Part III

Synchronization

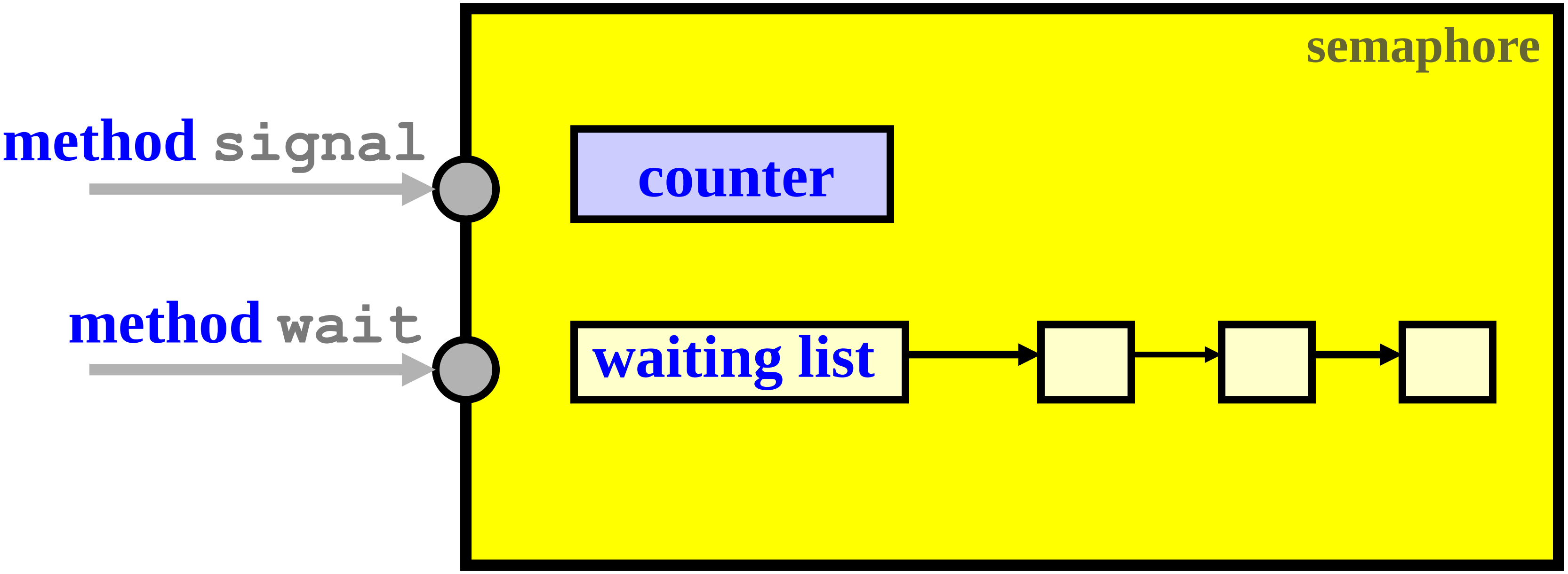
Semaphores

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 list of processes, 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 list;
        block();
    }
}
```

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

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 value is not positive (e.g., non-negative), 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 (e.g., 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 block 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--;
if (S.count<0) {
    add to list;
    block();
}
```

```
S.count++;
if (S.count<=0) {
    remove P;
    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.

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
```

Process 1		Process 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 or 1).

Use 2: Count-Down Counter

```
semaphore  S = 3;
```

Process 1

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

at most 3 processes can be here!!!

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

Process 2

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

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

notify notify notify

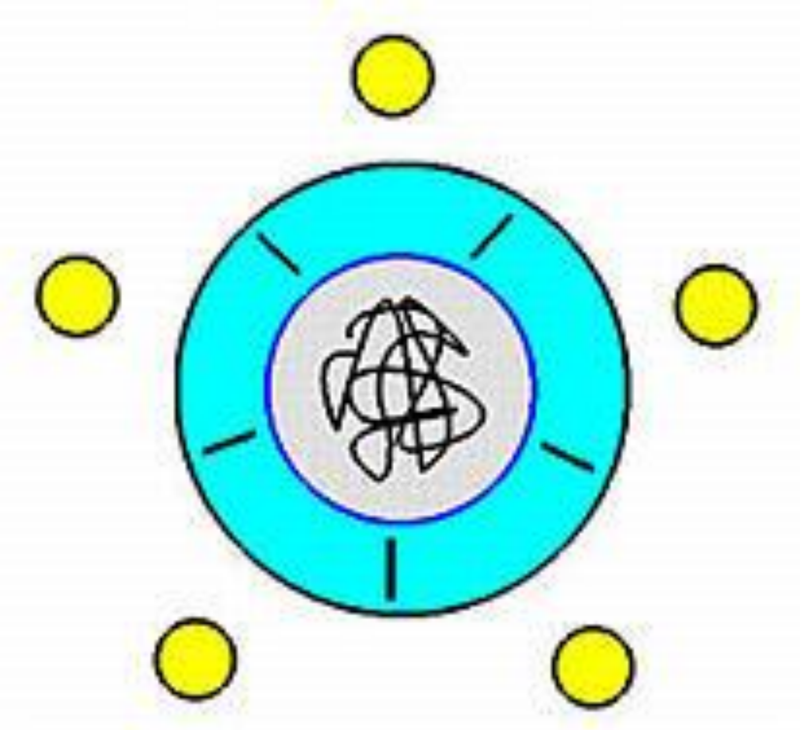
process 2

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

- **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 `S1` and `S2` are both 0's or both 1's?**
- **What if `S1` = 0 and `S2` = 1?**

