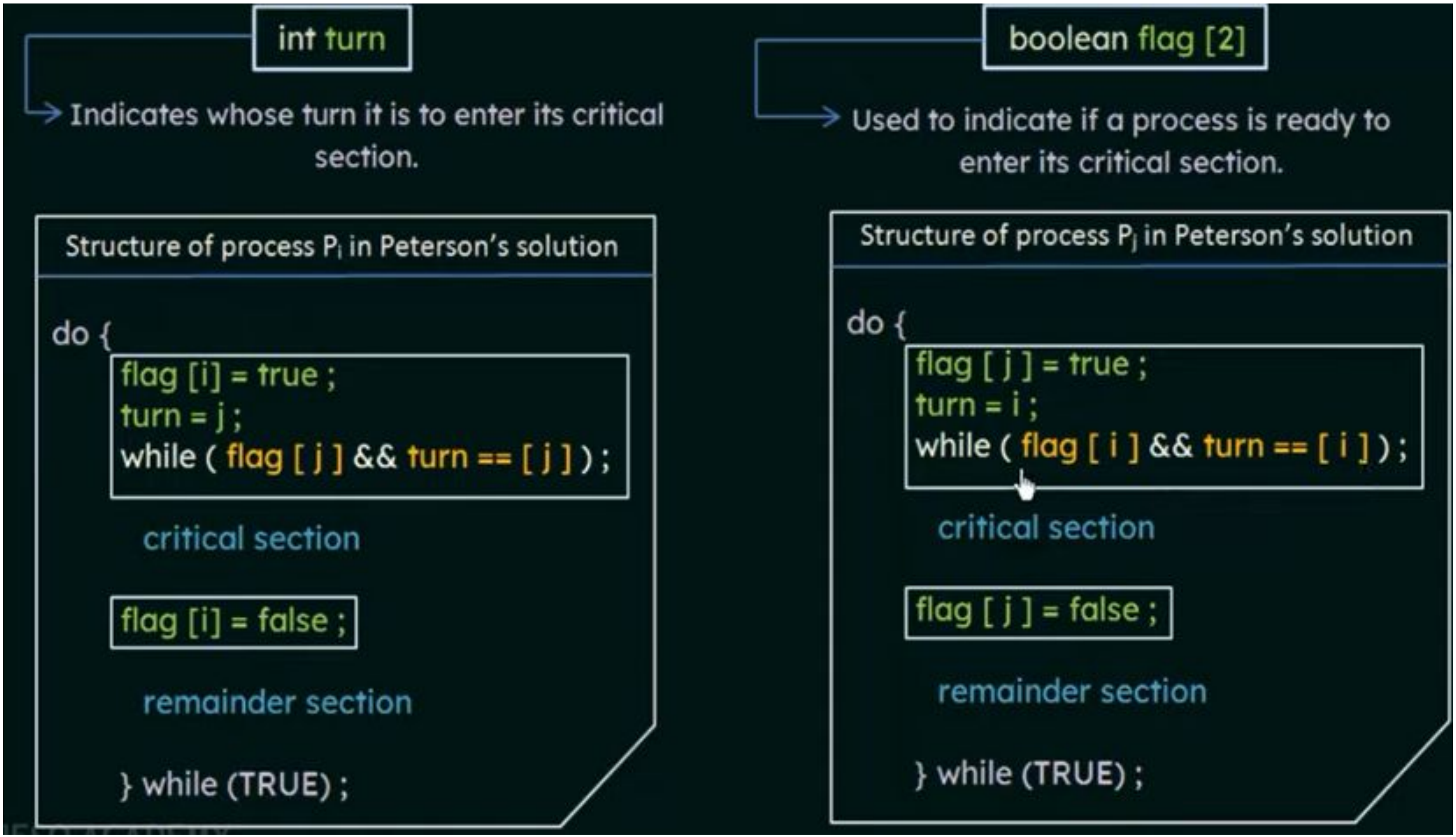
Process Synchronization

Unit 3

Peterson's Solution



Synchronization Hardware-Test and Set

- A hardware solution to the synchronization problem.
- There is a shared lock 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;
}
```

Atomic Operation

The definition of the TestAndSet () instruction

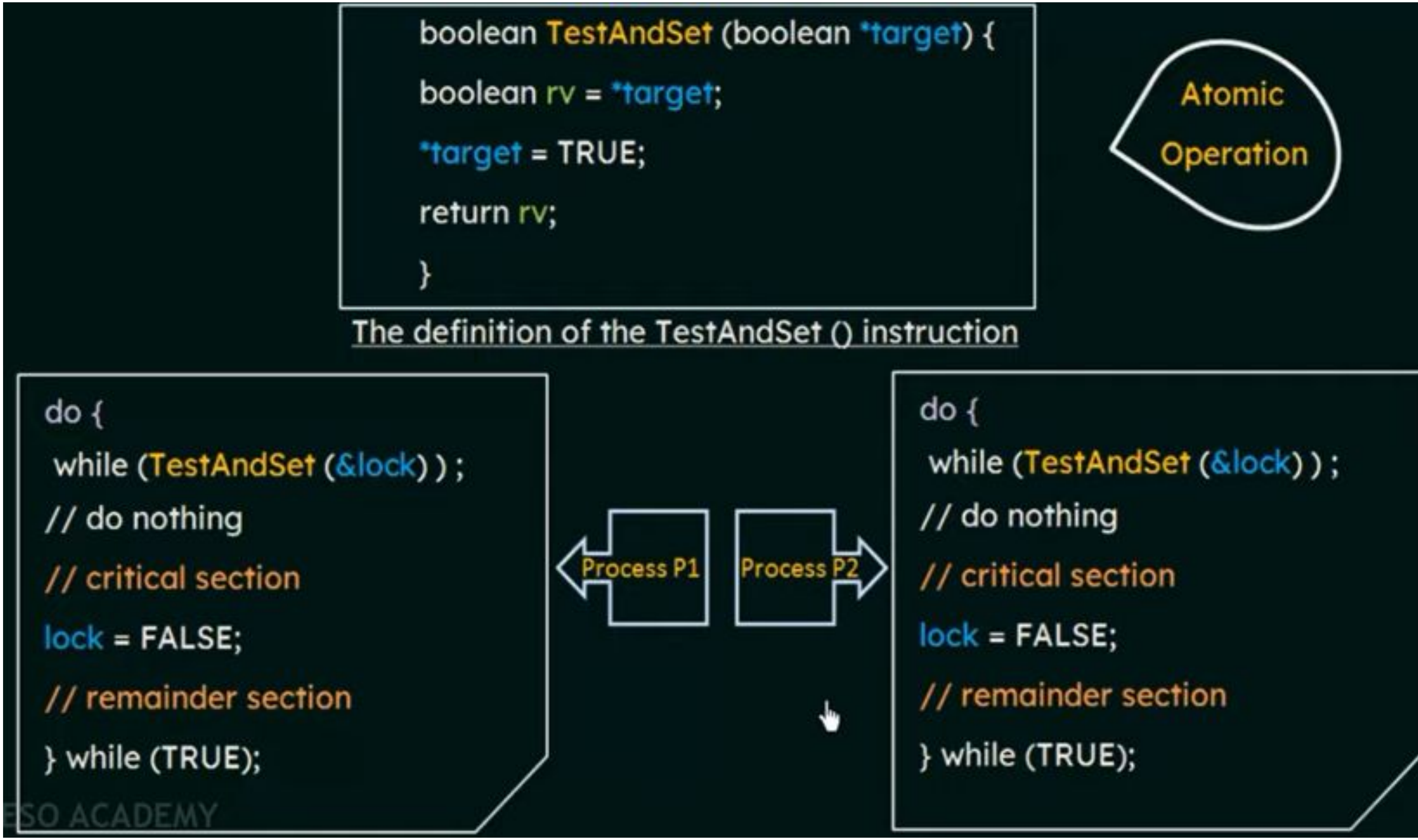
```
do {
    acquire lock
            critical section
    release lock
            remainder section
} while (TRUE);
```

Satisfies mutual-exclusion.

Does not satisfy bounded-waiting.

Satisfies Progress

Synchronization Hardware-Test and Set



Compare and Swap

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

    if (*value == expected)
       *value = new_value;

    return temp;
}
```

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

     /* critical section */

  lock = 0;

     /* remainder section */
} while (true);
```

- **Satisfies Mutual Exclusion**
- **Satisfies Progress**
- **Does not Satisfy Bounded Wait**

Bounded wait solution with TestAndSet

Entry section

```
waiting[i] = true;
key = true;
while (waiting[i] && key)
key = test and set(&lock);
waiting[i] = false;
```

lock

F

	P0	P1	P2	P3	P4	. . . . .	Pn
Waiting [i]	F	F	F	F	F		F
Key							

Critical Section

	P0	P1	P2	P3
Waiting [i]	F	F	F	F
Key				

Exit section

```
j = (i + 1) % n;
while ((j != i) && !waiting[j])
j = (j + 1) % n;
if (j == i)
lock = false;
 else
waiting[j] = false;
```

TestAndSet

if lock=false, return false,lock=true

if lock=true,return true,lock=true

Bounded Buffer Problem

- 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.

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.

Bounded Buffer Problem

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)
```

Readers and Writers Problem

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as Readers and to the latter as Writers.
- Obviously, if two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.

