# 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 count 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);
```

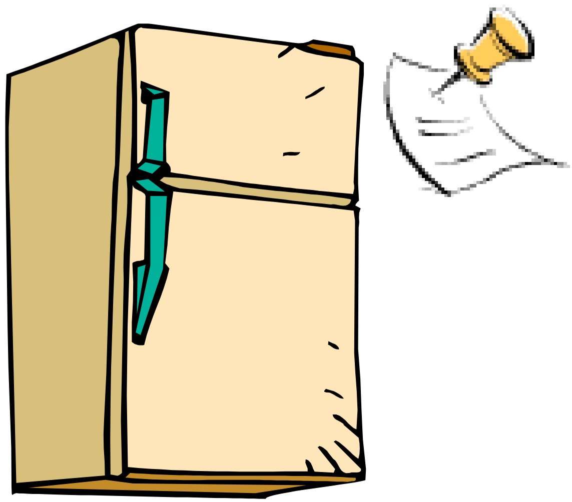
# Too Much Milk: Correctness Properties

- 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;
    }
}
```



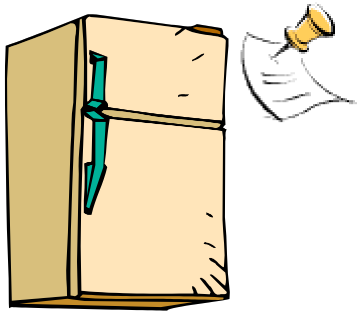
- Result?

# Too Much Milk: Solution #1

| Process A | Process B |
| --- | --- |
| if (no Milk)<br>    if (no Note) | |
| | if (no Milk)<br>    if (no Note) |
| leave Note;<br>buy milk;<br>remove Note; | |
| | leave Note;<br>buy milk;<br>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…



# 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

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

# 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

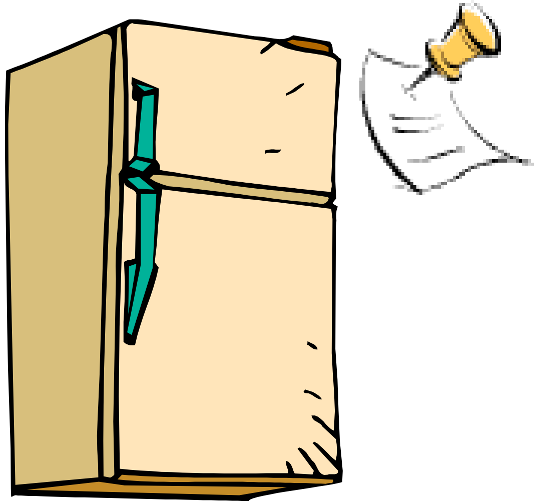
| Process A | Process B |
| --- | --- |
| leave note A; | |
| | leave note B; |
| if (no Note B)<br>  if (no Milk)<br>    buy Milk; | |
| | if (no Note A)<br>  if (no Milk)<br>    buy Milk; |
| remove note A; | |
| | remove note B; |

# 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

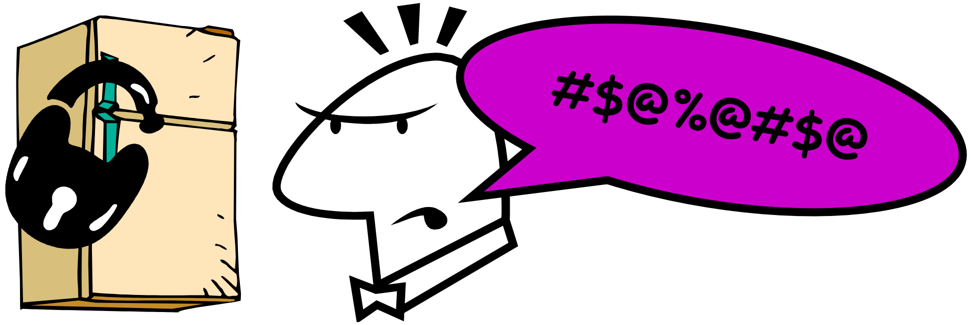
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

