Amirkabir University of Technology
(Tehran Polytechnic)
Department of Computer Engineering

Process Synchronization
(همگام سازی فرآیندها)

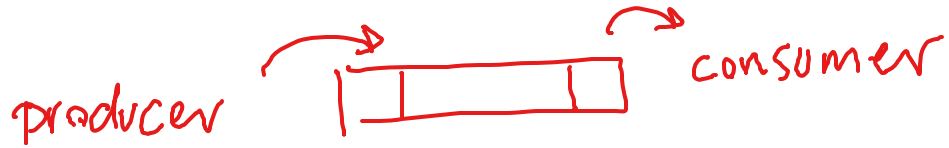
Hamid R. Zarandi
h_zarandi@aut.ac.ir

Motivation

- **Cooperating process/thread:** { message passing / shared memory
  - the one that can affect or be affected by other processes executing in system.
  - Processes, threads
- **Processes can execute concurrently**
  - May be interrupted at any time, partially completing execution
- **Problem: Data inconsistency** (ناسازگاری داده)
  - It may occur in the case of concurrent access to shared data
- **How to solve?**
  - Orderly execution of cooperating processes that share a logical address space

One example!

- A solution to consumer-producer problem that fills *all* the buffers.
  - We can have an integer `counter` that keeps track of the number of full buffers.
  - Initially, `counter` is set to 0.
  - It is incremented by the producer after it produces a new buffer ++
  - It is decremented by the consumer after it consumes a buffer. --



Circular buffer & producer-consumer problem

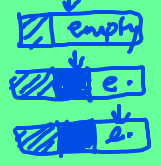
```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

counter ← تعداد داده‌های بافر

Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE))
                ; /* do nothing */

        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter ++;
}
```

اگه بافر پره نمیشه چیزی تولید کرد



یه داده به بافر اضافه شد

Consumer

```
item next_consumed;
while (true) {
        while (counter == 0)
                ;/* do nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter --;
        /* consume the item in next consumed */
}
```

اگه بافر خالیه چیزی نمیتونی برداری!

Race condition

- **`counter++`** could be implemented as

```
register1 = counter          MOV   AX, [100]
register1 = register1 + 1    ADD   AX, 1
counter = register1          MOV   [100], AX
```

counter

} = counter++

↳ LOAD → INCREMENT → STORE

- **`counter--`** could be implemented as

```
register2 = counter          MOV   BX, [100]
register2 = register2 - 1    SUB   BX, 1
counter = register2          MOV   [100], BX
```

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

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

swapping

قاعدتا باید ۵ میشد!

(5++)-- = 5

★ هر وقت ساعت شد حتما غلط میکنیم!

Another Race condition

- **Invoking *echo()* procedure:**

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

```
Process P1                      Process P2
•                               •
chin = getchar(); //a           •
•                               chin = getchar(); //b
chout = chin;                   chout = chin;
putchar(chout);                 •
•                               putchar(chout);
•                               •
```

خروجی هردو ← b

- **Same problem exists on:**
  - Multiprogramming environment
  - Multiprocessing environment
  - Distributed processing environment

* در پردازنده های تک هسته ای هم چنین چیزی وجود داد

دلیل: زیاد شدن core# ← race condition

* اگه به ترتیب اجرا میشد درست بود (ولی) این تیکه نباید هم روند اجرا کرد ← سریال سازی موقت

Other examples?

Have you ever seen other examples?

blah blah blah ...

Definition

- **Race condition**
  - Several process access and manipulate the same data concurrently
  - Outcomes of the execution depends on the order in which the access take place

- **How to remove Race Condition?**
  - Serial execution

مسئله ناحیه بحرانی

Critical Section Problem

Critical section problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section

