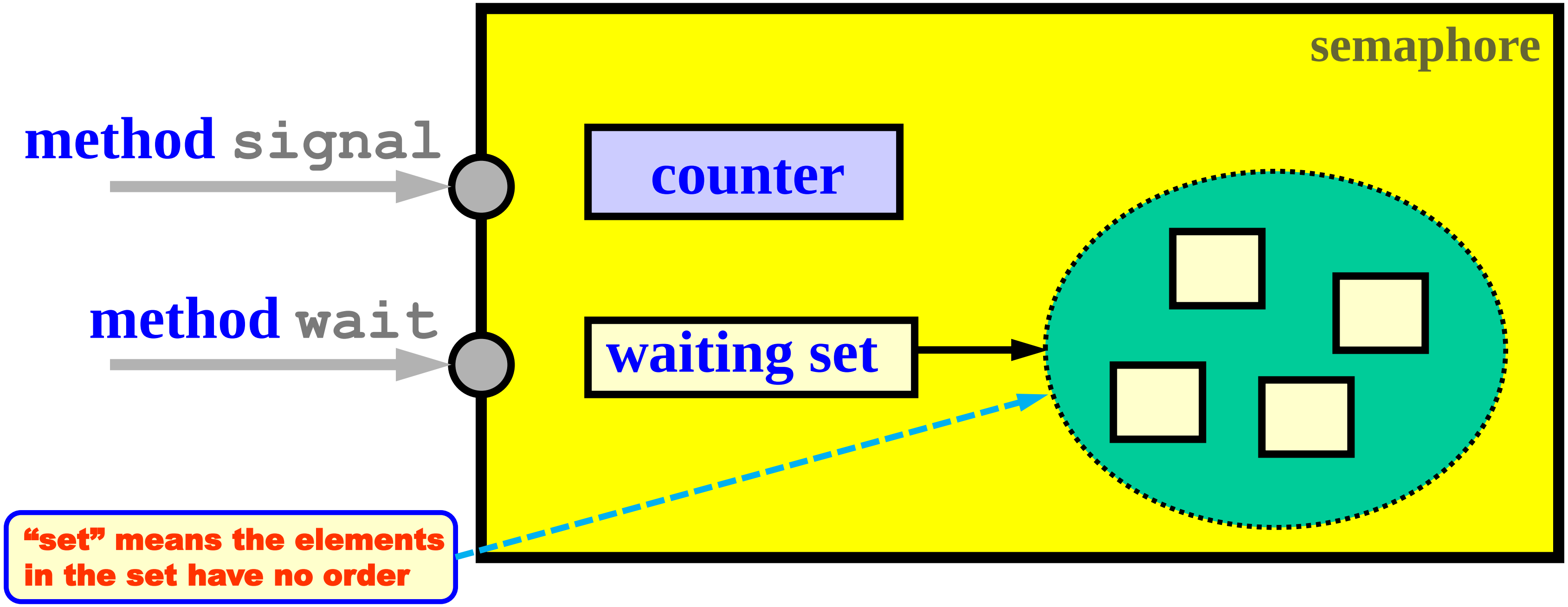
Part III
Synchronization
Semaphores

The bearing of a child takes nine months,
no matter how many women are assigned.

Frederick P. Brooks Jr.

Semaphores

- A ***semaphore*** is an object that consists of a private **counter**, a private **waiting set** of threads, and two public **methods** (e.g., member functions): `signal` and `wait`.



Semaphore Method: wait

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

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

Semaphore Method: signal

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

- **After increasing the counter by 1, if the new value is not positive (e.g., non-negative), then**
  - **remove a thread T from the waiting set,**
  - **resume the execution of thread T, and return**

Practice Example: 1/8

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

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

- **Suppose we have four threads, A, B, C and D.**
- **We also have a semaphore `S` with initial value 2.**
- **What would happen if A, B, C and D calls `wait(S)` in this order?**

Practice Example: 2/8

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

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

A	B	C	D	semaphore counter	semaphore waiting set
				2	∅
wait(S)					
	wait(S)				
		wait(S)			
			wait(S)		

initial case

Practice Example: 3/8

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

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

A	B	C	D	semaphore counter	semaphore waiting set
				2	∅
wait(S)				1	∅
	wait(S)				
		wait(S)			
			wait(S)		

A calls wait(S) **and gets through (no wait)**

Practice Example: 4/8

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

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

A	B	C	D	semaphore counter	semaphore waiting set
				2	∅
wait(S)				1	∅
	wait(S)			0	∅
		wait(S)			
			wait(S)		

B calls `wait(S)` **and gets through (no wait)**

Practice Example: 5/8

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

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

A	B	C	D	semaphore counter	semaphore waiting set
				2	∅
wait(S)				1	∅
	wait(S)			0	∅
		wait(S) (blocked)		-1	C
			wait(S)		

C calls wait(S) and is blocked

Practice Example: 6/8

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

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

A	B	C	D	semaphore counter	semaphore waiting set
				2	∅
wait(S)				1	∅
	wait(S)			0	∅
		wait(S) **(blocked)**		-1	C
			wait(S) **(blocked)**	-2	C and D

D calls `wait(S)` **and is also blocked**

Practice Example: 7/8

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

```
void signal(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a thread T from the waiting set;
        resume(T);
    }
}
```

- **What would happen if A calls `signal(S)` and then B calls `wait(S)`?**

Practice Example: 8/8

`void wait(sem S)`
`{`
`    S.count--;`
`    if (S.count < 0) {`
`        add the caller to the waiting set;`
`        block();`
`    }`
`}`
	`void signal(sem S)`
	`{`
	`    S.count++;`
	`    if (S.count <= 0) {`
	`        remove a thread T from the waiting set;`
	`        resume(T);`
	`    }`
	`}`

A	B	C	D	semaphore counter	semaphore waiting set
		`wait(S)`	`wait(S)`	-2	C and D
`signal(S)`			D released	-1	C
	`wait(S)` (blocked)			-2	B and C

- **Which one of C and D is released when A calls `signal(S)`?**
- **We don't know, because there is no ordering in the waiting set.**
- **Let us say D is lucky and released.**
- **B is blocked.**

Important Note: 1/4

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

- **If** `S.count < 0`**,** `abs(S.count)` **is the number of waiting threads.**
- **This is because threads are added to (*resp*., removed from) the waiting set 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 set;                   remove T;
    block();                      resume(T);
}                             }
```

- The waiting set can be implemented with a queue if FIFO order is desired.
- However, **the correctness of a program should not depend on a particular implementation (e.g., ordering) of the waiting set.**

Important Note: 3/4

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

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

The Most Important Note: 4/4

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

- `wait()` and `signal()` with respect to the same semaphore must be executed **atomically**. Otherwise, **race conditions** may occur.
- **Homework:** use execution sequences to show race conditions if `wait()` and/or `signal()` is not executed atomically. Also show that mutual exclusion cannot be guaranteed.

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 a semaphore has a private counter that can count.
  - **notification:**

    Wait for an event to occur and indicate an event has occurred.

Use 1: Mutual Exclusion (Lock)

initialization is important

```
semaphore S = 1;
int       count = 0;  // shared variable
```

Thread 1		Thread 2
`while (1) {`		`while (1) {`
`// do something`	entry	`// do something`
`S.wait();`		`S.wait();`
`count++;`	critical sections	`count--;`
`S.signal();`		`S.signal();`
`// do something`	exit	`// do something`
`}`		`}`

- What if the initial value of `S` is zero?
- `S` is a *binary semaphore* (`count` being 0/`FALSE` or 1/`TRUE`).

Discussion: 1/2

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

Thread 1	Thread 2
`while (1) {`	`while (1) {`
// do something	
`S.wait();`	`S.wait();`
`count++;`	`count--;`
`S.signal();`	`S.signal();`
// do something	
`}`	`}`

- **Note that the execution of `S.wait()` and `S.signal()` must be atomic. When multiple calls to `wait()` and `signal()` at the same time, the execution of these calls is sequential.**
- **Therefore, these two `S.wait()` calls cannot be executed at the same time. One will happen before the other.**

Discussion: 2/2

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

Thread 1

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

Thread 2

```
while (1) {

   S.wait();
      count--;
   S.signal();

}
```

- **Mutual Exclusion**: Because only 1 thread can call `wait()` at a time, if thread 1 gets into the critical section, the semaphore counter of `S` is 0. The next thread that calls `wait()` blocks. Thus, mutual exclusion holds!
- **Progress**: Suppose the critical section is empty and threads 1 and 2 are waiting to enter. Then, the system selects one thread to call `wait()`. The execution of `wait()` only needs finite instructions. Both take finite time!
- **Outsider Issue**? **Bounded Waiting**?

Use 2: Count-Down Counter

```
semaphore  S = 3;
```

Thread 1

```
while (1) {
   // do something
   S.wait();
```

at most 3 threads can be here!!!

```
   S.signal();
   // do something
}
```

Thread 2

```
while (1) {
   // do something
   S.wait();
```

at most 3 threads can be here!!!

```
   S.signal();
   // do something
}
```

- **After three threads passing through `wait()`, this section is locked until a thread calls `signal()`.**

Use 3: Notification

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

<u>Thread 1</u>

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

<u>Thread 2</u>

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

notify notify notify

- **Thread 1 uses `S2.signal()` to notify thread 2, indicating "I am done. Please go ahead."**
- **Thread 2 uses `S1.signal()` to notify thread 1, indicating "I am done. Please go ahead."**
- **The output is `1 2 1 2 1 2` ……**

Discussion: 1/10

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

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

- If T_2 reaches `S2.wait()` **before** T_1 reaches `S2.signal()`.

T_1	T_2	S1	S2	Comment
		1	0	Initial Values
`S1.wait()`		0	0	T_1 continues
	`S2.wait()`	0	-1	T_2 blocks
`cout<<"1"`		0	-1	
`S2.signal()`		0	0	T_1 signals T_2
`S1.wait()`	`cout<<"2"`	-1	0	T_2 is released & prints
	`S1.signal()`	0	0	T_2 signals T_1
`cout<<"1"`	`S2.wait()`	0	-1	T_2 waits

Discussion: 2/10

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

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

- If T_2 reaches `S2.wait()` **after** T_1 reaches `S2.signal()`.

T_1	T_2	S1	S2	Comment
		1	0	Initial Values
`S1.wait()`		0	0	
`cout<<"1"`		0	0	
`S2.signal()`		0	1	T_1 signals
	`S2.wait()`	0	0	T_2 continues
`S1.wait()`	`cout<<"2"`	-1	0	T_1 blocks
	`S1.signal()`	0	0	T_2 signals T_1
`cout<<"1"`	`S2.wait()`	0	-1	T_2 waits

Discussion: 3/10

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

Thread 1

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

Thread 2

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

- **What if the initial values are changed to `S1 = 0` and `S2 = 1`?**

Discussion: 4/10

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

Thread 1

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

Thread 2

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

- **What if the initial values are changed to `S1 = 0` and `S2 = 1`?**
- **Because the code for Thread 1 and Thread 2 are symmetric, the output is `2 1 2 1 2 1 ……` !**

Discussion: 5/10

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

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

- **What if the initial values are changed to `S1 = 0` and `S2 = 0`?**

Discussion: 6/10

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

Thread 1

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

Thread 2

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

- **What if the initial values are changed to `S1 = 0` and `S2 = 0`?**
- **Deadlock!**
  - **Both threads are blocked by their `wait()` calls, waiting for the other thread to call `signal()`.**
  - **Each of these two threads are blocked by an event that can only be caused to happen by the other (waiting) thread.**

Discussion: 7/10

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

Thread 1

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

Thread 2

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

- **What if the initial values are changed to `S1 = 1` and `S2 = 1`?**

Discussion: 8/10

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

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

- **What if the initial values are changed to `S1 = 1` and `S2 = 1`?**
  - **Because semaphores `S1` and `S2` have initial value 1, both threads can pass their `wait()` calls in an unpredictable order.**
  - **As a result, the order of printing `1` or `2` is also not predictable.**
  - **The output from Thread 1 and Thread 2 is kind of random.**

Discussion: 9/10

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

Thread 1

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

Thread 2

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

- **What if the initial values are changed to `S1 = 1` and `S2 = 1`?**
    - **It is easy to get two consecutive `1`'s (or `2`'s)**
    - **REASON: If Thread 1 calls `S2.signal()` before Thread 2 reaches its `S2.wait()`, the counter of semaphore `S2` is 2.**
    - **Therefore, Thread 2 can print two consecutive `2`'s in its output.**
    - **Because the code is symmetric, Thread 1 can print two consecutive `1`'s in its output.**

Discussion: 10/10

T_1	T_2	S1	S2	Comment
		1	1	
`S1.wait()`		0	1	T_1 gets through
`cout<<"1"`		0	1	
`S2.signal()`		0	2	T_1 signals T_2
	`S2.wait()`	0	1	T_2 does not wait
	`cout<<"2"`	0	1	
	`S1.signal()`	1	1	T_2 signals T_1
	`S2.wait()`	1	0	T_2 does not wait
	`cout<<"2"`	1	0	
	`S1.signal()`	2	0	T_2 signals T_1
	`S2.wait()`	2	-1	T_2 must wait
`S1.wait()`		1	-1	T_1 does not wait
`cout<<"1"`		1	-1	
`S2.signal()`		1	0	T_1 signals T_2
	T_2 released			

output = …1 2 2 1 …

Food for Thought

- **Is it possible that Thread 1 and Thread 2 can print 3 consecutive `1`'s or `2`'s such as …`1 2 2 2 1`…?**
- **Suppose Thread *i* can only print the value of *i*. Modify the sample code by adding semaphores and threads to do the following:**
  - **Print** `1 2 2 1 2 2 1 2 2 1 …`
  - **Print** `1 2 3 1 2 3 1 2 3 …`
  - **Print** `1 2 3 2 1 2 3 2 1 …`
  - **Print** `1 2 3 2 3 1 2 3 2 3 1 …`

Dining Philosophers

Dining Philosophers: Its Origin

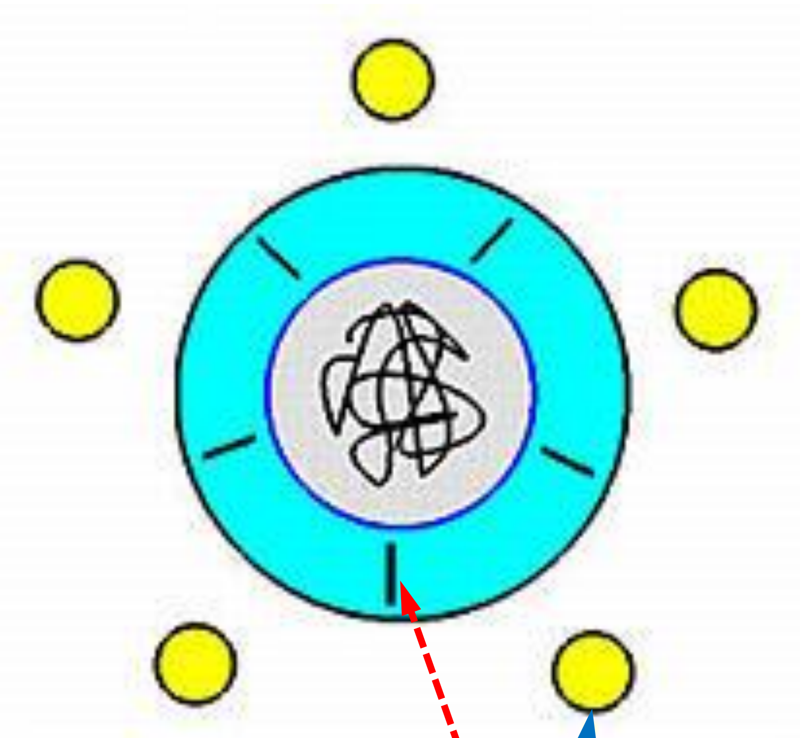
- **The Dining Philosophers problem, originally called Dining Quintuple – later dubbed "The Dining Philosophers" by Tony Hoare – was a problem in the final exam of a course taught by Edsger W Dijkstra.**

EWD1000-2

EWD123, "Cooperating Sequential Processes" was written in 1965 and served as lecture notes for my course in the fall semester of that year. (The problem of The Dining Quintuple - later dubbed "The Dining Philosophers" by Tony Hoare - was the examination problem at the end of that semester.) The rate with which the EWD-numbers increased was in those days not a measure of my productivity: I assigned numbers when I started on manu-scripts and many of them were not completed.

Dining Philosophers

- **Five philosophers are in a *thinking* - *eating* cycle.**
- **When a philosopher gets hungry, he sits down, picks up *his left* and then *his right* 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.**



There are five chairs and five chopsticks

Dining Philosopher: Ideas

- **Each philosopher is a thread.**
- **Chopsticks are shared items (by two neighboring philosophers) and must be protected.**
- **Each chopstick must be used in a mutually exclusive way (i.e., used in the critical section of that chopstick).**
- **Each chopstick has a semaphore with initial value 1 for mutual exclusion.**
- **A philosopher calls `wait()` to pick up a chopstick and calls `signal()` to release it.**

Dining Philosophers: Code

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

left chop critical section

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

right chop critical section

wait for my right chop

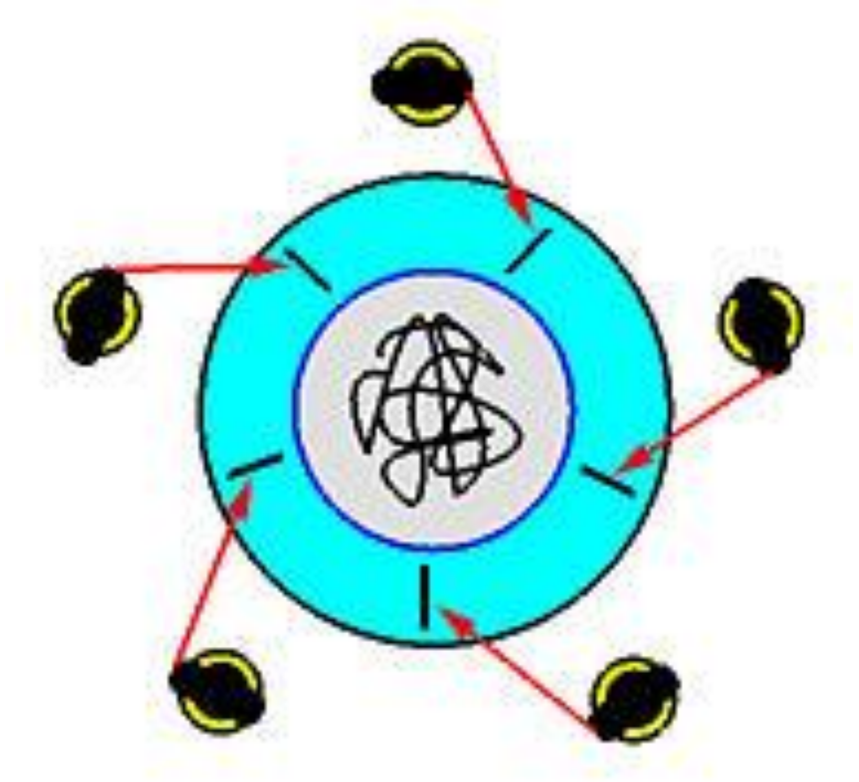
release my right chop

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 causes a **circular waiting** and the program 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: 1/2

- Let the philosophers be 0, 1, 2, 3 and 4, and the weirdo is 4.
- The "normal" philosophers (i.e., 0-3) always pick up their left chopsticks followed by their right ones.
- The weirdo (i.e., 4) always picks up his right chopstick followed by his left.
- This solution is also referred to as the lefty-righty solution.

Dining Philosophers: A Better Idea: 2/2

