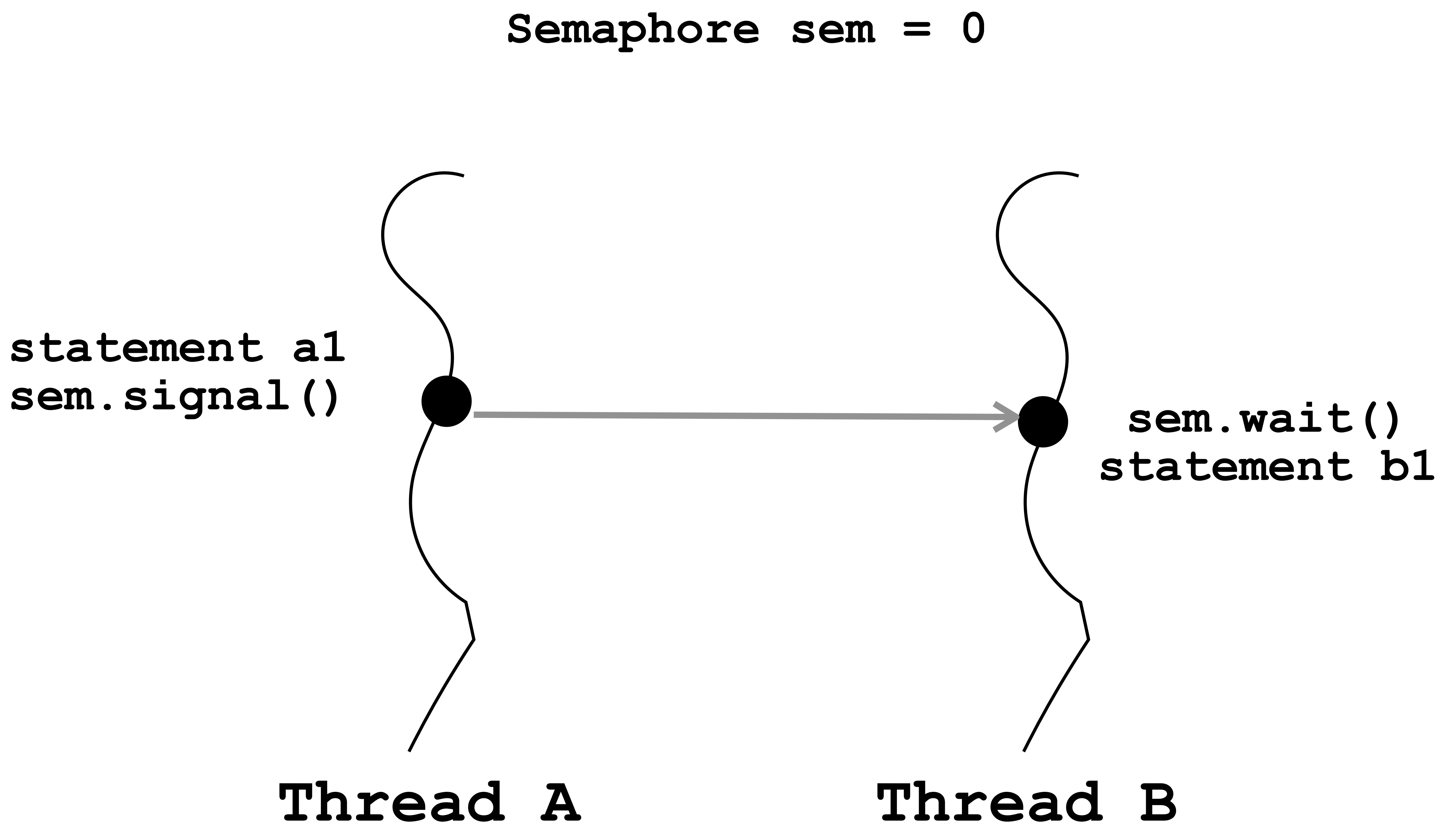
Synchronization Problems & Patterns

A Bunch of Semaphore Based Solutions

Signaling

- Simplest use of semaphores: one thread sends a signal to another thread
- Guarantee that a code segment in one thread runs before a code segment in the other thread



Rendezvous

- Generalize the signal pattern so that it works both ways
- Thread A waits for Thread B and vice versa
- Order of the threads don't matter
- A thread is not allowed to proceed until both have arrived at the rendezvous point

Thread A

```
statement a1
statement a2
```

Thread B

```
statement b1
statement b2
```

Hint: To use two semaphores

Rendezvous Solutions

Working solution

Thread A

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

Thread B

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

Non-working solution
Has deadlock

Thread A

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

Thread B

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

Mutual Exclusion

- Control concurrent access to shared variables

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

Hint: Use a single semaphore initialized to 1.
You already know how to do this one!

Exercise: Generalize mutual exclusion. Implement multiplex that allows n threads in the critical section.

Barrier Synchronization

- A generalization of Rendezvous for more than 2 threads
- Each thread runs the following code

```
rendezvous
critical point
```

- No thread executes the critical point until all threads execute rendezvous
- We assume total number of thread *n* is known ahead of time
- First *n-1* threads block waiting for the *n*-th thread

Hint: Keep thread count at rendezvous, unlock barrier after count has reached the required value.

