Process Synchronization

Outline

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples

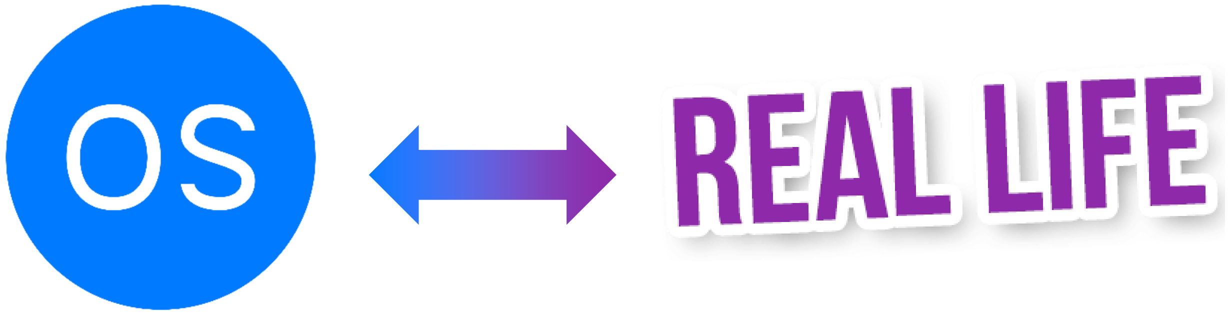
"Too much milk"

- People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Motivation: “Too much milk”

- Great thing about OS’s – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people



Background

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the **orderly** execution of cooperating processes

Race Condition

- Producer and Consumer
- count++ could be implemented as

```
    register1 = count
    register1 = register1 + 1
    count = register1
```

- count-- could be implemented as

```
    register2 = count
    register2 = register2 - 1
    count = register2
```

Race Condition

- Assume counter is initially 5.

count++	count--
register1 = count	register2 = count
register1 = register1 + 1	register2 = register2 - 1
count = register1	count = register2

Consider this execution interleaving with "count = 5" initially:

S0: producer	execute	register1 = count	{register1 = 5}
S1: producer	execute	register1 = register1 + 1	{register1 = 6}
S2: consumer	execute	register2 = count	{register2 = 5}
S3: consumer	execute	register2 = register2 - 1	{register2 = 4}
S4: producer	execute	count = register1	{count = 6 }
S5: consumer	execute	count = register2	{count = 4}

Race Condition

- We're off.
- P gets off to an early start
- C says "humph, better go fast" and tries really hard
- P goes ahead and writes "6"
- C goes and writes "4"
- P says "HUH??? I could have sworn I put a 6 there"

Consider this execution interleaving with "count = 5" initially:

S0: producer	execute	register1 = count	{register1 = 5}
S1: producer	execute	register1 = register1 + 1	{register1 = 6}
S2: consumer	execute	register2 = count	{register2 = 5}
S3: consumer	execute	register2 = register2 - 1	{register2 = 4}
S4: producer	execute	count = register1	{count = 6 }
S5: consumer	execute	count = register2	{count = 4}

Race Condition

- We’re off.
- P gets off to an early start
- C says “humph, better go fast” and tries really hard
- P goes ahead and writes “6”
- C goes and writes “4”
- P says “HUH??? I could have sworn I put a 6 there”

Could this happen on a uniprocessor?

Yes! Unlikely, but if you depending on it not happening, it will and your system will break…

Definitions

- **Race condition**: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

Definitions

- **Synchronization**: using atomic operations to ensure cooperation between processes
  - For now, only loads and stores are **atomic**
  - We are going to show that it's hard to build anything useful with only loads and stores

Definitions

- **Atomic Operation**: an operation that always runs to completion or not at all
  - It is indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for processes/threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

Definitions

- **Mutual Exclusion**: ensuring that only one process does a particular thing at a time
  - One process excludes the other while doing its task
- **Critical Section**: piece of code that only one process can execute at once. Only one process at a time will get into this section of code.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

The Critical-Section Problem

The Critical-Section Problem

- *n* processes are all competing to use some **shared data**
- Each process has a code segment, called **critical section**, in which the shared data is accessed.
- **Problem** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

The Critical-Section Problem

```
do {
  critical section


  remainder
  section


} while (TRUE);
```

```
do {
  buy milk;

  be in a daze;

} while (TRUE);
```

Too Much Milk: Correctness Properties

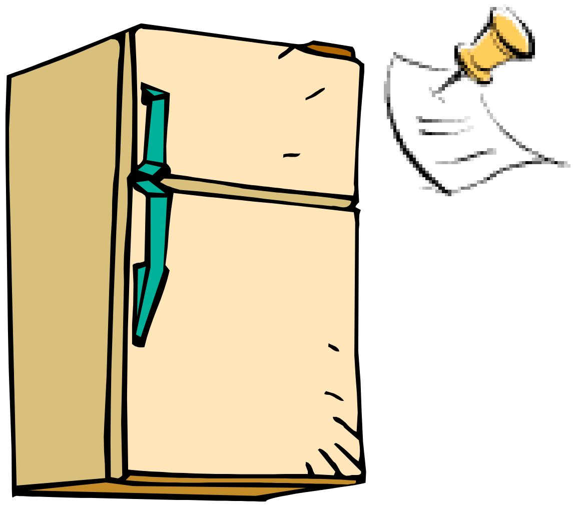
- Need to be careful about correctness of concurrent programs, since non-deterministic
    - Always write down behavior first
    - Impulse is to start coding first, then when it doesn’t work, pull hair out
    - Instead, think first, then code
- What are the correctness properties for the “Too much milk” problem???
    - Never more than one person buys
    - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (no Milk) {
    if (no Note) {
        leave Note;
        buy milk;
        remove Note;
    }
}
```

- Result?



Too Much Milk: Solution #1

Process A	Process B

if (no Milk)
    if (no Note)	

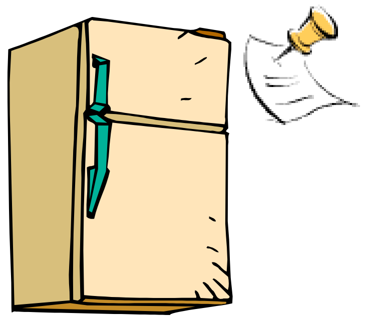
	if (no Milk)
	    if (no Note)

leave Note;
        buy milk;
        remove Note;	

	leave Note;
	        buy milk;
	        remove Note;

Too Much Milk: Solution #1

- Result?
  - Still too much milk but only occasionally!
  - Process can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!



Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let’s try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (no Milk) {
     if (no Note) {
        buy milk;
     }
  }
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Process A

```
leave note A;
  if (no Note B) {
    if (no Milk) {
      buy Milk;
    }
  }
remove note A;
```

Process B

```
leave note B;
  if (no Note A) {
    if (no Milk) {
      buy Milk;
    }
  }
remove note B;
```

- Does this work?

Too Much Milk Solution #2

Process A	Process B
leave note A;	
	leave note B;

if (no Note B)
  if (no Milk)
    buy Milk;	

	if (no Note A)
	  if (no Milk)
	    buy Milk;

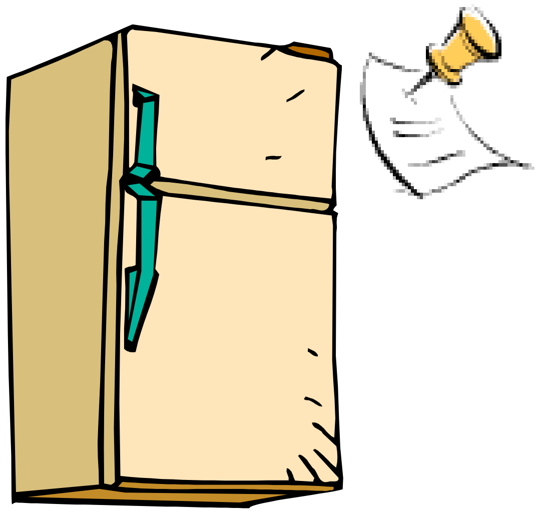
remove note A;	
	remove note B;

Too Much Milk Solution #2

- Possible for neither process to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely that this would happen, but will at worse possible time

Too Much Milk Solution #2: problem!

- I’m not getting milk, You’re getting milk
- This kind of lockup is called “starvation!”



Too Much Milk Solution #3

- Here is a possible two-note solution:

Process A
```
leave note A;
while (note B){
   do nothing;
}
if (no Milk) {
   buy milk;
}
remove note A;
```

Process B
```
leave note B;
   if (no Note A) {
      if (no Milk) {
         buy milk;
      }
   }
remove note B;
```

- Does this work?

Too Much Milk Solution #3

- Here is a possible two-note solution:

Process A
```
leave note A;
  while (note B) //X
    do nothing;
if (no Milk)
  buy milk;
remove note A;
```

Process B
```
leave note B;
if (no Note A) //Y
  if (no Milk)
    buy milk;
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy, otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy, otherwise, A is either buying or waiting for B to quit

Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each process:

```
if (no Milk) {
  buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of processes?
    - Code would have to be slightly different for each process
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"

Solution #3 discussion

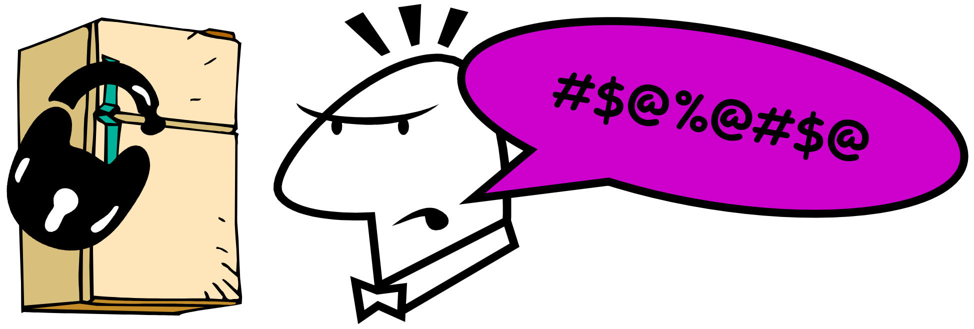
- There’s a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- **Lock**: prevents someone from doing something
- Suppose we have some sort of implementation of a lock.
  - Lock.Acquire() – Wait until lock is free, then grab
  - Lock.Release() – Unlock, waking up anyone waiting
  - These must be atomic operations – if two processes are waiting for the lock and both see it's free, only one succeeds to grab the lock

Too Much Milk: Solution #4

- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: person B angry if only wants other juice



  - Of course – We don’t know how to make a lock yet

Too Much Milk: Solution #4

- Then, our milk problem is easy:

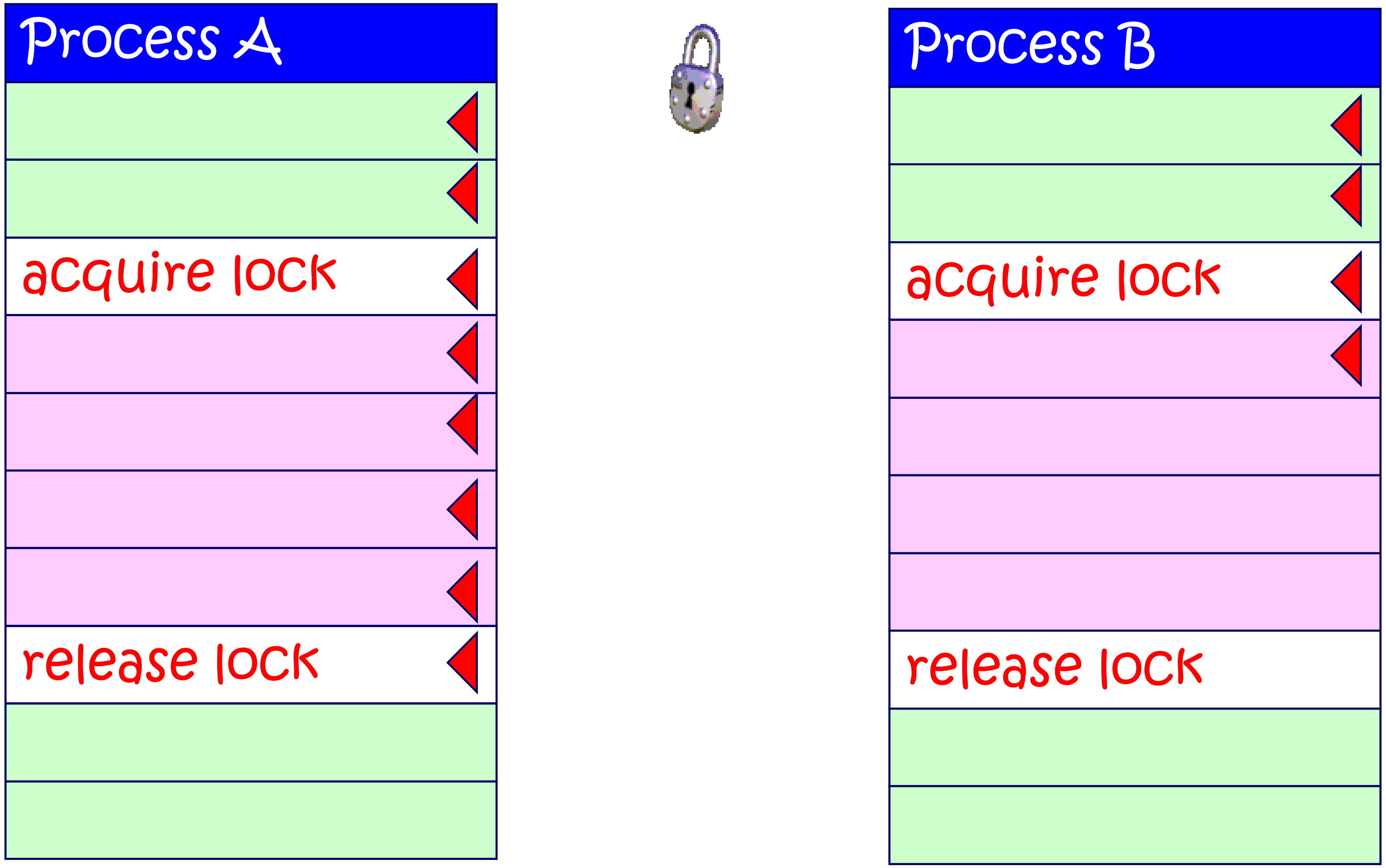
```
milklock.Acquire();
if (no milk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of beer instead of milk
  - Skip the test since you always need more beer.