```
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

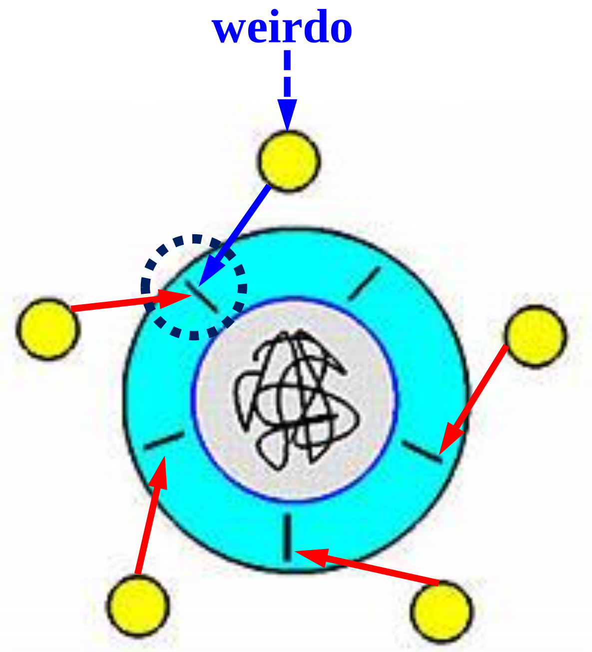
```
while (1) {
   // thinking
   C[(i+1)%5].wait();
   C[i].wait();
   // eating
   C[i].signal();
   C[(i+1)%5].signal();
   // finishes eating
}
```

lock left chop

lock right chop

Discussion: 1/2

- Suppose philosopher 4 is the weirdo.
- **Suppose there is a deadlock.**
- All the normal philosophers have their left and wait for their right.
- Because the weirdo picks up his right chopstick followed by his left, his right neighbor (i.e., Philosopher 1) cannot have his left.
- Or, if Philosopher 1 has his left, then the weirdo cannot have his right chopstick.
- This cannot happen, and the circular waiting pattern cannot happen!

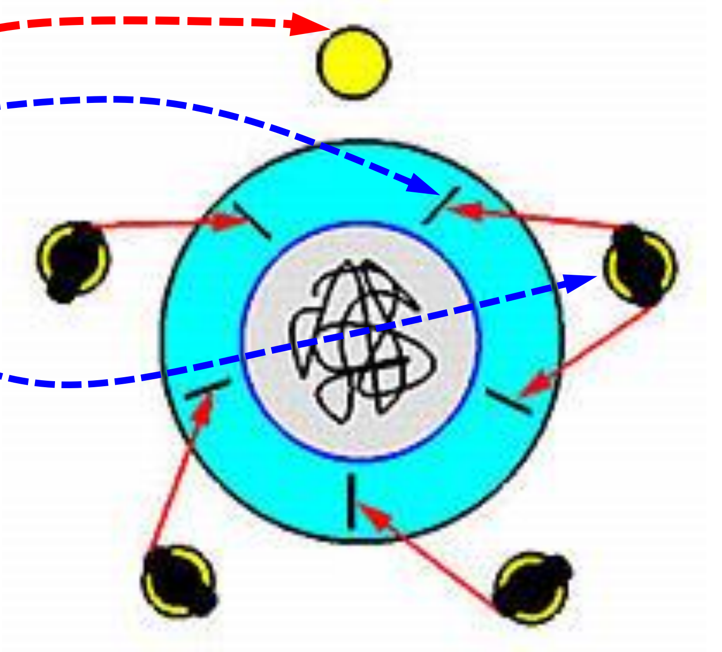


Discussion: 2/2

- **The following are some important questions:**
  - **We choose philosopher 4 to be the weirdo. Does this choice matter?**
  - **What if we have more than one weirdos?**
  - **How many weirdos can be so that this solution is still deadlock-free?**
  - **Can four or fewer philosophers cause a deadlock?**
- **This solution is not symmetric because not all threads run the same code. Furthermore, you need to write two versions of the philosopher code. More codes usually mean more troubles.**

Observation

- All philosophers are normal.
- If all four philosophers sit down, there will be an empty seat.
- The right-most chopstick is free, and the right-most philosopher has a chance to eat.
- What if the right-most philosopher cannot eat? **Exercise!**
- **Circular waiting is broken.**
- **If only four philosophers are allowed to sit down, no deadlock can occur!**



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 count-down lock only allows 4 to go!

this is our old friend

release my chair

Exercises: 1/2

- **We discussed the weirdo and 4-chair versions.**
  - **Use execution sequences to show that some philosophers may have no chance to eat indefinitely (i.e., starvation).**
  - **Prove that the 4-chair version is deadlock-free.**
- **Suppose chopsticks are numbered from 0 to 4. A philosopher always picks up the low number one followed by the high number one. Compare this solution with the weirdo (or lefty-righty) one. What do you find?**

Exercises: 2/2

- **Suppose all chopsticks are in a tray. A philosopher picks up a chopstick as his left followed by another as his right. Is this deadlock-free? How about starvation?**
- **Return to the original version. A philosopher sits down and flips a coin. If the result is a head, he picks up his left chopstick first. Otherwise, he picks his right first. Is this version deadlock-free?**
- **More solutions will be presented in later chapters/units.**

Classical Problems

- Producer/Consumer (Bounded Buffer)
- Readers-Writers
- Roller-Coaster

The Producer/Consumer Problem

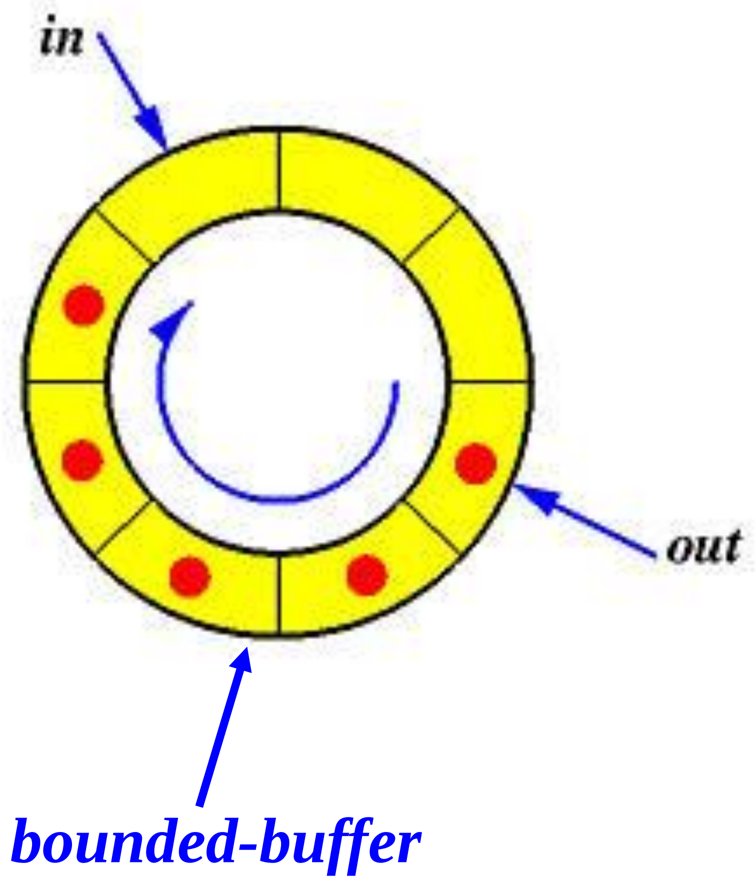
- The Producer/Consumer problem is also referred to as the Bounded Buffer problem.
- It was discussed in E. W. Dijkstra's seminal paper, actually his lecture notes for his course.

4.3. The Bounded Buffer.

I shall give a last simple example to illustrate the use of the general semaphore. In section 4.1 we have studied a producer and a consumer coupled via a buffer with unbounded capacity. This is a typically one-sided restriction: the producer can be arbitrarily far ahead of the consumer, on the other hand the consumer can never be ahaed of the producer. The relation becomes symmetric, if the two are coupled via a buffer of finite size, say N portions. We give the program without any further discussion; we ask the reader to convince himself of the complete symmetry. ("The consumer produces and the producer consumes empty positions in the buffer".) The value N, as the buffer, is supposed to be defined in the surrounding universe into which the following program should be embedded.

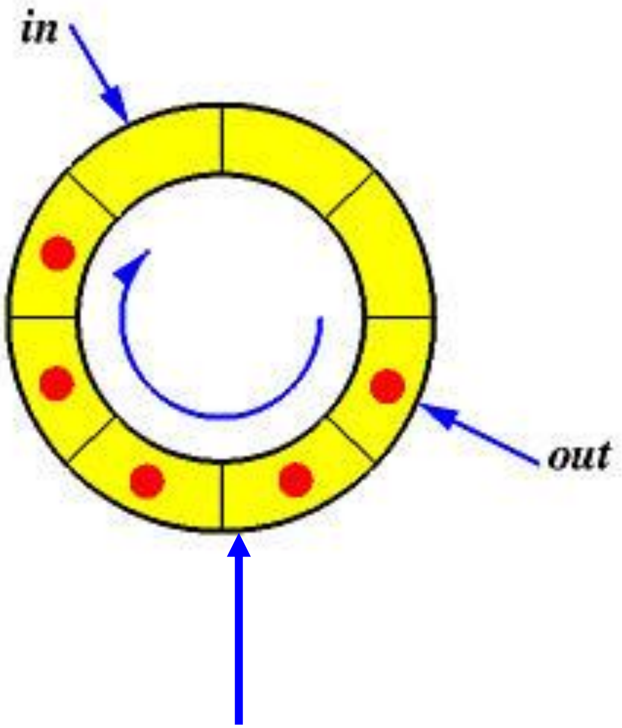
E.W.Dijkstra (1968),
Co-operating Sequential Processes,
in *Programming Languages: NATO Advanced Study Institute: Lectures given at a three weeks Summer School held in Villard-le-Lans 1996*,
edited by F. Genuys (pp. 43-112),
Academic Press, Inc.

