

**Cosmos College of Management and  
Technology**

Affiliated to Pokhara University

**Tutepani, Satdobato**



LAB REPORT ON:- Process Synchronization

LAB REPORT NO:- 4

**SUBMITTED BY:-**

NAME: Nischal Khanal

ROLL NO:200120

GROUP: A

SUBJECT: AOS

**SUBMITTED TO:-**

DEPARTMENT: **ICT**

Date: 2080/ 08 /29

FACULTY: BEIT

.....

# Process Synchronization - Producer-Consumer Problem

## Objectives

1. Explore the challenges of concurrent access to shared resources.
2. Implement the Producer-Consumer problem without process synchronization.
3. Analyze the consequences of unsynchronized access to shared data.
4. Introduce and compare solutions for process synchronization.

## Theory

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.

## Producer-Consumer Problem

The 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 <bits/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;
}

```

```
hunter420 .../programs/os/process_sync master ? 09:44 ./producer_consumer_problem
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
```

## Shared Variable

The 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>

/* shared variable */
int g = 5;

int main(){
    /* process id */
    int pid;

    /* creating parent and child process */
    pid = fork();

    /* executed by parent process */
    if(pid > 0){
        int x =g;
        x++;
        sleep(1);
        g =x;
        printf("Parent Process: %d",g);
    }

    /* executed by child process */
    else if(pid == 0){
        int y =g;
        y--;
        sleep(1);
        g = y;
    }
}
```

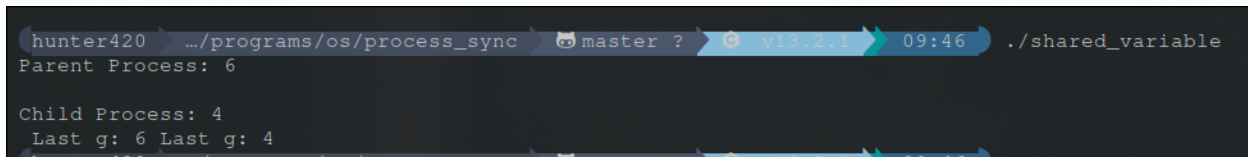
```

        printf("\nChild Process: %d",g);
    }

    /* executed by both and child process */
    printf("\n Last g: %d",g);

    return 0;
}

```



```

hunter420 .../programs/os/process_sync master ? v13.2.1 09:46 ./shared_variable
Parent Process: 6

Child Process: 4
Last g: 6 Last g: 4

```

## Solutions for Process Synchronization

### Semaphore Counting

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

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

void* producer(void* arg){
    int item = 1;
    while(1){
        /*Busy wait for empty slot */
        while(empty_slots == 0);

```

```

    /* produce item and add to the buffer with synchronization */
    buffer[in] = item;
    printf("Producer item %d at index %d\n",buffer[in],in);
    /* using circular queue */
    in = (in + 1) % BUFFER_SIZE;
    item++;

    /* signal that a slot is full */
    full_slots++;
    /* signal that a slot is empty */
    empty_slots--;

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

void* consumer(void* arg){
    int item;
    while(1){
        /* wait for full slot */
        while(full_slots == 0);
        /* consume item and remove from the buffer with synchronization */
        item = buffer[out];
        printf("Consumer item %d at index %d\n",item,out);
        /* using circular queue */
        out = (out + 1) % BUFFER_SIZE;

        /* signal that a slot is empty */
        empty_slots++;
        /* signal that a slot is full */
        full_slots--;

        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;
}

```

```

hunter420 ~/programs/os/process_sync master ? 09:47 ./solution_semaphore_producer_consumer_solution
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

### 1. Unprotected Shared Buffer:

- Lack of synchronization in the buffer access can cause race conditions and data corruption.
- The circular queue implementation helps manage the buffer, but synchronization is crucial.

### 2. Shared Variable:

- The parent and child processes modify a shared variable without synchronization.
- This may lead to race conditions and data inconsistency.

### 3. Producer and Consumer Behavior:

- The producer continuously adds items without checking for available space.
- The consumer removes items without checking for available items, risking buffer underrun.

## Conclusion

The Producer-Consumer problem emphasizes the importance of process synchronization. Unprotected access to shared resources can result in race conditions and unpredictable behavior. The shared variable (Code 2) and semaphore counting (Code 3) illustrate different synchronization methods, highlighting the need for coordination to maintain program reliability and data integrity in concurrent

programming. Understanding and implementing synchronization mechanisms, such as semaphores or mutexes, are essential skills for developing robust concurrent programs.