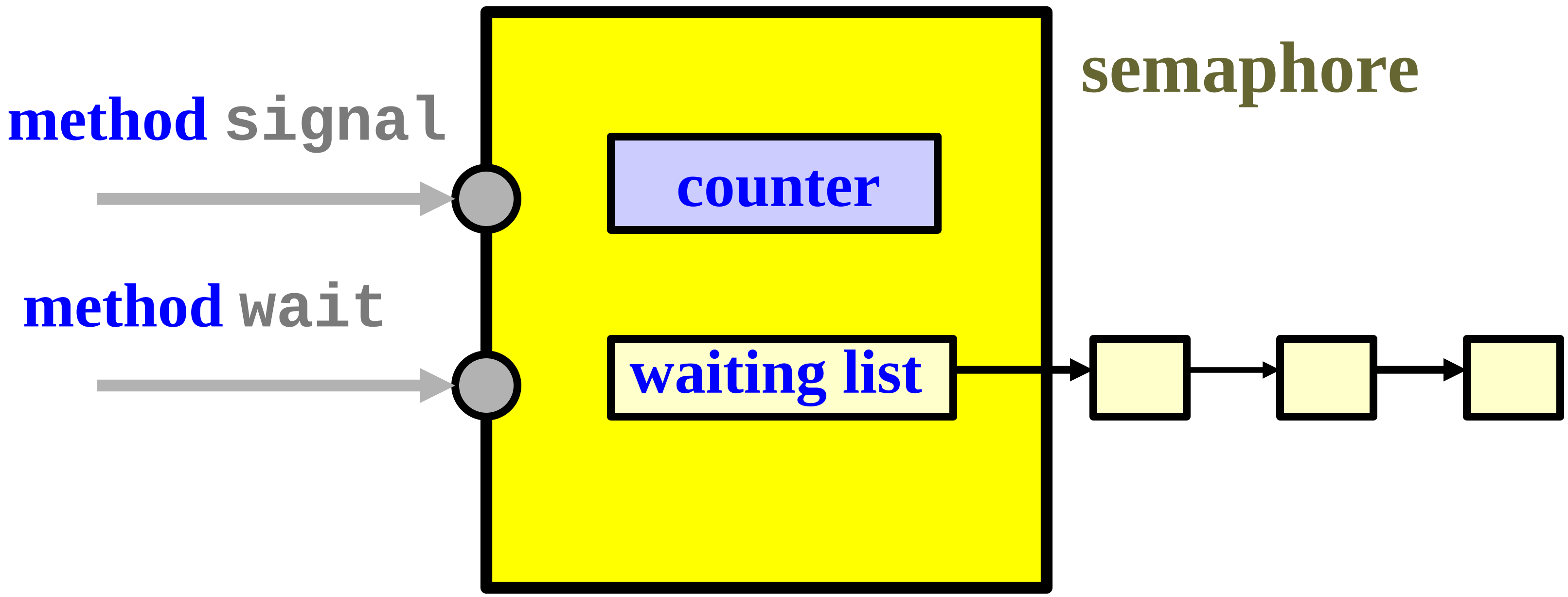
Semaphores

- A *semaphore* is an object that consists of a **counter**, a **waiting list** of processes and two **methods** (*e.g.*, functions): `signal` and `wait`.



Semaphore Method: wait

```
void wait(sem S)
{
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list;
        block();
    }
}
```

- After decreasing the counter by 1, if the counter value becomes negative, then
  - add the caller to the waiting list, and then
  - block itself.

Semaphore Method: signal

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume(P);
    }
}
```

- **After increasing the counter by 1, if the new counter value is not positive, then**
  - **remove a process P from the waiting list,**
  - **resume the execution of process P, and return**

Important Note: 1/4

```
S.count--;                      S.count++;
if (S.count<0) {                if (S.count<=0) {
    add to list;                    remove P;
    block();                        resume(P);
}                               }
```

- If `S.count < 0`, `abs(S.count)` is the number of waiting processes.
- This is because processes are added to (*resp*., removed from) the waiting list only if the counter value is `< 0` (*resp*., `<= 0`).

Important Note: 2/4

