

11:05am

# Chapter 6: Synchronization

Prof. Li-Pin Chang  
National Chiao Tung University

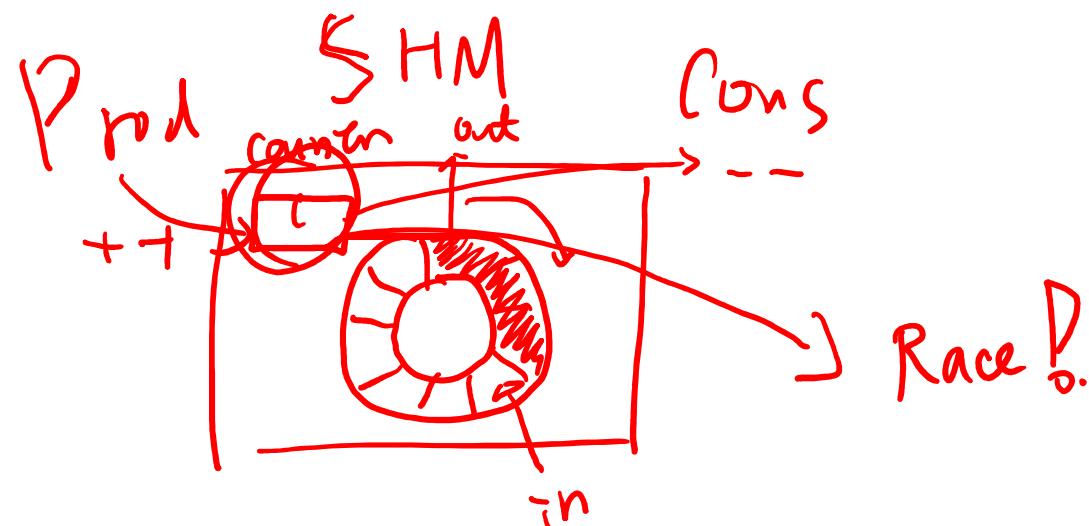


# Module 6: Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores → ~~加锁~~
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

- ① Item counter?
- ② busy waiting

## 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
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers. We can do so by having an integer **count** that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

# Producer

Item # ..

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

The diagram illustrates the execution flow of the producer code. A large red oval encloses the entire outer loop structure. Inside this oval, a red box highlights the condition of the inner loop, which is `count == BUFFER_SIZE`. Another red box highlights the post-increment statement `count++`. Red arrows point from the handwritten text "Item # .." towards the inner loop condition and the `count++` statement.

# Consumer

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

# Race Condition

- `count++` could be implemented as

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

不能打包

cxtsw

- `count--` could be implemented as

```
✓ register2 = count  
✓ register2 = register2 - 1  
✓ 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}

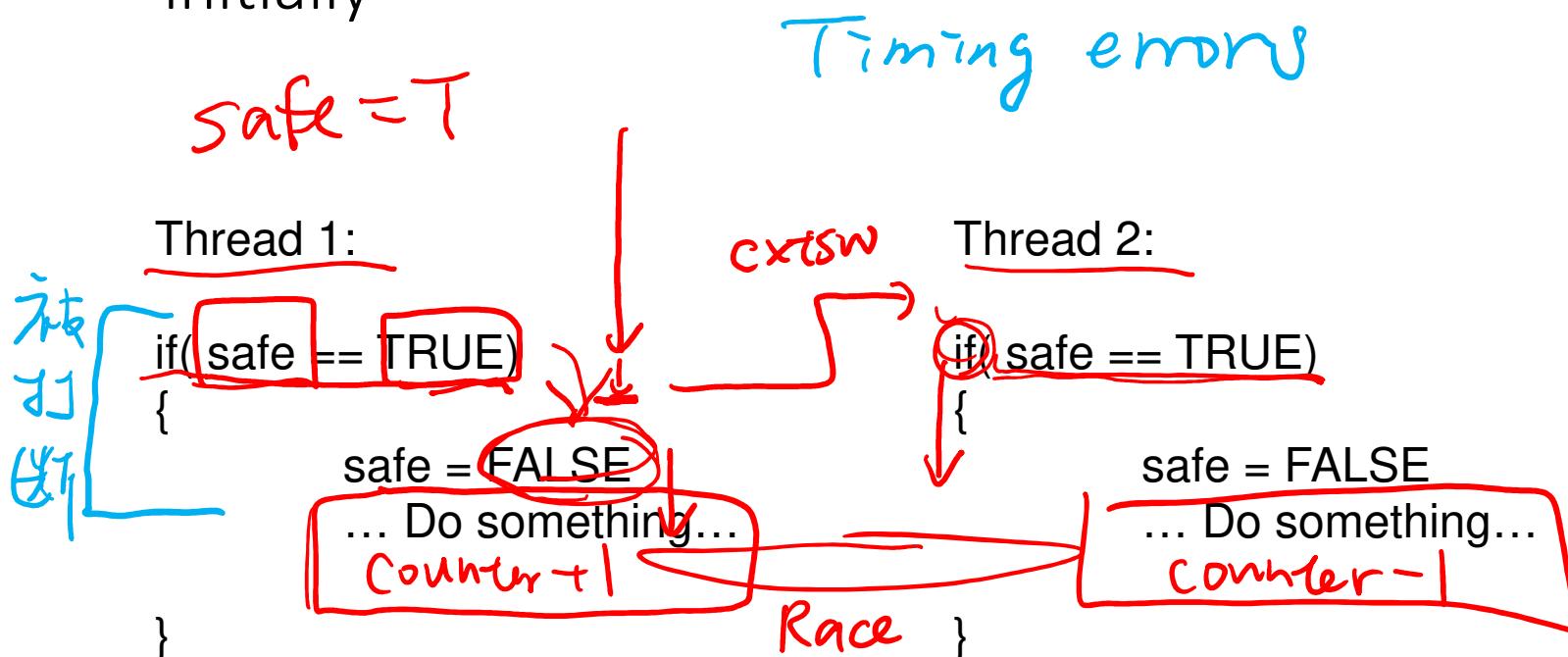
	r1	r2	count
			Mem
	5	?	5
	6	?	5
	6	5	5
	6	4	4
	6	5	5
	6	6	6
	6	4	4
			7

Counter may be 4 or 6 depends on the sequence of S4 and S5

Counter can even be 5 !?

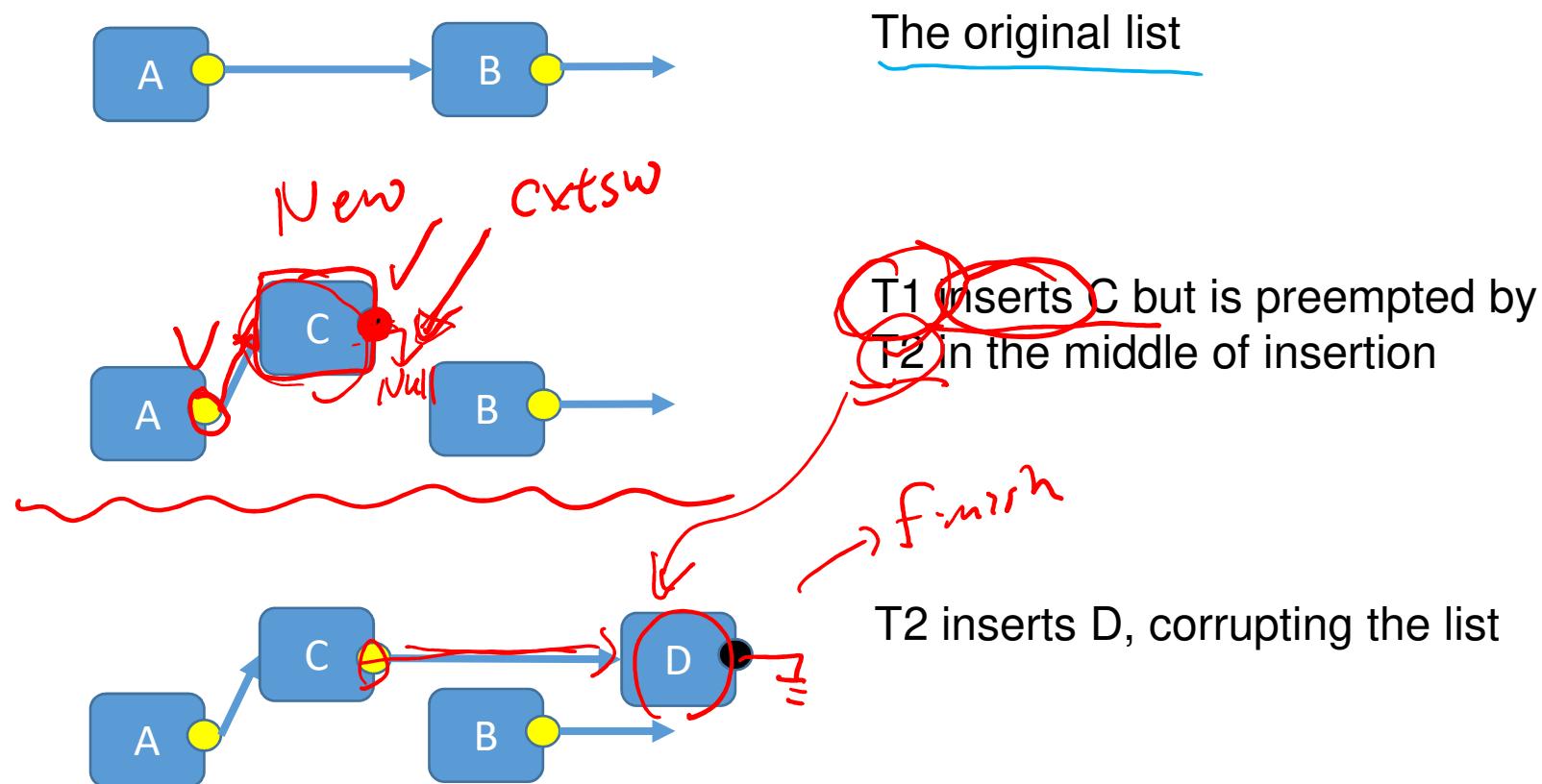
## Race Condition #2

- 2 threads, sharing a variable “safe” which is true initially



# Race Condition #3

- Threads T1 and T2 share a link list



# THE CRITICAL-SECTION PROBLEM

