CS & IT ENGINEERING

Operating System

REVISION

Process Synchronization (Part-03)

Lecture No.- 07

By- Dr. Khaleel Khan Sir

Recap of Previous Lecture

Topic

Peterson Solution

Hardware Synchronization

Semaphores

Topics to be Covered

- **Topic** Monitors
- **Topic** Classical IPC Problems
- **Topic** Concurrency
- **Topic**

Topic : Monitors

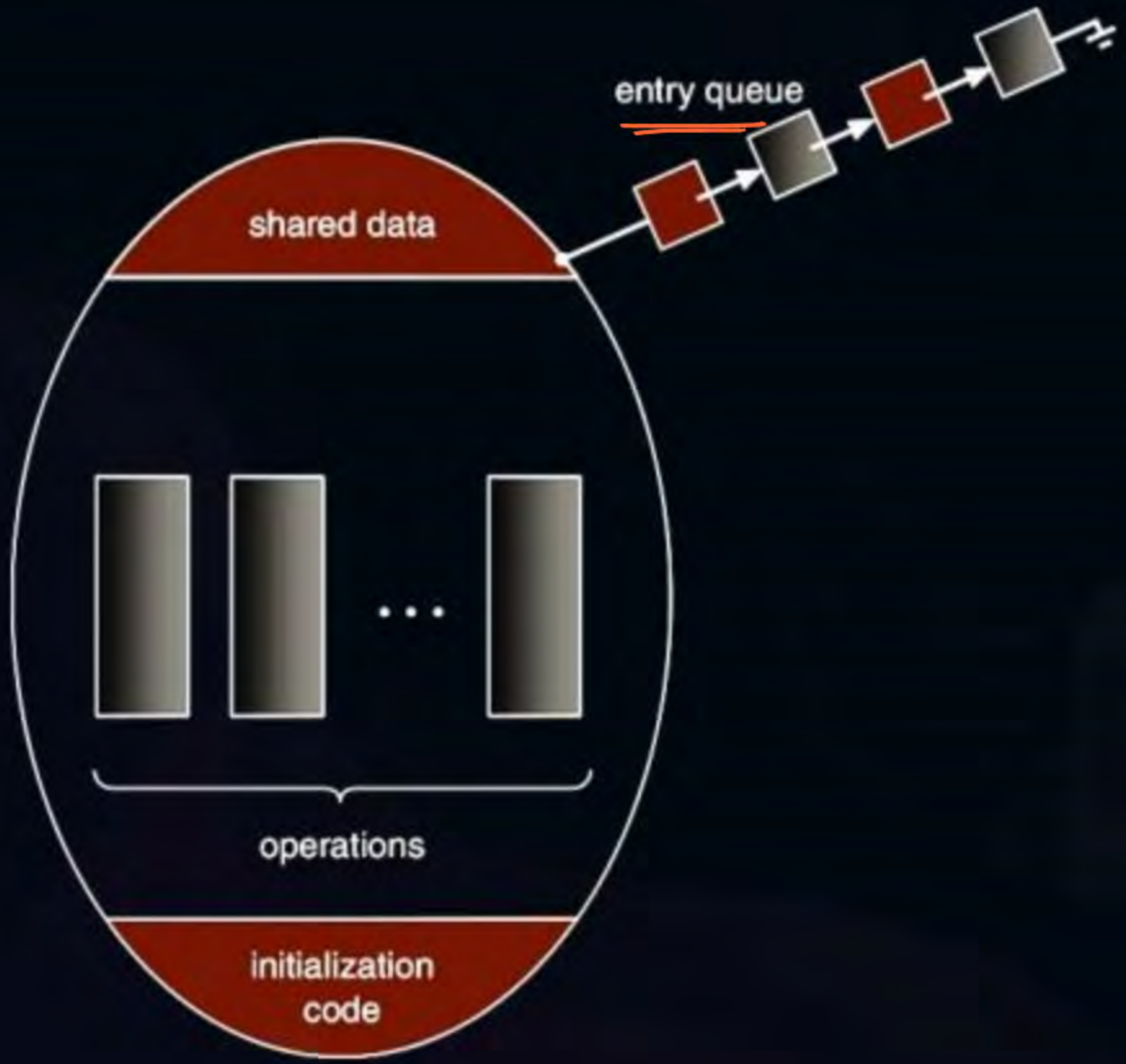
B. Hansen & Hoare :

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

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

        procedure P2 (...) { .... }

        procedure Pn (...) {......}

  initialization code (...) { ... }
}
```

Constructor

Topic : Schematic view of a Monitor



Topic : Monitor Implementation Using Semaphores

- Variables

  semaphore mutex
  mutex = 1

- Each procedure ***P*** is replaced by

