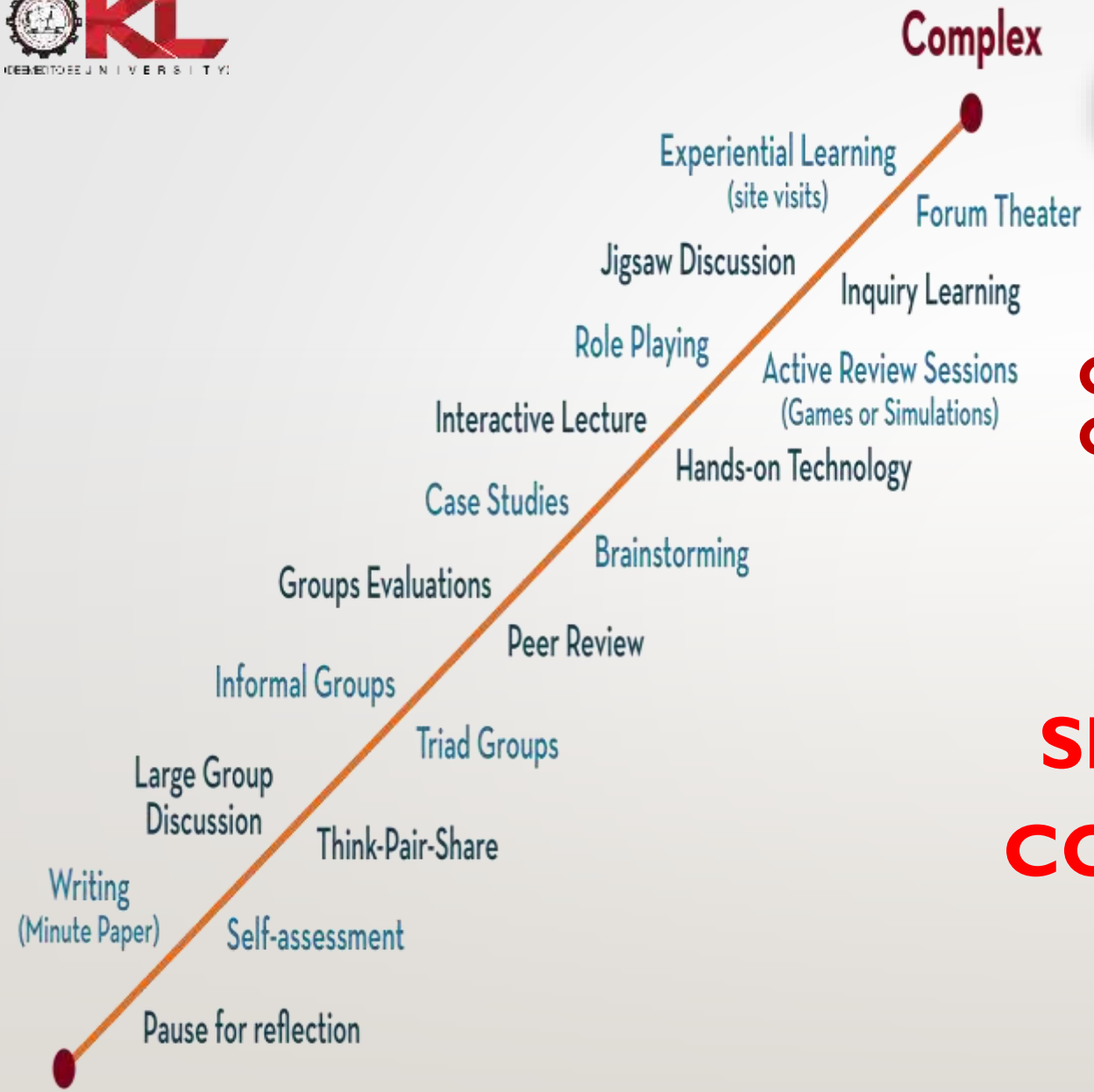


CO3

Session - 4

COURSE NAME: OPERATING SYSTEMS
COURSE CODE: 23CS2104RA

SEMAPHORES, CLASSICAL CONCURRENCY PROBLEMS, MONITORS.

AIM OF THE SESSION

To familiarize students with the basic concept of semaphores, Monitors, Inter Process Communication.