```
do {
```

*entry section* ←

( critical section

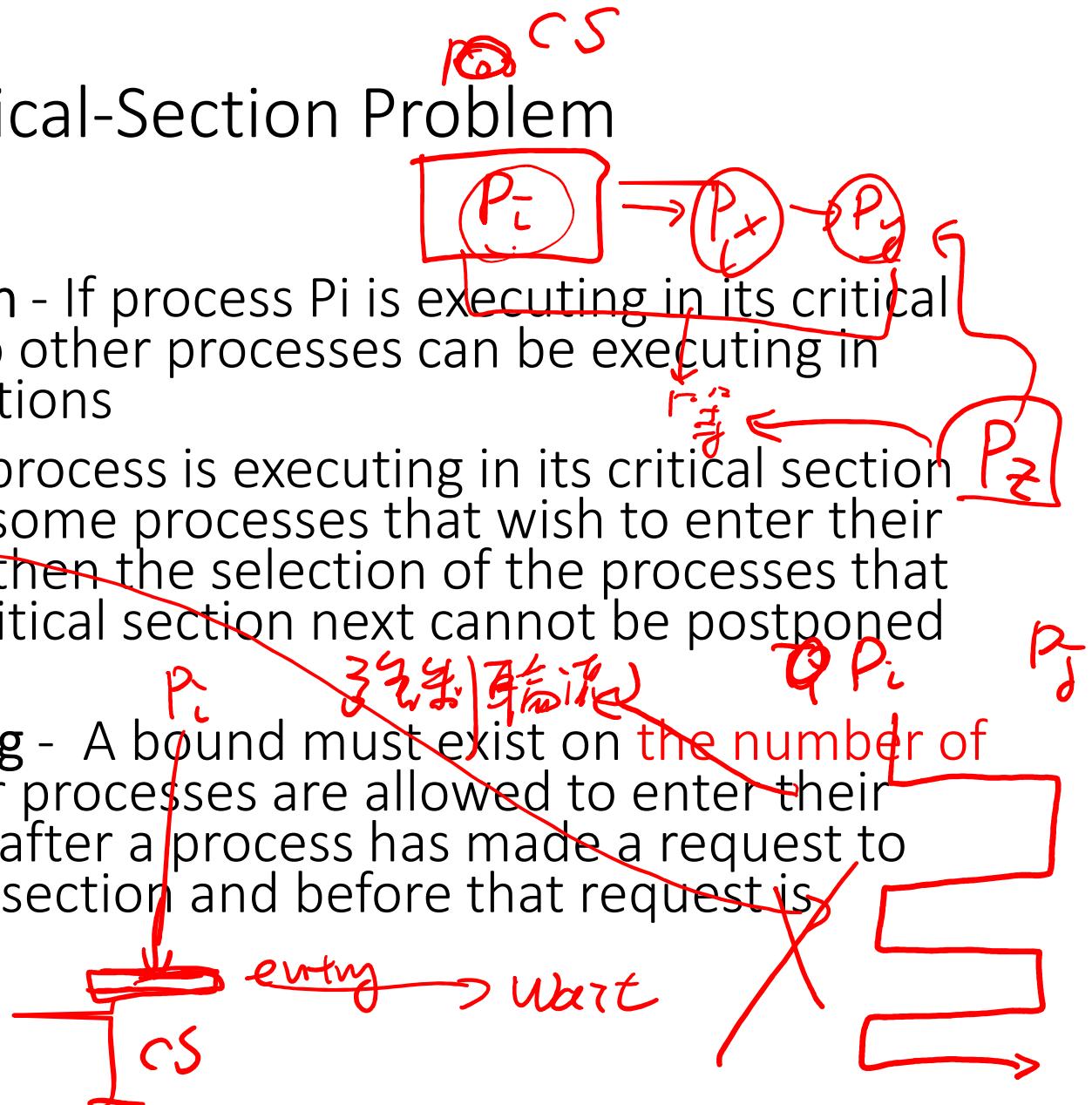
exit section →

remainder section

```
} while (TRUE);
```

## Solution to Critical-Section Problem

- ✓ Mutual Exclusion - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
- ✓ Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- ✓ Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted



# Hardware-based approaches to process synchronization

# Synchronization Hardware

- Many systems provide hardware support for critical section code
- Interrupt disabling → 停止接收  
  - Uniprocessor: good
  - Multiprocessor: does not work
- Test and set or swap
  - Uniprocessor: works, but wastes CPU cycles
  - Multiprocessor: works

## Interrupt Disabling

→ Timer

~~embddos => kernel~~

C.S.

(preemption)

→ I/O

- Uniprocessor

+ kernel completion  
privilege

- Source of preemption: timer, I/O completion

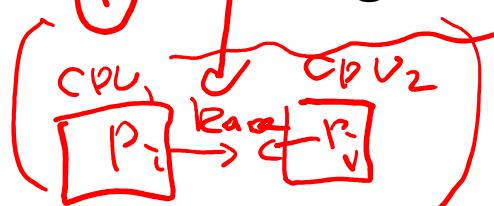
- Masking interrupts prevents the running process from being preempted

- Multiprocessor

- Masking the interrupt of a CPU does not prevent racing processes on the other CPUs from entering a critical section

problems

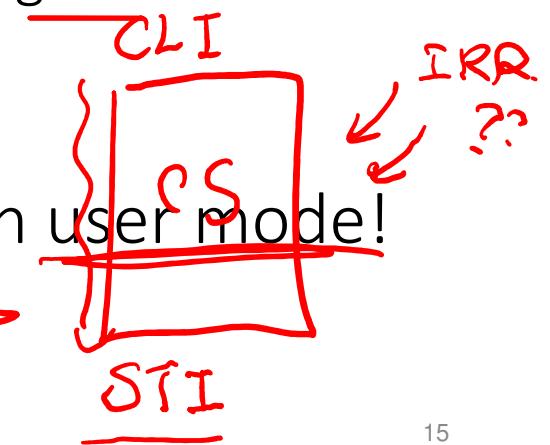
- Privilege instruction, cannot be used in user mode!



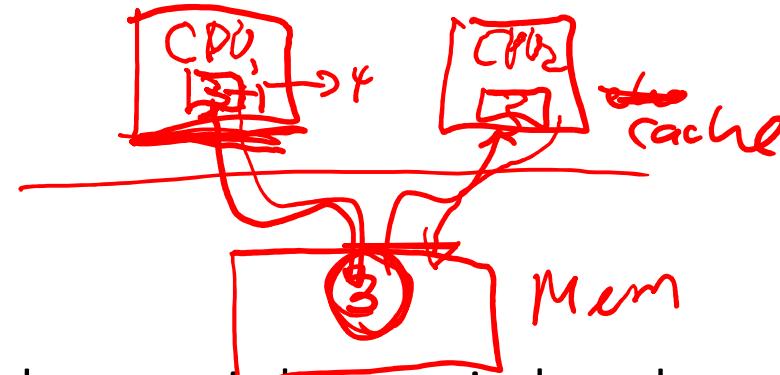
WORKS!

- (a) Nothing happens → x86  
(b) Trap → others

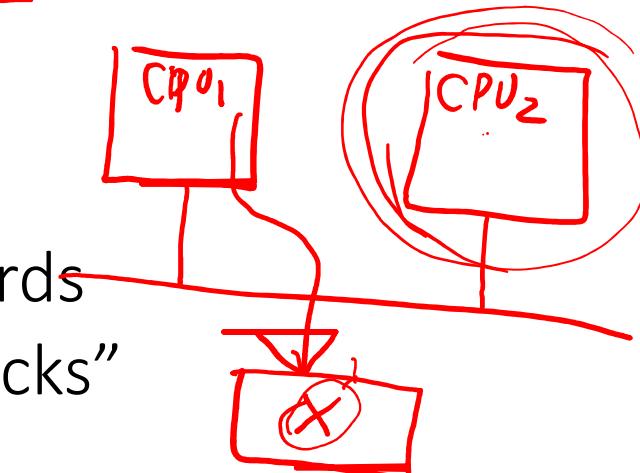
x86. CLI — disable FLAG.  
STI — enable



# Atomic Instructions



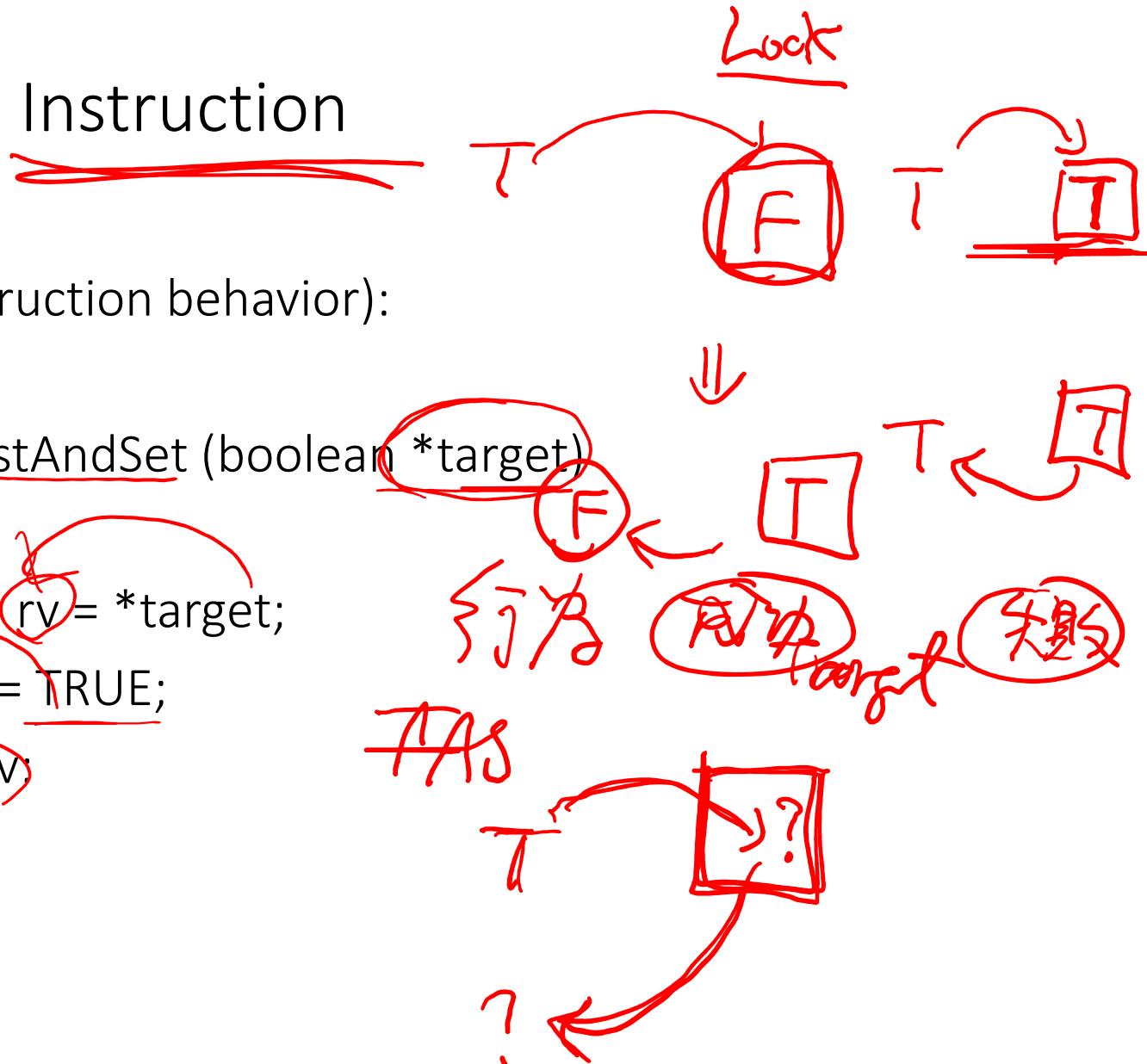
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - No interrupts in the middle of an atomic instruction
  - If in multi-processor environments, the CPU executing an atomic instruction has exclusive access to the target memory (e.g., XCHG in x86)
- Test memory word and set value
- Swap contents of two memory words
- Can be used to implement “spin locks”



## TestAndSet Instruction

- Definition (instruction behavior):

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

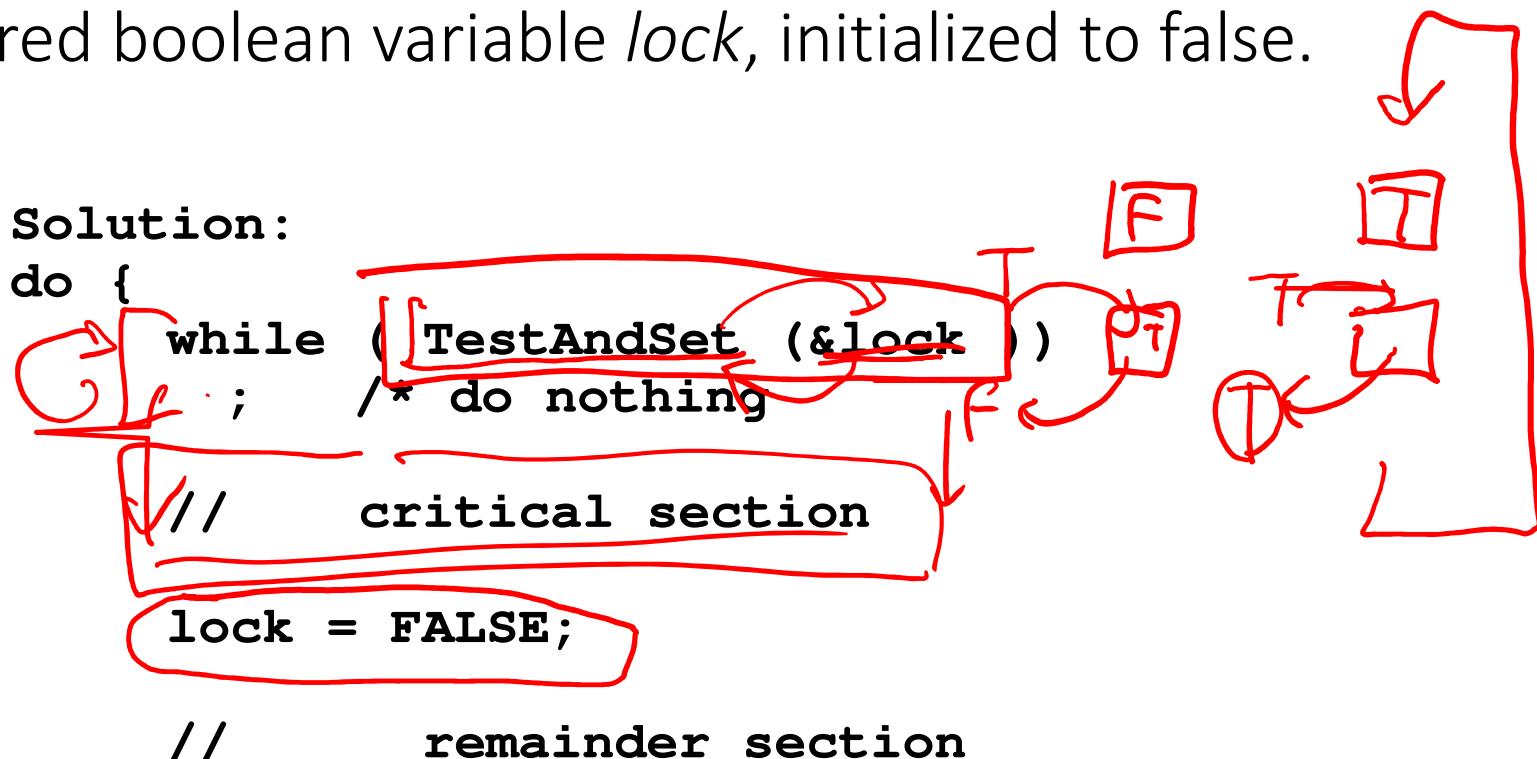


# Solution using TestAndSet

- Shared boolean variable *lock*, initialized to false.

**Solution:**