```
wait(mutex);
    ...
body of P;
    ...
signal(mutex);
```

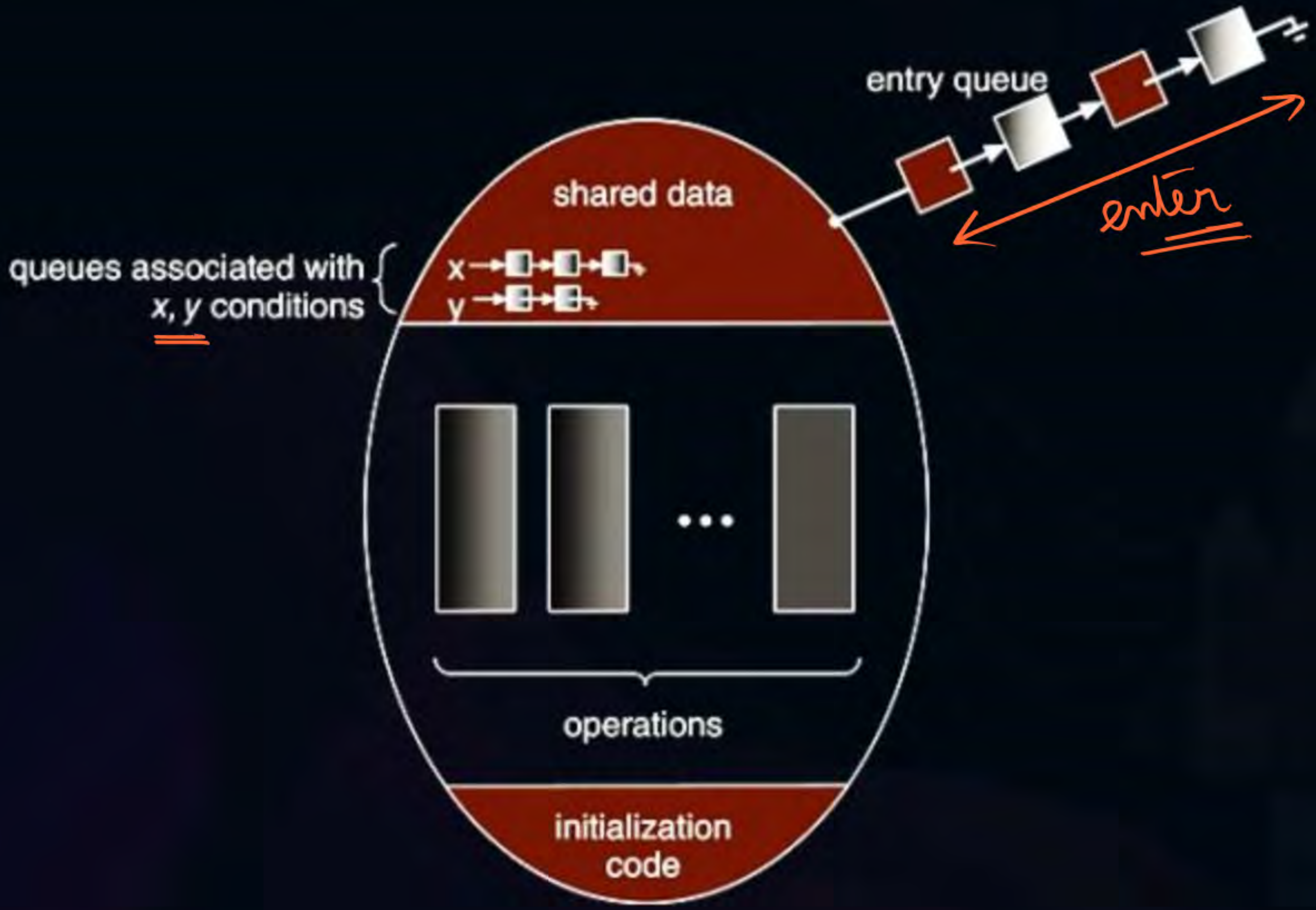
9 Mutual exclusion within a monitor is ensured

Topic : Condition Variables

- condition x, y;
- Two operations are allowed on a condition variable:
  - x.wait() – a process that invokes the operation is suspended until x.signal()

    Block
  - x.signal() – resumes one of processes (if any) that invoked x.wait()
  - If no x.wait() on the variable, then it has no effect on the variable

Topic : Monitor with Condition Variables



Topic : Usage of Condition Variable Example

- Consider P_1 and P_2 that that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
  - One condition variable "x" initialized to 0
  - One Boolean variable "done"
  - **F1:**

```
S1;
done = true;
x.signal();
```

  - **F2:**

```
if done = false
x.wait()
S2;
```

Topic : Resuming Processes within a Monitor

- If several processes queued on condition variable **x**, and **x.signal()** is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form **x.wait(c)**

where:

- **c** is an integer (called the priority number)
- The process with lowest number (highest priority) is scheduled next

Topic : Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource

```
R.acquire(t);
        ...
    access the resource;
        ...

