北京交通大學

Classic Problems of Synchronization

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- The three problems are important, because they are
  - examples for a large class of concurrency-control problems.
  - used for testing nearly every newly proposed synchronization scheme.
- Semaphores are used for synchronization in our solutions.

Producer-consumer problem with a bounded buffer

- **Problem Definition**
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- **Put a fixed-size buffer between them**
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- **Example: Coke machine**
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty

Bounded-Buffer

- Suppose that we wanted to provide a solution to the consumer-producer problem with a bounded-buffer.
- We can do so by having an integer count that keeps track of the number of products in the buffer.
- Initially, count is set to 0.
    - It is incremented by the producer after it produces a new product.
    - It is decremented by the consumer after it consumes a product.

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;
```

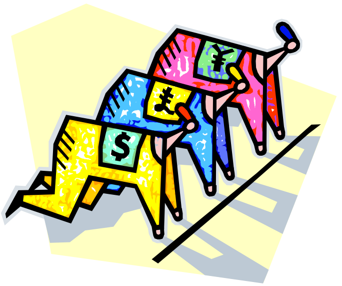
Producer and Consumer

```
while (true) {
    // produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

```
while (true)  {
    while (count == 0)
        ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    //consume the item in nextConsumed
}
```

Producer and Consumer

```
while (true) {
    while (count == BUFFER_SIZE)
      ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

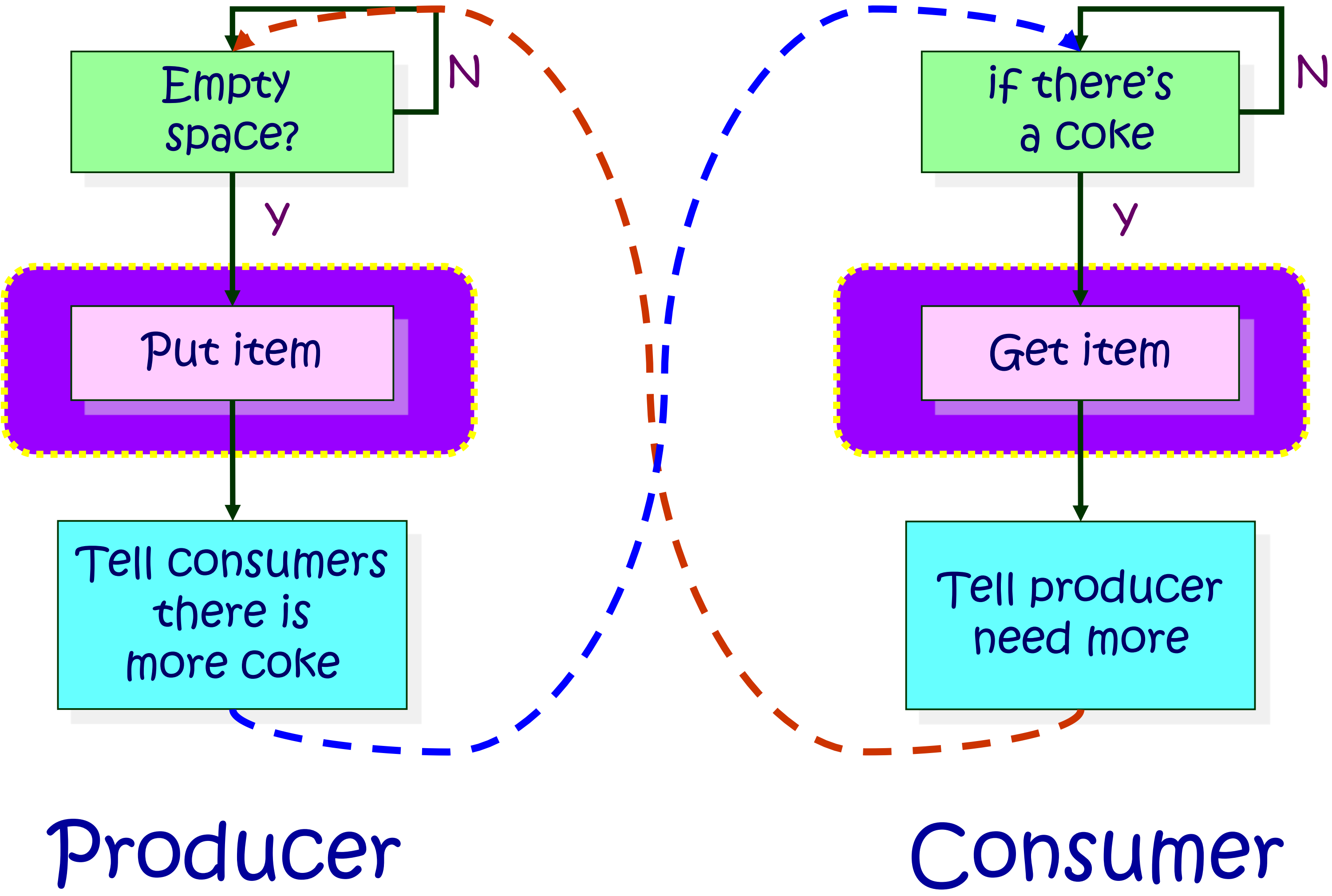


```
while (true)  {
    while (count == 0)
      ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
}
```

Correctness constraints for solution

- **Correctness Constraints:**
  - Consumer must wait for producer to fill the buffer, if no product in the buffer (scheduling constraint)
  - Producer must wait for consumer to empty the buffer, if the buffer is full (scheduling constraint)
  - Only one process can manipulate the buffer at a time (mutual exclusion)

Correctness constraints for solution



Correctness constraints for solution

- General rule of thumb: Use a separate semaphore for each constraint
  - `Semaphore full;`
    `// consumer's constraint`
    `// initialized to the value 0`
  - `Semaphore empty;`
    `// producer's constraint`
    `// initialized to the value N.`
  - `Semaphore mutex;`
    `// mutual exclusion`
    `// initialized to the value 1`

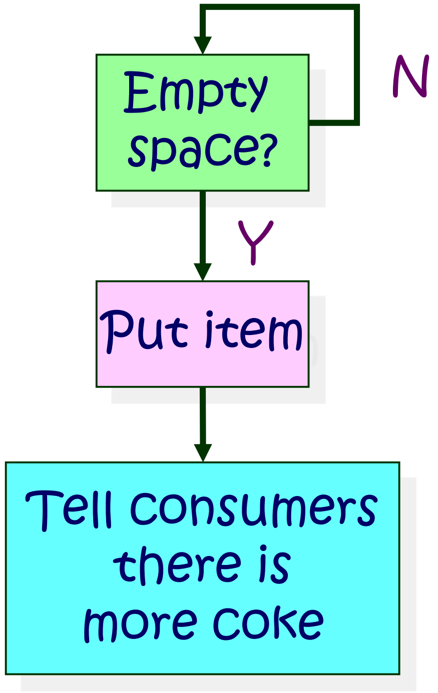
Full Solution

```
Semaphore full = 0;          // Initially, no coke
Semaphore empty = N;         // Initially, num empty slots
Semaphore mutex = 1;         // No one using machine

