synchronization

synchronization

- relationships among events—any number of events, and any kind of Relationship
- Serialization: event A must happen before B
- Mutual exclusive: Event A and B must not happen at the same time

Non-determinism

- Thread A
  - A1 print “yes”

- Thread B
  - B1 print “no”

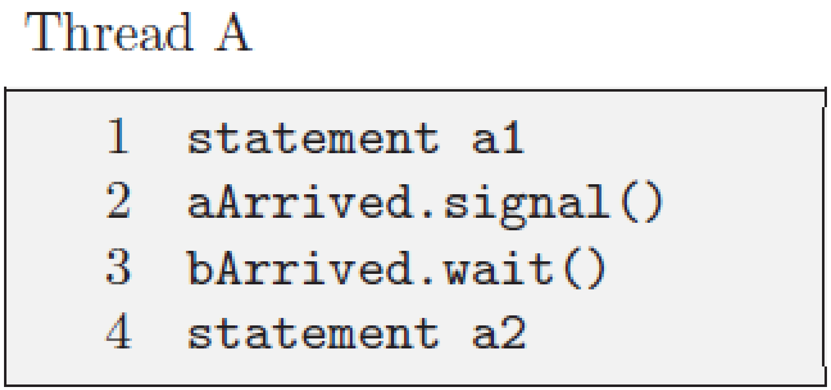
Shared variable

- Concurrent writes
- Concurrent updates

Semaphores

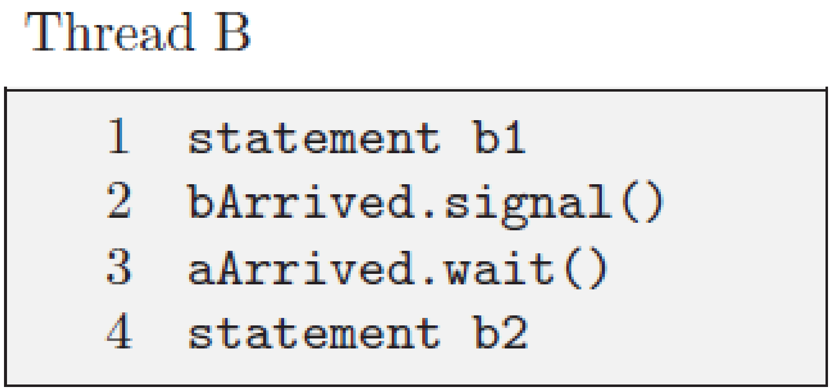
- A semaphore is like an integer, with three differences:
  - After initialize, only can increase or decrease that
  - Negative value → block thread
  - Positive value → unblock thread

Rendezvous solution



Thread A

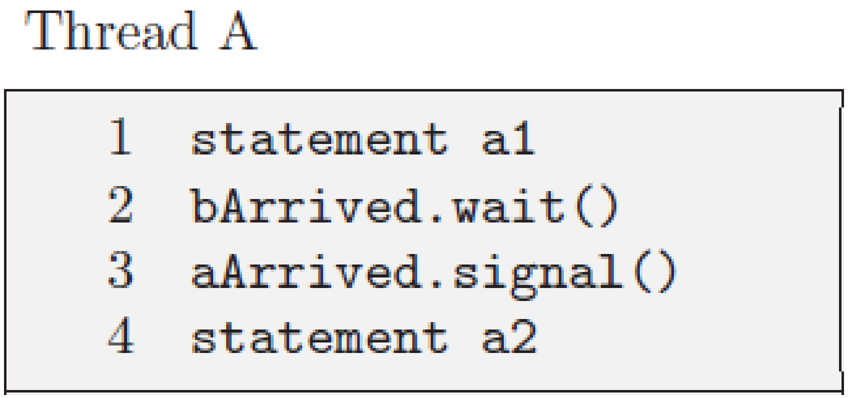
```
statement a1
aArrived.signal()
bArrived.wait()
statement a2
```



Thread B

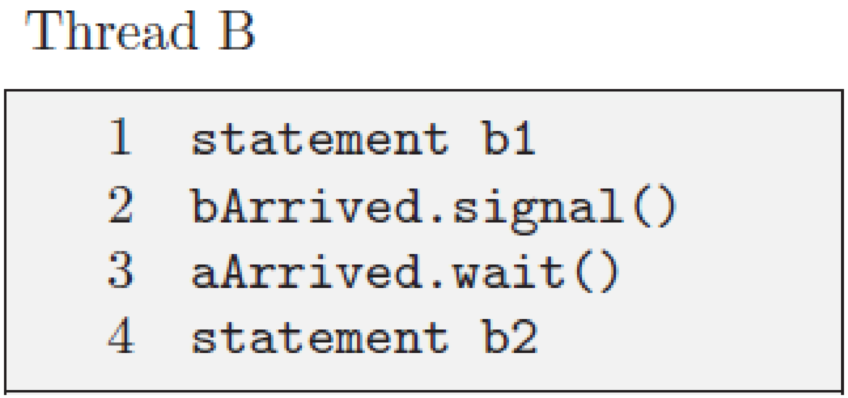
```
statement b1
bArrived.signal()
aArrived.wait()
statement b2
```

Less efficient



Thread A

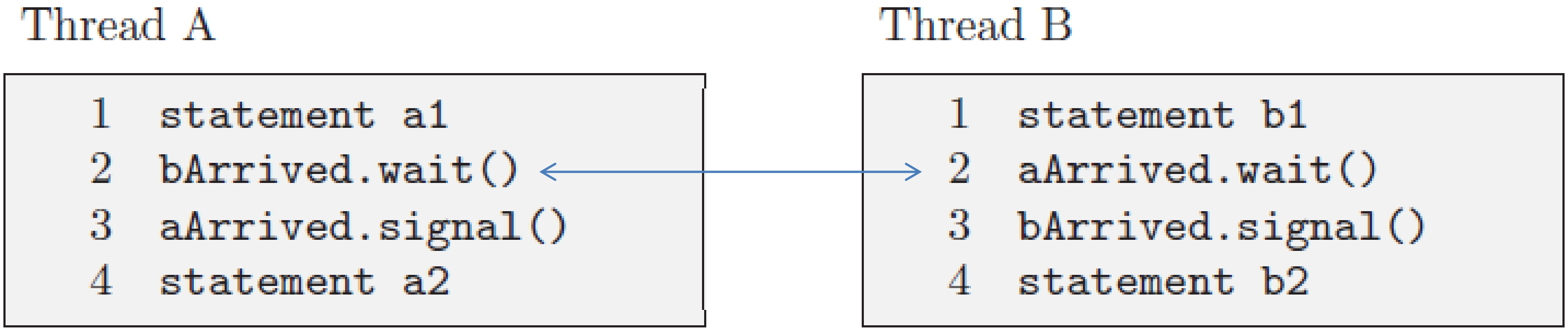
```
statement a1
bArrived.wait()
aArrived.signal()
statement a2
```



