FLORIDA TECH
FLORIDA'S **STEM** UNIVERSITY®

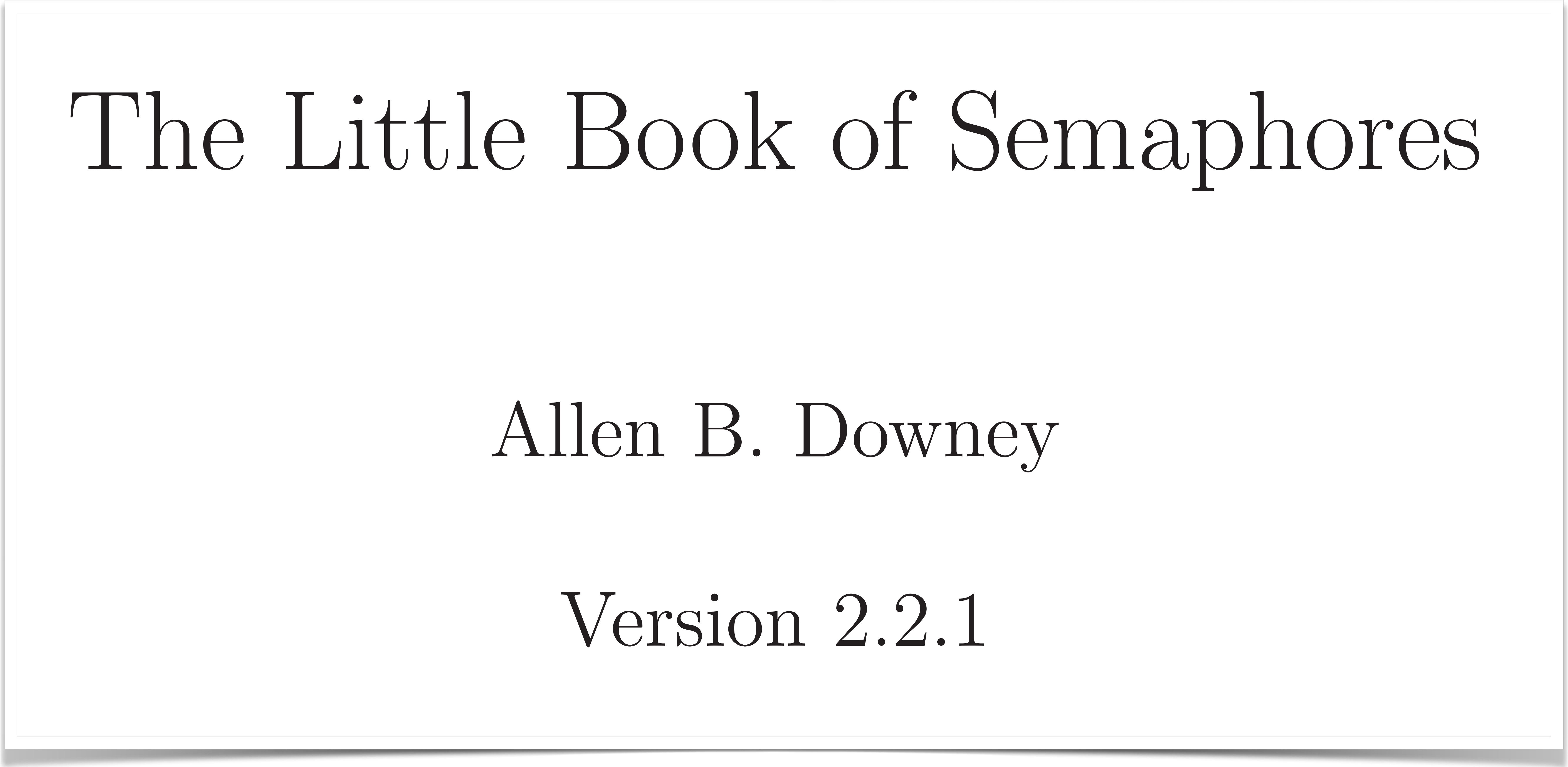
Thread Synchronization

CSE4001 Operating Systems Concepts

E. Ribeiro

Thread Synchronization (Part 1)

CSE 4001



The Little Book of Semaphores

Allen B. Downey

Version 2.2.1

Book URL: https://greenteapress.com/wp/semaphores/

Non-deterministic execution order

- Concurrent programs are often non-deterministic as order of execution depends on the scheduler.

Single program with two threads

Thread A

```
print "yes"
```

Thread B

```
print "no"
```

Concurrent writes on shared variables

- The value that gets printed depends on the order in which the statements are executed (i.e., the execution path).

Single program with two threads

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Concurrent updates on shared variables

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

Translation to machine language

Thread A

```
temp = count
count = temp + 1
```

Thread B

```
temp = count
count = temp + 1
```

Semaphores

A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Semaphore implementation

If semaphore is closed, block the thread that called **`wait()`** on a queue associated with the semaphore. Otherwise, let the thread that called `wait()` continue into the critical section.

The `wait()` function:

```
wait() {
  value = value - 1
  if (value < 0) {
     add this thread to list
     block thread
  }
}
```

Semaphore implementation

Wake up one of the threads that called wait(s), and run it so that it can continue into the critical section.

The signal() function:

```
signal() {
  value = value + 1
  if (value <= 0) {
     remove a thread from list
     wakeup thread
  }
}
```

Semaphores: syntax

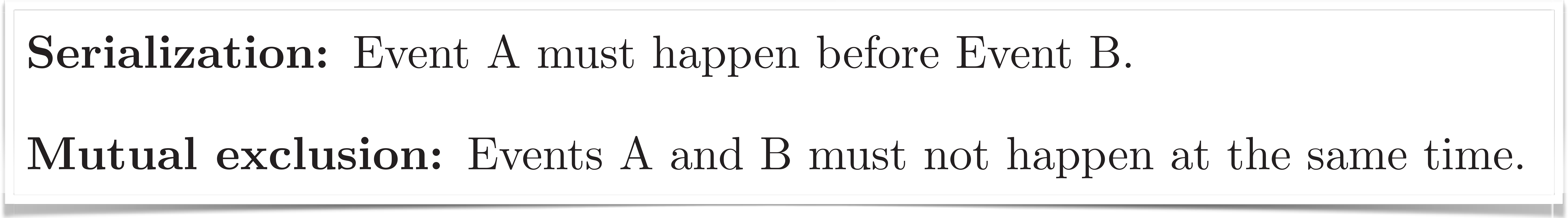
Semaphore initialization syntax

```
fred = Semaphore(1)
```

Semaphore operations

```
fred.signal()
fred.wait()
```

Synchronization Constraints



Serialization: Event A must happen before Event B.

Mutual exclusion: Events A and B must not happen at the same time.

- We will use a combination of these two constraints to solve most thread-synchronization problems

Basic synchronization patterns: Signaling

- **a1** must happen before **b1**

Thread A

```
statement a1

```

Thread B

```

statement b1
```

Basic synchronization patterns: Signaling

- **a1** must happen before **b1**

Thread A

```
statement a1

```

Thread B

```

statement b1
```

Basic synchronization patterns: Signaling

- **a1** must happen before **b1**

```
sem = semaphore(0)
```

Thread A

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

Thread B

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

Basic synchronization patterns: Signaling

- **a1** must happen before **b1**

```
sem = semaphore(0)
```

Thread A

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

Thread B

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

Basic synchronization patterns: Signaling

- **a1** must happen before **b1**
- Same solution using better semaphore naming

```
a1Done = semaphore(0)
```

Thread A

```
statement a1
a1Done.signal()
```

Thread B

```
a1Done.wait()
statement b1
```

Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**

Thread A

```
statement a1
statement a2
```

Thread B

```
statement b1
statement b2
```

Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**

Thread A

```
statement a1


statement a2
```

Thread B

```
statement b1


statement b2
```

Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**

Thread A

```
statement a1

bArrived.wait()
statement a2
```

Thread B

```
statement b1

aArrived.wait()
statement b2
```

Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**

Thread A

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

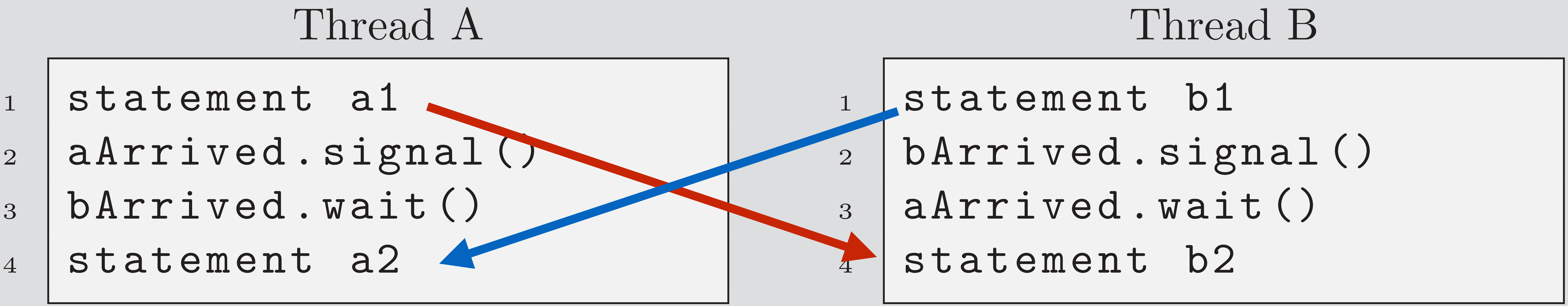
Thread B

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

Basic synchronization patterns: Rendezvous

- **a1** must happen before **b2**
- **b1** must happen before **a2**

```
aArrived = semaphore(0)
bArrived = semaphore(0)
```



Thread A

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

Thread B

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

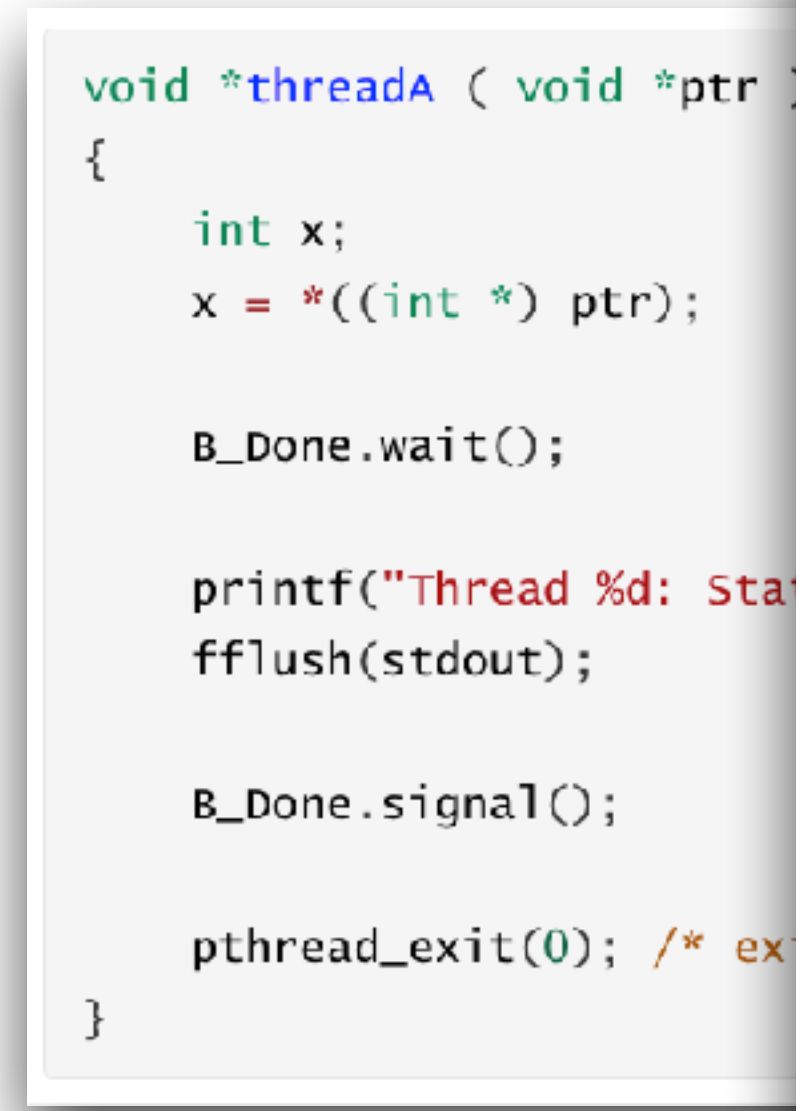
Example

```
#include "semaphore_class.h"

/* prototype for thread routine */
void *threadB ( void *ptr );
void *threadA ( void *ptr );

/* global vars */
Semaphore B_Done(0);

int main()
{
    int i[3];
    pthread_t thread_a;
    pthread_t thread_b;
    i[0] = 0; i[1] = 1;   /* argument to threads */

    pthread_create (&thread_a, NULL, threadA, (void *) &i[0]);
    pthread_create (&thread_b, NULL, threadB, (void *) &i[1]);

    exit(0);

} /* main() */
```

```
void *threadA ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    B_Done.wait();

    printf("Thread %d: Statement A: Must run after Statement B. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}
```