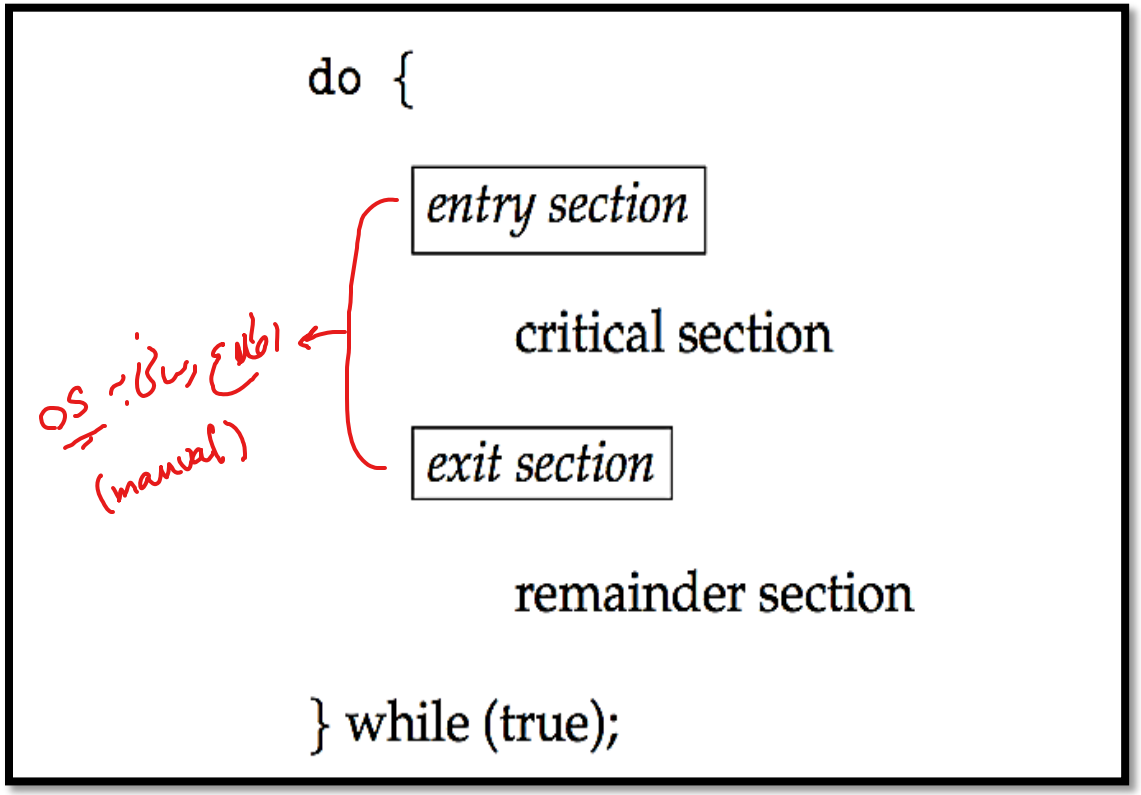
→ فقط به یه نفر اجازه بدیم

- *Critical section problem* is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

باید ابتدا و انتهای ناحیه بحرانی رو خبر بدیم

Critical section

- **General structure of process P_i**

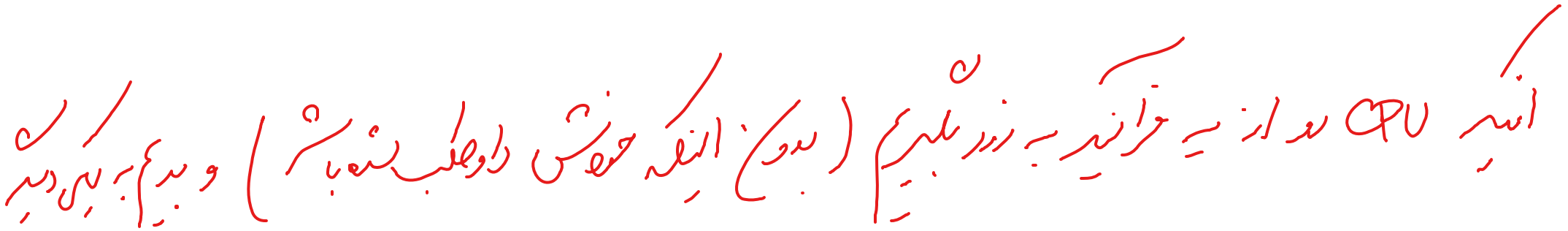


Requirements to solutions

- **Mutual exclusion (انحصار متقابل)**
  - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress (پیشرفت)**
  - 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 (انتظار محدود)**
  - 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
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the *n* processes

Preemption definition

- **Preemption (قبضه ای - قبضه شدنی)**
  - **The act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time [wiki]**



Handling critical-section by OS

- **Two approaches, depend on type of OS kernels**
  - **Preemptive**
    - Allows preemption of process when running in kernel mode
    - Difficult to design in SMP architectures (why?)
  - **Non-preemptive**
    - Runs until exits kernel mode, blocks, or voluntarily yields CPU
      - ✓Essentially free of race conditions in kernel mode (why?)

- **Which one**
  - is responsive?
  - is suitable for real-time programming?

1) Peterson's solution

- A classis SW solution

روشی ای نرم افزاری، وابسته هستند به نوع سخت افزار

- No guarantees in correct working of the method
  - Correctness depends on computer architecture
  - Atomic instructions are needed (which & where?)

← برخلاف -- و ++ که سه‌قسمت بودند.

- Good algorithm! (~~Best~~)

- Shared variables
  - `int turn; /* whose turn is */` نوبت کیه؟
  - `Boolean flag[2] /* who enters the critical-section */` کی آمادس؟

Peterson algorithm for P_i

(Pi, Pj)=(P0, P1)

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

            critical section

    flag[i] = false;

            remainder section

} while (true);
```

How requirements are satisfied?

- Mutual exclusion (?)
- Progress (?)
- Bounded waiting (?)

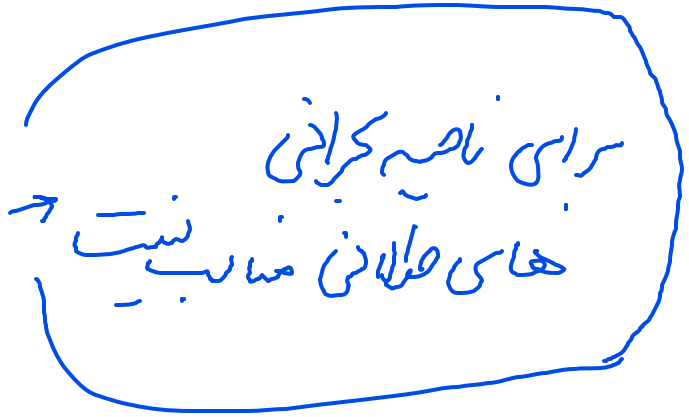
```
do {
    flag[j] = true;
    turn = i
    while (flag[i] & turn == i);

    flag[j] = false
```

2) Hardware solution

- Some hardwares support implementing the critical section code!

- All solutions are based on idea of locking
  - Protecting critical regions via locks

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable



- Multiprocessors – provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Either
    - test memory word and set value
    - swap contents of two memory words

Hardware solution for critical section

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

How requirements are satisfied?

- Mutual exclusion (?)
- Progress (?)
- Bounded waiting (?)

test_and_set instruction

* رفتارش شبیه کد زیره ولی پیاده سازیش دقیق این نیست.

Definition:

```
boolean test_and_set (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv;    /* old value */
 }
```

مقدار قدیم را بده و مقدار جدید را true کن.

از هرزبان ۲ تا TAS میبان TAS رو اجرا کنن، این ۲ تا TAS به ترتیب دلخواه سریالی اجرا میشن.

1. Executed atomically

2. Returns the original value of passed parameter

3. Set the new value of passed parameter to “TRUE”.

Hardware solution using *test_and_set()*

- Shared Boolean variable lock, initialized to FALSE

```
do {
     while (test_and_set(&lock))
          ; /* do nothing */
     /* critical section */
     lock = false;
     /* remainder section */
  } while (true);
```

```
bool testandSet (*target)
bool rv = *target
*target = true
return rv
}
```

lock → global
↳ init . false

compare_and_swap instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;     /* old value */
}
```

اگر مقدار متغیر همان مقدار مورد انتظار بود، به روزش کن.

1. Executed atomically

2. Returns the original value of passed parameter “value”

3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Hardware solution using *compare_and_swap()*

int CAS(*value, expected, new)
int temp = *value
if (*value = expected)
*value = new
return temp

lock → global
↳ init: 0

- Shared integer "lock" initialized to 0;

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

      /* critical section */

       lock = 0;

       /* remainder section */

      } while (true);
```

(TAS) test and set

lock=1 → expected=0 → ≠ → تغییری نکن

How requirements are satisfied?
- Mutual exclusion (?)
- Progress (?)
- Bounded waiting (?)

Bounded-waiting mutual exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&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);
```

bool waiting[n] { init. → false
bool lock

راه حل
(نرم افزاری)

waiting[i] == false
OR
key == false

کمک به رعایت عدالت
در نقل و انتقال حق ورود

؟ Peterson OR Lamport ؟
چون به معماری وابسته نیست.

3) OS solution!: Mutex locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
  - HW < TAS, CAS
  - Peterson
  - mutex (OS)

  } busy waiting
- Simplest is *mutex* lock (*mut*ual *ex*clusions)
  - mut: متقابل
  - ex: انحصار
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
    - (1) disable interrupt → CPU utilization ↓
    - TAS (Test And Set)
    - (2) CAS (Compare And Swap)
- But this solution requires busy waiting
  - This lock therefore called a spinlock
    - → while → CPU مشغول است!
    - → کسانی که در انتظار رفتن به ناحیه بحرانی هستند
    - → no context switch → برای زمان های کوتاه بهتر!

short duration

acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

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

How requirements are satisfied?

- Mutual exclusion (?)
- Progress (?)
- Bounded waiting (?)

What is the main problem of all mentioned methods?

Busy waiting!

4) Semaphore

Busy waiting → send processes from ready queue to another queue

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities
- Semaphore *S* – integer variable
- Can only be accessed via two indivisible (atomic) operations

داخل خود wait و signal هم ناحیه بحرانی داریم ← CAS/TAS

wait() and **signal()**

aquire = ↓ ↳ ≡ release

```
wait(S)
{
    while (S <= 0)
        ; // busy wait
    S--;
}
```

← برای یک بهتر اینو گذاشته ولی نباید while نباید و میفرسته توی یه صف دیگه

پیاده سازی داخل صفحه بعد

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

* ضمن فعالیت توی ناحیه بحرانی باشه ← مقدار اولیه S ← 1

No busy waiting in Semaphore

value = -x → x تا توصف صف
(x ∈ IN)

