Bansilal Ramnath Agarwal Charitable Trust's
Vishwakarma Institute of Technology
(An Autonomous Institute affiliated to Savitribai Phule Pune University)

Operating Systems Lab

Assignment No: 5

Name : Gourav Balaji Suram

Roll No : 39

PRN No : 12220032

1. Dining Philosopher Problem.

The dining philosopher's problem is a version of the classical synchronization problem, in which five philosophers sit around a circular table and alternate between thinking and eating. A bowl of noodles and five forks for each philosopher are placed at the centre of the table.

There are certain conditions a philosopher must follow:

A philosopher must use both their right and left forks to eat.

A philosopher can only eat if both of his or her immediate left and right forks are available. If the philosopher's immediate left and right forks are not available, the philosopher places their (either left or right) forks on the table and resumes thinking.

(Note: There are many variations of this problem, such as that instead of a fork, a chopstick can be used, and instead of noodles, a rice bowl can be used, but the logic and solution of the problem will be the same.)

Code :

```python
import threading
import time

philosophers = [i for i in range(2)]
forks = [threading.Semaphore(1) for i in range(2)]

def philosopher(id):
    for i in range(4):
        print(f"Philosopher {id} is thinking...")
        time.sleep(1)

        left_fork = id
        right_fork = (id + 1) % 2
        if id % 2 == 0:
            forks[left_fork].acquire()
            forks[right_fork].acquire()
        else:
            forks[right_fork].acquire()
            forks[left_fork].acquire()

        print(f"\nPhilosopher {id} is eating...")
        time.sleep(1)

        forks[left_fork].release()
        forks[right_fork].release()

if __name__ == '__main__':
    philosopher_threads = [threading.Thread(target=philosopher, args=(i,)) for i in philosophers]

    for thread in philosopher_threads:
        thread.start()

    for thread in philosopher_threads:
        thread.join()
```

Output :

```
~/vit-comp/Module-4/Operating-System/Assignment-5/python-prgms on main !3 ?5
❯ python3 dining-philosopher.py
Philosopher 0 is thinking...
Philosopher 1 is thinking...

Philosopher 0 is eating...
Philosopher 0 is thinking...

Philosopher 1 is eating...
Philosopher 1 is thinking...

Philosopher 0 is eating...
Philosopher 0 is thinking...

Philosopher 1 is eating...
Philosopher 1 is thinking...

Philosopher 0 is eating...
Philosopher 0 is thinking...

Philosopher 1 is eating...
Philosopher 1 is thinking...

Philosopher 0 is eating...

Philosopher 1 is eating...
```

2. Producer Consumer Problem

In the producer-consumer problem, there is one Producer who produces things, and there is one Consumer who consumes the products which are produced by the producer. The producers and consumers share the same fixed-size memory buffer.

The Producer's role is to produce data, store it in the buffer, and then generate data again. The Consumer's task is to consume the data from the buffer.

The Producer-Consumer problem is a classic multi-process synchronization problem, which implies we're aiming to synchronize many processes.

When the consumer is consuming an item from the buffer, the producer should not add items into the buffer, and vice versa. As a result, only one producer or consumer should access the buffer at a time.

Code :

```
import threading
import time
import random

queue = []
queue_max_size = 10
queue_mutex = threading.Semaphore(1)
queue_not_empty = threading.Semaphore(0)
queue_not_full = threading.Semaphore(queue_max_size)

def producer(id):
    for i in range(4):
        item = random.randint(1, 100)
        queue_not_full.acquire()
        queue_mutex.acquire()
        queue.append(item)
        queue_mutex.release()
        queue_not_empty.release()
        print(f"Producer {id} added item {item}. Queue size: {len(queue)}")
        time.sleep(1)

def consumer(id):
    for i in range(4):
        queue_not_empty.acquire()
        queue_mutex.acquire()
        item = queue.pop(0)
        queue_mutex.release()
        queue_not_full.release()
        print(f"Consumer {id} consumed item {item}. Queue size: {len(queue)} \n")
        time.sleep(1)


if __name__ == '__main__':
    producer1 = threading.Thread(target=producer, args=(1,))
    producer2 = threading.Thread(target=producer, args=(2,))
    consumer1 = threading.Thread(target=consumer, args=(1,))
    consumer2 = threading.Thread(target=consumer, args=(2,))

    producer1.start()
    producer2.start()
    consumer1.start()
    consumer2.start()

    producer1.join()
    producer2.join()
    consumer1.join()
    consumer2.join()
```