# 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:

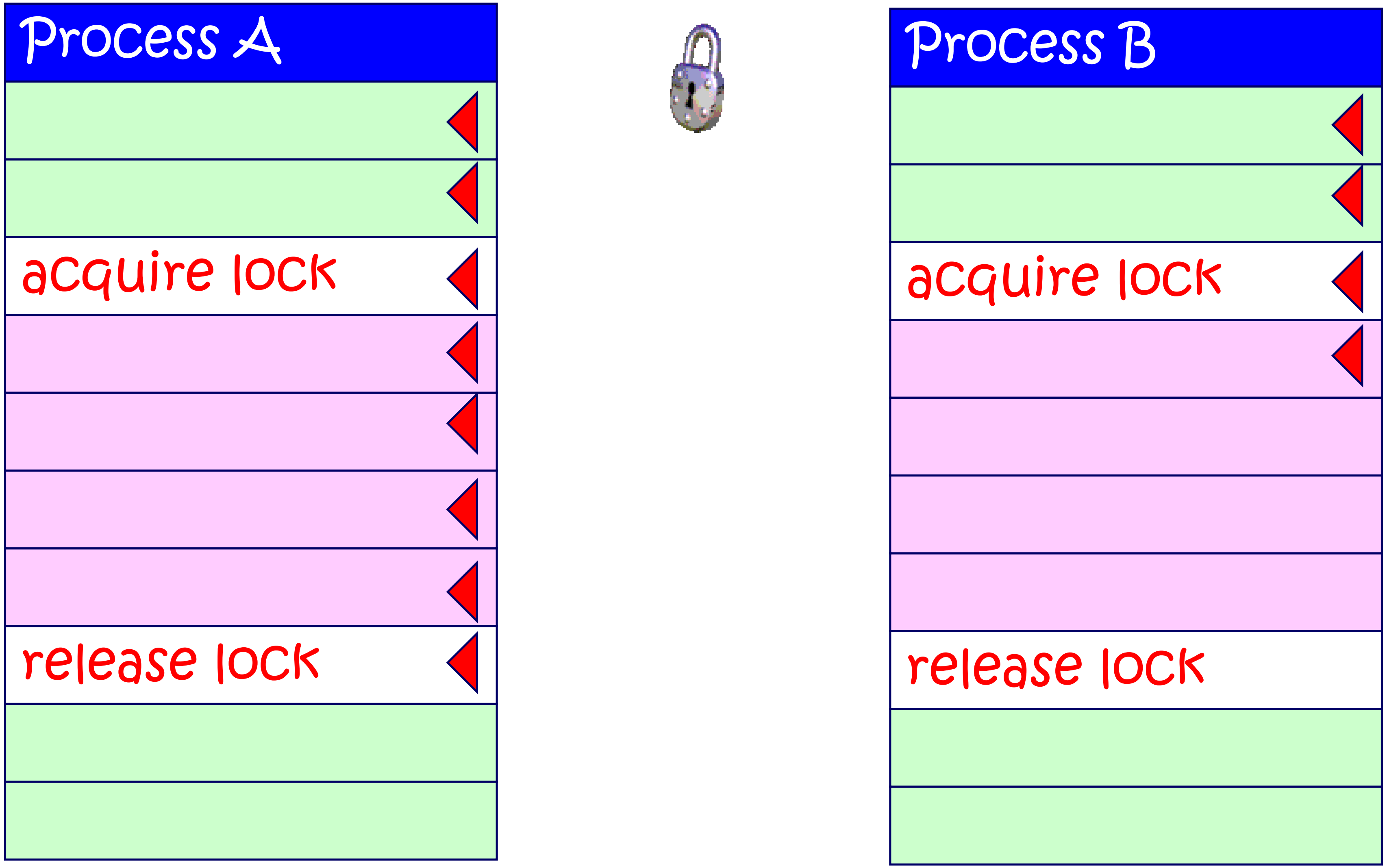
```
milklock.Acquire();
if (no milk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"