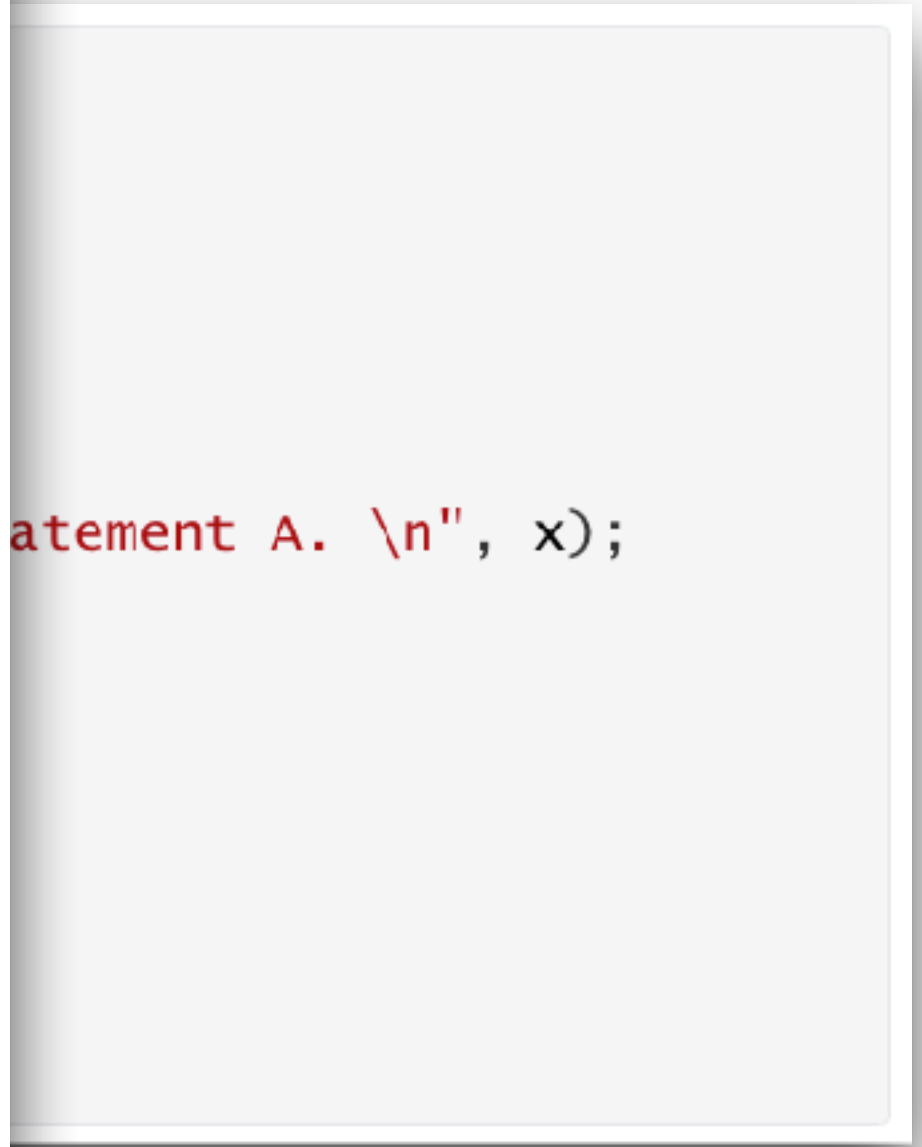
```
void *threadB ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    printf("Thread %d: Statement B: Must run before Statement A. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}
```

Example



```
void *threadA ( void *ptr
{
    int x;
    x = *((int *) ptr);

    B_Done.wait();

    printf("Thread %d: Sta
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* ex
}
```

```
#include "semaphore_class.h"

/* prototype for thread routine */
void *threadB ( void *ptr );
void *threadA ( void *ptr );

/* global vars */
Semaphore B_Done(0);

int main()
{
    int i[3];
    pthread_t thread_a;
    pthread_t thread_b;
    i[0] = 0; i[1] = 1;   /* argument to threads */

    pthread_create (&thread_a, NULL, threadA, (void *) &i[0]);
    pthread_create (&thread_b, NULL, threadB, (void *) &i[1]);

    exit(0);

} /* main() */
```



```
atement A. \n", x);
```

Example

```
#include "semaphore_class.h"
```

```
void *threadA ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    B_Done.wait();

    printf("Thread %d: Statement A: Must run after Statement B. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}
```

Example

```
#include "semaphore_class.h"

/* prototype for thread routine */
void *threadB ( void *ptr );
void *threadA ( void *ptr );

/* global vars */
Semaph

int ma
{
    in
    pt
    pt
    i[

    pt
    pt

    ex

} /* m
```

```
void *
{
    i
    x

    B_

    pr
    ff

    B_

    pt
}
```

```
void *threadB ( void *ptr )
{
    int x;
    x = *((int *) ptr);

    printf("Thread %d: Statement B: Must run before Statement A. \n", x);
    fflush(stdout);

    B_Done.signal();

    pthread_exit(0); /* exit thread */
}
```

Thread Synchronization (Part 2)

CSE 4001

Contents

- Semaphore implementation
- The mutual exclusion constraint
- The producer-consumer problem

Semaphore implementation

If semaphore is closed, block the thread that called **`wait()`** on a queue associated with the semaphore. Otherwise, let the thread that called `wait()` continue into the critical section.

The `wait()` function:

```
wait() {
  value = value - 1
  if (value < 0) {
      add this thread to list
      block thread
  }
}
```

Semaphore implementation

Wake up one of the threads that called wait(s), and run it so that it can continue into the critical section.

The signal() function:

```
signal() {
  value = value + 1
  if (value <= 0) {
     remove a thread from list
     wakeup thread
  }
}
```

Mutual exclusion

- A second common use of semaphores: to enforce mutual exclusion and controlling concurrent access to shared variables.
- The mutex guarantees that only one thread accesses the shared variable at a time.

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

Mutual exclusion hint

- Create a semaphore named `mutex` that is initialized to 1.
- A value of one means that a thread may proceed and access the shared variable;
- A value of zero means that it has to wait for another thread to release the mutex.

Mutual exclusion solution

Thread A

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

Thread B

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

Multiplex

- It allows multiple threads to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent threads.
- In other words, no more than `n` threads can run in the critical section at the same time.

hint: treat semaphores as a set of tokens. If no tokens are available when a thread arrives at the critical section, it waits until another thread releases one.

The producer-consumer problem

Producer

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

Consumer

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

Access to the buffer has to be exclusive, but `waitForEvent()` and `event.process()` can run concurrently.

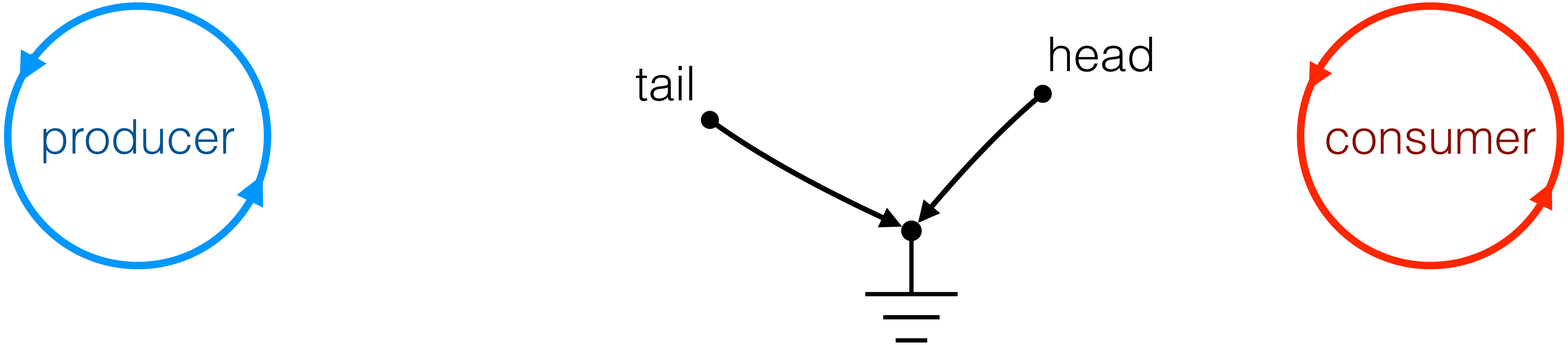
Empty buffer: consumer runs first

`producer()`

```
while true {
    item = getEvent()
    buffer.add(event)
}
```

`consumer()`

```
while true {
  event = buffer.get()
  event.process()
}
```



Empty buffer: consumer runs first

producer()

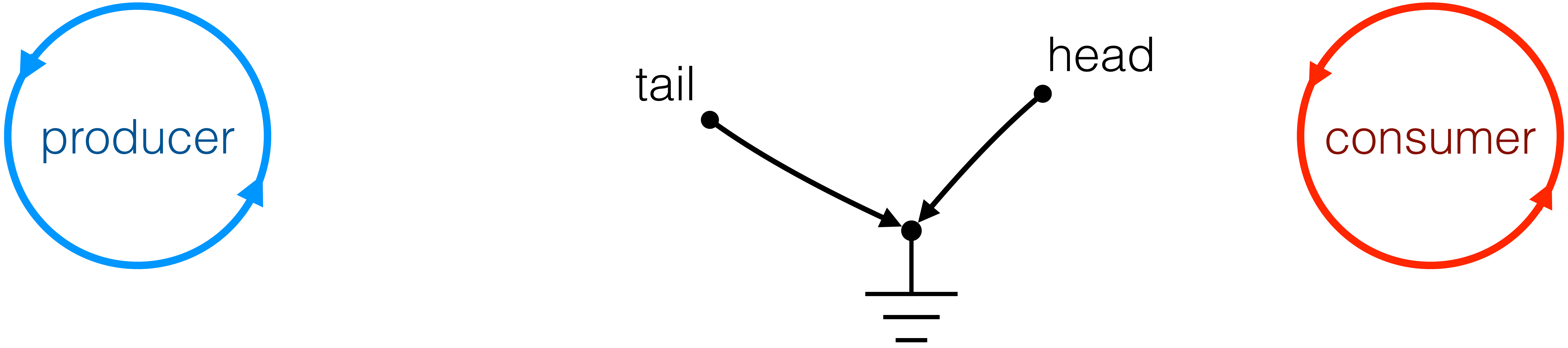
```
while true {
    item = getEvent()
    buffer.add(event)
}
```

signal()

wait()

consumer()

```
while true {
   event = buffer.get()
   event.process()
}
```



Empty buffer: consumer runs first

producer()

