SNS COLLEGE OF TECHNOLOGY

Coimbatore-35.

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A+’ Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME : OPERATING SYSTEMS

II YEAR/ IV SEMESTER

UNIT – II PROCESS SCHEDULING AND SYNCHRONIZATION

Topic: Classical Problems of Process Synchronization

Dr.B.Vinodhini

Associate Professor

Department of Computer Science and Engineering

Classical Problems of Synchronization

- ***Bounded Buffer Problem***
- ***Readers writers Problem***
- ***Dinning Philosophers Problem***

The Bounded Buffer Problem

The Bounded Buffer Problem(Producer Consumer) Problem) one of the classic problems of Synchronization

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, **Producer** and **Consumer**, which are operating on the buffer.

- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

Solution to the Bounded Buffer Problem

- We will make use of three Semaphores
- 1.**m(mutex),**a binary semaphore which is used to acquire and release the lock
- 2.**empty**,a counting semaphore whose initial value is the number of slots in the buffer ,since initially all slots are empty
- 3.**full**, a counting semaphore whose initial value is 0

Producer

```
do {
  wait (empty); // wait until empty>0
                   and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
                   then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

Producer & Consumer

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

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

```
register1 = counter
register1 = register1 + 1
counter = register1
```

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

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- 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}

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
- ***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**
  - General structure of process P_i

```
do {
      entry section
          critical section
      exit section
          remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **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
3. **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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Algorithm 1 for Process P_i

```
do {

    while (turn == j);

        critical section

    turn = j;

        remainder section

 } while (true);
```

Algorithm 2 -Peterson’s Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language 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!

Algorithm 2 -Peterson's Solution

```
do {
      flag[i] = true;
      turn = j;
      while (flag[j] && turn = = j);
               critical section
      flag[i] = false;
               remainder section
 } while (true);
```

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ enters CS only if:

     either **flag[j] = false** or **turn = i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below 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
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