The Producer/Consumer Problem



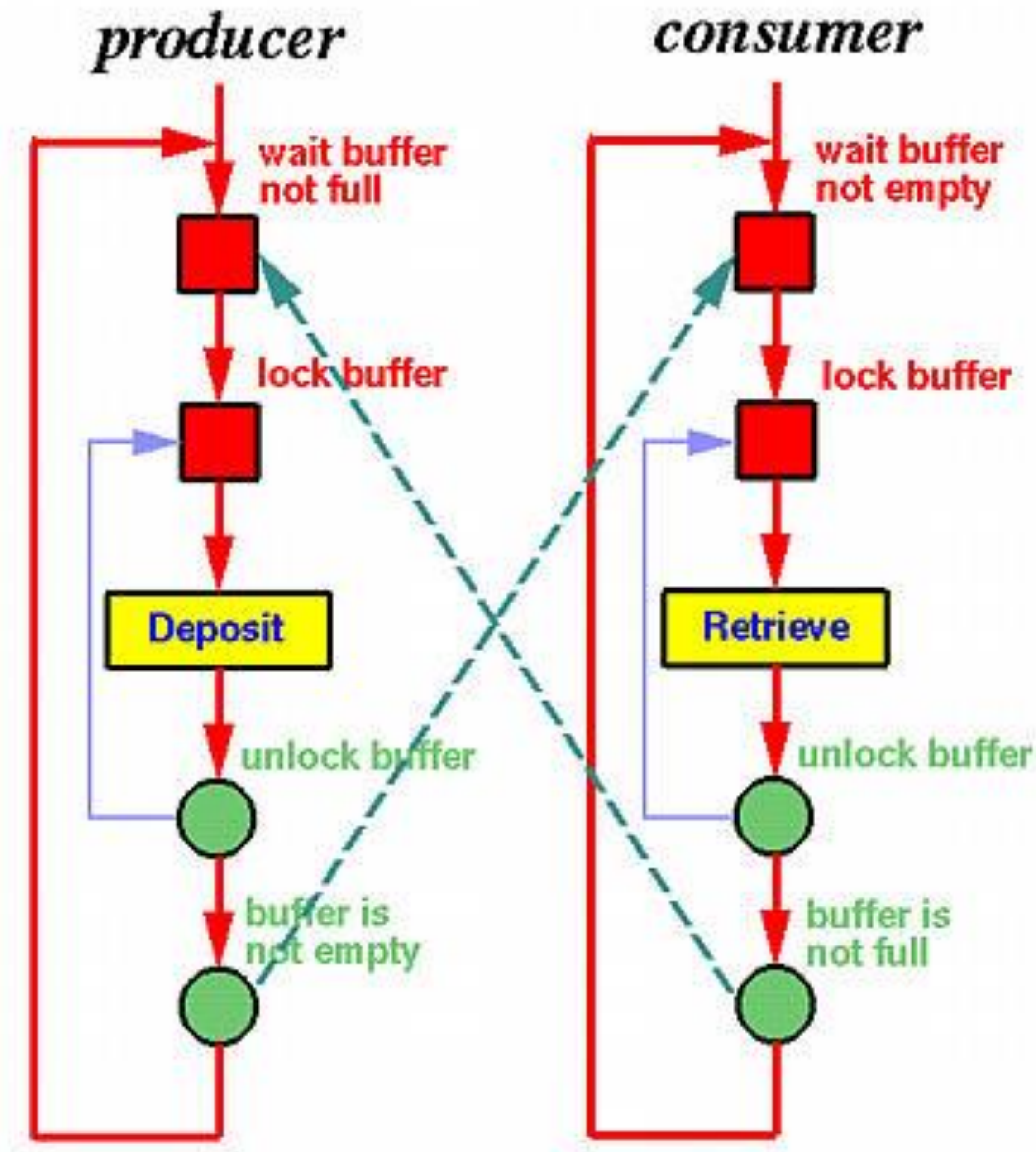
- Suppose we have a **circular buffer** of *n* slots.
- Pointer ***in*** (*resp*., ***out***) points to the first **empty** (*resp*., **filled**) slot.
- **Producer** threads keep adding data into the buffer.
- **Consumer** threads keep retrieving data from the buffer.

Problem Analysis



buffer is implemented with an array `Buf[ ]`

- **A producer deposits data 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.**



- **A semaphore to protect the buffer.**
- **A semaphore to block producers if the buffer is full.**
- **A semaphore to block consumers if the buffer is empty.**

a wait or lock
a notification or unlock

Solution

number of slots

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

<u>**producer**</u>

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

<u>**consumer**</u>

```
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

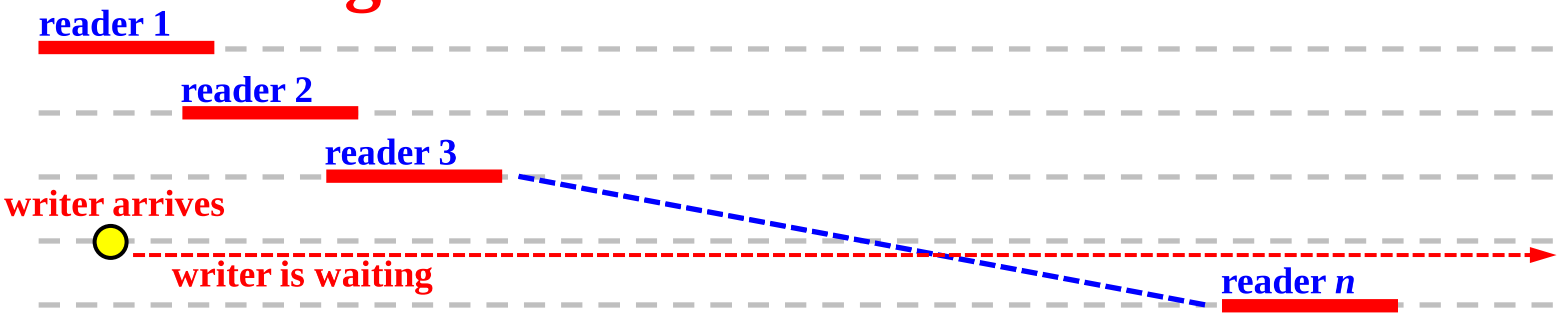
```
while (1) {
  Mutex.wait();
    NotFull.wait();
      Buf[in] = x;
      in = (in+1)%n;
    NotEmpty.signal();
  Mutex.signal();
}
```

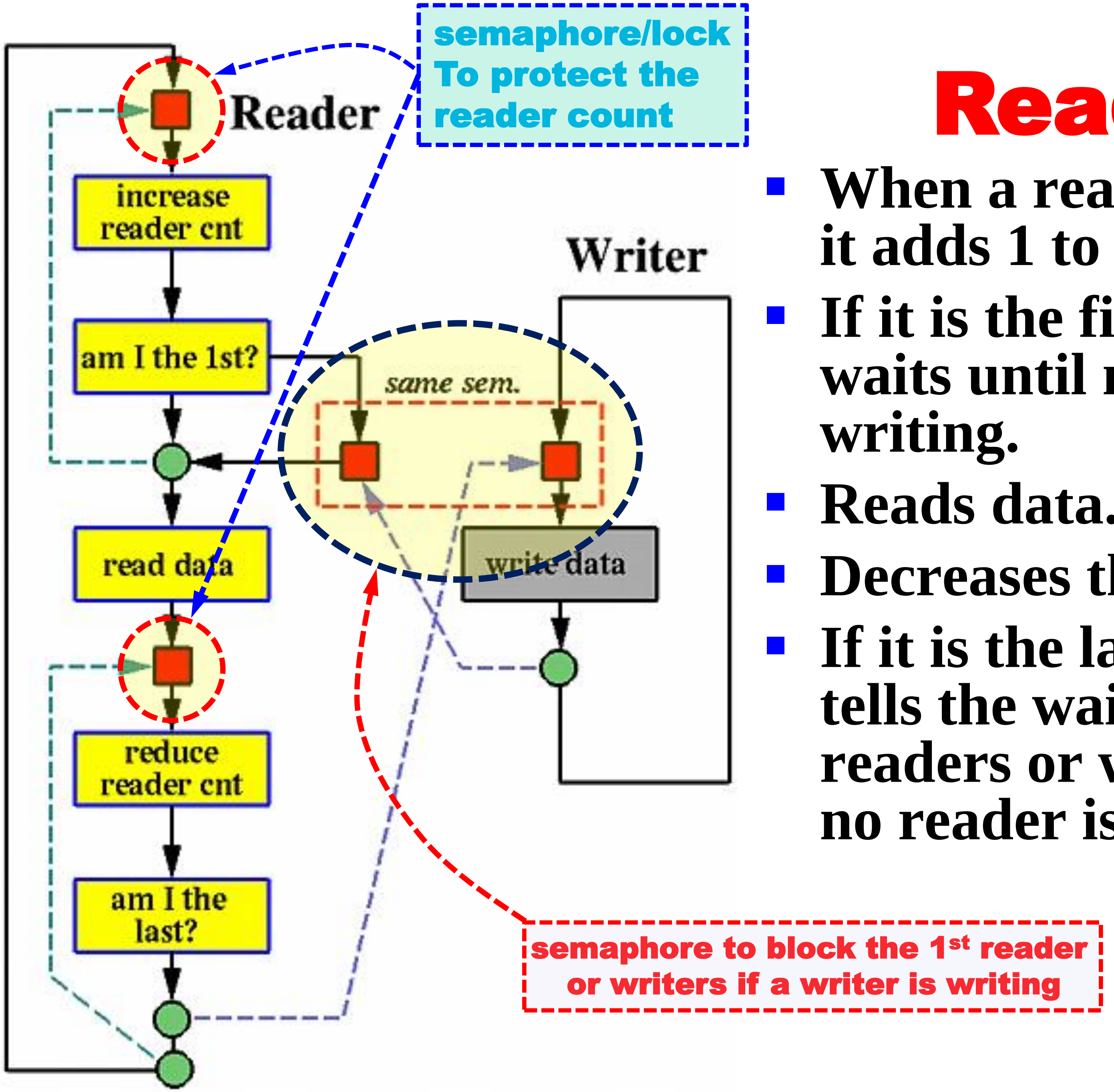
The Readers/Writers Problem

- Two groups of threads, readers and writers, access 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 a higher priority.

Problem Analysis

- A semaphore is needed to **block the first reader and writers if a writer is writing**.
- When a writer arrives, it must **know if there are readers reading**. A reader count is required and must be protected by a lock.
- This **reader-priority** version has a problem: **if readers keep coming in an overlapping way, waiting writers have no chance to write.**

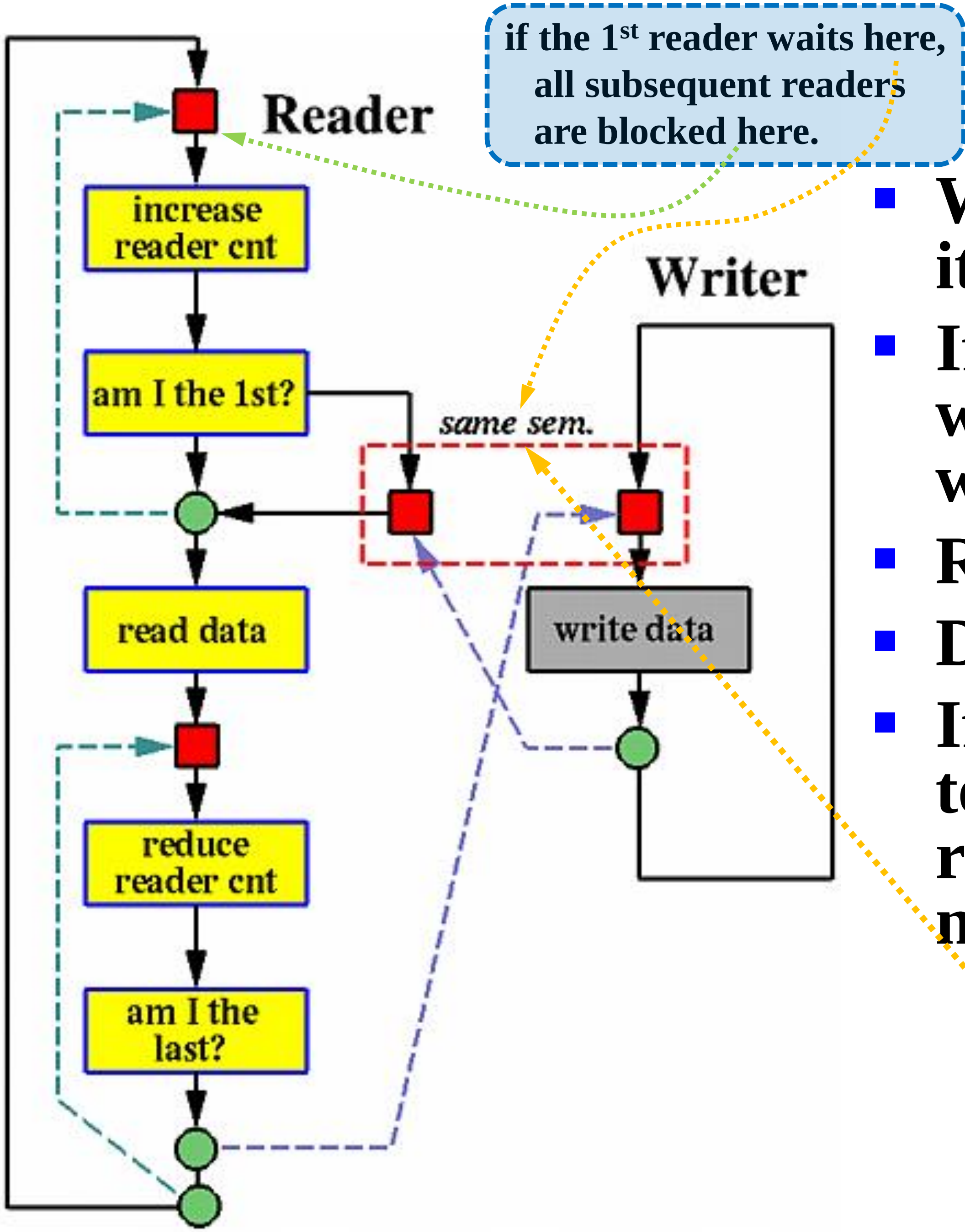




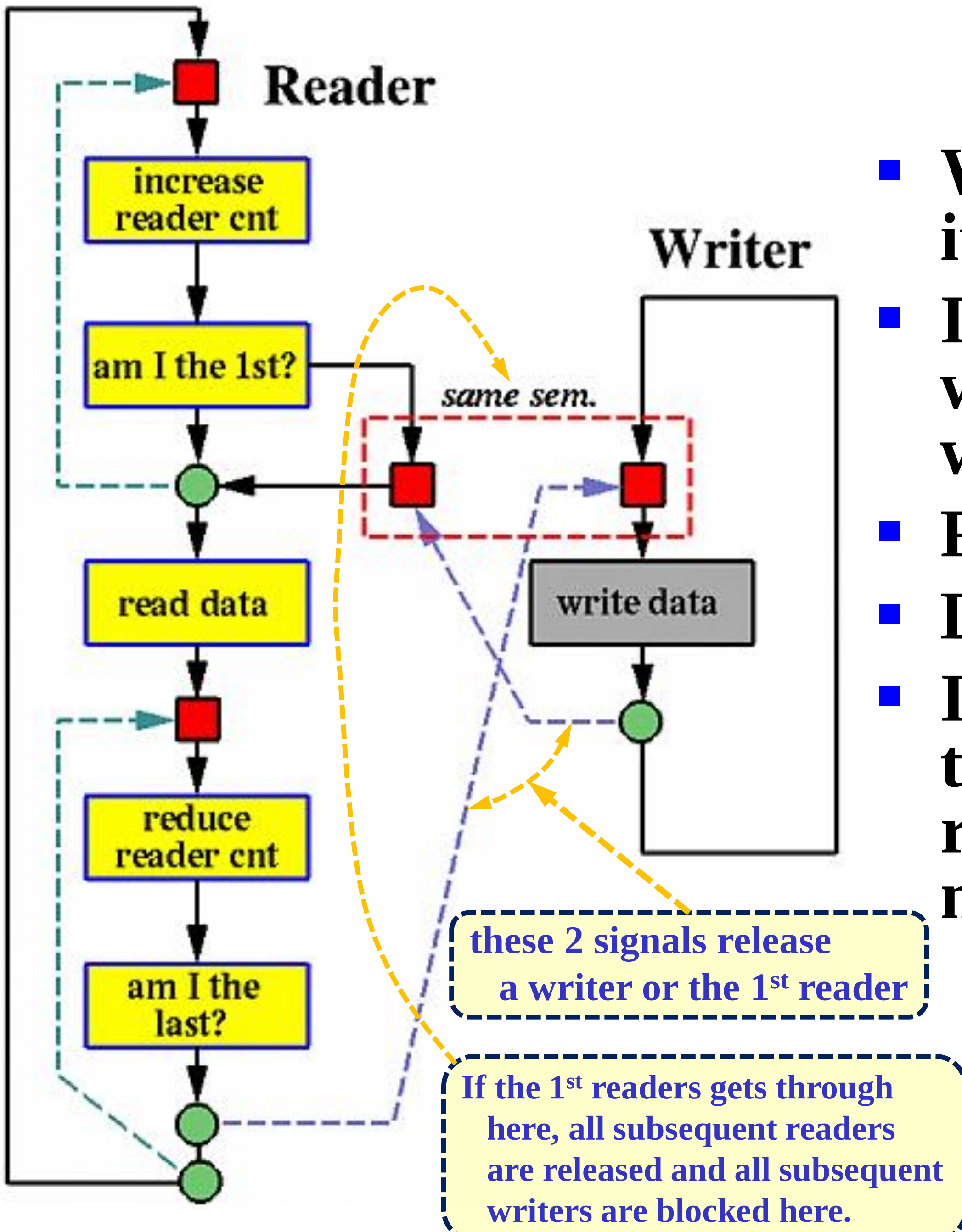
Readers

- When a reader arrives, it adds 1 to the counter.
- If it is the first reader, waits until no writer is writing.
- Reads data.
- Decreases the counter.
- If it is the last reader, tells the waiting readers or writers that no reader is reading.

Readers



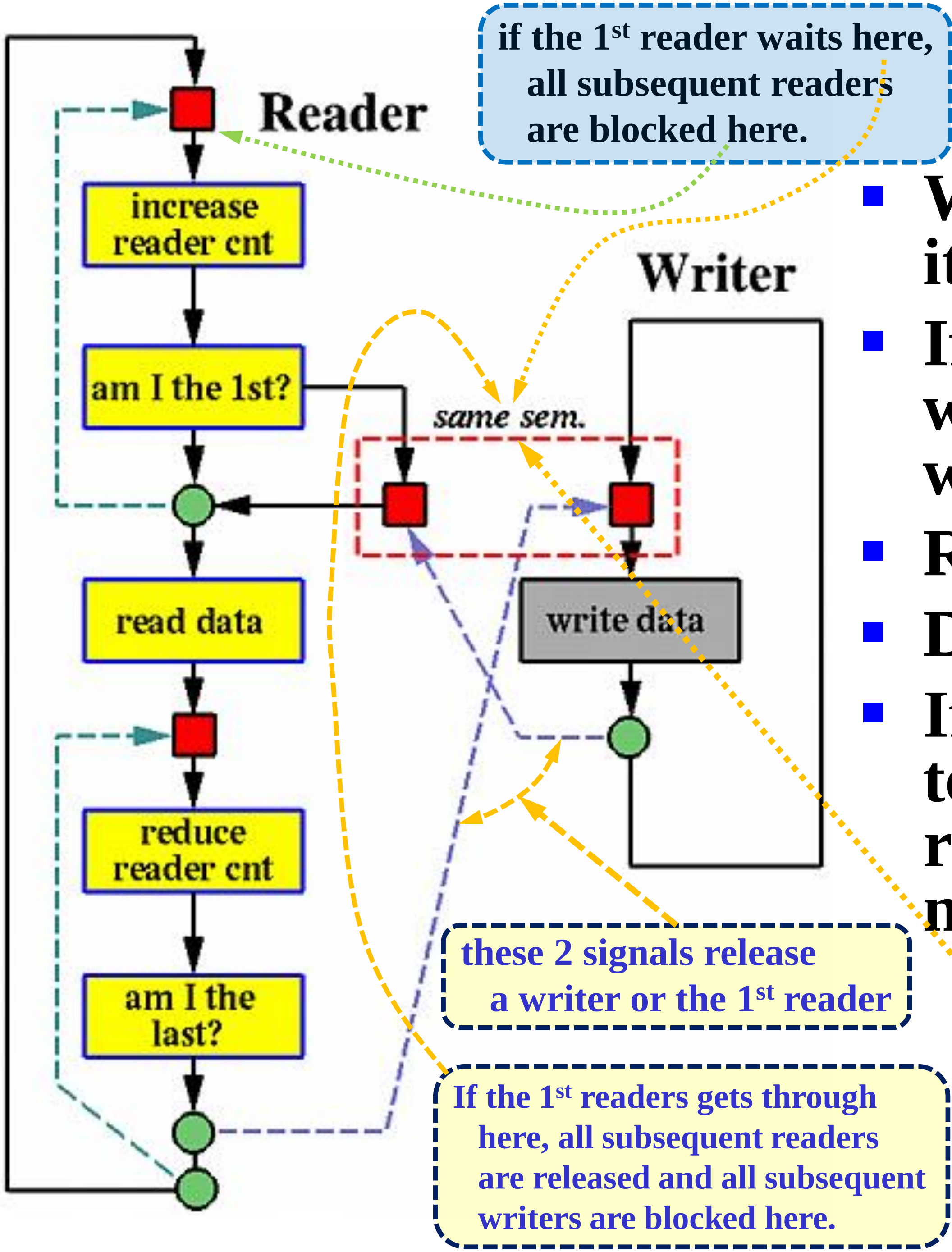
- When a reader arrives, it adds 1 to the counter.
- If it is the first reader, waits until no writer is writing.
- Reads data.
- Decreases the counter.
- If it is the last reader, tells the waiting readers or writers that no reader is reading.



Readers

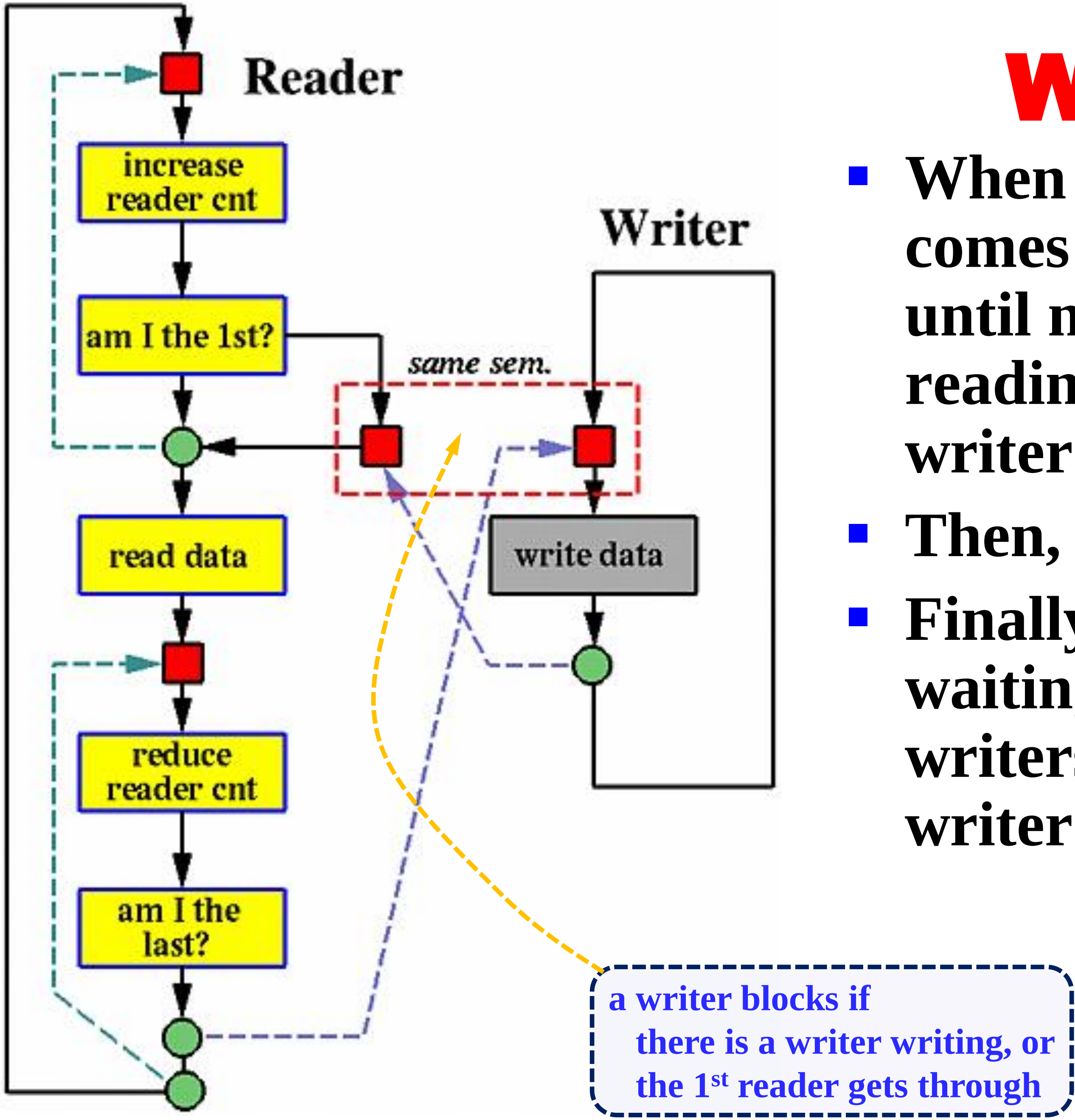
- When a reader arrives, it adds 1 to the counter.
- If it is the first reader, waits until no writer is writing.
- Reads data.
- Decreases the counter.
- If it is the last reader, tells the waiting readers or writers that no reader is reading.

Readers



- When a reader arrives, it adds 1 to the counter.
- If it is the first reader, waits until no writer is writing.
- Reads data.
- Decreases the counter.
- If it is the last reader, tells the waiting readers or writers that no reader is reading.

Writers



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

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();
}
```