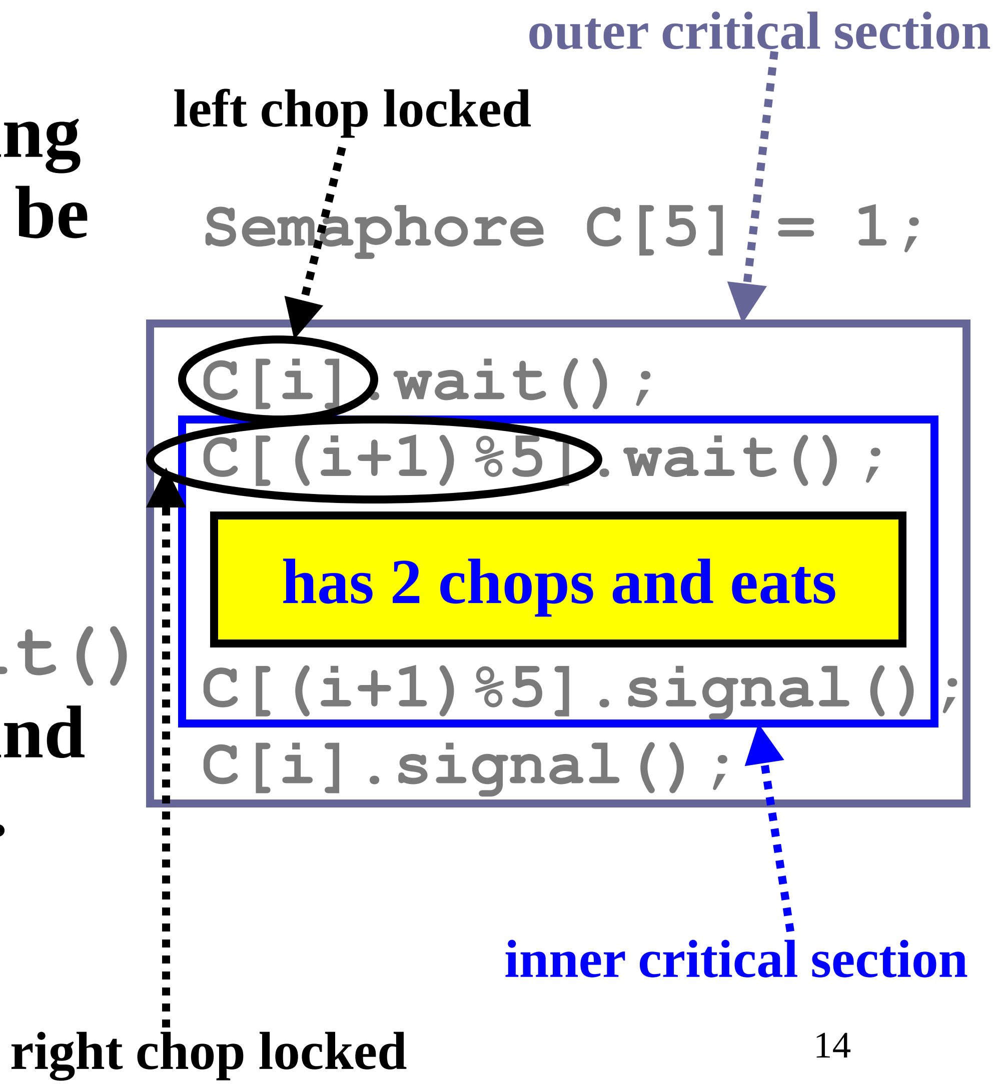
Dining Philosophers

- **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.**



Dining Philosopher: Ideas

- **Chopsticks are shared items (by two neighboring philosophers) and must be protected.**
- **Each chopstick has a semaphore with initial value 1 (i.e., available).**
- **A philosopher calls `wait()` to pick up a chopstick and `signal()` to release it.**