```
S.count--;                    S.count++;
if (S.count<0) {              if (S.count<=0) {
    add to list;                  remove P;
    block();                      resume(P);
}                             }
```

- The waiting list can be implemented with a queue if FIFO order is desired.
- However, the correctness of a program should not depend on a particular implementation of the waiting list.
- Your program should not make any assumption about the ordering of the waiting list.

Important Note: 3/4

```
S.count--;                  S.count++;
if (S.count<0) {            if (S.count<=0) {
    add to list;                remove P;
    block();                    resume(P);
}                           }
```

- The caller may be blocked in the call to `wait()`.
- The caller never blocks in the call to `signal()`. If `S.count > 0`, `signal()` returns and the caller continues. Otherwise, a waiting process is released and the caller continues. In this case, *two* processes continue.

The Most Important Note: 4/4

```
S.count--;                    S.count++;
if (S.count<0) {              if (S.count<=0) {
    add to list;                  remove P;
    block();                      resume(P);
}                             }
```

- `wait()` and `signal()` must be executed *atomically* (*i.e.*, as one uninterruptible unit).
- Otherwise, *race conditions* may occur.
- Homework: use execution sequences to show race conditions if `wait()` and/or `signal()` is not executed atomically.

Three Typical Uses of Semaphores

- **There are three typical uses of semaphores:**
  - **mutual exclusion:**

    Mutex (*i.e.*, *Mut*ual *Ex*clusion) locks
  - **count-down lock:**

    Keep in mind that semaphores have a counter.
  - **notification:**

    Indicate an event has occurred.

Use 1: Mutual Exclusion (Lock)

initialization is important

```
semaphore S = 1;
int       count = 0;
```

Process 1

```
while (1) {
   // do something
   S.wait();
     count++;
   S.signal();
   // do something
}
```

Process 2

```
while (1) {
   // do something
   S.wait();
     count--;
   S.signal();
   // do something
}
```

entry

critical sections

exit

- **What if the initial value of S is zero?**
- S **is a *binary semaphore* (similar to a *lock*).**

Use 2: Count-Down Counter

```
semaphore  S = 3;
```

Process 1	**Process 2**
`while (1) {`	`while (1) {`
`   // do something`	`   // do something`
`   S.wait();`	`   S.wait();`
at most 3 processes can be here!!!	
`   S.signal();`	`   S.signal();`
`   // do something`	`   // do something`
`}`	`}`

- **After three processes pass through `wait()`, this section is locked until a process calls `signal()`.**

Use 3: Notification

```
semaphore S1 = 1, S2 = 0;
```

process 1

```
while (1) {
   // do something
   S1.wait();
      cout << "1";
   S2.signal();
   // do something
}
```

process 2

```
while (1) {
   // do something
   S2.wait();
      cout << "2";
   S1.signal();
   // do something
}
```

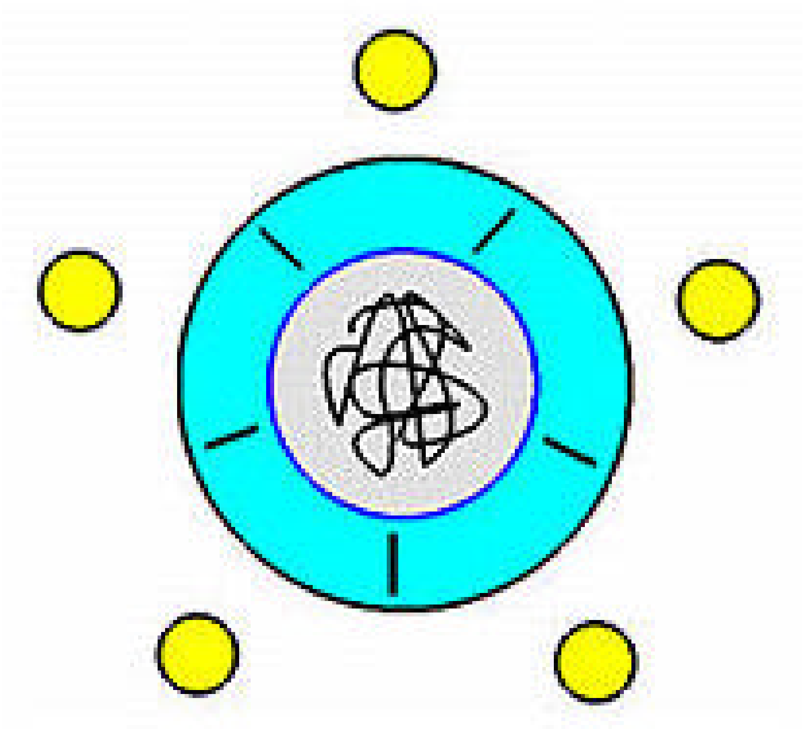
notify

notify

- Process 1 uses `S2.signal()` to notify process 2, indicating "I am done. Please go ahead."
- The output is `1 2 1 2 1 2` ......
- What if both `S1` and `S2` are both 0's or both 1's?
- What if `S1` = 0 and `S2` = 1?

Lock Example: Dining Philosophers

- **Five philosophers are in a *thinking* - *eating* cycle.**
- **When a philosopher gets hungry, he sits down, picks up *two nearest* chopsticks, and eats.**
- **A philosopher can eat only if he has *both* chopsticks.**
- **After eating, he puts down both chopsticks and thinks.**
- **This cycle continues.**



Dining Philosopher: Ideas

- Chopsticks are shared items (by two philosophers) and must be protected.
- Each chopstick has a semaphore with initial value 1.
- A philosopher calls `wait()` before picks up a chopstick and calls `signal()` to release it.

outer critical section

left chop locked