Producer(item) {
  P(empty);                  // Wait until space
  P(mutex);                  // Wait until machine free
  Enqueue(item);
  V(mutex);
  V(full);                   // Tell consumers there is more coke
}

Consumer() {
  P(full);                   // Check if there's a coke
  P(mutex);                  // Wait until machine free
  item = Dequeue();
  V(mutex);
  V(empty);                  // Tell producer need more
  return item;
}
```

Full Solution

- The structure of the producer process

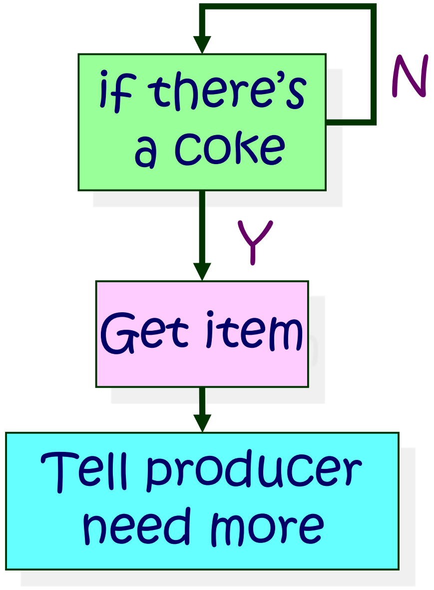
```
while (true) {

     //   produce an item

     wait (empty);
     wait (mutex);

     //  add the item to the buffer

     signal (mutex);
     signal (full);
}
```



Full Solution

- The structure of the consumer process

```
while (true) {

    wait (full);
    wait (mutex);

    //  remove an item from buffer

    signal (mutex);
    signal (empty);

   //  consume the removed item
}
```



Discussion about Solution

- Why asymmetry?
  - Producer does: `P(empty), V(full)`
  - Consumer does: `P(full), V(empty)`
- Is the order of P's important?
- Is the order of V's important?

```
Producer(item) {
   P(empty);
   P(mutex);
   Enqueue(item);
   V(mutex);
   V(full);
}

Consumer() {
   P(full);
   P(mutex);
   item = Dequeue();
   V(mutex);
   V(empty);
   return item;
}
```

Discussion about Solution

Is the order of P's important?

- Yes! Can cause deadlock
- Why?

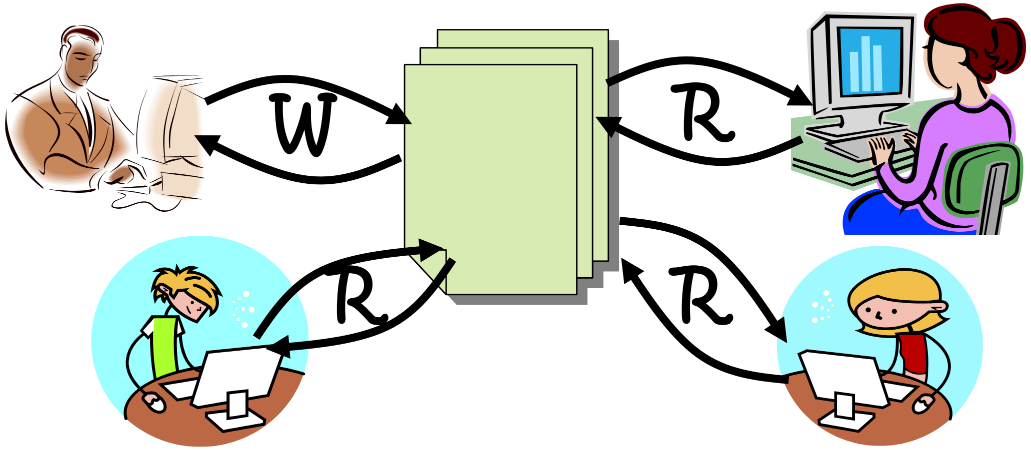
```
Semaphore full = 0;          // Initially, no coke
Semaphore empty = N;         // Initially, num empty slots
Semaphore mutex = 1;         // No one using machine

Producer(item) {
    P(mutex);                // Wait until buffer free
    P(empty);                // Wait until space
    Enqueue(item);
    V(mutex);
    V(full);                 // Tell consumers there is
                             // more coke
}