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
        critical section
        release lock
        remainder section
} while (TRUE);
```

Solution to Critical-section Problem Using Locks



How to implement Locks?

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
- Atomic Load/Store: get solution like Milk #3
  - Pretty complex and error prone

Synchronization Hardware

Naive use of Interrupt Enable/Disable

- Naive Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

- Problems with this approach:
  - Can't let user do this! Consider following:

    ```
    LockAcquire();
    While(TRUE) {;}
    ```

  - What happens with I/O or other important events?
    - "Reactor about to meltdown. Help?"
  - Generally too inefficient on multiprocessor systems
    - Disabling interrupts on all processors requires messages and would be very time consuming
    - Operating systems using this not broadly scalable

Atomic Read-Modify-Write instructions

- Alternative: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors
  - Many systems provide hardware support for critical section code

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Implementing Locks with TestAndSet

- A flawed, but simple solution:

```
 int value = 0; // Free
 Acquire() {
   while (TestAndSet(&value)); // while busy
}
 Release() {
   value = 0;
}
```

- Simple explanation:
  - If lock is free, *TestAndSet* reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, *TestAndSet* reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock
- Busy-Waiting: process consumes cycles while waiting

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor
- Negatives
  - This is very inefficient because the busy-waiting process will consume cycles waiting
  - Priority Inversion: If busy-waiting process has higher priority than process holding lock $\Rightarrow$ no progress!

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE
- Each process has a local Boolean variable key
- Solution:

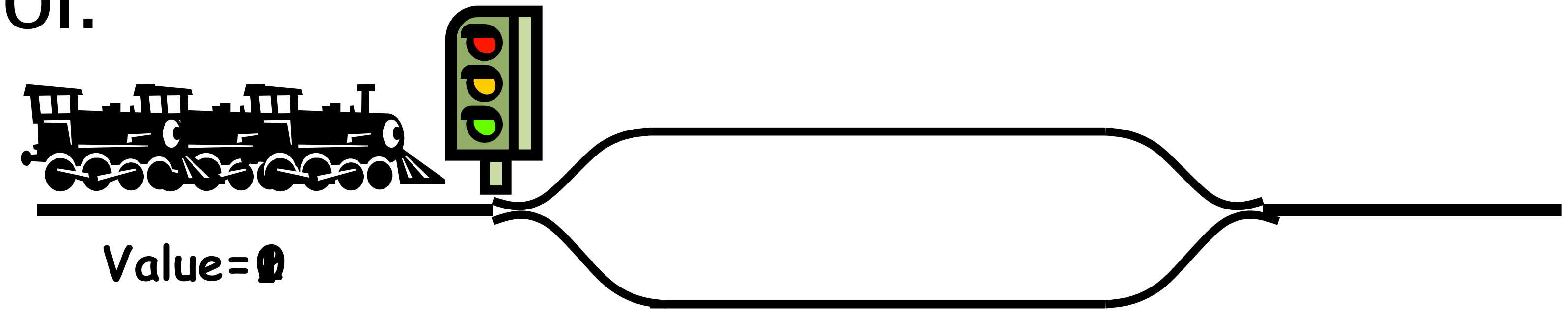
```
do {
    key = TRUE;
    while (key == TRUE)
         Swap (&lock, &key);

    //critical section

    lock = FALSE;

    //remainder section

} while (TRUE);
```

Semaphores

812
0

Semaphores

- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:

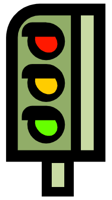


Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in 1965
  - Main synchronization primitive used in original UNIX
  - Does not require busy waiting
    - (if be implemented using wait queue)
  - Less complicated

Semaphores

- Definition: a Semaphore has a non-negative integer value and supports the following two operations (spin lock):
  - **P():** an **atomic** operation that **waits** for semaphore to become positive, then decrements it by 1
    - Think of this as the **wait()** operation
  - **V():** an **atomic** operation that increments the semaphore by 1, waking up a waiting P, if any
    - Think of this as the **signal()** operation



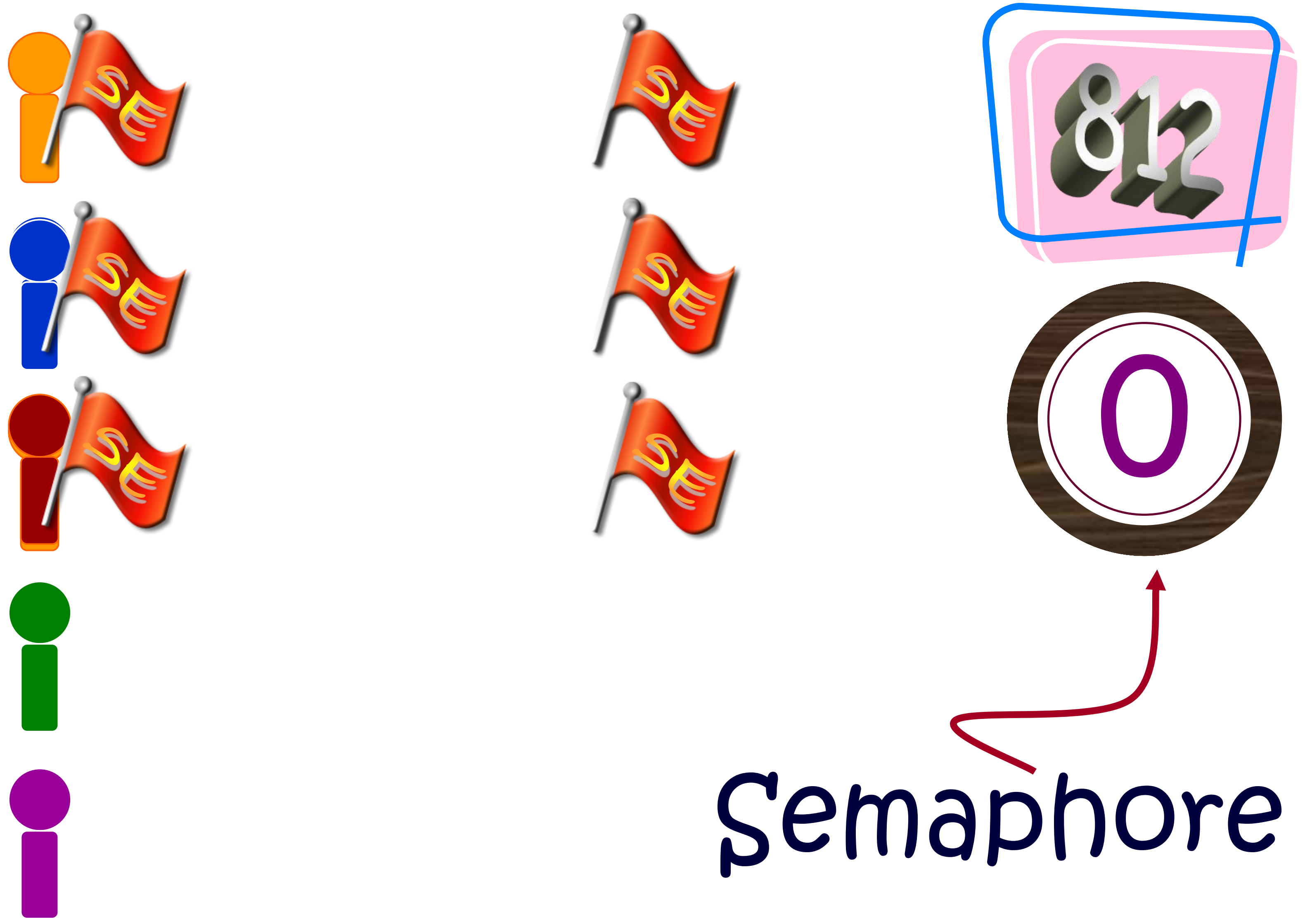
Semaphores

- The definition of **wait ()** is as follows:

```
wait (S) {
   while (S <= 0)
         ; // no-op
   S--;
}
```

- The definition of **signal ()** is as follows:

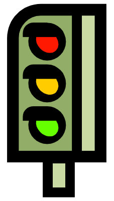
```
signal (S) {
   S++;
}
```



Semaphore

Semaphores

- Note that P stands for “proberen” (to test) and V stands for “verhogen” (to increment) in Dutch
  - P(S) and V(S)
  - Wait(S) and Signal(S)
  - S.wait() and S.signal()



Semaphores

- No negative values
- Only operations allowed are P and V – can’t read or write value, except to set it initially
- Operations must be **atomic**
  - Two P’s together can’t decrement value below zero
  - Similarly, process going to sleep in P won’t miss wakeup from V – even if they both happen at same time

Semaphore Implementation

- The main disadvantage of the semaphore definition given here is that it requires busy waiting.
  - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This type of semaphore is also called a spin lock.

Semaphores

- The definition of **wait** () is as follows:

```
wait (S) {
   while (S <= 0)
         ; // no-op
   S--;
}
```

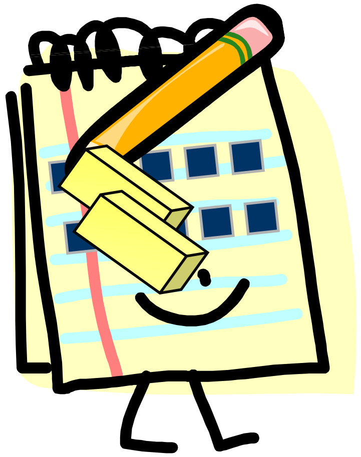
busy waiting

- The definition of **signal** ( ) is as follows:

```
signal (S) {
   S++;
}
```

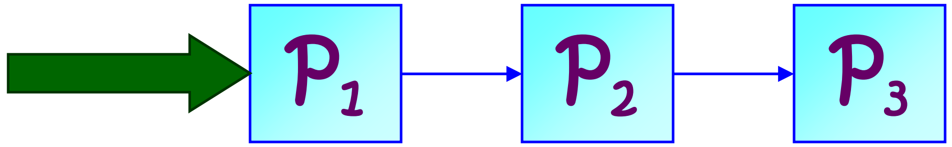
812

0

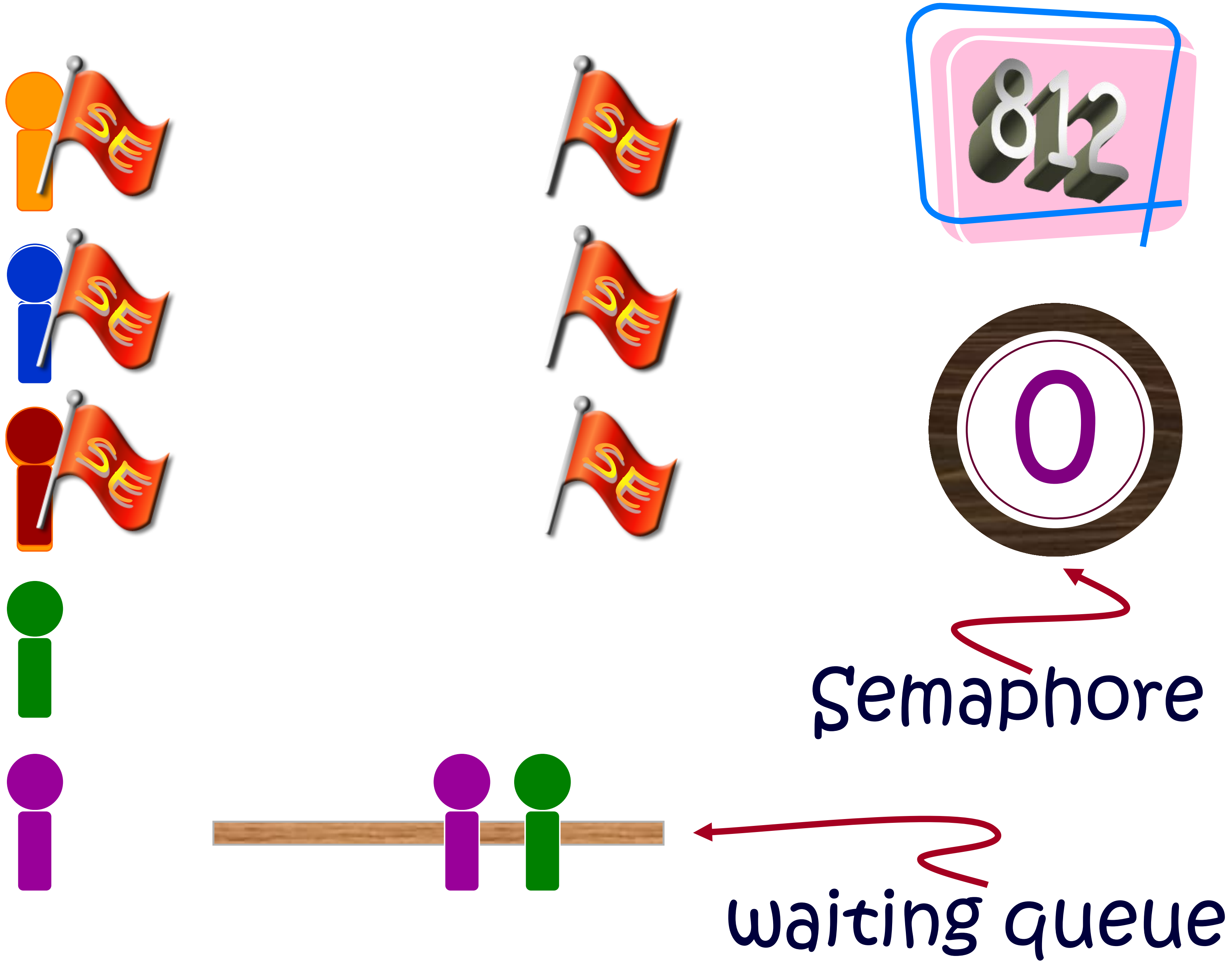


Semaphore Implementation with no Busy Waiting

- With each semaphore there is an associated waiting queue.



- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.


812
0
Semaphore
waiting queue

Semaphore Implementation with no Busy Waiting

- Implementation of wait:

```
Wait (S){
     value--;
     if (value < 0) {
            add this process to waiting queue
            block();
     }
}
```

- Implementation of signal:

```
Signal (S){
     value++;
     if (value <= 0) {
            remove a process P from the waiting queue
            wakeup(P);
     }
}
```

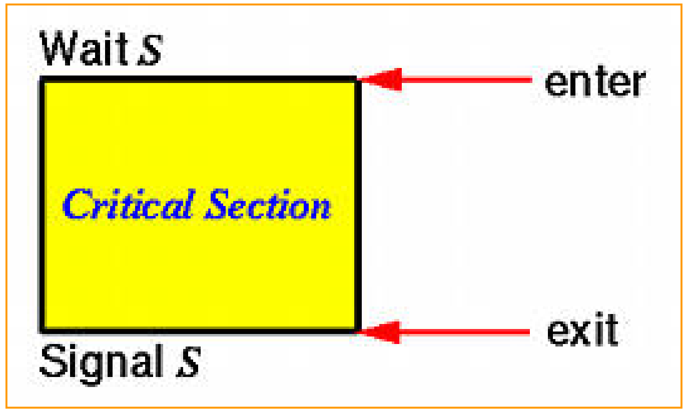
Two Types of Semaphores

- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement.
  - Also known as mutex locks
- **Counting** semaphore – integer value can range over an unrestricted domain.
  - Can implement a counting semaphore **S** as a binary semaphore.

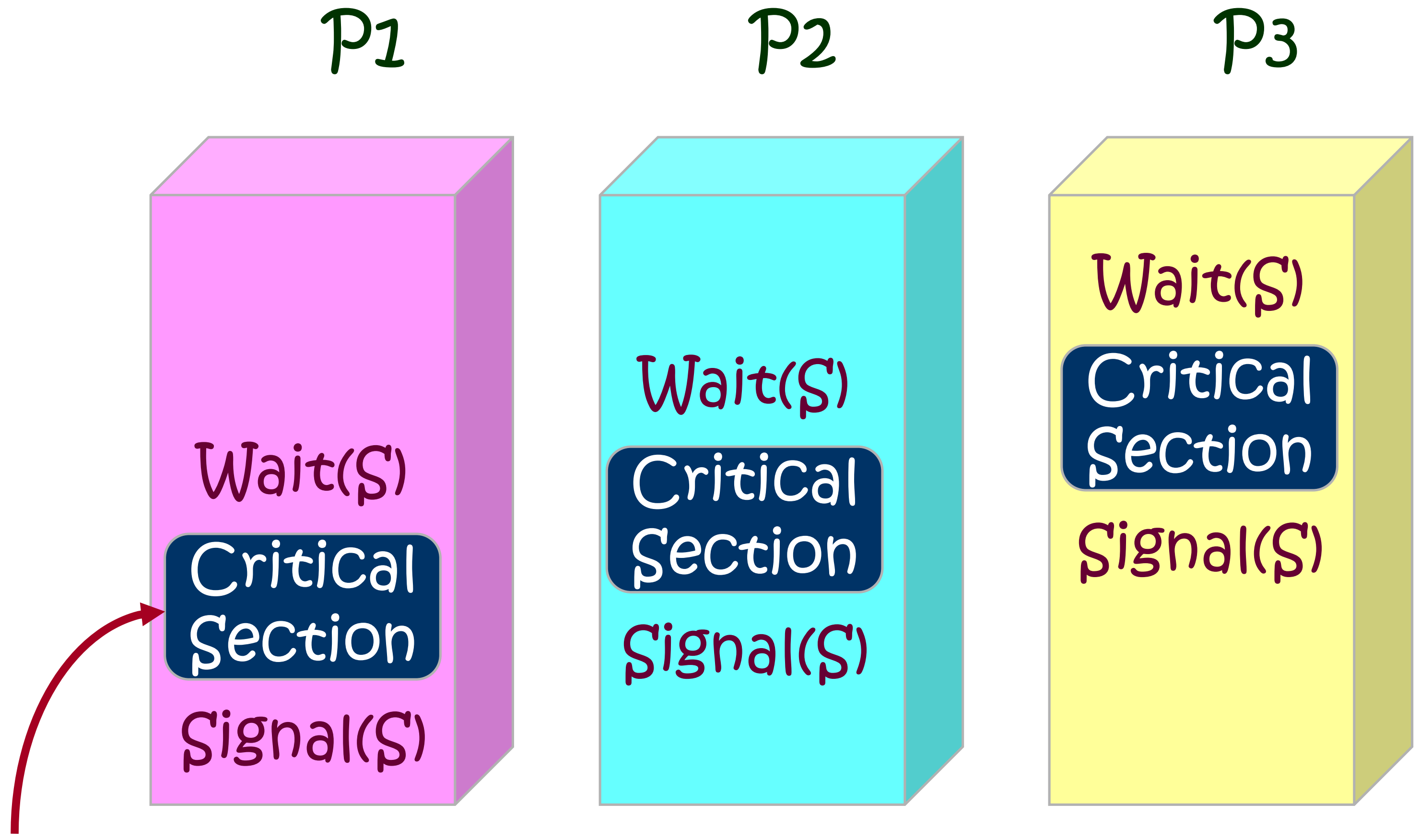
Two Types of Semaphores

- Provides mutual exclusion

```
Semaphore mutex; //  initialized to 1
do {
        wait (mutex);
        // Critical Section
        signal (mutex);
        // remainder section
} while (TRUE);
```

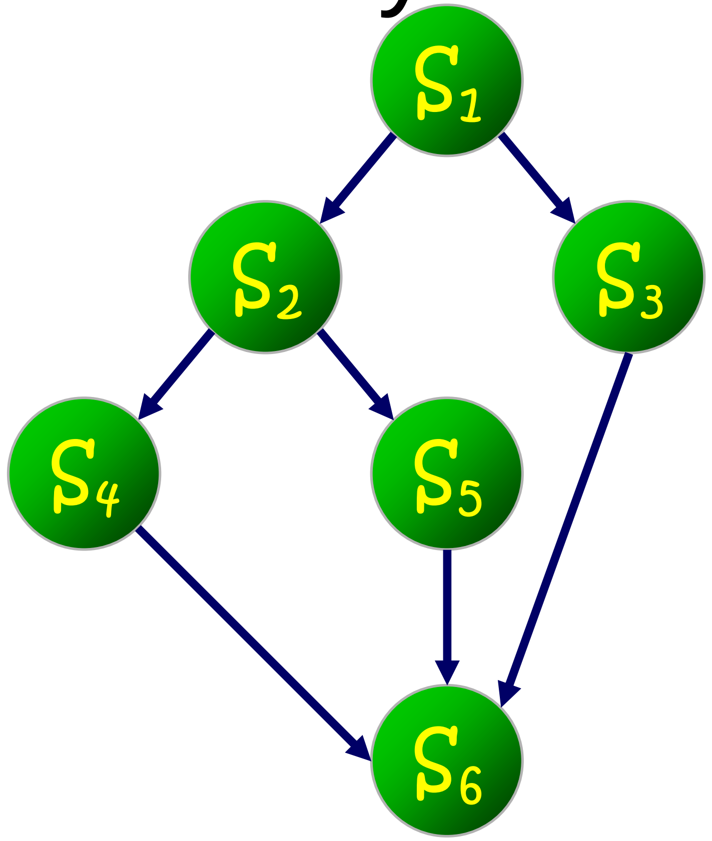


Semaphore


P1
P2
P3
Wait(S)
Critical Section
Signal(S)
Wait(S)
Critical Section
Signal(S)
Wait(S)
Critical Section
Signal(S)

Precedence Graph

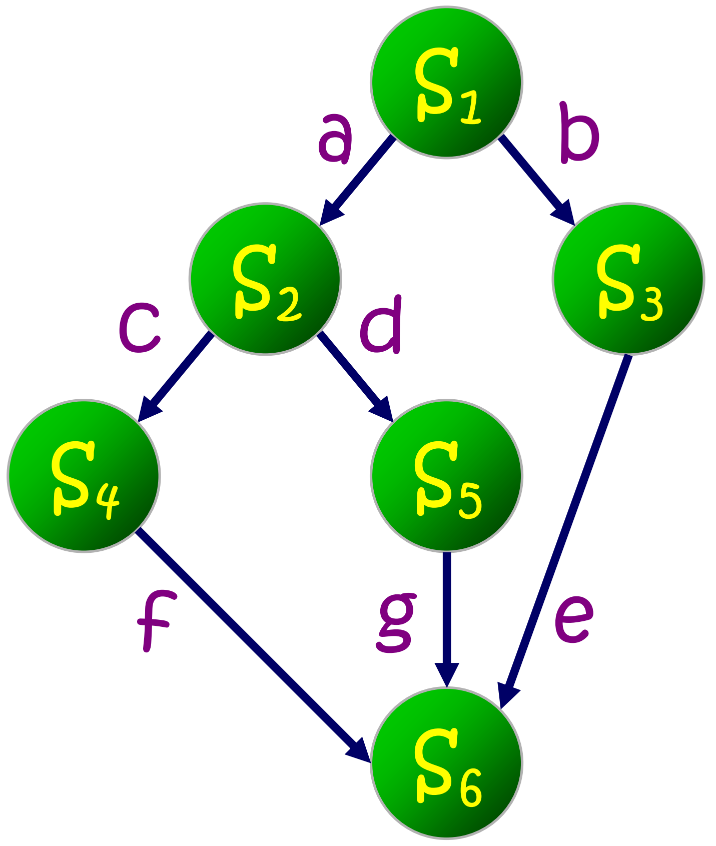
- A DAG (Directed Acyclic Graph)


$$\begin{tikzcd}
 & & S_1 \arrow[dl] \arrow[dr] & \\
 & S_2 \arrow[dl] \arrow[dr] & & S_3 \arrow[ddl] \\
S_4 \arrow[drr] & & S_5 \arrow[d] & \\
 & & S_6 &
\end{tikzcd}$$

Semaphore

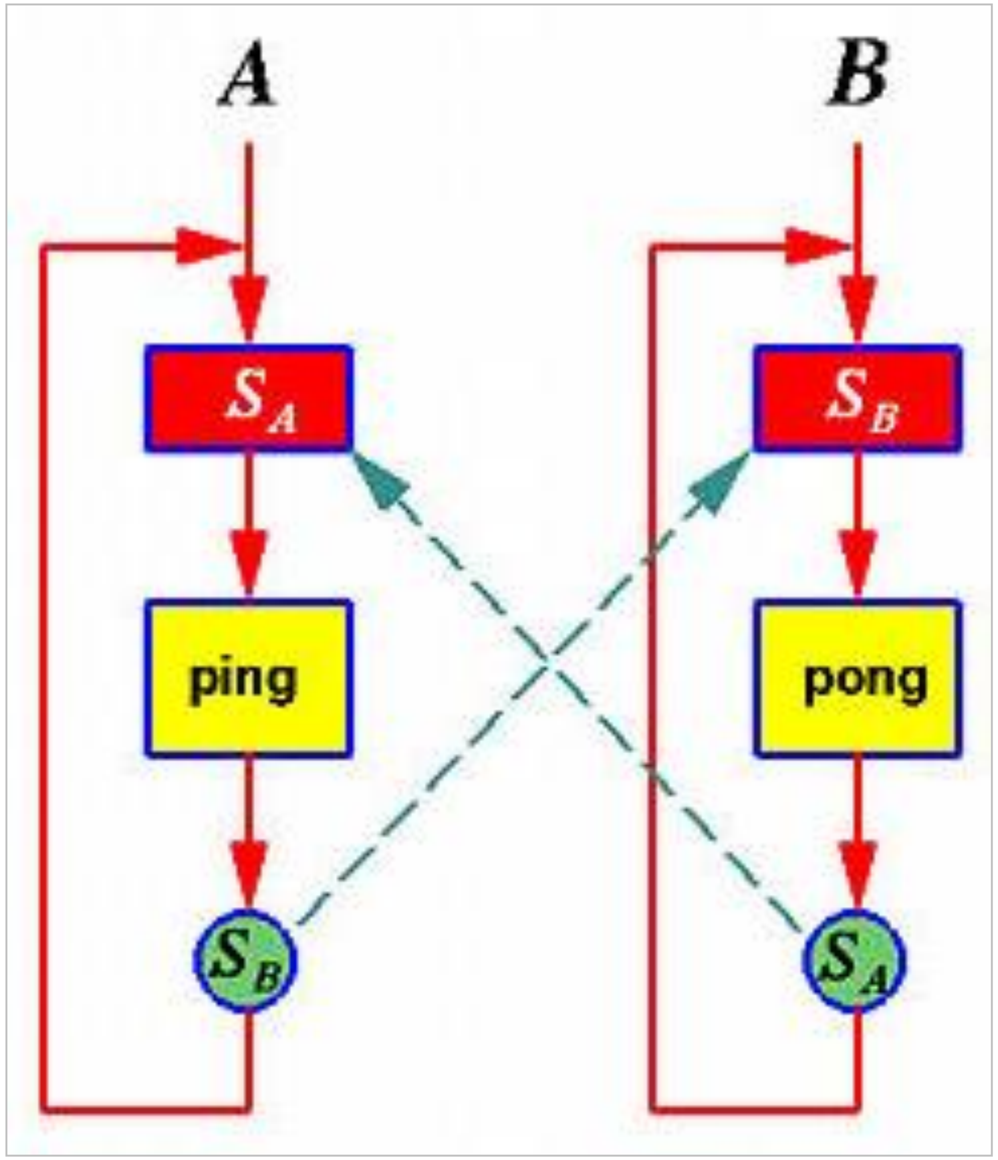
- Semaphore a, b, c, d, e, f, g;
- a=0, b=0, c=0, d=0, e=0, f=0, g=0;

- S1() { …; V(a); V(b);}
- S2() { P(a); …; V(c); V(d); }
- S3() { P(b); …; V(e); }
- S4() { P(c); …; V(f); }
- S5() { P(d); …; V(g); }
- S6() { P(f); P(g); P(e); …; }



Semaphore