Test_and_set Instruction

Definition **boolean test_and_set (boolean *target)**

```
{
   boolean rv = *target;
   *target = TRUE;
   return rv:
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

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

Semaphore

- 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
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**
- Definition of the **wait() operation**

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

- Definition of the **signal() operation**

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

Semaphore Usage

- **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:

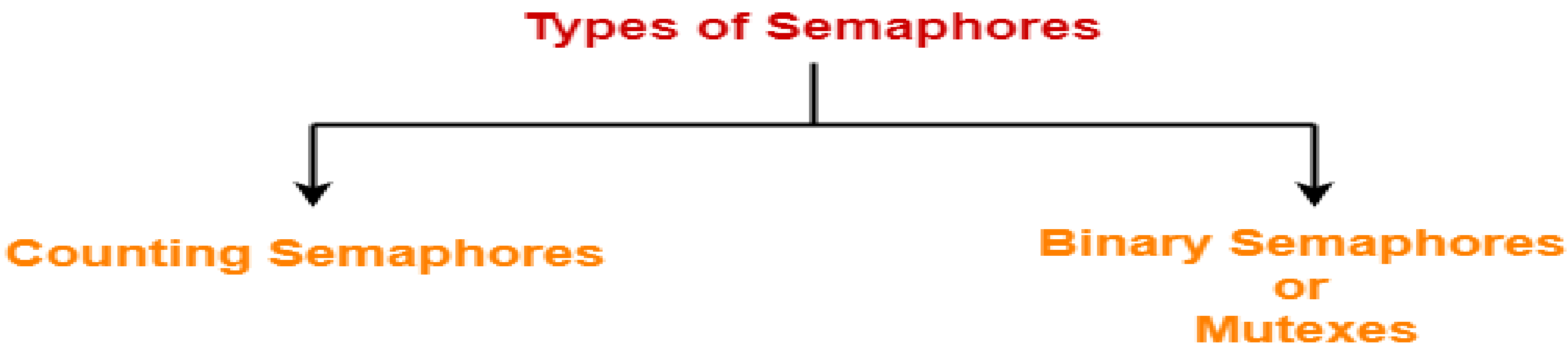
  wait(synch);

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

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But 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

Semaphores



- The main disadvantage of the semaphore – ***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
- Busy waiting wastes CPU cycles that some other process might be to use productively
- This type of semaphore is also called a ***spinlock*** because the process spins while waiting for the lock

