Little Book of Semaphores, Chapter 4

Geoffrey Matthews
Western Washington University

January 11, 2016

Producers and Consumers

Producer i

```
event[i] = waitForEvent()
buffer.add(event[i])
```

Consumer j

```
event[j] = buffer.get()
event[j].process()
```

- Threads must have exclusive access to the buffer. No adding and getting at the same time.
- If a consumer thread arrives when the buffer is empty, it blocks until a producer adds an item.
- `waitForEvent` and `process` can happen simultaneously, but not buffer access.
- `event` is a local variable for each thread—not shared.
- We could use an array, as above, with all producers and consumers having different indices.

Producers and Consumers Hint

```
mutex = Semaphore(1)
items = Semaphore(0)
local event
```

- Local events can be handled several ways:
  - Each thread has its own run-time stack. (We use this in scheme and python, where threads are functions.)
  - Threads could be objects, with local private variables.
  - Threads can use unique IDs as indices into an array.

Producer-consumer solution

Producer

```
event = waitForEvent()
mutex.wait()
  buffer.add(event)
  items.signal()
mutex.signal()
```

Consumer

```
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
event.process()
```

Producer-consumer solution

Producer

```
event = waitForEvent()
mutex.wait()
  buffer.add(event)
  items.signal()
mutex.signal()
```

Consumer

```
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
event.process()
```

- Could the `items.signal()` be taken out of the mutex?
- What would be the advantage?

Producer-consumer solution (slight improvement)

Producer

```
event = waitForEvent()
mutex.wait()
  buffer.add(event)
mutex.signal()
items.signal()
```

Consumer

```
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
event.process()
```

Producer-consumer solution (slight improvement)

Producer

```
event = waitForEvent()
mutex.wait()
  buffer.add(event)
mutex.signal()
items.signal()
```

Consumer

```
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
event.process()
```

- `items` could at times not accurately reflect the actual number of waiting consumers.

Producer-consumer solution (broken)

Producer

```
event = waitForEvent()
mutex.wait()
  buffer.add(event)
mutex.signal()
items.signal()
```

Consumer

```
mutex.wait()
  items.wait()
  event = buffer.get()
mutex.signal()
event.process()
```

- Why is this broken?

Producer-consumer solution (broken)

Producer

```
event = waitForEvent()
mutex.wait()
  buffer.add(event)
mutex.signal()
items.signal()
```

Consumer

```
mutex.wait()
  items.wait()
  event = buffer.get()
mutex.signal()
event.process()
```

- Why is this broken?
- Don't wait for a semaphore after grabbing a mutex!

Producer-consumer with finite buffer

Broken finite buffer solution

```
if items >= bufferSize:
  block()
```

- `items` is a semaphore, we can't check its size
- Even if we could, we could be interrupted between checking and blocking.
- We *don't* want to block inside a mutex!

Producer-consumer with finite buffer

Broken finite buffer solution

```
if items >= bufferSize:
  block()
```

- `items` is a semaphore, we can't check its size
- Even if we could, we could be interrupted between checking and blocking.
- We *don't* want to block inside a mutex!
- What to do?

Finite buffer producer-consumer hint

```
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(bufferSize)
```

- items + spaces = bufferSize

Finite buffer producer-consumer solution

```
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(bufferSize)
```

Producer

```
event = waitForEvent()

spaces.wait()
mutex.wait()
  buffer.add(event)
mutex.signal()
items.signal()
```

Consumer

```
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
spaces.signal()

event.process()
```

Finite buffer producer-consumer solution

```
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(bufferSize)
```

Producer

```
event = waitForEvent()

spaces.wait()
mutex.wait()
  buffer.add(event)
mutex.signal()
items.signal()
```

Consumer

```
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
spaces.signal()

event.process()
```

- Note that `items + spaces` is a constant.
- This is an *invariant*
- Using invariants is a good way to design programs and help prove properties.

Readers-writers problem

- Suppose a number of processes all access the same data.
- Any number of readers can be in the critical section simultaneously.
- Writers must have exclusive access to the critical section.
- Ideas?