writer

```
while (1) {

    WrtMutex.wait();

    // write data

    WrtMutex.signal();

}
```

blocks both readers and writers

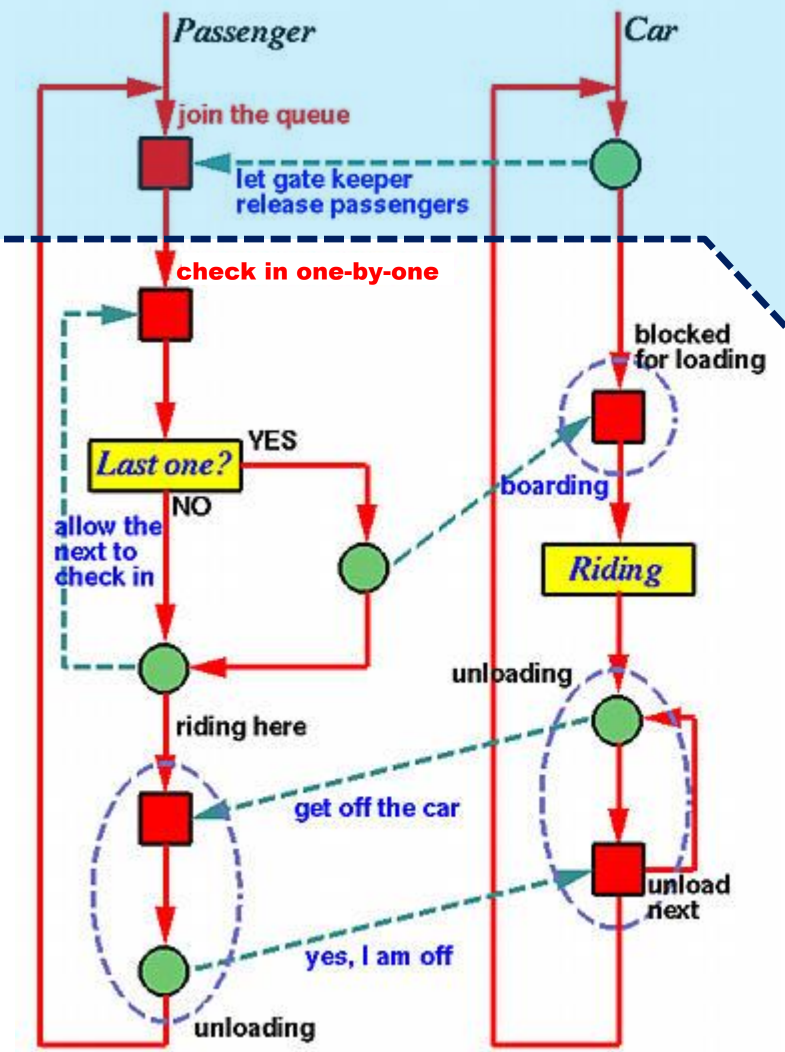
The Roller-Coaster Problem: 1/11

- **Suppose there are *n* passengers and one roller coaster car. The passengers repeatedly wait to ride in the car, which can hold maximum *C* passengers, where $C < n$.**
- **The car can go around the track only when it is full. After finishes a ride, each passenger gets off the car, and wanders around the amusement park before returning to the roller coaster for another ride.**
- **Due to safety concerns, the car only rides *T* times and then shut-down.**

The Roller-Coaster Problem: 2/11

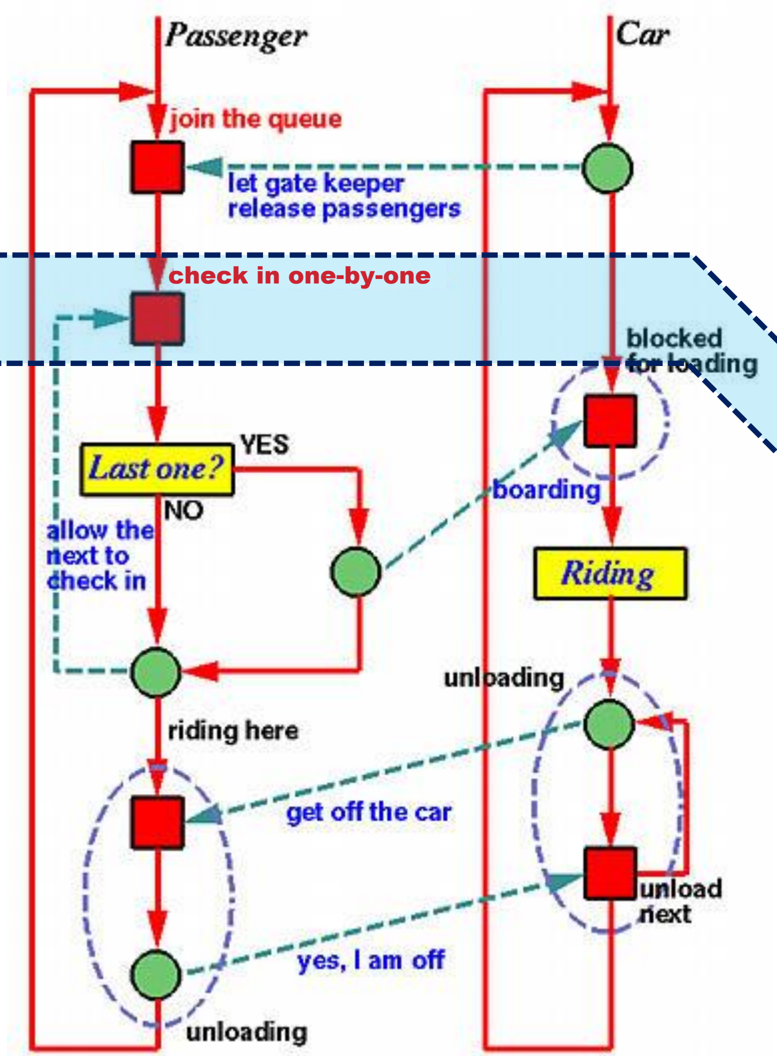
- The car always rides with exactly *C* passengers
- No passengers will jump off the car while the car is running
- No passengers will jump on the car while the car is running
- No passengers will request another ride before they get off the car.