Thread B

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

deadlock



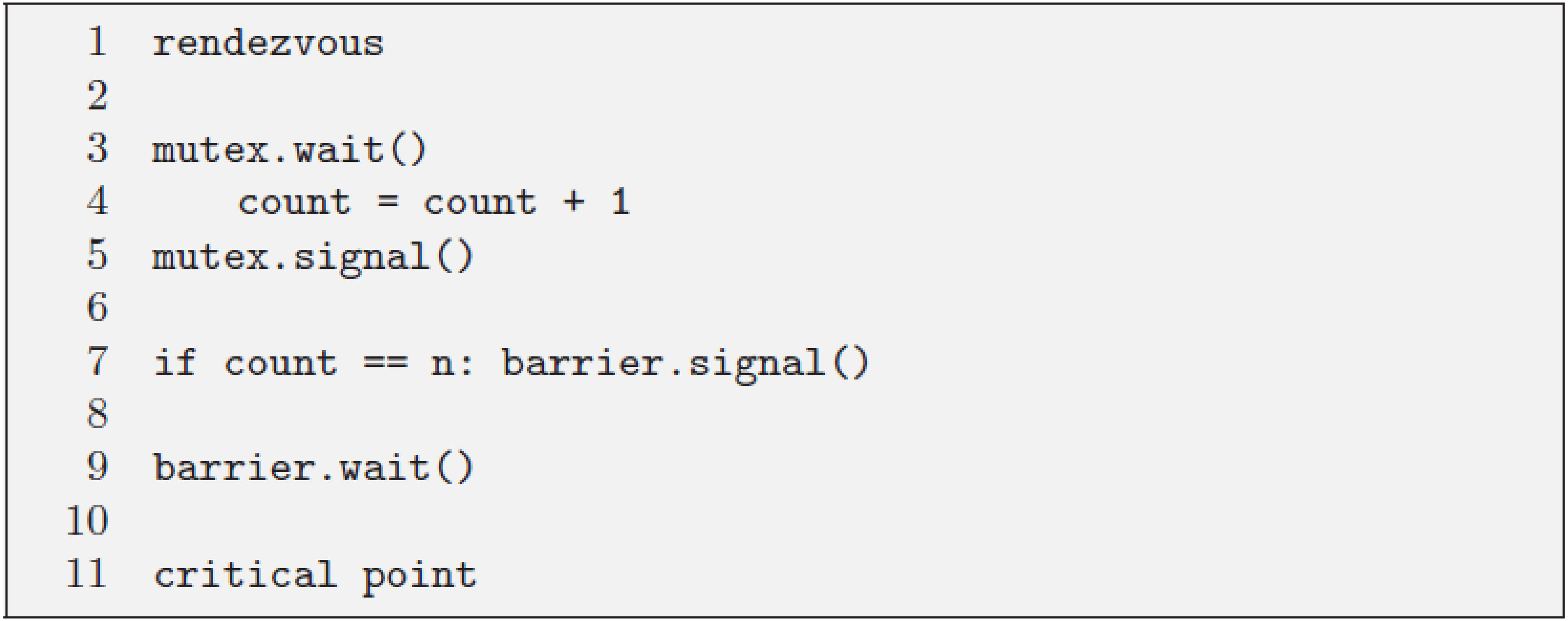
Thread A

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

Thread B

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

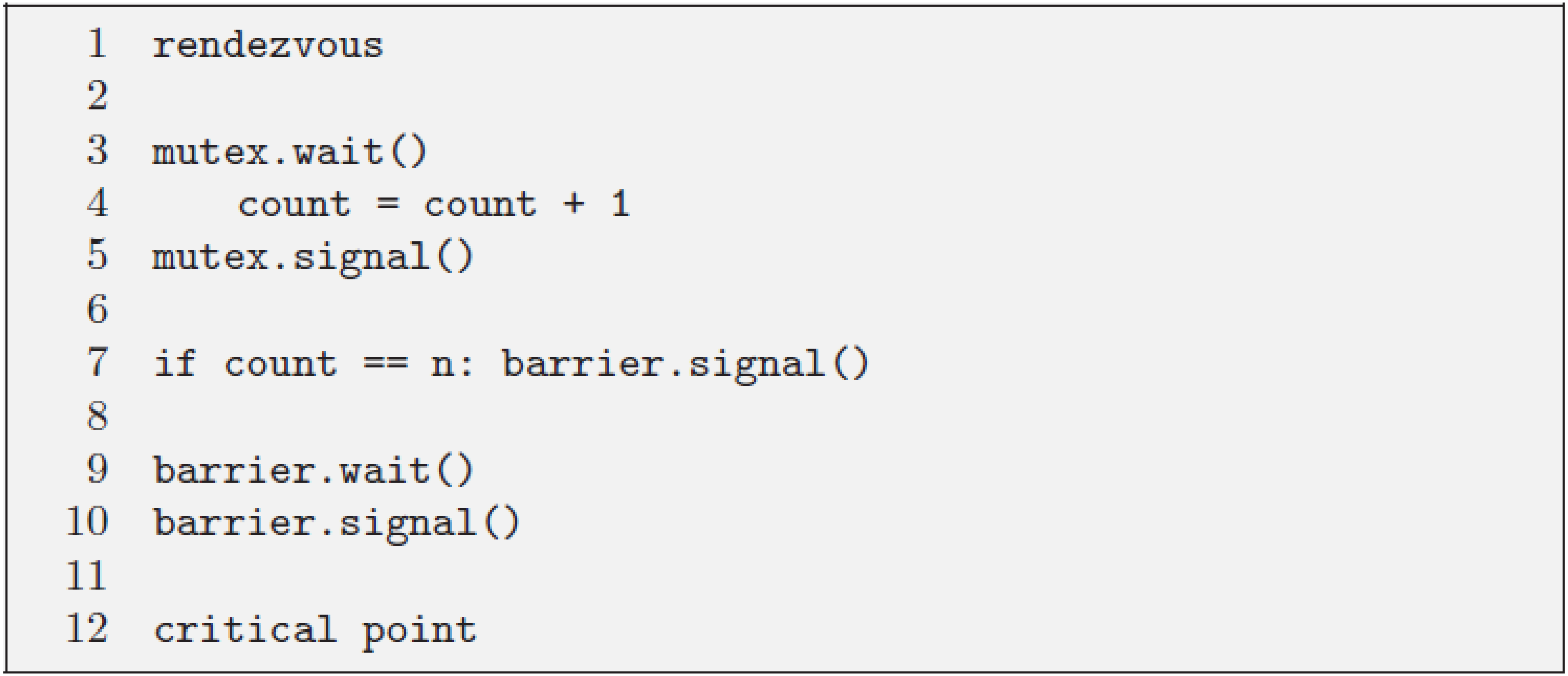
Barrier non solution (37)



```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()

critical point
```

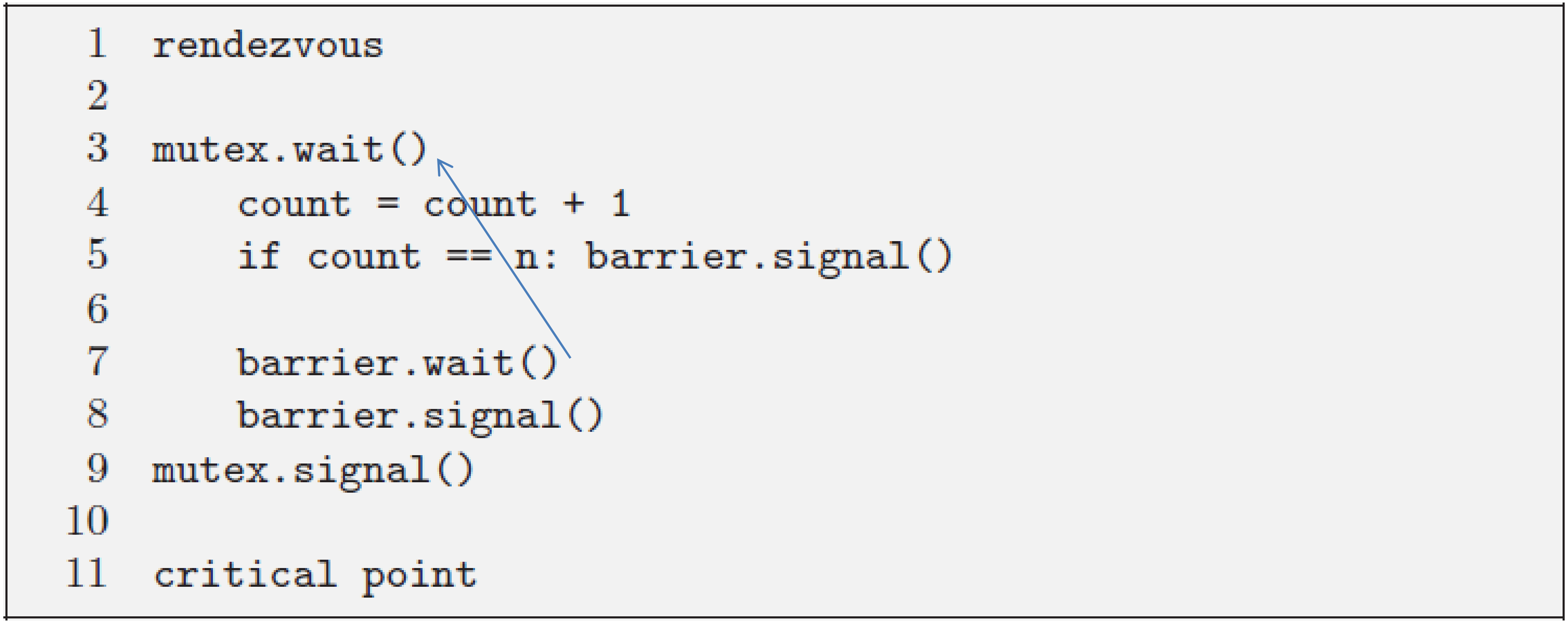
- As an example, imagine that n = 5 and that 4 threads are waiting at the barrier. The value of the semaphore is the number of threads in queue, negated, which is -4.