```
While(1){
   P(SA);
   ping;
   V(SB);
}
```



```
While(1){
   P(SB);
   pong;
   V(SA);
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
        P0                    P1
   wait (S);             wait (Q);
   wait (Q);             wait (S);
     ...                   ...
   signal  (S);          signal (Q);
   signal (Q);           signal (S);
```

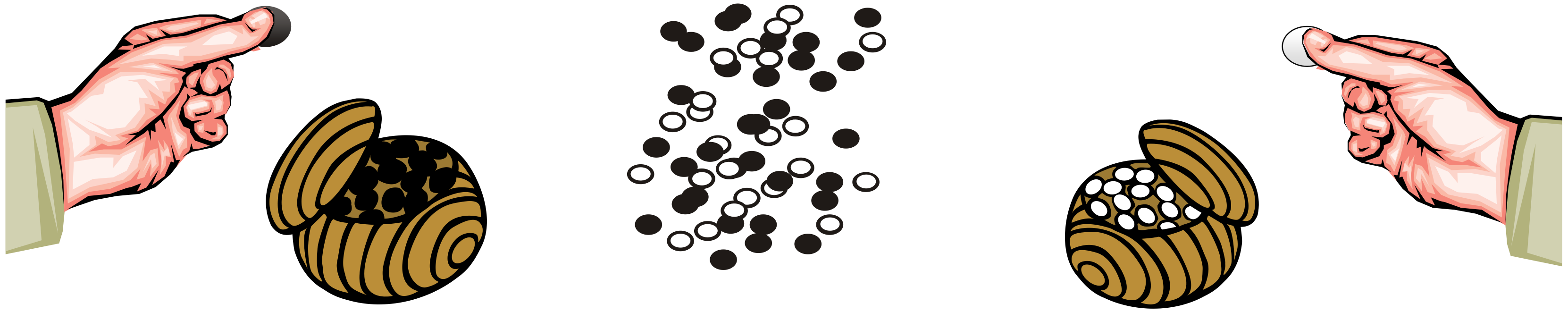
Deadlock and Starvation

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Some Examples

Example 1

- The worker who produces Go accidentally mixes an equal number of black and white pieces in a box, and now separates the black and white with an automatic sorting system consisting of two concurrent processes, which function as follows:
  - (1) Process A: picks up one black piece, Process B: picks up one white piece;
  - (2) Each process picks one and only one piece at a time. At any time, at most one process is permitted to pick up one piece.



Example 1

- Analysis

```
Process_A() {
   While (True)
   {

      pick a ●;

   }
}
```

```
Process_B() {
   While (True)
   {

      pick a ●;

   }
}
```

Example 1

- **Step 1**: Determine the relationship between processes. By (2), it is known that there is a mutually exclusive relationship between processes.
- **Step 2**: Determine the semaphore and its value. Because process A and process B enter the box mutually exclusive to pick pieces, and the box is the public resource, so set a semaphore **s**, the value depends on the number of public resources, because there is only one box, the initial value of **s** is set to 1.

Example 1

- Implementation： semaphore s=1;

```
Process_A() {
   While (True)
   {
      Wait(s);
      pick a ●;
      Signal(s);
   }
}
```

```
Process_B() {
   While (True)
   {
      Wait(s);
      pick a ●;
      Signal(s);
   }
}
```

Example

- The key to determining whether processes are mutually exclusive is to see whether a public resource is shared between processes, and a public resource corresponds to a semaphore. Determining the value of the semaphore is a key point, and the value represents the number of available resource entities.

Example

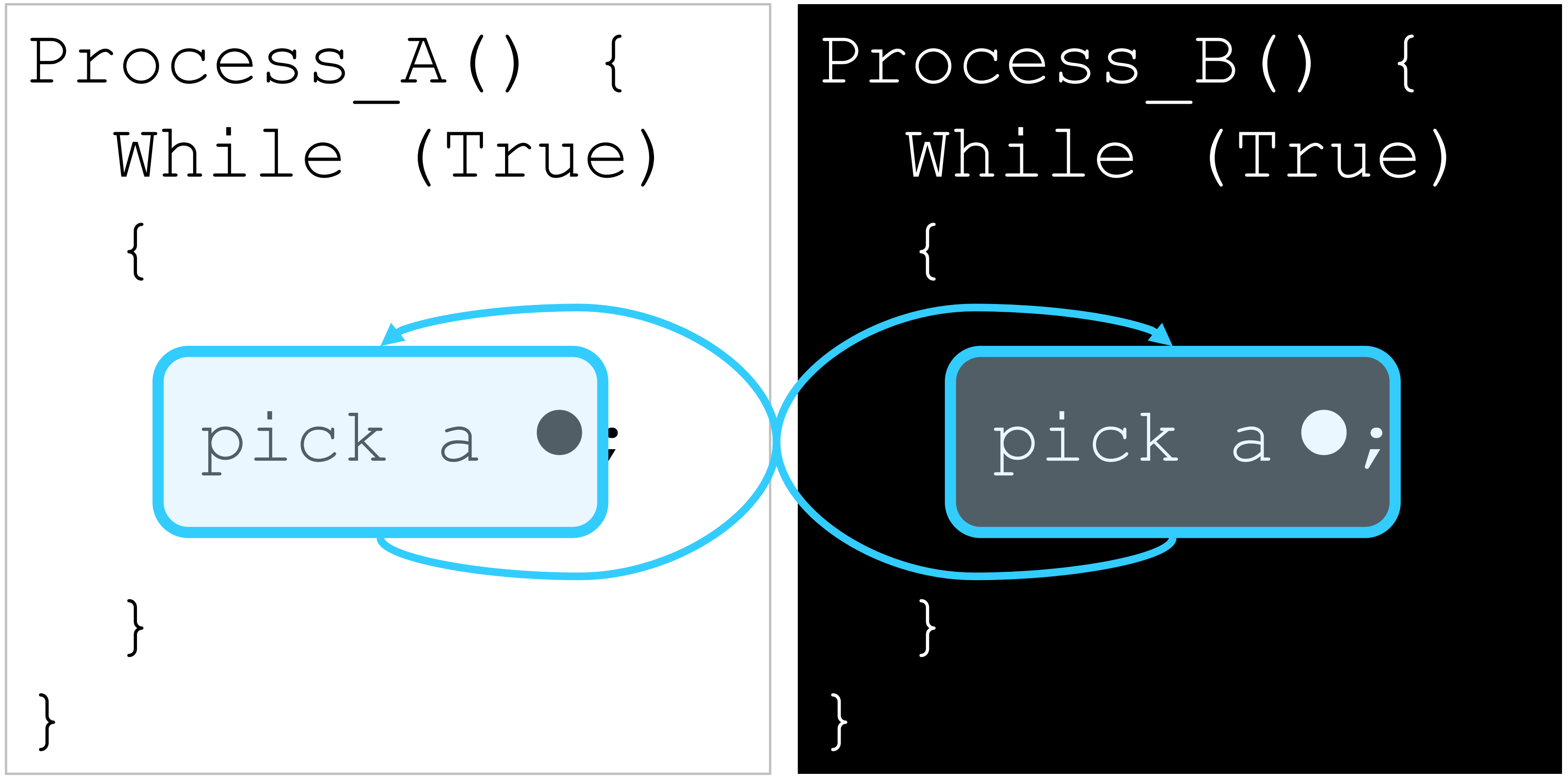
- Use PV primitives to implement process synchronization
  - Unlike process mutual exclusion, the semaphore in process synchronization is only related to the process and the restricted process, not to the whole group of concurrent processes. Therefore, the semaphore is called private semaphore.
  - Steps to realize process synchronization by using PV primitive :
    - 1) determine the relationship between processes is synchronous, set private semaphore for each concurrent process;
    - 2) assign initial value to private semaphore;
    - 3) use PV primitive and private semaphore to specify the execution sequence of each process.
- Let's add a condition to example 1 that the processes are synchronized.

Example 1+

- Additional constraints :
    - (3) When one process picks a piece (black or white), it must let another process pick a piece (white or black). So, the two process must be executed by turns.

Example 1+

- Analysis



```
Process_A() {
  While (True)
  {

    pick a ●;

  }
}
```

```
Process_B() {
  While (True)
  {

    pick a ○;

  }
}
```

Example 1+

- Analysis
  - Step 1: determine the relationship between processes. According to functions (1), (2) and (3), the relationship between processes is synchronous.
  - Step 2: determine the semaphore and its value. Processes A and B share the public resource of the box, but the two processes must take turns to pick different colored pieces, so they should exchange messages each other.
    - For process A, it sets a private semaphore B to determine whether process A can pick a black piece, with an initial value of 1.
    - For process B, a private semaphore W is used to determine whether process B can pick a white piece, and the initial value is 0.
    - Of course, you can also set the initial value of B to be 0 and W to be 1.

Example 1+

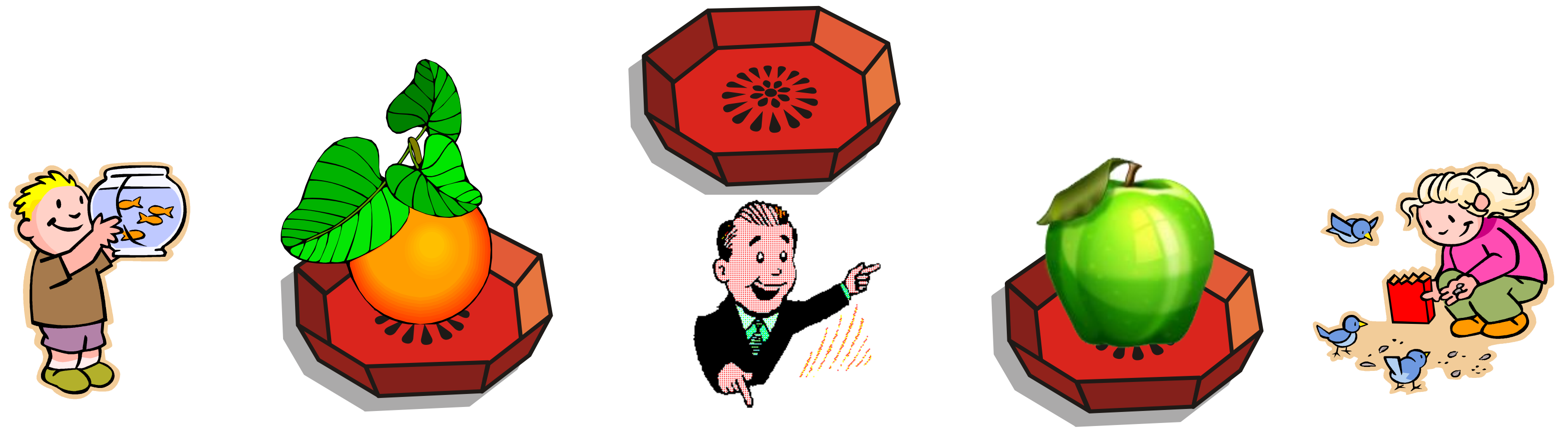
- Implementation: semaphore B=1, W=0; (or B=0, W=1)