Semaphores

To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations.

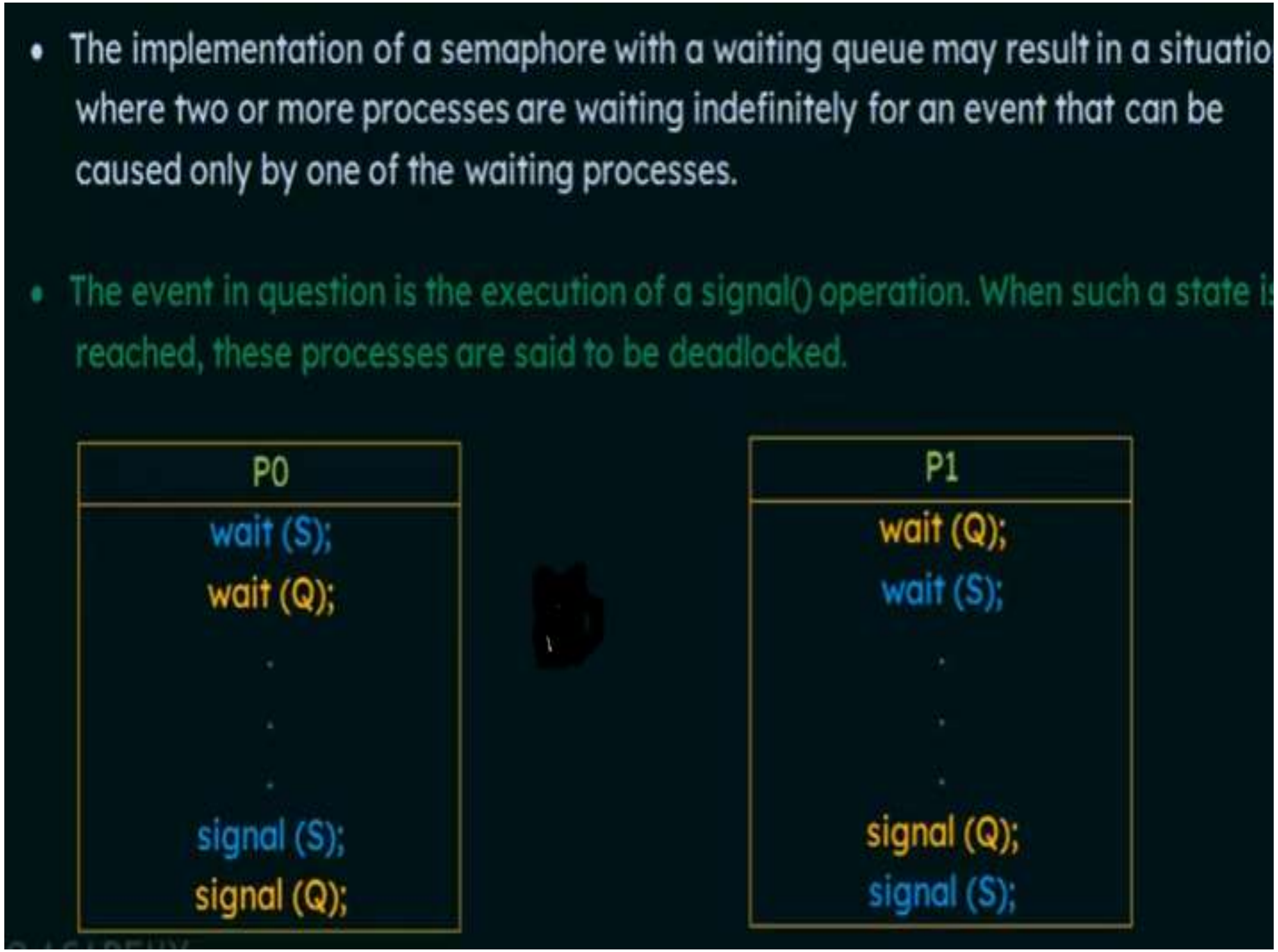
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.

- However, rather than engaging in busy waiting, the process can block itself.

- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.

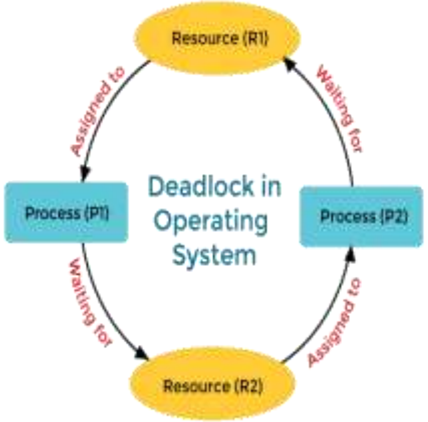
- Then control is transferred to the CPU scheduler, which selects another process to execute.

Disadvantages of Semaphores-Dead lock and Starvation



- The implementation of a semaphore with a waiting queue may result in a situatio where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- The event in question is the execution of a signal() operation. When such a state i reached, these processes are said to be deadlocked.

P0	P1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);



- ➢Deadlock and starvation are conditions in which the processes requesting a resource have been delayed for a long time.
- ➢Deadlock happens when every process holds a resource and waits for another process to hold another resource.