```
do {  
    while ( TestAndSet (&lock ) ) ; /* do nothing */  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while ( TRUE );
```



# 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*.

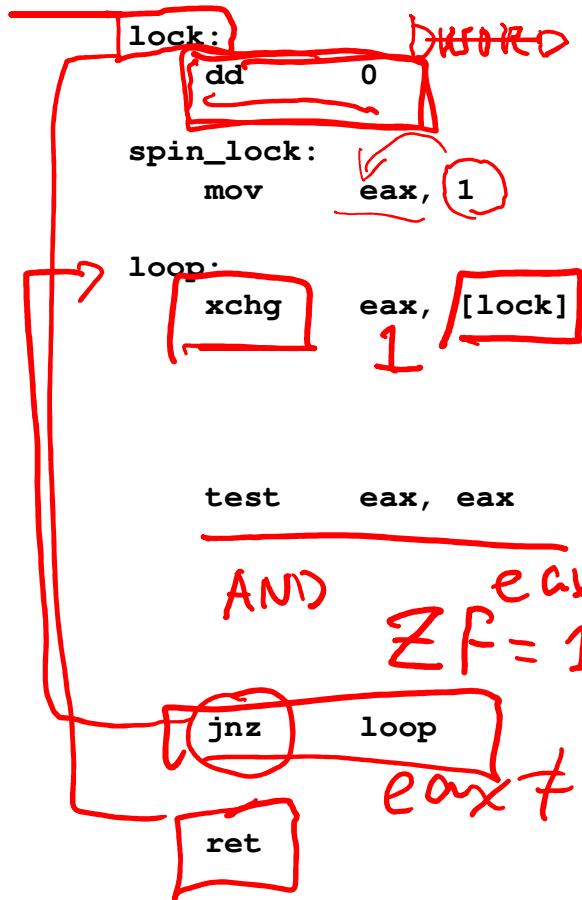
**Solution:**

```
do {
    key = TRUE;
    while ( key == TRUE )
        Swap ( &lock, &key );
    //      critical section
    lock = FALSE;
    //      remainder section
} while ( TRUE );
```

*Quantum expires*      *spin*      *spin lock*      *atomic.c*

The diagram illustrates the execution flow of the code. A vertical bracket on the left labeled "Quantum expires" points to a "spin" node, which then points to a "spin lock" node. The "spin lock" node is enclosed in a red box and labeled "atomic.c". Below the "spin lock" node is the text "// critical section". The entire sequence from "Quantum expires" to "atomic.c" is enclosed in a large red bracket.

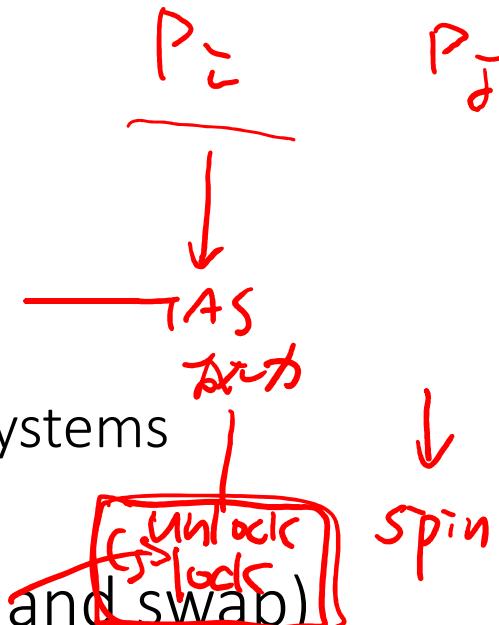
## Spin lock in Linux



```
# The lock variable. 1 = locked, 0 = unlocked.  
  
# Set the EAX register to 1.  
  
Exchange  
# Atomically swap the EAX register with  
# the lock variable.  
# This will always store 1 to the lock, leaving  
# previous value in the EAX register.  
  
# Test EAX with itself. Among other things, this will  
# set the processor's Zero Flag if EAX is 0.  
# If EAX is 0, then the lock was unlocked and  
# we just locked it.  
# Otherwise, EAX is 1 and we didn't acquire the lock.  
  
# Jump back to the XCHG instruction if the Zero Flag is  
# not set, the lock was locked, and we need to spin.  
  
# The lock has been acquired, return to the calling  
# function.  
  
spin_unlock:  
    mov eax, 0  
    xchg eax, [lock]  
    ret  
        # Set the EAX register to 0.  
        # Atomically swap the EAX register with  
        # the lock variable.  
        # The lock has been released.
```

## TAS and SWAP

- Can be used in
  - both uniprocessor and multiprocessor systems
  - both user mode and kernel mode
- Variants exist, such as CAS (compare and swap)
- Problems
  - Wasting CPU cycles in uniprocessor system
  - Because the contention is stateless, process starvation is possible



- The TAS/SWAP instruction as a solution of the critical section problem guarantees which one(s) of the following properties?
  - Mutual exclusive
  - Progressive
  - Bounded waiting

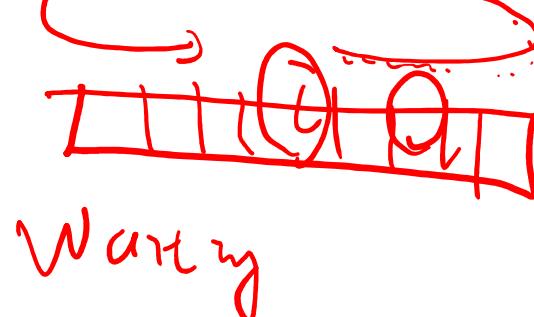
# A bounded-waiting solution based on TAS/SWAP

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // remainder section
}while (TRUE);
```

Waiting[i]=TRUE:  
Pi wishes to enter the critical section

Key=TRUE:  
there is at least 1 process waiting

To pick up the next process in  
waiting[] if there are any waiting  
processes



# Summary

- Uniprocessor
  - Interrupt disabling
    - Only available in kernel space
    - Increasing interrupt latency
  - Spin lock (test and set, swap)
    - Correct, but wasting CPU cycles
- Multiprocessor
  - Interrupt disabling
    - Does not work if two involved processes run on different processors
  - Spin lock
    - Correct, wasting CPU cycle in a minor degree

# Pure-software approach to process synchronization (Peterson's solution)

# Peterson's Solution

- Two-process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int `turn`;
  - boolean `flag[2]`
- The variable *turn* indicates whose turn it is to enter the critical section.
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process  $P_i$  is ready to get in!

