

Lab # 9

PRODUCER-CONSUMER PROBLEM USING SEMAPHORES

Introduction

In this lab, we will look at the producer-consumer pattern in C# and how we can implement this pattern using the System Threading Channels data structure. We will look at the advantages of using this data structure and see an example of it in action. Also study about semaphores with its types and implementation.

The Producer-Consumer Pattern

The Producer-Consumer pattern is where a producer generates some messages or data, as we may call it, and various consumers can read that data and work on it. The main advantage of this pattern is that the producer and consumer are not causally linked in any way. Hence, we can say this is a disconnected pattern. The System Threading Channels data structure can be used to achieve this pattern. It is easy to use and is thread-safe.

Producer-Consumer Problem is also known as bounded buffer problem. The Producer-Consumer Problem is one of the classic problems of synchronization.

Working of Producer-Consumer Pattern:

There is a buffer of N slots and each slot is capable of storing one unit of data. There are two processes running, i.e. Producer and Consumer, which are currently operated in the buffer.

There are certain restrictions/conditions for both the producer and consumer process, so that data synchronization can be done without interruption. These are as follows:

- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The producer must not insert data when the buffer is full.
- The consumer must not remove data when the buffer is empty.
- The producer and consumer should not insert and remove data simultaneously.

C# Implementation of Producer-Consumer Pattern:

```
using System;
using System.Collections.Generic;
using System.Threading.Channels;
using System.Threading.Tasks;
namespace ConsoleAppProducerConsumerPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            var pc = new ProducerConsumer();
            pc.StartChannel();
            Console.ReadKey();
        }
    }
    public class ProducerConsumer
```

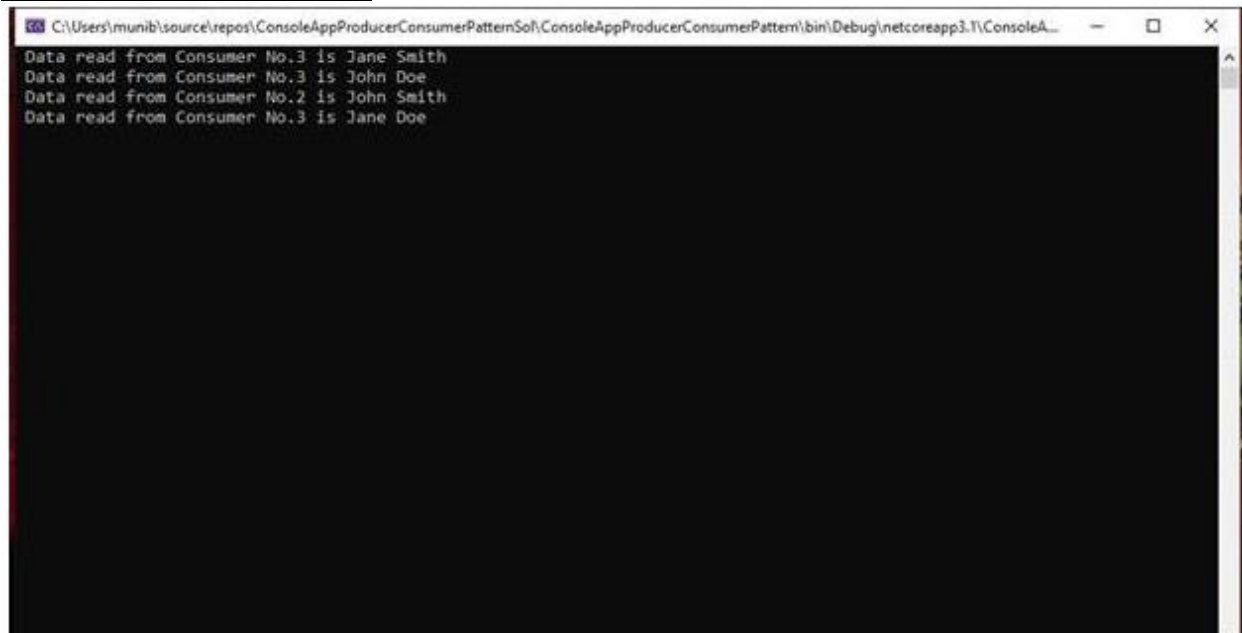
```

{
    static int messageLimit = 5;
    Channel<string> channel = Channel.CreateBounded<string>(messageLimit);
    public void StartChannel()
    {
        List<string> names = new List<string>
        {
            "John Smith",
            "Jane Smith",
            "John Doe",
            "Jane Doe"
        };
        Task producer = Task.Factory.StartNew(() =>
        {
            foreach (var name in names)
            {
                channel.Writer.TryWrite(name);
            }
            channel.Writer.Complete();
        });
        Task[] consumers = new Task[2];
        for (int i = 0; i < consumers.Length; i++)
        {
            consumers[i] = Task.Factory.StartNew(async () =>
            {
                while (await channel.Reader.WaitToReadAsync())
                {
                    if (channel.Reader.TryRead(out var data))
                    {
                        Console.WriteLine($"Data read from Consumer No.{Task.CurrentId} is
{data}");
                    }
                }
            });
        }

        producer.Wait();
        Task.WaitAll(consumers);
    }
}

```

OUTPUT OF THE CODE:



```
C:\Users\munib\source\repos\ConsoleAppProducerConsumerPatternSol\ConsoleAppProducerConsumerPattern\bin\Debug\netcoreapp3.1\ConsoleA...
Data read from Consumer No.3 is Jane Smith
Data read from Consumer No.3 is John Doe
Data read from Consumer No.2 is John Smith
Data read from Consumer No.3 is Jane Doe
```

In the above example, we create a channel. There are two types of channels, bound and un-bound. Here we are using a bound channel which gives us more control over the channel, like setting the maximum number of messages that the channel can carry. This is important in a scenario where we do not want to overload the channel with producer messages to the point that they cannot be handled by the consumers.