Consumer() {
    P(full);                 // Check if there's a coke
    P(mutex);                // Wait until machine free
    item = Dequeue();
    V(mutex);
    V(empty);                // tell producer need more
    return item;
}
```

Discussion about Solution

- **Is order of V's important?**
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

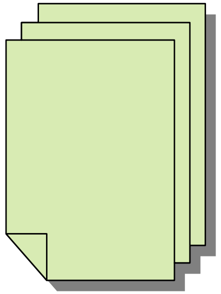
Readers/Writers Problem

- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – only read the data set; they do not perform any updates.
    - Writers – can both read and write.
  - Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.



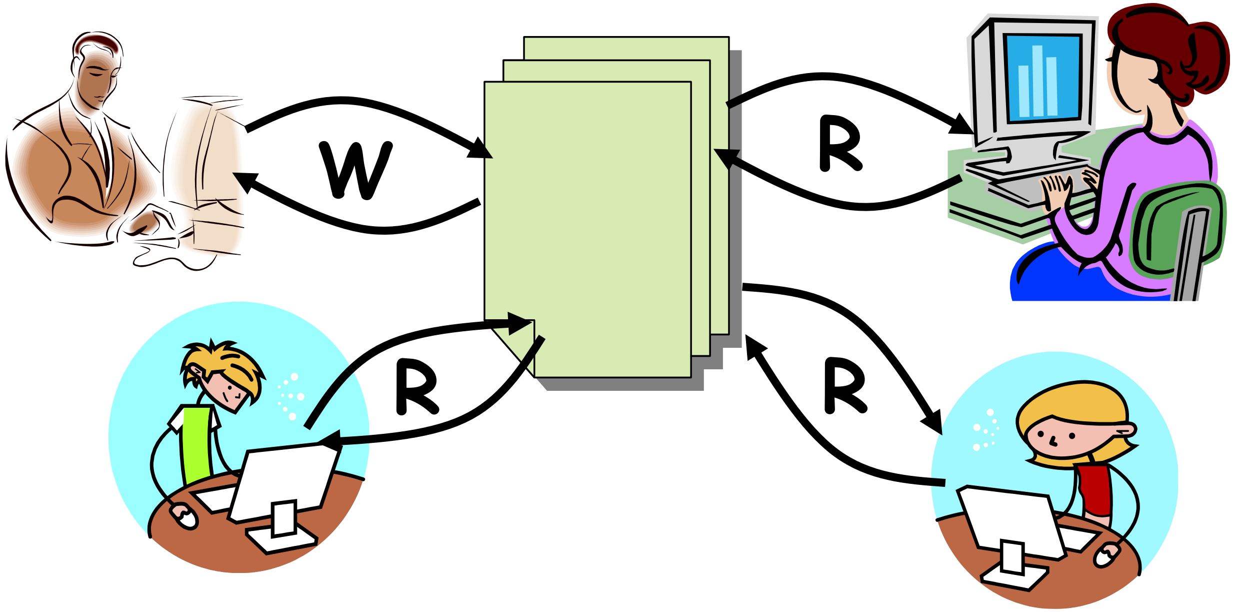
Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one process manipulates state variables at a time
- Basic structure of a solution:
  - `Reader()`
    ```
    Wait until no writers
    Access database
    Check out – wake up a waiting writer
    ```
  - `Writer()`
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writers
    ```



Readers/Writers Problem

- Is using a single lock on the whole database sufficient?



The first readers-writers problem

- Suppose we have a shared memory area with the constraints detailed above.
- It is possible to protect the shared data behind a mutex, in which case no process/thread can access the data at the same time.

The first readers-writers problem

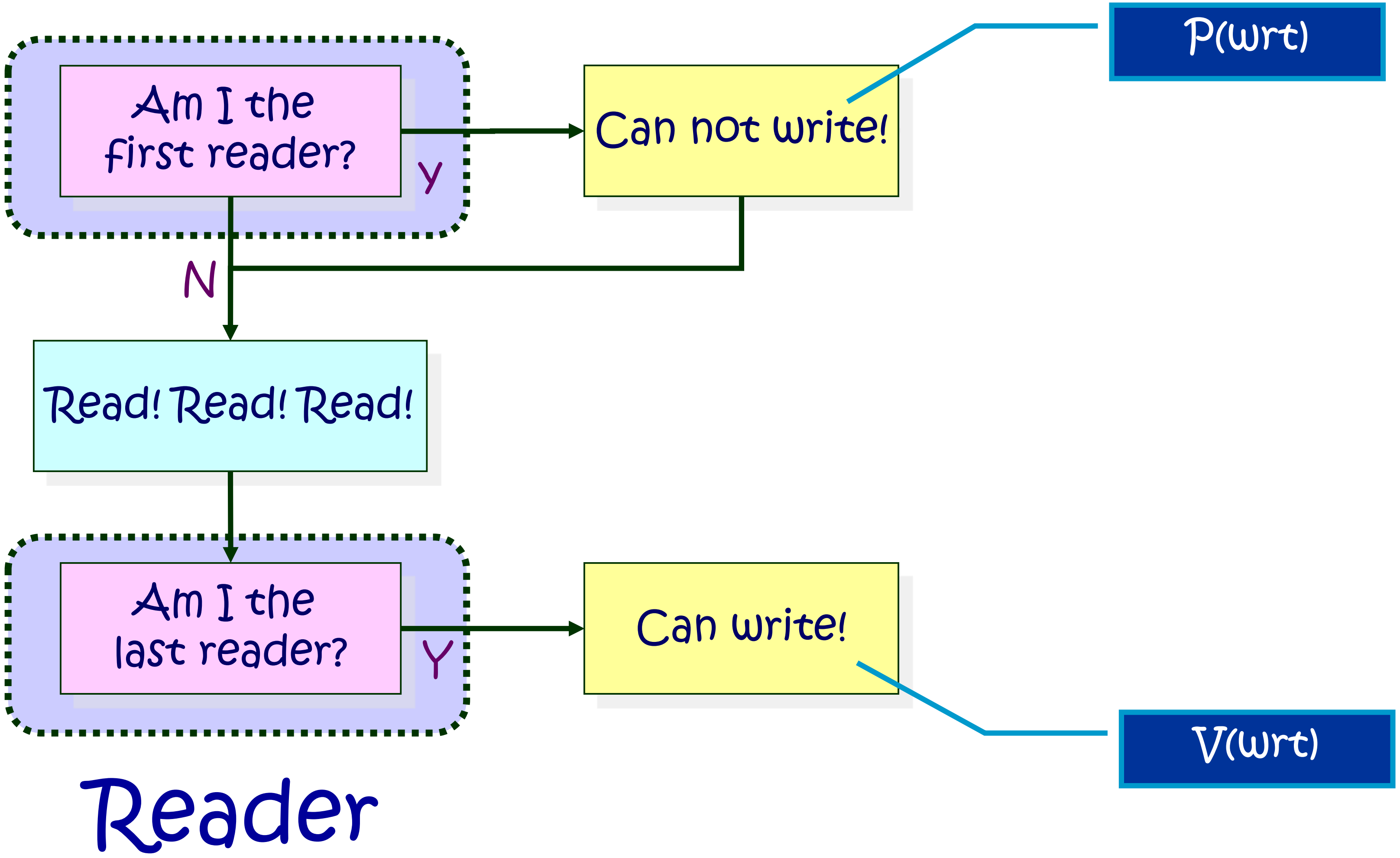
- The solution is sub-optimal.
  - Because it is possible that a reader R1 might have the lock, and then another reader R2 request access.
  - It would be foolish for R2 to wait until R1 was done before starting its own read operation.
  - Instead, R2 should start right away.
- This is the motivation for the first readers-writers problem, in which the constraint is added that no reader shall be kept waiting if the share is currently opened for reading.
- This is also called **readers-preference**.

Readers-Writers Problem: Solution #1

- int readcount = 0;
- semaphore mutex = 1, wrt = 1;
- The structure of a writer process