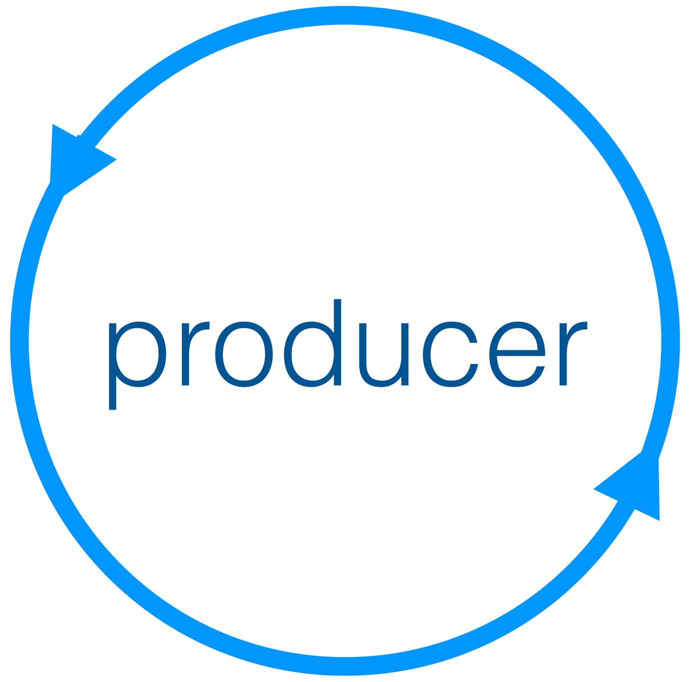
```
while true {
    item = getEvent()
    buffer.add(event)
}
```

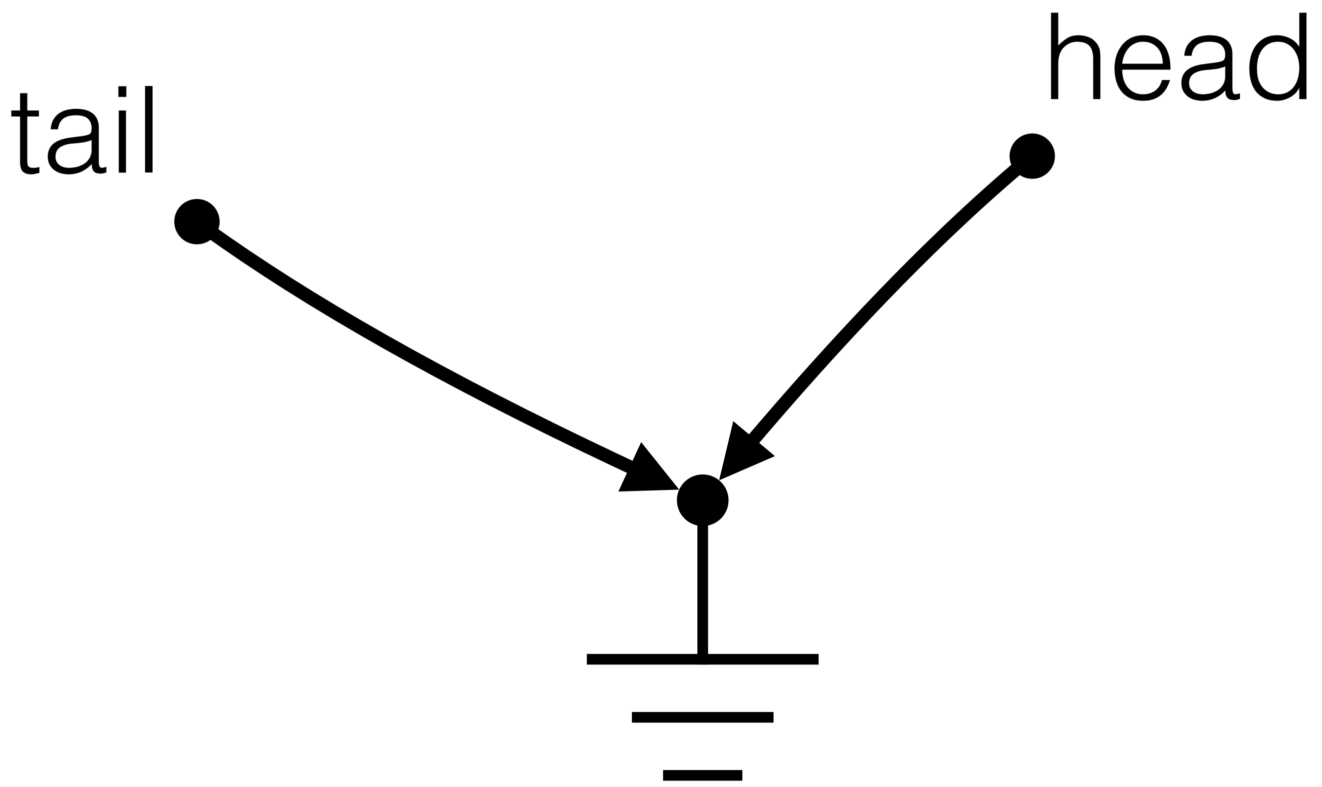
signal()

wait()

consumer()

```
while true {
  items.wait()
  event = buffer.get()
  event.process()
}
```





Empty buffer: consumer runs first

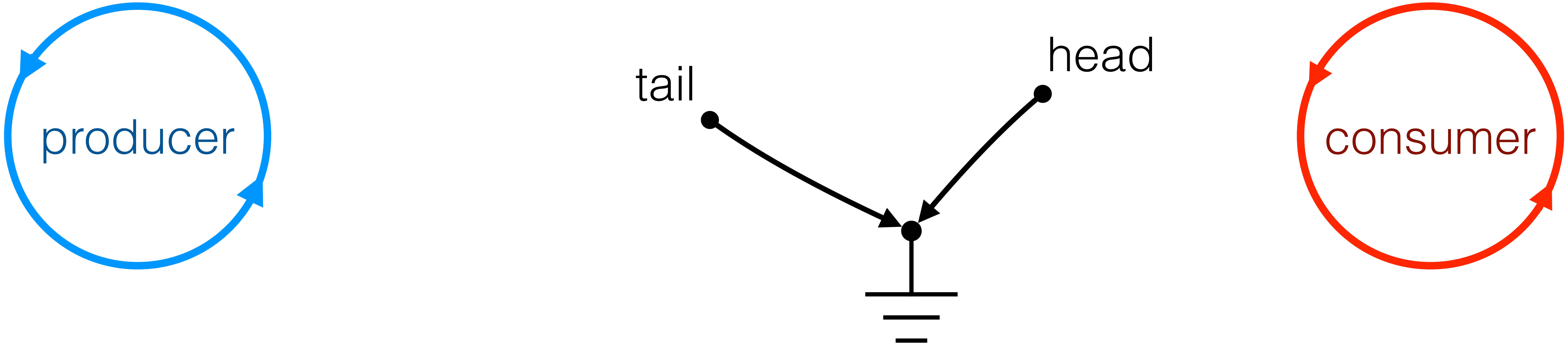
```
items = semaphore(0)
```

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
}
```

consumer()

```
while true {
  items.wait()
  event = buffer.get()
  event.process()
}
```



Empty buffer: consumer runs first

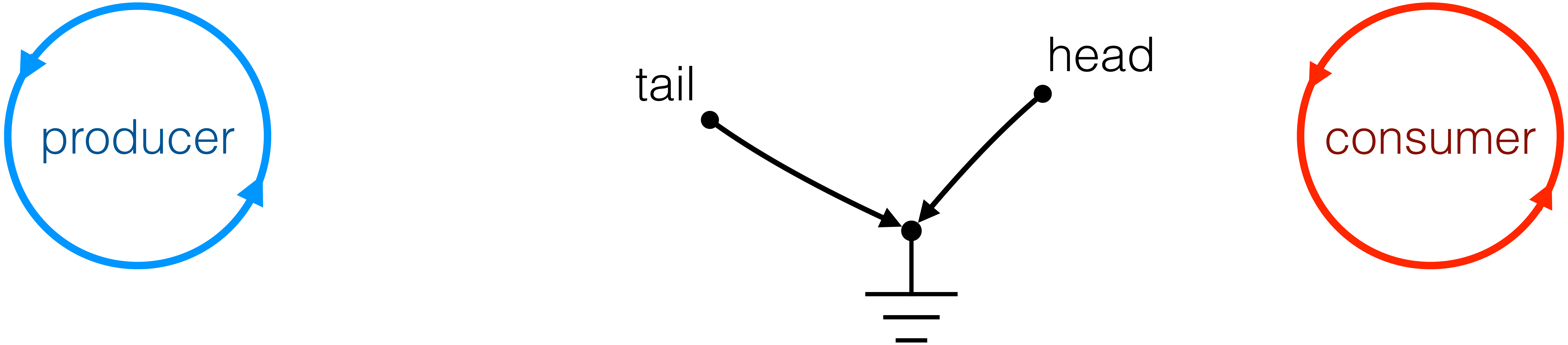
```
items = semaphore(0)
```

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
    items.signal()
}
```

consumer()

```
while true {
   items.wait()
   event = buffer.get()
   event.process()
}
```



Empty buffer: consumer runs first

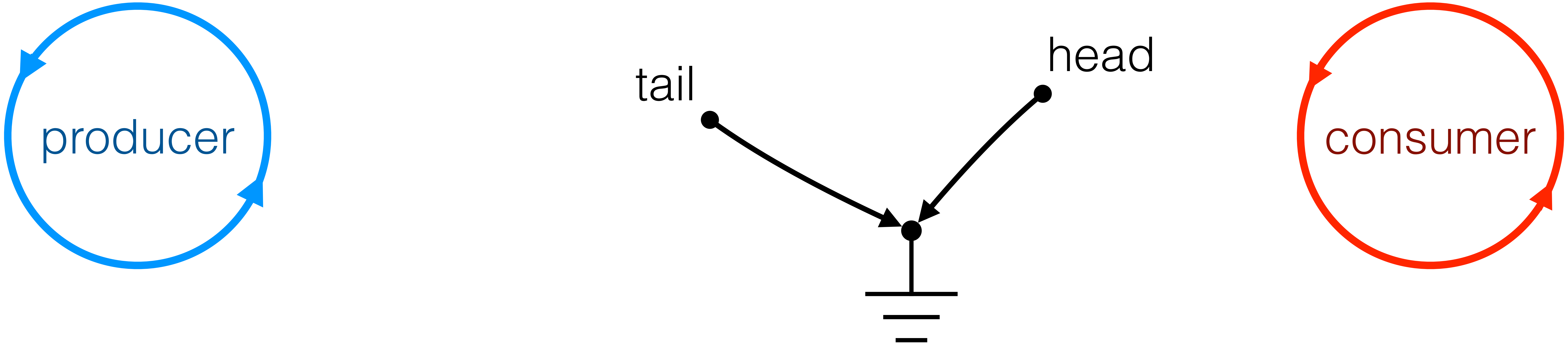
```
items = semaphore(0)
```

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
    items.signal()
}
```

consumer()