This synchronization problem is referred to as the readers-writers problem.

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 readcount 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.

Readers and Writers Problem

Writer Process	Reader Process

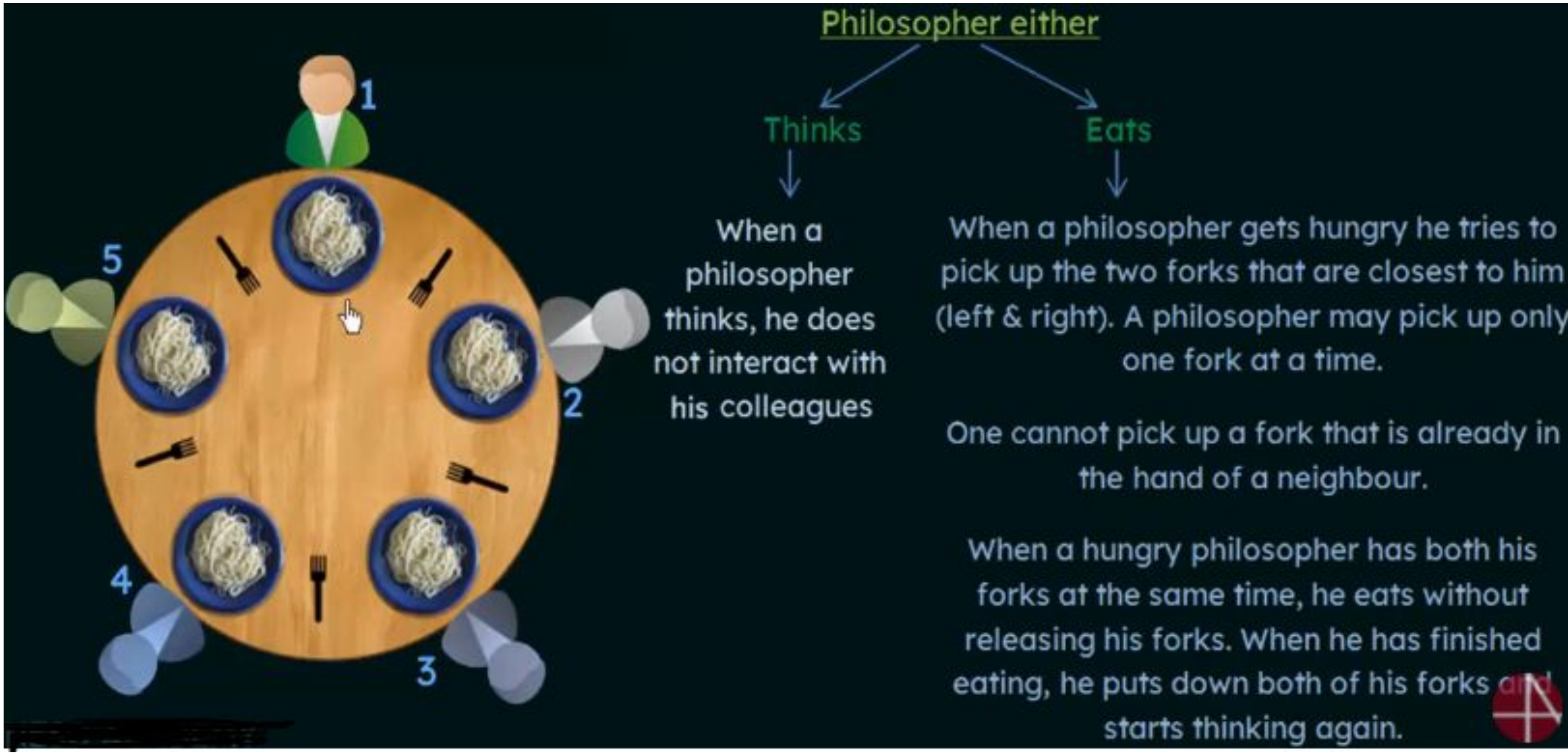
Writer Process

```
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);
```

Dining Philosopher Problem



Dining Philosopher Problem

where all the elements of chopstick are initialized to 1.