# 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:

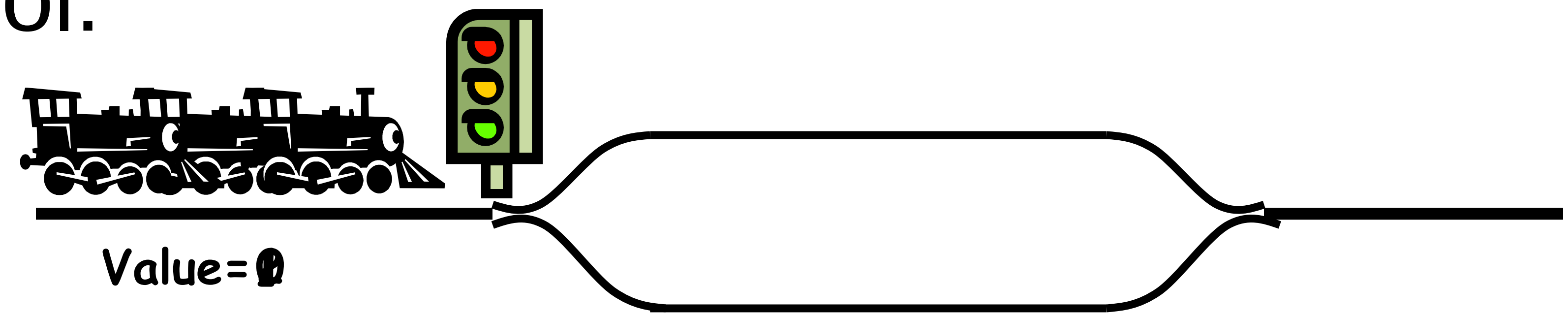
```
do {
     key = TRUE;
     while (key == TRUE)
          Swap (&lock, &key);