```
while true {
  items.wait()
  event = buffer.get()
  event.process()
}
```



Empty buffer: consumer runs first

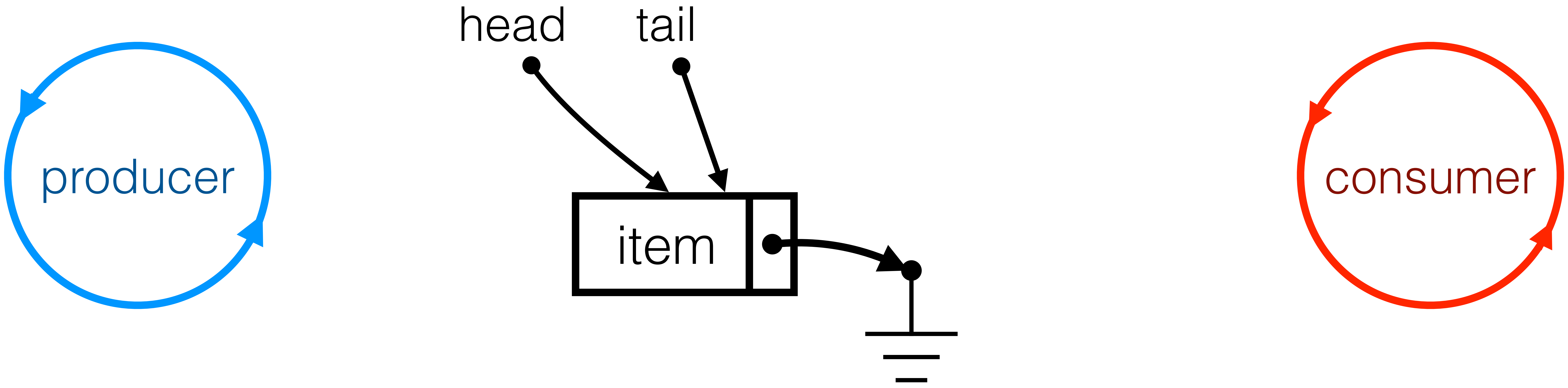
```
items = semaphore(0)
```

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
    items.signal()
}
```

consumer()

```
while true {
   items.wait()
   event = buffer.get()
   event.process()
}
```



Concurrent writes: add and get cannot take place at the same time

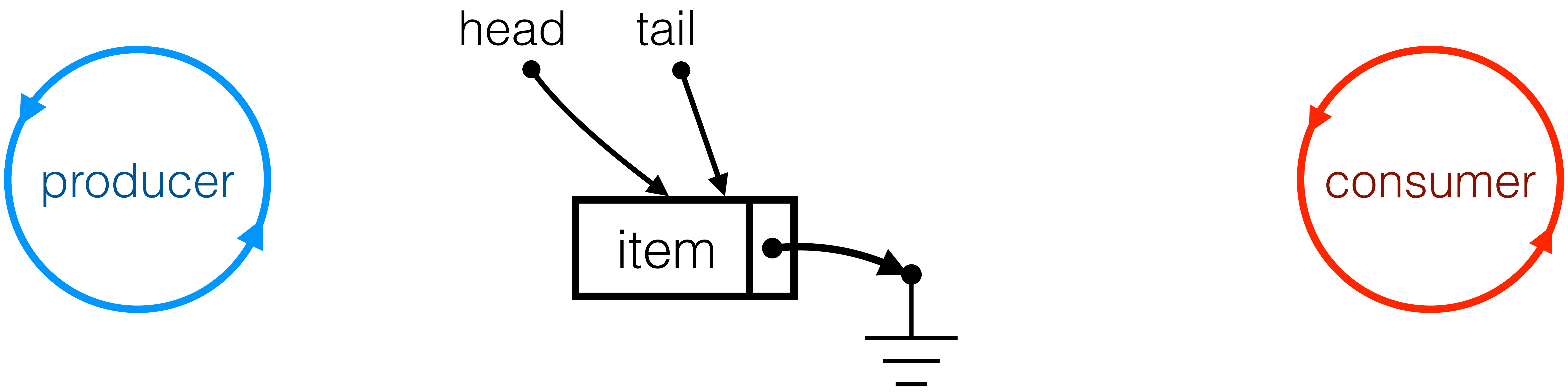
```
items = semaphore(0)
```

producer()

```
while true {
    item = getEvent()
    buffer.add(event)
    items.signal()
}
```

consumer()

```
while true {
    items.wait()
    event = buffer.get()
    event.process()
}
```



Concurrent writes: add and get cannot take place at the same time

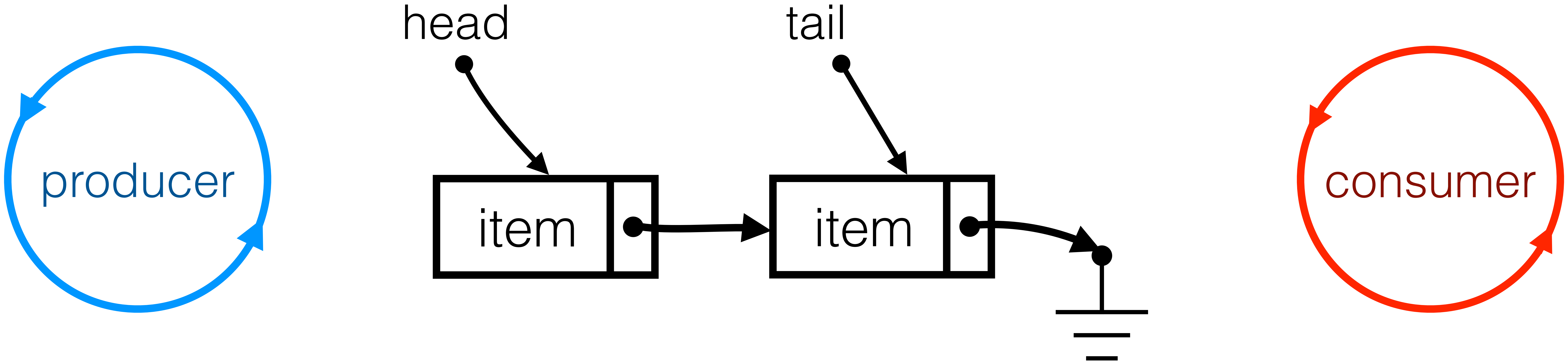
```
items = semaphore(0)
mutex = semaphore(1)
```

producer()

```
while true {
    item = getEvent()
    mutex.wait()
       buffer.add(event)
    mutex.signal()
    items.signal()
}
```

consumer()

```
while true {
  items.wait()
  mutex.wait()
    event = buffer.get()
  mutex.signal()
  event.process()
}
```



Limited buffer size: producer sleeps once the maximum buffer length is reached.

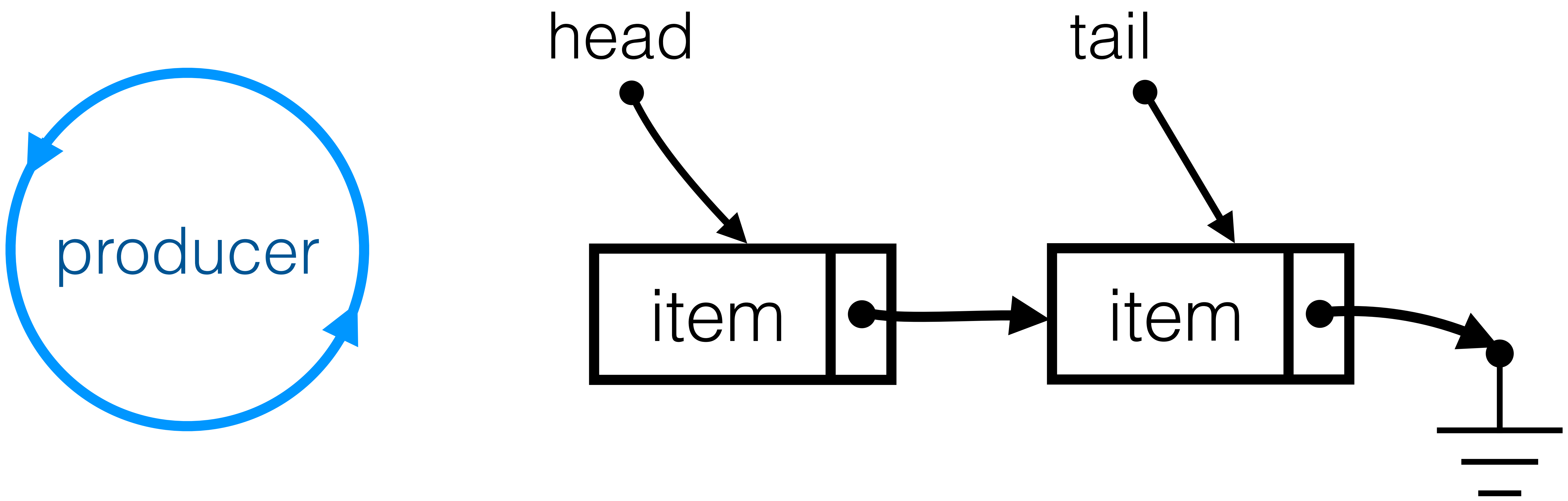
```
items = semaphore(0)
mutex = semaphore(1)
```

producer()

```
while true {
    item = getEvent()
     mutex.wait()
        buffer.add(event)
     mutex.signal()
    items.signal()
}
```

consumer()

```
while true {
   items.wait()
   mutex.wait()
     event = buffer.get()
   mutex.signal()
   event.process()
}
```



Limited buffer size: producer sleeps once the maximum buffer length is reached.

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

producer()

```
while true {
    item = getEvent()
    spaces.wait()
    mutex.wait()
       buffer.add(event)
    mutex.signal()
    items.signal()
}
```

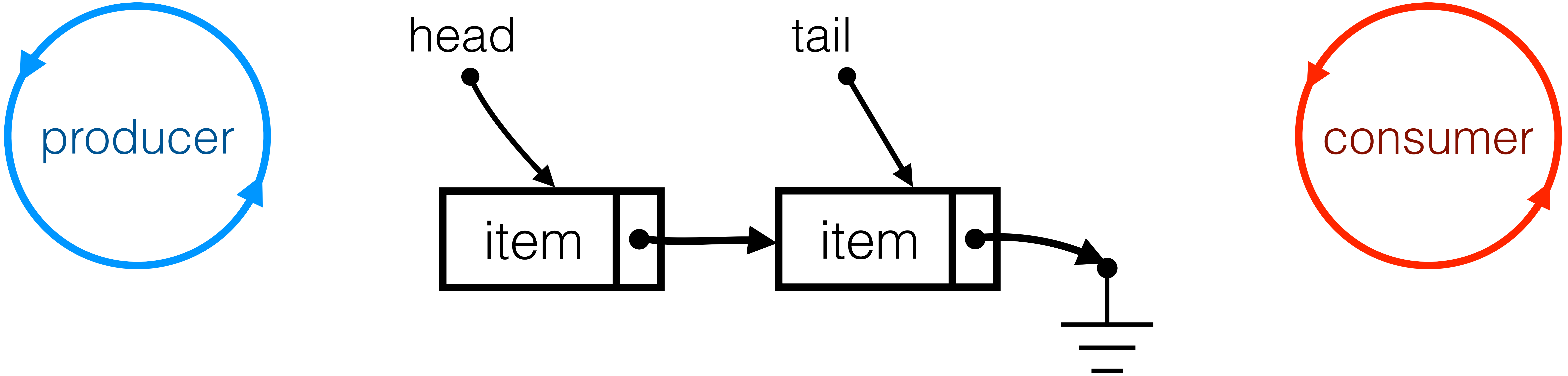
token interpretation of the multiplex

wait()

signal()

consumer()

```
while true {
   items.wait()
   mutex.wait()
     event = buffer.get()
   mutex.signal()
   spaces.signal()
   event.process()
}
```



Limited buffer size: producer sleeps once the maximum buffer length is reached.