The Roller-Coaster Problem: 3/11



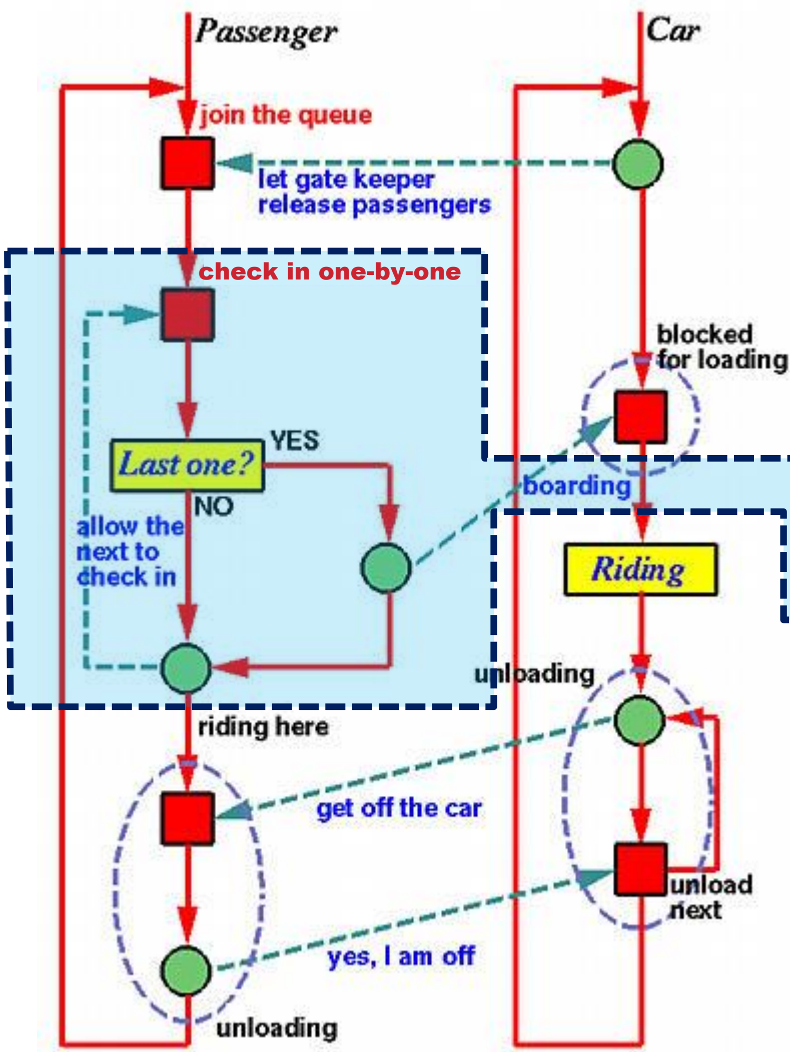
- A passenger decides to have a ride and joins the queue.
- The queue is managed by a gate keeper.
- Passengers check in one-by-one.
- The last passenger tells the car that all passengers are on board.
- Then, they have a ride.
- After riding, passengers get off the car one-by-one.
- Passengers play for a while and come back for a ride.

The Roller-Coaster Problem: 4/11



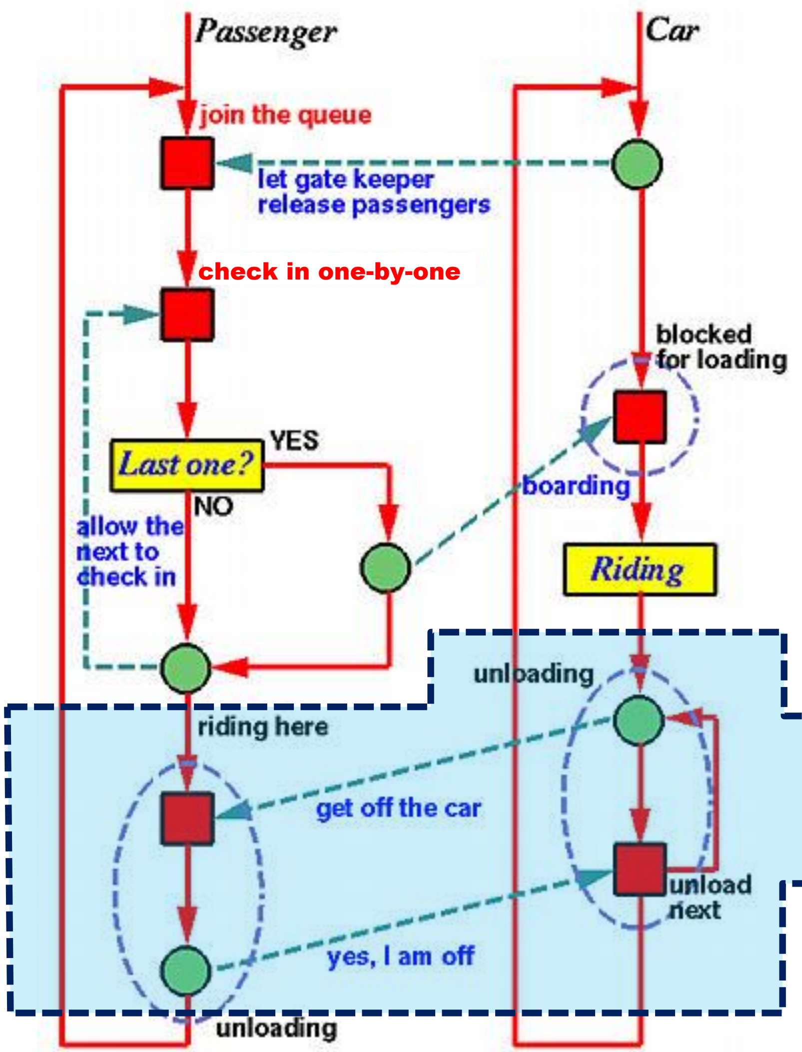
- A passenger decides to have a ride and joins the queue.
- The queue is managed by a gate keeper.
- Passengers check in one-by-one.
- The last passenger tells the car that all passengers are on board.
- Then, they have a ride.
- After riding, passengers get off the car one-by-one.
- Passengers play for a while and come back for a ride.

The Roller-Coaster Problem: 5/11



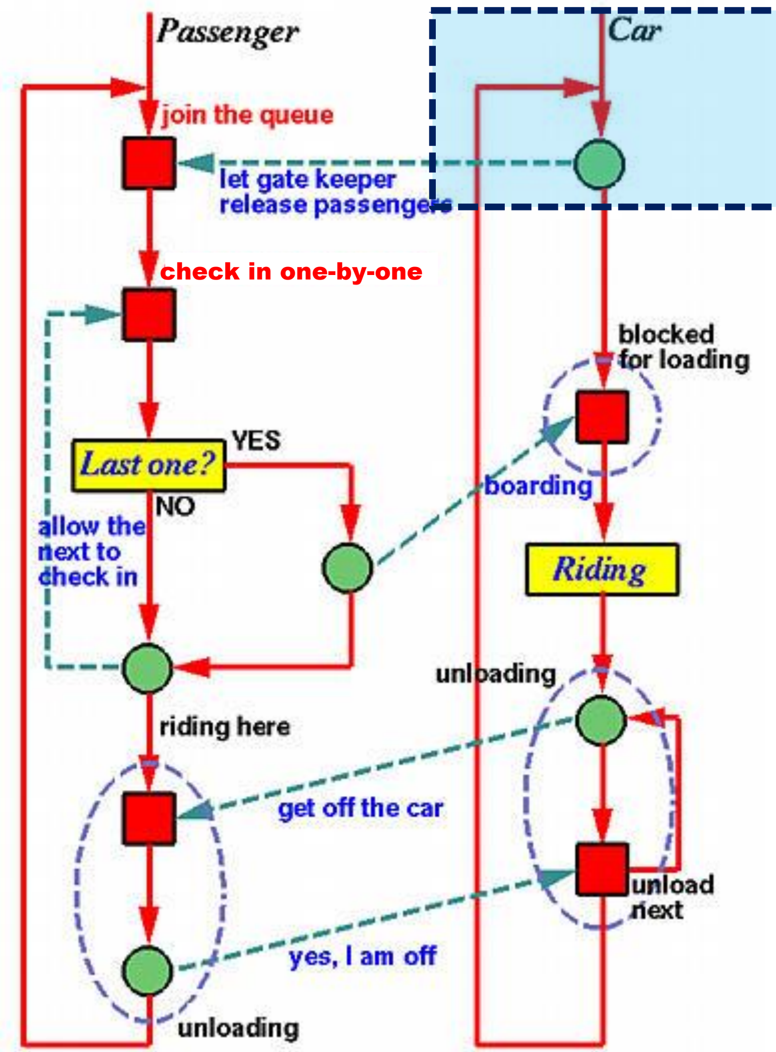
- A passenger decides to have a ride and joins the queue.
- The queue is managed by a gate keeper.
- Passengers check in one-by-one.
- The last passenger tells the car that all passengers are on board.
- Then, they have a ride.
- After riding, passengers get off the car one-by-one.
- Passengers play for a while and come back for a ride.

The Roller-Coaster Problem: 6/11



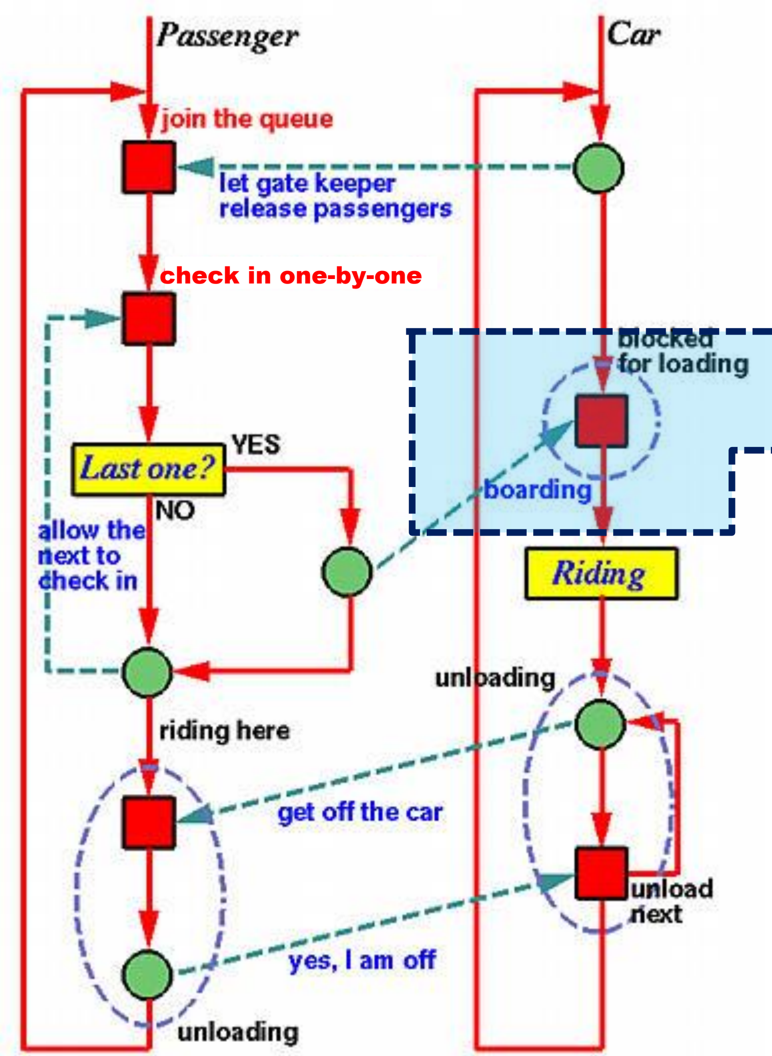
- A passenger decides to have a ride and joins the queue.
- The queue is managed by a gate keeper.
- Passengers check in one-by-one.
- The last passenger tells the car that all passengers are on board.
- Then, they have a ride.
- After riding, passengers get off the car one-by-one.
- Passengers play for a while and come back for a ride.

The Roller-Coaster Problem: 7/11



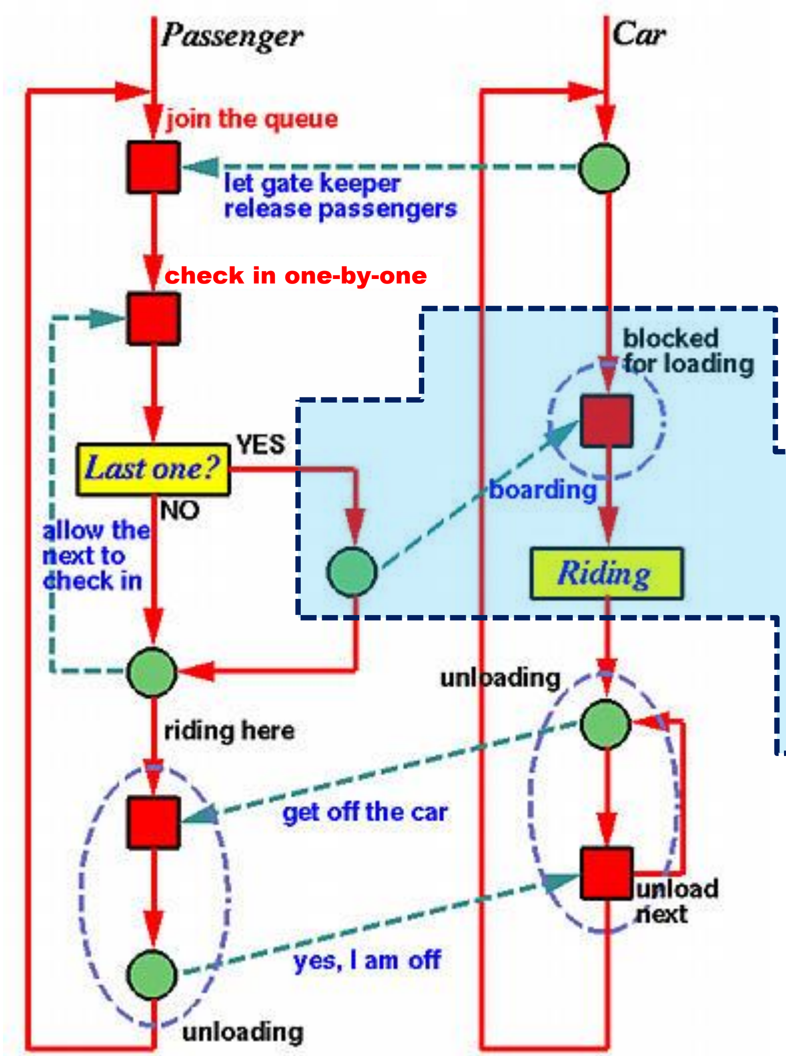
- The car comes and lets the gate keeper know it is available so that the gate keeper could release passengers to check in.
- The car is blocked for loading.
- When the last passenger is in the car, s/he informs the car that all passengers are on board, the car starts a ride.
- After this, the car waits until all passengers are off. Then, go for another round.

