**Batch: A-3 Roll No.: 16010122104**

**Experiment No. 07**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **TITLE:** Implementation of Process synchronization algorithms using semaphore - producer consumer problem , reader-writers problem |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** Implementation of Process synchronization algorithms using semaphore - producer consumer problem, reader-writers problem

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**CO 3.** To understand the concepts of process synchronization and deadlock.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems” McGraw Hill Third**

**Edition.**

1. **William Stallings, “Operating System Internal & Design Principles”, Pearson.**
2. **Andrew S. Tanenbaum, “Modern Operating System”, Prentice Hall.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Knowledge of Concurrency, Mutual Exclusion, Synchronization, Deadlock, Starvation.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Description of the chosen process synchronization algorithm:

The Readers-Writers Problem is a classic synchronization issue in operating systems that involves managing access to shared data by multiple threads or processes. The problem addresses the scenario where:

* **Readers**: Multiple readers can access the shared data simultaneously without causing any issues because they are only reading and not modifying the data.
* **Writers**: Only one writer can access the shared data at a time to ensure data integrity, as writers modify the data, and concurrent modifications could lead to data corruption or inconsistencies.

**Challenges of the Reader-Writer Problem**

The challenge now becomes how to create a synchronization scheme such that the following is supported:

* **Multiple Readers**: A number of readers may access simultaneously if no writer is presently writing.
* **Exclusion for Writers**: If one writer is writing, no other reader or writer may access the common resource.

**Solution of the Reader-Writer Problem**

There are two fundamental solutions to the Readers-Writers problem:

* **Readers Preference:** In this solution, readers are given preference over writers. That means that till readers are reading, writers will have to wait. The Writers can access the resource only when no reader is accessing it.
* **Writer’s Preference:** Preference is given to the writers. It simply means that, after arrival, the writers can go ahead with their operations; though perhaps there are readers currently accessing the resource.

**Implementation details:** (printout of code)

import threading

import time

import random

read\_count = 0

shared\_data = 0

read\_count\_mutex = threading.Lock()

resource\_access = threading.Semaphore(1)

reader\_priority = threading.Lock()

def reader(reader\_id, iterations=5):

    global read\_count

    for \_ in range(iterations):

        time.sleep(random.randint(1, 3))

        with reader\_priority:

            with read\_count\_mutex:

                read\_count += 1

                if read\_count == 1:

                    resource\_access.acquire()

        print(f"Reader {reader\_id} is reading the resource. Readers count: {read\_count}")

        time.sleep(random.randint(1, 2))

        with read\_count\_mutex:

            read\_count -= 1

            print(f"Reader {reader\_id} has finished reading. Readers count: {read\_count}")

            if read\_count == 0:

                resource\_access.release()

def writer(writer\_id, iterations=5):

    global shared\_data

    for \_ in range(iterations):

        time.sleep(random.randint(1, 3))

        resource\_access.acquire()

        shared\_data += 1

        print(f"Writer {writer\_id} is writing to the resource. Updated data: {shared\_data}")

        time.sleep(random.randint(1, 2))

        print(f"Writer {writer\_id} has finished writing.")

        resource\_access.release()

def simulate\_readers\_writers(num\_readers, num\_writers, iterations=5):

    reader\_threads = []

    writer\_threads = []

    for i in range(num\_readers):

        t = threading.Thread(target=reader, args=(i, iterations))

        reader\_threads.append(t)

        t.start()

    for i in range(num\_writers):

        t = threading.Thread(target=writer, args=(i, iterations))

        writer\_threads.append(t)

        t.start()

    for t in reader\_threads + writer\_threads:

        t.join()