```
Semaphore C[5] = 1;

C[i].wait();
C[(i+1)%5].wait();
```

has 2 chops and eats

```
C[(i+1)%5].signal();
C[i].signal();
```

inner critical section

right chop locked

Dining Philosophers: Code

```
semaphore  C[5] = 1;
```

philosopher *i*

```
while (1) {
   // thinking
   C[i].wait();
   C[(i+1)%5].wait();
   // eating
   C[(i+1)%5].signal();
   C[i].signal();
   // finishes eating
}
```

wait for my left chop

wait for my right chop

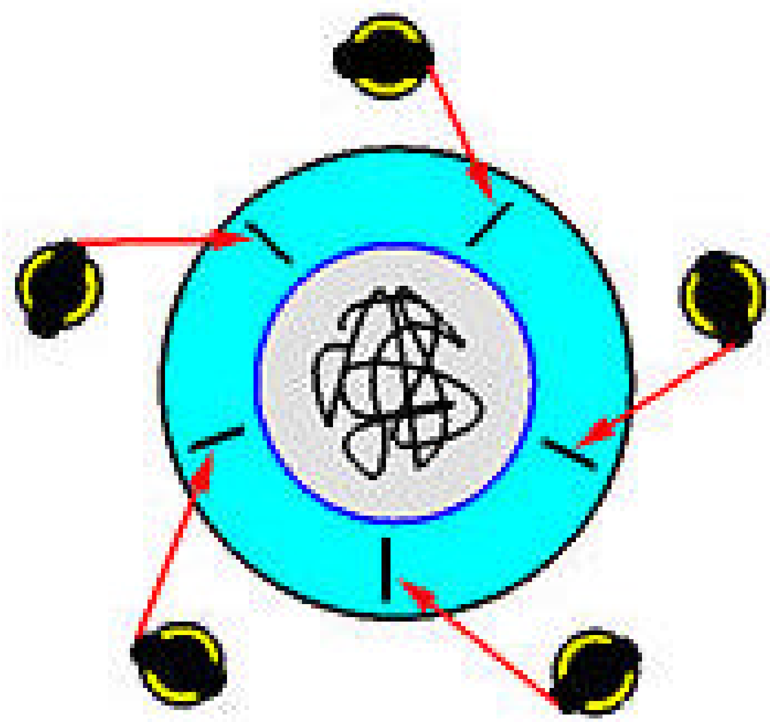
release my right chop

release my left chop

Does this solution work?

Dining Philosophers: Deadlock!

- If all five philosophers sit down and pick up their left chopsticks at the same time, this program has a ***circular waiting*** and deadlocks.
- An easy way to remove this deadlock is to introduce a weirdo who picks up his **right** chopstick first!



Dining Philosophers: A Better Idea

```
semaphore C[5] = 1;
```

philosopher *i* (0, 1, 2, 3)

```
while (1) {
   // thinking
   C[i].wait();
   C[(i+1)%5].wait();
   // eating
   C[(i+1)%5].signal();
   C[i].signal();
   // finishes eating;
}
```

Philosopher 4: the weirdo