The structure of philosopher i

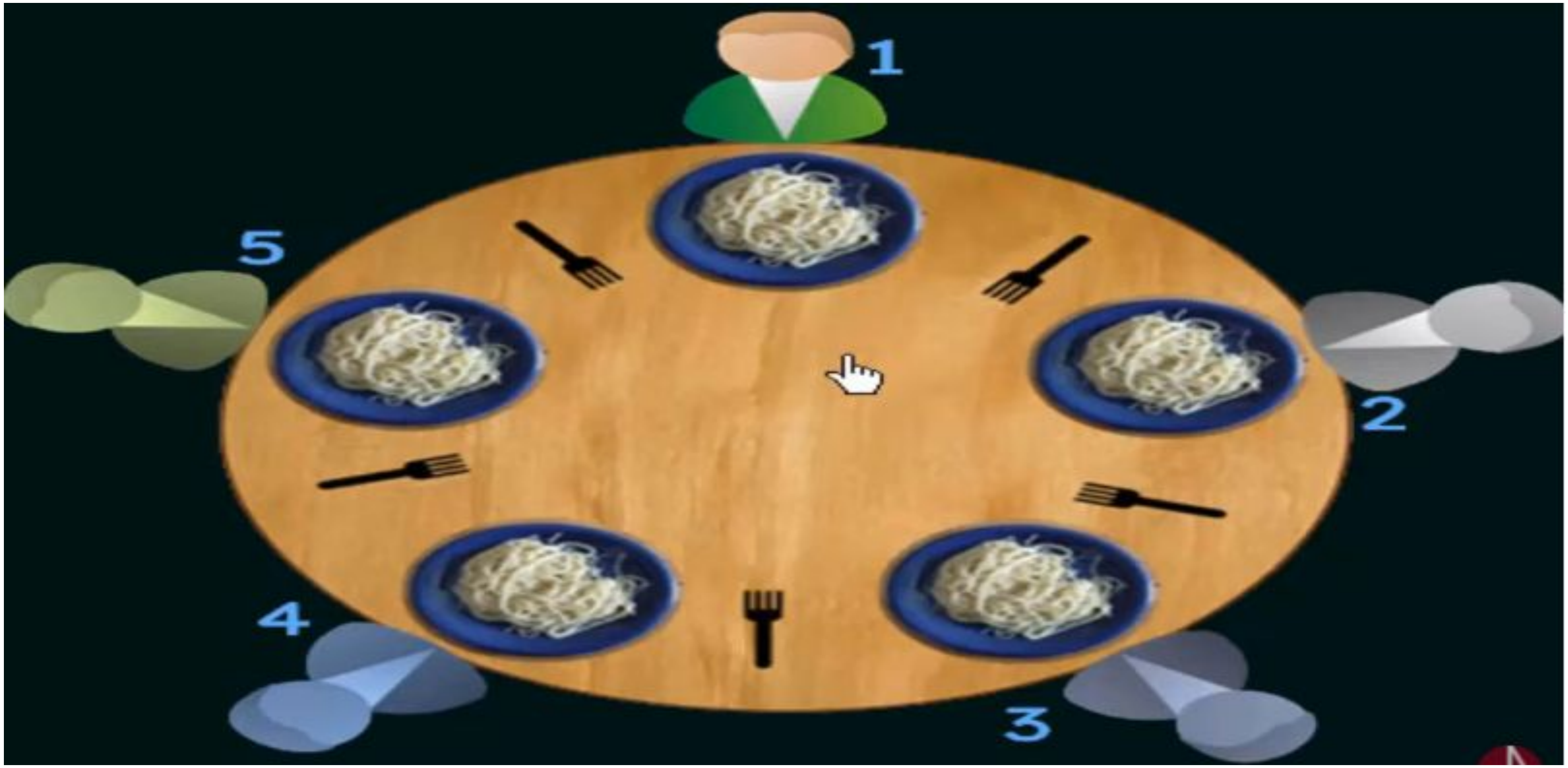
```
do {
wait (chopstick [i] ) ;
wait(chopstick [ (i + 1) % 5] ) ;
. . . .
// eat
signal(chopstick [i]) ;
signal(chopstick [(i + 1) % 5]) ;
// think
}while (TRUE);
```

Although this solution guarantees that no two neighbors are eating simultaneously,
it could still create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick.
All the elements of chopstick will now be equal to 0.