     //critical section

     lock = FALSE;

     //remainder section

} while (TRUE);
```

# Semaphores


SE
SE
SE
SE
SE
SE
812
0

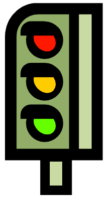
# Semaphores

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



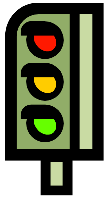
# Semaphores

- 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



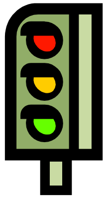
# Semaphores

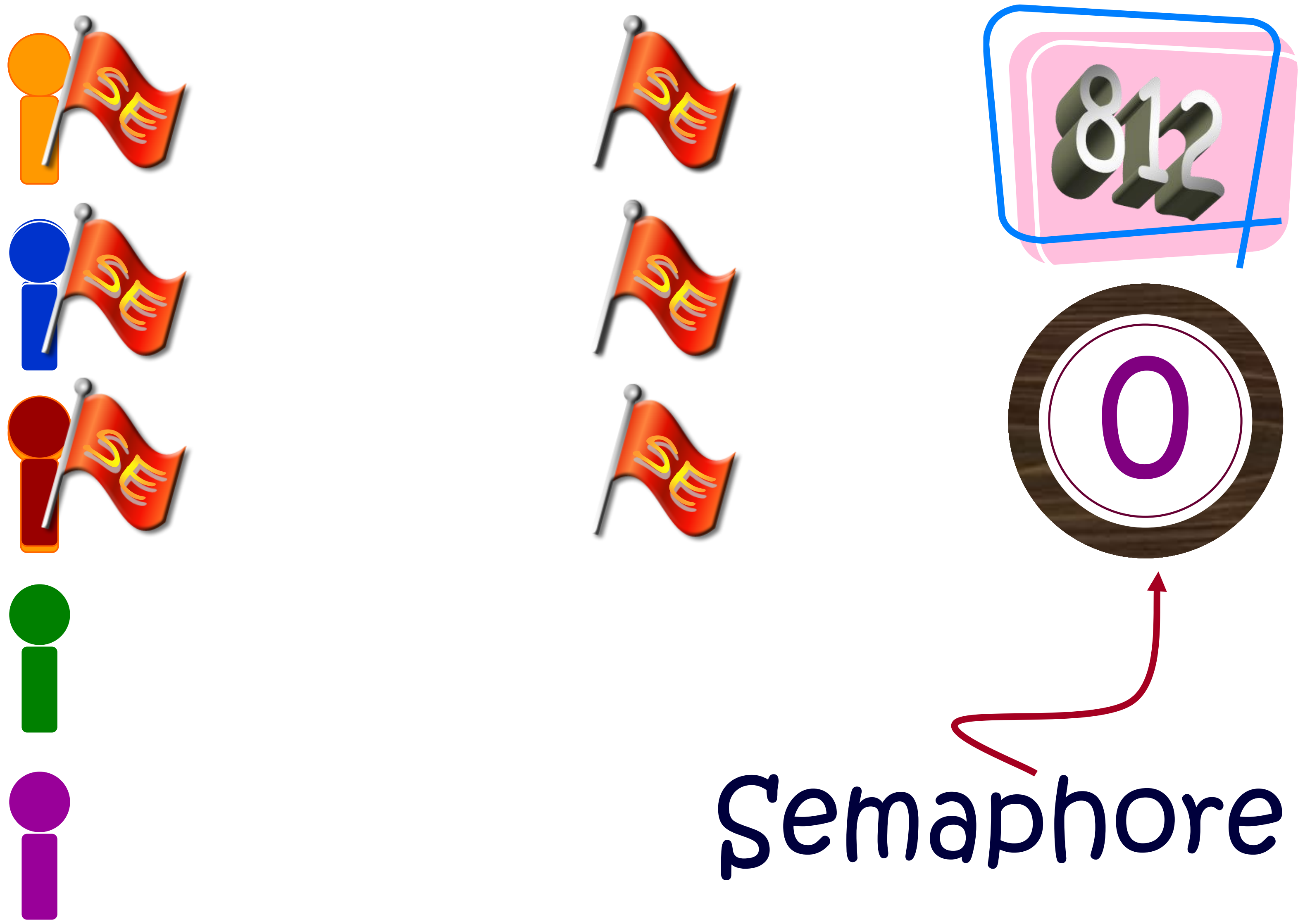
- The definition of **wait ()** is as follows:

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