The Roller-Coaster Problem: 8/11



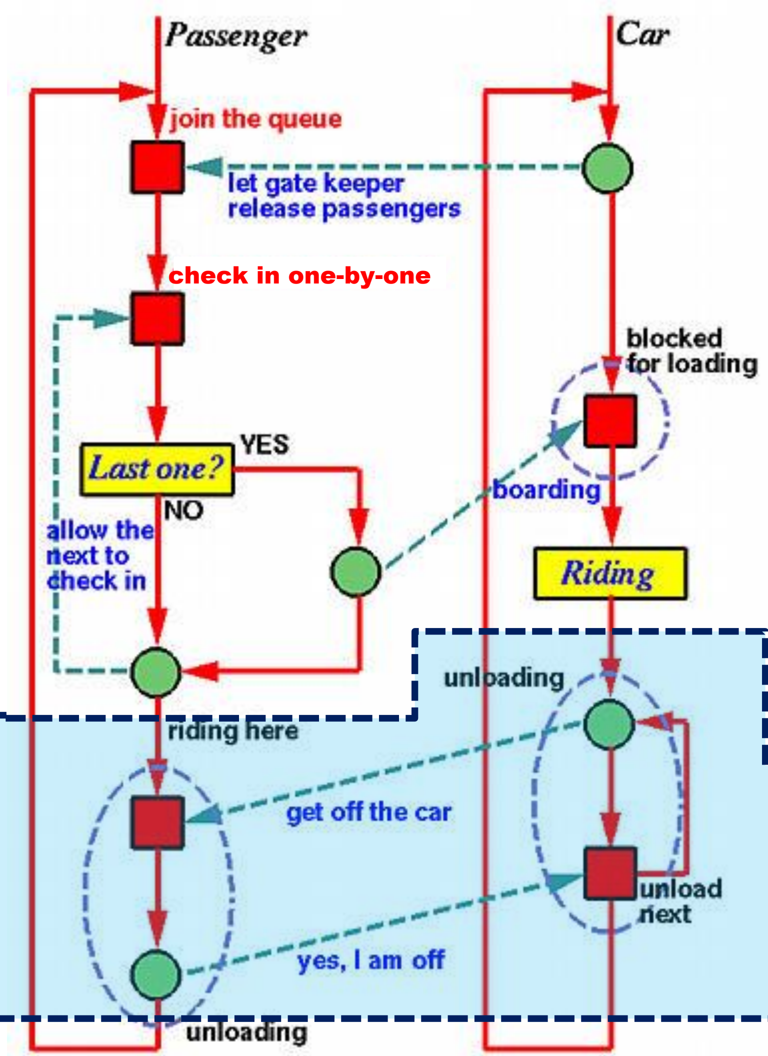
- The car comes and lets the gate keeper know it is available so that the gate keeper could release passengers to check in.
- The car is blocked for loading.
- When the last passenger is in the car, s/he informs the car that all passengers are on board, the car starts a ride.
- After this, the car waits until all passengers are off. Then, go for another round.

The Roller-Coaster Problem: 9/11



- The car comes and lets the gate keeper know it is available so that the gate keeper could release passengers to check in.
- The car is blocked for loading.
- When the last passenger is in the car, s/he informs the car that all passengers are on board, the car starts a ride.
- After this, the car waits until all passengers are off. Then, go for another round.

The Roller-Coaster Problem: 10/11



- The car comes and lets the gate keeper know it is available so that the gate keeper could release passengers to check in.
- The car is blocked for loading.
- When the last passenger is in the car, s/he informs the car that all passengers are on board, the car starts a ride.
- After this, the car waits until all passengers are off. Then, go for another round.

The Roller-Coaster Problem: 11/11

```
int count = 0;
Semaphore Queue = Boarding = Riding = Unloading = 0;
Semaphore Check-In = 1;
```

`count` is shared but not protected. why?

Passenger

```
Wait(Queue);
Wait(Check-In);
if (++count==Maximum)
   Signal(Boarding);
Signal(Check-In);
Wait(Riding);
Signal(Unloading);
```

Car

```
for (i = 0; i < #rides; i++) {
   count = 0; // reset counter before boarding
   for (j = 1; j <= Maximum; j++)
      Signal(Queue); // car available
   Wait(Boarding);
   // all passengers in car
   // and riding
   for (j = 1; j <= Maximum; j++) {
      Signal(Riding);
      Wait(Unloading);
   }
   // all passengers are off
}
```

one ride

Exercise:
This code unloads passengers one-by-one. Is this necessary?
Can `Unloading` be removed?

A Quick Summary: 1/2

- We have learned a few tricks in this component: locks, count-down locks and notification.
- Very often a counter is needed to determine if certain condition is met (e.g., number of readers in the readers-writers problem, check-in and boarding in the roller-coaster problem).
- Sometimes threads may have to be “paired-up” like the get-off process we saw in the roller-coaster problem.
- Use these basic and frequently seen patterns to solve other problems.

A Quick Summary: 2/2

- **Using many semaphores could mean more locking and unlocking activities and could be inefficient.**
- **Using only a few semaphores could produce very large critical sections, and a thread could stay in a critical section for a long time. Thus, other threads may have to wait very long to get in.**
- **Therefore, try your best to minimize the number of semaphores and reduce the length of locking time.**

Patterns and Pass-the-Baton

What Is a Pattern?

- A **pattern** is simply a description/template for solving a problem that can be used in several situations.
- A pattern is **NOT** a complete solution to a problem. It is just a template and requires extra work to make it a solution to a specific problem.
- We will discuss a few patterns related to the use of semaphores.

Mutual Exclusion – Of Course!

- This is the easiest one for enforcing mutual exclusion so that race conditions will not occur.
- A semaphore is initialized to 1. Then, use the `Wait()` and `Signal()` methods to lock and unlock the semaphore, respectively.

```
Semaphore Lock(1);

Wait(Lock);
   // critical section
Signal(Lock);
```

```
Semaphore   S(1);
int         c = 0;

Wait(S);                  Wait(S);
  c++;                      c--;
Signal(S);                Signal(S);
if (c >= 3) {             if (c == 0) {
   ...                       ...
```

while `c` is being tested, it could be updated!

Enter-and-Test: 1/2

- In many applications, a thread may enter a critical section and test for a condition. If that condition is met, the thread does **something$_1$**. Otherwise, its does **something$_2$**.
- Frequently, one of the two **something**s may involve a wait.

Reader: Enter

```
Mutex.wait();
    RdrCount++;
    if (RdrCount == 1)
      WrtMutex.wait();
Mutex.signal();
  // read data
```

critical section

If the condition is met (i.e., `RdrCount` being 1), then waits until it is released.

In this case, the first reader does something (*i.e.*, waiting) and at the same time has the `Mutex`.

In this way, no other threads can enter the critical section.

Enter-and-Test: 2/2

- Usually, a wait may be used in the entry part to wait for a particular condition to occur, and a signal is used upon exit to release a waiting thread.

Reader: Exit

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

critical section

if the condition is met (i.e., `RdrCount` being 0), then tell someone, a reader or a writer, to continue. In this case, the last reader does something.

Exit-Before-Wait: 1/2

- In many applications, a thread exits a critical section and then blocks itself.
- Usually, a thread updates some variables in a critical section, and then waits for a resource from another thread.

Roller-Coaster: Passenger

```
Wait(Queue);
Wait(Check-In);
if (++count==Maximum)
   Signal(Boarding);
Signal(Check-In);
Wait(Riding);
Signal(Unloading);
```

if the condition is met (i.e., `count` being the maximum), then notify some thread.

after exiting the critical section, wait for some event to happen.

critical section for `count`

Exit-Before-Wait: 2/2

- This `signal`ing an event followed by `wait`ing on another must be used with care.
- A context switch can happen between the `signal` and the `wait`.
- For example, a thread enters the critical section, signals `s1` upon exit, and gets swapped out before reaches the wait. This could cause some problems. **Why?** So, be careful!

```
Wait(s1);
  // critical section
Signal(s1);
Wait(s2);
```

A context switch could occur here!
This thread may not wait immediately
Is this OK? It all depends on the logic of your program.

Conditional Waiting/Signaling

- A thread waits or notifies another thread if a condition is satisfied.
- Make sure that no race conditions will occur while the condition is being tested.

```
if (count > 0)                    if (count == 0)
   Signal(OK_to_GO);                 Wait(Block_Myself);
```

are there other threads updating `count` at the same time?

Passing the Baton: 1/8

- **If a thread is in its critical section, it holds the baton (i.e., the critical section).**
- **Upon exit, if there are threads waiting to enter the CS, the exiting thread passes the baton (*i.e.*, the critical section) to one of them directly.**
- **In this way, we save a `signal-wait` pair.**
- **If no thread is waiting, the baton is passed to the next thread that will try to enter the CS later.**
- **This is a technique that can make the use of semaphores more efficient.**

Passing the Baton: 2/8

- The **Waiting** thread waits on `Condition` if `Event` is not there. The **Signaling** thread sets `Event` and releases a **Waiting** thread.

```
Semaphore Mutex(1);
Semaphore Condition(0);
Bool      Event = FALSE;
```

Waiting Thread

```
Wait(Mutex);
while (!Event) {
  Signal(Mutex);
  Wait(Condition);
  Wait(Mutex);
}
```

try again!

Signaling Thread

```
Wait(Mutex);
  Event = TRUE;

  Signal(Condition);
Signal(Mutex);
```

critical section for protecting `Event`

Passing the Baton: 3/8

- **Waiting** does not acquire the CS. Instead, **Signaling** has the CS, does not release it, and gives the CS to **Waiting** (i.e., baton passed)
- **Signaling** must be sure that **Waiting** will not do any harm to the CS.

Waiting Thread

`Wait(Mutex);`
`while (!Event) {` release CS
`   Signal(Mutex);`
`   Wait(Condition);`
`   ~~Wait(Mutex);~~`
`}`

Signaling Thread

`Wait(Mutex);` acquire CS
`   Event = TRUE;`
`Signal(Condition);` pass
`~~Signal(Mutex);~~`

has the CS and does not have to wait

Passing the Baton: 4/8

Threads waits for an event to occur before they can continue.

A **Waiting** thread checks to see if the event is there. If the event is not there, it waits.

Variable `Waiting` is used to count the number of waiting thread.

Semaphore `Condition` is used to block threads if the event is not there.

```
Semaphore Mutex(1), Condition(0);
int       Event = FALSE, Waiting = 0;
```

Waiting Thread

```
Wait(Mutex);             // lock Event/Waiting
if (!Event) {            // if Event not there
   Waiting++;            //    join the waiting
   Signal(Mutex);        //    release the lock
   Wait(Condition);      //    wait!
}
      // has the event & does something
      // upon exit, releases a waiting thread
if (Waiting > 1) {       // anyone waiting?
   Waiting--;            //    try to release
   Signal(Condition);    //    release a thread
}
Signal(Mutex);           // release the lock
```

Original **Waiting**

Passing the Baton: 5/8

```
Semaphore Mutex(1), Condition(0);
int       Event = FALSE, Waiting = 0;

Signaling Thread
Wait(Mutex);                  // lock Event/Waiting
   Event = TRUE;              // Event is there
   if (Waiting > 0) {         // Anyone waiting?
      Signal(Condition);      //   release it
   }
Signal(Mutex);                // release the lock
```

- A **Signaling** thread locks the semaphore `Mutex` and hence `Waiting`.
- Then, this **Signaling** thread makes the event to happen.
- It checks if there are threads waiting for this event.
- If there are waiting threads, release one of them.
- Finally, this **Signaling** thread releases the lock.

Original **Signaling**

Passing the Baton: 6/8

```
Semaphore Mutex(1), Condition(0);
int       Event = FALSE, Waiting = 0;
```

Waiting Thread

```
Wait(Mutex);
if (!Event) {
   Waiting++;
   Signal(Mutex);
   Wait(Condition);
}
// Process Event
if (Waiting > 1) {
   Waiting--;
   Signal(Condition);
}
Signal(Mutex);
```

Signaling Thread

```
Wait(Mutex);
Event = TRUE;

if (Waiting > 0) {

   Signal(Condition);
}
Signal(Mutex);
```

Only needed if `Waiting` = 0

The signaling thread has the CS and can pass it to the next

Without Baton Passing (Original Version)

Passing the Baton: 7/8

```
Semaphore Mutex(1), Condition(0);
int       Event = FALSE, Waiting = 0;
```

Waiting Thread

```
Wait(Mutex);
if (!Event) {
  Waiting++;
  Signal(Mutex);
  Wait(Condition);
}
...
if (Waiting > 1) {
  Waiting--;
  Signal(Condition);
}
else
  Signal(Mutex);
...
```

Signaling Thread

```
Wait(Mutex);
Event = TRUE;
if (Waiting > 0)
  Signal(Condition);
else
  Signal(Mutex);
...
```

baton acquired

a Mutex needed to protect Waiting

baton passed

baton released

With Baton Passing

Passing the Baton: 8/8

- **Passing the baton** technically transfers the ownership of a critical section from a thread to another thread.
- The thread that has the baton does not need a signal to release it. Instead, the CS is directly given to another that needs it. The receiving thread does not need a wait for the CS.
- In this way, mutual exclusion may be destroyed; but, we reduce the number of entering and exiting a mutex.

Pass-the-Baton Example

Passing the Baton: Example 1/23

- We shall use the reader-priority version of the readers/writers problem as a more complex example.
- Note the following conditions:
  - If there is no writer writing, a reader can read.
  - If there is no readers reading and there are waiting writers, allow a writer to write (i.e., better!).
  - If there are readers reading **OR** a writer writing, no writer can write.
  - If there are waiting readers, a finishing writer should allow a reader to read (i.e., reader priority).
  - If there are waiting writers and no waiting reader, a finishing writer should allow a writer to write.

