| | |
|---|---|
| **Batch: A2** | **Roll No.: 16010121045** |
| **Experiment No. 05** | |
| **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.**
3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**
4. **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 Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.
Below are a few points that considered as the problems occur in Producer-Consumer:

- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

A semaphore S is an integer variable that can be accessed only through two standard operations : wait() and signal().
The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

```
wait(S){
while(S<=0); // busy waiting

S--;

}

signal(S)
{

S++;

}
```

Semaphores are of two types:
1. **Binary Semaphore –** This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

**Producer:**

```
do{
//produce an item
wait(empty);
wait(mutex);
//place in buffer
signal(mutex);
signal(full);
}while(true)
```

**Consumer:**

```
do{
wait(full);
wait(mutex);
// remove item from buffer
signal(mutex);
signal(empty);
// consumes item
}while(true)
```

**Implementation details:**

```python
import threading
import random
import time

# Use a list as the queue
queue = []
# Use threading.Semaphore to control the queue size
queueIsAvailable = threading.Semaphore(5)
dataIsAvailable = threading.Semaphore(0)
mutex = threading.Lock()

def producer():
    nums = range(5)
```

```
    global queue
    while True:
        num = random.choice(nums)
        queueIsAvailable.acquire()
        mutex.acquire()
        queue.append(num)
        print("Produced", num, "\tBuffer:", queue)
        mutex.release()
        dataIsAvailable.release()
        time.sleep(random.uniform(0, 3))  # Use
random.uniform for non-integer sleep times

def consumer():
    global queue
    while True:
        dataIsAvailable.acquire()
        mutex.acquire()
        num = queue.pop(0)
        print("Consumed", num, "\tBuffer:", queue)
        mutex.release()
        queueIsAvailable.release()
        time.sleep(random.uniform(0, 3)  # Use
random.uniform for non-integer sleep times

producerThread = threading.Thread(target=producer)
consumerThread = threading.Thread(target=consumer)

producerThread.start()
consumerThread.start()
```

```
Produced 2      Buffer :  [2]
Consumed 2      Buffer :  []
Produced 4      Buffer :  [4]
Consumed 4      Buffer :  []
Produced 4      Buffer :  [4]
Produced 1      Buffer :  [4, 1]
Consumed 4      Buffer :  [1]
Produced 3      Buffer :  [1, 3]
Consumed 1      Buffer :  [3]
Produced 3      Buffer :  [3, 3]
Consumed 3      Buffer :  [3]
Consumed 3      Buffer :  []
Produced 1      Buffer :  [1]
Consumed 1      Buffer :  []
Produced 0      Buffer :  [0]
Consumed 0      Buffer :  []
Produced 1      Buffer :  [1]
Produced 2      Buffer :  [1, 2]
Consumed 1      Buffer :  [2]
Consumed 2      Buffer :  []
Produced 1      Buffer :  [1]
Produced 1      Buffer :  [1, 1]
Consumed 1      Buffer :  [1]
Produced 3      Buffer :  [1, 3]
Consumed 1      Buffer :  [3]
```

```
Produced 0      Buffer :  [2, 0]
Consumed 2      Buffer :  [0]
Produced 2      Buffer :  [0, 2]
Consumed 0      Buffer :  [2]
Produced 4      Buffer :  [2, 4]
Produced 2      Buffer :  [2, 4, 2]
Produced 3      Buffer :  [2, 4, 2, 3]
Produced 0      Buffer :  [2, 4, 2, 3, 0]
Consumed 2      Buffer :  [4, 2, 3, 0]
Produced 3      Buffer :  [4, 2, 3, 0, 3]
Consumed 4      Buffer :  [2, 3, 0, 3]
Produced 1      Buffer :  [2, 3, 0, 3, 1]
Consumed 2      Buffer :  [3, 0, 3, 1]
Consumed 3      Buffer :  [0, 3, 1]
Produced 4      Buffer :  [0, 3, 1, 4]
Produced 3      Buffer :  [0, 3, 1, 4, 3]
Consumed 0      Buffer :  [3, 1, 4, 3]
Produced 4      Buffer :  [3, 1, 4, 3, 4]
Consumed 3      Buffer :  [1, 4, 3, 4]
Produced 4      Buffer :  [1, 4, 3, 4, 4]
Consumed 1      Buffer :  [4, 3, 4, 4]
Consumed 4      Buffer :  [3, 4, 4]
Consumed 3      Buffer :  [4, 4]
Consumed 4      Buffer :  [4]
Consumed 4      Buffer :  []
```

**Conclusion:** We learned about process synchronization problems(producer consumer problem and reader writer problem) and successfully implemented producer consumer problem using semaphores.

## Post Lab Objective Questions

1) A semaphore is a shared integer variable
a) That can't drop below zero
b) That can't be more than
c) That can't drop below one
**Ans:**

2) Mutual exclusion can be provided by the
a) Mute locks
b) Binary semaphores
c) Both a and b
d) None of these
**Ans:**

3) A monitor is a module that encapsulates
a) Shared data structures
b) Procedures that operate on shared data structure
c) Synchronization between concurrent procedure invocation
d) All of the above
**Ans:**

4) To enable a process to wait within the monitor
a) A condition variable must be declared as condition
b) Condition Variables must be used as Boolean objects
c) Semaphore must be used
d) All of the above
**Ans:**

**Date:** _____ **Signature of faculty in-charge**