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

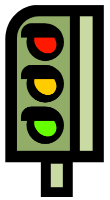
```
signal (S) {
   S++;
}
```




SE
812
0
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:

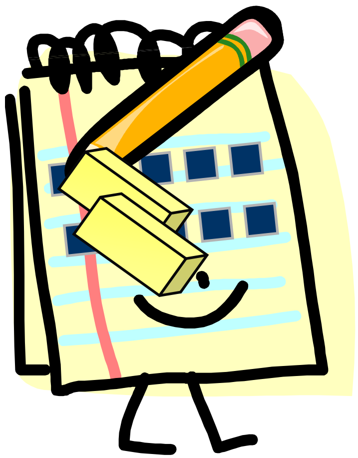
```
signal (S) {
   S++;
}
```


SE
SE
SE


SE
SE
SE

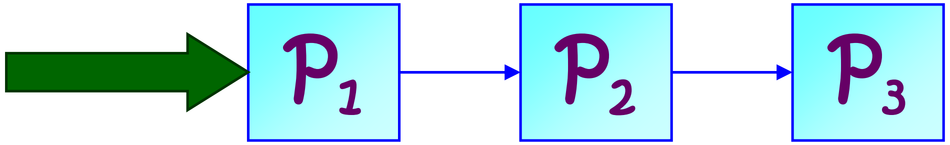

812


0

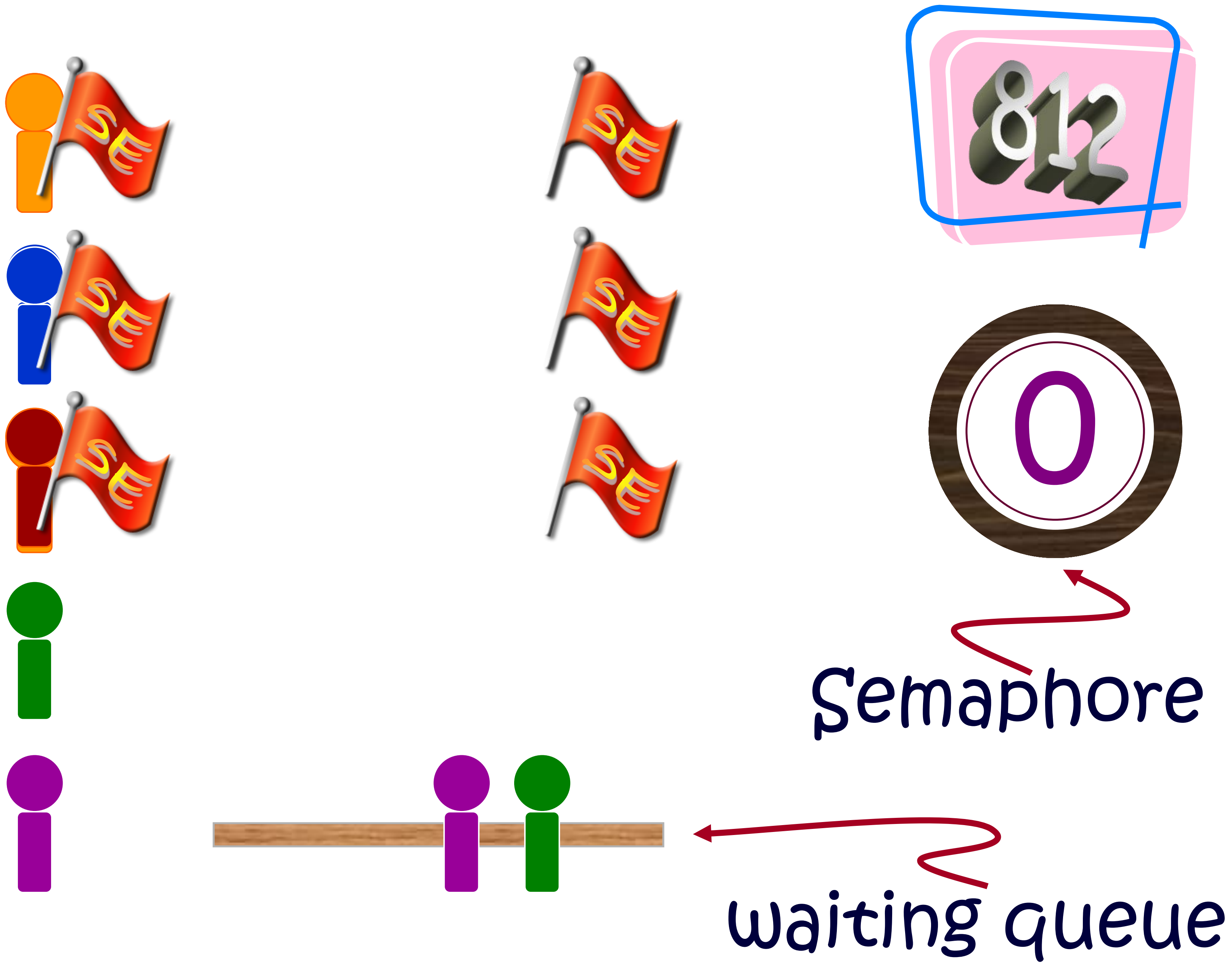


# Semaphore Implementation with no Busy Waiting

- 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);
      }
}
```