```
Process_A() {
  While (True)
  {
    Wait(B);
    pick a ●;
    Signal(W);
  }
}
```

```
Process_B() {
  While (True)
  {
    Wait(W);
    pick a ●;
    Signal(B);
  }
}
```

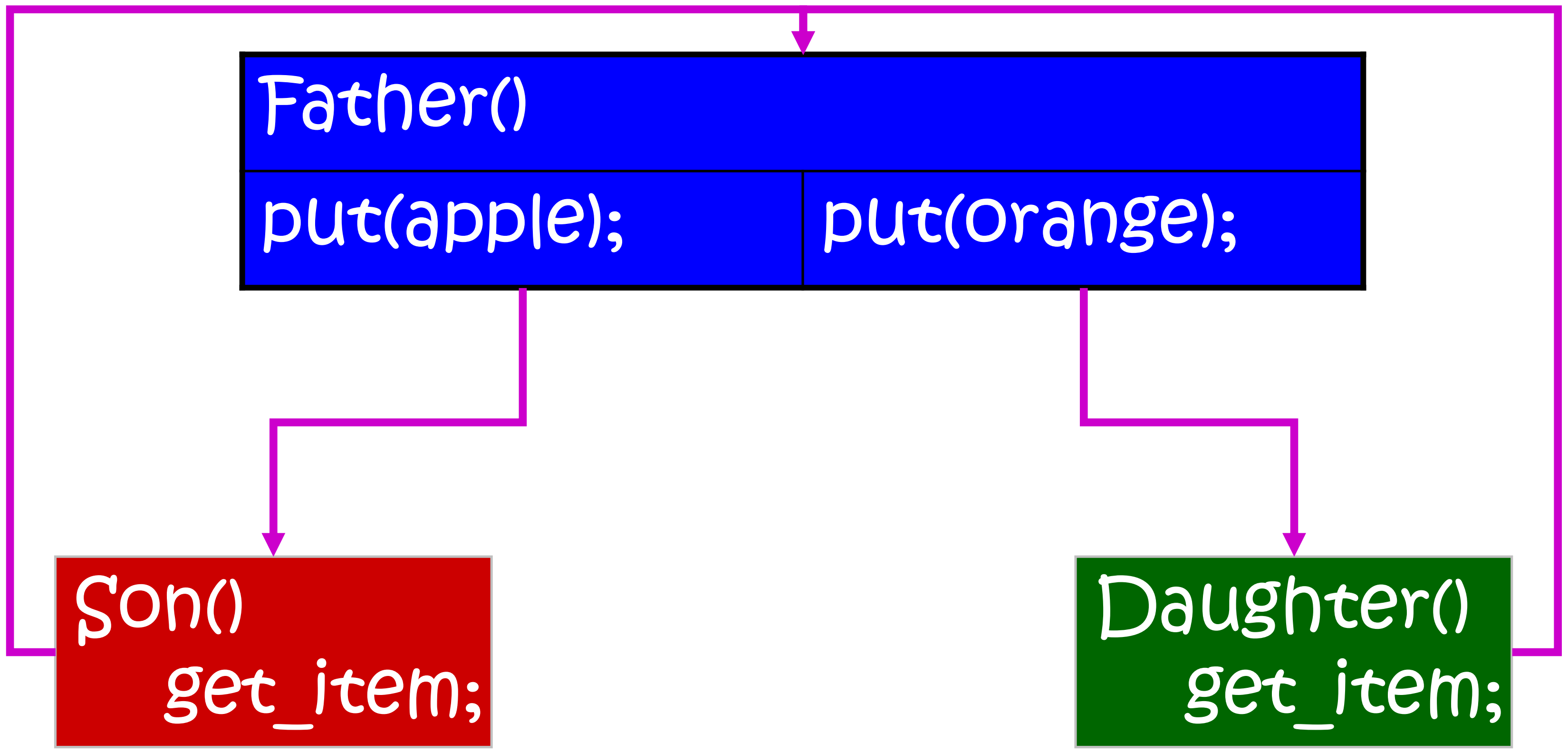
Example 2

- There is an empty plate on the table, which allows only one fruit. The father can put fruit in the plate, and only oranges and apples are put. The son only eats the oranges in the plate, while the daughter waits for the apples in the plate. Only one fruit can be placed at a time when the plate is empty.
- Please use P,V primitives to implement the three concurrent processes: **father**, **son** and **daughter**.



Example 2

- Analysis



Example 2

- Semaphore empty=1;
- Semaphore orange=0,
- Semaphore apple=0;

```
Father() {
    while (TRUE) {
        wait(empty);
        put_item;
        if (item==apple)
            signal(apple);
        else
            signal(orange);
    }
}
```

```
Son() {
    while (TRUE) {
        wait(orange);
        get_item;
        signal(empty);
    }
}
```

```
Daughter() {
    while (TRUE) {
        wait(apple);
        get_item;
        signal(empty);
    }
}
```

Example 3

- There are three concurrent processes R, M, and P that share the same buffer, assuming that the buffer can hold only one record.
  - Process R is responsible for reading information from the input device and putting it into the buffer after each record is read in.
  - Process M processes the read records in the buffer.
  - Process P prints out the processed record.
  - After the input record is processed and output, the buffer can store the next record.
- Write concurrent programs that they can execute correctly.

Example 3

- PV primitive: three processes share a buffer, they must work synchronously, three semaphores can be defined:
  - S1: indicates whether the read record can be put into the buffer, with an initial value of 1.
  - S2: indicates whether records in the buffer can be processed, with an initial value of 0.
  - S3: indicates whether the record is processed and can be output, with an initial value of 0.

Example 3

- Implementation: semaphore S1=1, S2=0, S3=0;

```
process R {
    while (TRUE)    {
        // read record;
        P(S1);
        // put record in to buffer;
        V(S2);
    }
}
```

```
process M {
    while (TRUE)    {
        P(S2);
        // process record;
        V(S3);
    }
}
```

```
process P {
    while (TRUE)    {
        P(S3);
        // print processed record;
        V(S1);
    }
}
```

Example 4

- On a bus, the activities of the driver and the conductor are:
  - Driver: start the vehicle, drive normally, stop at the station.
  - Conductor: get passengers on the bus, close the door, sell ticket, open the door, let passengers get off.
- Use PV operations to control them.

Example 4

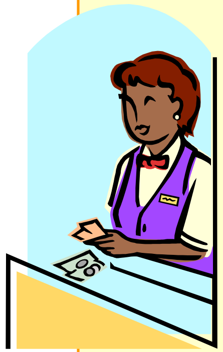
- Analysis
  - Step 1: determine the relationship between processes.
    - The conductor can only work when the driver stops at the station.
    - Similarly, the conductor closes the door before the driver can work.
    - So, there is a synchronous relationship between the driver and the conductor.
  - Step 2: determine the semaphore and its value.
    - Since the driver and the conductor need to exchange messages, the driver process sets a private semaphore run, which is used to determine whether the driver can do the work. The initial value is 0.
    - The conductor process sets a private semaphore, stop, to determine whether to stop and whether the conductor can open the door, with an initial value of 0.

Example 4

- Semaphore stop=0, run=0;

```
driver() {
    while(TRUE) {
        P(run)
        //start the vehicle;
        //drive normally;
        //stop at the station;
        V(stop);
    }
}
```

```
conductor() {
    while(TRUE) {
        //Get passengers on;
        //close the door;
        V(run);
        //sell tickets;
        P(stop);
        //open the door;
        //let passengers off;
    }
}
```



Summary: Use semaphore to solve problem

1. Determine concurrent and sequential operations;
2. Determine the rules of mutual exclusion and synchronization;
3. Determine the procedure of mutual exclusion and synchronization;
4. Determine the number and meaning of semaphores;
5. Determine the initial value of the semaphore;
6. Determine the position of P and V operations.

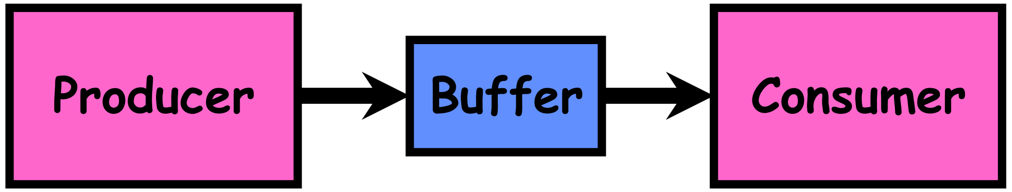
Note: whenever a process needs to be blocked, it must be thought of waking it up at some point.

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;
```

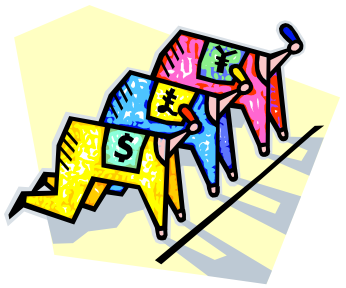
Producer and Consumer

```
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++;
}
```

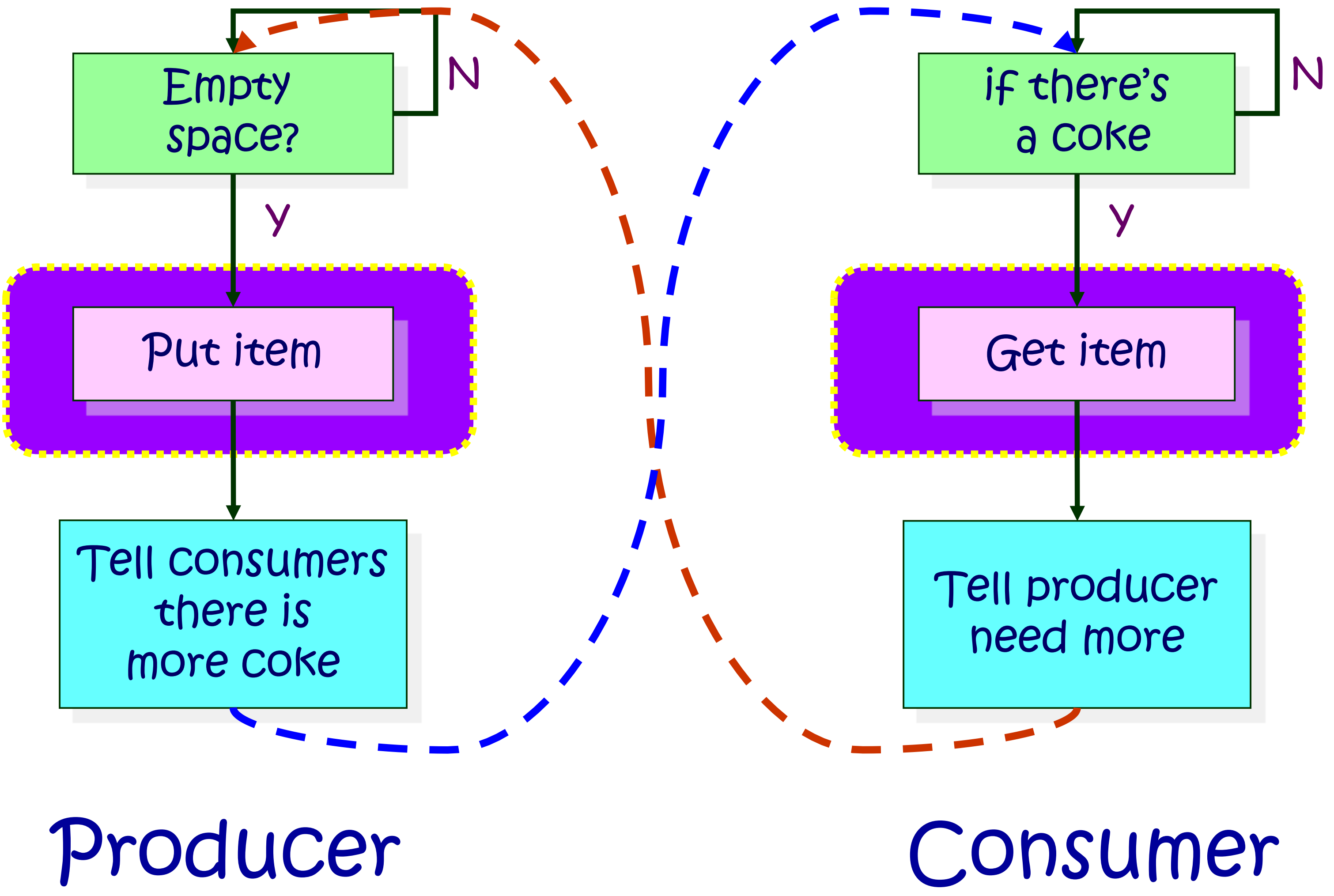


```
while (true)  {
    while (count == 0)
     ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
}
```