- Have a FIFO queue for waiting process

مقدار اولیه ← 1

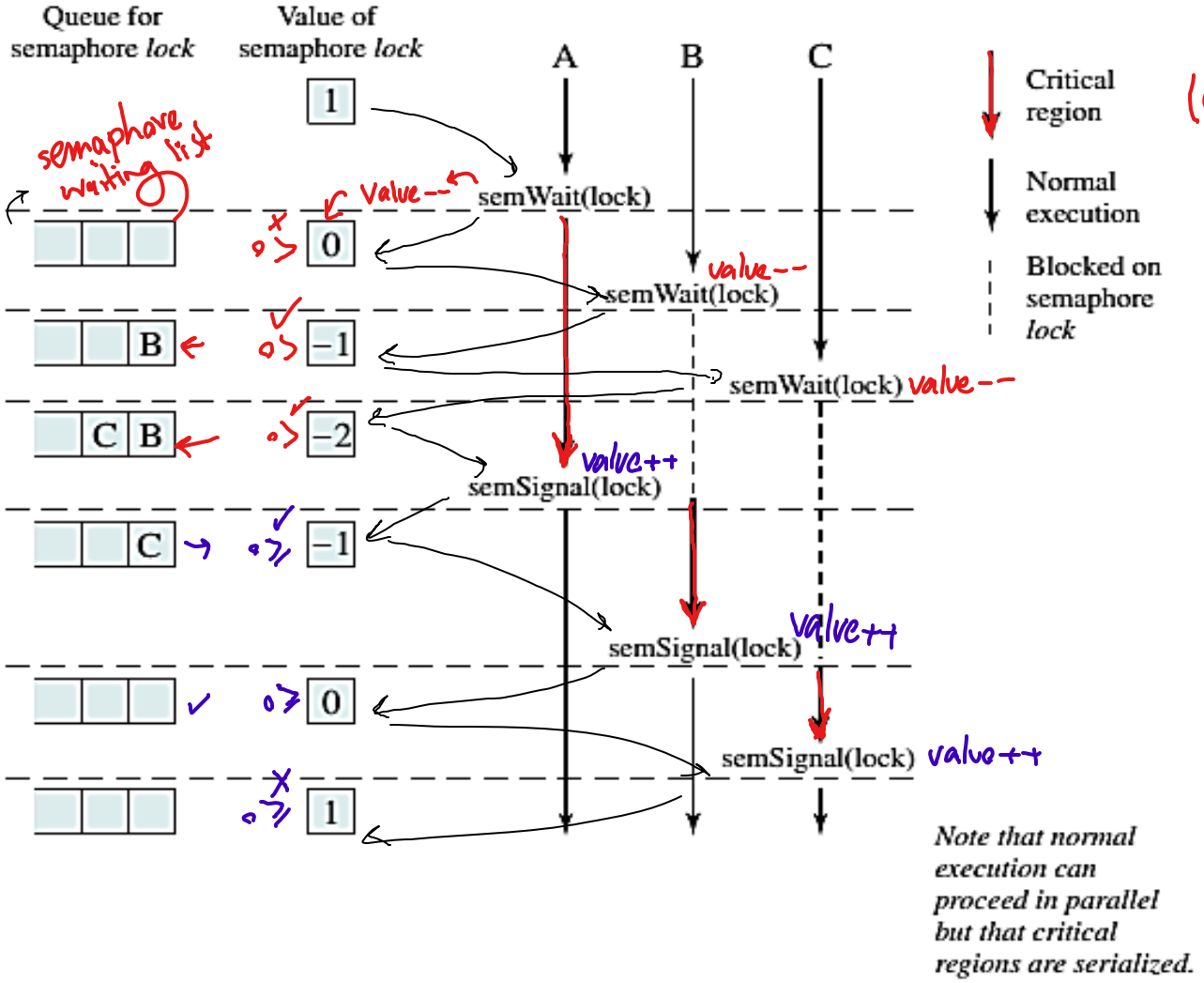
```
typedef struct{
   int value;
   struct process *list;
   } semaphore;
```

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

آوردن P توی Ready Queue

Accessing shared data by Semaphore



Types of semaphore

- **Types**
  - **Binary semaphore** (same as mutex lock)
  - **Counting semaphore** (suitable for managing number of resources)
- **Can solve various synchronization problems**
- **Example:**
  - Consider P_1 and P_2 that require S_1 to happen before S_2

Create a semaphore "**synch**" initialized to zero

```
P1:
    S1;
    signal(synch);
```

```
P2:
    wait(synch);
    S2;
```

Semaphore points

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time (why?)

  - wait() and signal() must be atomic!
  - wait() and signal() generate a Critical Section Problem!
  - How to solve?
    - Uniprocessors
      - ✓ Disabling interrupts
    - SMP (Multiprocessors)
      - ✓ Disabling interrupts (bad performance effect)
      - ✓ Other methods: compare_and_swap() and spinlock (is it good to have busy waiting?)

Two implementations of semaphores

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set
s.flag to 0) */;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count<= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow inter-
rupts*/;
    }
    else
        allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count<= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

Problems with semaphores

- ➢ **Be careful in the usage**
  - o Deadlock, Starvation, Priority inversion

P_0

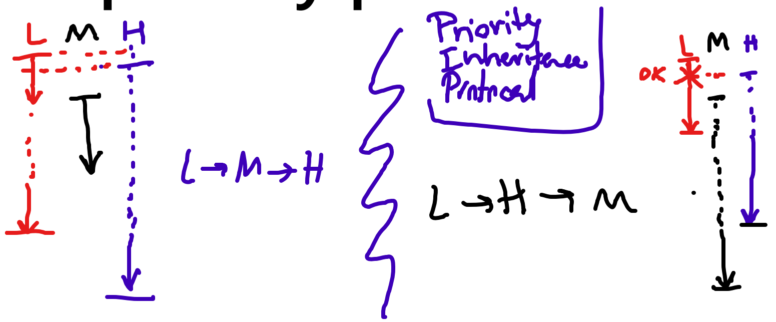
```
wait(S);
wait(Q);
...
signal(S);
signal(Q);
```

P_1

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

- ➢ **Starvation**
  - o LIFO queue

- ➢ **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - o Example: L (R) < M < H (R)
  - ➢ Solved via priority-inheritance protocol



Classic synchronization problems

- The bounded-buffer problem
- The readers-writers problem
- The dining-philosophers problem

How can semaphore solve these problems?

