclusive Queue Solution

Leader

```
mutex.wait()
if followers > 0:
  followers--
  followerQueue.signal()
else:
  leaders++
  mutex.signal()
  leaderQueue.wait()
dance()
rendezvous.wait()
mutex.signal()
```

Follower

```
mutex.wait()
if leaders > 0:
  leaders--
  leaderQueue.signal()
else:
  followers++
  mutex.signal()
  followerQueue.wait()
dance()
rendezvous.signal()
```

- “Wait” means “wait on this queue”
- “Signal” means “let someone go from this queue”

FIFO Queue

Initialization

```
local mySem = Semaphore(0)
```

```
class Fifo:
  def __init__(self):
    self.queue = Queue()
    self.mutex = Semaphore(1)

  def wait(self, mySem):
    self.mutex.wait()
    self.queue.add(mySem)
    self.mutex.signal()
    mySem.wait()

  def signal(self):
    self.mutex.wait()
    sem = self.queue.remove()
    self.mutex.signal()
    sem.signal()
```