Barrier solution



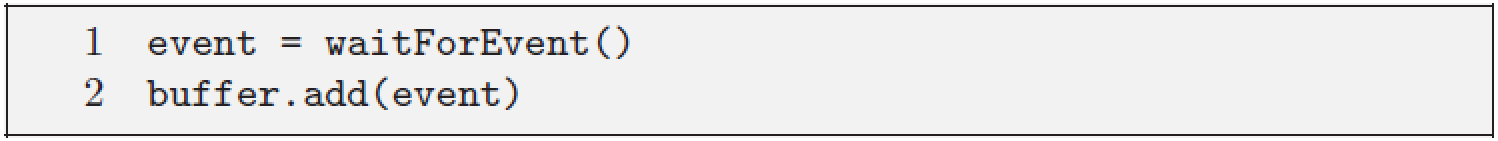
```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

Bad barrier solution (43)

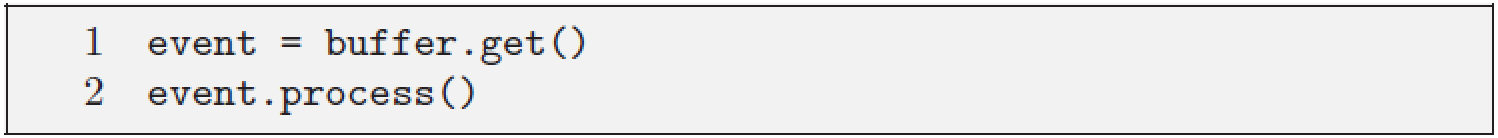


```
rendezvous

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

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

critical point
```

Producer-consumer

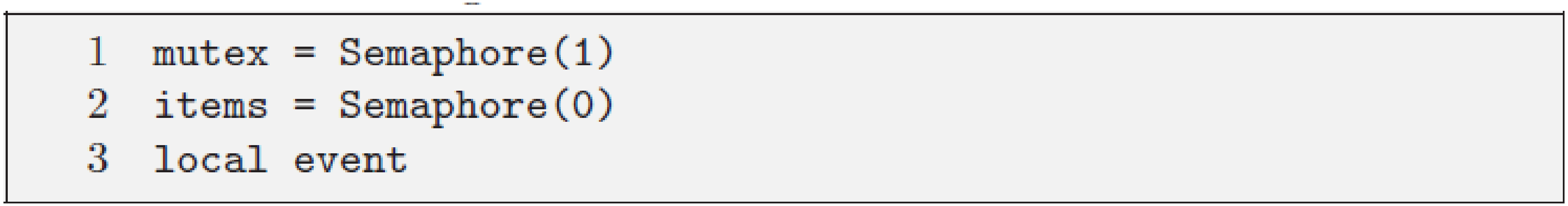


```
event = waitForEvent()
buffer.add(event)
```

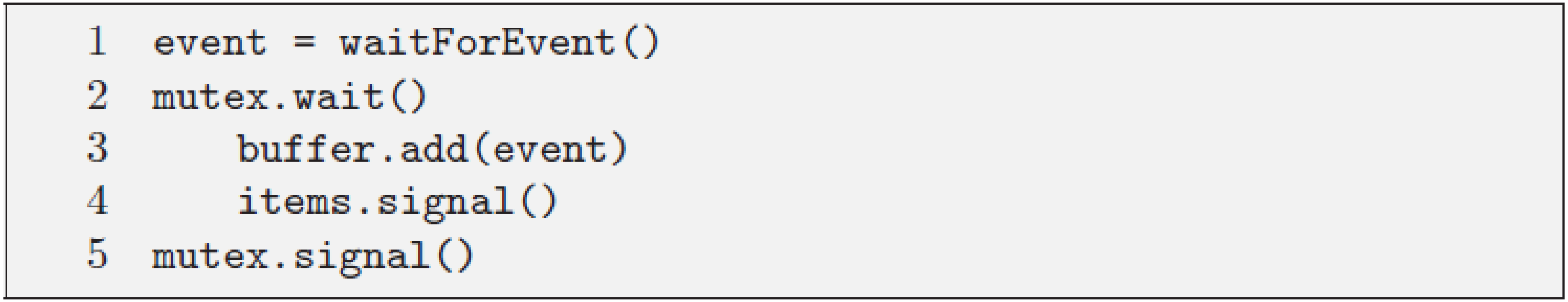


```
event = buffer.get()
event.process()
```

producer

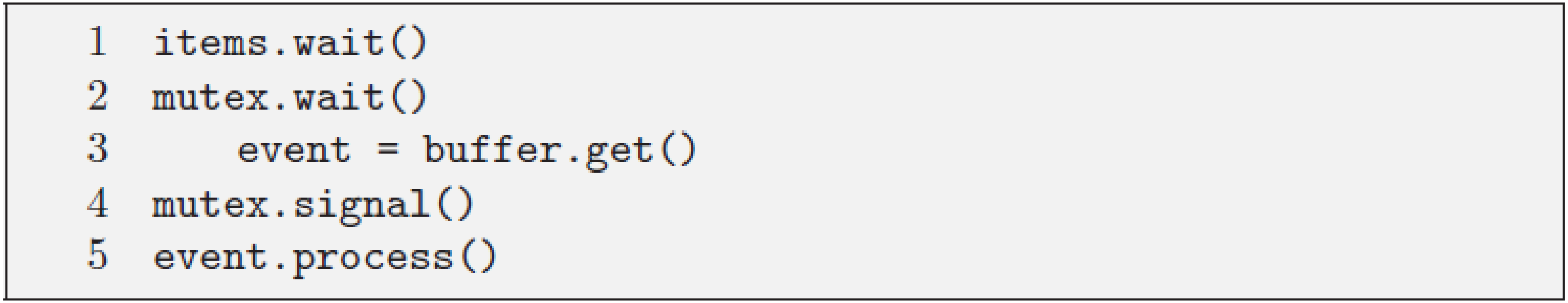


```
mutex = Semaphore(1)
items = Semaphore(0)
local event
```



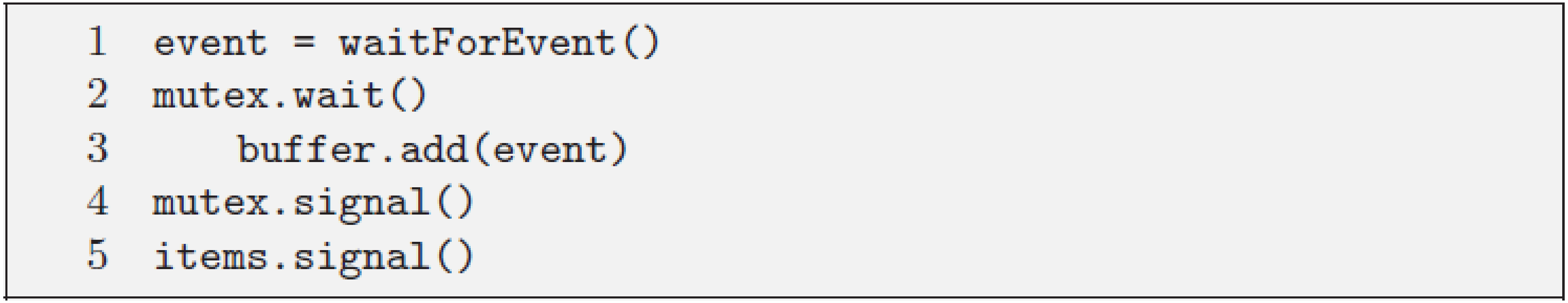
```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
    items.signal()
mutex.signal()
```

Consumer (77)



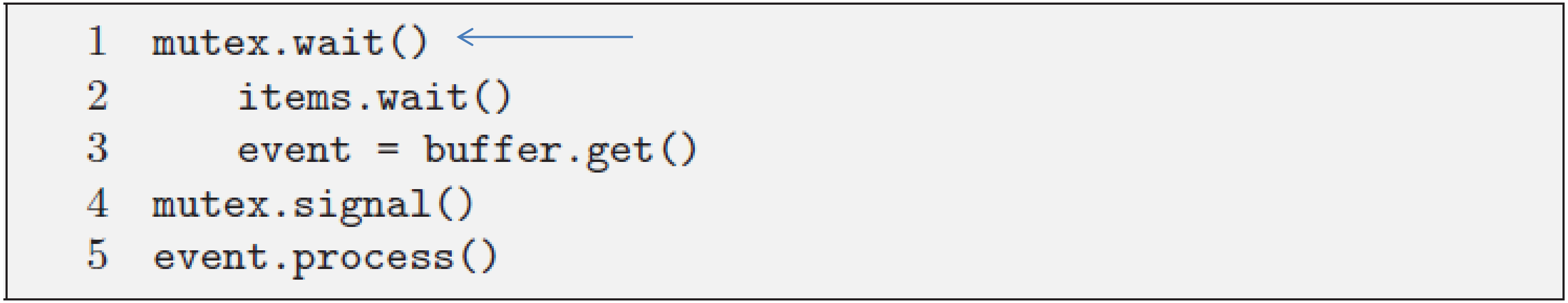
```
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
event.process()
```

Improved producer



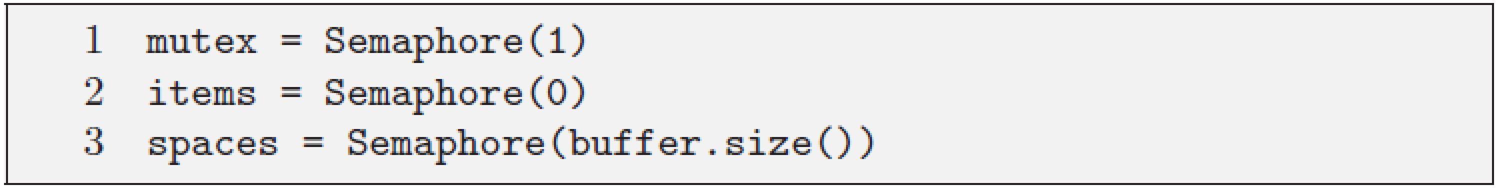
```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```

