Operating Systems Lab
Fall 2024-25(L59+60)

Student Name:- Parth Suri
Registration No:- 22BDS0116

Class No. :- VL2024250102445

Faculty Name:- Rahul Srivastava

Experiment 1: Dining Philosopher Problem

Objective: To implement the Dining Philosopher problem using binary semaphores to avoid deadlock and starvation.

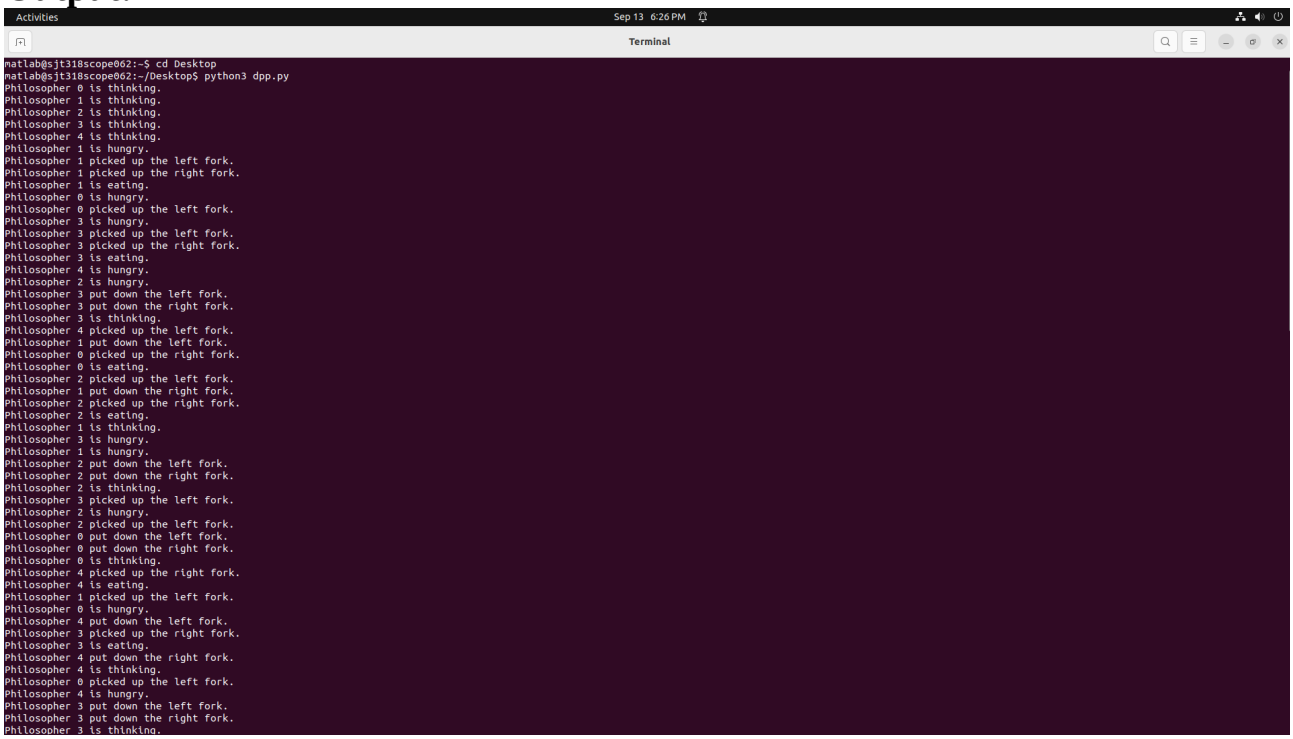
Algorithm:
1. There are N philosophers seated at a round table with one fork between each pair.
2. Each philosopher must alternately think and eat.
3. A philosopher needs both the fork on their left and the fork on their right to eat.
4. Use binary semaphores to control access to the forks and avoid deadlock.
5. Ensure that no two neighboring philosophers can eat at the same time.

Steps:
1. Initialize N forks (binary semaphores) and philosophers.
2. Each philosopher performs the following in a loop:
o Wait (think for some time). o Attempt to pick up the left fork (semaphore wait).
o Attempt to pick up the right fork (semaphore wait). o Eat (for a specific time).
o Put down both forks (semaphore signal).
3. Ensure that the binary semaphores prevent simultaneous access to the same fork.