- ➢In contrast, in starvation, the processes with high priorities continuously consume resources, preventing low priority processes from acquiring resources.

Hardware Based Solution-Test and Set Lock

Atomic Operation

All are happened in Single operation without any Interrupt

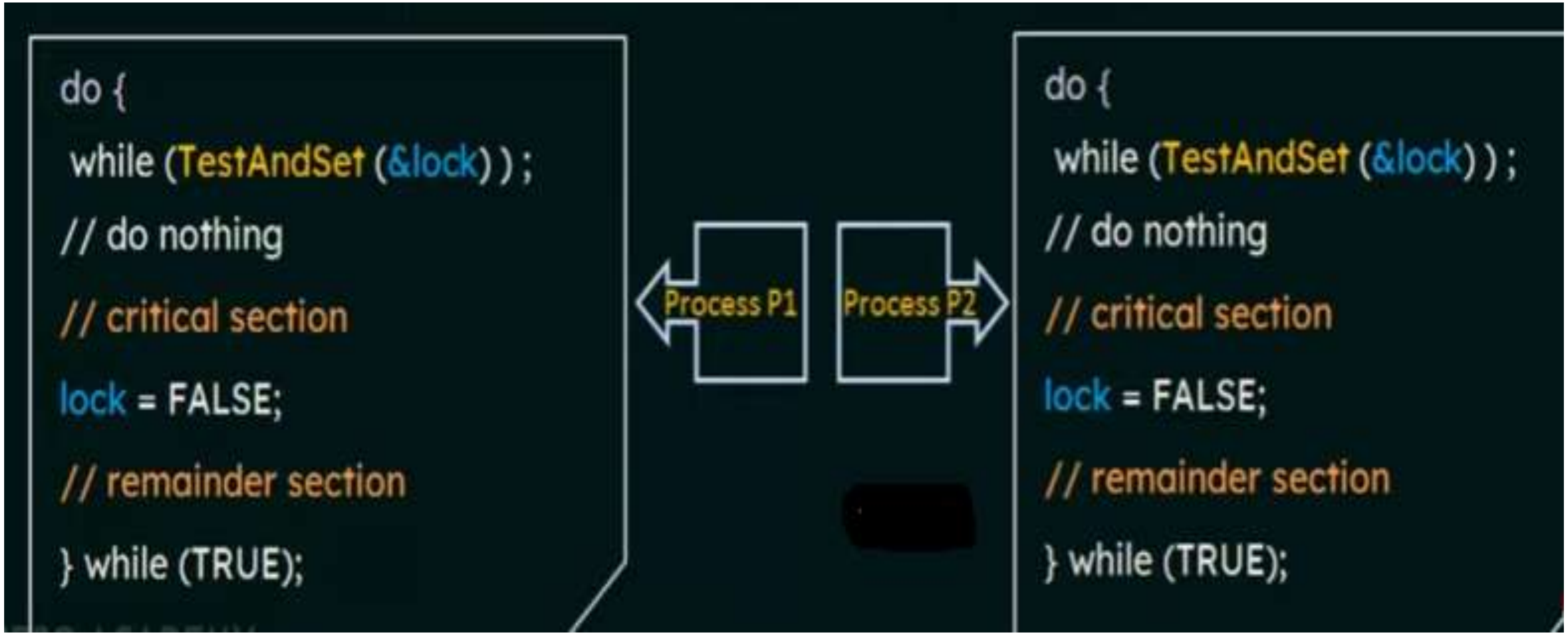
- A hardware solution to the critical section Problem
- There is a shared variable which can take either of the two values 0 or 1
- Before Entering into the critical section a process inquires about the lock
- If it is locked it keeps on waiting till it becomes free
- If it is not locked it takes the lock and executes the critical section

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

The definition of the TestAndSet () instruction

Lock value set to 0-If it is Unlocked
Lock Value set to 1-It is locked

Hardware Based Solution-Test and Set Lock



Satisfies Mutual Exclusion

Does Not Satisfy Bounded Waiting

Classical Problems of Synchronization

- ***Bounded Buffer Problem***
- ***Readers writers Problem***
- ***Dinning Philosophers Problem***

The Bounded Buffer Problem