When each philosopher tries to grab his right chopstick,
he will be delayed forever.

Dining Philosopher Problem

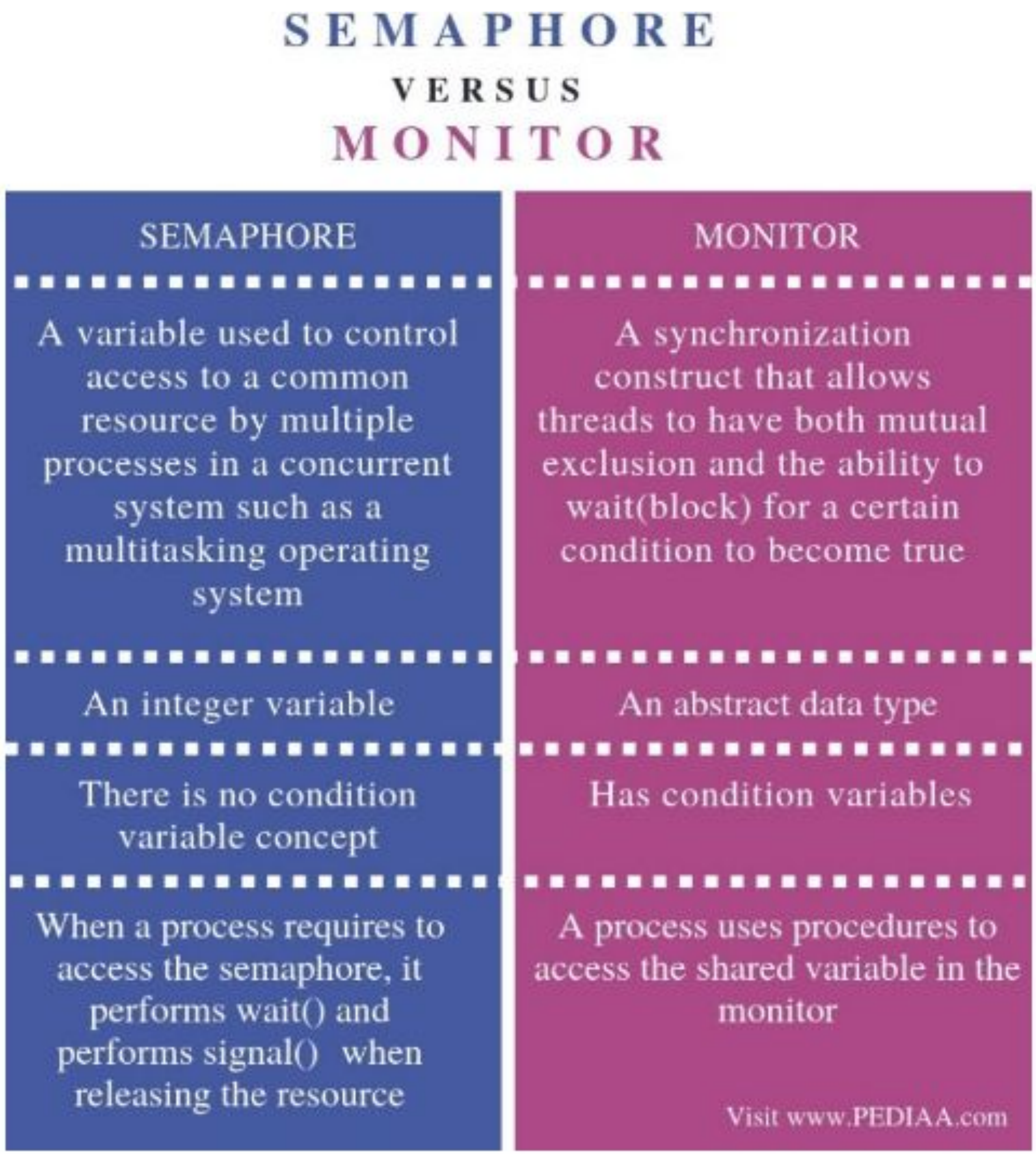


Dining Philosopher Problem

Some possible remedies to avoid deadlocks:

- 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.