# Algorithm for Process Pi & Pj

Pi

do {

```
    flag[i] = TRUE;  
    turn = j;  
    while ( flag[j] && turn == j);
```

CRITICAL SECTION

```
    flag[i] = FALSE;
```

REMAINDER SECTION

```
} while (TRUE);
```

Pj

do {

```
    flag[j] = TRUE;  
    turn = i;  
    while ( flag[i] && turn == i);
```

CRITICAL SECTION

```
    flag[j] = FALSE;
```

REMAINDER SECTION

```
} while (TRUE);
```

# Algorithm for Process Pi

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while ( flag[j] && turn == j);  
        CRITICAL SECTION  
        flag[i] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

- Proof of
- Mutual exclusion
    - $\text{flag}[i], \text{flag}[j]$  are both true
    - Turn is either i or j
    - Will “turn==i” become invalid after Pi enters CS?
      - Impossible because only Pi itself do the change (i.e.,  $\text{turn} \leftarrow j$ )
  - Progressive
    - (!!) Pi will enter the critical section when Pj has no interest in entering the critical section (i.e.,  $\text{flag}[j]=\text{FALSE}$ )
  - Bounded-waiting
    - Consider an example
    - $P_i \leftarrow \rightarrow P_j$  Pi wins
    - Pi completes and Pi arrives again
    - $\rightarrow$ pi always gives the chance away first

# SEMAPHORES

-- a general approach



# Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore  $S$  – integer variable
  - Initial value of  $S$  cannot be negative
  - Can only be accessed via these two indivisible (atomic) operations
- Two standard operations modify  $S$ : `wait()` and `signal()`
  - Originally called `P()` and `V()`
  - Less complicated

*pend()*      *post()*

# Semaphore Implementation with no Busy waiting

- A semaphore is associated with a **waiting queue**
  - If a process is blocked on a semaphore, it is added to the waiting queue of the semaphore
- Two operations:
  - Block – place the process invoking the operation on the appropriate waiting queue: running → waiting
  - Wakeup – remove one of processes in the waiting queue and place it in the ready queue: waiting → ready

# Semaphore Implementation

- Implementation of wait:

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

out sw

Running  
↓  
Waiting

- Implementation of signal:

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

Running

Ready

# Semaphore Implementation

- Semaphores themselves are critical sections
  - Techniques such as interrupt disabling or test-and-set, are used to implement signal() and wait()
  - plus a waiting queue

# Semaphore as a General Synchronization Tool

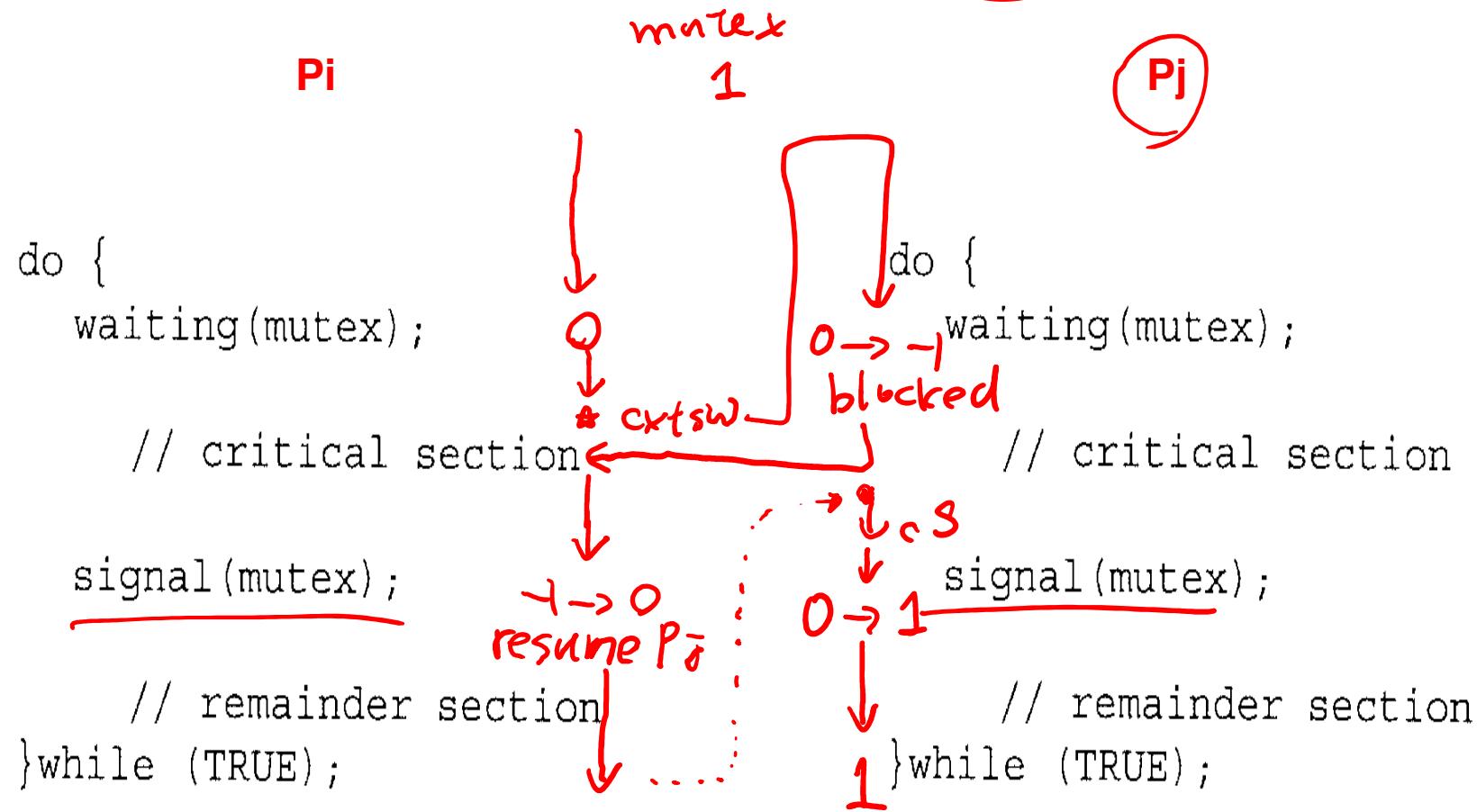
- ① • Counting semaphore – integer value can range over an unrestricted domain
  - Negative runtime values are legit
  - Negative initial values are not allowed in POSIX, however
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
- Can implement a counting semaphore S using a binary semaphore; and vice versa

# Typical Usages of a Counting Semaphore

- The purpose of a (counting) semaphore can typically be determined by the initial value of the semaphore

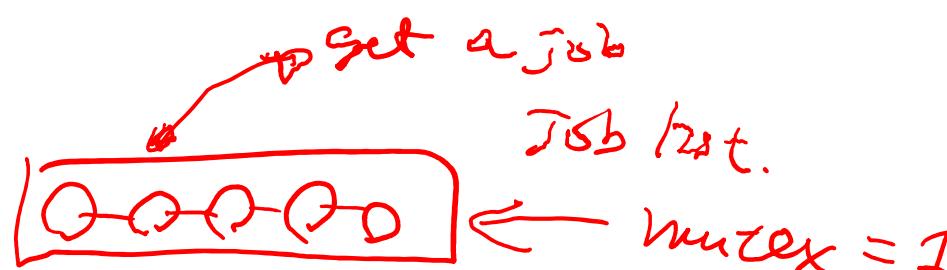
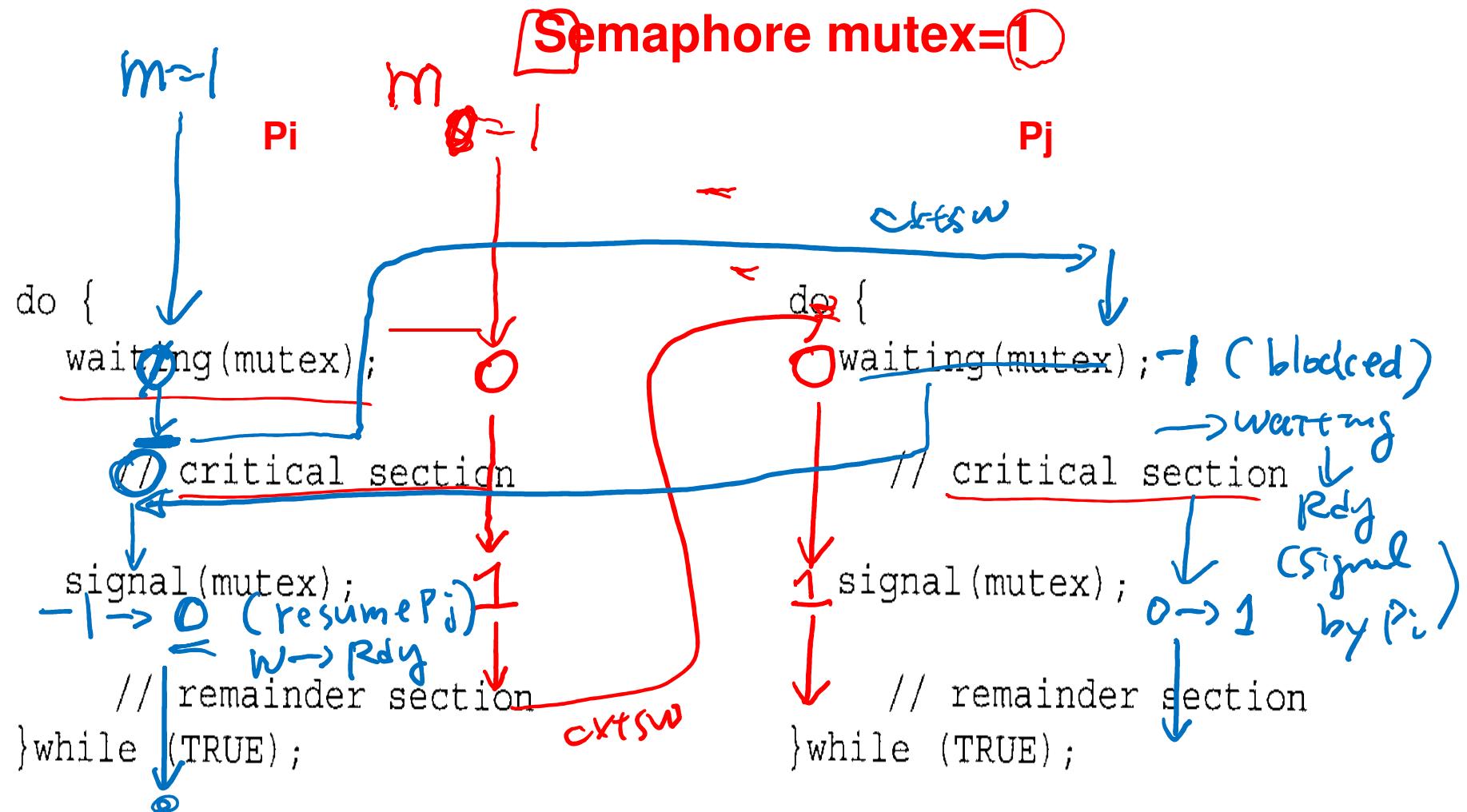
- *Mutex lock*: init value = 1
- *Sequencing or event*: init value = 0
- *Capacity control*: initial value=capacity