```
while (true) {
        wait (wrt);
        // writing is performed
        signal (wrt);
}
```

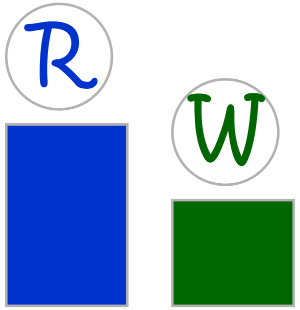
Readers-Writers Problem: Solution #1



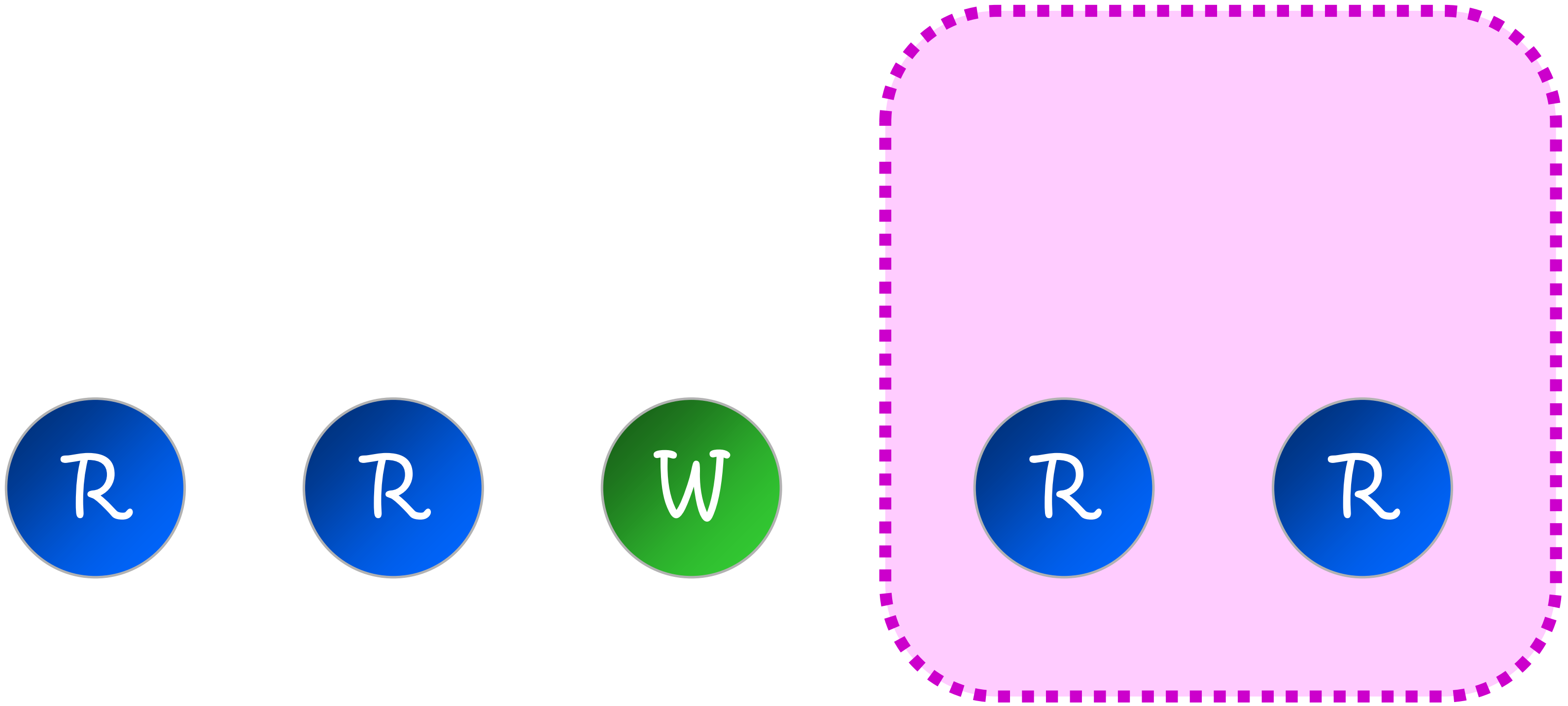
Readers-Writers Problem: Solution #1

- The structure of a reader process

```
while (true) {
     wait(mutex);
        readcount++;
        if (readcount == 1)  wait(wrt);
     signal(mutex);

        // reading is performed

     wait(mutex);
        readcount--;
        if (readcount == 0)  signal(wrt);
     signal(mutex);
}
```



The first readers-writers problem



The second readers-writers problem

- The former solution is sub-optimal.
  - Because it is possible that a reader R1 might have the lock, a writer W be waiting for the lock, and then a reader R2 request access.
  - It would be foolish for R2 to jump in immediately, ahead of W; if that happened often enough, W would **starve**.
  - Instead, W should start as soon as possible.
- This is the motivation for the second readers-writers problem, in which the constraint is added that no writer, once added to the queue, shall be kept waiting longer than absolutely necessary.
- This is also called **writers-preference**.

Readers-Writers Problem: Solution #2

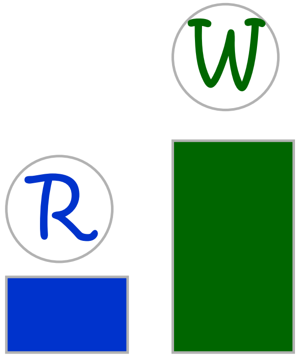
- int readcount = 0, writecount = 0;
- semaphore x = 1, y = 1, wrt = 1, red = 1;
- The structure of a writer process

```
while (true) {
        wait(y);
          writecount++;
          if (writecount == 1)
        wait(red);
        signal(y);
        wait(wrt);
          // writing is performed
        signal(wrt);
        wait(y);
          writecount--;
          if (writecount == 0)
        signal(red);
        signal(y);
    }
```

Readers-Writers Problem: Solution #2

- The structure of a reader process

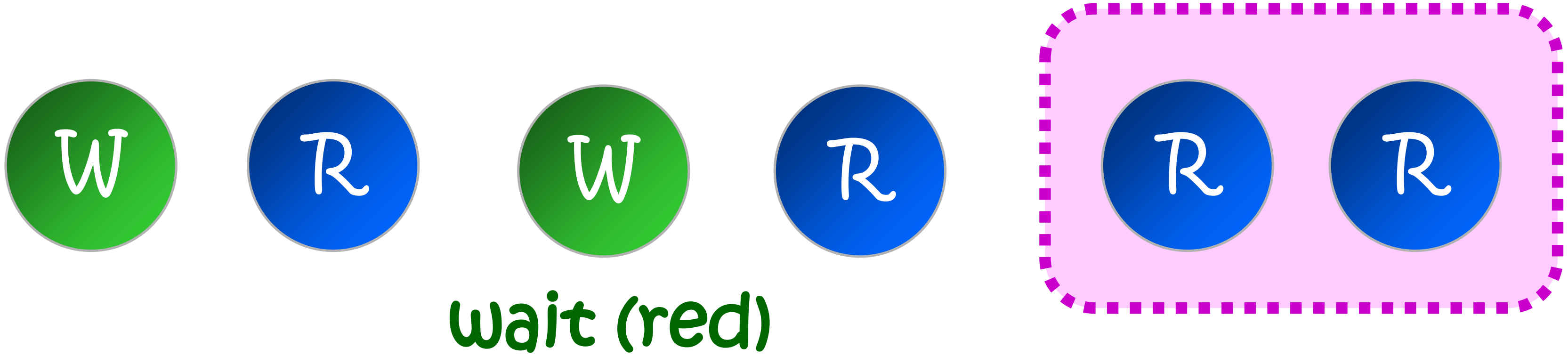
```
while (true) {
    wait(red);
      wait(x);
        readcount++;
        if (readcount == 1)  wait(wrt);
      signal (x);
    signal(red);
      // reading is performed
    wait (x);
      readcount--;
      if (readcount == 0)  signal(wrt);
    signal(x);
}
```



The second readers-writers problem

```
wait (y);
   writecount++;
  if (writecount == 1) wait(red) ;
signal (y);
wait (wrt);
……
```

```
wait (red);
……
```



The third readers-writers problem

- In fact, the solutions implied by both problem statements result in starvation — the first readers-writers problem may starve writers in the queue, and the second readers-writers problem may starve readers.
- Therefore, the third readers-writers problem is sometimes proposed, which adds the constraint that no process/thread shall be allowed to starve; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.
- Solutions to the third readers-writers problem will necessarily sometimes require readers to wait even though the share is opened for reading, and sometimes require writers to wait longer than absolutely necessary.

Readers-Writers Problem: Solution #3

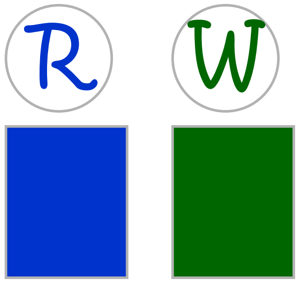
- int readcount = 0;
- semaphore mutex = 1, wrt = 1, S=1;
- The structure of a writer process