INSTRUCTIONAL OBJECTIVES

This Session is designed to:

1. Demonstrate what is meant by semaphore.
2. Describe the types of semaphores.
3. List out the differences between mutex and semaphore .
4. Describe the role of semaphores in solving critical section problems.
5. A Case Study of classical synchronization problems using semaphores.
6. Describe the role of Monitors.
7. Describe the role of Inter Process Communication.

LEARNING OUTCOMES

At the end of this session, you should be able to:

1. Defines what is semaphore.
2. Describe binary semaphore and counting semaphore.
3. Summarize the Role of semaphores in solving critical section problems.
4. A Comparative Study of Classical Synchronization Problems.
5. Defines what is Monitors.

SEMAPHORES

- Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using integer value, which is known as Semaphore.

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.

- Semaphore S is an integer variable which is used in mutual exclusive manner used by concurrent cooperative processes in order to achieve synchronization.

- A semaphore is an integer variable, shared among multiple processes. The main aim of using a semaphore is process synchronization and access control for a common resource in a concurrent environment.

INTERACT WITH SEMAPHORE

Semaphore S can only be accessed and modified via two indivisible (atomic) operations

Sem_wait () or P() -→ Proberen → "to test".

Sem_signal () or sem_post() or V() → Verhogen → "to increment".

- Definition of the **sem_wait() or P() operation**

```
int sem_wait(sem_t *s)
{
   while (S <= 0);      /* busy wait */

      S--;
                        /*decrement the value of
                             semaphore S by one.
                          wait if value semaphore S is
                             negative */
}
```

- Definition of the **sem_post()** or **V()** operation

```
Int sem_post(sem_t *s)
{
      S++;      /*increment the value
                 of semaphore s by one
                if there are one or more threads
                waiting wake          one*/
}
```

TYPES OF SEMAPHORE

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1 Same as a **mutex lock.**
  - Can solve various synchronization problems.
  - Consider P_1 and P_2 that require S_1 to happen before S_2

    Create a semaphore "**synch**" initialized to 0

    P1:

    S_1;

    signal(synch);

    P2:

    wait(synch);

    S_2;
  - Can implement a counting semaphore ***S*** as a binary semaphore

SEMAPHORE IMPLEMENTATION

- Must guarantee that no two processes can execute the **sem_wait()** and **sem_signal()** on the same semaphore at the same time.
- Thus, the implementation becomes the critical section problem where the **sem_wait** and **sem_signal** code are placed in the critical section.
  - Could now have **busy waiting** in critical section implementation
    - But the implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

SEMAPHORE IMPLEMENTATION WITH NO BUSY WAITING

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to the next record in the list
- Two operations:
  - **Block** – place the process invoking the operation on the appropriate waiting queue
  - **Wakeup** – remove one of the processes in the waiting queue and place it in the ready queue.

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

IMPLEMENTATION WITH NO BUSY WAITING (CONT.)

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

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

THREAD TRACE: SINGLE THREAD USING A SEMAPHORE

Value of Semaphore	Thread 0	Thread 1
1	call sema_wait()	
1	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
0	sem_post() returns	
1		

THREAD TRACE: TWO THREADS USING A SEMAPHORE (1 CPU)

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	Running		Ready
0	(crit set: begin)	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0)→sleep	sleeping
-1		Running	*Switch → T0*	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

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
sem_wait(S);	**sem_wait(Q);**
sem_wait(Q);	**sem_wait(S);**
...	**...**
sem_signal(S);	**sem_signal(Q);**
sem_signal(Q);	**sem_signal(S);**