The Bounded Buffer Problem(Producer Consumer) Problem) one of the classic problems of Synchronization

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, **Producer** and **Consumer**, which are operating on the buffer.

- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

Solution to the Bounded Buffer Problem

- We will make use of three Semaphores
- 1.**m(mutex),**a binary semaphore which is used to acquire and release the lock
- 2.**empty**,a counting semaphore whose initial value is the number of slots in the buffer ,since initially all slots are empty
- 3.**full**, a counting semaphore whose initial value is 0

Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
               then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

2.Readers Writers Problem

The readers-writers problem is a classical problem of process synchronization,
It relates to a data set such as a file that is shared between more than one process at a time.
Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets.

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

Solution to the Readers Writers Problem Using Semaphores

We will make use of two Semaphores and an Integer Variable

- 1.**Mutex** , a semaphore(initialized to 1) which is used to ensure Mutual exclusion when read count is Updated i.e when any Reader enters or exit from the critical section
- 2.**wrt** ,a semaphore (initialized to 1) common to both reader and writer processes
- 3.**readcount** ,an integer variable(initialized to 0) that keeps track of how many processes are currently reading the object

Solution to the Readers Writers Problem Using Semaphores

Writer Process

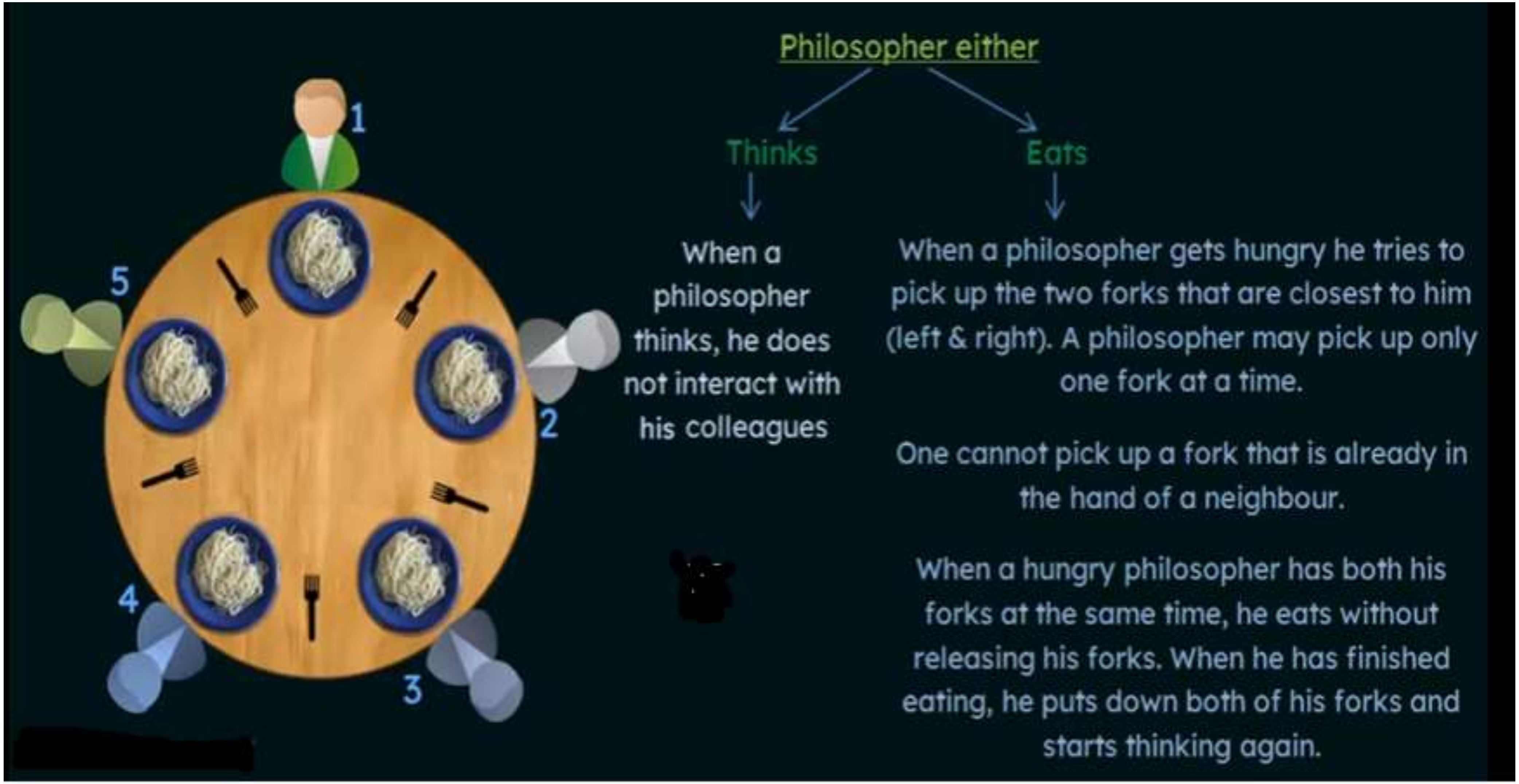
```
do {
/* writer requests for critical
section */
    wait(wrt);
   /* performs the write  */
    // leaves the critical section
    signal(wrt);
} while(true);
```

Reader Process

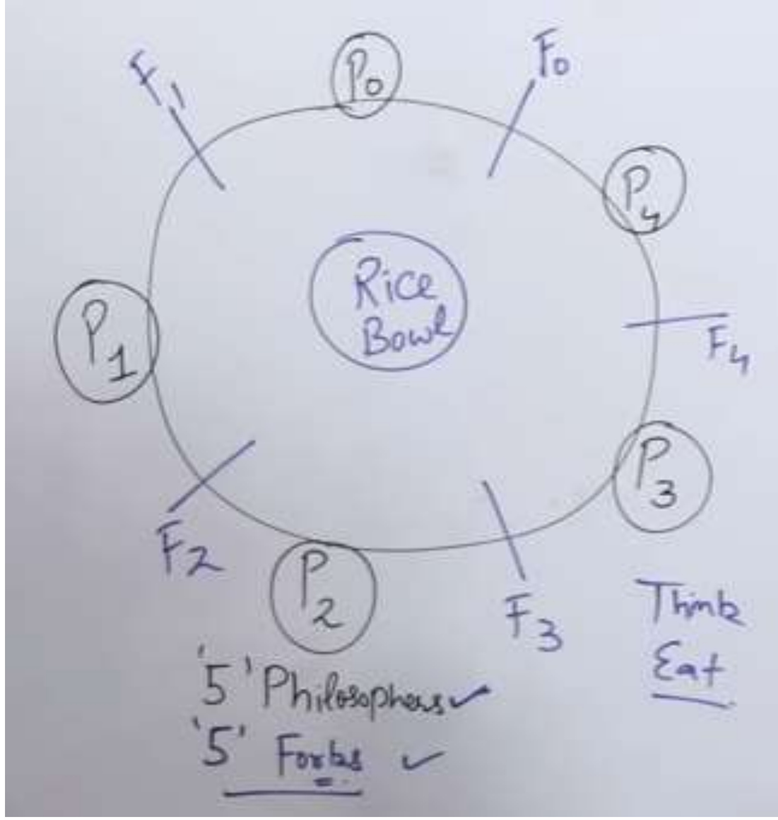
```
do {
  wait (mutex);
 readcnt++;  // The number of readers has now increased by 1
 if (readcnt==1)
     wait (wrt); // this ensure no writer can enter if there is even one reader
   signal (mutex);   // other readers can enter while this current reader is
                          inside the critical section
 /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)      //no reader is left in the critical section