The bounded-buffer problem

```
int n;
semaphore mutext = 1;
semaphore empty = n;
semaphore full = 0;
```

Producer

```
do {
        ...
    /* produce an item in next_produced */
        ...

    wait(empty);
    wait(mutex);

        ...

    /* add next produced to the buffer */

        ...

    signal(mutex);
    signal(full);

} while (true);
```

Consumer

```
do {
        wait(full);
        wait(mutex);
            ...

        /* remove an item from buffer to next_consumed */
            ...

        signal(mutex);
        signal(empty);

            ...

        /* consume the item in next consumed */
            ...

    } while (true);
```

The readers-writers problem

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

read count ← mutex

Writers

```
do {
        wait(rw_mutex);

            ...

        /* writing is performed */

            ...

        signal(rw_mutex);


    } while (true);
```

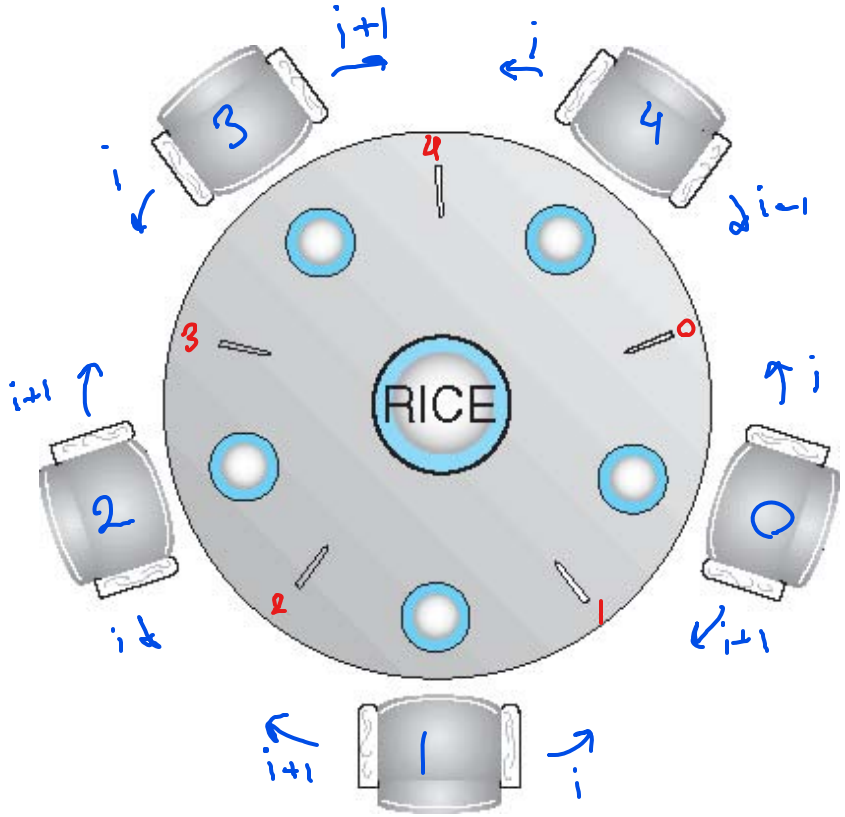
Readers

```
do {
            wait(mutex);
            read_count++;
            if (read_count == 1)
               wait(rw_mutex);
            signal(mutex);
                ...

            /* reading is performed */

                ...

            wait(mutex);
            read_count--;
            if (read_count == 0)
                signal(rw_mutex);
            signal(mutex);
        } while (true);
```

* reader ها باید نوشتن و سپس هم زمان بخوانند ، NP

The dining-philosophers problem

Thinking and eating alternatively




```
semaphore chopstick[5];
```

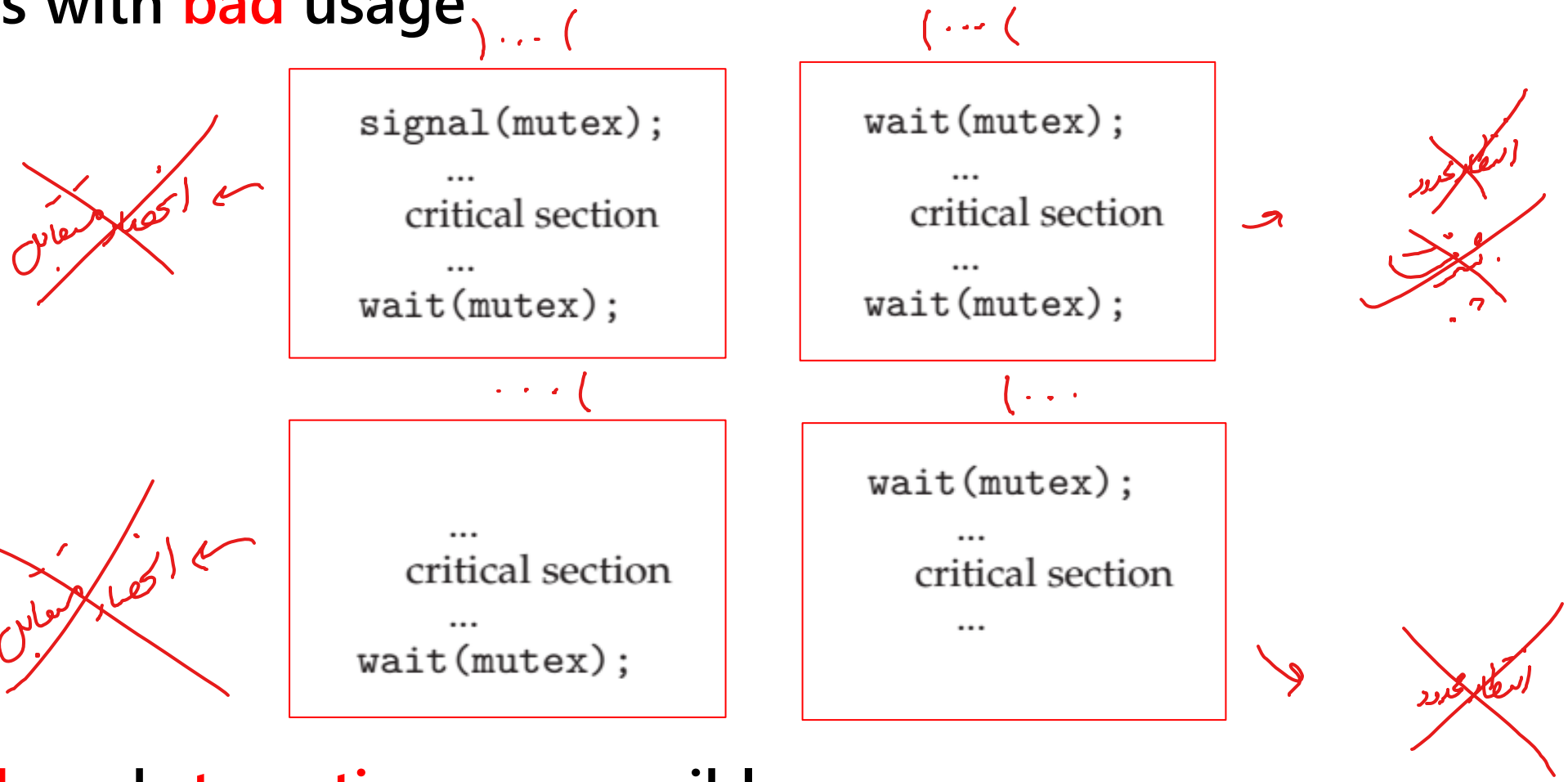
```
do {
     wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );


       //  eat


     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

       //  think

} while (TRUE);
```