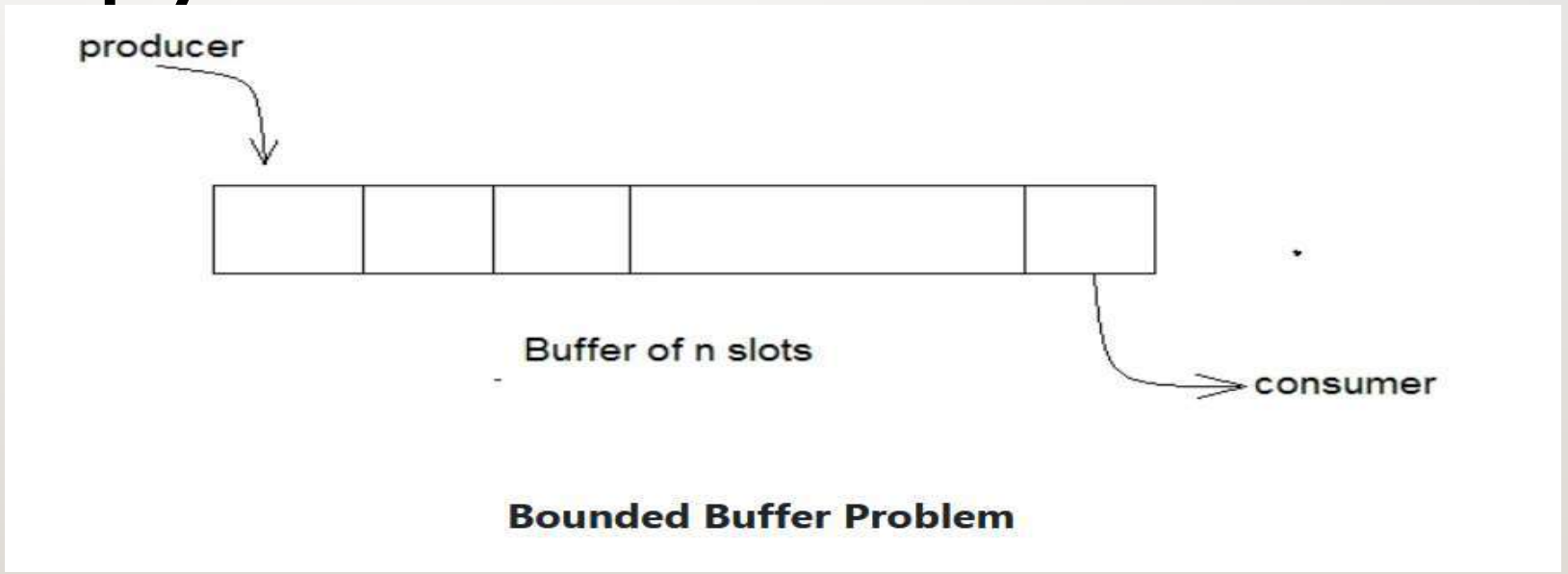
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when a lower-priority process holds a lock needed by a higher-priority process
  - Solved via **priority-inheritance protocol**

CLASSICAL PROBLEMS OF SYNCHRONIZATION

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

BOUNDED-BUFFER PROBLEM

- ***Buffer with n slots*** , each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value **0**
- Semaphore **empty** initialized to the value **n**



Bounded Buffer Problem

READERS-WRITERS PROBLEM

- 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
- Several variations of how readers and writers are considered – all involve some form of priorities

There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

READERS-WRITERS PROBLEM

Three variables are used: **mutex, wrt, readcount** to implement solution

1. Semaphore **mutex, wrt;**
   // semaphore **mutex** is used to ensure mutual exclusion when **readcount** is updated i.e. when any reader enters or exit from the critical section. semaphore **wrt** is used by both readers and writers.
2. **int** readcount;
   // **readcount** tells the number of processes performing read in the critical section, initially 0
3. Initial values mutex=1,readcount=0,wrt=1

WRITER PROCESS

1. Writer requests the entry to critical section.
2. If allowed i.e. **wait()** gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

WRITER PROCESS

The structure of a writer process.

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

READER PROCESS

1. Reader requests the entry to critical section.
2. If allowed:
    2. it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
    3. It then, signals mutex as any other reader is allowed to enter while others are already reading.
    4. After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