```
while (true) {
    wait (S);
      wait (wrt);
        // writing is performed
      signal (wrt);
    signal (S);
}
```

Readers-Writers Problem: Solution #3

- The structure of a reader process

```
while (true) {
    wait (S);
      wait(mutex);
        readcount++;
        if (readcount == 1)  wait(wrt);
      signal(mutex);
    signal (S);

      // reading is performed

    wait(mutex);
      readcount--;
      if (readcount == 0)  signal(wrt);
    signal(mutex);
}
```

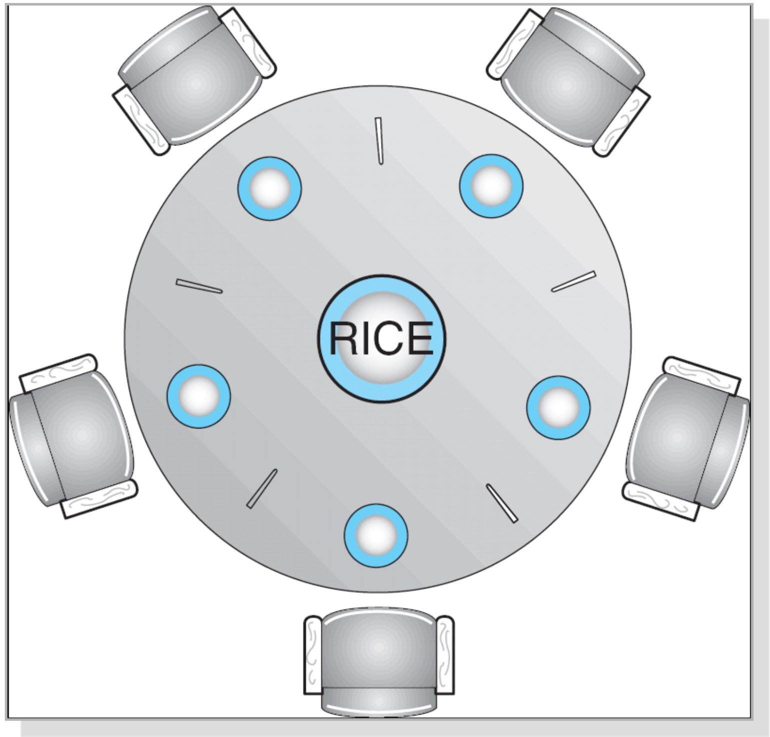



```
while (true) {
    wait(mutex);
      readcount++;
      if (readcount == 1)  wait(wrt);
    signal(mutex);

        // reading is performed

    wait(mutex);
      readcount--;
      if (readcount == 0)  signal(wrt);
    signal(mutex);
}
```

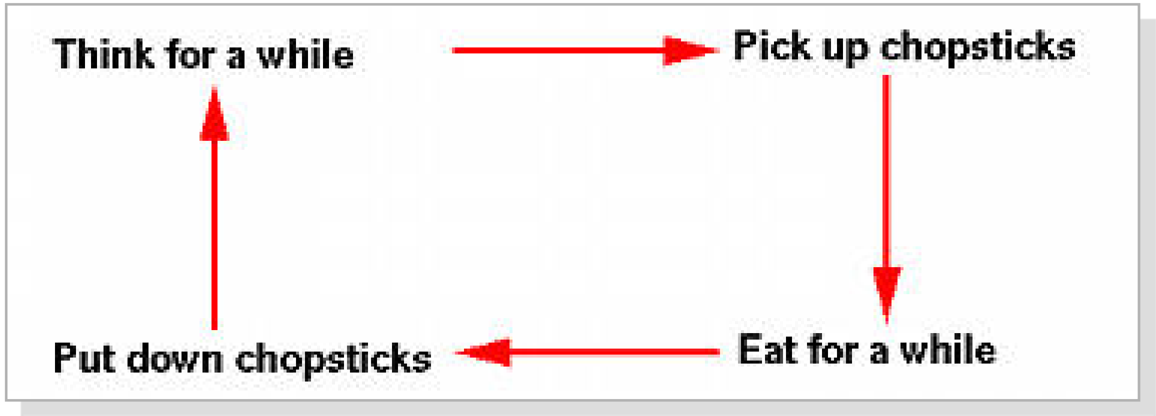
Solution#1

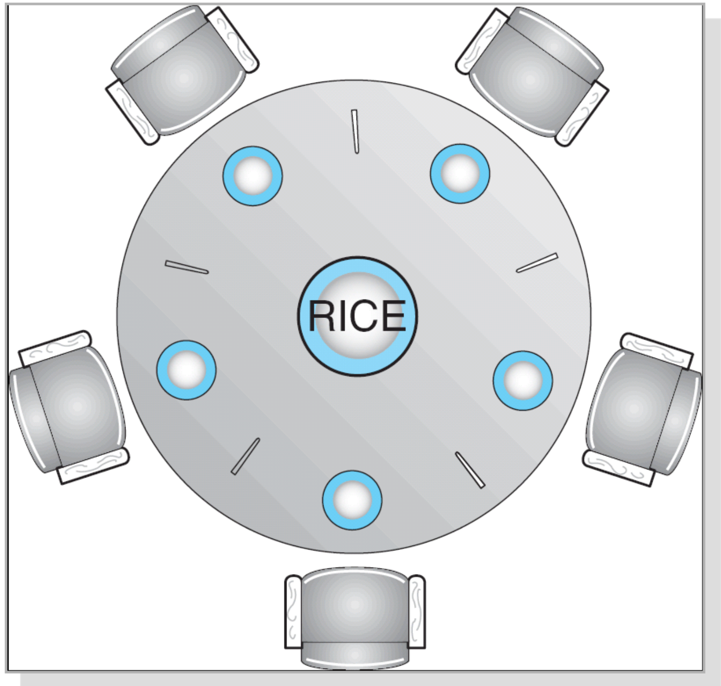
Dining-Philosophers Problem

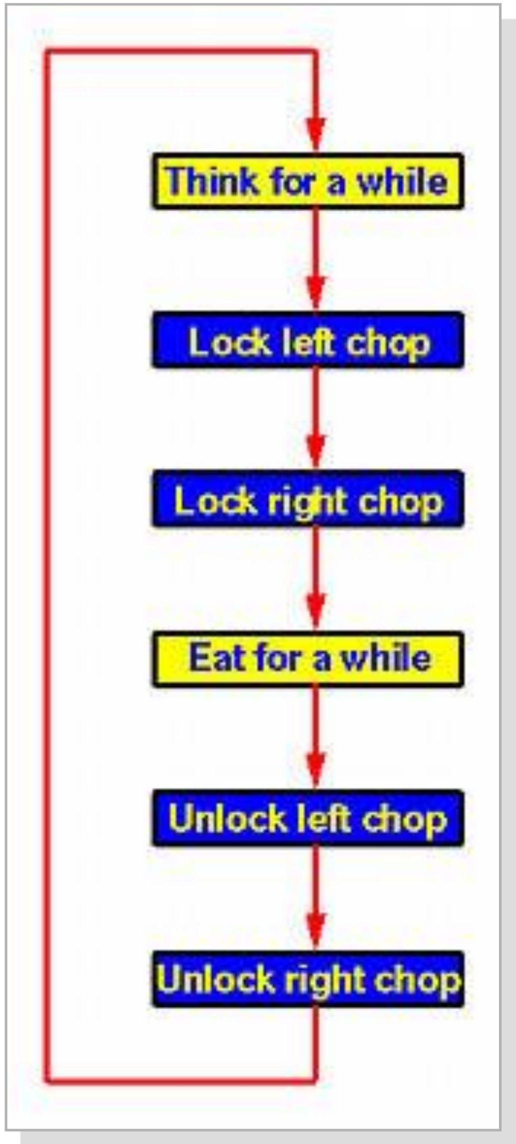
- Five philosophers, either thinking or eating
- To eat, two chopsticks are required
- Taking one chopstick at a time



Dining-Philosophers Problem

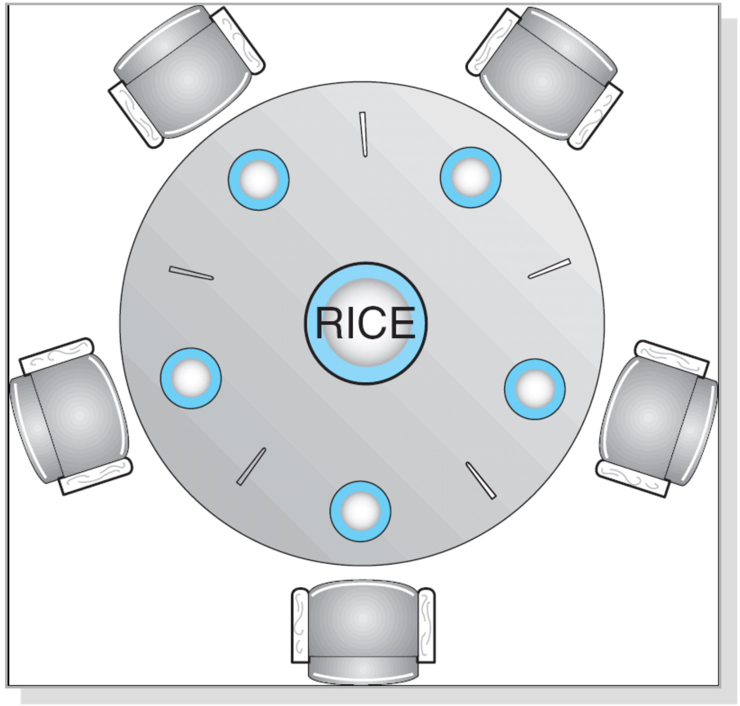






Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem

- The structure of Philosopher i:

```
While (true)  {
      wait ( chopstick[i] );
      wait ( chopstick[ (i + 1) % 5] );
        //  eat;
      signal ( chopstick[i] );
      signal ( chopstick[ (i + 1) % 5] );
        //  think;
}
```

get chopsticks
left
right

free
chopsticks
left
right

Dining-Philosophers Problem

- Possible solutions to the deadlock problem
  - Allow at most **four** philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if **both chopsticks are available** (note that she must pick them up in a critical section).
  - Use an asymmetric solution; that is,
    - odd philosopher: **left first, and then right**
    - even philosopher: **right first, and then left**
- Besides deadlock, any satisfactory solution to the DPP must avoid the problem of **starvation**.

Solution 2 for Dining-Philosophers Problem

- semaphore mutex=1;

```
void philosopher(int i) {
    while(TRUE) {
        think( );
        P(mutex);
        take_chopstick (i);
        take_chopstick ((i + 1) % N);
        eat( );
        put_chopstick (i);
        put_chopstick ((i + 1) % N);
        V(mutex);
    }
}
```

Solution 3 for Dining-Philosophers Problem

- S1 THINKING…
- S2 I am HUNGRY
- S3 If my left neighbor or my right neighbor is EATING then block myself; else goto S4
- S4 Pick up both chopsticks
- S5 EATING …
- S6 Put down the chopsticks and wake up the left neighbor if he can EAT
- S7 Put down the chopsticks and wake up the right neighbor if he can EAT
- S8 Goto S1

Solution 3 for Dining-Philosophers Problem

- Define the data structures:

```
#define  N          5
#define  LEFT       (i+N-1)%N
#define  RIGHT      (i+1)%N
#define  THINKING   0
#define  HUNGRY     1
#define  EATING     2
int   state[N];
semaphore   mutex;      // initial value 1
semaphore   s[N];       // initial value 0
```

Solution 3 for Dining-Philosophers Problem

北京交通大学

```
void philosopher(int i)  // i：0~N-1
{
   while(TRUE)
       {
S1 ———— think( );

S2–S4 ——— take_chopsticks(i);

S5 ———— eat( );

S6–S7 ——— put_chopsticks(i);
   }
}
```

Solution 3 for Dining-Philosophers Problem

- // Pick up both chopsticks, or block

```
void take_chopsticks(int i)  // i: 0~N-1
{
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}
```

```
void test (int i)
{
    if(state[i] ==  HUNGRY &&
      state[LEFT]  !=  EATING  &&
      state[RIGHT]  !=  EATING )
    {
        state[i] = EATING;
        V(s[i]);
    }
}
```

Solution 3 for Dining-Philosophers Problem

- // put down the two chopsticks and wake up the neighbors if necessary