Any problem?

deadlock!

Other problems with semaphore

- **Problems with bad usage**



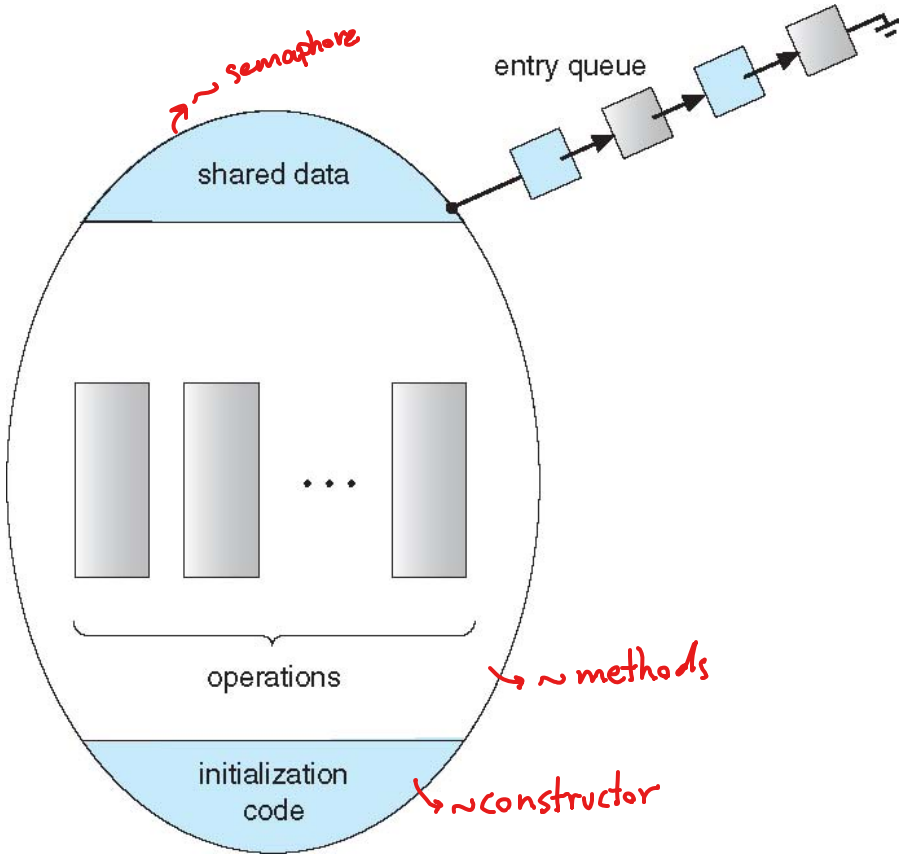
- **Deadlock and starvation are possible.**

5) Monitor → class شبیه ساختار (semaphore پیشرفته)