      signal (wrt);        // writers can enter
      signal (mutex);  // reader leaves
} while(true);
```

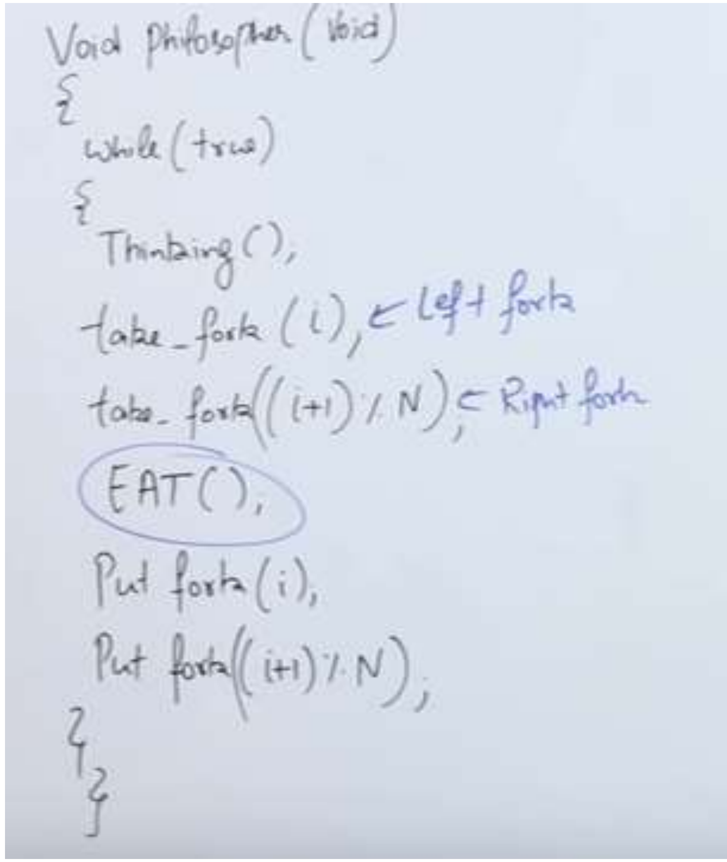
NESO ACADEMY

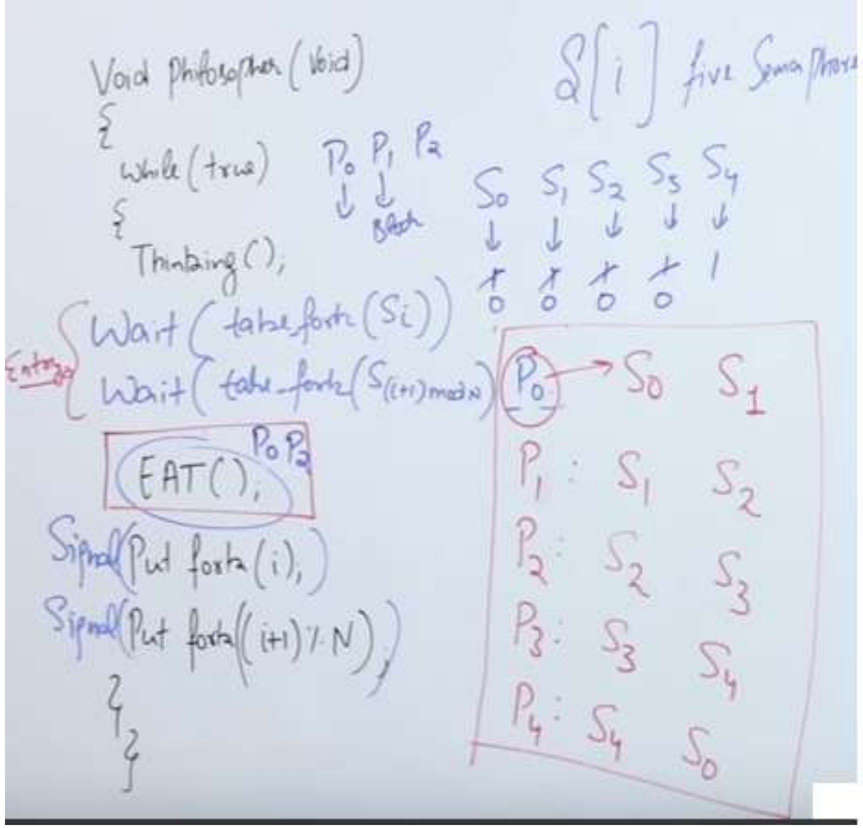
3.The Dinning Philosophers Problem



3.The Dinning Philosophers Problem







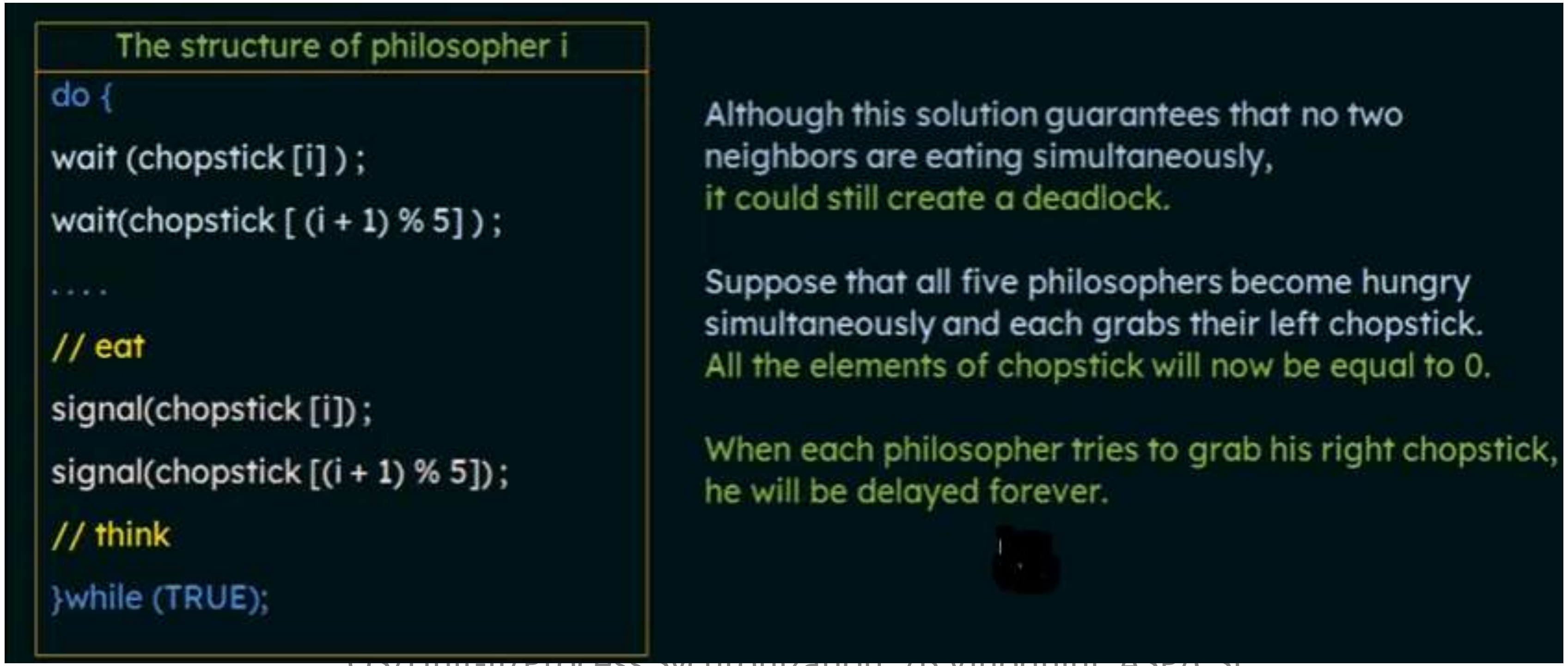
3.The Dinning Philosophers Problem

One simple solution to represent each fork/chopstick with a semaphore

A Philosopher tries to grab a fork/chopstick by Executing a wait() operation on that semaphore

He releases his fork /chopsticks by executing the signal() operation on the appropriate semaphores

Thus the shared data are **semaphore chopstick[5];**



3.The Dinning Philosophers Problem

Possible remedies to avoid Deadlock

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available (to do this he must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

References

1. Silberschatz, Galvin, and Gagne, “Operating System Concepts”, Ninth Edition, Wiley India Pvt Ltd, 2009.
2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition, Pearson Education, 2010.


Thank you