```
while (1) {
   // thinking
   C[(i+1)%5].wait();
   C[i].wait();
   // eating
   C[i].signal();
   C[(i+1)%5].signal();
   // finishes eating
}
```

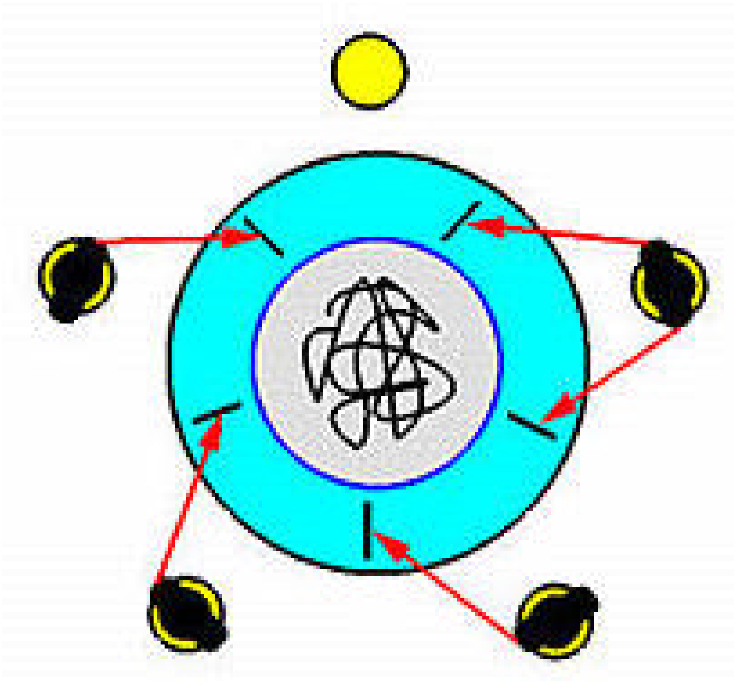
lock left chop

lock right chop

Dining Philosophers: Questions

- **The following are some important questions for you to work on.**
  - **We choose philosopher 4 to be the weirdo. Does this choice matter?**
  - **Show that this solution does not cause *circular waiting*.**
  - **Show that this solution will not have *circular waiting* if we have more than 1 and less than 5 weirdoes.**
- **These questions may appear as exam problems.**

Count-Down Lock Example



- The naïve solution to the dining philosophers causes circular waiting.
- If only four philosophers are allowed to sit down, no deadlock can occur.
- Why? If all four of them sit down at the same time, the right-most philosopher can have both chopsticks!
- How about fewer than four? This is obvious.

Count-Down Lock Example

```
semaphore C[5]= 1;
semaphore Chair = 4;