R.release;
```

- Where R is an instance of type Resource Allocator

Topic : Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
- The process with the shortest time is allocated the resource first
- Let R is an instance of type ResourceAllocator (next slide)
- Access to ResourceAllocator is done via:

```
R.acquire(t);
    ...
access the resurce;
    ...
R.release;
```

- Where t is the maximum time a process plans to use the resource

Topic : A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
        boolean busy;
        condition x;
        void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = true;
        }
        void release() {
            busy = false;
            x.signal();
        }
  initialization code() {
        busy = false;
        }
}
```

Topic : Single Resource Monitor (Cont.)

- Usage:

    acquire

    ...

    release

- Incorrect use of monitor operations
    - release() ... acquire()
    - acquire() ... acquire())
    - Omitting of acquire() and/or release()

Topic : Classical Problems of Synchronization

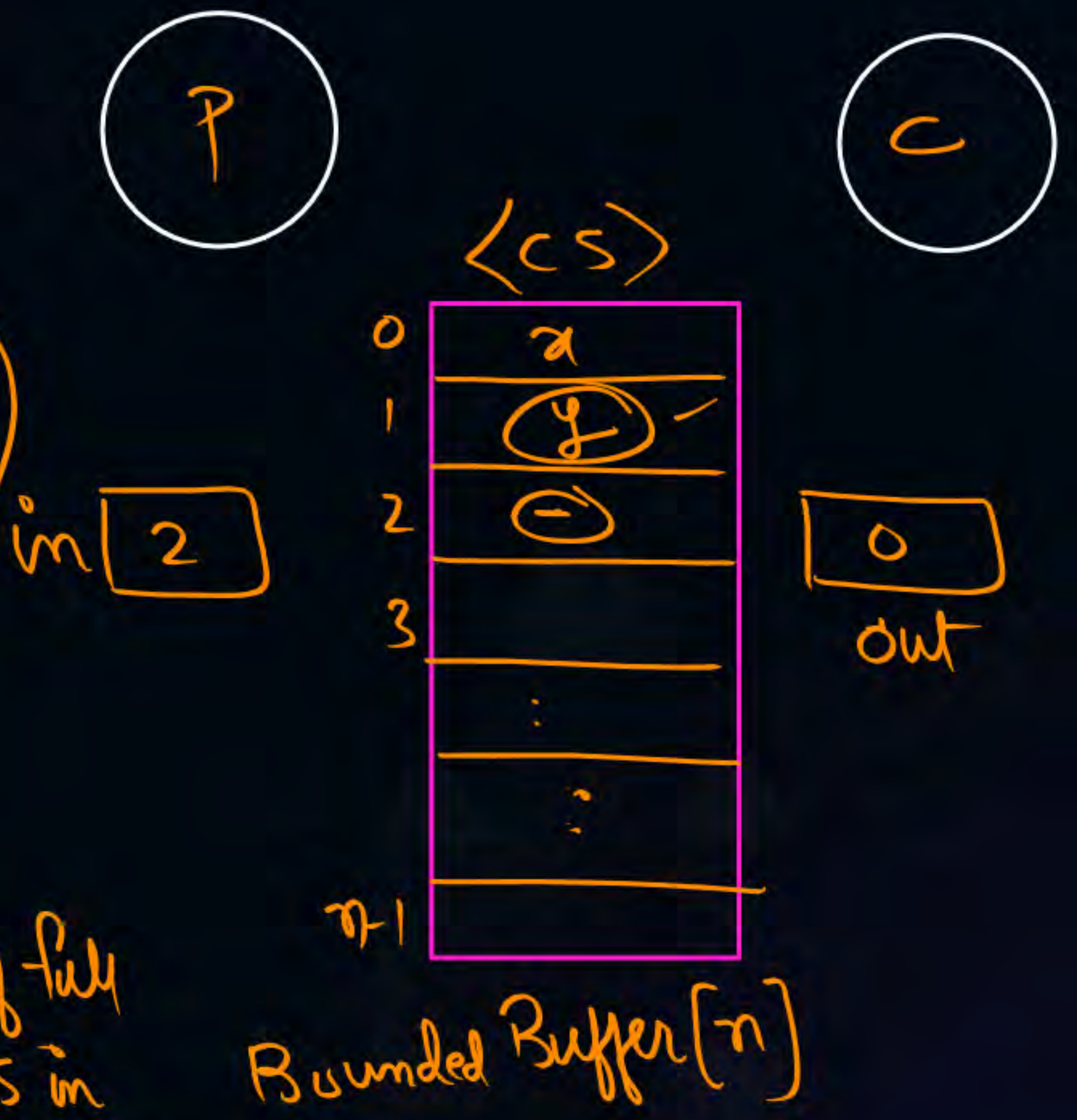
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem Producer-Consumer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Topic : Bounded-Buffer Problem