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

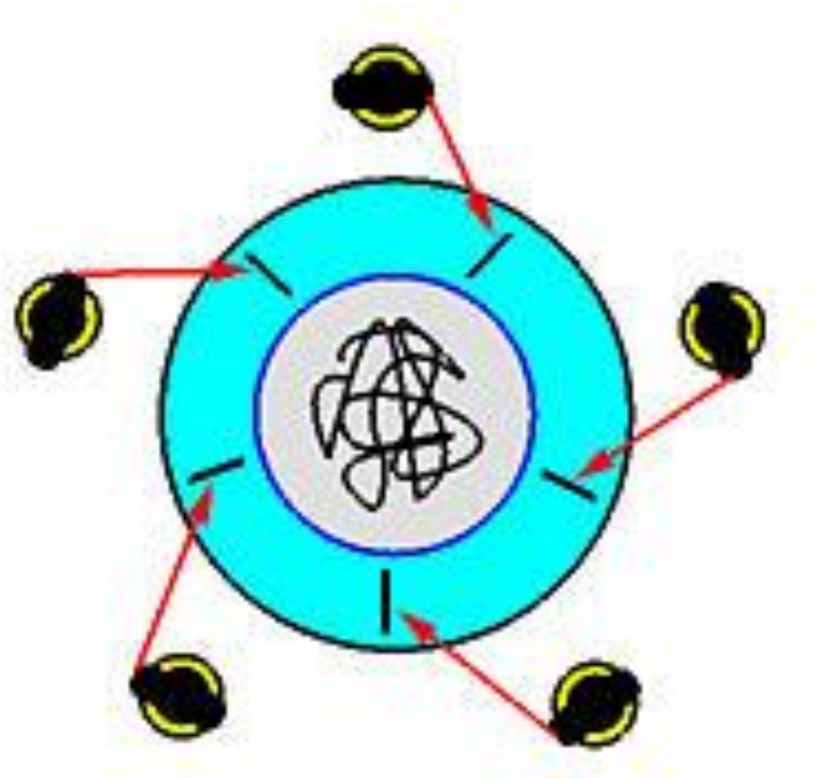
release my right chop

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 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

```
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
}
```

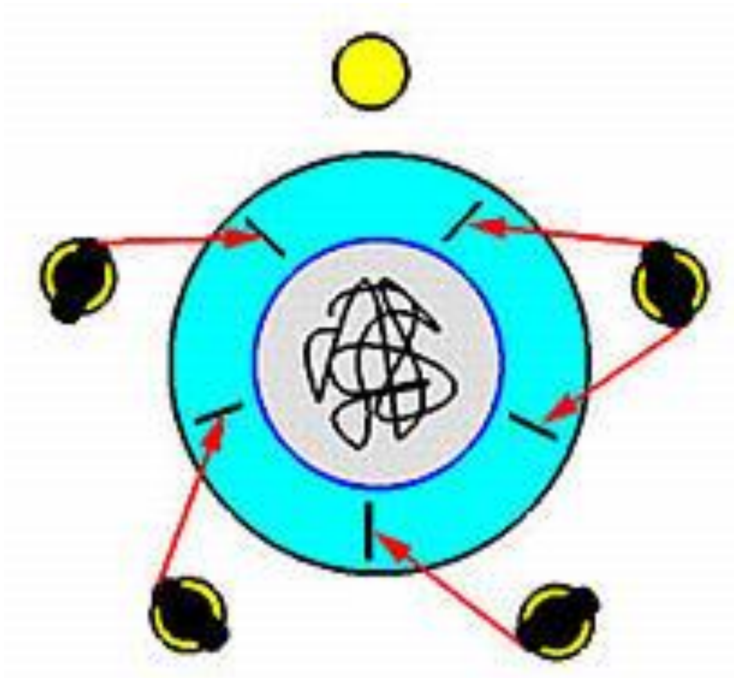
lock left chop

lock right chop

Dining Philosophers: Questions

- The following are some important questions for you to think about.
  - 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 does not cause **circular waiting** even if we have more than 1 and less than 5 weirdoes.
- This solution is not **symmetric** because not all threads run the same code.

