Kaushik

__/__/____

Operating Systems Assignment - 1

By - S Kaushik Rao
23FE10CA100526

1Qn: Dining philosophers problem, is a classic problem in Concurrent programming, first proposed by Edsger
It's just 5 philosophers sitting at a round table. Alternating between thinking & Eating. To Eat, a philosopher needs two forks, one from left & right. But there are only 5 forks available.

What is really the deadlock?
if Each philosopher picks up the fork on the left and waits for one on the right, no one can proceed, leading to deadlock.

→ To prevent it, We Can Use → Resource Hierarchy, Chandy/Misra, Assymetric Solution.

i) Resource Hierarchy → Assign a Unique number to Each fork. pick up forks in strict Order of Number. prevents Circular wait.

ii) Asymmetric : Break Symmetry, atleast One philosopher finishes eating.

Using Semaphores.

→ Controlling the Access of forks.

→ Each fork is represented by a Semaphore initialized to 1, saying it's available.

→ Even number pick up the left fork first, odd, pick right

→ put fork down -> after Eating, The reverse then.

```
NUM_Philosophers = 5
forks = [Semaphore[1] for _ in range (Num_Philosophers)]

def Philosopher (id):
    while True:
        think()
        pick_up_forks (id)
def pick_up_forks (id):
        left_fork = id
        right_fork = (id+1) %. NUM_PHILOSOPHERS
```

→ # To Prevent Deadlock, Specific order

```
if id %.2 == 0
  forks [left_for]. wait()
else.
  forks [left_fork]. wait()
def think():
    pass
def eat ()
    pass;
# for P in philosophers:
    P. start().
```

__/__/____

Ans 2.) Reader-Writer problem is a classic Synchronization Issue in Concurrent programming where multiple readers and writers access a shared resource.

Synchronization Issues:

→ Data Inconsistency: (First Readers-Writer Problem) Readers are given priority. If a reader is access the resource, other readers can also access it Concurrently.

→ Writer Preference: (Second Reader-writer) it gets exclusive to the resource. Reads are only allowed to access the resource when no writers are waiting.

→ Using Semaphores to Manage Access.

```
# From threadpy import Semaphore, Thread.
#Initialize Semaphore
read_count = 0
read_mutex = Semaphore(1)

def reader(id):
    global read_Count
    While True;
    read_mutex.acquire()
        read_count += 1
        if read_Count == 1;
        read_mutex.release
```

```
read - resource ()
read - mutex . acquire ()
read - count -= 1
    if read - count == 0;
read - mutex . release ()

def writer (id);
    while True :
        resource
        resource-mutex - release ().

def read - resource ):
def write - resource ):

readers = [Thread (target [target = reader, args = (i,)])
writers = [Thread (target = write, args = (i, 0), for i in rg]

# Start reader & writer threads
    for r in readers ;
        r . start ()
    for w in writer ;
        w . join ()

# for reader and writer threades
    for r in readers;
        r . join ()
    for w in writers :
        w . join ().
```