## Mutual exclusion Semaphore mutex=1 *init*

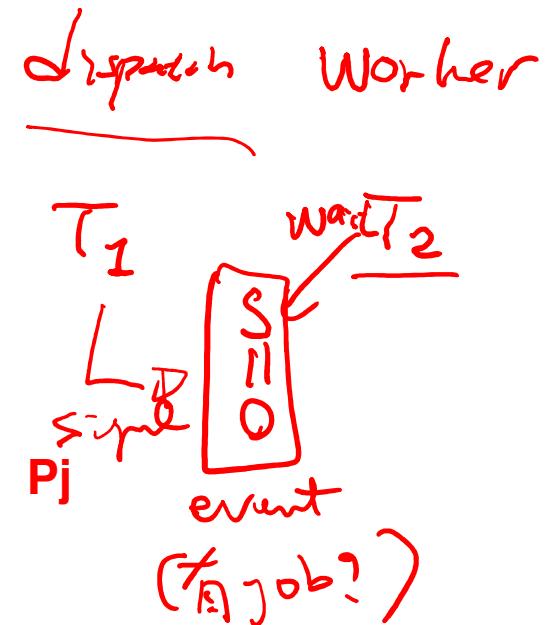
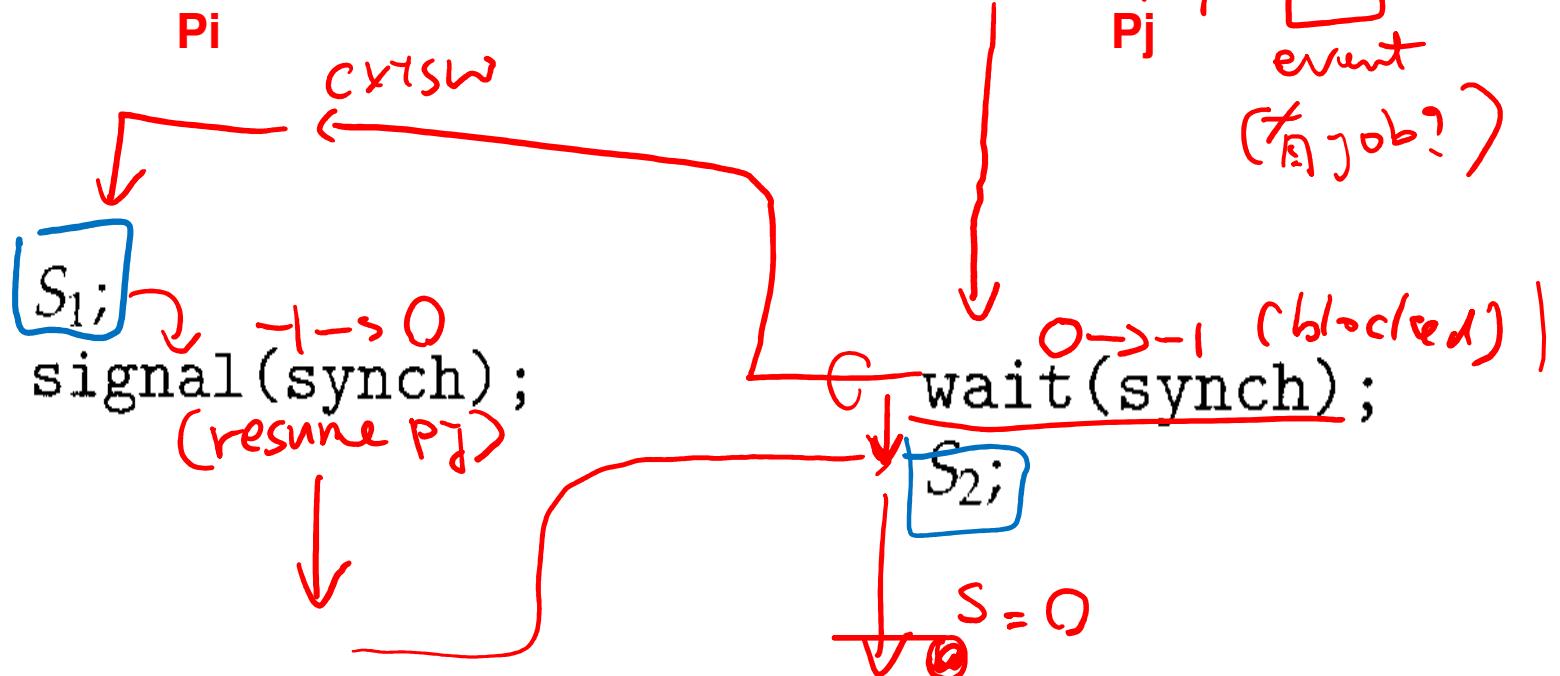


$$\text{init} = 1$$

## Mutual exclusion



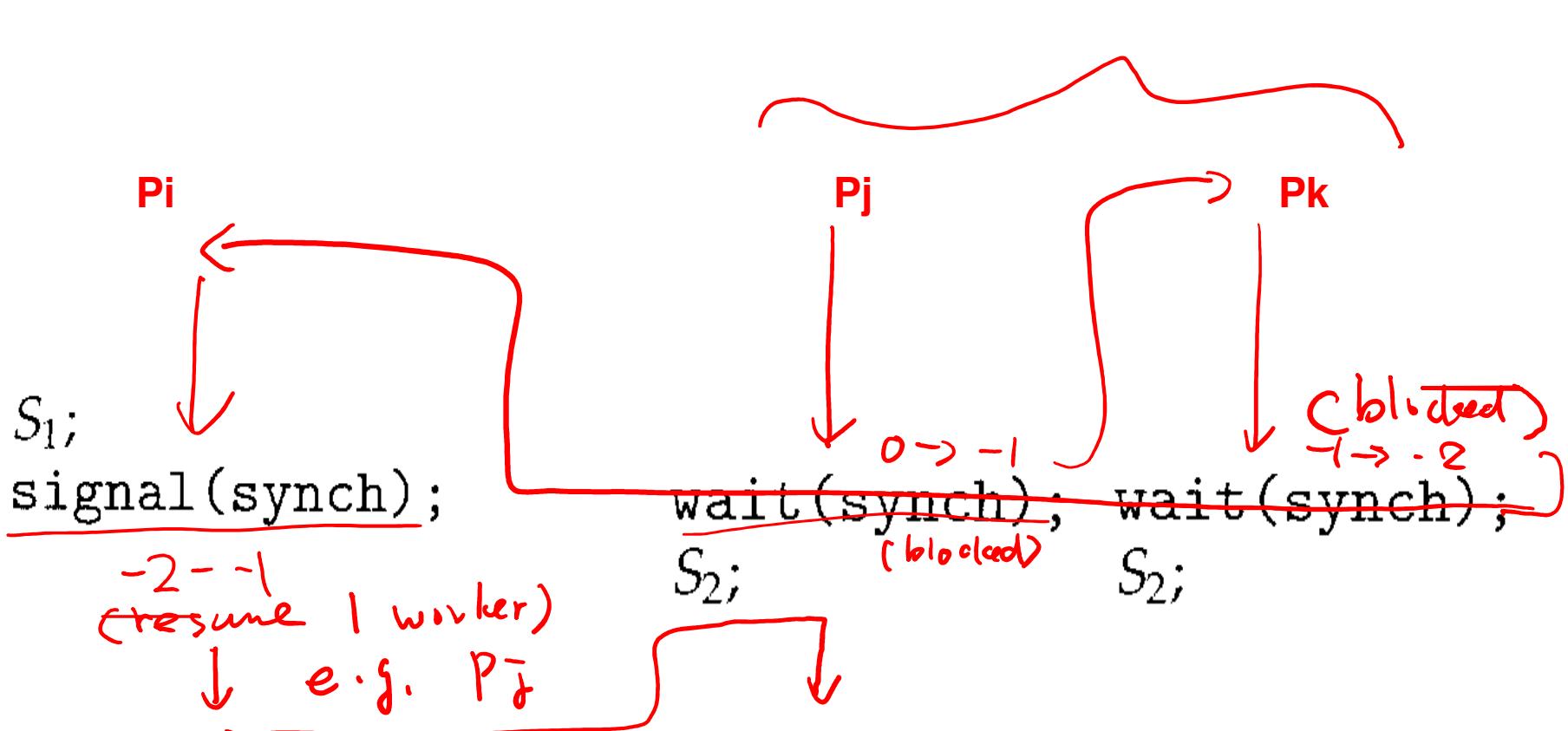
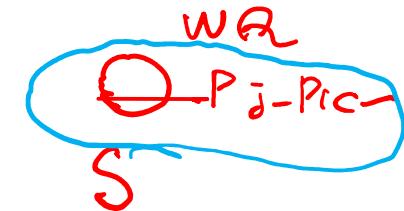
Sequencing or event  
Semaphore synch=0