READER PROCESS

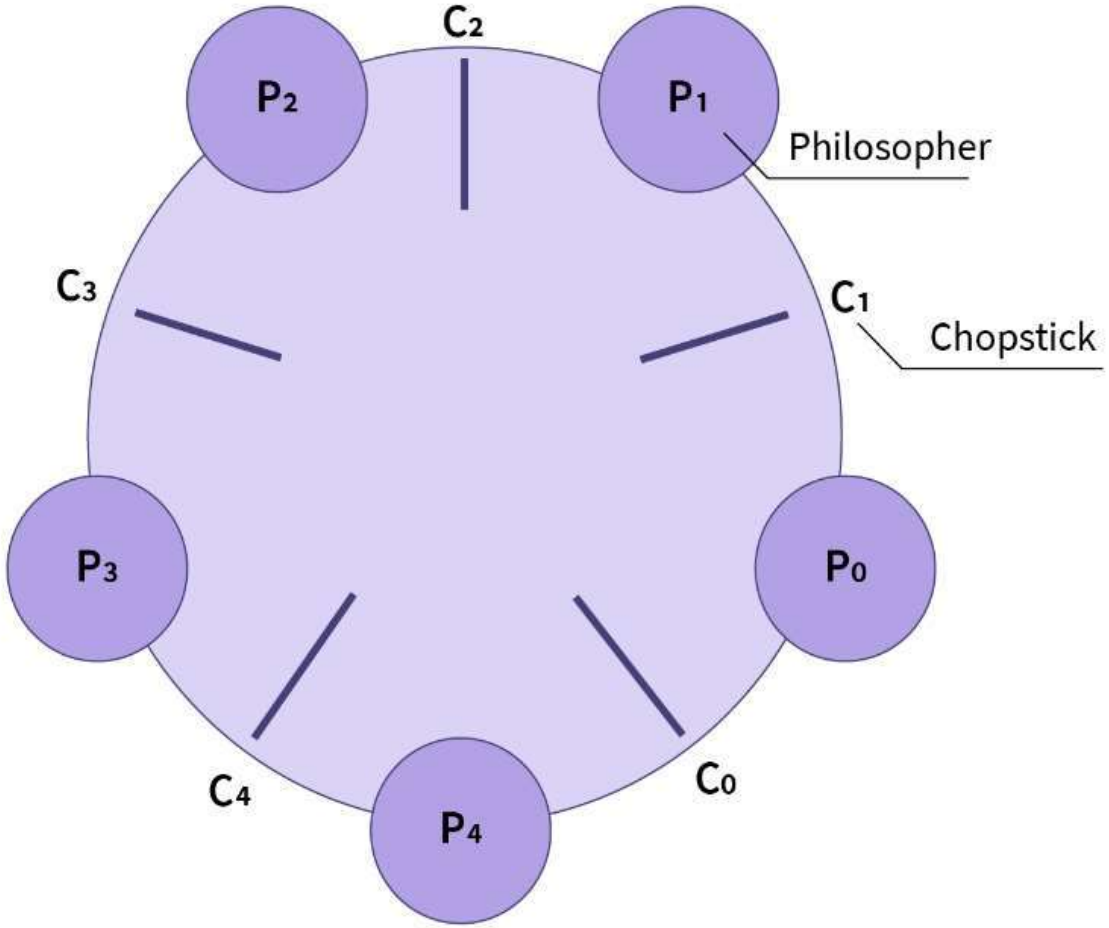
The structure of a reader process

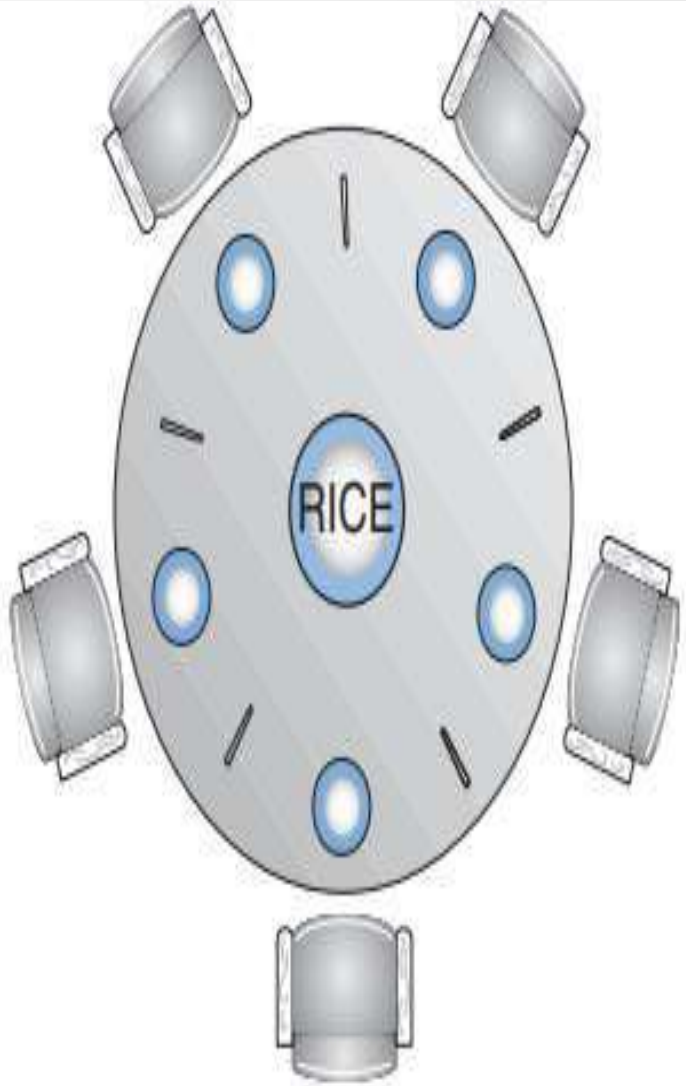
```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        . . .
    /* reading is performed */
        . . .
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

DINING PHILOSOPHER PROBLEM

- The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively.
- A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

Dining Philosopher problem





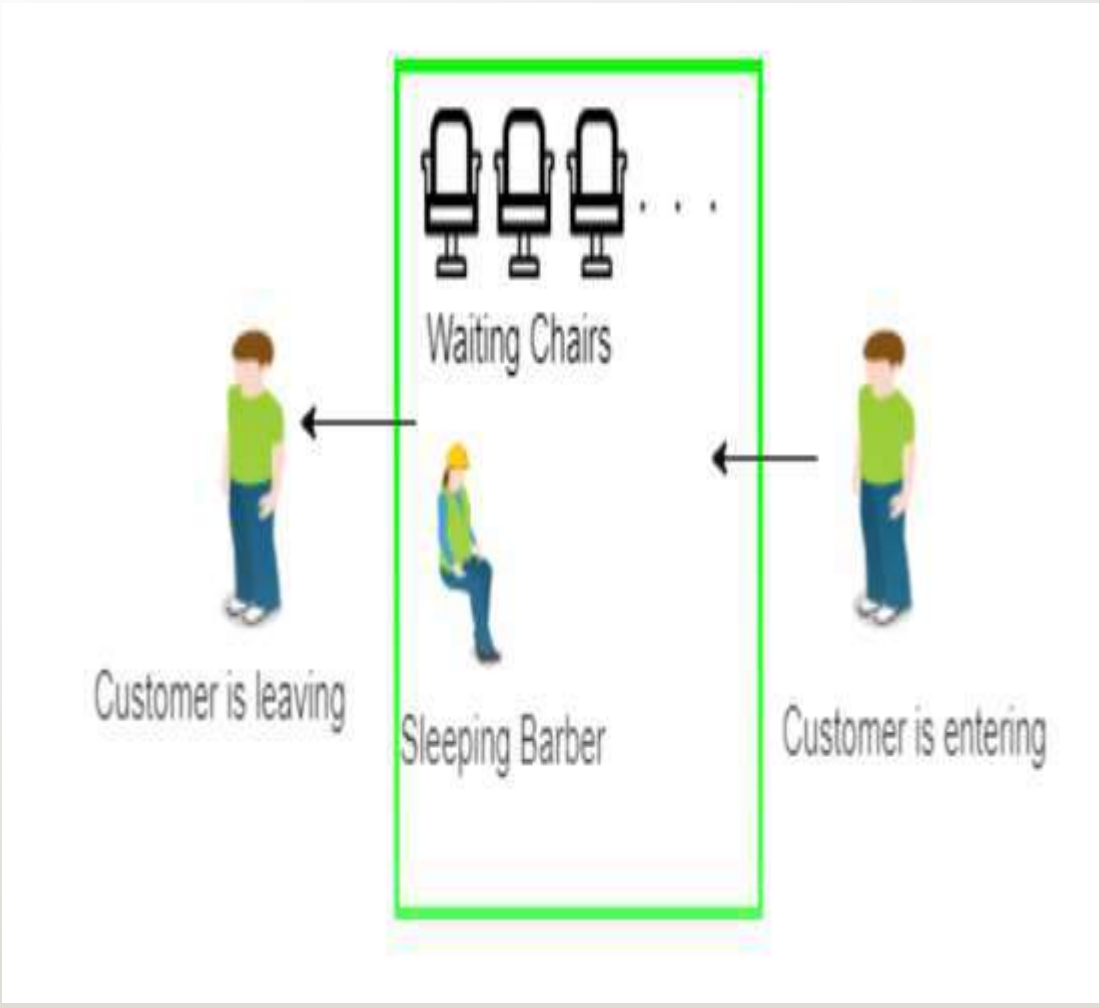
The structure of philosopher i:

semaphore chopstick[5];

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

SLEEPING BARBER PROBLEM

- The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system. The problem is as follows:
- There is a barber shop with one barber and several chairs for waiting customers. Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available. If there are no chairs available, the customer leaves. When the barber finishes with a customer, he checks if there are any waiting customers. If there are, he begins cutting the hair of the next customer in the queue. If no customers are waiting, he goes to sleep.



Sleeping Barber problem

Shared variables:

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;
```

```
Barber {
        while(true) {
                /* waits for a customer (sleeps). */
                down(Customers);

                /* mutex to protect the number of available seats.*/
                down(Seats);

                /* a chair gets free.*/
                FreeSeats++;

                /* bring customer for haircut.*/
                up(Barber);

                /* release the mutex on the chair.*/
                up(Seats);
                /* barber is cutting hair.*/
        }
}
```

