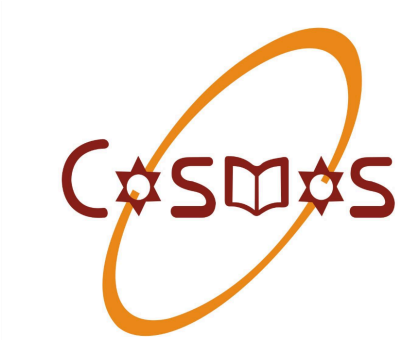
Cosmos College of Management and Technology
Affiliated to Pokhara University
Saddobato, Lalitpur



Lab Report on: Process synchronization: Producer Consumer Problem

Lab report number: 04

Submitted By

Name: Aayush Karki

Roll num: 200101

Faculty: BE IT

Semester: V

Sub: AOS

Group: A

Submitted To

Department of ICT

2080/09/27

.....

Lab Number: 04

Lab Title: Process synchronization: Producer Consumer Problem

Lab Objective

1. Understand the concept of process synchronization and its application in the operating system.
2. Understand the producer consumer problem.
3. Observe the consequences of concurrent access to the same resources without any synchronization.
4. Introduce the concept of process synchronization and compare the solutions.

Theory

Process Synchronization is the management of execution of multiple processes in a multi-process system to make sure that they access shared resources in a controlled and predictable manner. The main aim of process synchronization is to resolve the problem of race conditions and other synchronization issues in a concurrent system. To achieve this, different synchronization techniques such as semaphores, monitors, and critical sections are implemented.

In a multiprocessing system, synchronization is necessary to ensure data consistency and hence avoid the risk of deadlocks and other synchronization problems. On the basis of synchronization, processes are categorized as one of the following two types:

1. ***Independent Process:*** That type of process where the execution of that process doesn't affect other processes is known as Independent Process.
2. ***Cooperative Process:*** That type of process where the execution of that process does affect other processes or on the execution of other process, it gets affected is known as Cooperative Process

Process synchronization problem arises in the case of Cooperative processes since they share the resources in cooperative manner. A ***race condition*** occurs in concurrent processes when multiple threads or processes access shared data or execute critical sections concurrently, leading to unpredictable outcomes dependent on the timing and order of their execution. It results in incorrect or unexpected results, and it can be mitigated by treating critical sections as atomic instructions or implementing proper thread synchronization mechanisms, such as locks or atomic variables.

The ***Producer-Consumer problem*** exemplifies the issues of concurrent programming when multiple processes or threads share a fixed-size buffer. Synchronization is essential to prevent race conditions and data corruption.

CODE

This initial code exhibits a Producer-Consumer scenario without synchronization. The producer adds items to the buffer, and the consumer retrieves items, both without coordination, leading to potential race conditions.

```
#include <stdio.h>
#include <unistd.h>
#include <unistd.h>
#include <pthreadtypes.h>
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
void *producer(void *arg)
{
        int item = 1;
        while (1)
      {
                /* produce item and add to the buffer without synchronization */
                buffer[in] = item;
                printf("Producer item %d at index %d\n", buffer[in], in);
                /* using circular queue */
                in = (in + 1) % BUFFER_SIZE;
                item++;
                sleep(1);
    }
pthread_exit(NULL);
}
void *consumer(void *arg)
{
int item;
        while (1)
        {
                /* consume item and remove from the buffer without synchronization
                */
                item = buffer[out];
                printf("Consumer item %d at index %d\n", item, out);
```

```
                /* using circular queue */
                out = (out + 1) % BUFFER_SIZE;
                sleep(2);
        }
pthread_exit(NULL);
}
int main()
{
        pthread_t producer_thread, consumer_thread;
        /* creating producer and consumer thread */
        pthread_create(&producer_thread, NULL, producer, NULL);
        pthread_create(&consumer_thread, NULL, consumer, NULL);
        /* waiting for the threads to terminate */
        pthread_join(producer_thread, NULL);
        pthread_join(consumer_thread, NULL);
        return 0;
}
```