Readers-writers hint

```
readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

- "wait" means "wait for the condition to be true"
- "signal" means "signal that the condition is true"

Readers-writers solution

Writers

```
roomEmpty.wait()
# critical section for writer
roomEmpty.signal()
```

Readers

```
mutex.wait()
  readers += 1
  if readers == 1:
    # first in locks:
    roomEmpty.wait()
mutex.signal()

# critical section for reader

mutex.wait()
  readers -= 1
  if readers == 0:
    # last out unlocks
    roomEmpty.signal()
mutex.signal()
```

- A **Lightswitch**

Claims useful in a proof of correctness

- Only one reader can queue on `roomEmpty`
- Several writers might be queued on `roomEmpty`
- When a reader signals `roomEmpty` the room is empty

A lightswitch object

```
class Lightswitch:
  def __init__(self):
    self.counter = 0
    self.mutex = Semaphore(1)

  def lock(self, semaphore):
    self.mutex.wait()
      self.counter += 1
      if self.counter == 1:
        semaphore.wait()
    self.mutex.signal()

  def unlock(self, semaphore):
    self.mutex.wait()
      self.counter -= 1
      if self.counter == 0:
        semaphore.signal()
        self.mutex.signal()
```

Initialization

```
readswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Readers

```
readswitch.lock(roomEmpty)
# critical section
readwitch.unlock(roomEmpty)
```

Writers

```
roomEmpty.wait()
# critical section for writer
roomEmpty.signal()
```

Starvation

- No deadlock in the above readers-writers solution.
- However, it is possible for a writer to **starve**.
- While a writer is blocked, readers can come and go, and the writer never progresses.
- (In the buffer problem, readers eventually empty the buffer, but we can imagine readers who simply examine the buffer without removing an item.)

Starvation

- No deadlock in the above readers-writers solution.
- However, it is possible for a writer to **starve**.
- While a writer is blocked, readers can come and go, and the writer never progresses.
- (In the buffer problem, readers eventually empty the buffer, but we can imagine readers who simply examine the buffer without removing an item.)
- Puzzle: extend the solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter.

Starvation

- No deadlock in the above readers-writers solution.
- However, it is possible for a writer to **starve**.
- While a writer is blocked, readers can come and go, and the writer never progresses.
- (In the buffer problem, readers eventually empty the buffer, but we can imagine readers who simply examine the buffer without removing an item.)
- Puzzle: extend the solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter.
- Hint: Add a turnstyle and allow the writers to lock it.

No-starve readers-writers hint

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)
turnstile = Semaphore(1)
```

No-starve readers-writers hint

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)
turnstile = Semaphore(1)
```

- `turnstile` is a turnstile for readers and a mutex for writers

No-starve readers-writers solution

Writers

```
turnstile.wait()
  roomEmpty.wait()
  # critical section
turnstile.signal()

roomEmpty.signal()
```

Readers

```
turnstile.wait()
turnstile.signal()

readSwitch.lock(roomEmpty)
  # critical section
readSwitch.unlock(roomEmpty)
```

- `turnstile` is a turnstile for readers and a mutex for writers

No-starve readers-writers solution

Writers

```
turnstile.wait()
  roomEmpty.wait()
  # critical section
turnstile.signal()

roomEmpty.signal()
```

Readers

```
turnstile.wait()
turnstile.signal()

readSwitch.lock(roomEmpty)
  # critical section
readSwitch.unlock(roomEmpty)
```

- `turnstile` is a turnstile for readers and a mutex for writers
- It is now possible for *readers* to starve!

Priority Scheduling

- Some schedulers allow priority scheduling.
- Puzzle: Write a solution to readers-writers that gives priority to writers. In other words, once a writer arrives, no readers are allowed in the critical section until *all* writers have left the system.

Priority Scheduling

- Some schedulers allow priority scheduling.
- Puzzle: Write a solution to readers-writers that gives priority to writers. In other words, once a writer arrives, no readers are allowed in the critical section until *all* writers have left the system.
- Hint: use two lightswitches

Writer-priority readers-writers hint

```
readSwitch = Lightswitch()
writeSwitch = Lightswitch()
mutex = Semaphore(1)
noReaders = Semaphore(1)
noWriters = Semaphore(1)
```

Writer-priority readers-writers solution

Writers

```
writeSwith.lock(noReaders)
  noWriters.wait()

    # critical section

  noWriters.signal()
writeSwitch.unlock(noReaders)
```

Readers

```
noReaders.wait()
  readSwitch.lock(noWriters)
noReaders.signal()

# critical section

readSwitch.unlock(noWriters)
```

- Writers in critical section hold *both* `noReaders` and `noWriters`.
- `writeSwitch` allows writers to queue on `noWriters`, but keeps `noReaders` locked
- The last writer signals `noReaders`

Writer-priority readers-writers solution

Writers

```
writeSwith.lock(noReaders)
  noWriters.wait()

    # critical section

  noWriters.signal()
writeSwitch.unlock(noReaders)
```

Readers

```
noReaders.wait()
  readSwitch.lock(noWriters)
noReaders.signal()

# critical section

readSwitch.unlock(noWriters)
```

- Writers in critical section hold *both* `noReaders` and `noWriters`.
- `writeSwitch` allows writers to queue on `noWriters`, but keeps `noReaders` locked
- The last writer signals `noReaders`
- Readers in critical section hold `noWriters` but don't hold `noReaders`, so a writer can lock `noReaders`
- The last reader signals `noWriters` so writers can go

Thread starvation

- We just addressed **categorical starvation**: one category of threads makes another category starve.
- **Thread starvation** is the more general possibility of a thread waiting indefinitely while other threads proceed.