- *n* buffers, each can hold one item
- BS — Semaphore **mutex** initialized to the value 1
- CS — Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n

(No. of empty (free) slots)

(No. of full slots in Buffer)



Topic : Bounded Buffer Problem (Cont.)

The structure of the producer process

```
while (true) {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next_produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Data item

Entry

Exit

Topic : Bounded Buffer Problem (Cont.)

- The structure of the consumer process

Deadlock:

```
while (true) {
  wait(full);
    wait(mutex);
  ...
   /* remove an item from buffer to next_consumed */
  ...
signal(mutex);
signal(empty);
  ...
    /* consume the item in next consumed */
  ...
 }
```

mutex = ~~1~~ 0

full = ~~0~~ -1

Empty = n

C : P(mutex); P(full);

Blocked

→ If there are multiple Producers ($n \geq 1$) & multiple consumers ($m \geq 1$), then will the Code continue to work Correctly? [Sufficient ?]

Topic : Readers-Writers Problem

DB (records)

Starvation to W's

- 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

First R-W

- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities

Topic : Readers-Writers Problem (Cont.)

- Shared Data
  - Data set "DB"
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1 "rc"
  - Integer **read_count** initialized to 0

(rc) → No. of Reader's in DB

Topic : Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
    wait(rw_mutex);
        ...
    /* writing is performed */
        ...
    signal(rw_mutex);
}
```

Topic : Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
    wait(mutex);   r2 r3 - - - rm
    read_count++;  <CS>
    if (read_count == 1) /* first reader */
        wait(rw_mutex);  ✓ r1
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0) /* last reader */
    signal(rw_mutex);
    signal(mutex);
}
```

Topic : Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the "First reader-writer" problem.
- The "Second reader-writer" problem is a variation the first reader-writer problem that state:

  "Principle of Turnstile"

  - Once a writer is ready to write, no "newly arrived reader" is allowed to read.
- Both the first and second may result in starvation. leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Code the Problem of $\dfrac{\text{Second R-W}}{\text{First W-R}}$

$r_1 \quad r_2$

r_3

<DB>

$t_0 : r_1$

$t_1 : r_2$

$t_2 : r_3$

$t_3 : w_1$

$t_4 : r_4$ ✗

Topic : Dining-Philosophers Problem

Foundation of Deadlock

- N philosophers' sit at a round table with a bowl of rice in the middle.
- They spend their lives alternating thinking and eating.
- They do not interact with their neighbors.
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Topic : Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher i :

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

     /* eat for awhile */        CS

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

      /* think for awhile */

}
```

What is the problem with this algorithm?

BSEM chopstick[N] = {1}

→ possibility of deadlock :

- → All Hungry
- → Pi's get Pre Empted, b/w Two wait op-ns

N

N-1 | 1
LR | RL

$P_0 \leftarrow f_0$

$P_1 \leftarrow f_1$

$P_2 \leftarrow f_2$

$P_3 \leftarrow f_3$

$P_4 \leftarrow f_4$

"Mutual Exclusion"

Max Concurrency: "2" ✓

Topic : Monitor Solution to Dining Philosophers

```
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;
    }
                                    Block

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

```
        void test (int i) {          L
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {   R
                state[i] = EATING ;
                    self[i].signal () ;
            }
        }
    initialization_code() {
            for (int i = 0; i < 5; i++)
            state[i] = THINKING;
        }
}
```

Topic : Solution to Dining Philosophers (Cont.)

- Each philosopher "i" invokes the operations **pickup()** and **putdown()** in the following sequence:

```
DiningPhilosophers.pickup(i);
    /** EAT **/
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

→ Barber Shop Simulation [Tanenbaum]