Code:-

```
import threading
import time
import random
class Philosopher(threading.Thread):
    def __init__(self, index, left_fork, right_fork):
        threading.Thread.__init__(self)
        self.index = index
        self.left_fork = left_fork
        self.right_fork = right_fork
    def run(self):
        while True:
            print(f"Philosopher {self.index} is thinking.")
            time.sleep(random.uniform(1, 3)) # Simulating thinking
            print(f"Philosopher {self.index} is hungry.")
            # Pick up left and right forks (binary semaphores)
            self.left_fork.acquire()
            print(f"Philosopher {self.index} picked up the left fork.")
            self.right_fork.acquire()
            print(f"Philosopher {self.index} picked up the right fork.")
            print(f"Philosopher {self.index} is eating.")
            time.sleep(random.uniform(3, 13)) # Simulating eating
            # Release forks after eating
            self.left_fork.release()
            print(f"Philosopher {self.index} put down the left fork.")
            self.right_fork.release()
            print(f"Philosopher {self.index} put down the right fork.")
# Number of philosophers
num_philosophers = 5
# Forks are represented by semaphores
forks = [threading.Semaphore(1) for _ in range(num_philosophers)]
# Creating philosopher threads
philosophers = []
for i in range(num_philosophers):
    left_fork = forks[i]
    right_fork = forks[(i + 1) % num_philosophers]
    philosopher = Philosopher(i, left_fork, right_fork)
    philosophers.append(philosopher)
# Start the philosopher threads
for philosopher in philosophers:
    philosopher.start()
```

Output:-


Activities
Sep 13 6:26 PM
Terminal
matlab@sjt318scope062:~$ cd Desktop
matlab@sjt318scope062:~/Desktop$ python3 dpp.py
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is hungry.
Philosopher 1 picked up the left fork.
Philosopher 1 picked up the right fork.
Philosopher 1 is eating.
Philosopher 0 is hungry.
Philosopher 0 picked up the left fork.
Philosopher 3 is hungry.
Philosopher 3 picked up the left fork.
Philosopher 3 picked up the right fork.
Philosopher 3 is eating.
Philosopher 4 is hungry.
Philosopher 2 is hungry.
Philosopher 3 put down the left fork.
Philosopher 3 put down the right fork.
Philosopher 3 is thinking.
Philosopher 4 picked up the left fork.
Philosopher 1 put down the left fork.
Philosopher 0 picked up the right fork.
Philosopher 0 is eating.
Philosopher 2 picked up the left fork.
Philosopher 1 put down the right fork.
Philosopher 2 picked up the right fork.
Philosopher 2 is eating.
Philosopher 1 is thinking.
Philosopher 3 is hungry.
Philosopher 1 is hungry.
Philosopher 2 put down the left fork.
Philosopher 2 put down the right fork.
Philosopher 2 is thinking.
Philosopher 3 picked up the left fork.
Philosopher 2 is hungry.
Philosopher 2 picked up the left fork.
Philosopher 0 put down the left fork.
Philosopher 0 put down the right fork.
Philosopher 0 is thinking.
Philosopher 4 picked up the right fork.
Philosopher 4 is eating.
Philosopher 1 picked up the left fork.
Philosopher 0 is hungry.
Philosopher 4 put down the left fork.
Philosopher 3 picked up the right fork.
Philosopher 3 is eating.
Philosopher 4 put down the right fork.
Philosopher 4 is thinking.
Philosopher 0 picked up the left fork.
Philosopher 4 is hungry.
Philosopher 3 put down the left fork.
Philosopher 3 put down the right fork.
Philosopher 3 is thinking.

Experiment 2: Producer-Consumer Problem

Objective: To implement the Producer-Consumer problem using binary semaphores to synchronize access to the buffer between producer and consumer.

Algorithm:
1. A shared buffer is used by producers to place items and consumers to remove items.
2. Two semaphores are used:
o full: to count the number of items in the buffer.
o empty: to count the number of free slots in the buffer.
3. A binary semaphore (mutex) is used to ensure mutual exclusion while accessing the buffer.
4. Producers wait for an empty slot before producing, and consumers wait for an item to consume.