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

producer()

```
while true {
    item = getEvent()
    spaces.wait()
    mutex.wait()
        buffer.add(event)
    mutex.signal()
    items.signal()
}
```

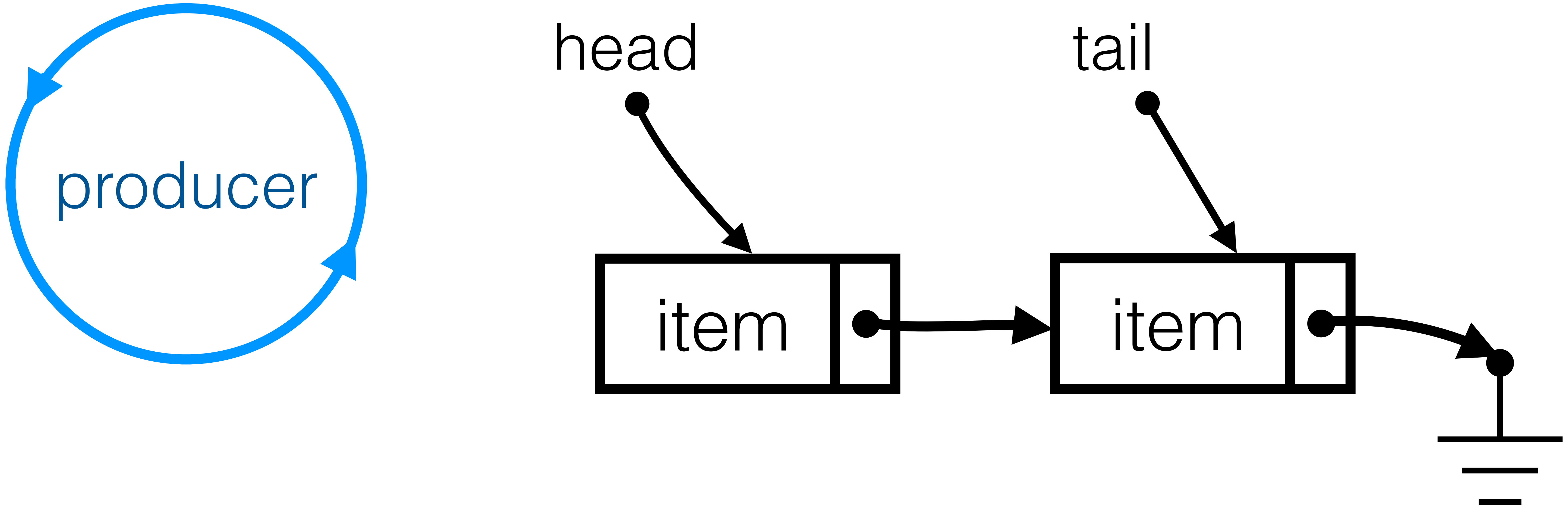
token interpretation of the multiplex

wait()

signal()

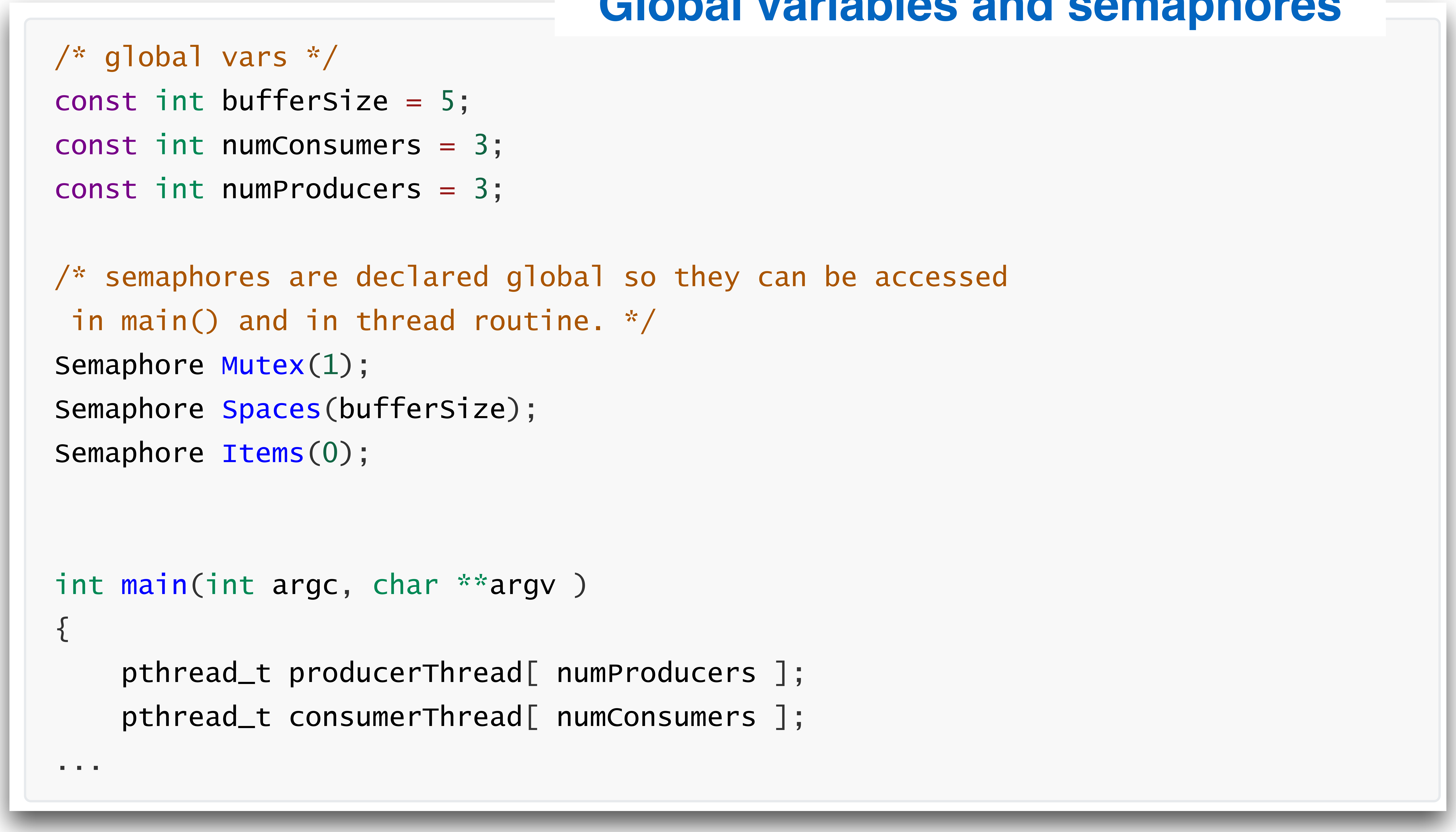
consumer()

```
while true {
   items.wait()
   mutex.wait()
     event = buffer.get()
   mutex.signal()
   spaces.signal()
   event.process()
}
```



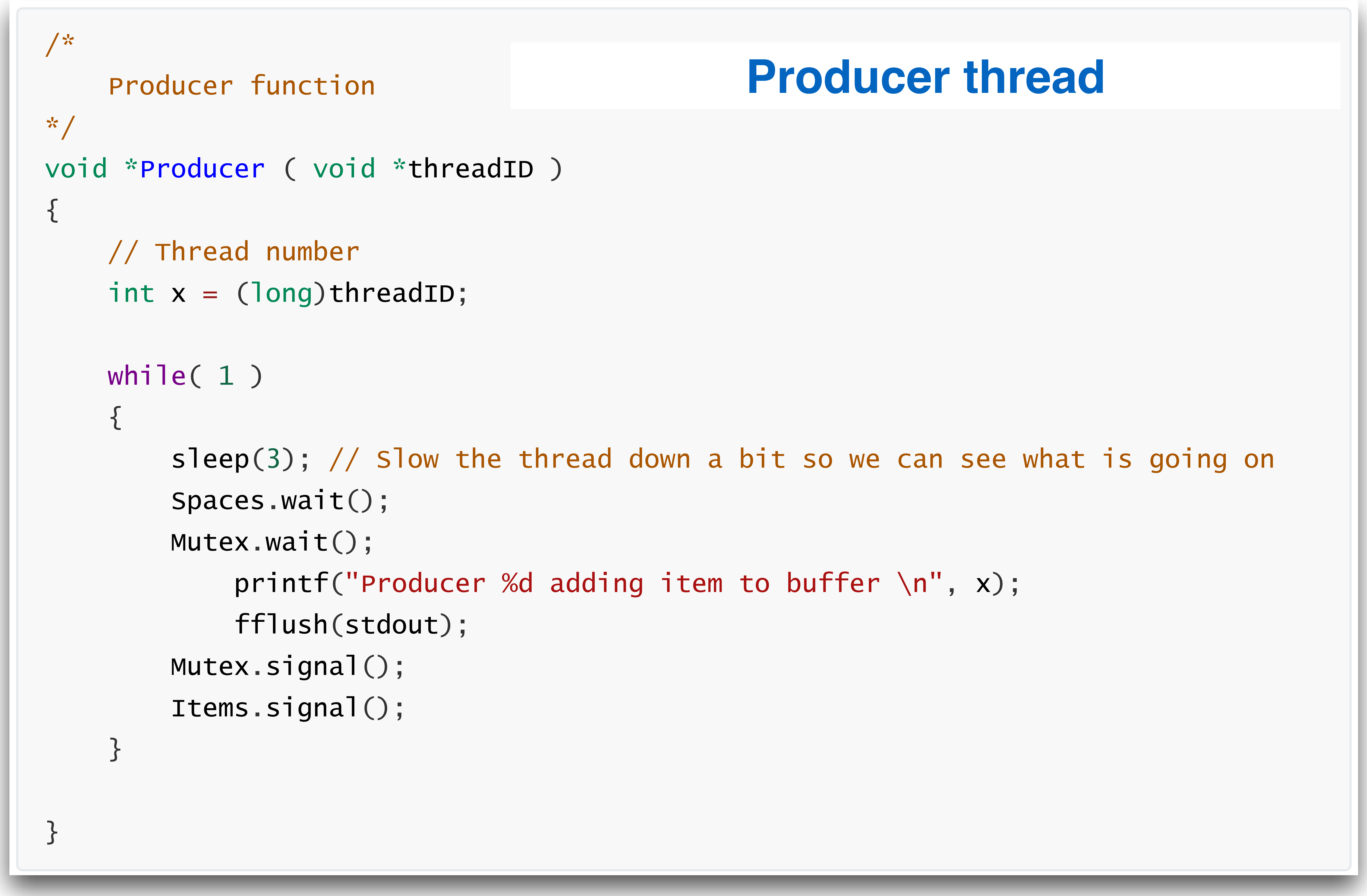
Implementation Example: **Producer-Consumer**

Global variables and semaphores



```
/* global vars */
const int bufferSize = 5;
const int numConsumers = 3;
const int numProducers = 3;

/* semaphores are declared global so they can be accessed
 in main() and in thread routine. */
Semaphore Mutex(1);
Semaphore Spaces(bufferSize);
Semaphore Items(0);


int main(int argc, char **argv )
{
    pthread_t producerThread[ numProducers ];
    pthread_t consumerThread[ numConsumers ];
...
```