while (1) {
   // thinking
   Chair.wait();
      C[i].wait();
      C[(i+1)%5].wait();
      // eating
      C[(i+1)%5].signal();
      C[i].signal();
   Chair.signal();
}
```

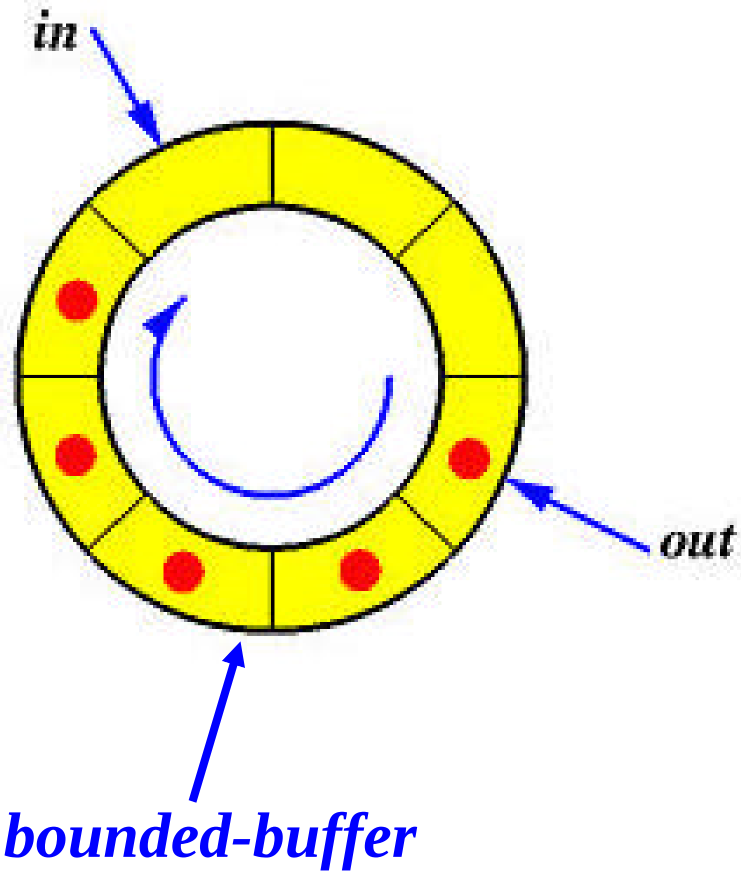
get a chair

this is a count-down lock that only allows 4 to go!

this is our old friend

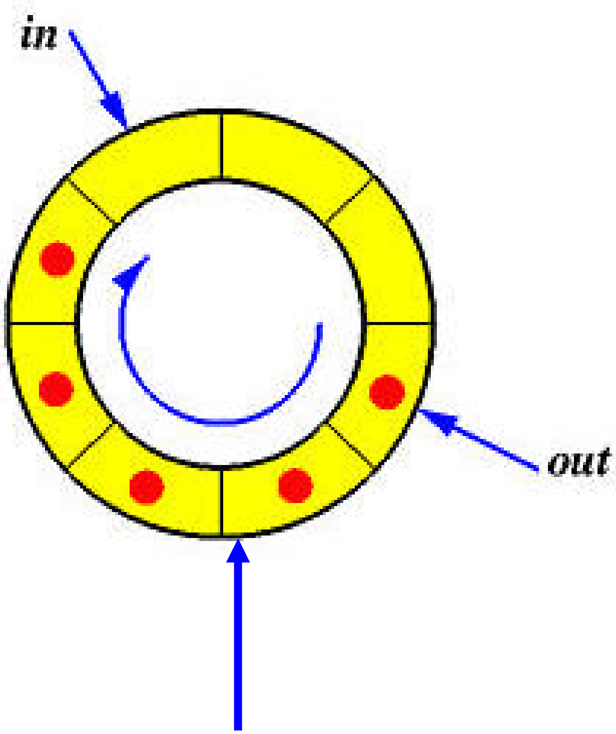
release my chair

The Producer/Consumer Problem



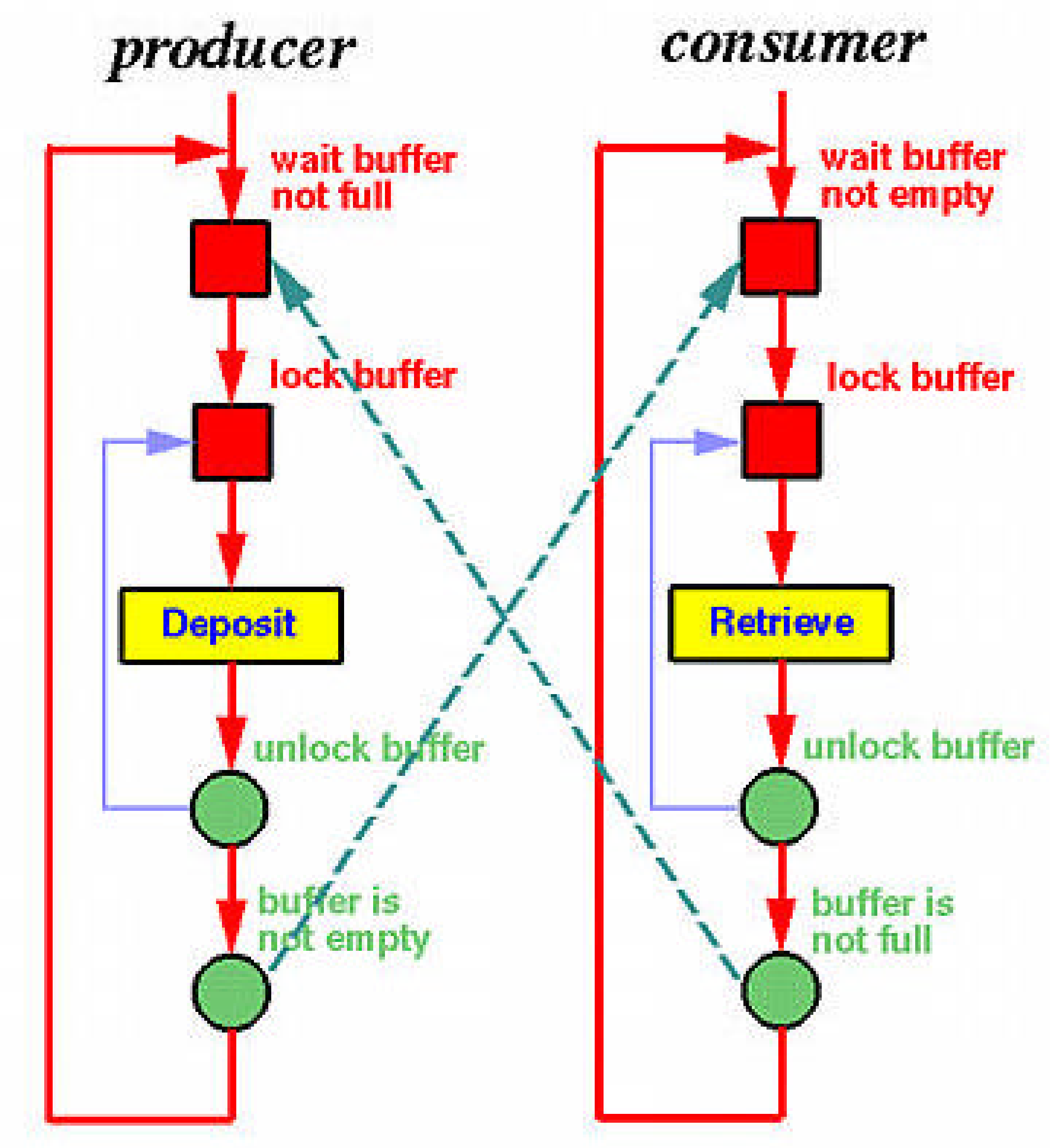
- **Suppose we have a circular buffer of *n* slots.**
- **Pointers *in* (*resp*., *out*) points to the first empty (*resp*., filled) slot.**
- **Producer processes keep adding info into the buffer**
- **Consumer processes keep retrieving info from the buffer.**

Problem Analysis



buffer is implemented with an array **`Buf[ ]`**

- **A producer deposits info into `Buf[in]` and a consumer retrieves info from `Buf[out]`.**
- **`in` and `out` must be advanced.**
- **`in` is shared among producers.**
- **`out` is shared among consumers.**
- **If `Buf` is full, producers should be blocked.**
- **If `Buf` is empty, consumers should be blocked.**



- **We need a sem. to protect the buffer.**
- **A second sem. to block producers if the buffer is full.**
- **A third sem. to block consumers if the buffer is empty.**

Solution

no. of slots