Passing the Baton: Example 2/23

- We will need counters for counting waiting readers and writers and active readers and writer.
- A semaphore for protecting all counters.
- A semaphore for blocking readers.
- A semaphore for blocking writers.

```
int aReaders = 0;        // number of active readers (>= 0)
int aWriters = 0;        // number of active writer (0 or 1)
int wReaders = 0;        // number of waiting readers
int wWriters = 0;        // number of waiting writers

Semaphore Mutex(1);      // semaphore for protecting counters
Semaphore Reader(0);     // semaphore for blocking readers
Semaphore Writer(0);     // semaphore for blocking writers
```

Passing the Baton: Example 3/23

Developing the Entry Section for **Readers** : 1/6

```
Wait(Mutex);
if (aWriters > 0) {
    wReaders++;
    Signal(Mutex);
    Wait(Reader);
    Wait(Mutex);
}
aReaders++;
if (wReaders > 0) {
    wReaders--;
    Signal(Reader);
}
Signal(Mutex);
```

Mutex is used to protect all counters

Passing the Baton: Example 4/23

Developing the Entry Section for Readers : 2/6

```
Wait(Mutex);
if (aWriters > 0) {
   wReaders++;
   Signal(Mutex);
   Wait(Reader);
   Wait(Mutex);
}
aReaders++;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
Signal(Mutex);
```

If there is a writer writing, this reader must wait. As a result, this reader must release the Mutex and wait on Reader.

If a reader is released from Reader, this reader must reacquire the Mutex in order to update aReaders and wReaders, and test wReaders.

Passing the Baton: Example 5/23

Developing the Entry Section for **Readers** : 3/6

```
Wait(Mutex);
if (aWriters > 0) {
   wReaders++;
   Signal(Mutex);
   Wait(Reader);
   Wait(Mutex);
}
aReaders++;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
Signal(Mutex);
```

If there is a writer writing, this reader must wait. As a result, this reader must release the `Mutex` and wait on `Reader`.

If a reader is released from `Reader`, this reader must reacquire the `Mutex` in order to update `aReaders` and test `wReaders`.

If a reader reaches here, then
(1) it is released from `Reader`,
or
(2) there is no writer writing.
Then, this reader can read.

Passing the Baton: Example 6/23

Developing the Entry Section for **Readers : 4/6**

```
Wait(Mutex);
if (aWriters > 0) {
    wReaders++;
    Signal(Mutex);
    Wait(Reader);
    Wait(Mutex);
}
aReaders++;
if (wReaders > 0) {
    wReaders--;
    Signal(Reader);
}
Signal(Mutex);
```

If a reader reaches here, then
(1) it is released from Reader,
or
(2) there is no writer writing.
Then, this reader can read.

This reader adds 1 to the active reader count.
If there are waiting readers, release one.
This released reader releases other waiting readers in a cascading way.

Passing the Baton: Example 7/23

Developing the Entry Section for Readers : 5/6

```
Wait(Mutex);
if (aWriters > 0) {
   wReaders++;
   Signal(Mutex);
   Wait(Reader);
   Wait(Mutex);
}
aReaders++;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
Signal(Mutex);
```

This first reader that sees no writing writer can read immediately.

This reader starts to signal `Reader` to release other waiting readers.

The first reader executes `Signal()` to `Reader`.

If a reader is released, this reader can read because readers can read simultaneously.

This reader signals `Reader` to release the next reader waiting on `Reader`.

Therefore, a sequence of signals releases the waiting readers on `Reader` one-by-one.

This is a **cascading** signal/release.

Passing the Baton: Example[8/23]

Developing the Entry Section for **Readers : 6/6 (Summary)**

```
Wait(Mutex);        acquire
if (aWriters > 0) {
   wReaders++;
   Signal(Mutex);   release
   Wait(Reader);
   Wait(Mutex);     re-acquire
}
aReaders++;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
Signal(Mutex);      re-release
```

The first reader, who sees **no writer writing** and **some readers waiting**, releases the waiting readers.

The release of readers is in a **cascading** way (i.e., one after the other).

The first thread acquires the baton, and **may** pass it to each of the released thread.

After releasing a waiting reader, this reader releases the baton (`Mutex`) and starts reading

Passing the Baton: Example 9/23

Developing the Entry Section for **Readers**: Passing the Baton

```
Wait(Mutex);
if (aWriters > 0) {
    wReaders++;
    Signal(Mutex);
    Wait(Reader);
    Wait(Mutex);
}
aReaders++;
if (wReaders > 0) {
    wReaders--;
    Signal(Reader);
}
else
    Signal(Mutex);
```

The first reader, who gets the baton and sees no writer writing and some readers waiting, will release the waiting readers and pass its baton to the released reader.

Each released reader, after releasing the next reader, passes the baton to it. Then, it goes away without releasing the baton.

Only the last released returns the baton.

Note that the critical section (i.e., the baton) is locked throughout this cascading release, and no other thread can pass through the first `Wait(Mutex)`.

Passing the Baton: Example[10/23]

Developing the Exit Section for **Readers**

```
Wait(Mutex);
aReaders--;
if (aReaders=0 & wWriters>0) {
   wWriters--;
   Signal(Mutex);
   Signal(Writer);
}
else
   Signal(Mutex);
```

acquire

release

If this is the last reader and there are waiting writers, let one writer go.

Otherwise, simply go away.

This signal is not needed, because the last reader can pass the baton to the released writer!

Passing the Baton: Example 11/23

Developing the Exit Section for **Readers**: Passing the Baton

```
Wait(Mutex);
aReaders--;
if (aReaders=0 & wWriters>0) {
    wWriters--;
    Signal(Mutex);
    Signal(Writer);
}
else
    Signal(Mutex);
```

If this is the last reader and there are waiting writers, let one writer go.

Otherwise, simply go away.

This release of the baton is not needed, because the last reader passes the baton to the released writer.

The baton is passed to the released writer.

Does the following work?

```
Wait(Mutex);
aReaders--;
if (aReaders=0 & wWriters>0) {
    wWriters--;
    Signal(Writer);
}
Signal(Mutex);
```

Would this version work properly in terms of passing the baton? Why? Exercise!

Passing the Baton: Example 12/23

Developing the Entry Section for **Writers**

```
Wait(Mutex);
if (aReaders>0 | aWriters>0) {
    wWriters++;
    Signal(Mutex);
    Wait(Writer);
    Wait(Mutex);
}
aWriters++;
Signal(Mutex);
```

acquire

release

re-acquire

re-release

A writer, who gets the baton and sees some readers reading **OR** a writer writing, will release the baton and wait.

Later, this writer reacquires the baton to update `aWriters`.

Otherwise (i.e., no readers reading **AND** no writer writing), this writer adds 1 to the number of active writers, releases the baton, and starts writing.

Passing the Baton: Example 13/23

Developing the Entry Section for **Writers**: Passing the Baton

```
Wait(Mutex);              acquire
if (aReaders>0 | aWriters>0) {
    wWriters++;                        baton
    Signal(Mutex); release
    Wait(Writer); receives the baton
    ~~Wait(Mutex);~~     from a Reader
}
aWriters++;     release the received baton
Signal(Mutex);
```

The exiting reader acquires the baton, releases a writer w/o releasing the baton.

Hence, the released writer has the baton, increases the active writers count, releases the baton and writes.

baton

Exit Section of Readers: Passing the Baton

```
Wait(Mutex);
aReaders--;
if (aReaders=0 & wWriters>0) {
    wWriters--;
    Signal(Writer);
}
else
    Signal(Mutex);
```

Passing the Baton: Example

Developing the Exit Section for **Writers**

```
Wait(Mutex);
aWriters--;
if (wReaders > 0) {
    wReaders--;
    Signal(Mutex);
    Signal(Readers);
}
else if (wWriters > 0) {
    wWriters--;
    Signal(Mutex);
    Signal(Writer);
}
else
    Signal(Mutex);
```

acquire

release

release

release

Lock the `Mutex` first.

If there are waiting readers, let one of them go.

If there is no waiting readers but there are waiting writers, let one of them go.

If there is no waiting readers and no waiting writers, then do nothing.

But, don't forget to release the `Mutex`.

Passing the Baton: Example

Developing the Exit Section for **Writers**: Passing the Baton

```
Wait(Mutex);
aWriters--;
if (wReaders > 0) {
    wReaders--;
    Signal(Mutex);
    Signal(Readers);
}
else if (wWriters > 0) {
    wWriters--;
    Signal(Mutex);
    Signal(Writer);
}
else
    Signal(Mutex);
```

This writer acquires the baton.

This writer may pass the baton to a released reader or a released writer.

The baton is not passed if there is no waiting readers and there is no waiting writers.

In this case, this writer must release the baton explicitly.

Summary
Passing the Baton

1/2

```
Wait(Mutex);
if (aWriters > 0) {
   wReaders++;
   Signal(Mutex);
   Wait(Reader);
}
aReaders++;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
else
   Signal(Mutex);
```

acquire

pass

loops here to release all readers

release by the last released reader

// READING

```
Wait(Mutex);
aReaders--;
if (aReaders = 0 & wWriters > 0) {
   wWriters--;
   Signal(Writer);
}
else
   Signal(Mutex);
```

acquire

pass

```
Wait(Mutex);
if (aReaders > 0 | aWriters > 0) {
   wWriters++;
   Signal(Mutex);
   Wait(Writer);
}
aWriters++;
Signal(Mutex);
```

release by the finishing writer

// WRITING

```
Wait(Mutex);
aWriters--;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
else if (wWriters > 0) {
   wWriters--;
   Signal(Writer);
}
else // wReaders = wWriters = 0
   Signal(Mutex);
```

Summary
Passing the Baton

2/2

```
Wait(Mutex);
if (aWriters > 0) {
   wReaders++;
   Signal(Mutex);
   Wait(Reader);
}
aReaders++;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
else
   Signal(Mutex);
```

loops here to release all readers

release by the last released reader

pass

```
Wait(Mutex);
if (aReaders > 0 | aWriters > 0) {
   wWriters++;
   Signal(Mutex);
   Wait(Writer);
}
aWriters++;
Signal(Mutex);
```

release by the finishing writer

pass

// READING

```
Wait(Mutex);
aReaders--;
if (aReaders = 0 & wWriters > 0) {
   wWriters--;
   Signal(Writer);
}
else
   Signal(Mutex);
```

acquire

// WRITING

```
Wait(Mutex);
aWriters--;
if (wReaders > 0) {
   wReaders--;
   Signal(Reader);
}
else if (wWriters > 0) {
   wWriters--;
   Signal(Writer);
}
writers // wReaders = wWriters = 0
   Signal(Mutex);
```

acquire

Passing the Baton: Example 18/23

- One of the several advantages of this solution is that it can easily be modified to achieve other goals. Here is a writer-priority version.
- A **writer-priority** version should satisfy the following conditions:
  1. New readers are blocked if a writer is waiting, and
  2. A waiting reader is released if no writer is writing.

Study the conditions for releasing readers and writers.

Passing the Baton: Example

- To meet **Condition (1)**, we must change the first (enter) `if` statement of the reader thread:

added component

```
                                              // as long as there
                                              //  is a writer waiting
if (aWriters > 0 || wWriters > 0) {           // yield to writers
   wReaders++;                                // joint the line,
   Signal(Mutex);                             // release the baton,
   Wait(Reader);                              // and wait
}
```

When a reader arrives, if there are writers waiting or writing, this reader blocks, and hence yields to writers.

Passing the Baton: Example[20/23]

- To meet **Condition (2)**, we must strengthen the last (exit) `if` statement of the writer thread:

```
if (wReaders > 0 && wWriters == 0) { // release readers
   wReaders--;                       //   only if there
   Signal(Reader);                   //   is no writers
}
else if (wWriters > 0) {            // if no readers
   wWriters--;                      //   but some writers
   Signal(Writer);                  //   release one
}
else
   Signal(Mutex);                   // no readers/writers
```

added component

Release a reader if there is no writer waiting.

Otherwise (i.e., no waiting readers or some waiting writers), release a writer.

Finally (i.e., no waiting readers and no waiting writers), do nothing.

A waiting reader is released if no writer is writing.

Passing the Baton: Example

- A **fair** version should allow readers and writers take turns (i.e., no starvation).
- We assume the semaphores being used is implemented so that every blocked thread will be eventually released (i.e., no starvation). This is difficult to achieve.
- A fair version must satisfy the following:
  1. When a writer finishes, all waiting readers get a turn
  2. When all current readers finish reading, one waiting writer can write.

Study the conditions for releasing readers and writers.

Passing the Baton: Example 22/23

- We need to change the last (exit) `if` of the writer.
- A new `Bool` variable `Writing` is need to indicate whether a writer is writing. `Writing` is set to `TRUE` when a writer starts writing and is set to `FALSE` when a reader starts reading.

```
if (wReaders > 0 & (wWriters = 0 | Writing)) {
   wReaders--;
   Signal(Reader);
}
else if (wWriters > 0 & (wReaders = 0 | ¬Writing)) {
   wWriters--;
   Signal(Writer);
}
else // wReaders = wWriters = 0
   Signal(Mutex);
```

NEW: no waiting **or** writing writer

a reader can read

This is set by the current exiting writer. Because it exits, no writer is writing.

NEW: no waiting readers **or** no writing writer

a writer can writer

Why is the starvation free assumption needed?