Output

Producer item 1 at index 0
Consumer item 1 at index 0
Producer item 2 at index 1
Consumer item 2 at index 1
Producer item 3 at index 2
Consumer item 3 at index 2
Producer item 4 at index 3
Consumer item 3 at index 2
Producer item 5 at index 4

This second code introduces a shared variable without synchronization. The parent and child processes modify the variable independently, potentially leading to race conditions and data inconsistency.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
// global variable
int g = 5;

int main()
{
        int pid;
        // creating child and parent process
        pid = fork();
        // Executed by parent process
        if (pid > 0)
        {
```

```
                int x = g;
                x++;
                sleep(1);
                g = x;
                printf("\n parent g=%d", g);
        }
        // Executed by child process
        else
        {
                int y = g;
                y--;
                sleep(1);
                g = y;
                printf("\n Child g=%d", g);
        }
        // Executed by both parent & child process
        printf("\n last g=%d", g);
        return 0;
}
```

Output

Parent process: 6
Child process: 4

Last g: 6 Last g:4

The third code implements a solution using ***semaphore counting***. Semaphores `empty_slots` and `full_slots` synchronize buffer access, ensuring proper coordination between the producer and consumer.

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>

#define BUFFER_SIZE 5

//Array named buffer is created
int buffer[BUFFER_SIZE];
int in=0, out =0;

//Flags to indicate whether a slot is empty or full
int empty_slots=BUFFER_SIZE;
int full_slots=0;
```

```
void* producer(void* arg){
	int item=1;

	while(1){
		//Wait until there is an empty slot
		while(empty_slots==0);

		buffer[in]=item;
		printf("Produced item %d at index %d\n",item,in);
		in=(in+1)%BUFFER_SIZE;
		item++;

		//update empty and full slots count
		empty_slots--;
		full_slots++;

		sleep(1);
	}
	pthread_exit(NULL);
}

void consumer(void *arg){
	while(1){
		while(full_slots==0);
		int item=buffer[out];
		printf("Consumed item %d from index %d\n",item,out);
		out=(out+1)%BUFFER_SIZE;
		//update empty and full slot counts
		empty_slots++;
		full_slots--;
		sleep(2);
	}
	pthread_exit(NULL);
}


int main(){
	pthread_t producer_thread, consumer_thread;
	//create producer and consumer thread
	pthread_create(&producer_thread, NULL, producer, NULL);
	pthread_create(&consumer_thread, NULL, consumer, NULL);
	//Join Threads
	pthread_join(producer_thead, NULL);
```

```
    pthread_join(consumer_thread, NULL);
    return 0;
}
```

Output

Producer item 1 at index 0
Consumer item 1 at index 0
Producer item 2 at index 1
Consumer item 2 at index 1
Producer item 3 at index 2
Producer item 4 at index 3
Consumer item 3 at index 2
Producer item 5 at index 4
Producer item 6 at index 0

DISCUSSION

The discussion underscores the critical importance of synchronization in concurrent programming, evident in the potential risks posed by an unprotected shared buffer, where a circular queue is implemented efficiently but without proper synchronization, leading to the risk of race conditions and data corruption. Similarly, the shared variable modifications by both parent and child processes without synchronization highlight the potential for race conditions and data inconsistency, emphasizing the need for coordinated synchronization methods. The discussion on producer and consumer behavior further accentuates the importance of proper synchronization, as the producer adds items without checking for space, and the consumer removes items without checking availability, risking buffer underrun. Overall, these instances underscore the necessity of synchronization mechanisms to ensure coordinated and controlled access to shared resources in concurrent programming.

CONCLUSION

In conclusion, the discussion highlights the vital role of process synchronization in avoiding race conditions, data corruption, and unpredictable behavior in concurrent programming. The examples of unprotected shared resources underscore the risks involved, with the circular queue implementation indicating an awareness of resource management but emphasizing the paramount importance of synchronization. The discussion on producer and consumer behavior stresses the need for checks and coordination to prevent inefficiencies. Overall, implementing synchronization mechanisms like semaphores or mutexes is essential for maintaining program reliability and data integrity in concurrent programming, ensuring controlled and predictable access to shared resources.