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`

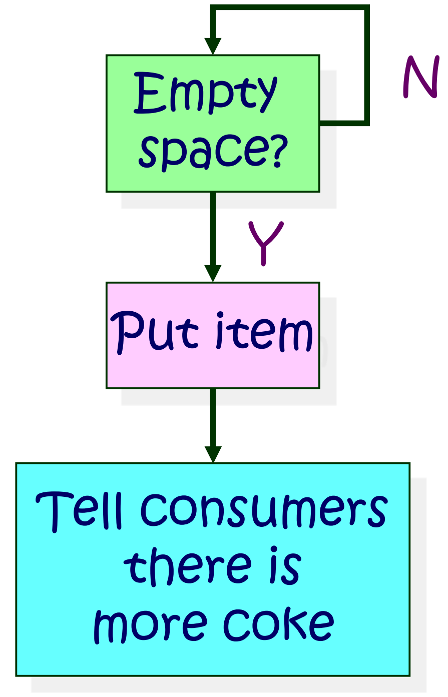
Full Solution

```
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

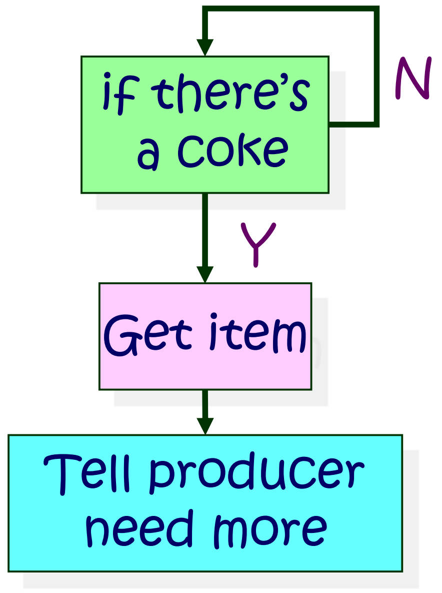
```
while (true) {

     //   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?
  - Yes! Can cause deadlock
  - Why?

```
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

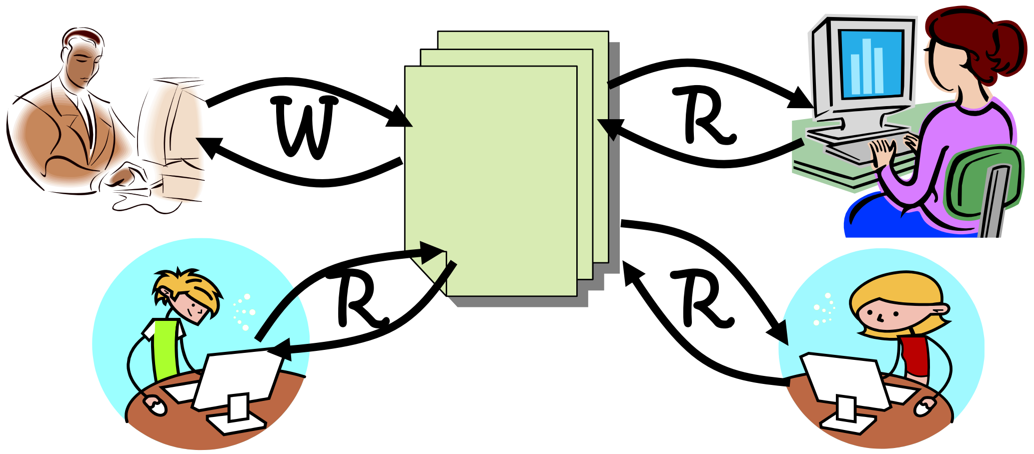
```
Semaphore full = 0;          // Initially, no coke
Semaphore empty = N;         // Initially, num empty slots
Semaphore mutex = 1;         // No one using machine

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.



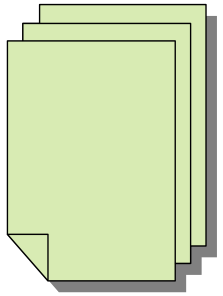
Basic Readers/Writers Solution

- 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
```

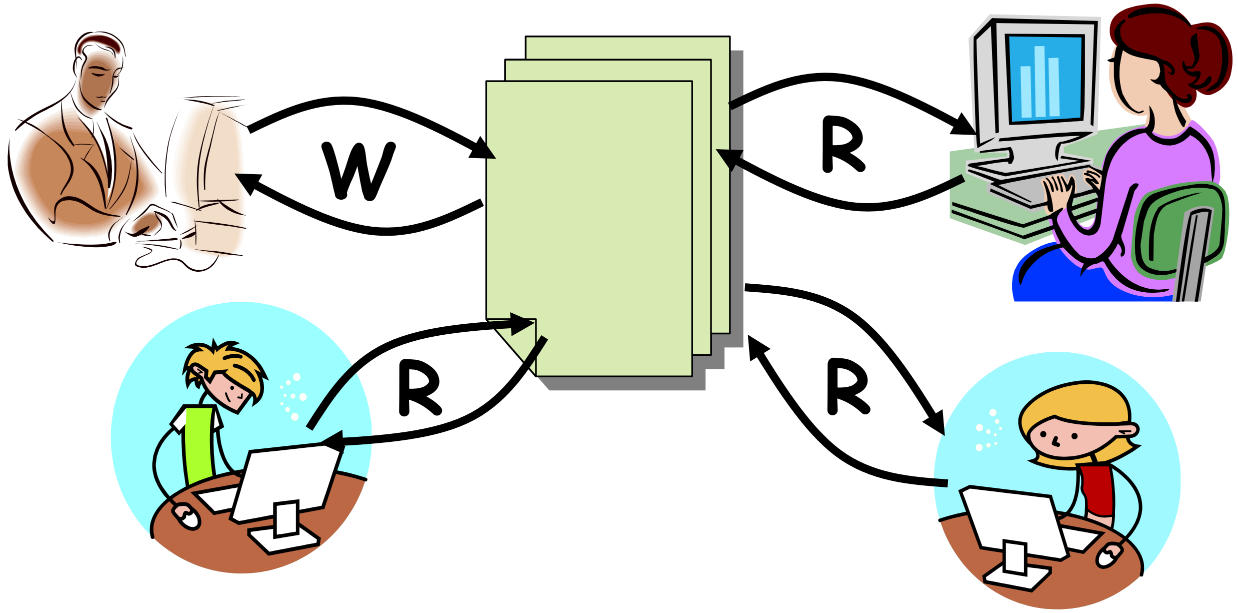
  - Writer()

```
    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

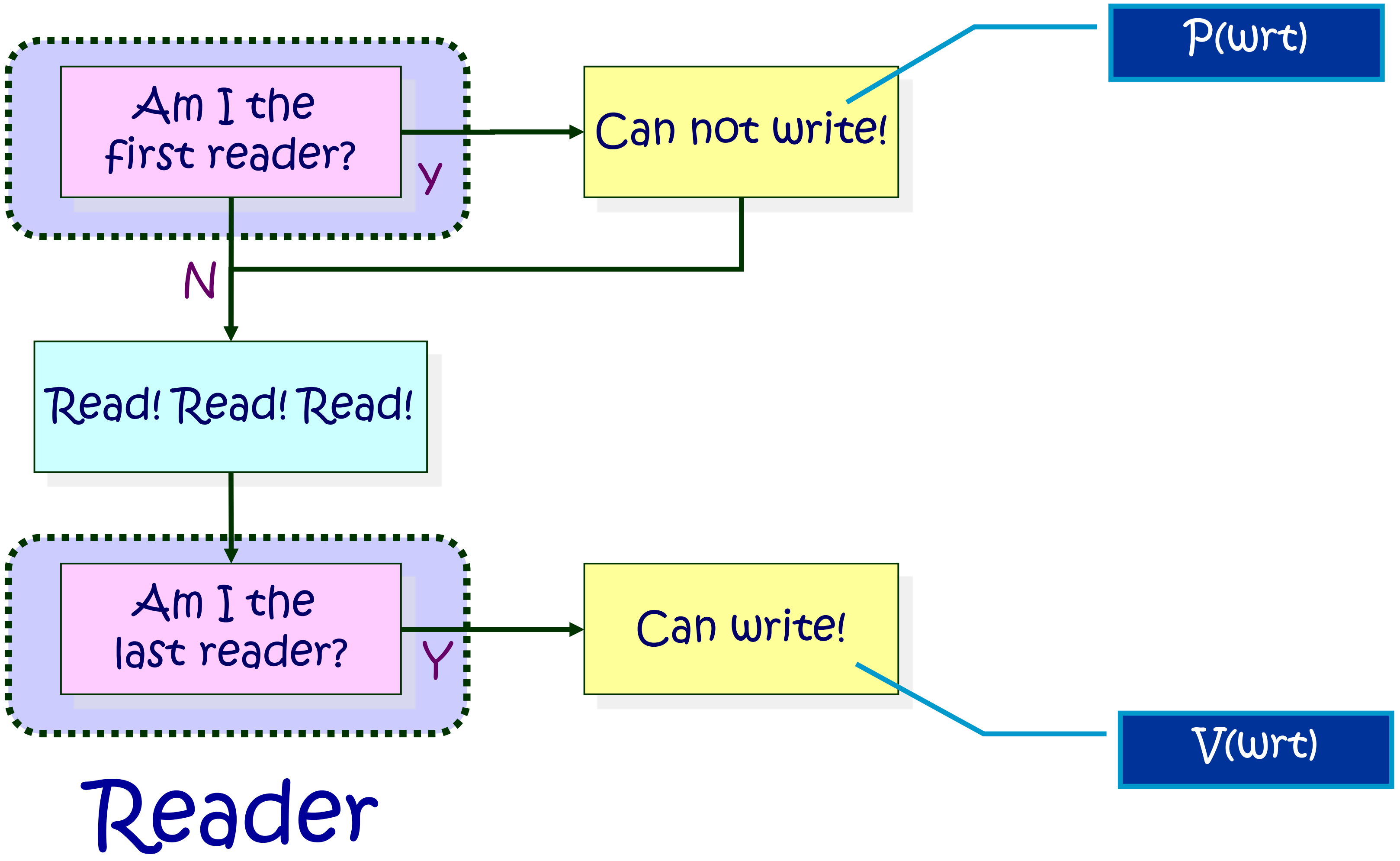
- The solution is sub-optimal.
  - Because it is possible that a reader R1 might have the lock, and then another reader R2 request access.
  - It would be foolish for R2 to wait until R1 was done before starting its own read operation.
  - Instead, R2 should start right away.
- This is the motivation 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);
}
```

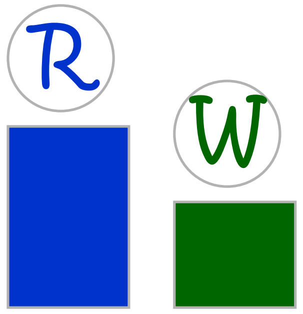
Readers-Writers Problem: Solution #1



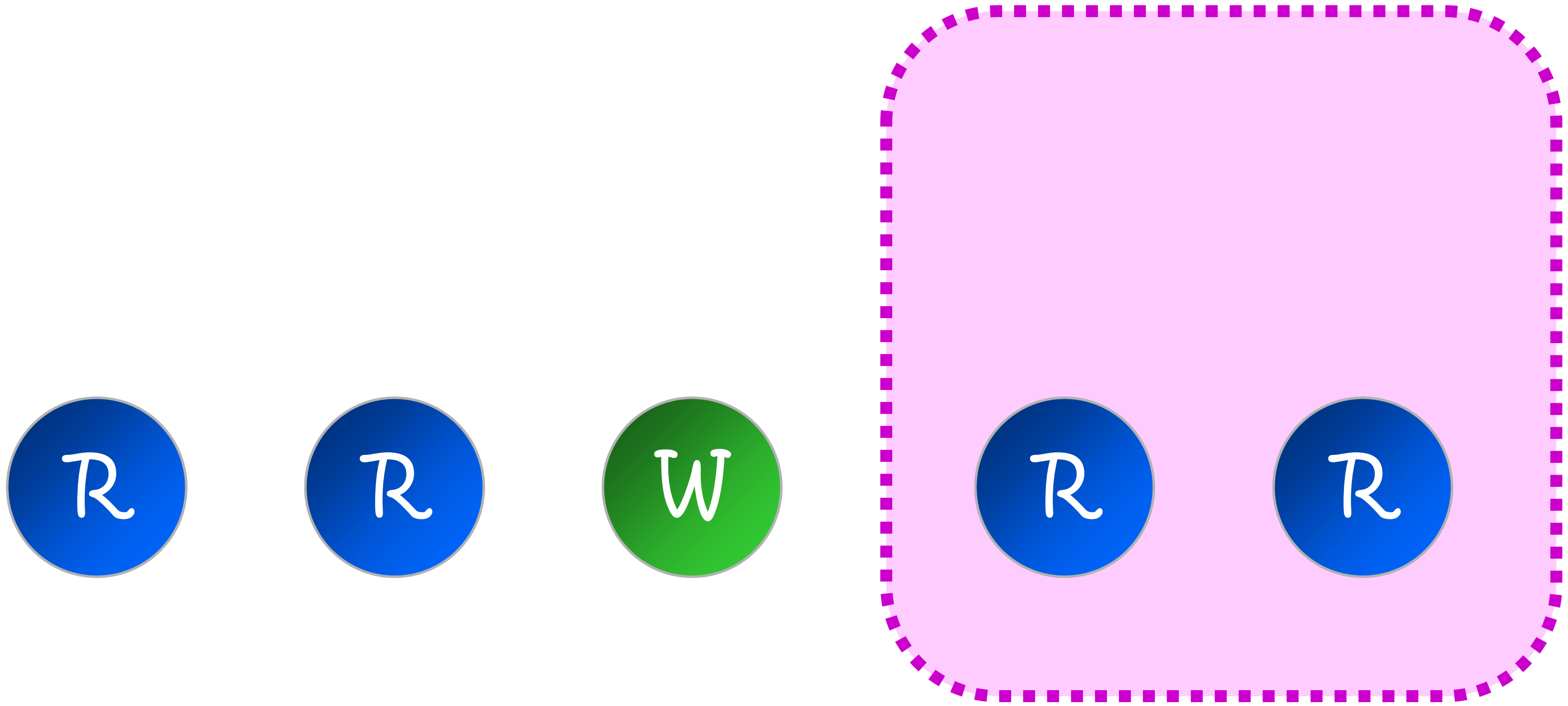
Readers-Writers Problem: Solution #1

- 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

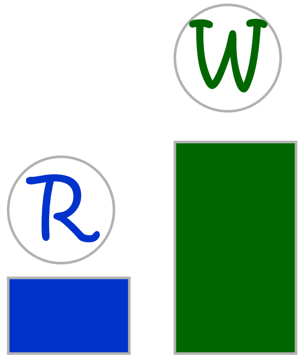
- int readcount = 0, writecount = 0;
- semaphore x = 1, y = 1, wrt = 1, red = 1;
- The structure of a writer process