Count-Down Lock Example



- The naïve solution to the dining philosophers problem causes circular waiting.
- If only four philosophers are allowed to sit down, deadlock cannot occur.
- Why? If all four of them sit down at the same time, the right-most philosopher may 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();
}
```

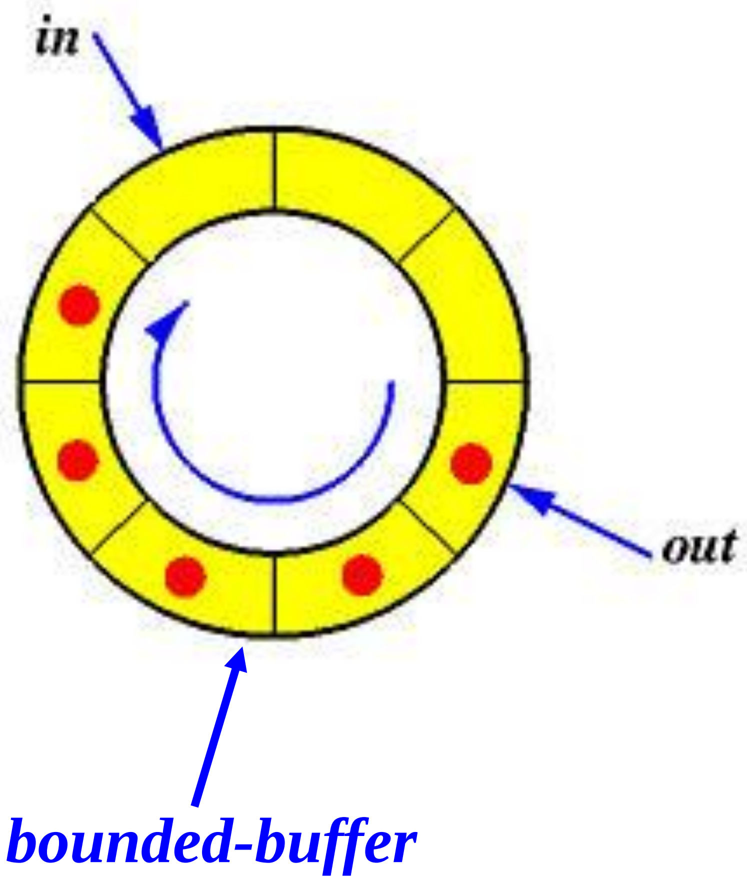
get a chair

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

this is our old friend

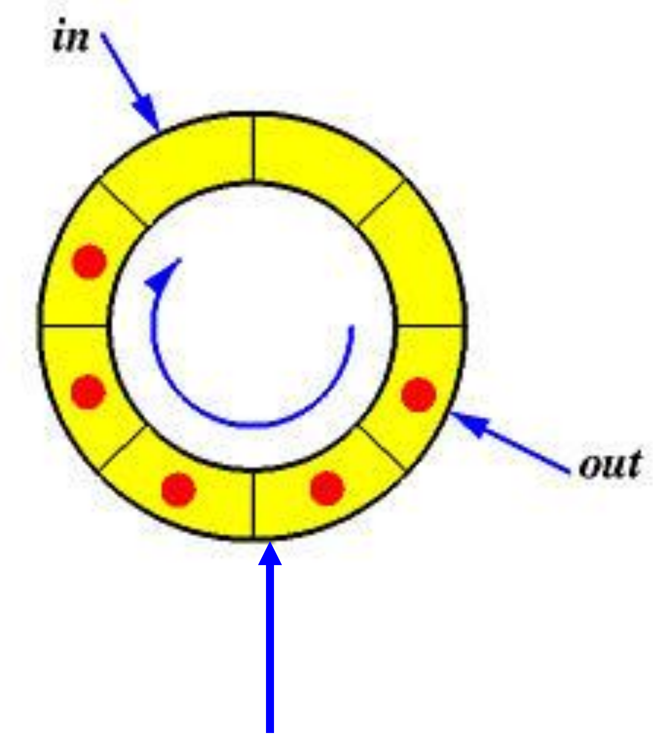
release my chair

The Producer/Consumer Problem



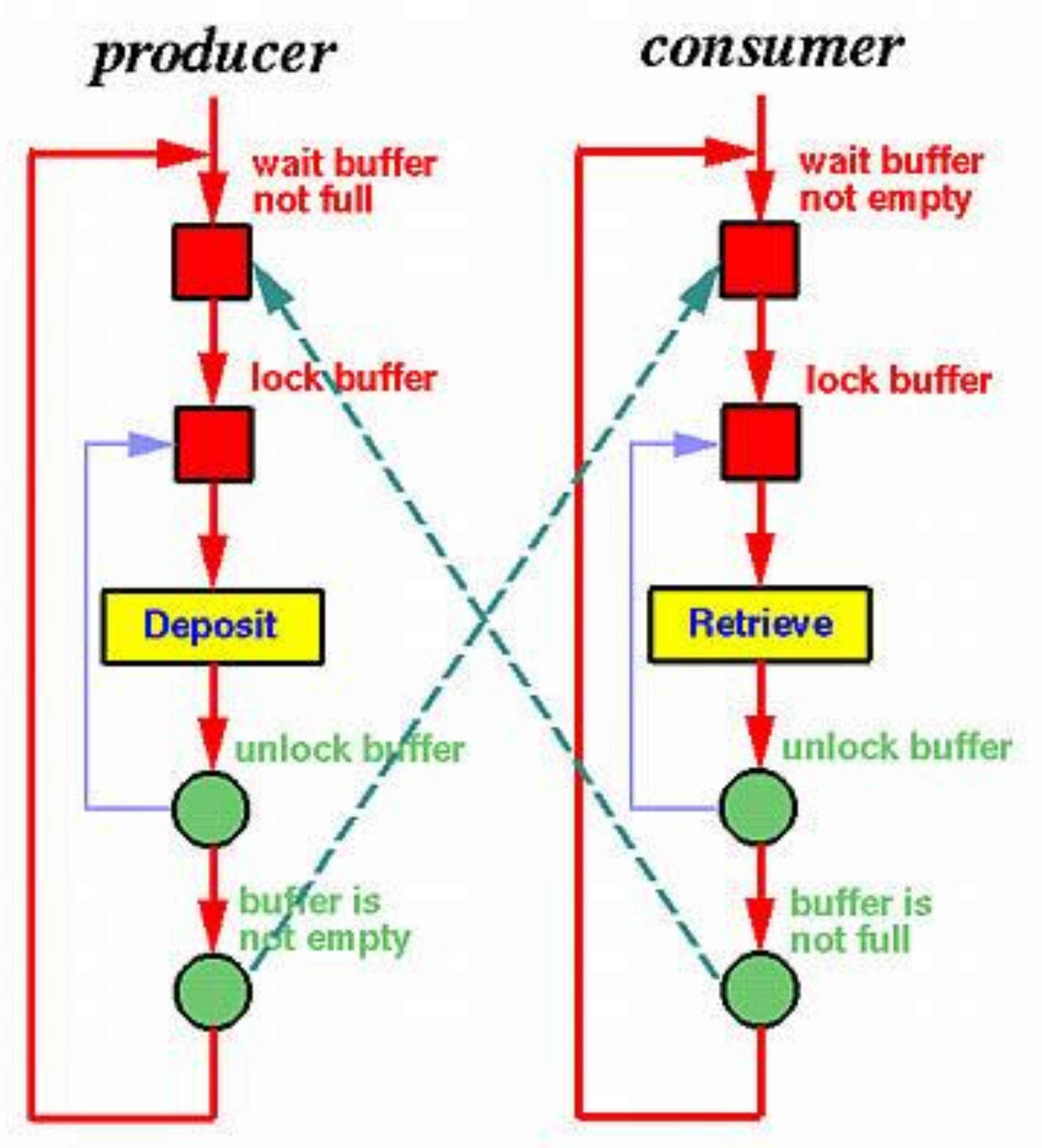
- Suppose we have a **circular buffer** of ***n*** slots.
- Pointer ***in*** (*resp*., ***out***) points to the first **empty** (*resp*., **filled**) slot.
- **Producer** processes keep adding data into the buffer.