Sleeping Barber problem

```
Customer {
      while(true) {
            /* protects seats so only 1 customer tries to sit
               in a chair if that's the case.*/
            down(Seats); //This line should not be here.
            if(FreeSeats > 0) {

                  /* sitting down.*/
                  FreeSeats--;

                  /* notify the barber. */
                  up(Customers);

                  /* release the lock */
                  up(Seats);

                  /* wait in the waiting room if barber is busy. */
                  down(Barber);
                  // customer is having hair cut
            } else {
                  /* release the lock */}}}.
```

PROBLEMS WITH SEMAPHORES

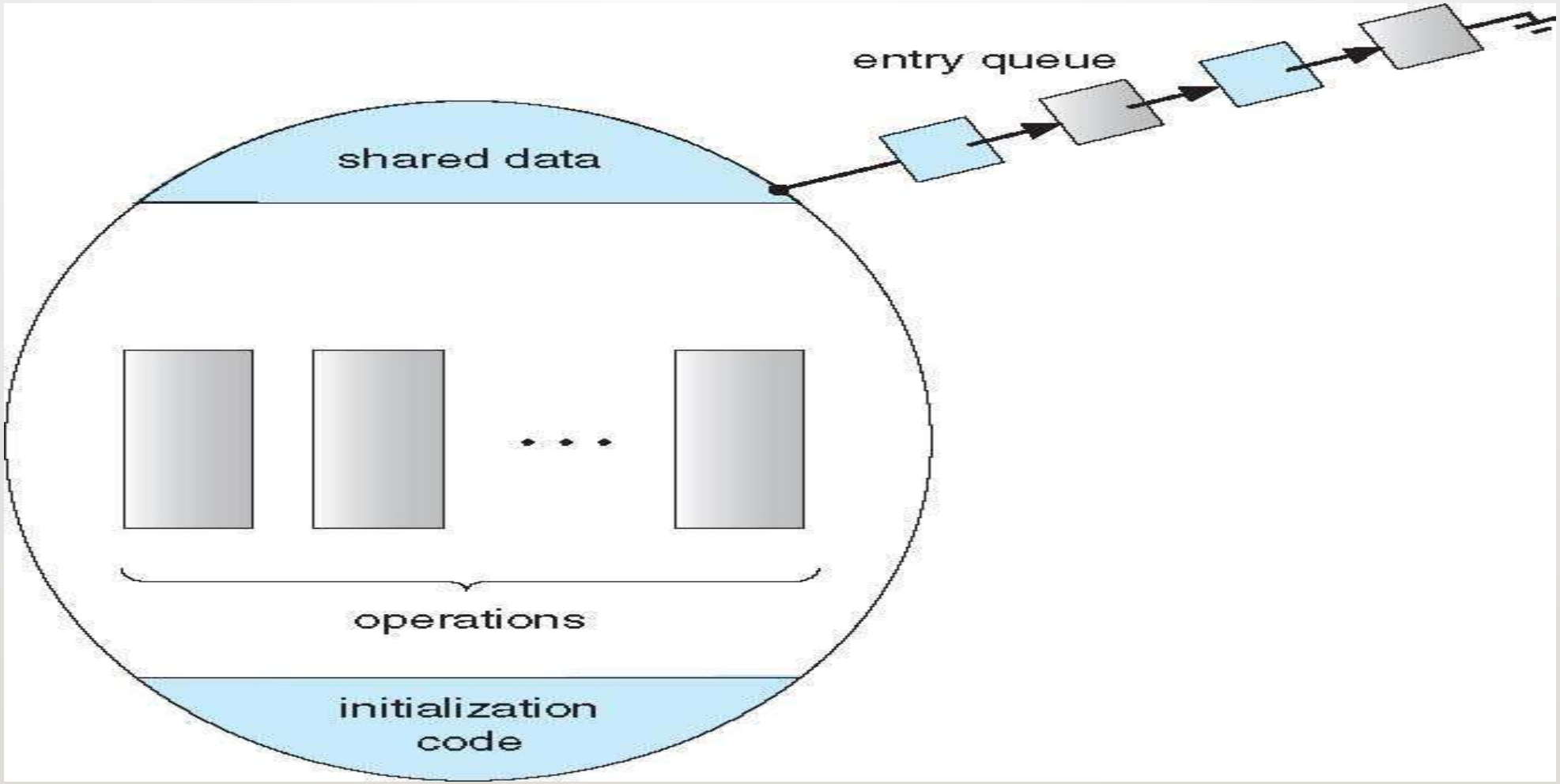
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