```
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

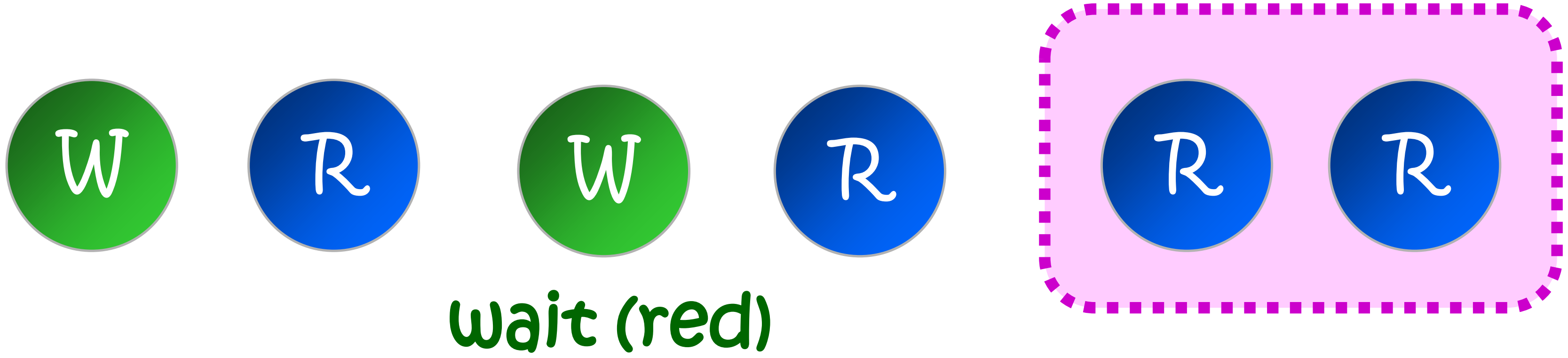
```
while (true) {
    wait(red);
      wait(x);
        readcount++;
        if (readcount == 1)  wait(wrt);
      signal (x);
    signal(red);
      // reading is performed
    wait (x);
      readcount--;
      if (readcount == 0)  signal(wrt);
    signal(x);
}
```



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

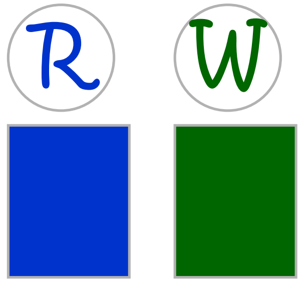
- int readcount = 0;
- semaphore mutex = 1, wrt = 1, S=1;
- The structure of a writer process

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

Readers-Writers Problem: Solution #3

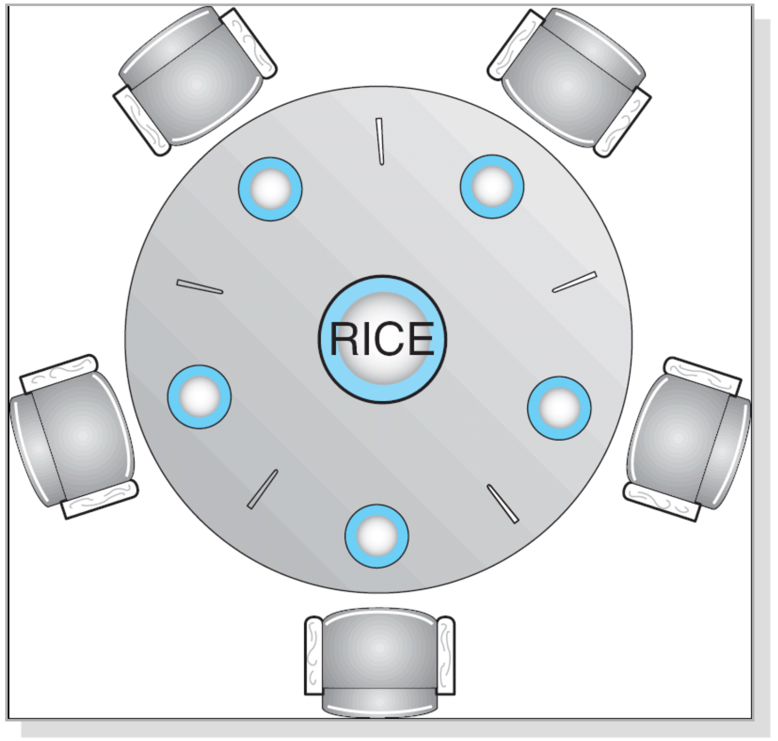
- The structure of a reader process

```
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);
}
```

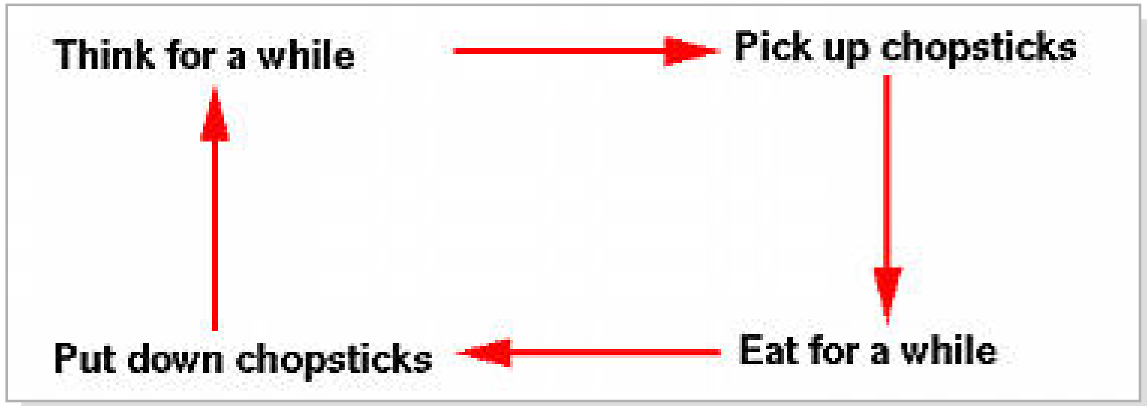


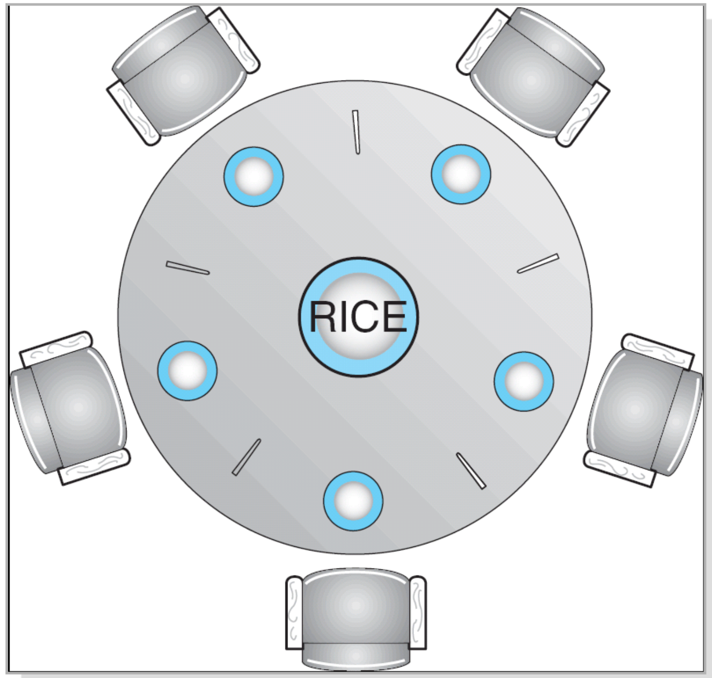
Dining-Philosophers Problem

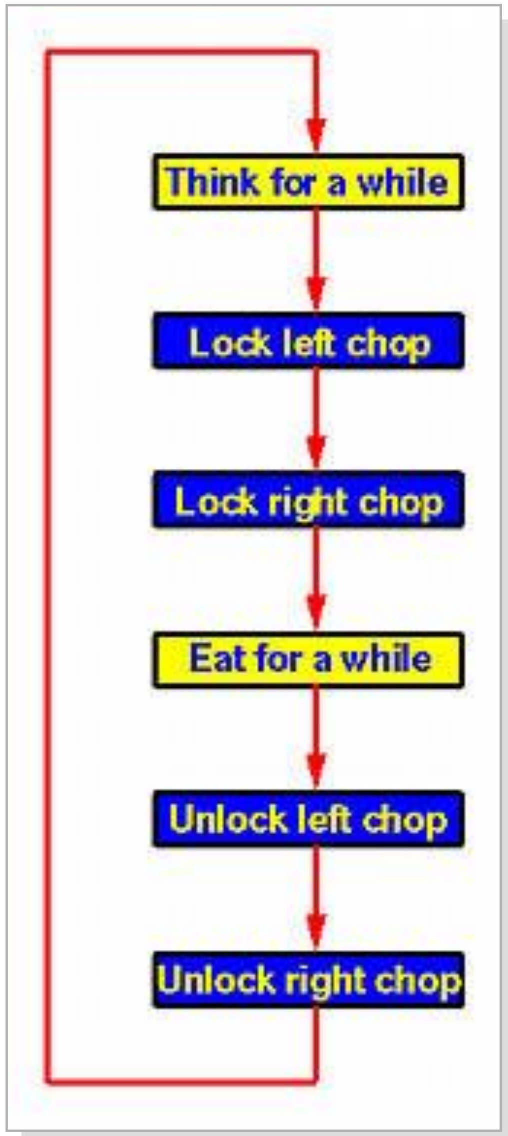
- Five philosophers, either thinking or eating
- To eat, two chopsticks are required
- Taking one chopstick at a time