Bad consumer

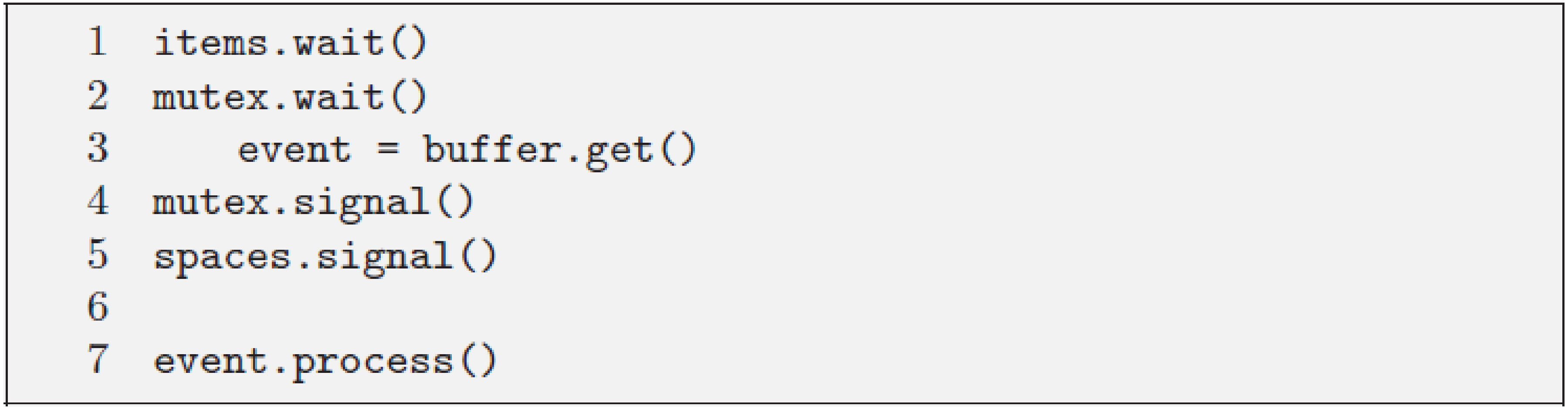


```
mutex.wait()
    items.wait()
    event = buffer.get()
mutex.signal()
event.process()
```

Consumer finite buffer (83)

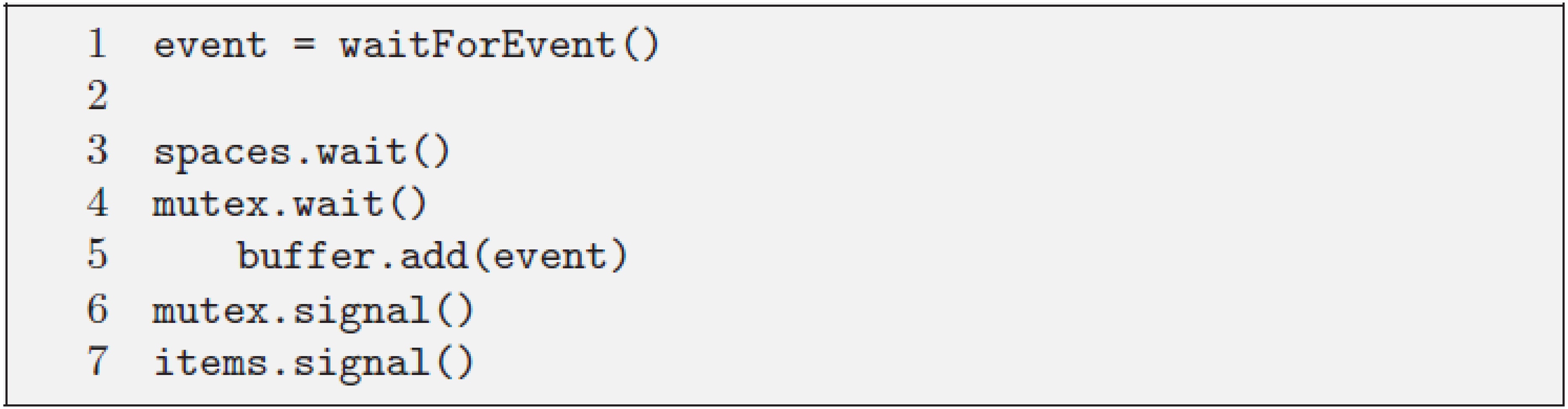


```
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(buffer.size())
```



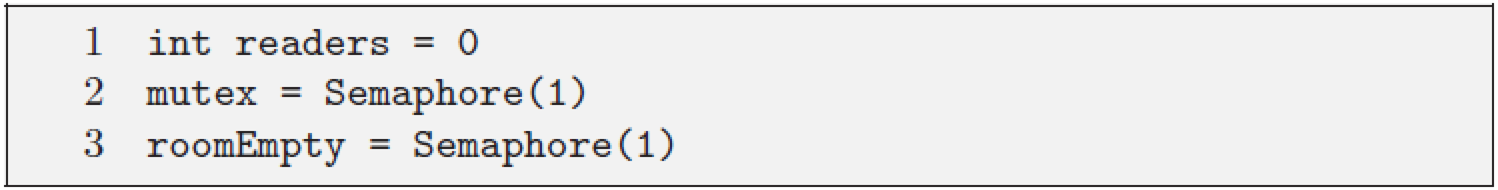
```
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
spaces.signal()

event.process()
```

Producer finite buffer

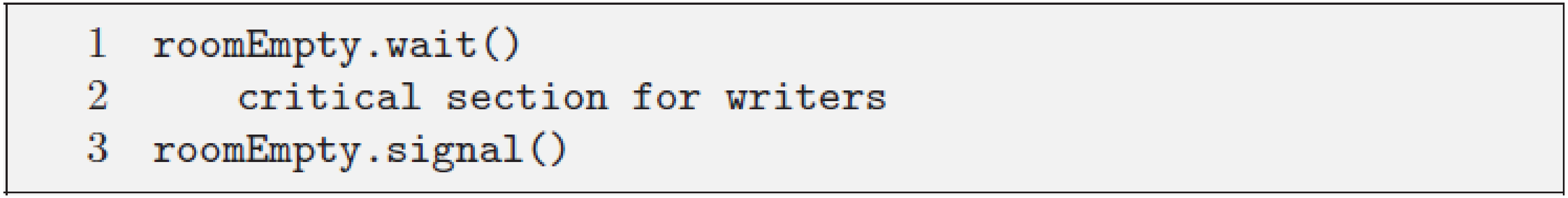


```
event = waitForEvent()

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

Writers (87)

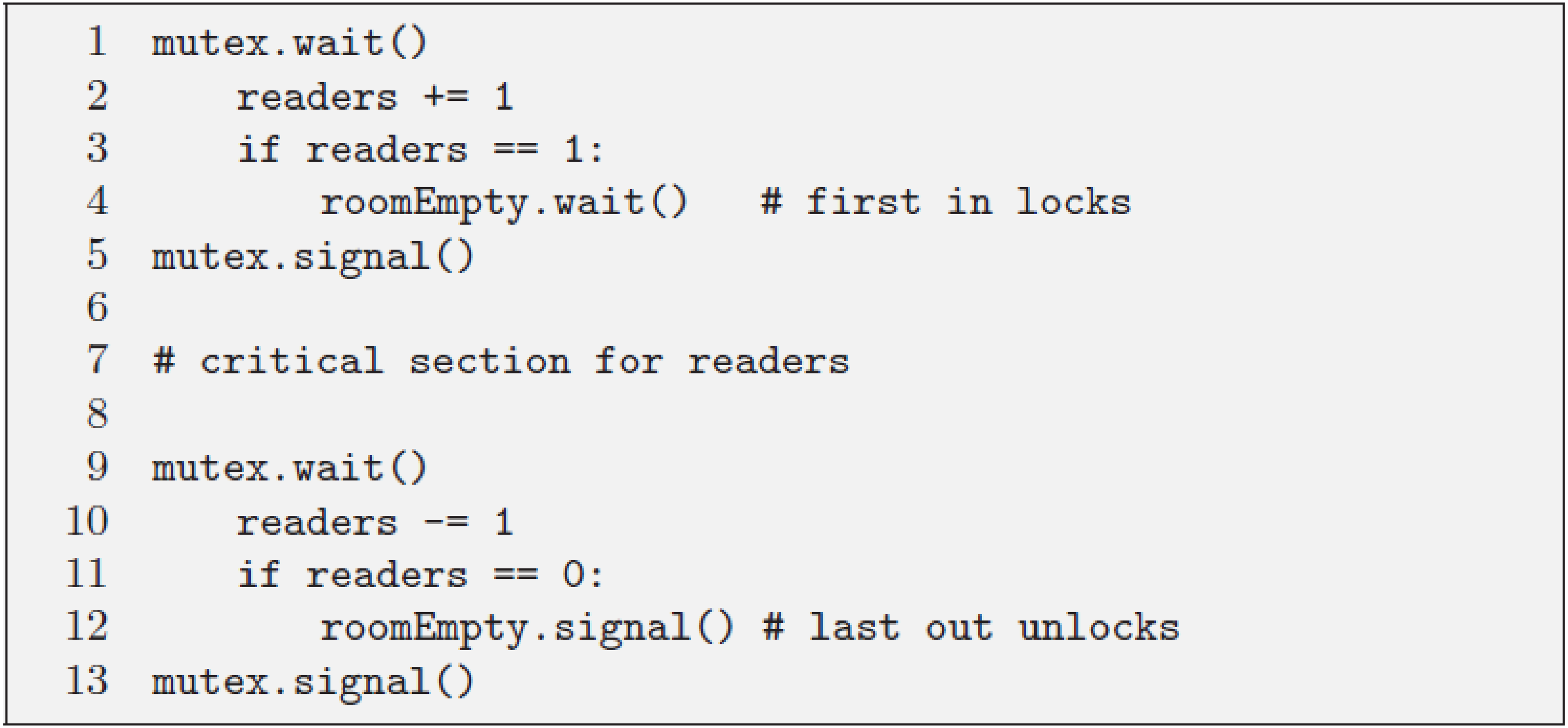