Barrier Solutions

- One is a working solution and other is not!

```
rendezvous

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

if count == n: barrier.signal()

barrier.wait()

critical point
```

```
rendezvous

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

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

Turnstile Pattern

- wait and signal is used in rapid succession
- Initially the turnstile is locked, the *n*-th thread is unlocking it for all others

```
rendezvous

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

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

Another solution to Barrier with a defect

- This solution can cause a deadlock
- Look at the first thread: enters the critical section (takes the mutex)
- Enters the turnstile and blocks with the mutex taken
- No other thread can enter the critical section so count cannot increment

```
rendezvous

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

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

critical point
```

Reusable Barrier

- Barrier can be used in a loop
- After all threads have passed through the barrier we want to use the barrier with the next batch of threads - barrier needs to be put back to the initial state

Reusable Barrier: Non Solution

- An interrupt at Line 7 can make n-th and n-1-th thread to signal (instead of only the n-th thread signalling)

```
1  rendezvous
2
3  mutex.wait()
4      count += 1
5  mutex.signal()
6
7  if count == n: turnstile.signal()
8
9  turnstile.wait()
10 turnstile.signal()
11
12 critical point
13
14 mutex.wait()
15     count -= 1
16 mutex.signal()
17
18 if count == 0: turnstile.wait()
```

Reusable Barrier Solution

- Also known as the two-phase barrier - it forces the threads to wait twice
- Once at the entry to the critical point
- Again at the exit from the critical point

```
# 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()
```

Producer-Consumer Problems

- Producer-consumer with a finite buffer
- Producer-consumer with infinite buffer

Hint: You have seen both problems and their solutions

Readers-Writers Problem

- A common database is being updated by reading and writing threads
- Writers need exclusive access: no reader or no writer can access the database while a writer is accessing it
- Readers can access it concurrently
- First type - Readers have priority - no reader kept waiting unless writer has already started
- Second type - Writers have priority - writers when they are ready do the write as soon as possible

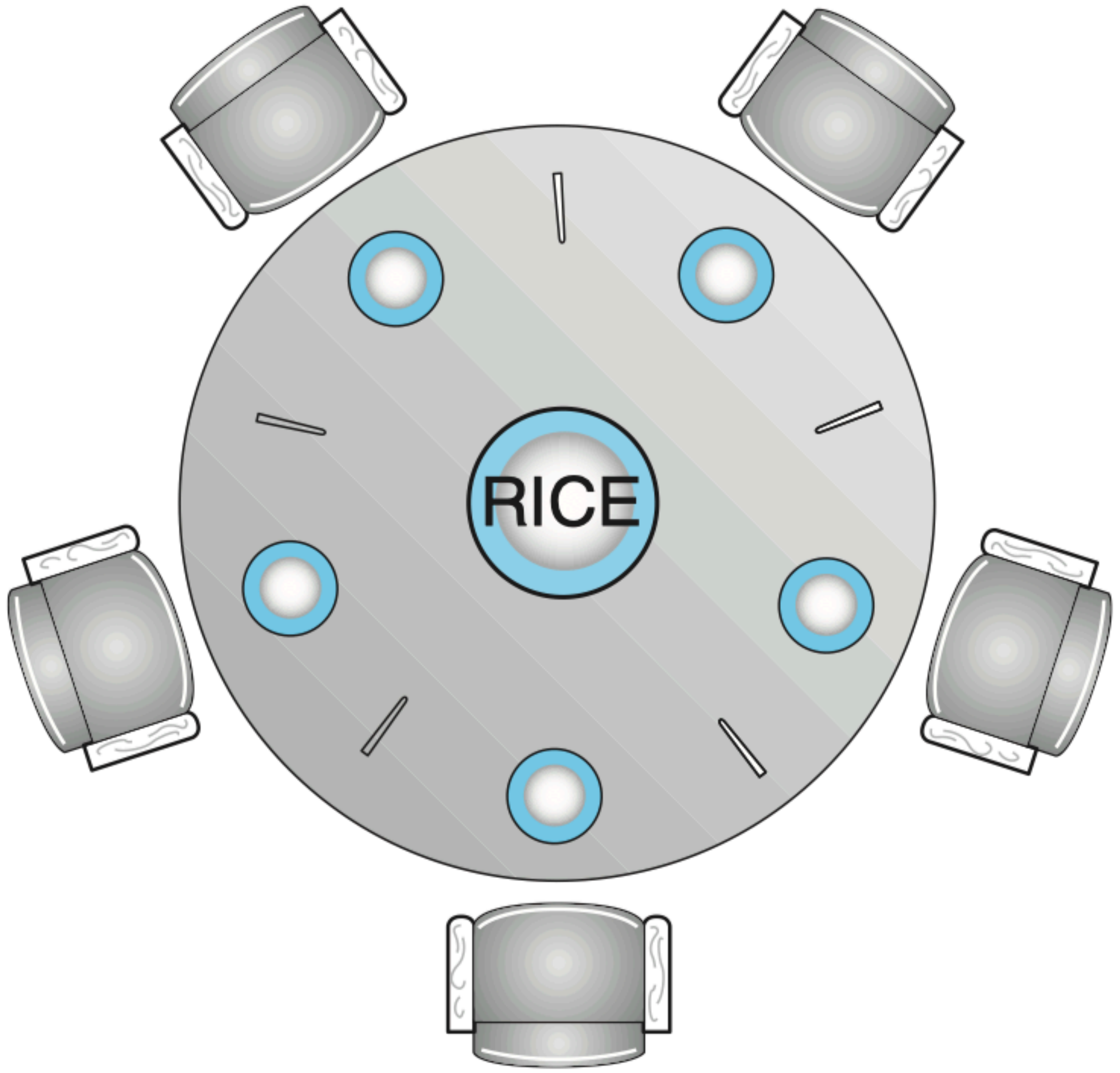
Solution to First Readers-Writers

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

Dining Philosophers Problem




```
semaphore chopstick[5];
```

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

Exercise: This has a deadlock, how to solve it?