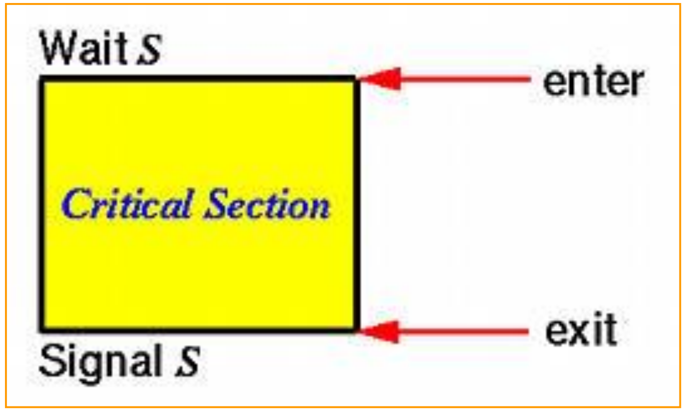
# Two Types of Semaphores

- **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.

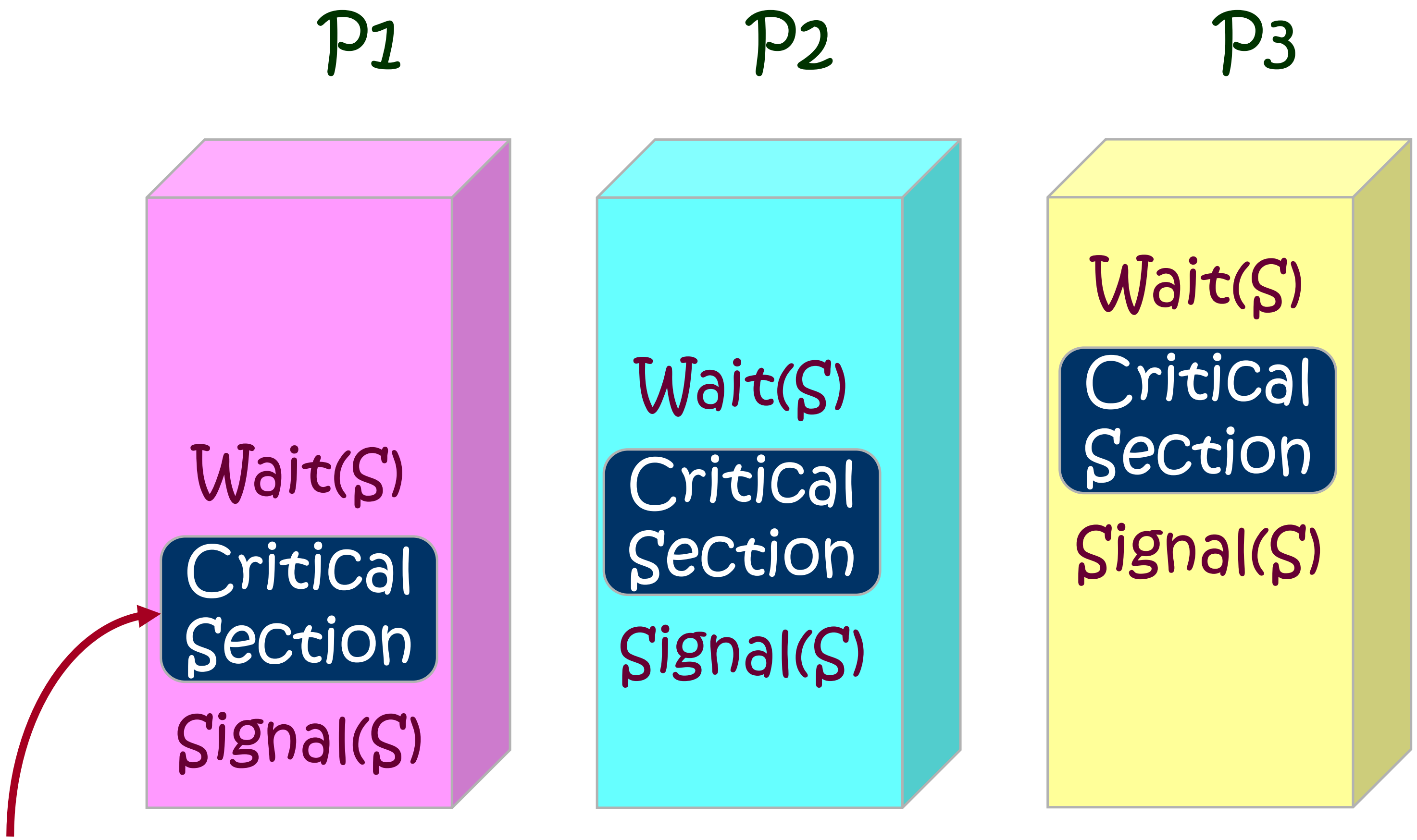
# Two Types of Semaphores

- Provides mutual exclusion