- **Consumer** processes keep retrieving data from the buffer.

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.
- The second semaphore to block producers if the buffer is full.
- The third semaphore to block consumers if the buffer is empty.

Solution

number 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

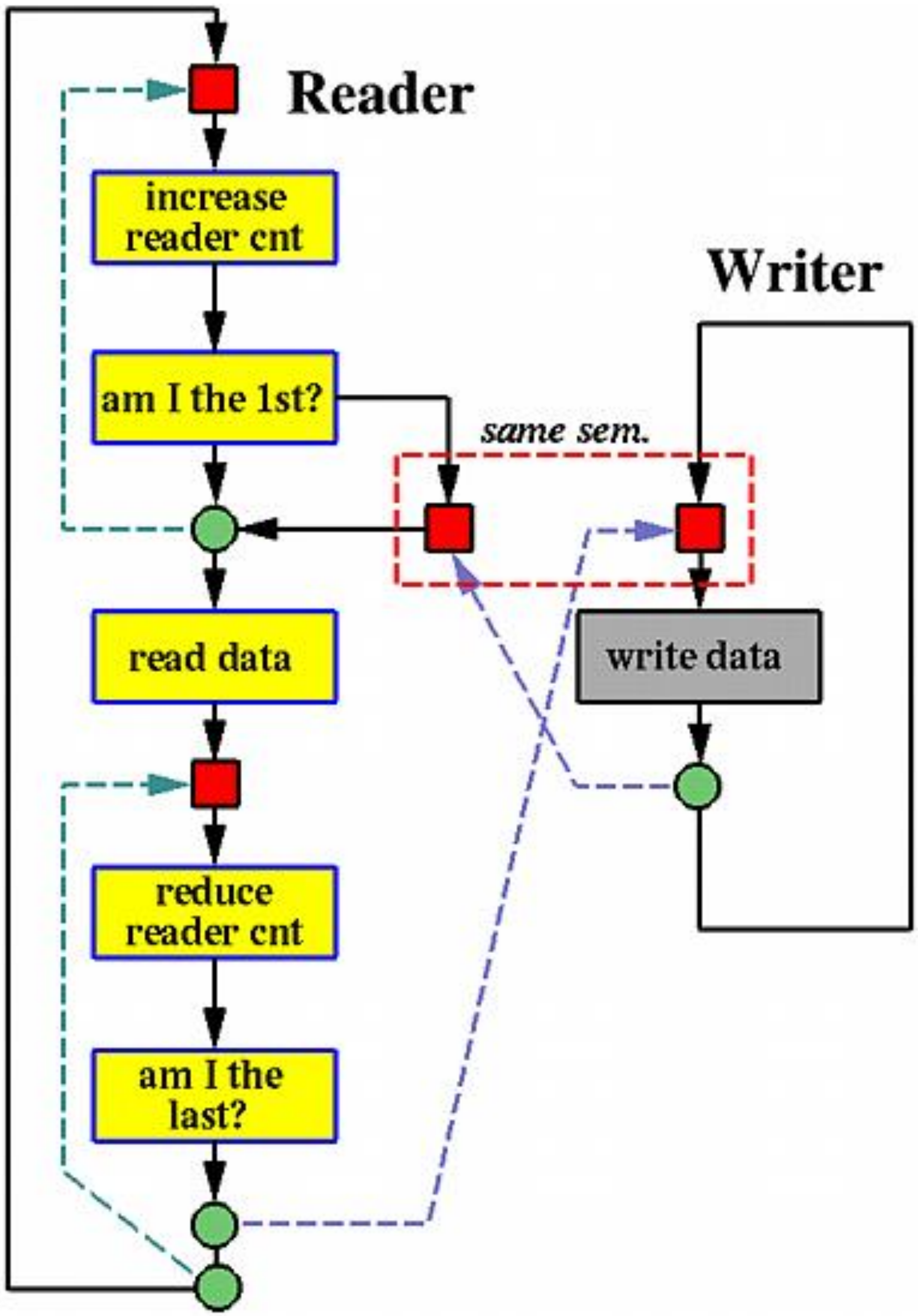
```
while (1) {
  Mutex.wait();
    NotFull.wait();
      Buf[in] = x;
      in = (in+1)%n;
    NotEmpty.signal();
  Mutex.signal();
}
```

The Readers/Writers Problem

- **Two groups of processes, readers and writers, 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 higher priority.**

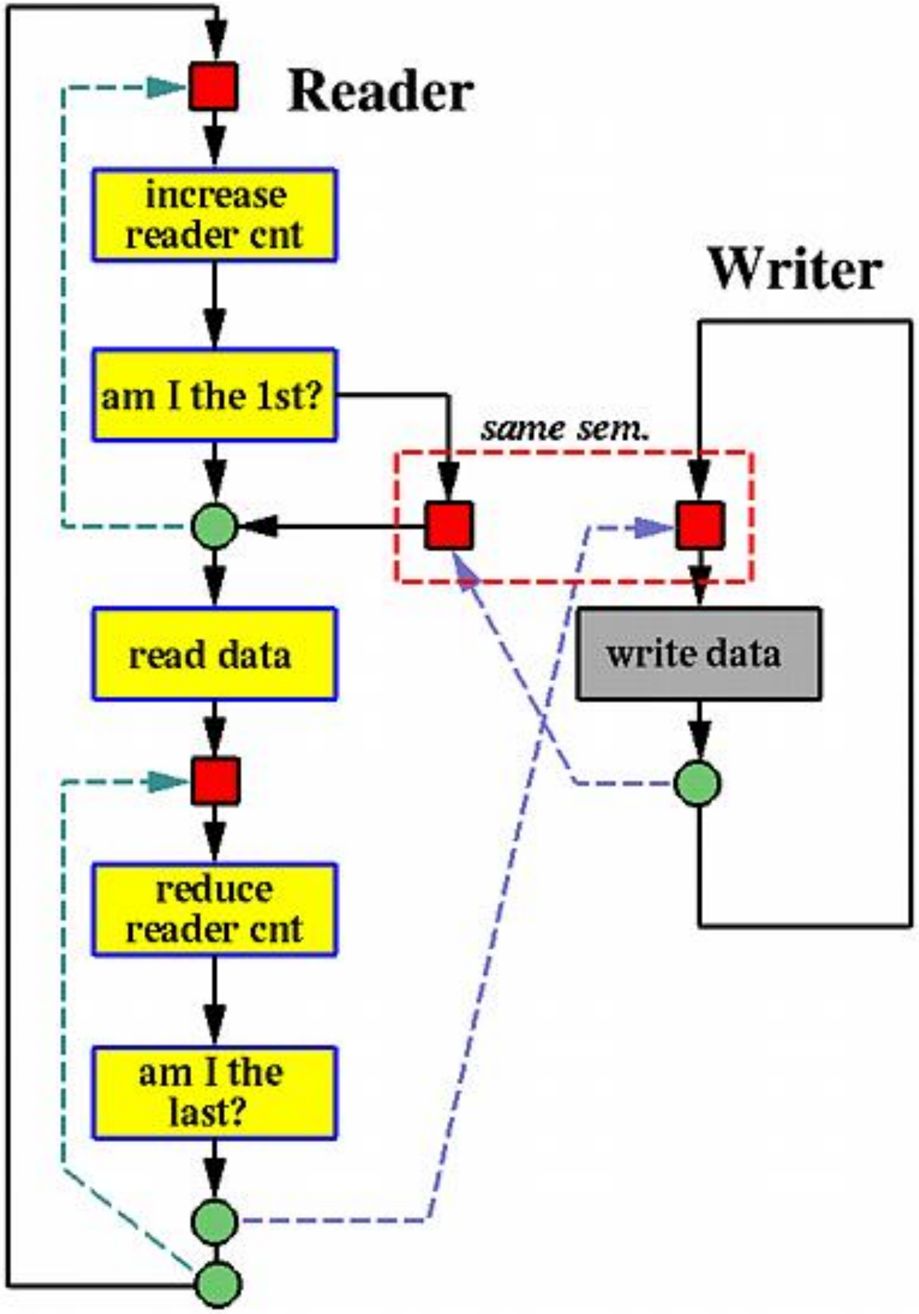
Problem Analysis