```
void put_chopsticks(int i)
{
    P(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    V(mutex);
}
```

Sleeping Barber Problem

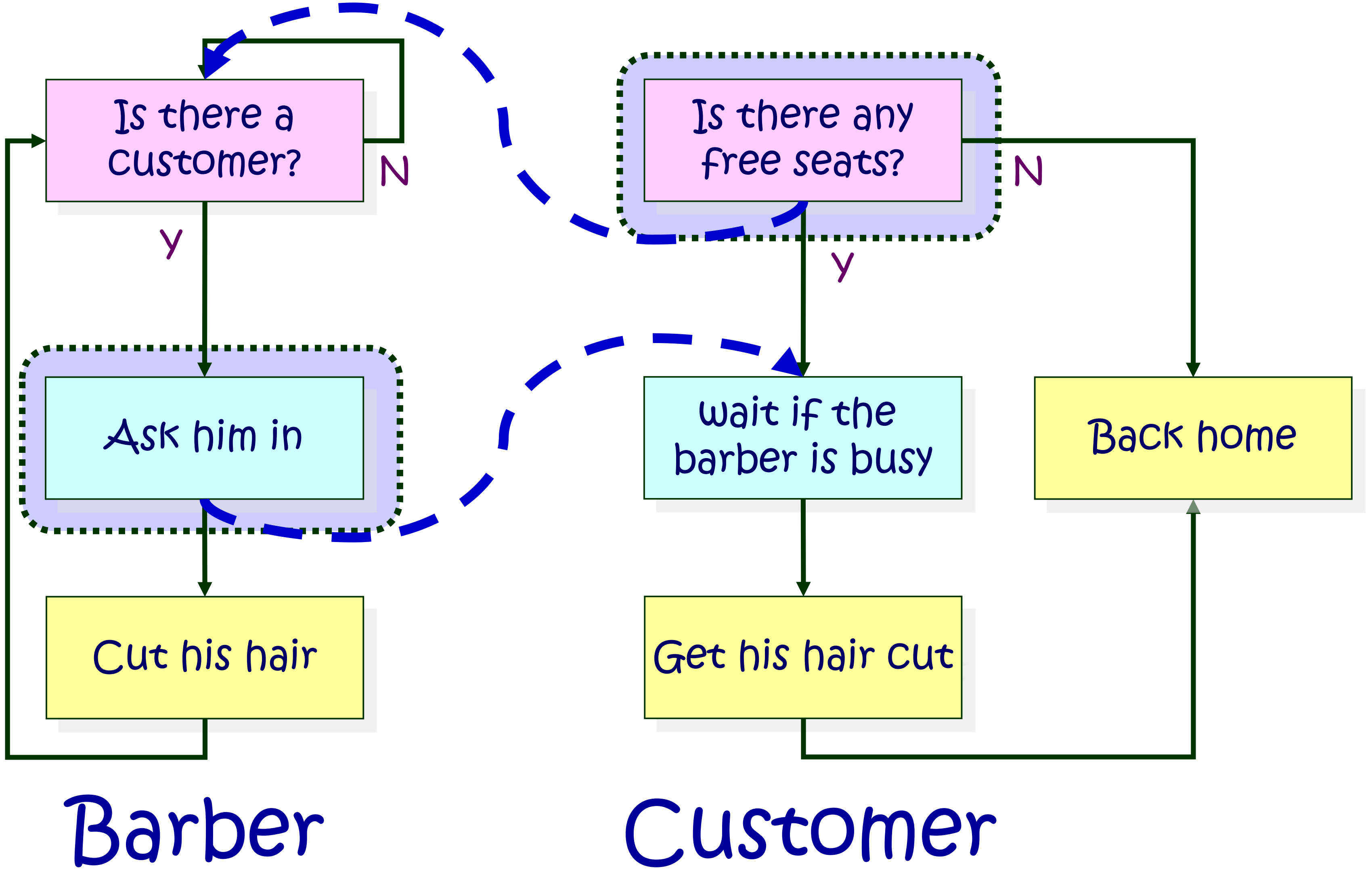
Sleeping Barber Problem

- The sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system processes.
- There is a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it.

Sleeping Barber Problem

- When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting.
  - If yes, he brings one of them back to the chair and cuts his or her hair.
  - If no, he returns to his chair and sleeps in it.
- Each customer, when he arrives, looks to see what the barber is doing.
  - If the barber is sleeping, then he wakes him up and sits in the chair.
  - If the barber is cutting hair, then he goes to the waiting room.
    - If there is a free chair in the waiting room, he sits in it and waits his turn.
    - If there is no free chair, then the customer leaves.

Sleeping Barber Problem



Sleeping Barber Problem

- You need (as mentioned above):

  Semaphore Customers = 0;   //#waiting customers

  Semaphore Barber = 0;

  Semaphore accessSeats (mutex) = 1;

  int NumberOfFreeSeats = N   //total number of seats

$$
\text{Customers}\begin{cases} > 0\text{, there are waiting customers} \\ = 0\text{, there are no waiting customers} \\ = -1\text{, the barber is sleeping} \end{cases}
$$

$$
\text{Barber}\begin{cases} = 1\text{, the barber is not busy} \\ = 0\text{, the barber is busy cutting or sleeping} \\ < 0\text{, there are waiting customers} \end{cases}
$$

The Barber Process

```
Void barber(void)
{
    while(true) {                         // runs in an infinite loop
      P(Customers)                  // tries to acquire a customer
                                    // – if none is available he goes to sleep
      P(accessSeats)                // at this time he has been awakened
                                    //  - want to modify the number of available seats
      NumberOfFreeSeats++  // one chair gets free
      V(Barber)                     // the barber is ready to cut
      V(accessSeats)                // we don't need the lock on the   chairs anymore
      cut_hair();                   //here the barber is cutting hair
    }
}
```

The Customer Process

```
void customer(void)
{
    while(true) {                          // runs in an infinite loop
        P(accessSeats)                     // tries to get access to the chairs
        if ( NumberOfFreeSeats > 0 ) {     // if there are any free seats
            NumberOfFreeSeats--            // sitting down on a chair
            V(Customers)                   // notify the barber, who's waiting that there is a customer
             V(accessSeats)                // don't need to lock the chairs anymore
             P(Barber)                     // now it's this customer's turn, but wait if the barber is busy
             //here the customer is having his hair cut
           } else {                        // there are no free seat, tough luck
               V(accessSeats)              // but don't forget to release the lock on the seats
               //customer leaves without a haircut
          }
      }
}
```

Monitors

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - Both mutual exclusion and scheduling constraints
    - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
  - Cleaner idea: Use **locks** for mutual exclusion and **condition variables** for scheduling constraints

Motivation for Monitors and Condition Variables

- **Monitor**: a **lock** and zero or more **condition variables** for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
  - Some languages like *Java* provide monitors in the language
  - Most others use actual locks and condition variables
- The lock provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free

Monitor: A High-level Language Constructs

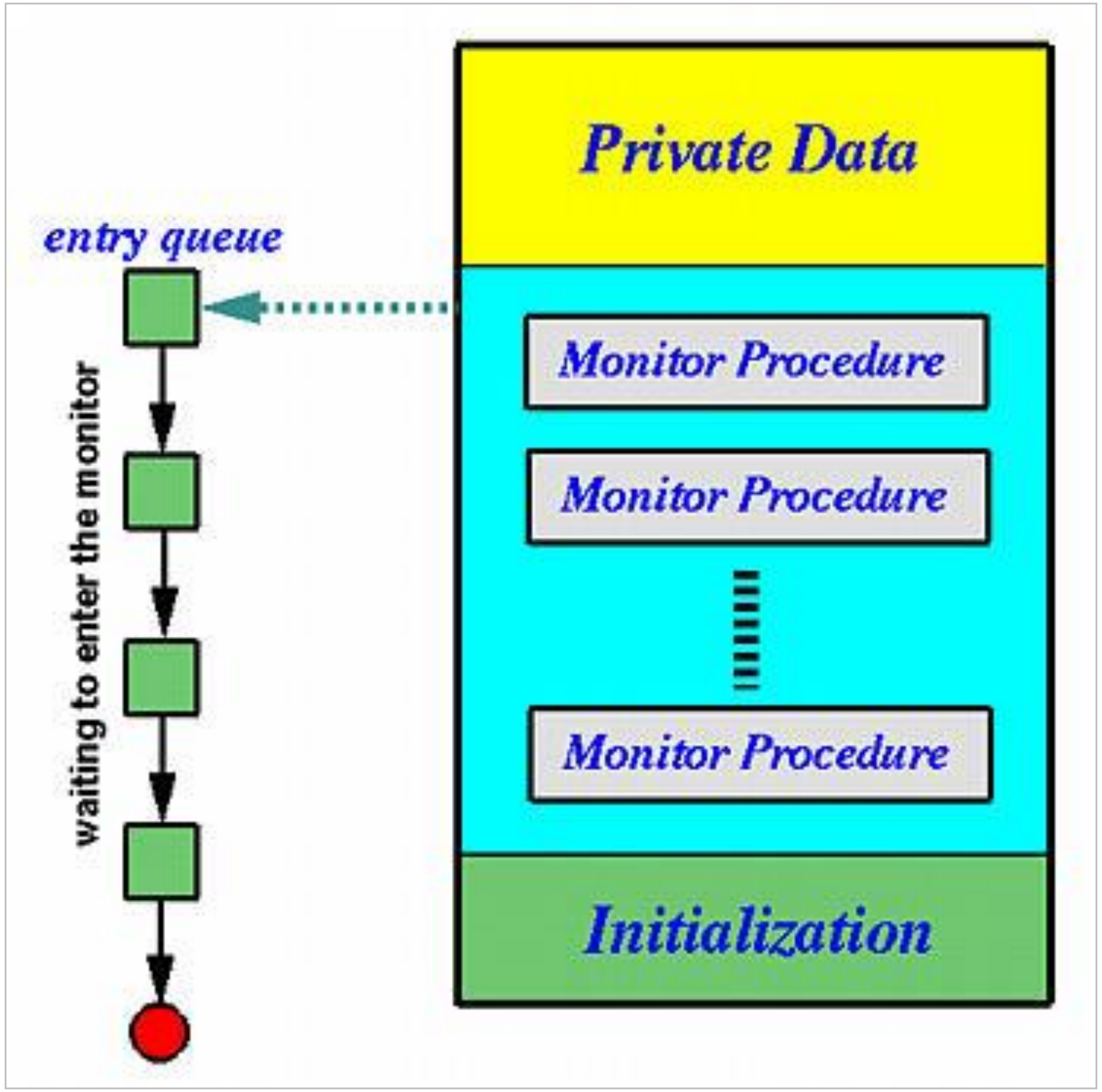
- The representation of a monitor type consists of
  - declarations of variables whose values define the state of an instance of the type
  - procedures or functions that implement operations on the type.
- A procedure within a monitor can access only variables defined in the monitor and the formal parameters.
- The local variables of a monitor can be used only by the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

Monitors

- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
 // shared variable declarations
 procedure P1 (…) { … }
     …
 procedure Pn (…) {……}
 Initialization code ( … ) { … }
}
```