Monitor



SEMAPHORE

VERSUS

MONITOR

SEMAPHORE	MONITOR
A variable used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system	A synchronization construct that allows threads to have both mutual exclusion and the ability to wait(block) for a certain condition to become true
An integer variable	An abstract data type
There is no condition variable concept	Has condition variables
When a process requires to access the semaphore, it performs wait() and performs signal() when releasing the resource	A process uses procedures to access the shared variable in the monitor

Visit www.PEDIAA.com

Monitor

Syntax of a Monitor

```
monitor monitor_name
{
// shared variable declarations
procedure P1 (. . .) {
. . .
}
procedure P2 (. . .) {
. . .
}
.
.
procedure Pn (. . .) {
. . .
}
initialization code (. . .) {
. . .
}
}
```

- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

<u>Condition Construct</u>- condition x, y ;

The only operations that can be invoked on a condition variable are wait ( ) and signal ( ).

The operation x.wait( ) ; means that the process invoking this operation is suspended until another process invokes x.signal ( );

The x. signal ( ) operation resumes exactly one suspended process.

https://www.youtube.com/watch?v=ufdQ0GR855M

Monitor

Syntax of a Monitor

```
monitor monitor_name
{
// shared variable declarations
procedure P1 ( . . . ) {
. . .
}
procedure P2 ( . . . ) {
. . .
}
.
.
procedure Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```

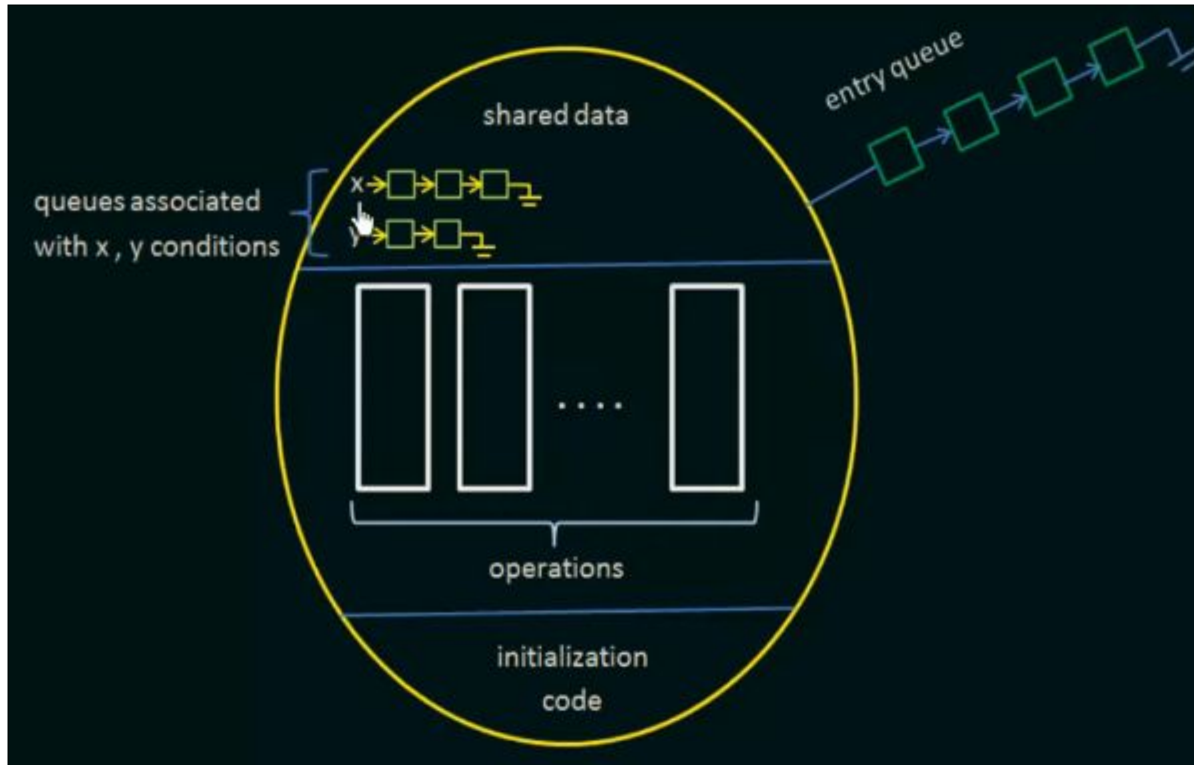


Fig: Schematic view of a monitor

Monitor

A monitor solution to the dining-philosopher problem

```
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 ((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 ;
        }
}
```