Passing the Baton: Example 23/23

- **This example, including its extensions and the passing the baton pattern, is due to Gregory R. Andrews. Refer to the following for more detailed discussions.**

1. **Gregory R. Andrews, *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, 1991.**
2. **Gregory R. Andrews, A Method for Solving Synchronization Problems, *Science of Computer Programming*, Vol. 13 (1989/1991), pp. 1-21.**

Semaphores with ThreadMentor

Semaphores with ThreadMentor

- **ThreadMentor** has a class `Semaphore` with two methods `Wait()` and `Signal()`.
- Class `Semaphore` requires a non-negative integer as an initial value.
- A name is optional.

```
Semaphore Sem("S",1);
Sem.Wait();
// critical section
Sem.Signal();

Semaphore *Sem;
Sem = new
   Semaphore("S",1);
Sem->Wait();
// critical section
Sem->Signal();
```

Dining Philosophers: 4 Chairs

```
Semaphore Chairs(4);
Mutex Chops[5]={1,..,1};

class phil::public Thread
{
  public:
    phil(int n, int it);
  private:
    int  Number;
    int  iter;
    void ThreadFunc();
};
```

Count-Down and Lock!

```
Void phil::ThreadFunc()
{
 int i, Left=Number,
       Right=(Number+1)%5;
 Thread::ThreadFunc();
 for (i=0; i<iter; i++) {
   Chairs.Wait();
       Chops[Left].Lock();
       Chops[Right].Lock();
       // Eat
       Chops[Left].Unlock();
       Chops[Right].Unlock();
   Chairs.Signal();
}
```

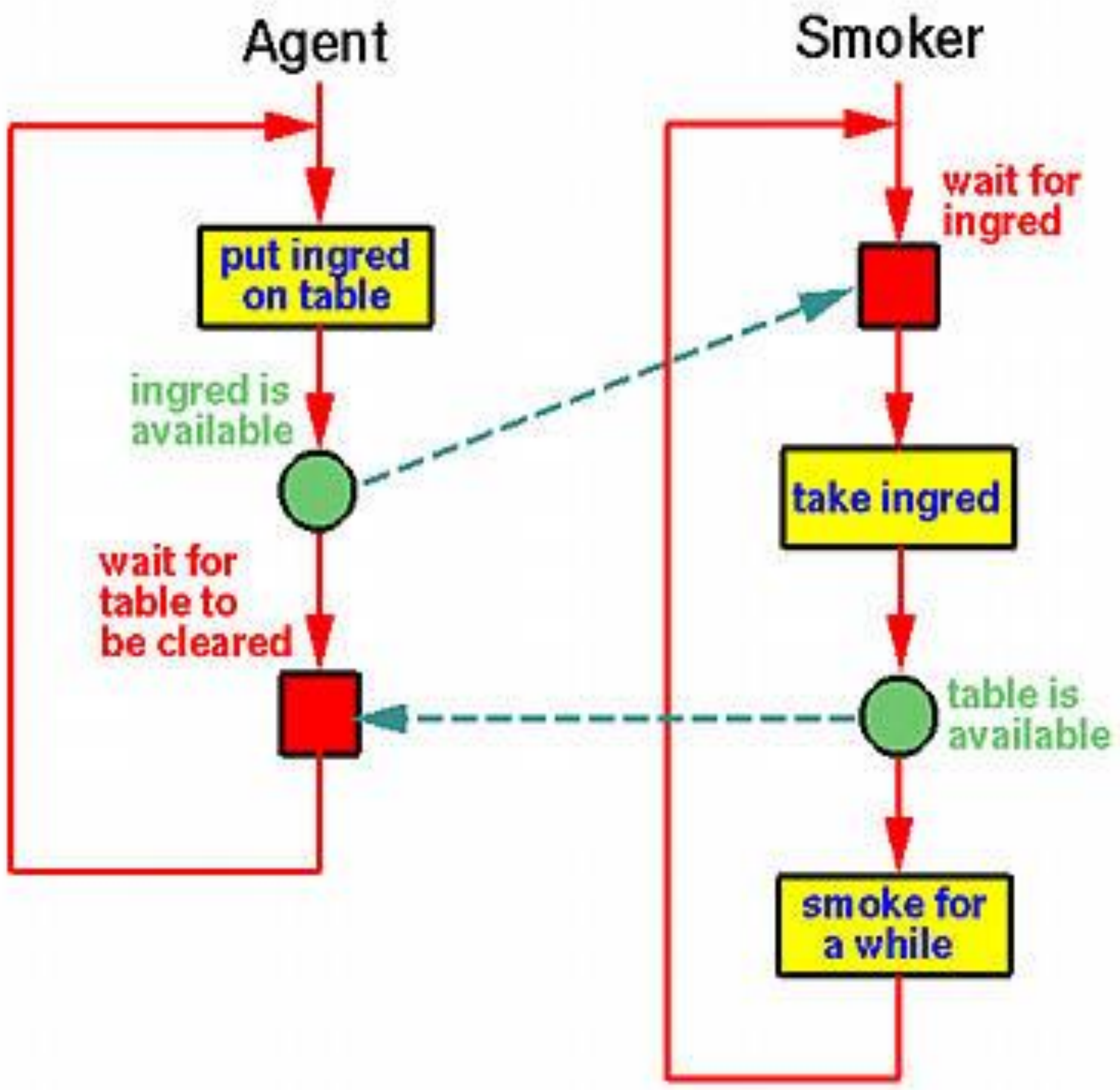
The Smokers Problem: 1/6

- **Three ingredients are needed to make a cigarette: tobacco, paper and matches.**
- **An agent has an infinite supply of all three.**
- **Each of the three smokers has an infinite supply of one ingredient only. That is, one of them has tobacco, the second has paper, and the third has matches.**
- **They share a table.**

The Smokers Problem: 2/6

- **The agent adds two randomly selected different ingredients on the table and notifies the needed smoker.**
- **A smoker waits until agent's notification. Then, takes the two needed ingredients, makes a cigarette, and smokes for a while.**
- **This process continues forever.**
- **How can we use semaphores to solve this problem?**

The Smokers Problem: 3/6



The Smokers Problem: 4/6

- Semaphore `Table` protects the table.
- Three semaphores `Sem[3]` are used, one for each smoker:

Smoker #	*Has*	*Needs*	*Sem*
0	0	1 & 2	`Sem[0]`
1	1	2 & 0	`Sem[1]`
2	2	0 & 1	`Sem[2]`

The Smokers Problem: 5/6

```
class A::public Thread
{
  private:
    void ThreadFunc();
};
```

agent thread

smoker thread

```
class Smk::public Thread
{
  public:
    Smk(int n);
  private:
    void ThreadFunc();
    int  No;
};
```

```
Smk::Smk(int n)
{
  No = n;
}

Void Smk::ThreadFunc()
{
  Thread::ThreadFunc();
  while (1) {
    Sem[No]->Wait();
    Table.Signal();
    // smoker a while
  }
}
```

waiting for ingredients

clear the table

The Smokers Problem: 6/6

```
void A::ThreadFunc()
{
  Thread::ThreadFunc();
  int  Ran;
  while (1) {
    Ran = // random #
          // in [0,2]
    Sem[Ran]->Signal();
    Table.Wait();
  }
}
```

ingredients are ready

waiting for the table to be cleared

```
void main()
{
  Smk *Smoker[3];
  A   Agent;

  Agent.Begin();
  for (i=0;i<3;i++) {
   Smoker = new Smk(i);
   Smoker->Begin();
  }
  Agent.Join();
}
```

The General Smokers Problem: 1/13

- **The original version of the Smokers problem was due to Suhas Patil in 1971. The agent and smokers have codes like the following:**

```
Semaphore Table = 1, Sem[3] = {0,0,0};
```

Agent

```
while (1) {
   generate ingredients i and j
   Table.Wait();
   Sem[i].Signal();
   Sem[j].Signal();
}
```

Smoker who needs m and n

```
while (1) {
   Sem[m].Wait();
   Sem[n].Wait();
   Table.Signal();
   smoke for a while
}
```

Suhas S. Patil, Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes, Project MAC, Computational Structures Group Memo 57, February 1971.

The General Smokers Problem: 2/13

- However, deadlock can happen.
- Use an execution sequence to reveal a possible deadlock.

```
Semaphore Table = 1, Sem[3] = {0,0,0};
```

Agent

```
while (1) {
   generate ingredients i and j
   Table.Wait();
   Sem[i].Signal();
   Sem[j].Signal();
}
```

Smoker who needs m and n

```
while (1) {
   Sem[m].Wait();
   Sem[n].Wait();
   Table.Signal();
   smoke for a while
}
```

The General Smokers Problem: 3/13

- **Solving this general smokers problem with the semaphores discussed here is tedious, but doable.**
- **David L. Parnas published a deadlock-free solution and Nico Habermann proved that Parnas' solution was correct.**
- **It is interesting to point out that E. W. Dijkstra and Suhas Patil proposed the parallel `Wait()` approach in 1971.**

David L. Parnas, On a Solution to the Cigarette Smoker's Problem (without conditional statements), *Communications of the ACM*, Vol. 18 (1975), No. 3, pp. 181-183.

Nico Habermann, On a Solution and a Generalization of the Cigarette Smokers' Problem, Computer Science Department, Carnegie-Mellon University, August 1972.

E. W. Dijkstra, Hierarchical Ordering of Sequential Processes, *Acta Informatica*, Vol. 1 (1971), pp. 115-138.

The General Smokers Problem: 4/13

- **Dijkstra proposed to extend the `Wait()` and `Signal()` calls to use multiple semaphores.**
- **A process calls $\texttt{Wait}_p(\texttt{S}_1, \texttt{S}_2, \ldots, \texttt{S}_n)$ does not wait if and only if every semaphore $\texttt{S}_i$ has a positive counter value. Each semaphore counter value is decreased by `1` as usual.**
- **$\texttt{Signal}_p(\texttt{S}_1, \texttt{S}_2, \ldots, \texttt{S}_n)$ is simple: adding `1` to the counter of each semaphore $\texttt{S}_i$. Of course, some waiting process may be released due to the change of the values of semaphore counters.**
- **Dijkstra calls this type of $\texttt{Wait}_p()$ parallel `Wait()`.**

The General Smokers Problem: 5/13

- A general form of semaphores, called *semaphore array*, was proposed by Tilak Agerwala in 1977.
- Let $S_1, S_2, \ldots, S_n$ and $\underline{T}_1, \underline{T}_2, \ldots, \underline{T}_m$ be semaphores.
- `Wait_e()` and `Signal_e()` are defined as follows:

```
Waite(S1,S2,…,Sn,T1,T2,…,Tm)
{
    if (for all i in [1,n]: Si > 0 and for all j in [1,m]: Tj = 0)
        for all i in [1,n]: Si--;
    else
        the caller process/thread is blocked
}

Signale(S1,S2,…,Sn)
{
    for all i in [1,n]: Si++;
}
```


Tilak Agerwala, Some Extended Semaphore Primitives, *Acta Informatica*, Vol. 8 (1977), pp. 201-220.

The General Smokers Problem: 6/13

- Counters of Semaphores S_1, S_2, …, S_n are tested and decreased at the same time.
- Semaphores $\underline{T}_j$'s are semaphores T_j's used for 0 testing.

```
Wait_e(S_1,S_2,…,S_n,T_1,T_2,…,T_m)
{
   if (for all i in [1,n]: S_i > 0 and for all j in [1,m]: T_j = 0)
      for all i in [1,n]: S_i--;
   else
      the caller process/thread is blocked
}

Signal_e(S_1,S_2,…,S_n)
{
   for all i in [1,n]: S_i++;
}
```


Tilak Agerwala, Some Extended Semaphore Primitives, *Acta Informatica*, Vol. 8 (1977), pp. 201-220.

The General Smokers Problem: 7/13

- If the semaphores $\underline{T}_1$, $\underline{T}_2$, …, $\underline{T}_m$ are not used, Agerwala's extension becomes Dijkstra's **parallel** `Wait()`.

```
Wait_e(S_1,S_2,…,S_n)
{
    if (for all i in [1,n]: S_i > 0)
        for all i in [1,n]: S_i--;
    else
        the caller process/thread is blocked
}

Signal_e(S_1,S_2,…,S_n)
{
    for all i in [1,n]: S_i++;
}
```


Tilak Agerwala, Some Extended Semaphore Primitives, *Acta Informatica*, Vol. 8 (1977), pp. 201-220.

The General Smokers Problem: 8/13

- If semaphores S_i are not used, this extension becomes waiting for all the counters of $\underline{T}_1$, $\underline{T}_2$, …, $\underline{T}_m$ to become zero.
- Any semaphore S can be use for 0 testing and is denoted as $\underline{S}$.

```
Waite(T1,T2,…,Tm)
{
    if (for all j in [1,m]: Tj = 0)
        do nothing
    else
        the caller process/thread is blocked
}
```


Tilak Agerwala, Some Extended Semaphore Primitives, *Acta Informatica*, Vol. 8 (1977), pp. 201-220.

The General Smokers Problem: 9/13

- **The philosophers problem can be solved very easily.**
- **Show that this solution is deadlock-free.**

Philosophger i

```
Semaphore C[5] = { 1, 1, 1, 1, 1 };

while (1) {
    // thinking
    Wait_e(C[(i+4)%5], C[(i+1)%5]);
    // eating
    Signal_e(C[(i+4)%5], C[(i+1)%5]);
}
```

The General Smokers Problem: 10/13

- **The general smokers problem can also be solved very easily.**
- **Show that this solution is deadlock-free.**

Agent

```
Semaphore Table = 1;
Semaphore Ingred[3] = { 0, 0, 0 };

while (1) {
   Wait(Table);
   // generate two ingredients i and j
   Signal_e(Ingred[i], Ingred[j]);
}
```

Smoker *k* who needs ingredients *m* and n

```
while (1) {
   Wait_e(Ingred[m], Ingred[n]);
   Signal_e(Table);
   // smoke for a while
}
```

The General Smokers Problem: 11/13

- How do we use the T semaphores? Here is an example.
- Process P_i has higher priority than process P_{i+1} ($0 \leq i \leq n-2$). There are n processes.
- The processes request access to the resource and are allocated in a mutually exclusive way based in the priorities.
- A request by a process is not honored until all higher priority requests are taken care of.

The General Smokers Problem: 12/13

- Process P_i sets its semaphore to 1, making a request.
- P_i waits for higher priority processes' semaphores $\underline{S}_1$, $\underline{S}_2$, …, $\underline{S}_{i-1}$ to become 0, withdraws its request, and accesses the resource via semaphore R.

Process i

```
Semaphore S[n] = { 0, 0, …, 0 };
Semaphore R    = 1; // for mutual exclusion

while (1) {
   Signal_e(S[i]);    // make a request
   Wait_e(R,S[1],S[2],…,S[i-1]); // wait until all higher
                                //   priority processes done
                                //   and R is available
   Wait_e(S[i]);                // withdraw my request
   USE RESOURCE                 // use the resource.  Still has the Mutex
   Signal_e(R);                 // release the Mutex
 }
```

The General Smokers Problem: 13/13

- Unix/Linux supports semaphore arrays like Agerwala's semaphore extension.
- Use `semget()` to obtain a semaphore set using a key, which is similar to shared memory.
- Use `semop()` to operate on a semaphore set.
- For each semaphore element in a semaphore set, the following are permitted:
  1. There is no operation
  2. A value can be added or subtracted from the semaphore counter.
  3. A zero means waiting for 0.
- Refer to Unix/Linux manual for the details.

Food for Thought: 1/2

- **Dijkstra suggested the following solution to the philosophers problem.**

Global Items

```
Semaphore  Mutex = 1;
Semaphore  toEat[5] = { 0, 0, 0, 0, 0 };
int State[5] = { THINKING, THINKING, THINKING, THINKING, THINKING};

void CanEat(i)
{  // if i's left and right neighbors are not eating
   //      but i is hungry, then i can eat.
   if (State[(i+4)%5] != EATING
          && State[i] == HUNGRY && State[(i+1)%5] != EATING))
      Signal(toEat(i));  // signal i to eat
}
```


E. W. Dijkstra, Hierarchical Ordering of Sequential Processes, *Acta Informatica*, Vol. 1 (1971), pp. 115-138.

Food for Thought: 2/2

- **Dijkstra suggested the following solution to the philosophers problem.**

Philosopher *i*

```
Wait(Mutex);                // lock the Mutex to change state
   State[i] = HUNGRY;       // i is hungry
   CanEat(i);               // Can i eat? CanEat() is called in a Mutex
Signal(Mutex);

Wait(toEat[i]);             // if i can eat, it was signaled in CanEat()
   State[i] = EATING;       //    state changed to EATING
   // eat

Wait(Mutex);                // after eating, change state again
   State[i] = THINKING;     // i is thinking
   CanEat((i+4)%5);         // allow left neighbor to eat
   CanEat((i+1)%5);         // allow right neighbor to eat
Signal(Mutex);
```


E. W. Dijkstra, Hierarchical Ordering of Sequential Processes, *Acta Informatica*, Vol. 1 (1971), pp. 115-138.

The End