MONITORS

- 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
- But not powerful enough to model some synchronization schemes

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

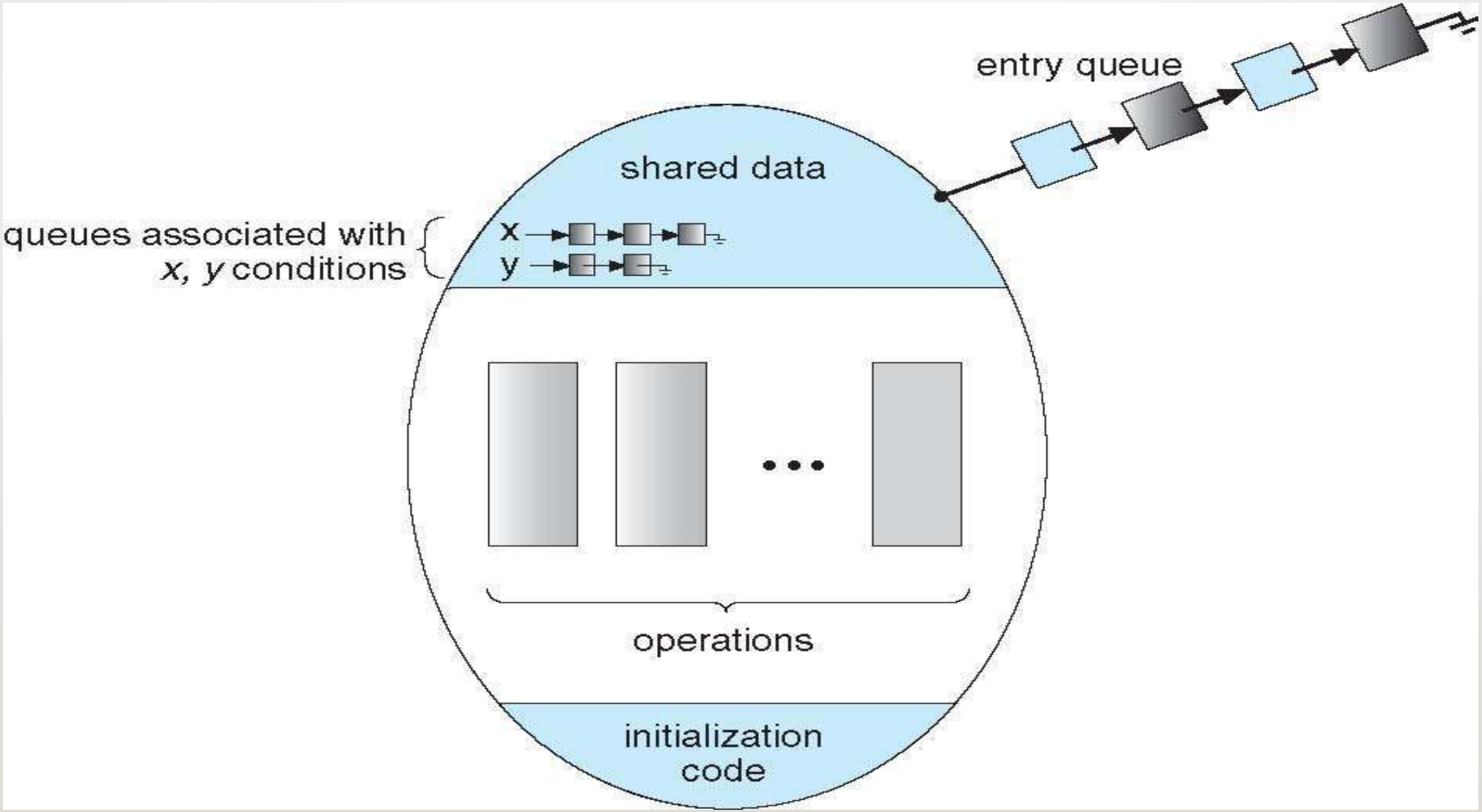
SCHEMATIC VIEW OF A MONITOR



CONDITION VARIABLES

- **condition x, y;**
- Two operations on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of the processes (if any) that invoked **x.wait()**
    - If no **x.wait()** on the variable, then it does not affect the variable

MONITOR WITH CONDITION VARIABLES



CONDITION VARIABLES CHOICES

- If process P invokes **`x.signal()`**, with Q in **`x.wait()`** state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves the monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementers can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

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

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;
    }
}
```

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

MONITOR IMPLEMENTATION USING SEMAPHORES

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
body of F;
    …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

MONITOR IMPLEMENTATION – CONDITION VARIABLES

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

MONITOR IMPLEMENTATION (CONT.)

- The operation **x.signal** can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

RESUMING PROCESSES WITHIN A MONITOR

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is the **priority number**
  - Process with lowest number (highest priority) is scheduled next

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;   }}
```

SELF-ASSESSMENT QUESTIONS

1. What is a semaphore, and why is it used in concurrent programming?

2. Differentiate between binary semaphores and counting semaphores. Provide examples of situations where each type might be more appropriate.