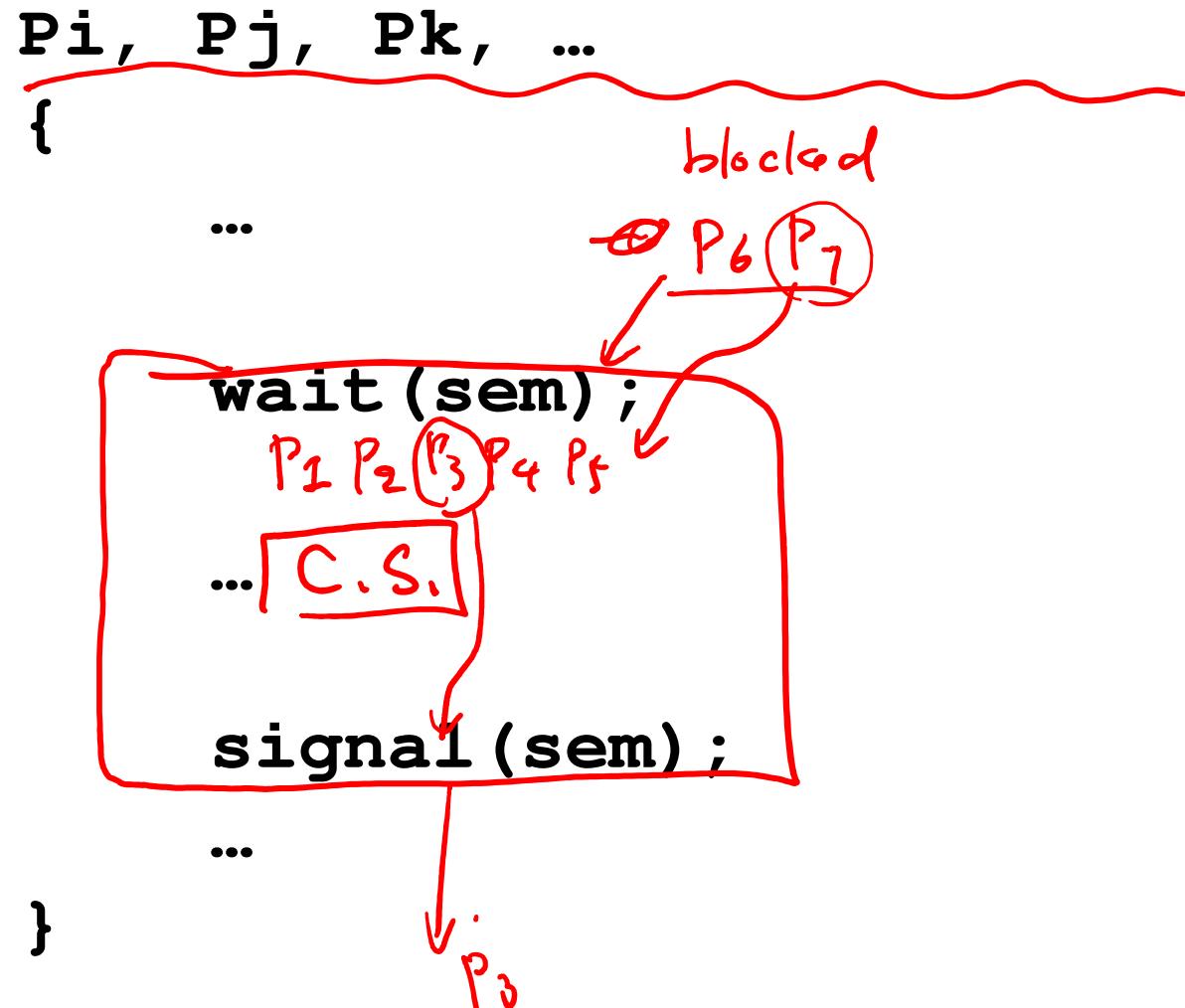
事件是什么？ event

$S_1 < S_2$

Sequencing or event  
Semaphore synch=0

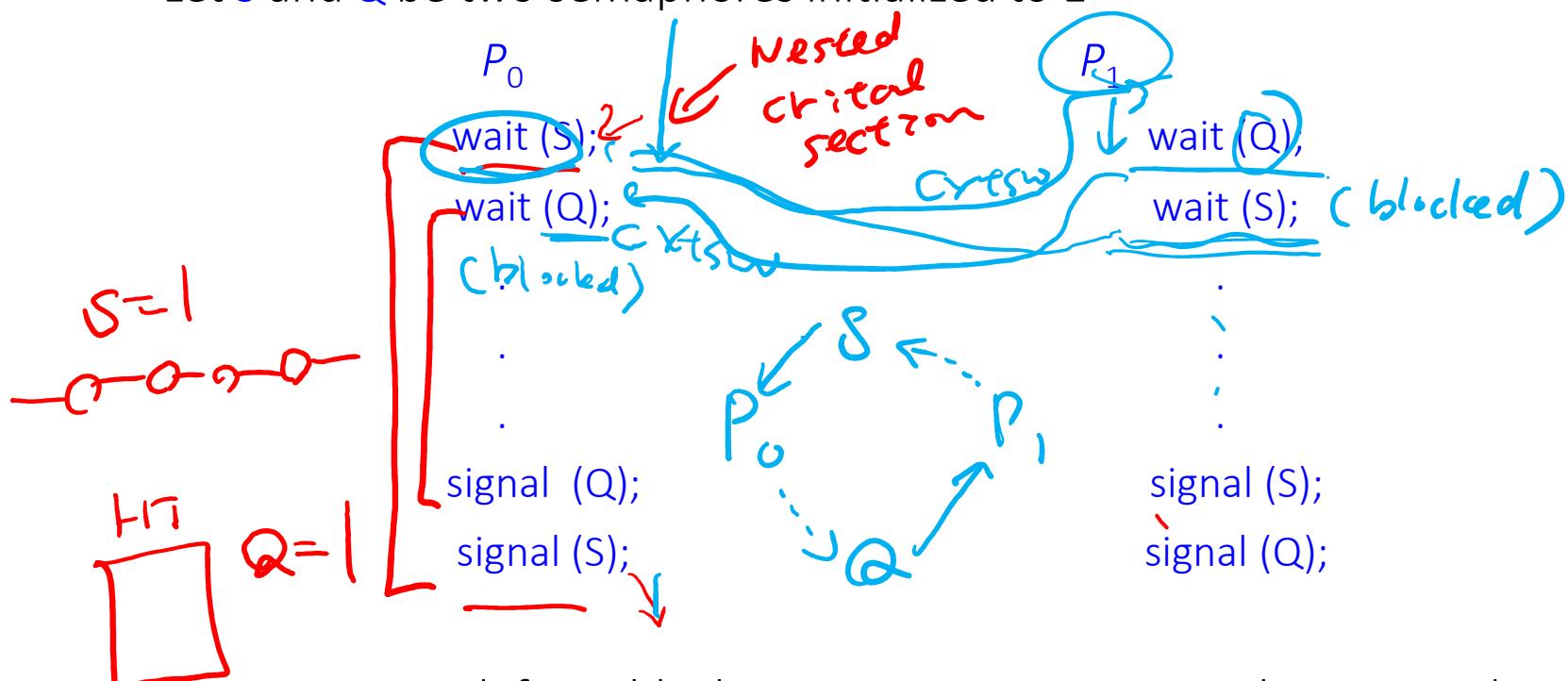


Capacity control  
Semaphore sem=capacity



# 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



- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
  - Waking up blocked processes in a first-come first-served manner is a solution

# Classical Problems of Synchronization

## Example #1

- [有酒食，先生饌] 小明和老師中午一起吃飯，請使用 semaphore 幫助小明，讓他禮讓老師先用餐。

```
S=0;  
ming()  
{  
    wait(S);  
    // eat  
}  
  
professor()  
{  
    / eat  
    signal(S);  
}
```

## Example #2

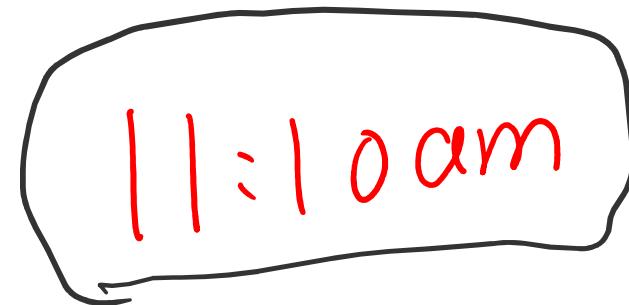
- [蜘蛛人] 傑克和羅絲相約去看電影，兩人約在電影院門口見面，如果有人先到的話，要等另一人到了，才可以進入電影院。

```
R=0;J=0
jack()
{
        signal(J);
        wait(R);
    ...
    // see the movie
}
ross()
{
        signal(R);
        wait(J);
    ...
    // see the movie
}
```

## Example #3

- [睡成一片] 資工系期中考快到了，總共有1000位同學想進入自習室。因為座位有限，所以只能50個人同時進入。

```
S=50  
  
student()  
{  
    wait(S);  
    ↓  
    // go studying  
  
    signal(S);  
}
```



## Example #4

- A DMA controller supports four channels of data xfer

```
S=4;T=1;c[4]={F,F,F,F};  
proc()  
{  
    wait(S);  
    wait(T);  
    // pick one unused channel among c[0],c[1],c[2],c[3]  
    // setup DMA transfer  
    signal(T);  
    // start DMA  
    // wait for DMA completion  
    signal(S);  
}
```

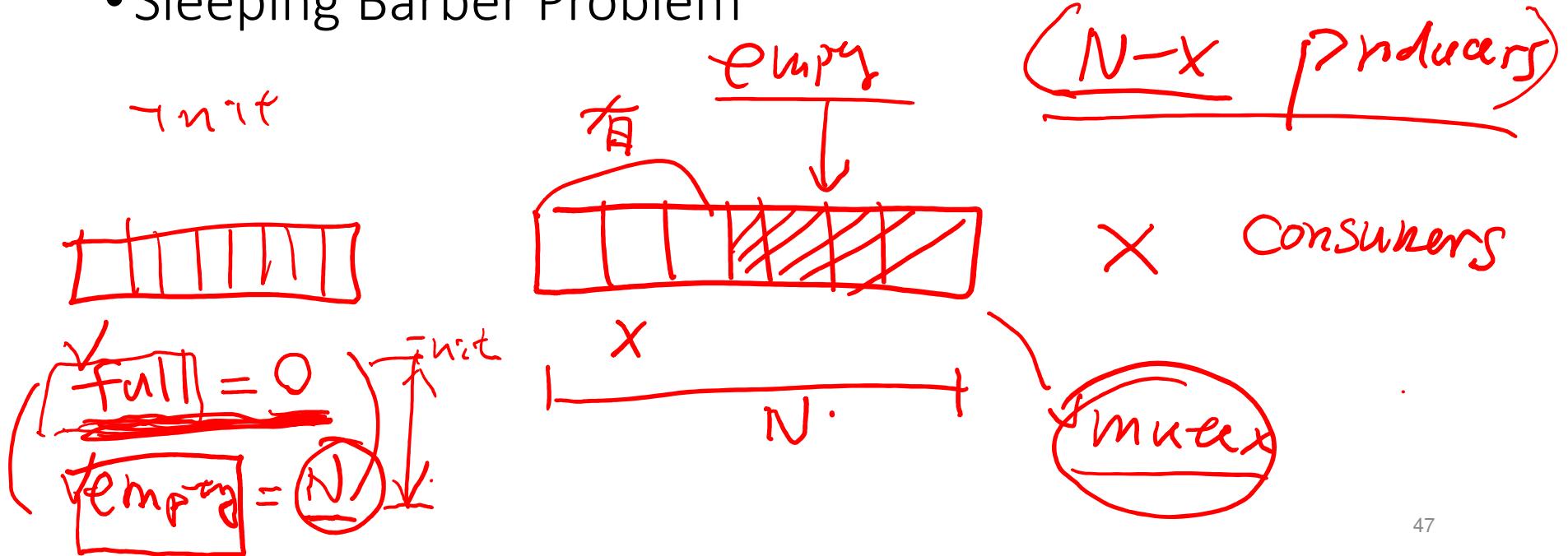
UP to 4.