Implementation Example: **Producer-Consumer**

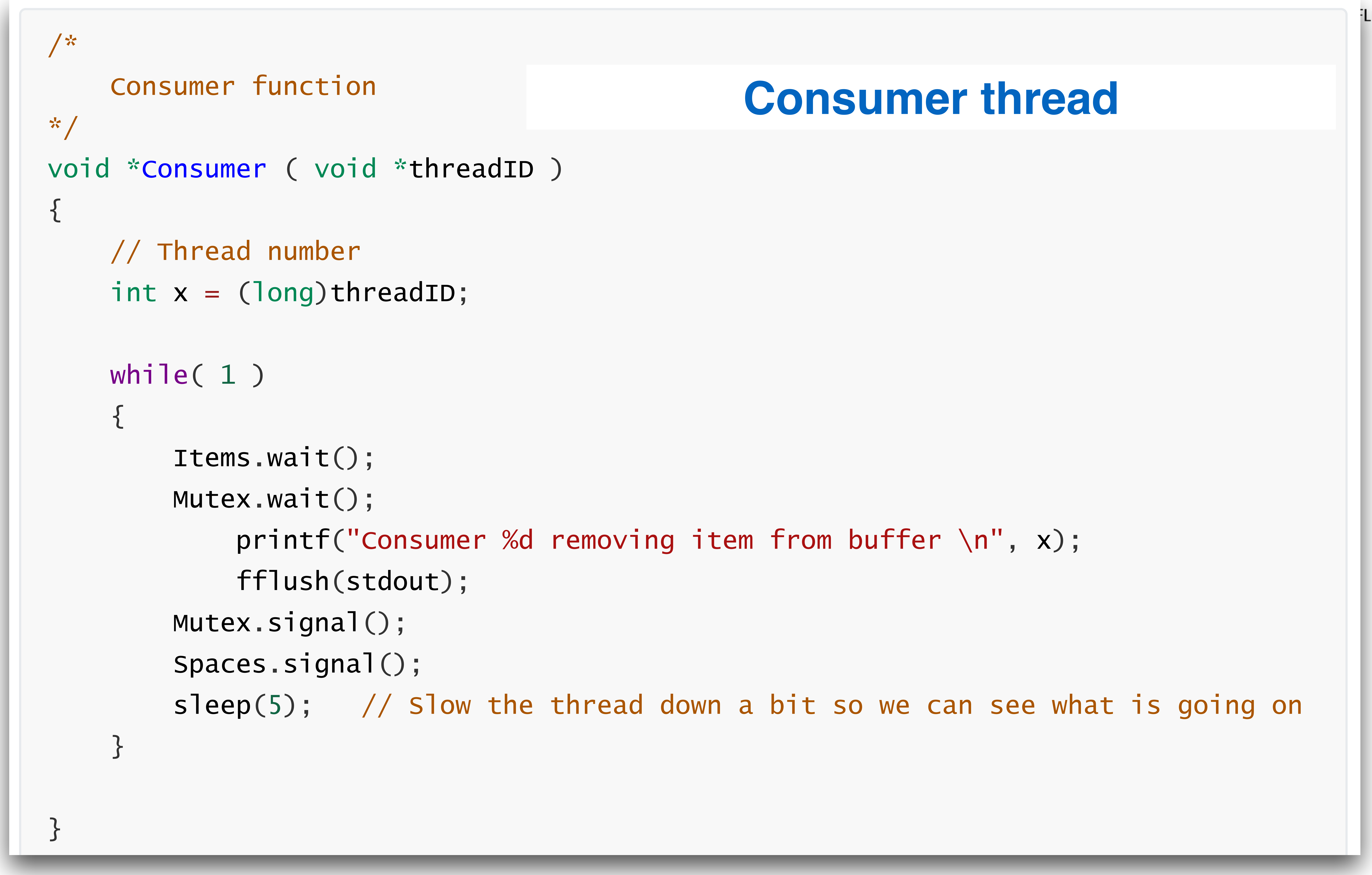


Producer thread

```
/*
    Producer function
*/
void *Producer ( void *threadID )
{
    // Thread number
    int x = (long)threadID;

    while( 1 )
    {
        sleep(3); // Slow the thread down a bit so we can see what is going on
        Spaces.wait();
        Mutex.wait();
            printf("Producer %d adding item to buffer \n", x);
            fflush(stdout);
        Mutex.signal();
        Items.signal();
    }

}
```

Implementation Example: **Producer-Consumer**

Consumer thread



```
/*
    Consumer function
*/
void *Consumer ( void *threadID )
{
    // Thread number
    int x = (long)threadID;

    while( 1 )
    {
        Items.wait();
        Mutex.wait();
            printf("Consumer %d removing item from buffer \n", x);
            fflush(stdout);
        Mutex.signal();
        Spaces.signal();
        sleep(5);   // Slow the thread down a bit so we can see what is going on
    }

}
```

Thread Synchronization (Part 3)

CSE 4001

Contents

- The readers-writers problem

Readers-Writers

A data set is shared among a number of concurrent threads

- **Readers**: Only read the data set; they do not perform any updates
- **Writers**: Can both read and write

Problem: Allow multiple readers to read at the same time. Only one single writer can access the shared data at any time.

Readers-Writers

Here is a set of variables that is sufficient to solve the problem

```
int readers = 0               // no. of readers in the room
mutex = Semaphore(1)          // protects the counter
roomEmpty = Semaphore(1)      // 1 if room is empty
```

Readers-Writers

Writer

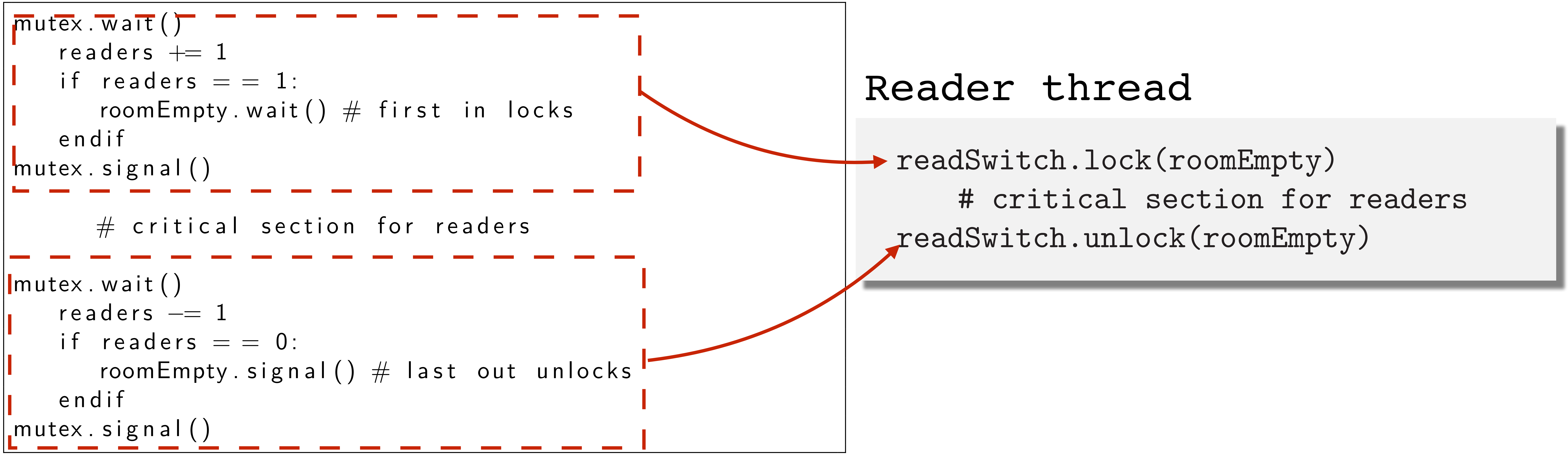
```
roomEmpty.wait()
    critical section for writers
roomEmpty.signal()
```

Readers-Writers

Reader

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

       # critical section for readers

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

Readers-Writers



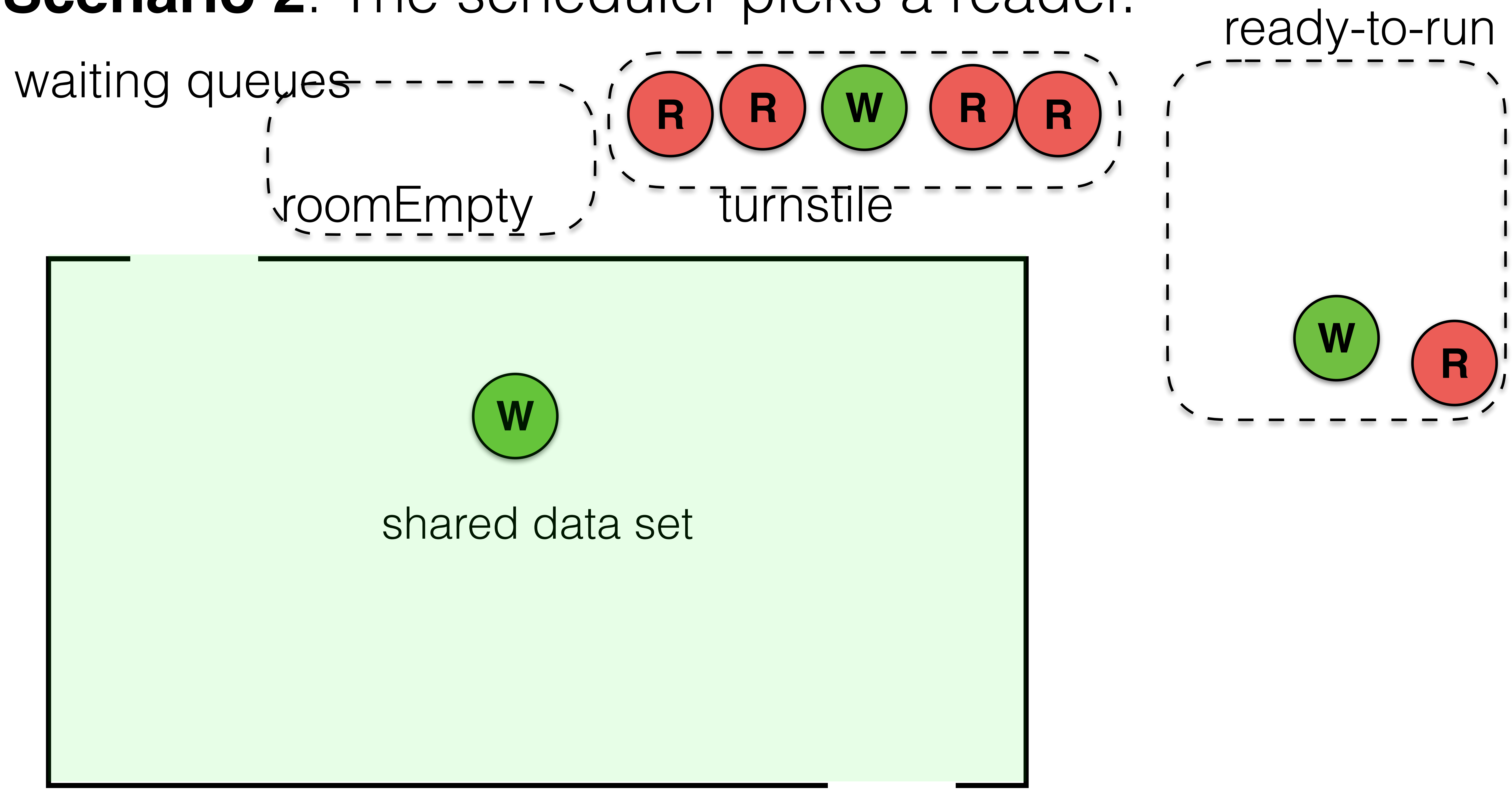
Reader

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

      # critical section for readers

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

Reader thread

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

Scenario 2: The scheduler picks a reader.



writer thread

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

roomEmpty.signal()
```

reader thread

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

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

Readers-Writers

Depending on the application, it might be a good idea to give more priority to writers. For example, if writers are making time-critical updates to a data structure, it is best to minimize the number of readers that see the old data before the writer has a chance to proceed.

Readers-Writers

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

`turnstile` is a turnstile for readers and a mutex for writers. Readers will need to queue on the turnstile if a writer gets stuck inside it.

Readers-Writers

writer thread

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

roomEmpty.signal()
```

reader thread

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

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

Thread Synchronization (Part 4)