→ Cigarette-Smokers Problem :

Concurrency vs Parallelism

(Uniprocessor)

<Interleaved Execution>

(Multi-Core)

<Real parallelism>

Conditions of Concurrency

$S_i \; ; \; S_j$

$R(S_i) \quad W(S_i) \quad R(S_j) \quad W(S_j)$

(i) $R(S_i) \cap W(S_j) = \phi$

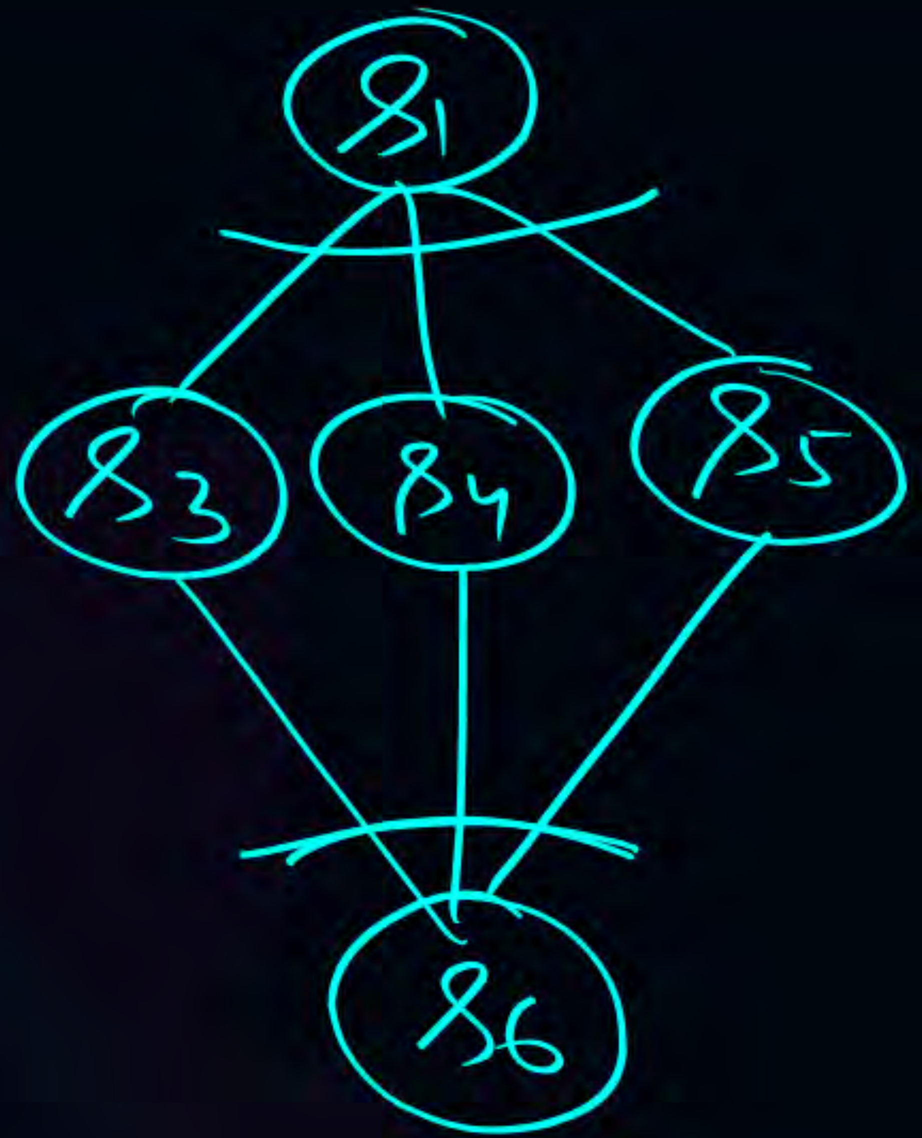
(ii) $R(S_j) \cap W(S_i) = \phi$

(iii) $W(S_i) \cap W(S_j) = \phi$

Bernstien's conc. conditions

(i) Parbegin-Parend / Cobegin-Coend

S1;
Parbegin
S3;
S4;
S5;
Parend
S6;



integer x = 0, y = 30;
BSem mx = 1, my = 1;

Cobegin
begin P(mx);
1. x = 1;
2. y = y + x;
end V(mx);
begin P(mx);
3. y = 2;
4. x = x + 3;
end V(mx)
Coend

1) 1, 2, 3, 4
x = 4
y = 2

2) 3, 4, 1, 2
x = 1
y = 3

3) 1, 3, 2, 4
x =
y =

THANK - YOU