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



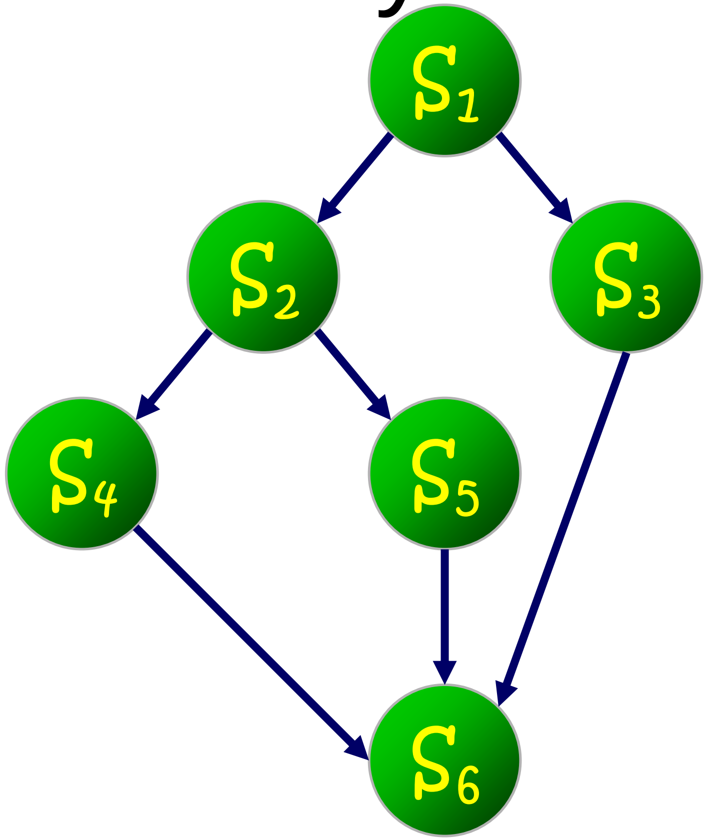
# Semaphore



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

# Precedence Graph

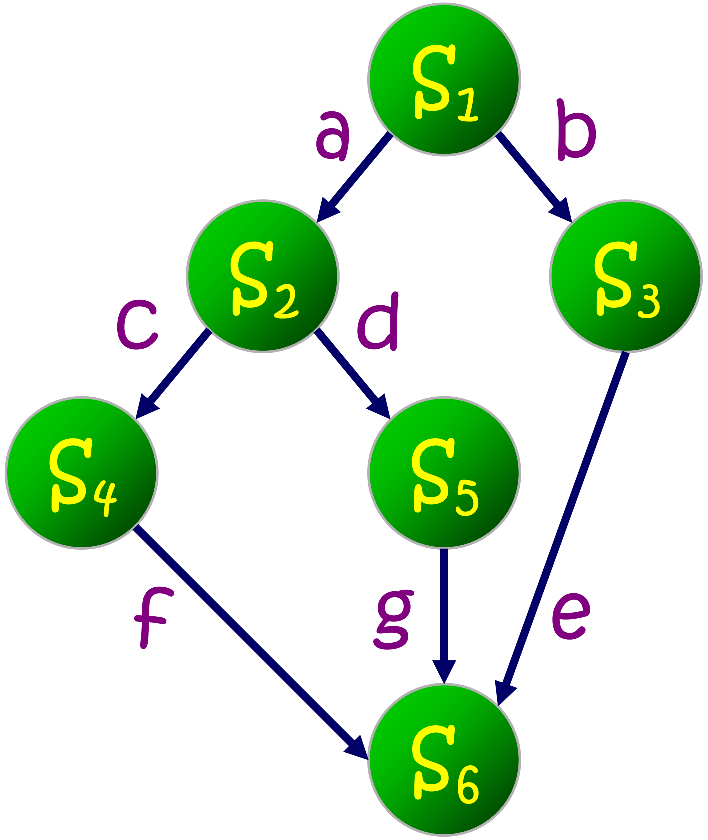
- A DAG (Directed Acyclic Graph)