Output :

```
~/vit-comp/Module-4/Operating-System/Assignment-5/python-prgms on main !3 ?5
❯ python3 producer-consumer.py
Producer 1 added item 73. Queue size: 1
Producer 2 added item 82. Queue size: 2
Consumer 1 consumed item 73. Queue size: 1

Consumer 2 consumed item 82. Queue size: 0

Producer 2 added item 71. Queue size: 1
Producer 1 added item 9. Queue size: 2
Consumer 1 consumed item 71. Queue size: 1

Consumer 2 consumed item 9. Queue size: 0

Producer 2 added item 73. Queue size: 1
Producer 1 added item 78. Queue size: 2
Consumer 1 consumed item 73. Queue size: 1

Consumer 2 consumed item 78. Queue size: 0

Producer 2 added item 41. Queue size: 1
Producer 1 added item 62. Queue size: 2
Consumer 1 consumed item 41. Queue size: 1

Consumer 2 consumed item 62. Queue size: 0
```

3. Readers-Writers Problem.

A database must be shared by numerous concurrent processes, some of which may simply want to read the database, while others may wish to update (read and write) the database.

We differentiate between these two processes by referring to the former as Readers and the latter as Writers.

There will be no negative consequences if two readers access the shared data simultaneously.

If a writer and another thread (either a reader or a writer) access the common data simultaneously, chaos may follow.

To avoid these problems, we need that the writers have exclusive access to the shared database.

This synchronization issue is known as the readers-writers problem.

Parameters of the problem:

- Several processes share a single set of data.
- When a writer is ready, it begins writing.
- At any given moment, only one writer may work.
- No other process can read what a process is writing.
- No other process can write if at least one reader is reading.
- Readers may not write and must rely only on what they read.

Code :

```python
import threading
import time

resource_lock = threading.Semaphore(1)
readers_counter = 0
readers_counter_lock = threading.Lock()
resource = "This is a shared resource"

def reader():
    global readers_counter
    with readers_counter_lock:
        readers_counter += 1
        if readers_counter == 1:
            resource_lock.acquire()

    print(f"Reader is reading the resource: {resource}")
    time.sleep(1)

    with readers_counter_lock:
        readers_counter -= 1
        if readers_counter == 0:
            resource_lock.release()

def writer():
    resource_lock.acquire()
    resource = "This is a shared resource, updated by the writer"
    print(f"Writer is writing to the resource: {resource}")
    time.sleep(1)
    resource_lock.release()

readers = [threading.Thread(target=reader) for i in range(2)]
writers = [threading.Thread(target=writer) for i in range(2)]

if __name__ == '__main__':
    for reader in readers:
        reader.start()
    for writer in writers:
        writer.start()

    for reader in readers:
        reader.join()
    for writer in writers:
        writer.join()
```

Output :

```
~/vit-comp/Module-4/Operating-System/Assignment-5/python-prgms on main !3 ?5
❯ python3 reader-writer.py
Reader is reading the resource: This is a shared resource
Reader is reading the resource: This is a shared resource
Writer is writing to the resource: This is a shared resource, updated by the writer
Writer is writing to the resource: This is a shared resource, updated by the writer

~/vit-comp/Module-4/Operating-System/Assignment-5/python-prgms on main !3 ?5
❯
```