```
int readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```



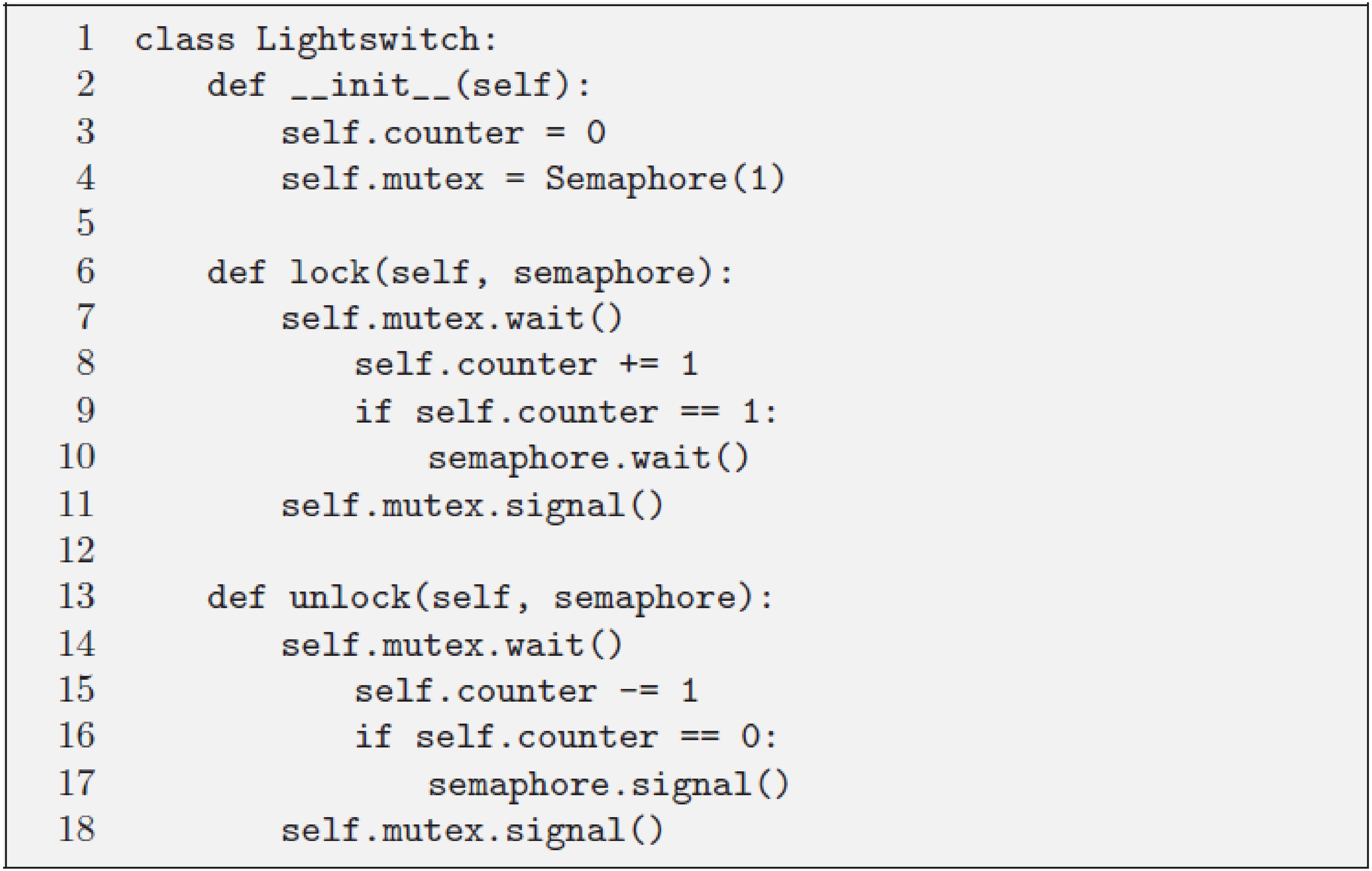
```
roomEmpty.wait()
    critical section for writers
roomEmpty.signal()
```

Readers



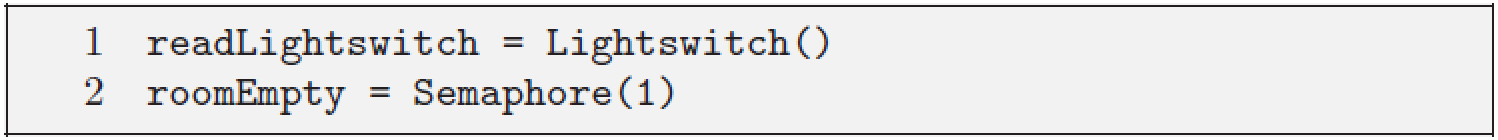
```
mutex.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()   # first in locks
mutex.signal()

# critical section for readers

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

Light switch

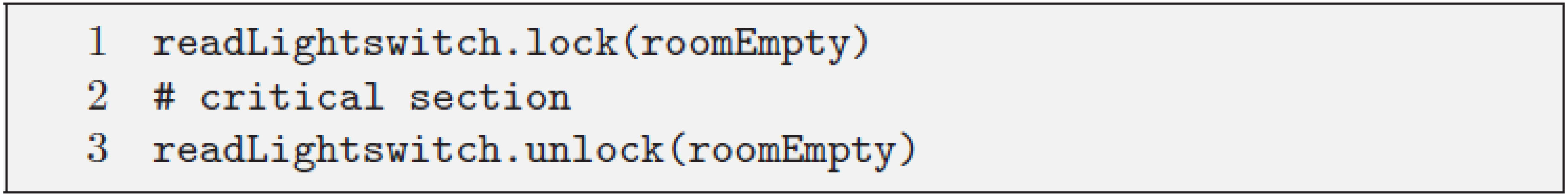


```
class Lightswitch:
    def __init__(self):
        self.counter = 0
        self.mutex = Semaphore(1)

    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()
```

Readers with LS

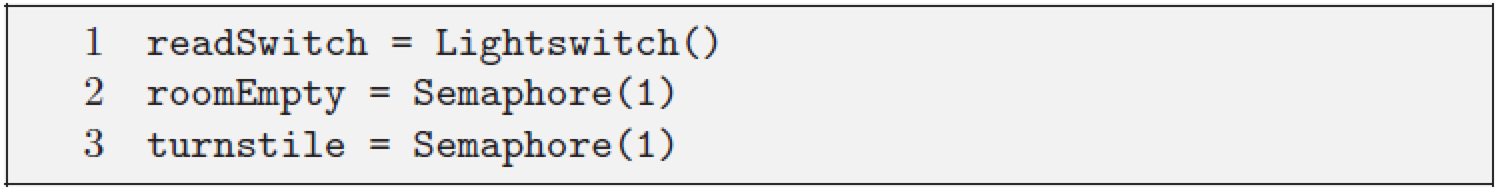