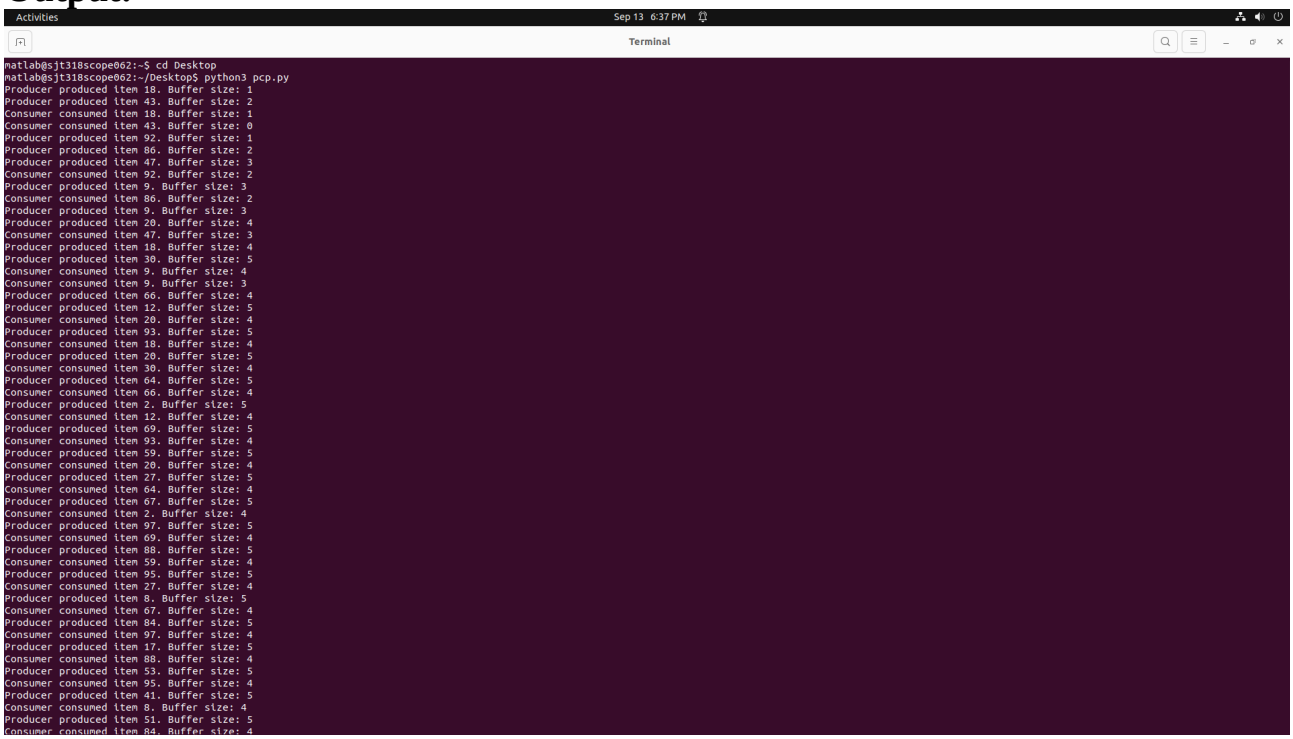
Steps:
1. Initialize buffer, semaphores, and mutex.
2. Producer loop:
o Wait for an empty slot (empty semaphore).
o Acquire the mutex to place an item in the buffer.
o Signal full after producing.
3. Consumer loop:
o Wait for an item (full semaphore).
o Acquire the mutex to remove an item from the buffer. o Signal empty after consuming.