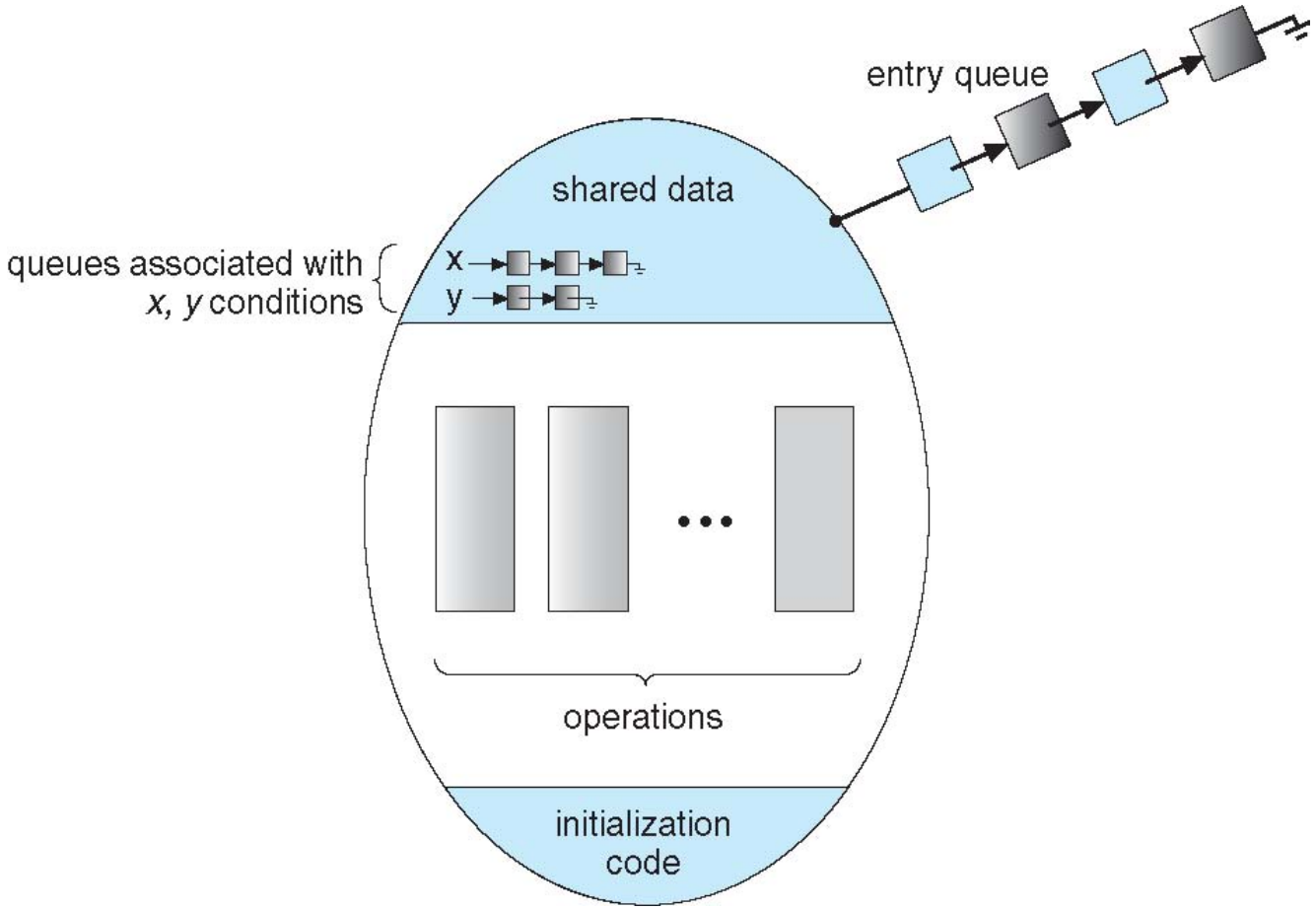
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

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

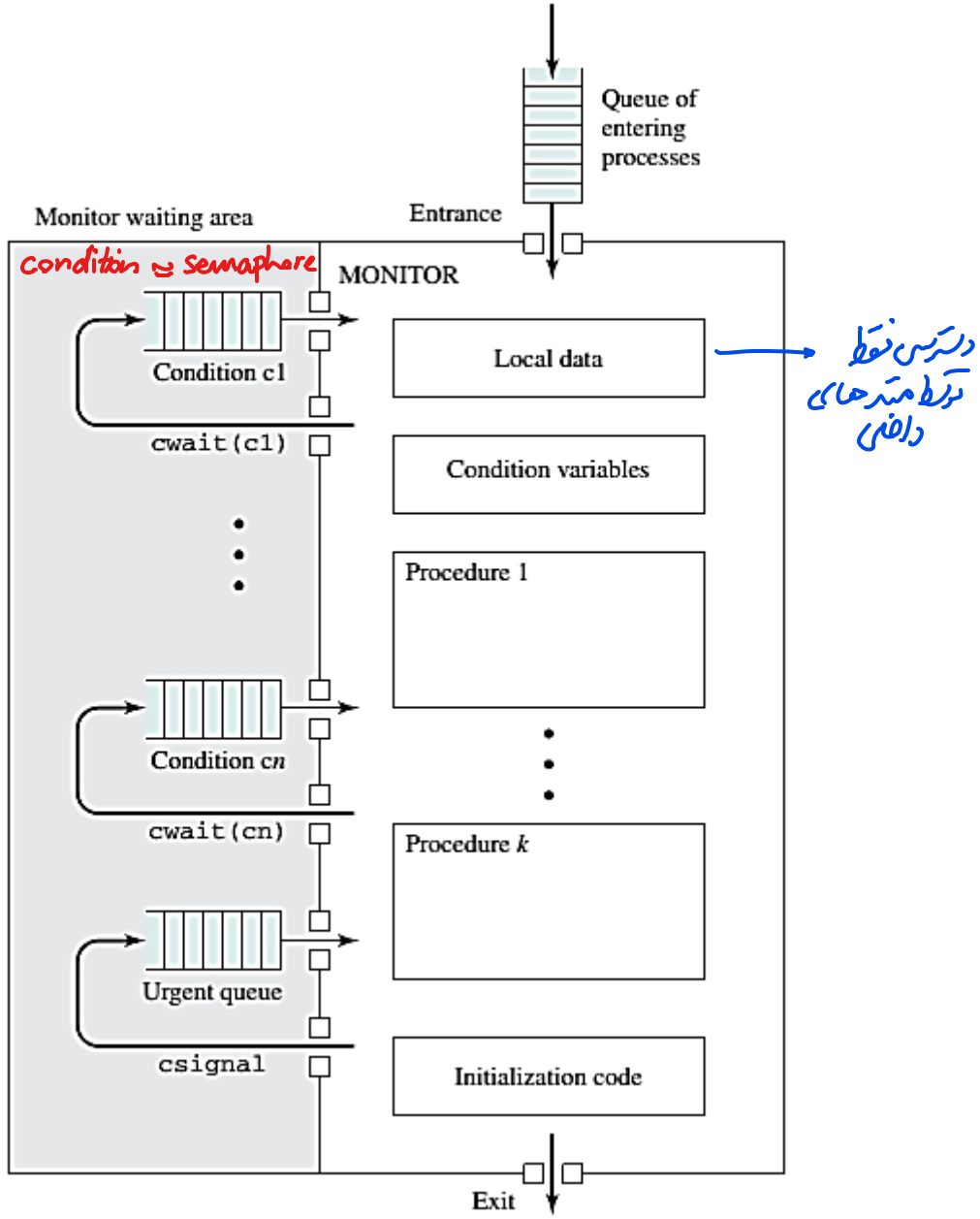
```
monitor monitor-name
{
        // shared variable declarations

        procedure P1 (…) { …. }

        procedure Pn (…) {……}

      Initialization_Code (…) { … }

}
```