3. What is a deadlock, and how can semaphores be involved in deadlock situations? How can you prevent or avoid deadlocks when using semaphores?

4. What is a Monitor, and how it used in concurrent programming?

5. What is meant by Inter-Process Communication?

TERMINAL QUESTIONS

1. What is the primary purpose of using semaphores in concurrent programming, and how do they help in achieving synchronization?
2. How do you create and initialize a semaphore in a typical Unix-based terminal environment using system calls or command-line tools?
3. Explain the fundamental semaphore operations, "wait" (P) and "signal" (V). How are these operations used to control access to shared resources?
4. In a Unix-based terminal, how can you check the current value of a semaphore and its associated resources using command-line tools?
5. Describe a scenario where semaphores are useful in a multi-process environment. Provide an example of how you could use semaphores to address a synchronization problem in the terminal.
6. Can you explain how to handle errors or exceptions when working with semaphores in a terminal-based environment?

REFERENCES FOR FURTHER LEARNING OF THE SESSION

- **Reference Books:**

1. Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. *Operating systems: Three easy pieces*. Boston: Arpaci-Dusseau Books LLC, 2018. http://pages.cs.wisc.edu/~remzi/OSTEP/
2. Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. *Operating system concepts*. 10th edition, John Wiley & Sons, 2018.
3. Tanenbaum, Andrew. *Modern operating systems*. Pearson Education, Inc., 2009.

- **Sites and Web links:**

1. https://www.cse.iitb.ac.in/~mythili/os/
2. https://cse.iitkgp.ac.in/~sumantra/courses/os/os_pg.html
3. https://www.cse.iitd.ernet.in/os-lectures

THANK YOU

Team – Operating System