Code:-

```
import threading
import time
import random
import queue
# Buffer size (maximum number of items that can be held in the queue)
BUFFER_SIZE = 5
# Queue to represent the buffer
buffer = queue.Queue(BUFFER_SIZE)
# Semaphore for managing the number of items in the buffer
empty = threading.Semaphore(BUFFER_SIZE) # Semaphore representing empty
slots
full = threading.Semaphore(0) # Semaphore representing filled slots
# Lock to prevent simultaneous access to the buffer
mutex = threading.Lock()
class Producer(threading.Thread):
    def run(self):
        while True:
            item = random.randint(1, 100) # Produce an item
            empty.acquire() # Wait if the buffer is full (no empty slots)
            mutex.acquire() # Lock the buffer access
            # Add the item to the buffer
            buffer.put(item)
            print(f"Producer produced item {item}. Buffer size: {buffer.qsize()}")
            mutex.release() # Unlock the buffer
            full.release() # Signal that there's a filled slot
            time.sleep(random.uniform(1, 3)) # Simulate time taken to produce an item
class Consumer(threading.Thread):
    def run(self):
        while True:
            full.acquire() # Wait if the buffer is empty (no filled slots)
            mutex.acquire() # Lock the buffer access
            # Remove an item from the buffer
            item = buffer.get()
            print(f"Consumer consumed item {item}. Buffer size: {buffer.qsize()}")
            mutex.release() # Unlock the buffer
            empty.release() # Signal that there's an empty slot
            time.sleep(random.uniform(2, 5)) # Simulate time taken to consume an item
# Create and start producer and consumer threads
num_producers = 2
num_consumers = 2
producers = [Producer() for _ in range(num_producers)]
consumers = [Consumer() for _ in range(num_consumers)]
for producer in producers:
    producer.start()
```

```
for consumer in consumers:
    consumer.start()
```

Output:-



Experiment 3: Reader-Writer Problem

Objective: To implement the Reader-Writer problem using semaphores to ensure synchronization between multiple readers and writers. The goal is to allow multiple readers to access the shared resource concurrently but ensure exclusive access for writers.

Algorithm:
1. There is a shared resource (e.g., a database) that both readers and writers need access to.
2. Readers can read the shared resource concurrently, but a writer must have exclusive access to the resource to avoid race conditions.
3. A mutex semaphore is used to control access to the reader count variable.
4. A write_lock semaphore ensures that only one writer or multiple readers access the shared resource at a time.
5. The solution must avoid starvation of either readers or writers.

Steps:
1. Initialize read_count to 0 (tracks how many readers are reading).
2. Initialize the semaphores: o mutex: Binary semaphore to control access to read_count. o write_lock: Binary semaphore to ensure exclusive access for the writer.
3. Reader loop: o Acquire mutex to update read_count. o If it's the first reader, acquire the write_lock to block writers. o Read from the shared resource. o If it's the last reader, release the write_lock to allow writers. 4. Writer loop: o Wait for write_lock to gain exclusive access. o Write to the shared resource. o Release write_lock after writing.

Code:-

```
import threading
import time
import random
# Shared data and lock variables
shared_data = 0 # The resource being read/written
read_count = 0 # Keeps track of the number of readers
# Locks for synchronization
mutex = threading.Lock() # Mutex to protect the read_count
rw_lock = threading.Lock() # Lock for writer access
class Reader(threading.Thread):
    def __init__(self, reader_id):
        threading.Thread.__init__(self)
        self.reader_id = reader_id
    def run(self):
        global read_count
        while True:
            # Reader entry section
            mutex.acquire() # Ensure only one reader modifies the read_count at a time
            read_count += 1
            if read_count == 1:
                rw_lock.acquire() # First reader locks out writers
            mutex.release()
            # Critical section (reading)
            print(f"Reader {self.reader_id} is reading the shared data: {shared_data}")
            time.sleep(random.uniform(1, 2)) # Simulate reading time
            # Reader exit section
            mutex.acquire()
            read_count -= 1
            if read_count == 0:
                rw_lock.release() # Last reader releases the writer lock
            mutex.release()
            time.sleep(random.uniform(1, 3)) # Simulate time between reads
class Writer(threading.Thread):
    def __init__(self, writer_id):
        threading.Thread.__init__(self)
        self.writer_id = writer_id
    def run(self):
        global shared_data
        while True:
            rw_lock.acquire() # Writers need exclusive access
            # Critical section (writing)
            shared_data = random.randint(1, 100) # Modify the shared resource
            print(f"Writer {self.writer_id} is writing new value: {shared_data}")
            time.sleep(random.uniform(2, 4)) # Simulate writing time
```

```
            rw_lock.release()
            time.sleep(random.uniform(3, 5)) # Simulate time between writes
# Number of readers and writers
num_readers = 3
num_writers = 2
# Create and start reader threads
readers = [Reader(i) for i in range(num_readers)]
for reader in readers:
    reader.start()
# Create and start writer threads
writers = [Writer(i) for i in range(num_writers)]
for writer in writers:
    writer.start()
```

Output:-