Monitors


Private Data
entry queue
waiting to enter the monitor
Monitor Procedure
Monitor Procedure
Monitor Procedure
Initialization

Monitors-Condition Variables

- Condition Variable: a queue of processes waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- To allow a process to wait **within** the monitor, a **condition variable** must be declared, as

condition x, y;

Monitors-Condition Variables

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation **x.wait()** means that the process invoking this operation is suspended until another process invokes **x.signal();**
  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has **no effect**.

Monitors

```
monitor monitor-name
{
 // shared variable declarations
 condition a,b, …… ;
 procedure P1 (…) { … }
     …
 procedure Pn (…) {……}
 Initialization code ( … ) { … }
}
```

Monitors–Condition Variables

- When a process P calls Wait() on a condition variable, several things happen atomically.
  - First, P releases the monitor lock, which will allow other processes to enter the monitor while this process is waiting.
  - Then, a reference to P is placed on the queue associated with the condition variable.
  - Finally, P removes itself from the CPU ready queue, and yields the processor to somebody else. Since P is no longer on the ready queue, it will not get the CPU again until some other process with the CPU places P back on the ready queue again.

Mesa vs. Hoare monitors

- P wakes up Q
- Hoare-style (most textbooks):
  - Signaler(P) gives lock, CPU to waiter(Q); waiter(Q) runs immediately
- Mesa-style (most real operating systems):
  - Signaler(P) keeps lock and processor
  - Waiter(Q) placed on ready queue with no special priority

Monitors

- **Monitor** Producer_Comsumer

```
condition full, empty;
integer count;
void insert (item) {
        if (count == N) then wait (empty);
        insert (item);
        count++;
        if (count == 1) then signal (full); }
item remove() {
        if (count==0) then wait (full);
        remove an item;
        count--;
        if (count == N-1) then signal (empty); }
count=0;
```

Monitors

- producer {
```
        While (true) {
                item = producer_item;
                Producer_Comsumer.insert (item);
        }
    }
```
- consumer {
```
        While (true) {
                item=Producer_Comsumer.remove ();
                return item;
        }
  }
```

Semaphore vs. Monitor

Semaphores	Condition Variables
Can be used anywhere in a program, but should not be used in a monitor	Can only be used in monitors
Wait() does not always block the caller (i.e., when the semaphore counter is greater than zero).	**Wait()** always blocks the caller.
Signal() either releases a blocked process, if there is one, or increases the semaphore counter.	**Signal()** either releases a blocked process, if there is one, or the signal is lost as if it never happens.
If **Signal()** releases a blocked process, the caller and the released process **both** continue.	If **Signal()** releases a blocked process, the caller yields the monitor (Hoare type) or continues (Mesa Type). Only one of the caller or the released process can continue, but not both.

Solution to Dining Philosophers (self study)

- Each philosopher I invokes the operations pickup() and putdown() in the following sequence:

diningPhilosopher.pickup (i)

EAT

diningPhilosopher.putdown (i)

Solution to Dining Philosophers (con't)

```
monitor diningPhilosopher {
  enum { THINKING, HUNGRY, EATING) state [5] ;
  condition self [5];

  void pickup (int i) {
       state[i] = HUNGRY;
       test(i);
       if (state[i] != EATING) self [i].wait;
  }

   void putdown (int i) {
       state[i] = THINKING;
       // test left and right neighbors
       test((i + 4) % 5);
       test((i + 1) % 5);
  }
```

Solution to Dining Philosophers (con't)

```
void test (int i) {
     if (  (state[(i + 4) % 5] != EATING) &&
           (state[i] == HUNGRY) &&
           (state[(i + 1) % 5] != EATING) ) {
           state[i] = EATING ;
           self[i].signal () ;
     }
}

initialization_code() {
     for (int i = 0; i < 5; i++)
     state[i] = THINKING;
}
}
```

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

  - Doesn't work: Wait() may sleep with lock held

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each procedure **F** will be replaced by

```
wait(mutex);
    …
     body of F;
    …
if (next-count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Additional material

Summary

Summary

- Concurrent processes (threads) are a very useful abstraction
- Concurrent processes (threads) introduce problems when accessing shared data
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives

Summary

- Showed how to protect a critical section with only atomic load and store $\Rightarrow$ pretty complex!
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, TestAndSet, Swap
- Talked about Semaphores, Monitors, and Condition Variables

Summary

- Semaphores: Like integers with restricted interface
  - Two operations:
    - `P()`
    - `V()`
    - Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - The Operations: `Wait()`, `Signal()`