*modify C[4]*

# Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- Sleeping Barber Problem

count      race  
mutex



# Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
  - To protect the buffer
  - There can be **many** producers and consumers!!
- Semaphore **full** initialized to the value 0
  - 0 items (for the consumer)
  - Block on no items
- Semaphore **empty** initialized to the value N.
  - N free slots (for the producer)
  - Block on no free slot

# Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    N+1 // produce an item  
    wait (empty);  
    wait (mutex);  
    // add the item to the buffer  
    signal (mutex);  
    signal (full);  
} while (true);
```

Producers produce items  
Consumers “produce” free slots

What are the initial values of  
empty and full?

What happens if mutex is placed  
in the outer scope?

# Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
  
} while (true);
```

Annotations:

- A red box highlights the line `wait (full);`.
- A red circle highlights the line `signal (empty);`.
- A red arrow points from the label `~init=0` to the opening brace of the `do {` loop.

# Readers-Writers Problem

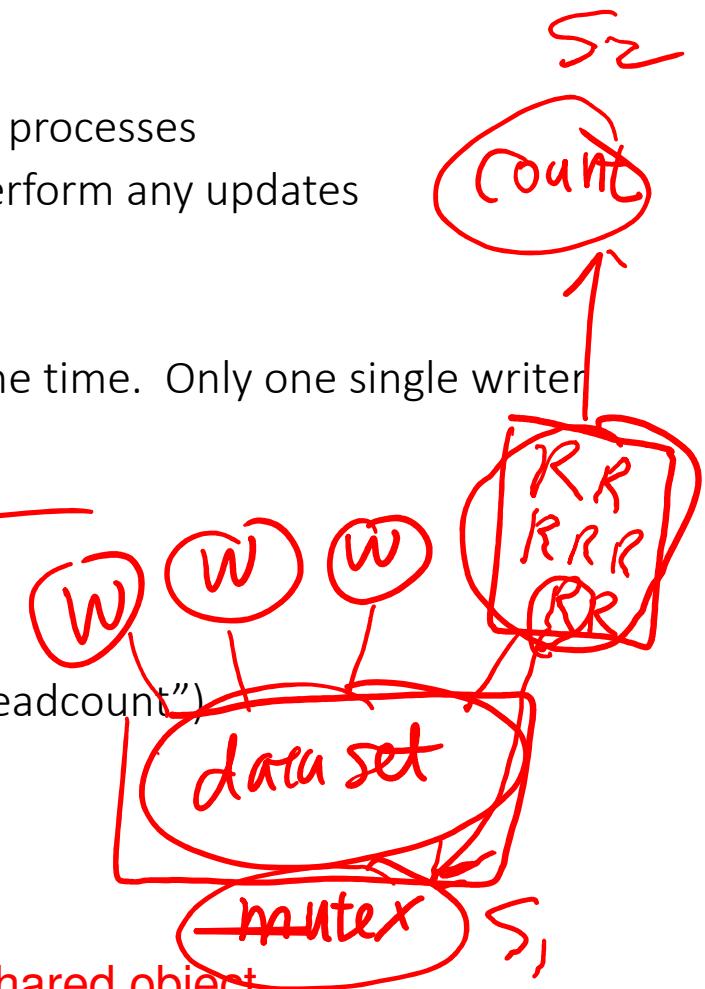
## RW Lock

- A data set is shared among a number of concurrent processes
  - 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.

- Shared Data
  - Data set
  - Semaphore **mutex** initialized to 1. (to protect “readcount”)
  - Semaphore **wrt** initialized to 1.
  - Integer **readcount** initialized to 0.

Version 1



The first R-W problem

- No readers will wait until the writer locked the shared object
- Readers need not to synch with each other

# Readers-Writers Problem (Cont.)

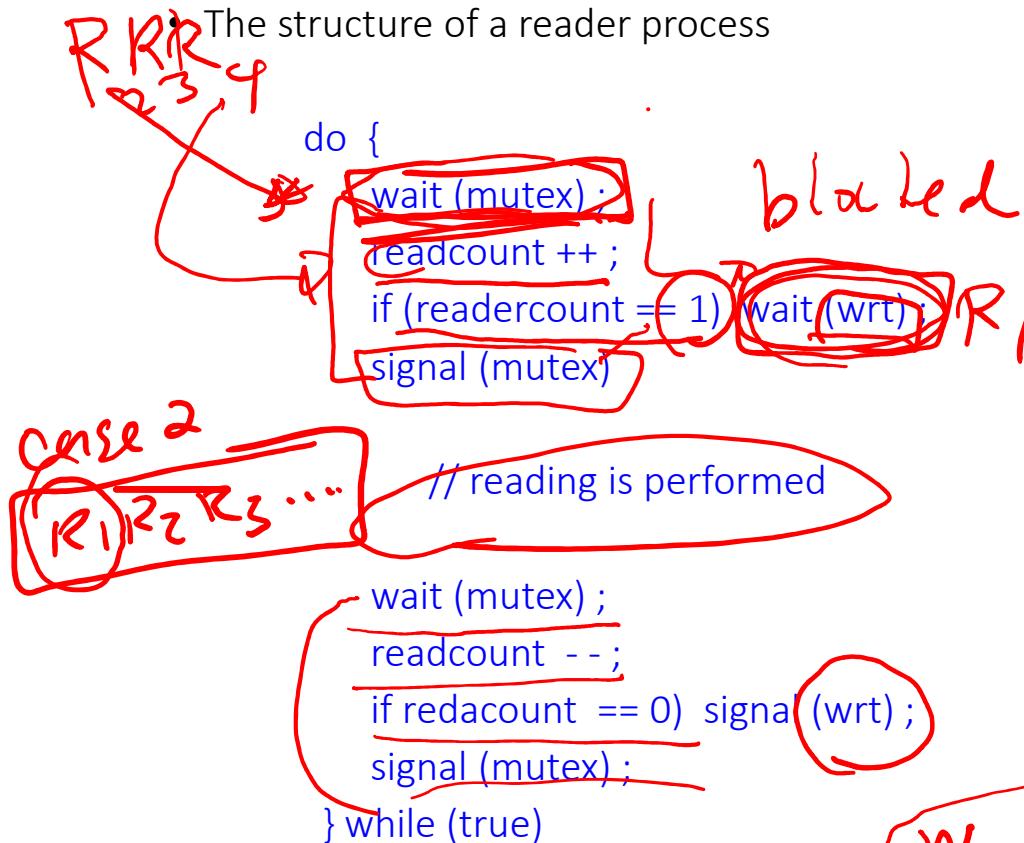
- The structure of a writer process

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

Initially, wrt=1 mutex=1

# Readers-Writers Problem (Cont.)

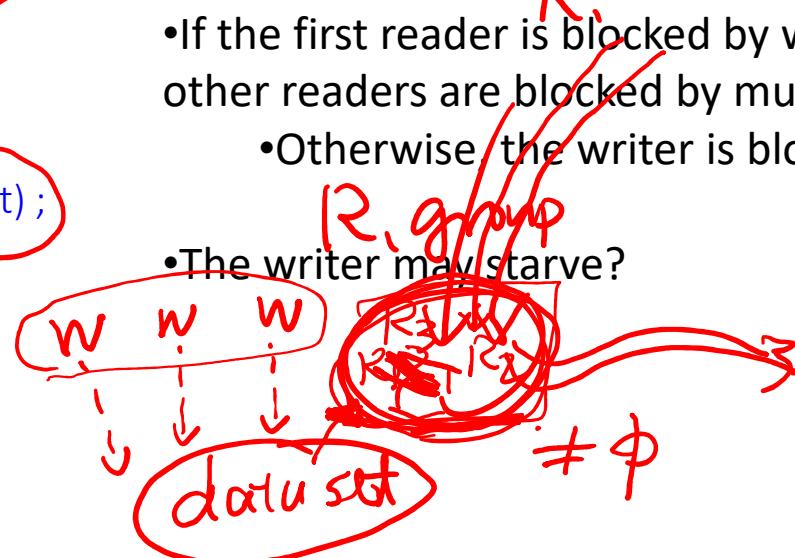
- The structure of a reader process



1. No readers will wait until the writer locked the shared object
2. Readers need not to synch with each other

- Simply using one mutex for R/W violates the second condition
- If the first reader is blocked by wrt, then other readers are blocked by mutex
  - Otherwise, the writer is blocked

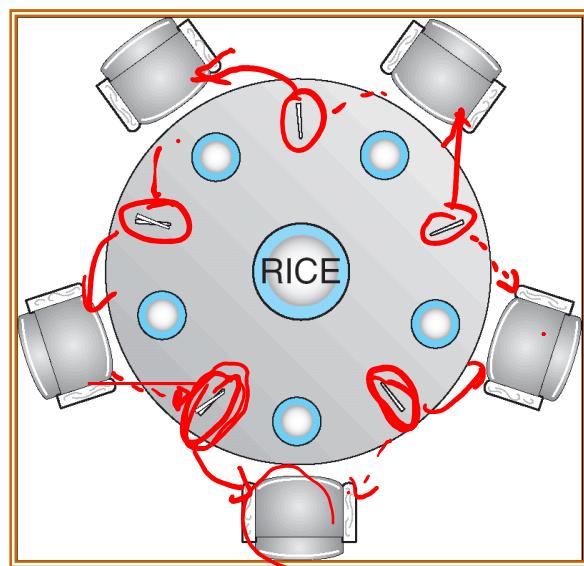
• The writer may starve?