```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

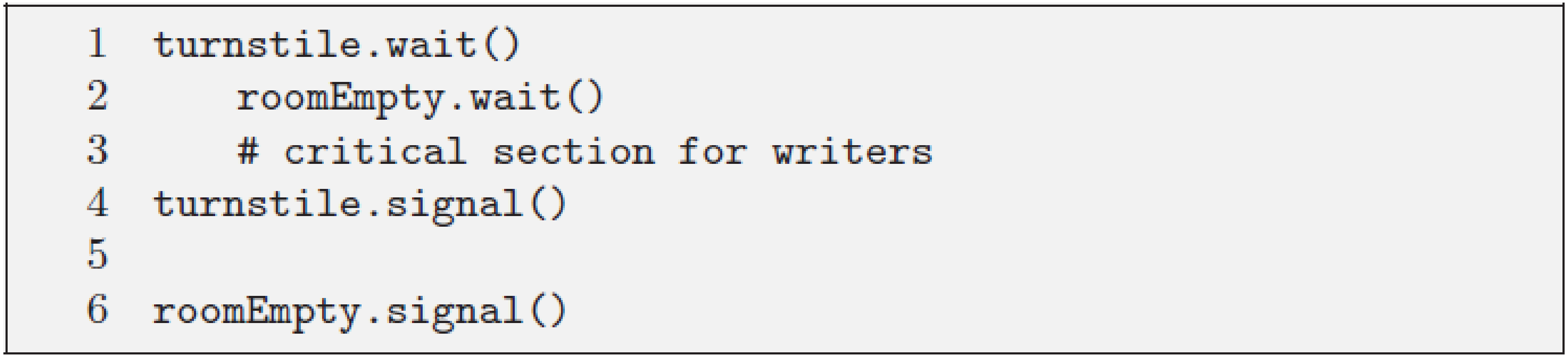


```
readLightswitch.lock(roomEmpty)
# critical section
readLightswitch.unlock(roomEmpty)
```

No-starve writer (93)

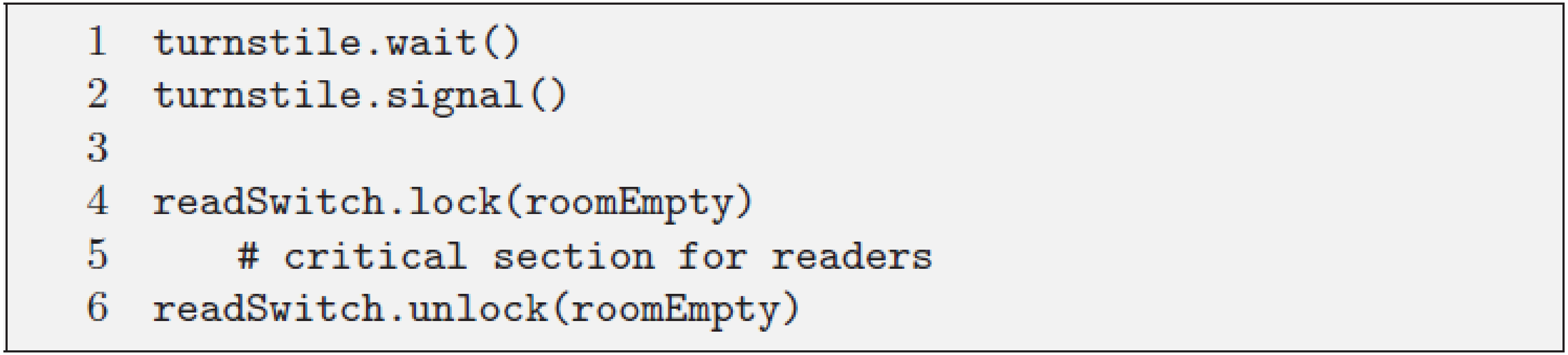


```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)
turnstile = Semaphore(1)
```



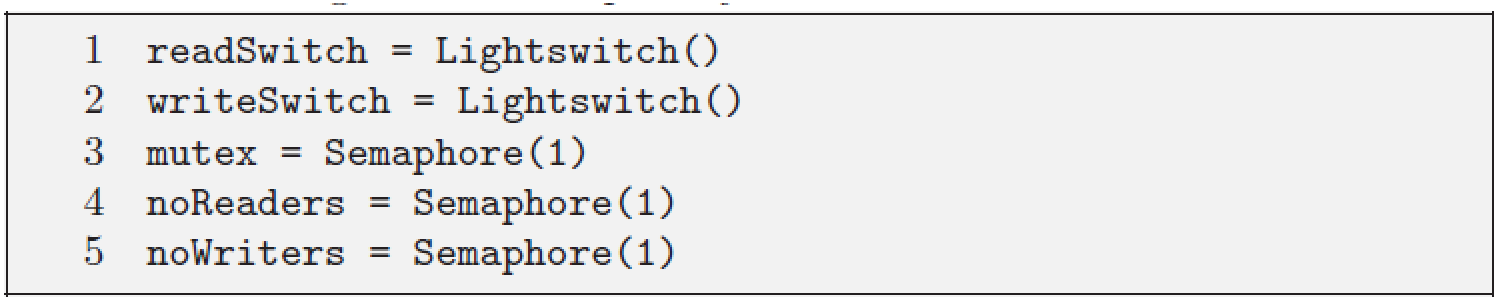
```
turnstile.wait()
    roomEmpty.wait()
    # critical section for writers
turnstile.signal()

roomEmpty.signal()
```

No-starve reader

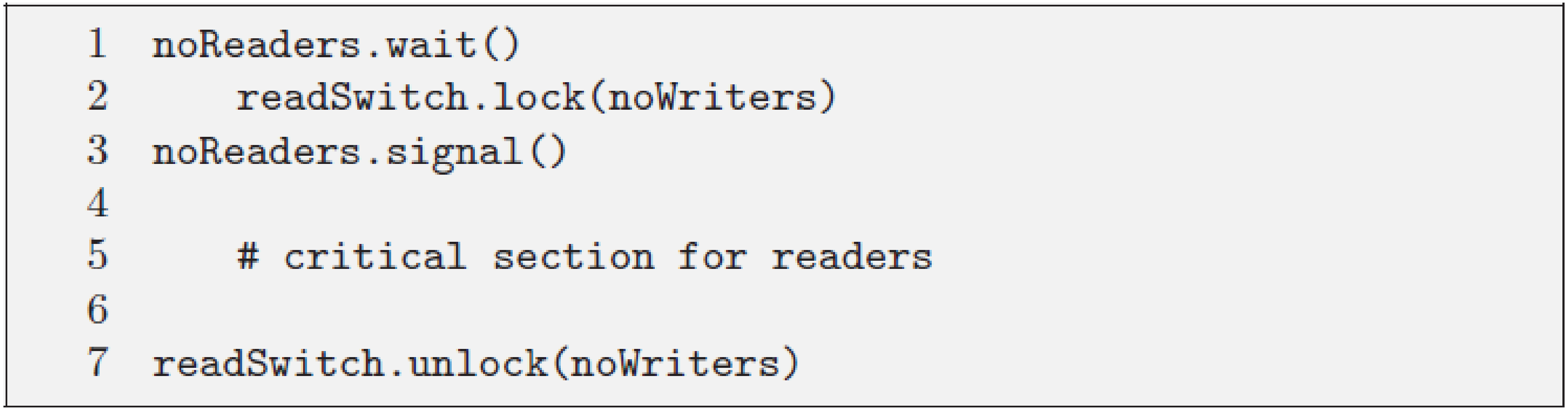


```
turnstile.wait()
turnstile.signal()

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

Writer-priority readers

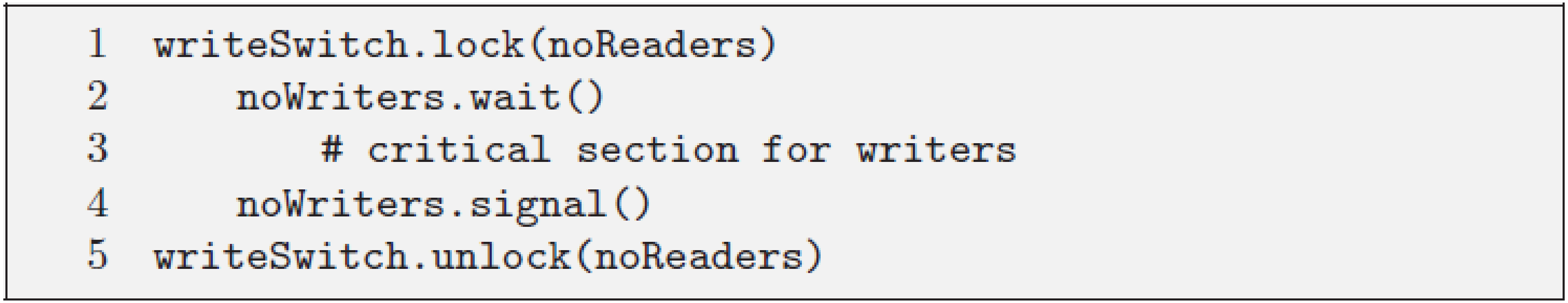