CSE 4001

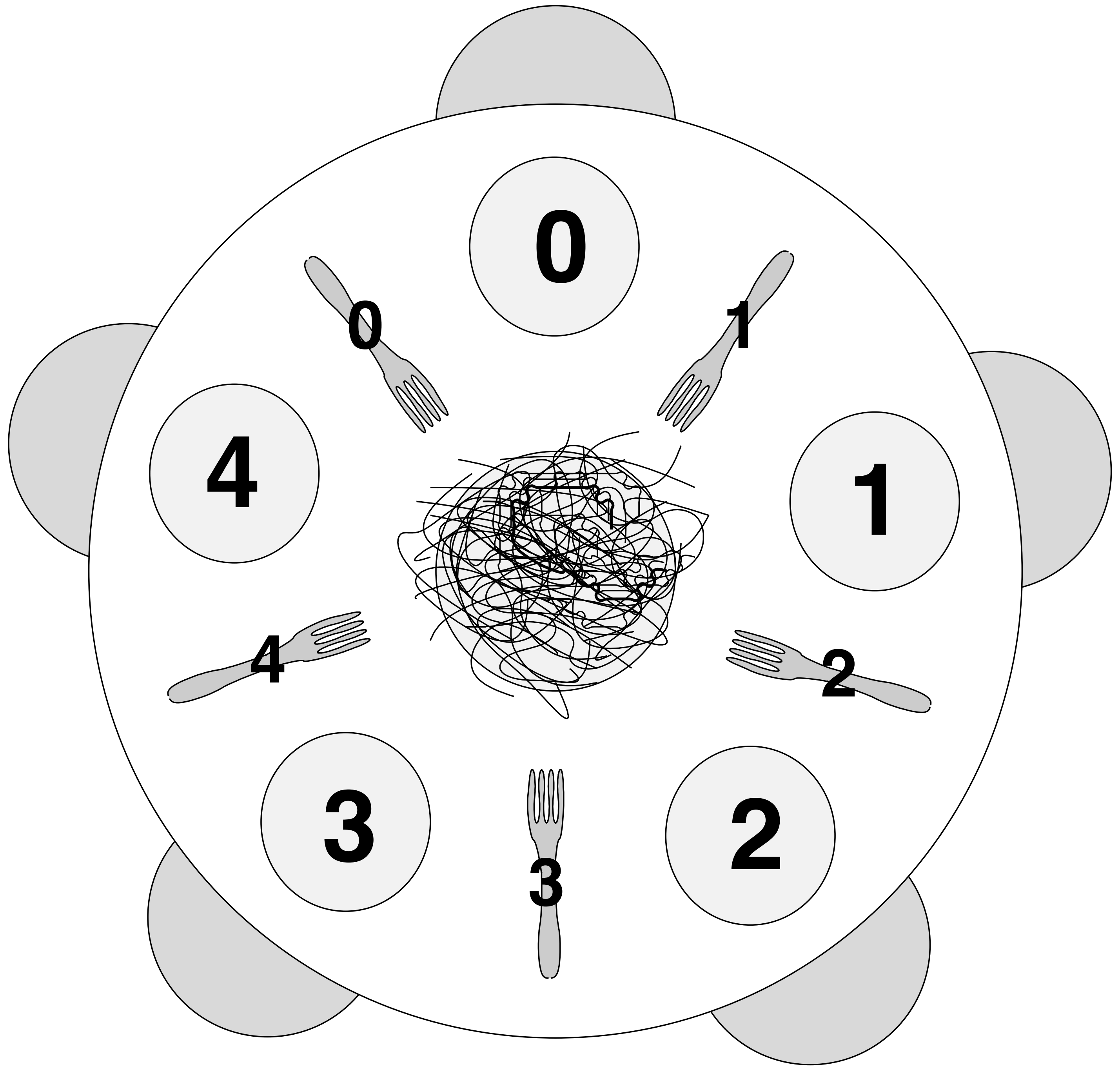
Contents

- The readers-writers problem
- **The dinning-philosophers problem**

Dining philosophers

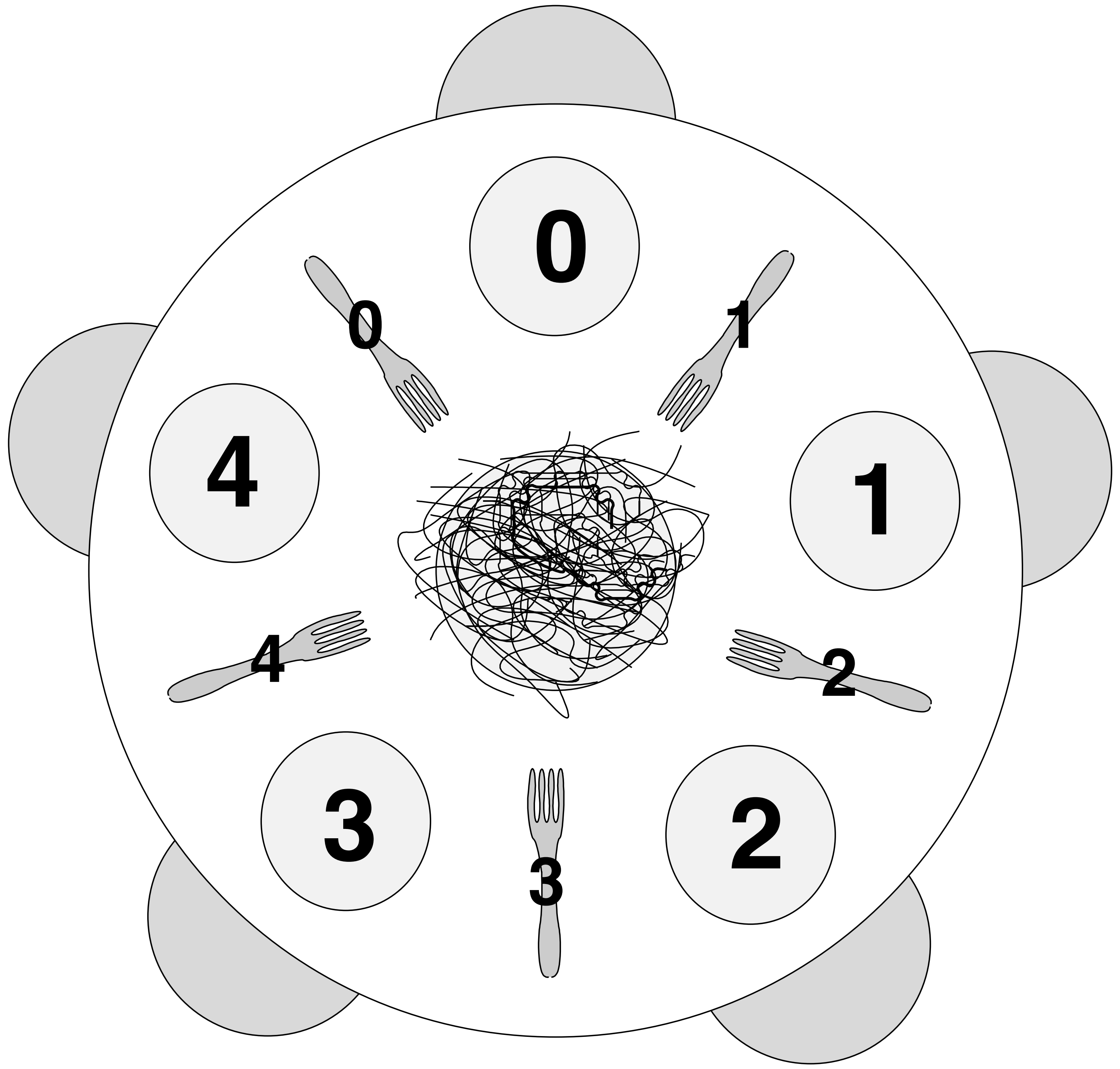
The Dining Philosophers Problem was proposed by Dijkstra in 1965. The standard version features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent interacting threads, come to the table and execute the following loop:

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



Dining philosophers

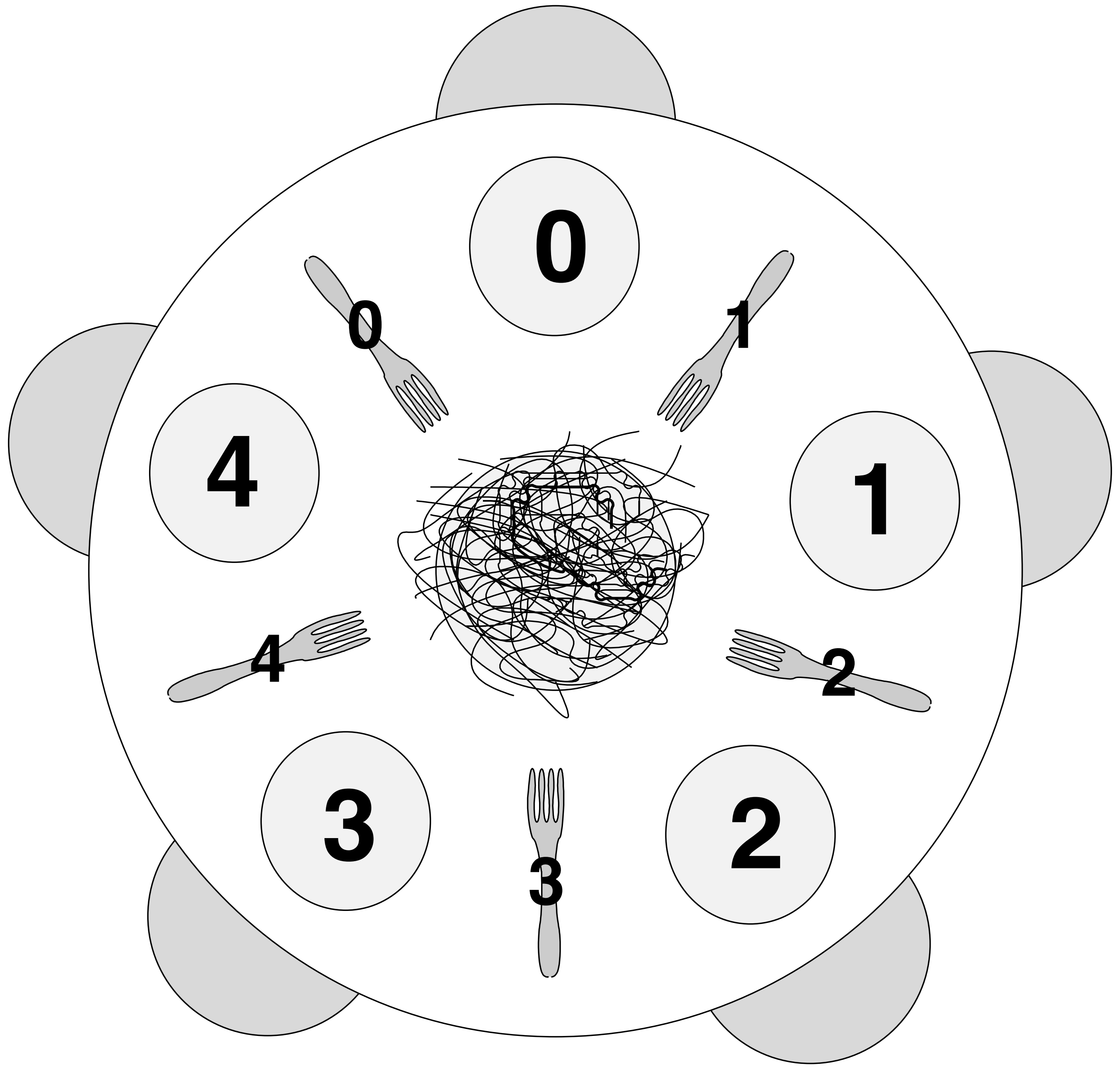
- The forks represent resources that the threads have to hold exclusively in order to make progress.
- The philosophers need *two* forks to eat, so a hungry philosopher might have to wait for a neighbor to put down a fork.