- **We need a semaphore to block readers 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: the bounded waiting condition may be violated if readers keep coming, causing the waiting writers 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, notifies the waiting readers and writers that no reader is reading.



Writer

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

blocks both readers and writers

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

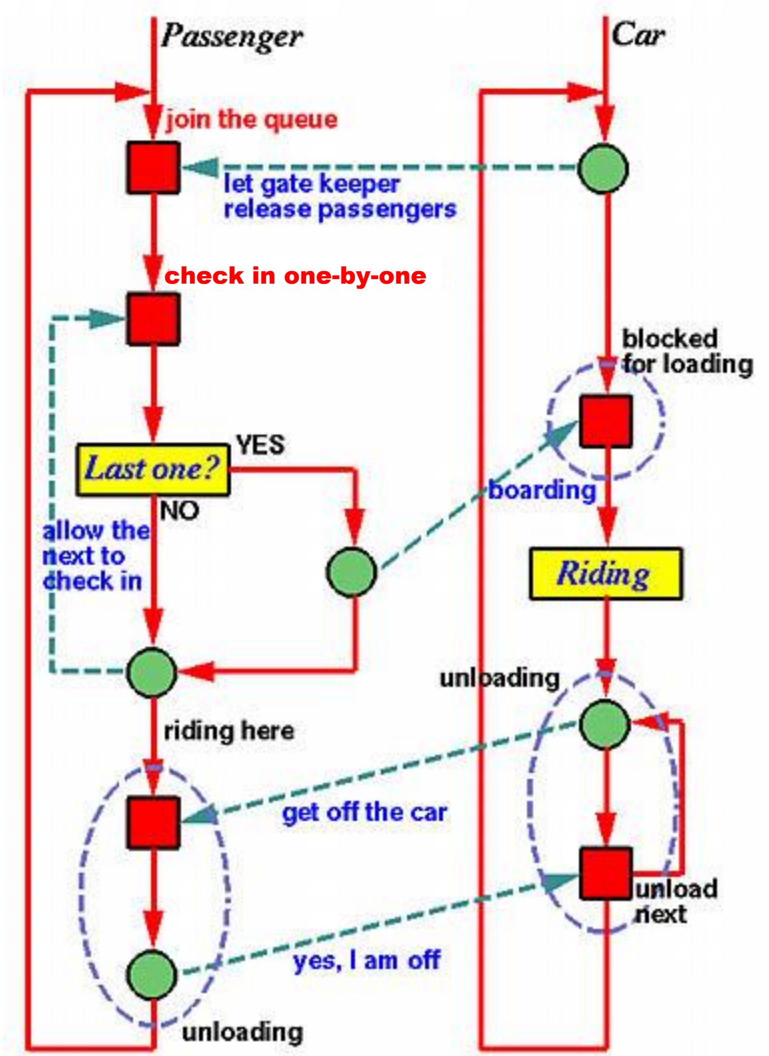
The Roller-Coaster Problem: 1/5

- **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 wanders around the amusement park before returning to the roller coaster for another ride.**
- **Due to safety reasons, the car only rides *T* times and then shut-down.**

The Roller-Coaster Problem: 2/5

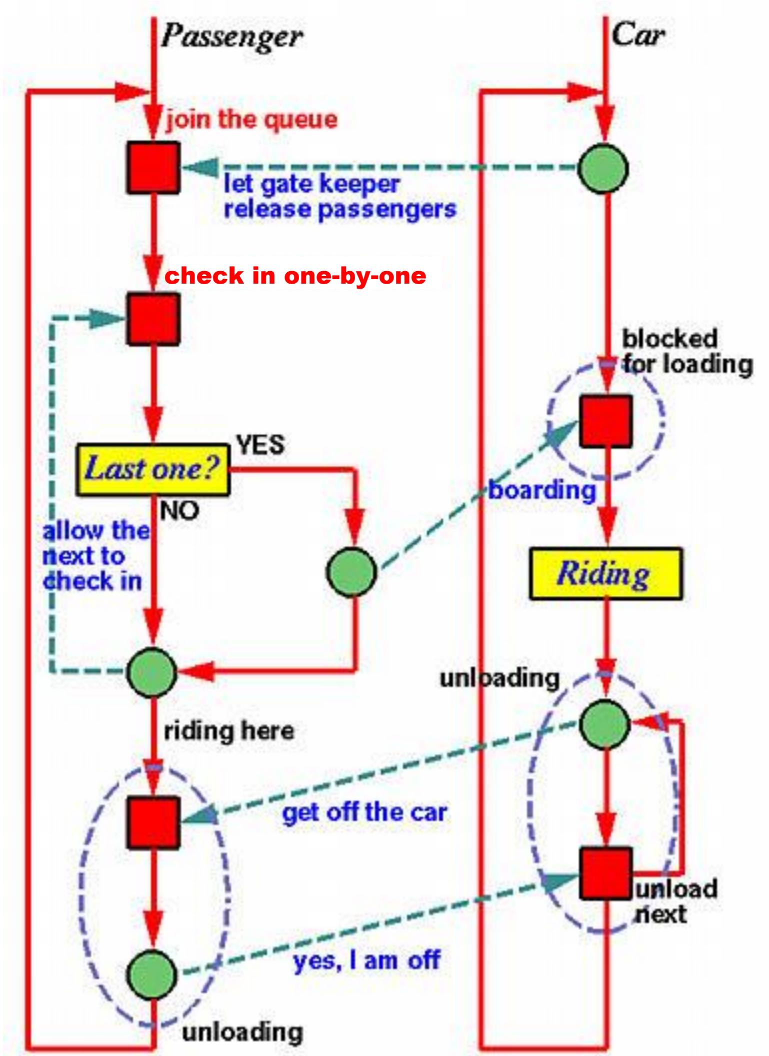
- **The car always rides with exactly *C* passengers**
- **No passengers will jump off the car while the car is running**
- **No passengers will jump on the car while the car is running**
- **No passengers will request another ride before they get off the car.**

The Roller-Coaster Problem: 3/5



- A passenger makes a decision 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.
- They go back to play for a while and come back for a ride.

The Roller-Coaster Problem: 4/5



- 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 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: 5/5

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

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

Unload passengers one-by-one

Is this absolutely necessary?

Can `Unloading` be removed? **Ex.**

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];

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();
}
```