Dining-Philosophers Problem

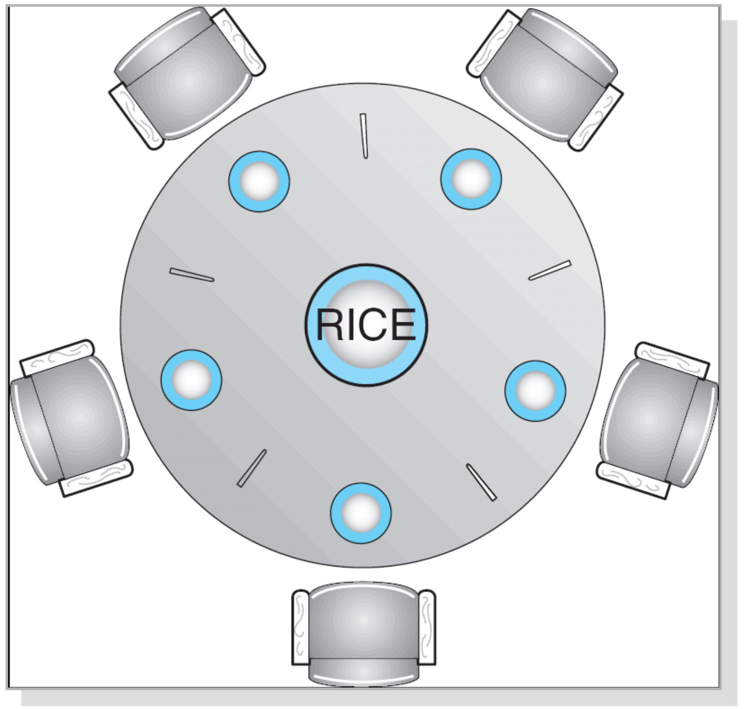






Dining-Philosophers Problem

- 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

```
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

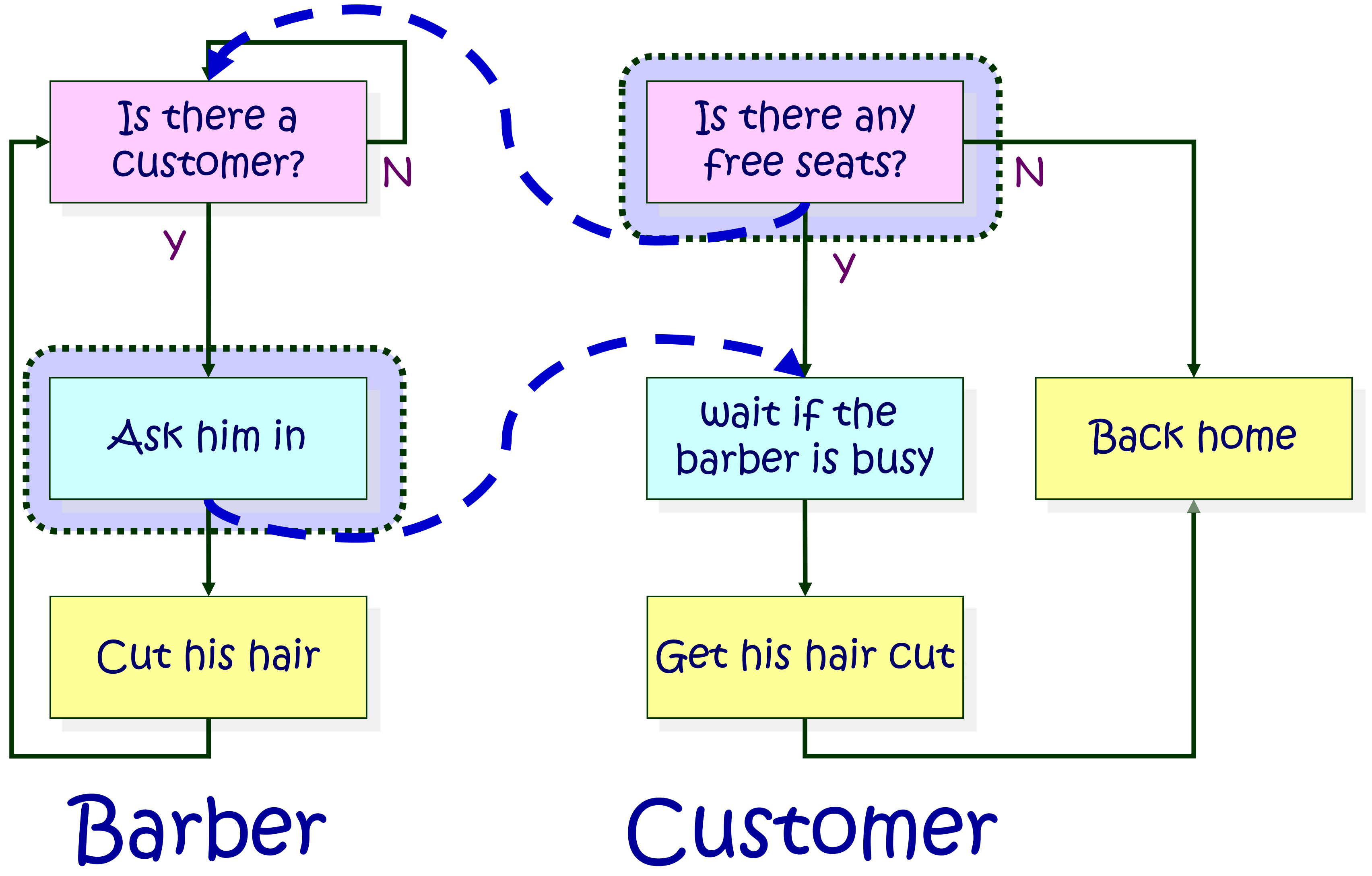
Sleeping Barber Problem

- 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

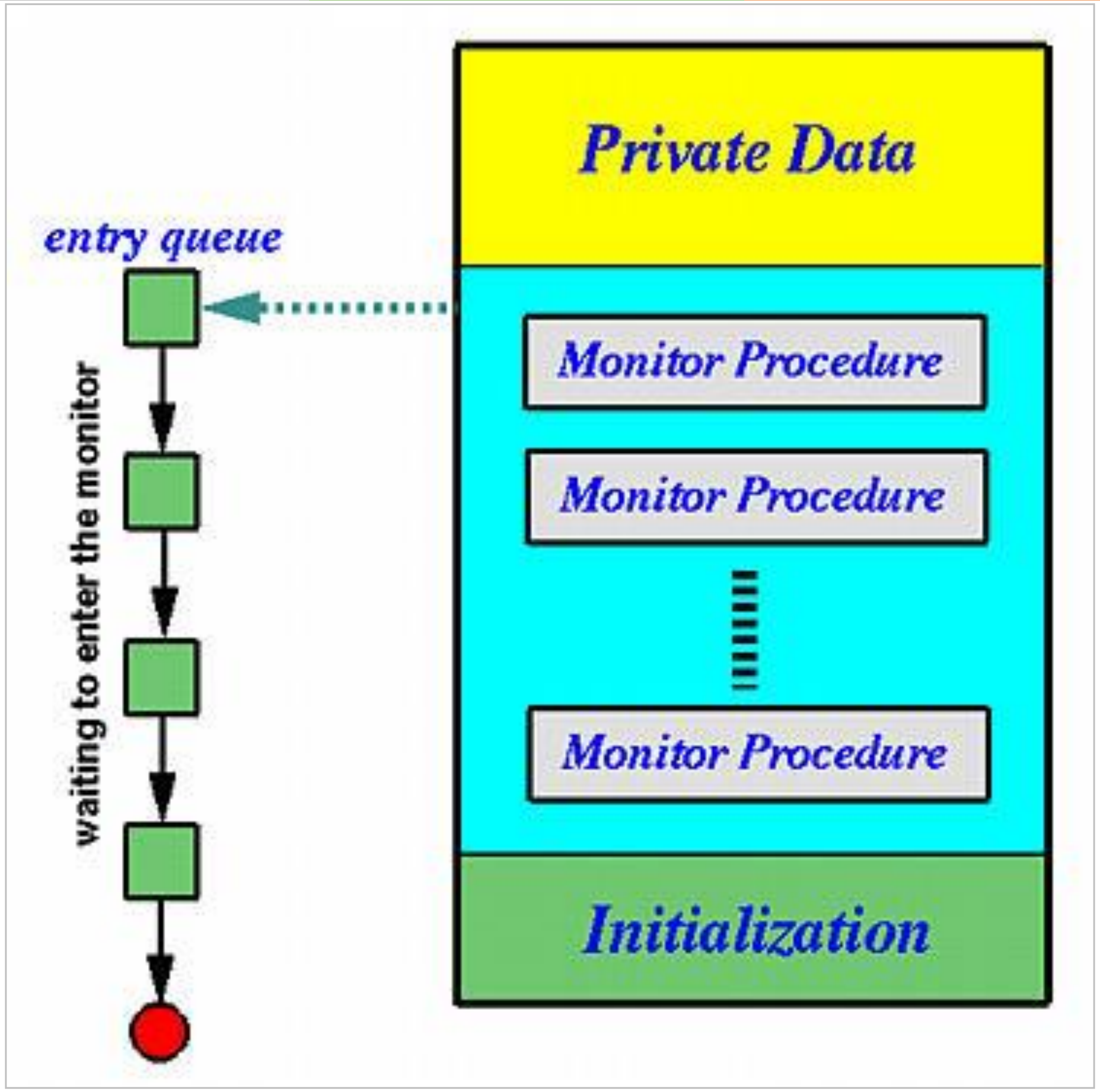
- The representation of a monitor type consists of
  - declarations of variables whose values define the state of an instance of the type
  - procedures or functions that implement operations on the type.
- A procedure within a monitor can access only variables defined in the monitor and the formal parameters.
- 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


entry queue
waiting to enter the monitor
Private Data
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;

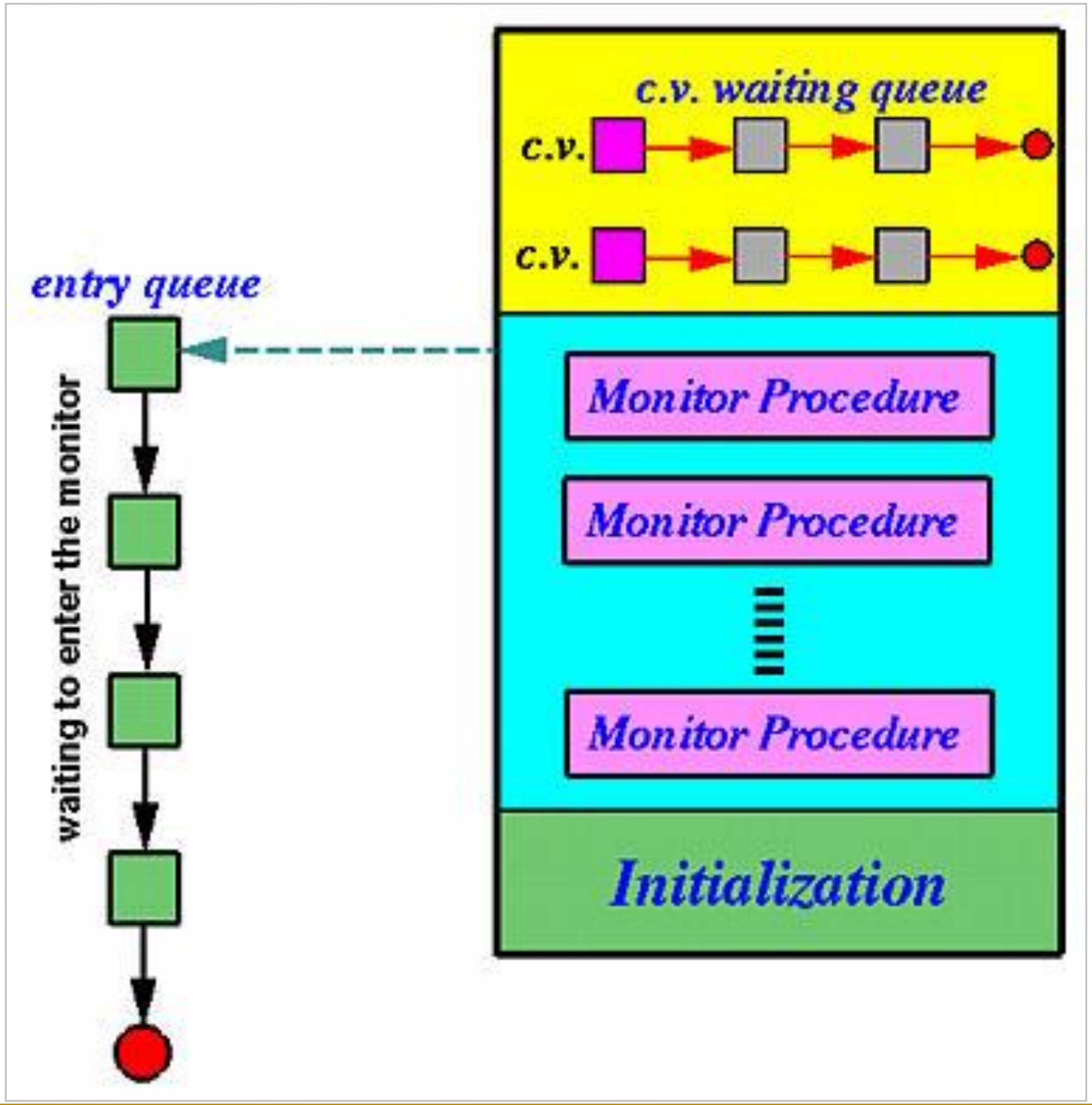
Monitors-Condition Variables

- 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


c.v. waiting queue
c.v.
c.v.
entry queue
waiting to enter the monitor
Monitor Procedure
Monitor Procedure
Monitor Procedure
Initialization

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:

```
dinningPhilosopher.pickup (i)
        EAT
dinningPhilosopher.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;
  }
}
```

Synchronization Examples

Windows XP Synchronization

- Interlock
- CriticalSection
- Event
- Mutexes
- Semaphores

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

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()`

Homework