# Semaphore

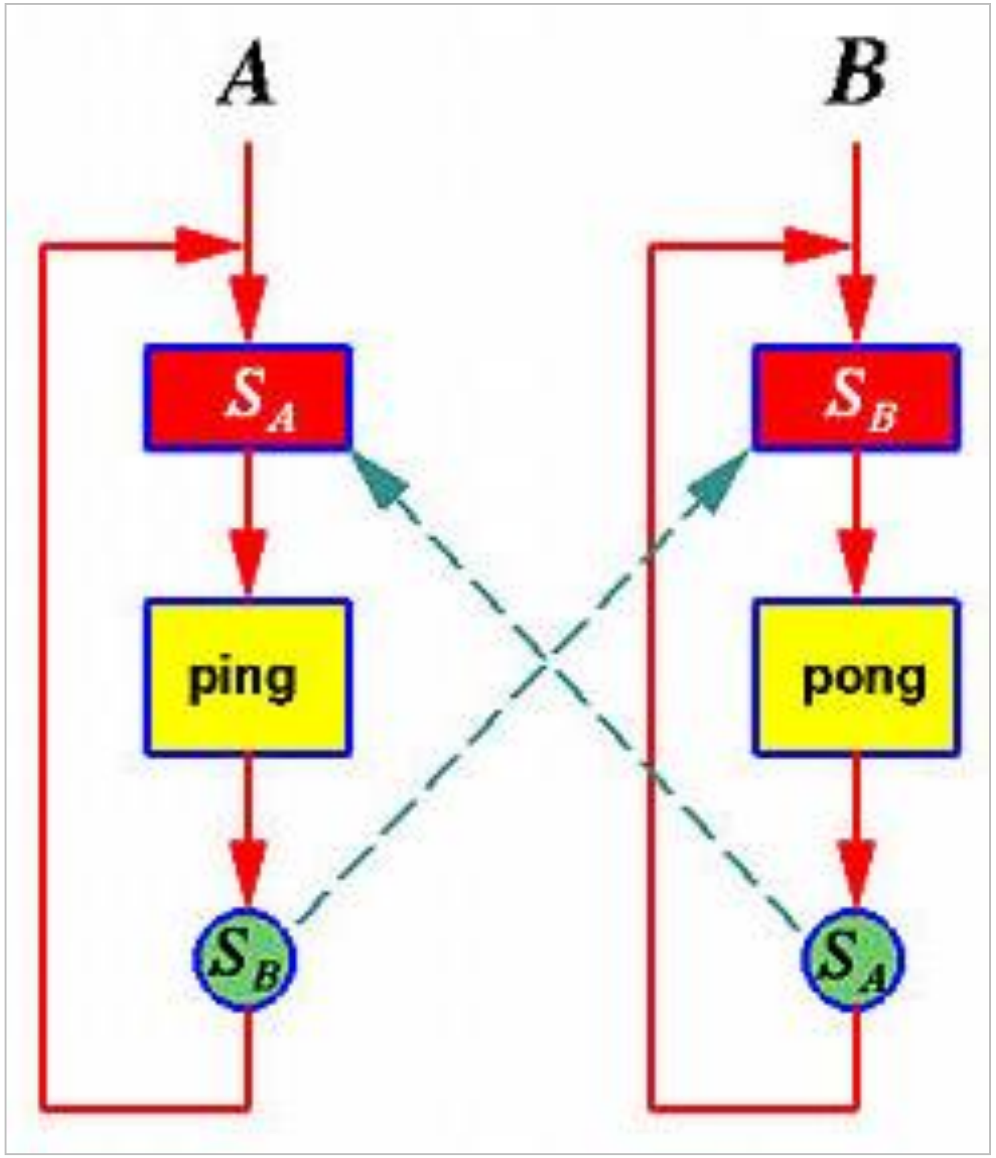
- Semaphore a, b, c, d, e, f, g;
- a=0, b=0, c=0, d=0, e=0, f=0, g=0;

- 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

| $P_0$ | $P_1$ |
|---|---|
| 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.