First, we create a simple generic list of strings. These will be the messages that will be sent on the channel by the producer to be read by the consumers. Next, we create a task for the producer, which reads each string from the list and writes it to the channel. Next, we create two tasks for the consumers, which will read the messages (strings) from the channel and write them to the console. This is a simple example for demonstration purposes only. In a real-life scenario, we would probably pass some concrete object from the producer, and in the consumer, we would process that object.

SEMAPHORES:

Semaphores are just normal variables used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.

The process of using Semaphores provides two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

A semaphore is a special kind of synchronization data that can be used only through specific synchronization primitives. When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0 . If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.

Types of Semaphores:

Semaphores are of two types:

Binary Semaphore –

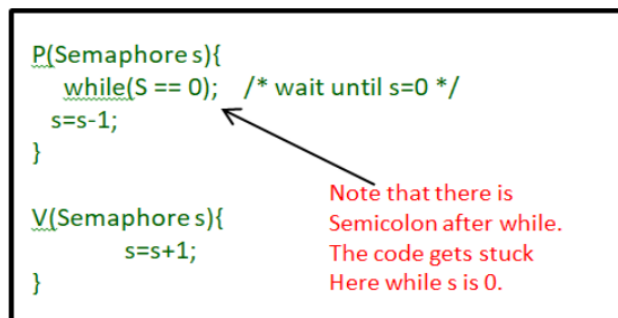
This is also known as a mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

Counting Semaphore –

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Now let us see how it does so.

First, look at two operations that can be used to access and change the value of the semaphore variable.



```
P(Semaphore s){
    while(S == 0); /* wait until s=0 */
    s=s-1;
}

V(Semaphore s){
    s=s+1;
}
```

Note that there is Semicolon after while. The code gets stuck Here while s is 0.

Some points regarding P and V operation:

1. P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in-between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization.

Advantages of Semaphores:

- A simple and effective mechanism for process synchronization
- Supports coordination between multiple processes
- Provides a flexible and robust way to manage shared resources.
- It can be used to implement critical sections in a program.
- It can be used to avoid race conditions.

Disadvantages of Semaphores:

- It Can lead to performance degradation due to overhead associated with wait and signal operations.

- Can result in deadlock if used incorrectly.
- It was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. A semaphore is simply an integer variable that is shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
- It can cause performance issues in a program if not used properly.
- It can be difficult to debug and maintain.
- It can be prone to race conditions and other synchronization problems if not used correctly.
- It can be vulnerable to certain types of attacks, such as denial of service attacks.

C# Implementation of Binary Semaphore:

```
using System.Collections.Generic;
class Semaphore {
    public enum value { Zero, One }
    public Queue<Process> q = new Queue<Process>();
    public void P(Semaphore s, Process p) {
        if (s.value == value.One) {
            s.value = value.Zero;
        } else {
            // add the process to the waiting queue
            q.Enqueue(p);
            p.Sleep();
        }
    }
    public void V(Semaphore s) {
        if (s.q.Count == 0) {
            s.value = value.One;
        } else {
            // select a process from waiting queue
            Process p = q.Peek();
            // remove the process from waiting as it has been
            // sent for CS
            q.Dequeue();
            p.Wakeup();
        }
    }
}
```

The description above is for binary semaphore which can take only two values 0 and 1 and ensure mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instances is 4. Now we initialize $S = 4$ and the rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or waits for function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 processes P1, P2, P3, P4, and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls the signal function and the value of semaphore becomes positive.

C# Implementation of Counting Semaphore:

```
using System.Collections.Generic;
public class Semaphore {
    public int value;
    // q contains all Process Control Blocks(PCBs)
    // corresponding to processes got blocked
    // while performing down operation.
    Queue<Process> q = new Queue<Process>();

    public void P(Process p)
    {
        value--;
        if (value < 0) {
            // add process to queue
            q.Enqueue(p);
            p.block();
        }
    }

    public void V()
    {
        value++;
        if (value <= 0) {
            // remove process p from queue
            Process p = q.Dequeue();
            p.wakeup();
        }
    }
}
```

In this implementation whenever the process waits it is added to a waiting queue of processes associated with that semaphore. This is done through the system call block() on that process. When a process is completed it calls the signal function and one process in the queue is resumed. It uses the wakeup() system call.

LAB TASKS:

1. Implement the Producer-Consumer pattern ?
2. What is the purpose of using semaphores?
3. How are semaphores used in concurrent programming?
4. Implement the semaphore types with various processes?