```
readSwitch = Lightswitch()
writeSwitch = Lightswitch()
mutex = Semaphore(1)
noReaders = Semaphore(1)
noWriters = Semaphore(1)
```



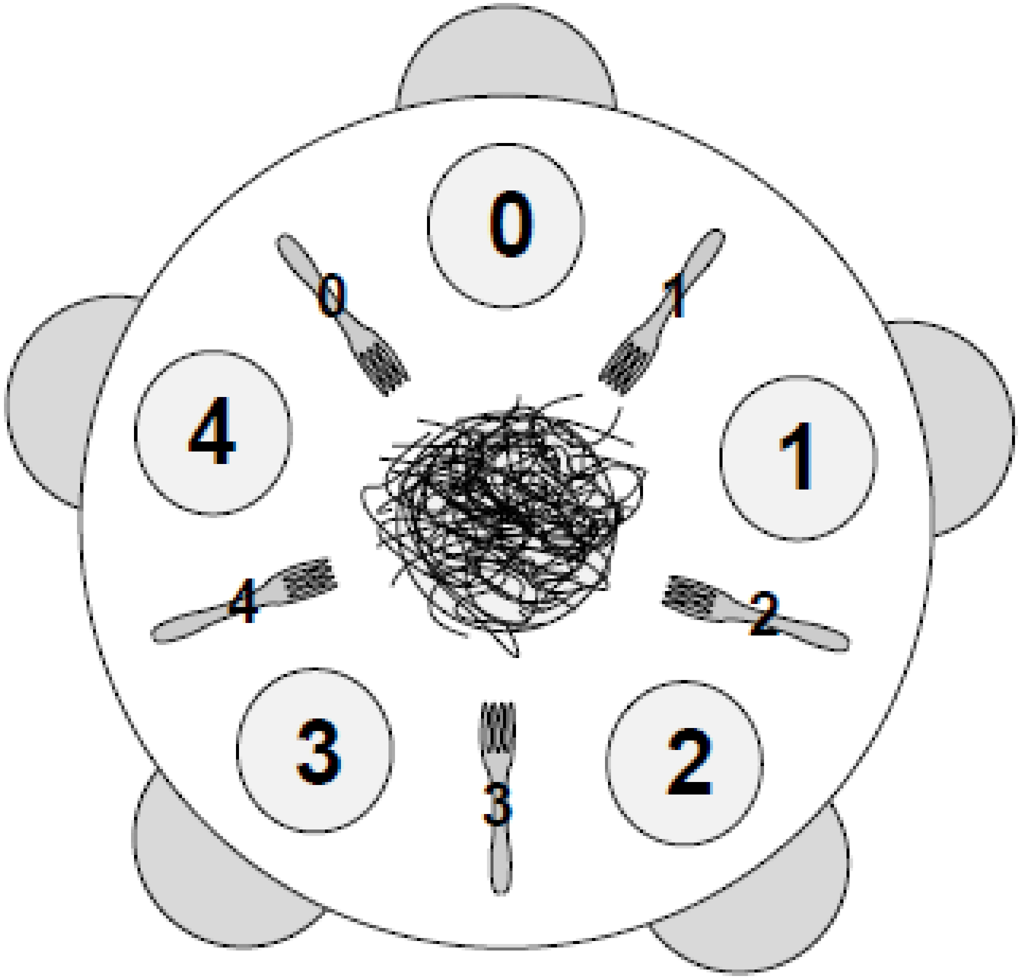
```
noReaders.wait()
    readSwitch.lock(noWriters)
noReaders.signal()

    # critical section for readers

readSwitch.unlock(noWriters)
```

Writer-priority writers

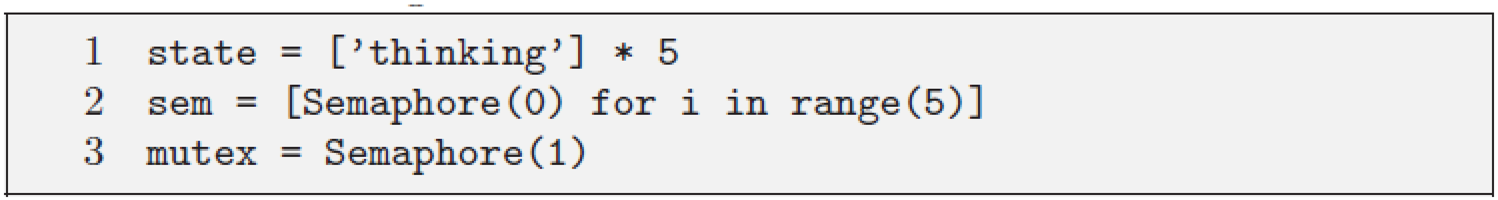


```
writeSwitch.lock(noReaders)
    noWriters.wait()
        # critical section for writers
    noWriters.signal()
writeSwitch.unlock(noReaders)
```

Dining philosophers

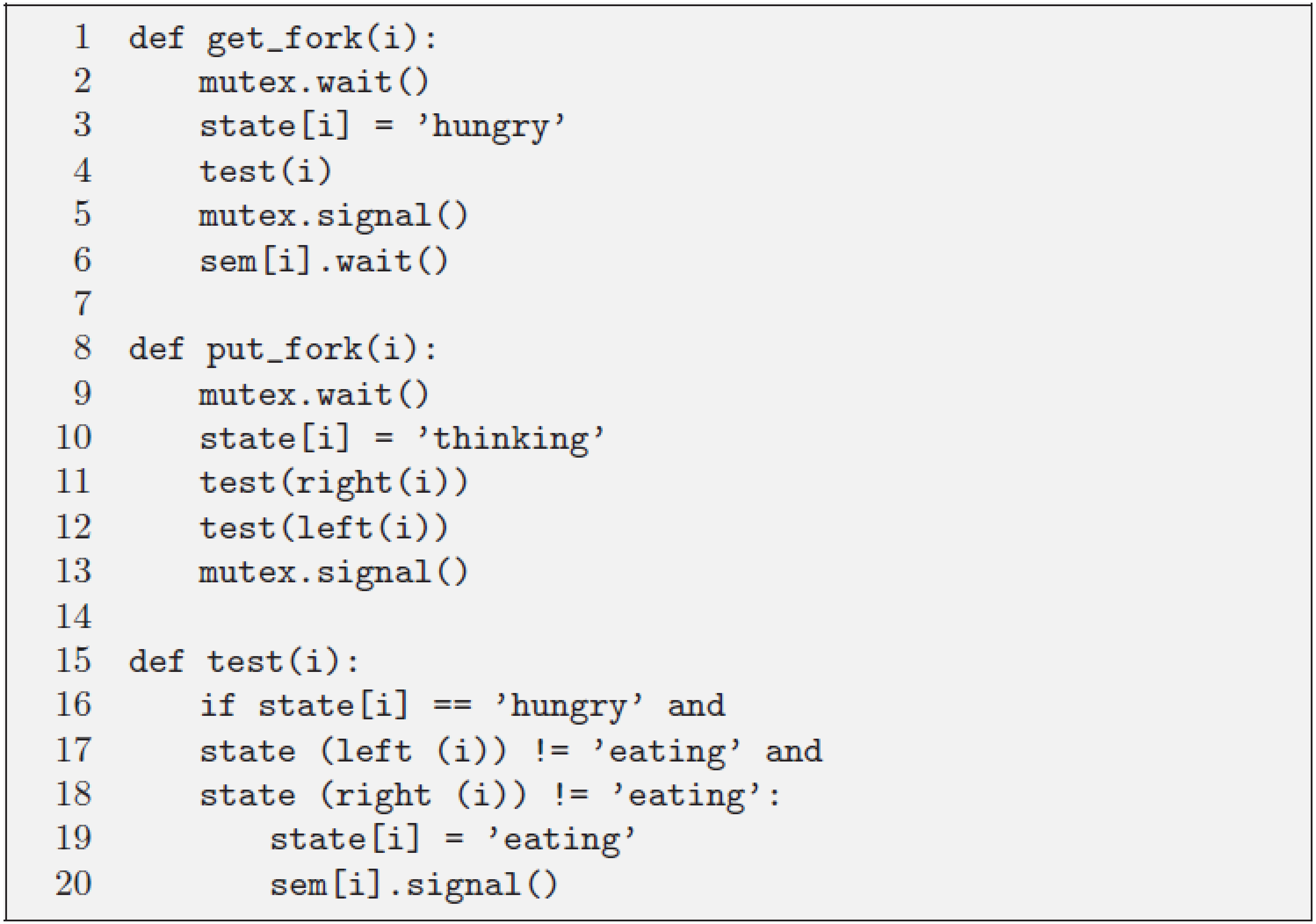


Dining philosophers



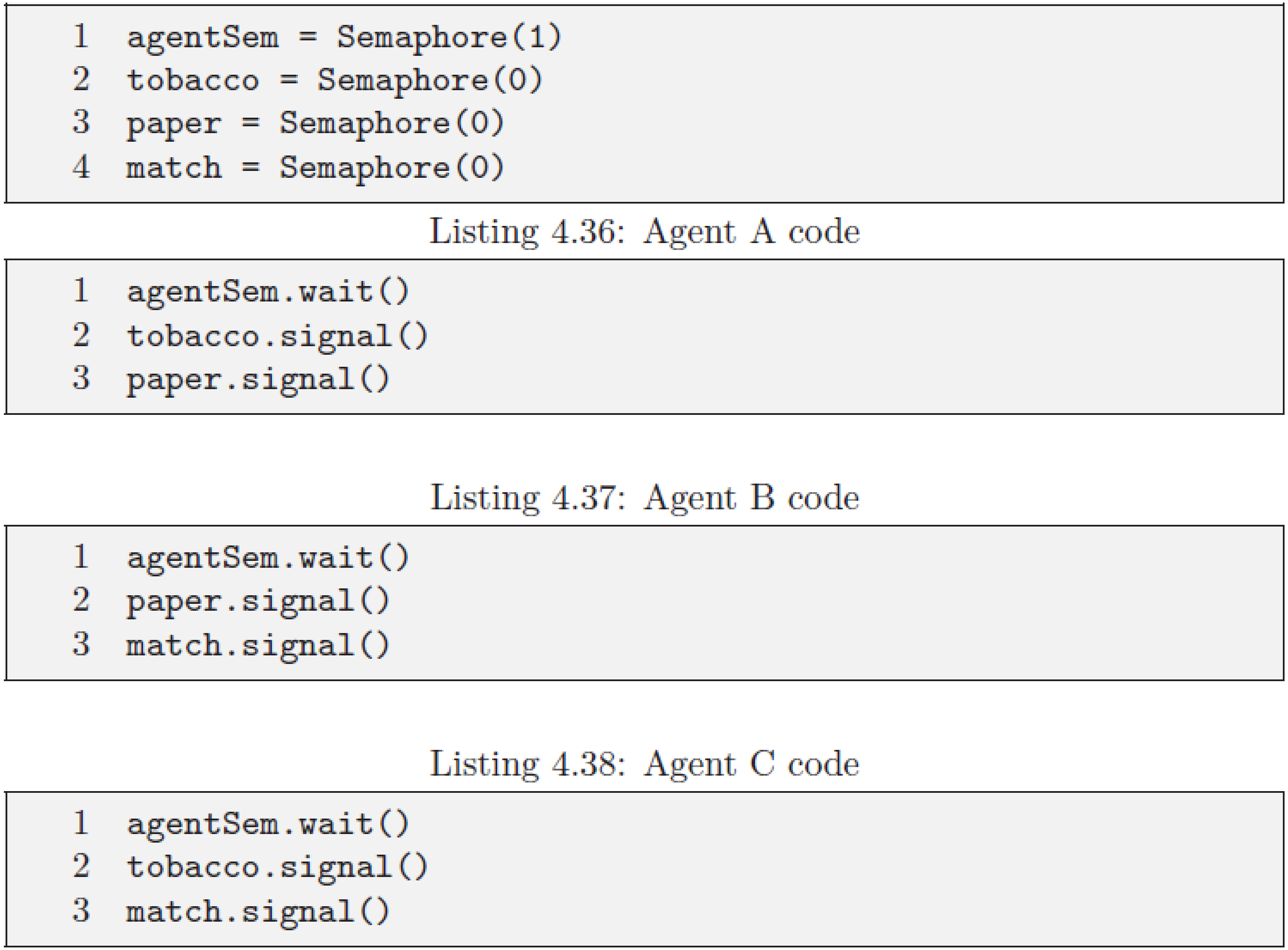
```
state = [’thinking’] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