```
semaphore NotFull=n, NotEmpty=0, Mutex=1;
```

producer

```
while (1) {
  NotFull.wait();
    Mutex.wait();
      Buf[in] = x;
      in = (in+1)%n;
    Mutex.signal();
  NotEmpty.signal();
}
```

consumer

```
while (1) {
  NotEmpty.wait();
    Mutex.wait();
      x = Buf[out];
      out = (out+1)%n;
    Mutex.signal();
  NotFull.signal();
}
```

notifications

critical section

Question

- What if the producer code is modified as follows?
- **Answer:** a deadlock may occur. Why?

order changed

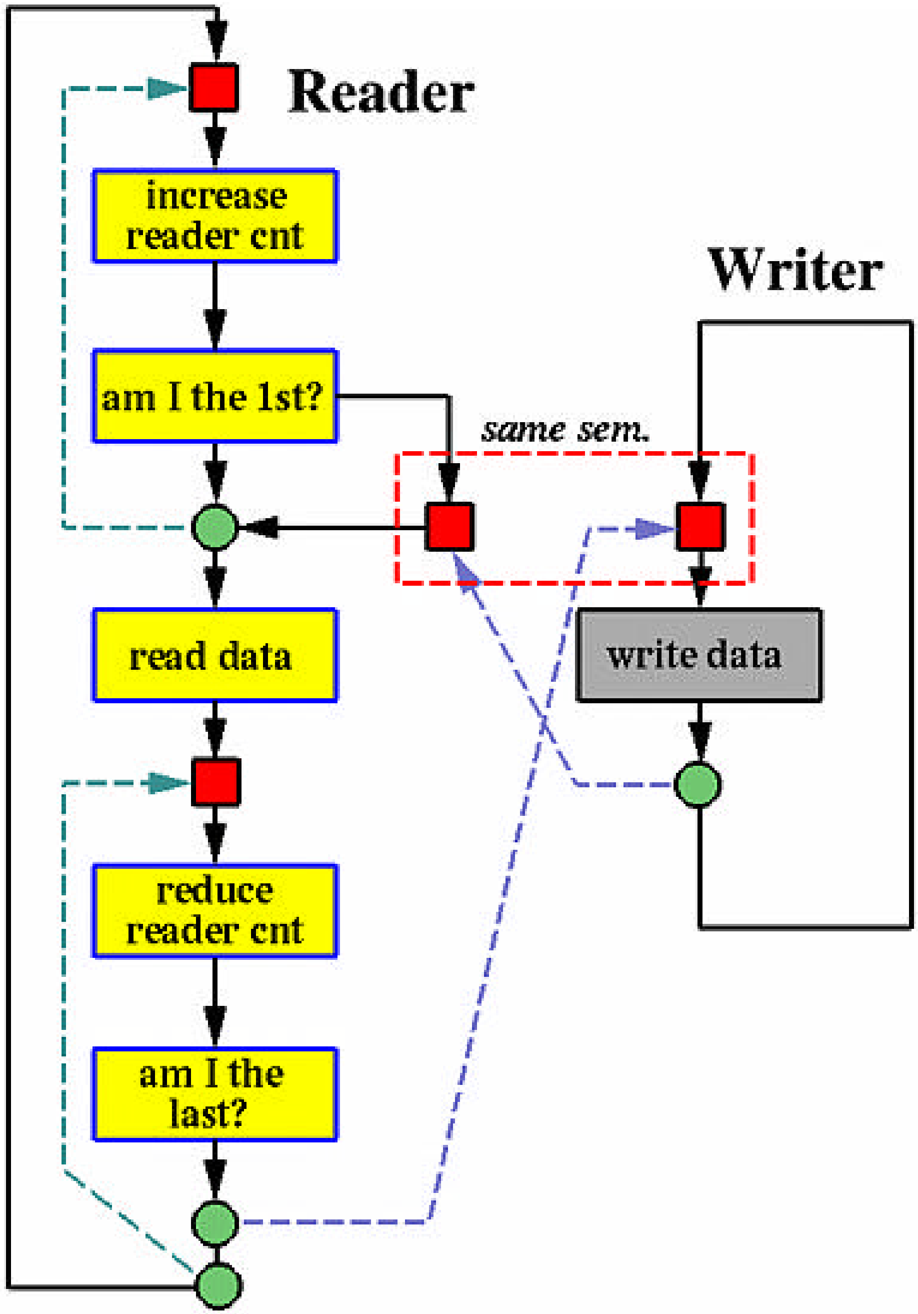
```
while (1) {
  Mutex.wait();
    NotFull.wait();
      Buf[in] = x;
      in = (in+1)%n;
    NotEmpty.signal();
  Mutex.signal();
}
```

The Readers/Writers Problem

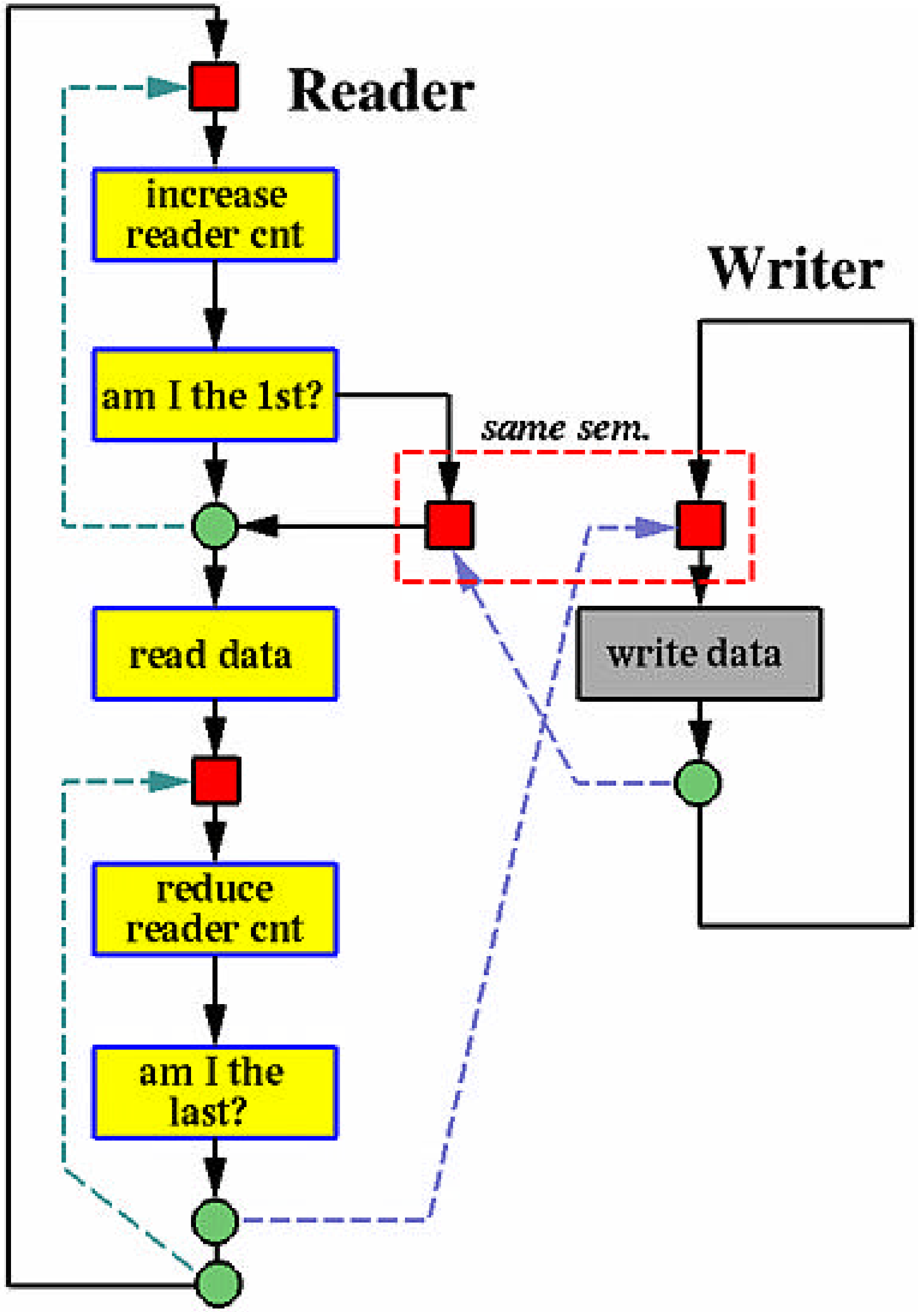
- **Two groups of processes, readers and writers, are accessing a shared resource by the following rules:**
  - **Readers can read simultaneously.**
  - **Only one writer can write at any time.**
  - **When a writer is writing, no reader can read.**
  - **If there is any reader reading, all incoming writers must wait. Thus, readers have higher priority.**

Problem Analysis

- **We need a semaphore to block readers if a writer is writing.**
- **When a writer arrives, it must be able to know if there are readers reading. So, a reader count is required which must be protected by a lock.**
- **This reader-priority version has a problem: bounded waiting condition may be violated if readers keep coming, causing the waiting writers no chance to write.**



- When a reader comes in, it increase the count.
- If it is the 1st reader, waits until no writer is writing,
- Reads data.
- Decreases the counter.
- Notifies the writer that no reader is reading if it is the last.



- **When a writer comes in, it waits until no reader is reading and no writer is writing.**
- **Then, it writes data.**
- **Finally, notifies readers and writers that no writer is in.**

Solution

```
semaphore Mutex = 1, WrtMutex = 1;
int       RdrCount;
```

reader

```
while (1) {
  Mutex.wait();
    RdrCount++;
    if (RdrCount == 1)
      WrtMutex.wait();
  Mutex.signal();
  // read data
  Mutex.wait();
    RdrCount--;
    if (RdrCount == 0)
      WrtMutex.signal();
  Mutex.signal();
}
```

blocks both readers and writers

writer

```
while (1) {
  WrtMutex.wait();
  // write data
  WrtMutex.signal();
}
```