Thread starvation

- We just addressed **categorical starvation**: one category of threads makes another category starve.
- **Thread starvation** is the more general possibility of a thread waiting indefinitely while other threads proceed.
- Part of the problem is the responsibility of the scheduler. If a thread is never scheduled, it is starved.

Thread starvation

- We just addressed **categorical starvation**: one category of threads makes another category starve.
- **Thread starvation** is the more general possibility of a thread waiting indefinitely while other threads proceed.
- Part of the problem is the responsibility of the scheduler. If a thread is never scheduled, it is starved.
- Some schedulers use algorithms that guarantee bounded waiting.

Thread starvation

- If we don't want to assume too much about the scheduler, can we assume:
- **Property 1:** if there is only one thread that is ready to run, the scheduler has to let it run.
- This would be sufficient for the boundary problem.
- In general we need a stronger assumption.

Thread starvation

- **Property 2:** if a thread is ready to run, then the time it waits until it runs is bounded.
- We use this assumption in all our work.
- Some schedulers in the real world do not guarantee this strictly.
- Property 2 is not strong enough if we use semaphores. Why?

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

- Prevents a thread from signalling a semaphore, racing around a loop and catching its own signal!

Thread i

```
while True:
  mutex.wait()
  # critical section
  mutex.signal()
```

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.
- Prevents a thread from signalling a semaphore, racing around a loop and catching its own signal!

Thread i

```
while True:
  mutex.wait()
  # critical section
  mutex.signal()
```

- However, if A, B, and C are using a mutex in a loop, A and B could race around and around, starving C.

Semaphore starvation

- The weakest assumption about semaphores that makes it possible to avoid starvation is:
- **Property 3:** if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

- Prevents a thread from signalling a semaphore, racing around a loop and catching its own signal!

Thread i

```
while True:
  mutex.wait()
  # critical section
  mutex.signal()
```

- However, if A, B, and C are using a mutex in a loop, A and B could race around and around, starving C.
- A semaphore with Property 3 is called a **weak semaphore**.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.
- A semaphore with Property 4 is called a **strong semaphore**.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.
- A semaphore with Property 4 is called a **strong semaphore**.
- Dijkstra (inventor of semaphores) conjectured in 1965 that it was impossible to solve the mutex problem without starvation with weak semaphores.

Semaphore starvation

- **Property 4:** if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.
- FIFO queues satisfy this property.
- A semaphore with Property 4 is called a **strong semaphore**.
- Dijkstra (inventor of semaphores) conjectured in 1965 that it was impossible to solve the mutex problem without starvation with weak semaphores.
- Morris showed you could do it in 1979.

Morris's algorithm

Initialization

```
room1 = room2 = 0
mutex = Semaphore(1)
t1 = Semaphore(1)
t2 = Semaphore(0)
```

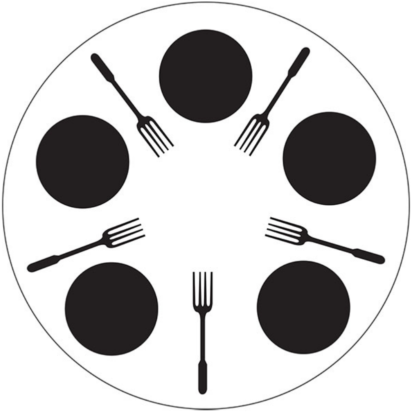
```
mutex.wait()
  room1 += 1
mutex.signal()
t1.wait()
  room2 += 1
  mutex.wait()
  room1 -= 1
  if room1 == 0:
    mutex.signal()
    t2.signal()
t2.wait()
  room2 -= 1
  # critical section
  if room2 == 0:
    t1.signal()
  else:
    t2.signal()
```

The Dining Philosophers

- Five philosophers are eating spaghetti.
- There are five forks.
- Eating spaghetti requires two forks.
- More than one philosopher can eat at a time.

Philosopher i

```
while True:
  think()
  get_forks()
  eat()
  put_forks()
```



The Dining Philosophers

Which fork?

```
def left(i) = return i
def right(i) = return (i+1)%5
```

Initialization

```
forks =
  [Semaphore(1)
    for i in range(5)]
```

Non-solution

```
def get_forks(i):
  fork[right(i)].wait()
  fork[left(i)].wait()

def put_forks(i):
  fork[right(i)].signal()
  fork[left(i)].signal()
```