Dining philosophers



```
def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))
    test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' and
    state (left (i)) != 'eating' and
    state (right (i)) != 'eating':
        state[i] = 'eating'
        sem[i].signal()
```

Cigarette smokers problem



```
agentSem = Semaphore(1)
tobacco = Semaphore(0)
paper = Semaphore(0)
match = Semaphore(0)
```

Listing 4.36: Agent A code

```
agentSem.wait()
tobacco.signal()
paper.signal()
```

Listing 4.37: Agent B code

```
agentSem.wait()
paper.signal()
match.signal()
```

Listing 4.38: Agent C code

```
agentSem.wait()
tobacco.signal()
match.signal()
```

Listing 4.39: Smoker with matches

```
tobacco.wait()
paper.wait()
agentSem.signal()
```

Listing 4.40: Smoker with tobacco

```
paper.wait()
match.wait()
agentSem.signal()
```

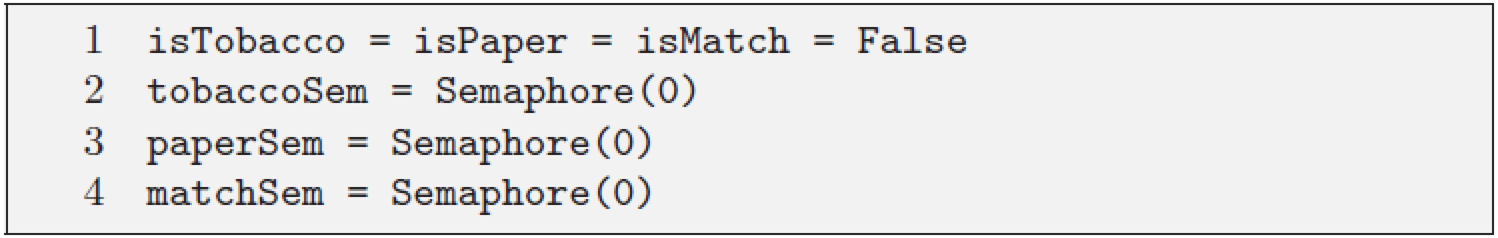
Listing 4.41: Smoker with paper

```
tobacco.wait()
match.wait()
agentSem.signal()
```

Deadlock

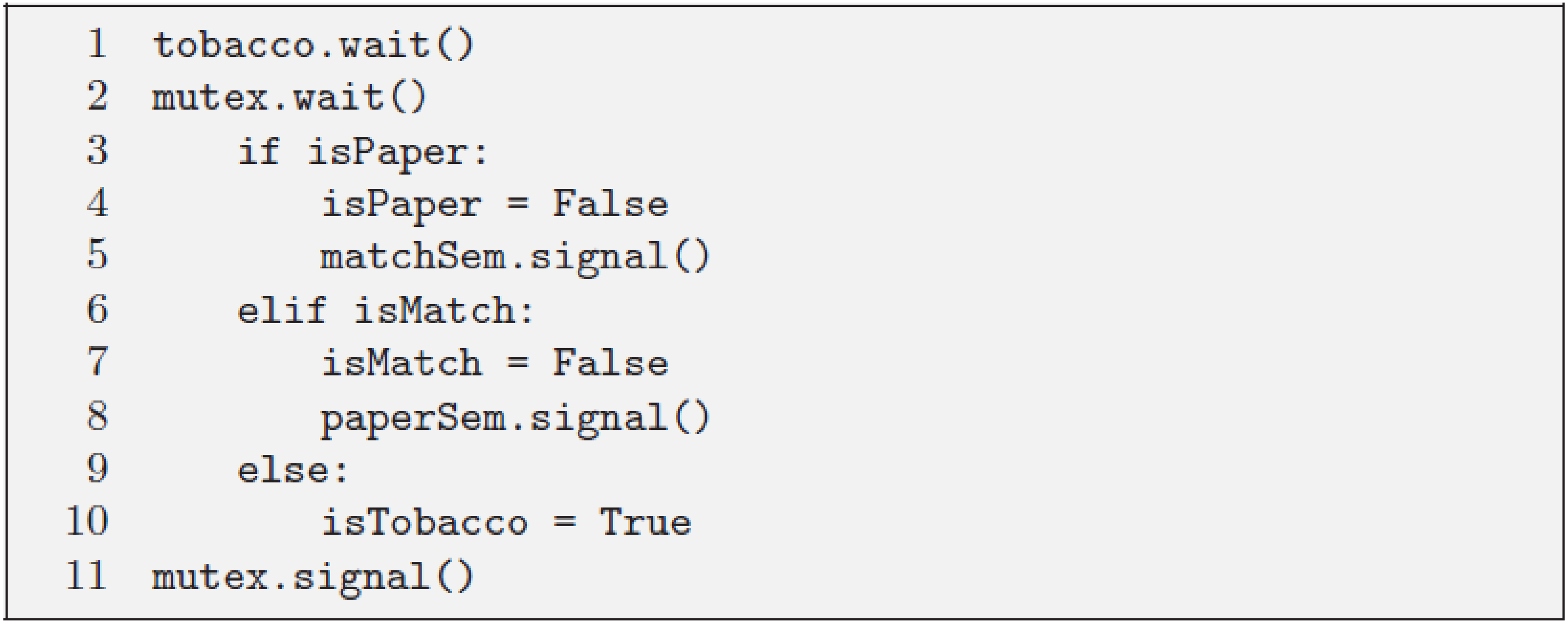
- Two resource, Two request

Cigarette smokers



```
isTobacco = isPaper = isMatch = False
tobaccoSem = Semaphore(0)
paperSem = Semaphore(0)
matchSem = Semaphore(0)
```

Listing 4.43: Pusher A



```
tobacco.wait()
mutex.wait()
    if isPaper:
        isPaper = False
        matchSem.signal()
    elif isMatch:
        isMatch = False
        paperSem.signal()
    else:
        isTobacco = True
mutex.signal()
```

Listing 4.44: Smoker with tobacco

```
tobaccoSem.wait()
makeCigarette()
agentSem.signal()
smoke()
```