# Readers-Writers Problem (Cont.)

- Mutex
  - Protect the data set and the “readcount”
- Wrt
  - Mutex, ensure mutual exclusion among
    - A writer
    - A writer
    - ...
    - A group of readers

# Dining-Philosophers Problem



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

sem x5

# Dining-Philosophers Problem (Cont.)

- The structure of Philosopher  $i$ :

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

Potential deadlocks!

## Dining-Philosophers Problem (Cont.)

- Possible ways to prevent deadlocks
  - One person get the left stick first, the rest get the right stick first
  - Allow up to 2 (or less than 5) people having stick(s)
  - Picking up two sticks simultaneously

# Sleeping Barber Problem



# Sleeping Barber Problem

- A barbershop consists of awaiting room with  $n$  chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep.
- If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.
- If chairs are available but the barber is busy, then the customer sits in one of the free chairs.
  - If the barber is asleep, the customer wakes up the barber.

# Sleeping Barber Problem

- Semaphore Customers = 0;
  - Event: there are waiting customers
  - The barber waits on it if there is no customer
- Semaphore Barber = 0;
  - Event: barber is ready
  - The customer waits on it if the barber is busy
- Semaphore accessSeats = 1;
  - int NumberOfFreeSeats = N; //total number of seats

## Costumer's process

```
while(1) {  
    wait(accessSeats) //mutex protect the number of available seats  
  
    if ( NumberOfFreeSeats > 0 )  
    {  
        //if any free seats  
        NumberOfFreeSeats--; //sitting down on a chair  
        signal(Customers) ; //notify the Barber  
        signal (accessSeats); //release the lock  
        wait(Barber); //wait if the B is busy  
        .... //here the C is having his hair cut  
    }  
    else  
    {  
        //there are no free seats  
        signal (accessSeats); //release the lock on the seats  
        ... //C leaves without a haircut  
    }  
}/while(1)
```

Barber process

```
while(1) {
    wait(Customers); //wait for C and sleep
    wait (accessSeats); //mutex protect the number of
                         // available seats

    NumberOfFreeSeats++; //one chair gets free

    signal(Barber);      //Bring in a C for haircut
    signal (accessSeats); //release the mutex on the chairs

    .....
               //here the B is cutting hair

}//while(1)
```

# Mutexes and Monitors

# Mutex locks

- “MUTually EXclusive” access
- Conceptually equivalent to semaphores with initial value = 1
- Only the locker of a mutex can unlock the mutex
- APIs
  - `pthread_mutex_lock()`
  - `pthread_mutex_unlock()`
  - ...

# Semaphores vs. mutexes

- `pthread_mutex_xxxx()`
  - `pthread.h`
  - Functionally equivalent to semaphore with init value=1
  - Applicable to threads only
- `sem_xxx()`
  - `semaphore.h`
  - Applicable to threads and processes
  - A semaphore can be signaled by any process/thread
  - `sem_wait()`, `sem_post()`, ... \

# Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (s) .... signal(s)
  - wait (s) ... wait (s)
  - Omitting of wait (s) or signal (s)
- Monitor provides a higher level abstraction of critical section to avoid these programming errors

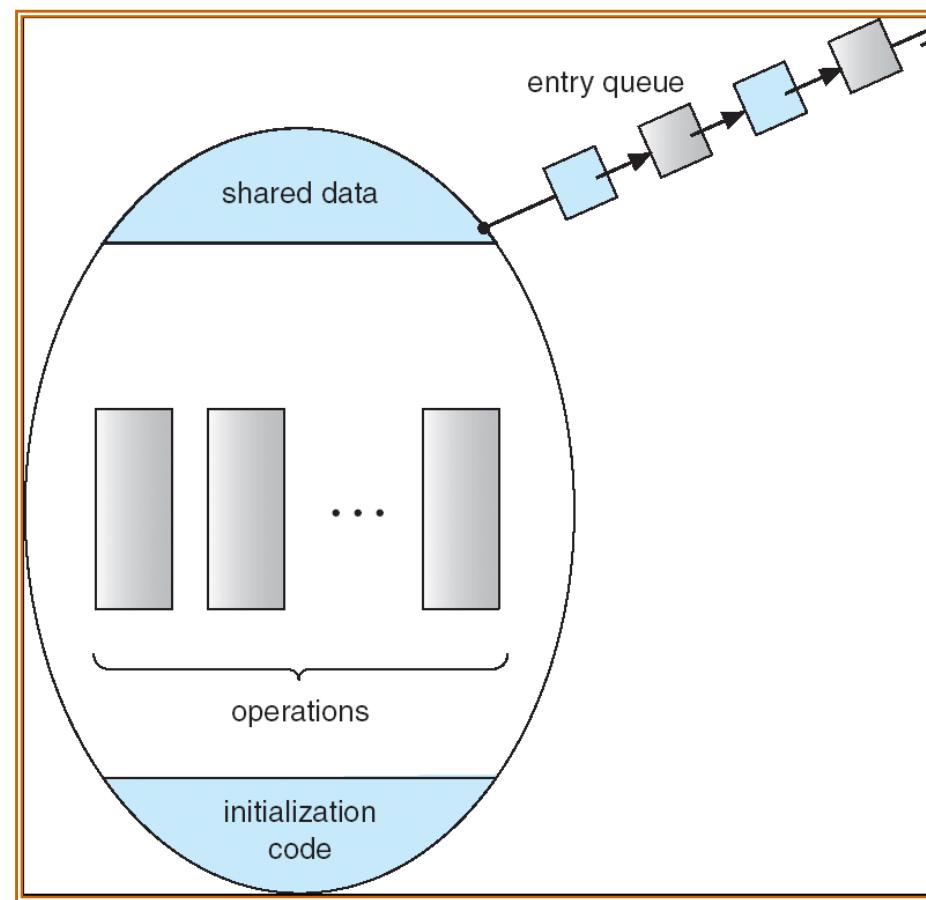
# Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

}
```

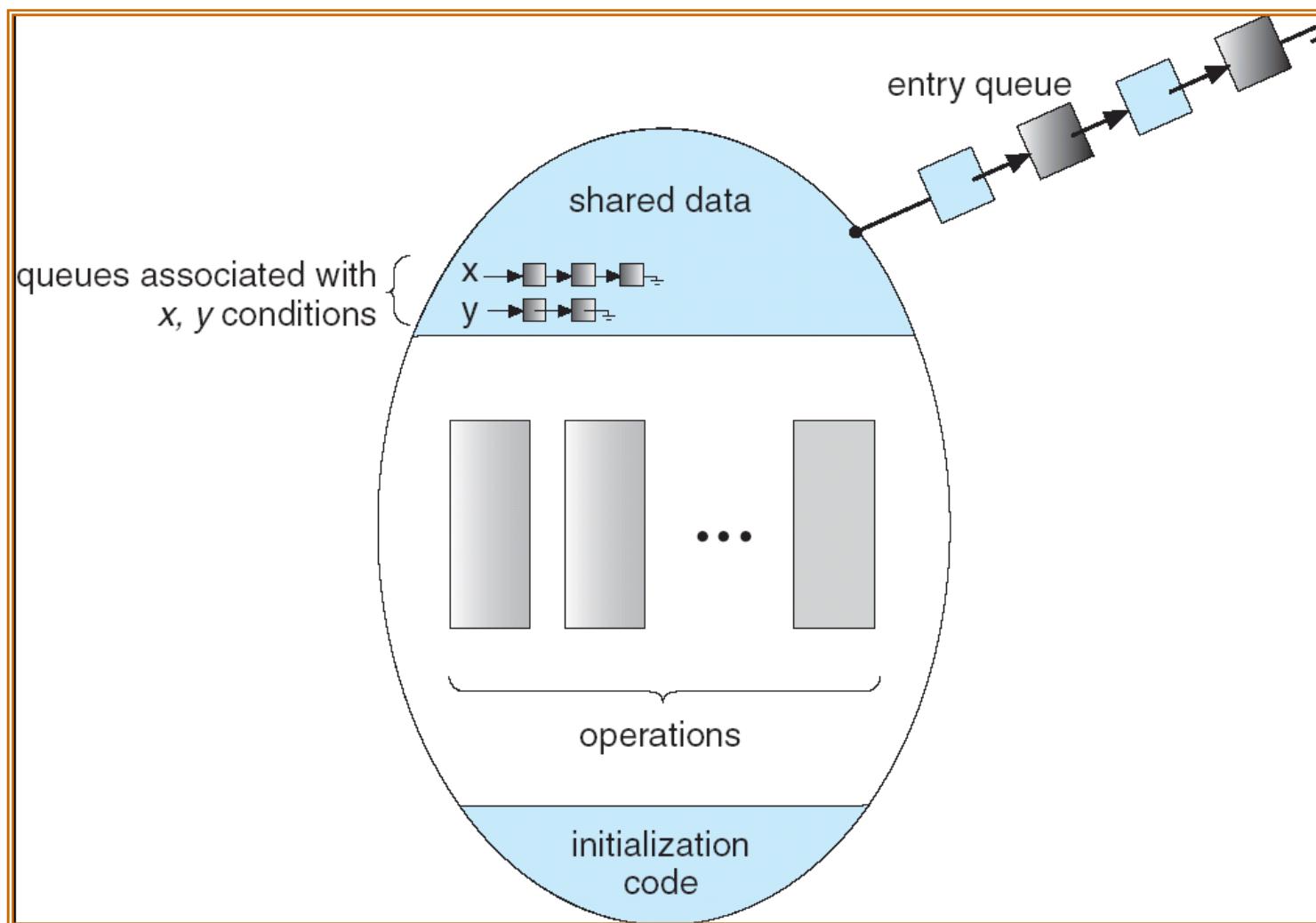
# Schematic view of a Monitor



# Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
- Monitor itself provides nothing but mutex
  - To implement other synch policy, conditional variables are needed
- signal → if there is no process waiting, **nothing happens** and the next process calls wait is blocked
  - Different from semaphore. For capacity control, a monitor must contain a counter

# Monitor with Condition Variables



# Solution to Dining Philosophers

```
monitor DP
{
    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);
    }
}
```

## Highlights to this solution:

- A philosopher picks up 2 chopsticks at a time
  - If he can not pick up 2 chopsticks, he waits
- After a philosopher done eating, he will check if his 2 neighbors can eat.

# Solution to Dining Philosophers (cont)

```
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;  
}  
}
```

- How to implement semaphore using a monitor?
  - Signals to condition variables do not affect processes that are not in a monitor
  - Use a counter to keep track of “signals to semaphores”
- How to implement monitor using semaphores?
  - See the text book

# End of Chapter 6