Dining philosophers

Assuming that the philosophers know how to `think` and `eat`, our job is to write a version of `get_forks` and `put_forks` that satisfies the following constraints:

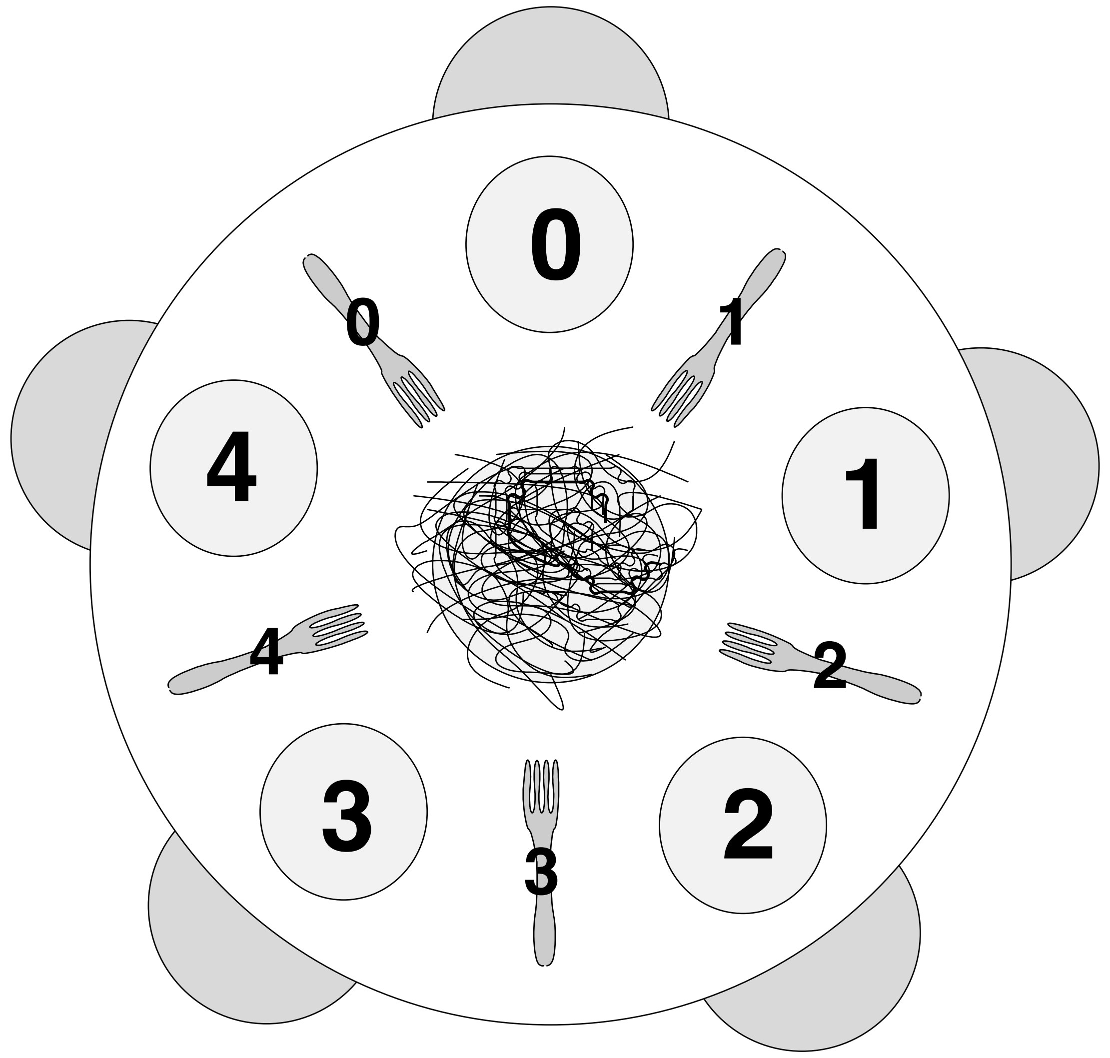
- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.



Dining philosophers

Let us define the functions `left` and `right` to refer to the forks' position.

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



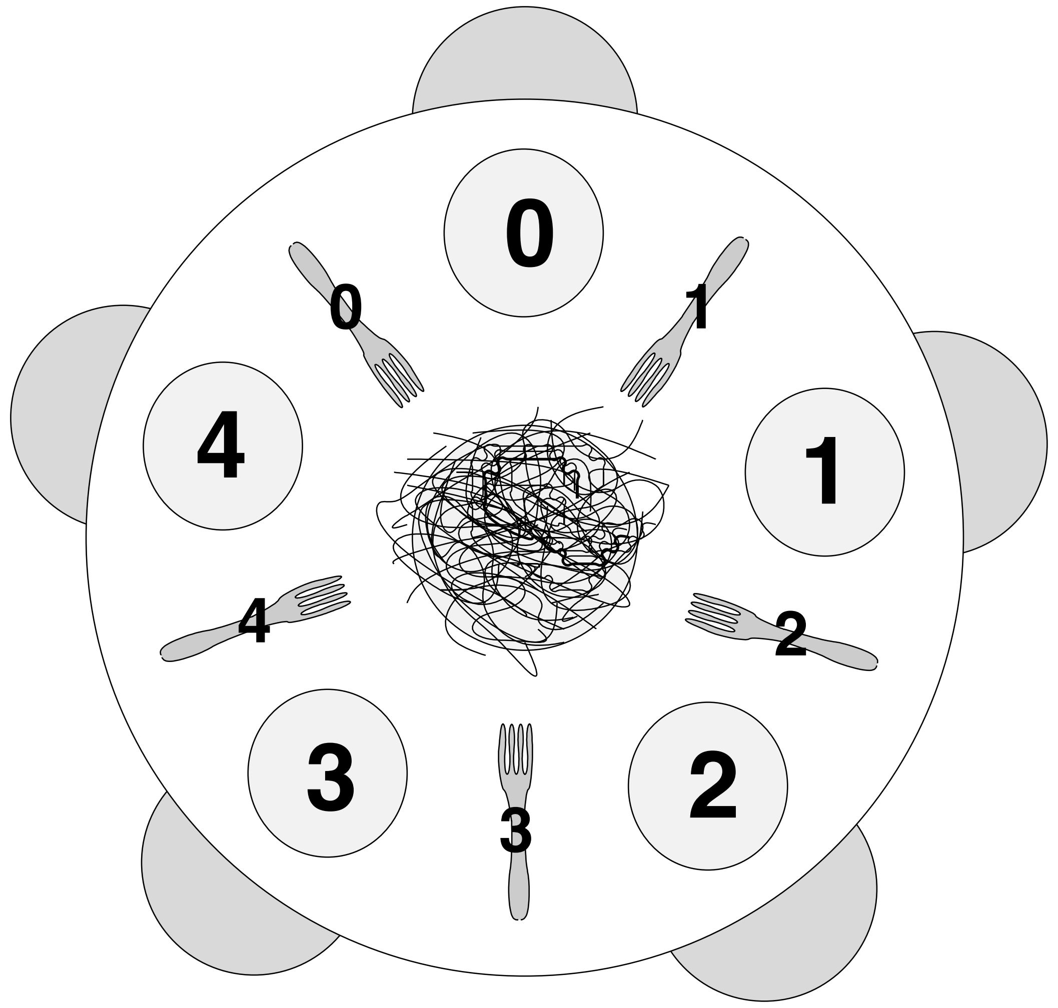
Use a list of Semaphores, one for each fork. Initially, all the forks are available.

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

Dining philosophers

First attempt of a 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()
```



Which constraints are satisfied by this solution?

1. Only one philosopher can hold a fork at a time.
2. It must be impossible for a deadlock to occur.
3. It must be impossible for a philosopher to starve waiting for a fork.
4. It must be possible for more than one philosopher to eat at the same time.

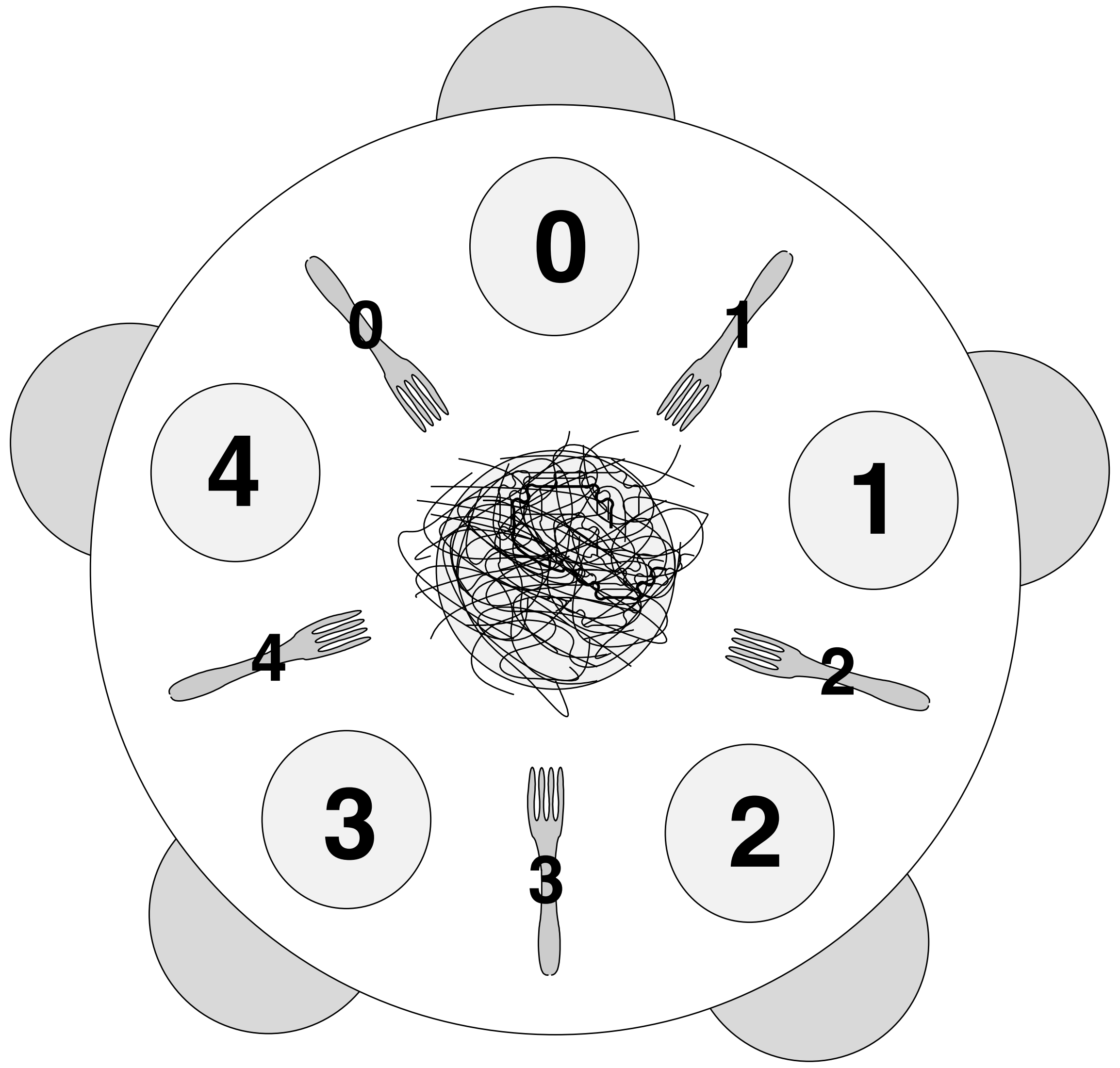
Dining philosophers

- **Limit the number of philosophers at the table at a time:** If only four philosophers are allowed at the table at a time, deadlock is impossible. There is always a fork on the table.

- We can control the number of philosophers at the table with a Multiplex named `footman` that is initialized to 4.

Dining philosophers

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

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



Which constraints are satisfied by this solution?

1. Only one philosopher can hold a fork at a time.
2. It must be impossible for a deadlock to occur.
3. It must be impossible for a philosopher to starve waiting for a fork.
4. It must be possible for more than one philosopher to eat at the same time.

Dining philosophers

- Another way to avoid deadlock is to change the order in which the philosophers pick up forks. In the original non-solution, the philosophers are “righties”; that is, they pick up the right fork first. But what happens if Philosopher 0 is a leftie?