Monitor (with *condition variables)*



Structure of a Monitor

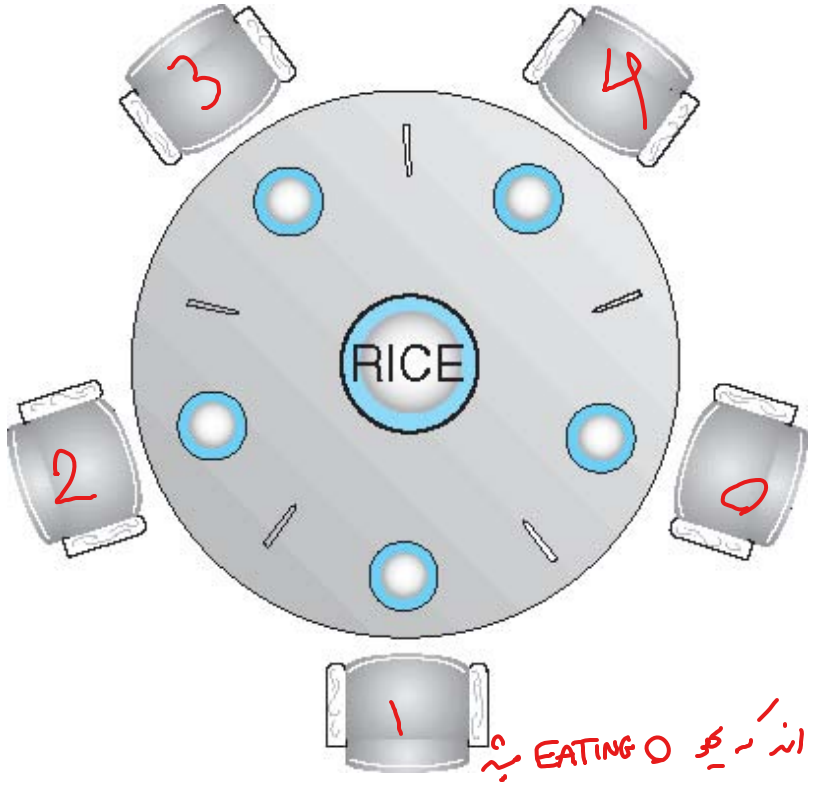


The dining-philosophers problem

```
monitor DiningPhilosophers
{
    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);
    }
```

```
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;
}

}
```

The dining-philosophers problem




```
DiningPhilosophers.pickup(i);

EAT

DiningPhilosophers.putdown(i);
```

Any problem?

No deadlock

Starvation is possible

Solving bounded-buffer using a Monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;                 /* condition variables for synchronization */
void append (char x)

{
      if (count == N) cwait(notfull);           /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (notempty);                       /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);      /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                  /* one fewer item in buffer */
      csignal (notfull);                        /* resume any waiting producer */
}
{                                                   /* monitor body */
      nextin = 0; nextout = 0; count = 0;           /* buffer initially empty */
}
```

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
      take(x);
      consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

Points to monitor

- Monitors can be implemented by semaphores (See the textbook).
- OSes support
  - Monitor, semaphore, spinlock, mutex
  - Examples
    - Solaris
    - Windows
    - Linux
    - Pthreads
- Alternative approaches
  - Transactional Memory
  - OpenMP
  - Functional Programming Languages

```
void update(int value)
{
        #pragma omp critical
        {
          count += value
        }
}
```

Questions?