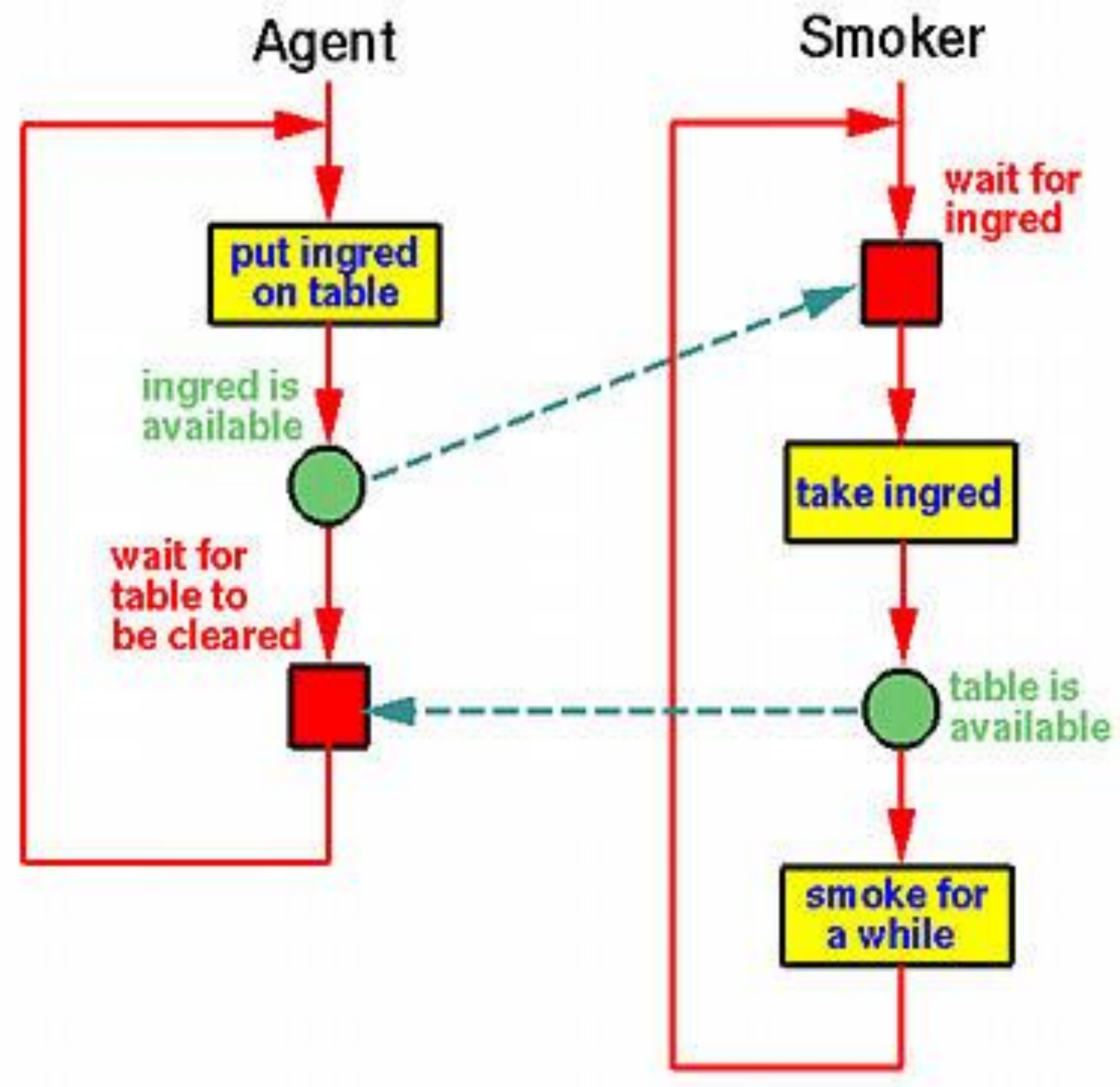
The Smoker Problem: 1/6

- **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

agent thread

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

smoker thread

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

clear the table

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

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

waiting for ingredients

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 End