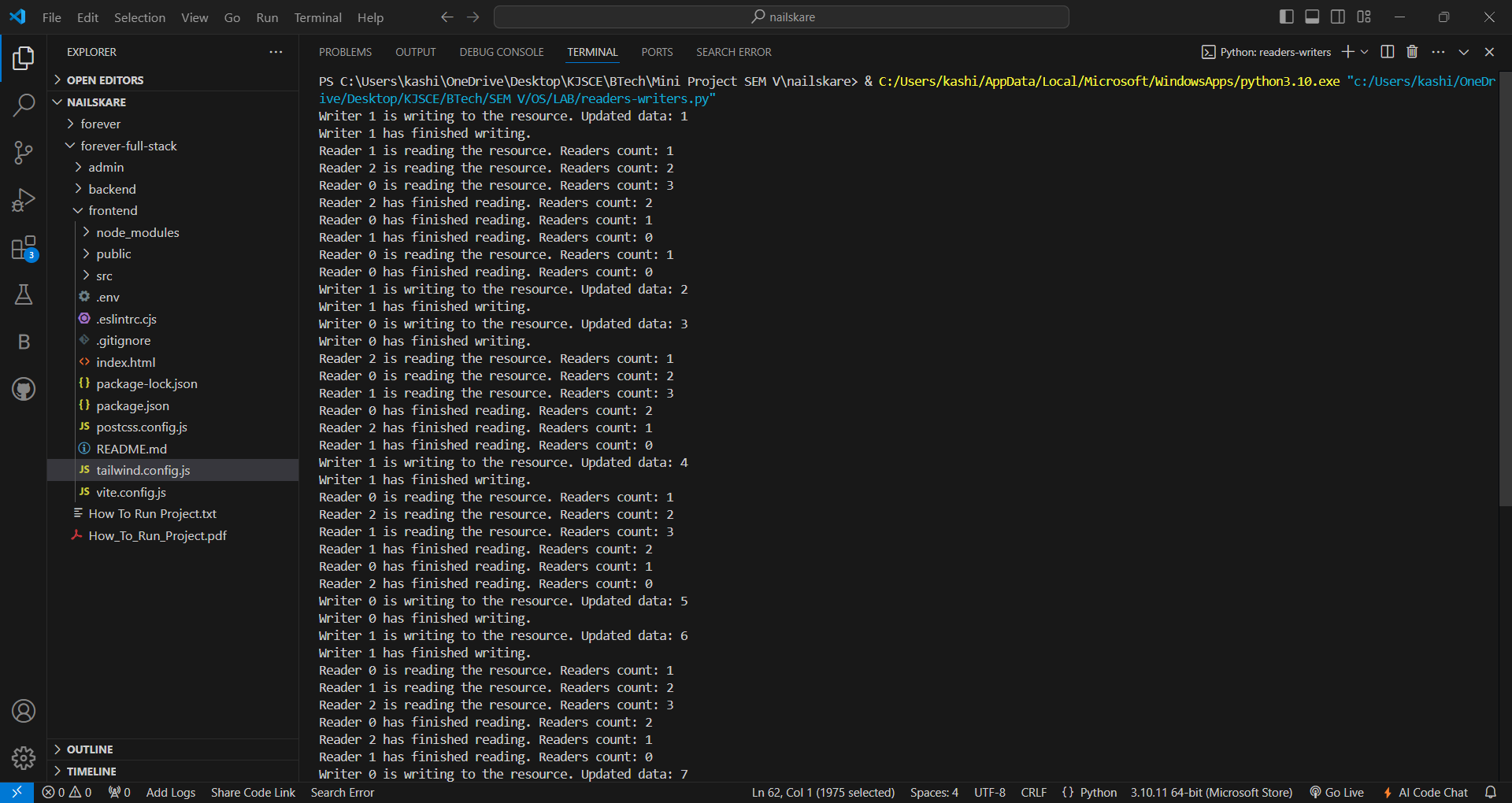
if \_\_name\_\_ == "\_\_main\_\_":

    num\_readers = 3

    num\_writers = 2

    iterations = 5

    simulate\_readers\_writers(num\_readers, num\_writers, iterations)



**Conclusion:**

Implementation of Process synchronization algorithms using reader-writers problem**.**

**Post Lab Objective Questions**

1. A semaphore is a shared integer variable
   1. That can’t drop below zero
   2. That can’t be more than
   3. That can’t drop below one

**Ans:** **(a) That can’t drop below zero**

1. Mutual exclusion can be provided by the
   1. Mute locks
   2. Binary semaphores
   3. Both a and b
   4. None of these

**Ans:** **(c) Both a and b (Mute locks and Binary semaphores)**

1. A monitor is a module that encapsulates
   1. Shared data structures
   2. Procedures that operate on shared data structure
   3. Synchronization between concurrent procedure invocation
   4. All of the above

**Ans:** **(d) All of the above (Shared data structures, procedures that operate on shared data structure, and synchronization between concurrent procedure invocation)**

1. To enable a process to wait within the monitor
   1. A condition variable must be declared as condition
   2. Condition Variables must be used as Boolean objects
   3. Semaphore must be used
   4. All of the above

**Ans:** **(a) A condition variable must be declared as condition**

**Date: 25/10/2024 Signature of faculty in-charge**