

**Department of Computer Engineering**

**Course: Operating Systems**

**Course Code: BTECCE21502**

**Mini-Project -Report**

**PART - B**

**Guidance By - Prof. Noshir Tarapore**

**Topic: IPC-Based Process Coordination System**

**By**

|  |  |  |
| --- | --- | --- |
| **Roll No** | **SRN** | **Name of Student** |
| 38 | 202200930 | Shyamal S. Patil |
| 39 | 202200931 | Shaikh M. Asad |
| 33 | 202101389 | Tanmay Gawali |

**Part B: Implementation of synchronization primitives**

In this section, we will implement basic synchronization primitives, specifically mutexes and semaphores, using the POSIX threads library (pthread). These primitives are essential for coordinating and synchronizing concurrent threads in a multi-threaded environment.

Algorithm:

**Step 1: Initialization**

1.1. Include necessary header files: **stdio.h**, **stdlib.h**, **pthread.h**, **semaphore.h**, **unistd.h**.

1.2. Define global variables:

* **shared\_resource** to represent a shared resource.
* **counter** to keep track of the number of shared resource updates.
* **mutex** as a pthread mutex for synchronization.
* **semaphore** as a semaphore for signaling between producer and consumer.
* **condition** as a pthread condition variable for signaling the main thread.

**Step 2: Function Definitions**

2.1. Define a function **do\_work** to simulate work for threads. It prints a message indicating which thread is working.

2.2. Define a **producer** function to be executed by the producer thread. The producer increments the **shared\_resource**, does work, signals the consumer using the semaphore, and repeats this process five times.

2.3. Define a **consumer** function to be executed by the consumer thread. The consumer waits for the semaphore signal from the producer, decrements the **shared\_resource**, does work, and signals the main thread using the condition variable when the **counter** reaches 0.

**Step 3: Main Function**

3.1. In the **main** function:

* Declare two pthread variables for the producer and consumer threads.

3.2. Initialize synchronization primitives:

* Initialize the **mutex** using **pthread\_mutex\_init**.
* Initialize the **semaphore** with an initial value of 0 using **sem\_init**.
* Initialize the **condition** using **pthread\_cond\_init**.

3.3. Create producer and consumer threads:

* Create the producer thread with **pthread\_create**, executing the **producer** function.
* Create the consumer thread with **pthread\_create**, executing the **consumer** function.

3.4. Wait for the consumer to finish:

* Lock the **mutex** using **pthread\_mutex\_lock**.
* Use a loop to check if the **counter** is greater than 0, indicating that the consumer is still working.
* Wait on the **condition** using **pthread\_cond\_wait** until the consumer signals that it has finished.
* Unlock the **mutex** using **pthread\_mutex\_unlock**.

3.5. Clean up synchronization primitives:

* Destroy the **mutex** using **pthread\_mutex\_destroy**.
* Destroy the **semaphore** using **sem\_destroy**.
* Destroy the **condition** using **pthread\_cond\_destroy**.

3.6. Return 0 to indicate successful execution.

This program demonstrates a simple producer-consumer scenario with thread synchronization. The producer and consumer threads coordinate through a shared **semaphore**, and the main thread waits for the consumer to complete its work using a **condition variable**. The synchronization primitives ensure that the producer and consumer work together as intended.

**Implementation Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

// Define a shared resource and a counter

int shared\_resource = 0;

int counter = 0;

// Define mutex, semaphore, and condition variables

pthread\_mutex\_t mutex;

sem\_t semaphore;

pthread\_cond\_t condition;

// Function to simulate work

void do\_work(int id) {

printf("Thread %d is working.\n", id);

}

// Function for the producer thread

void \*producer(void \*arg) {

for (int i = 0; i < 5; i++) {

pthread\_mutex\_lock(&mutex);

shared\_resource++; // Increment the shared resource

counter++;

do\_work(1);

sem\_post(&semaphore); // Signal the consumer

pthread\_mutex\_unlock(&mutex);

sleep(1);

}

return NULL;

}

// Function for the consumer thread

void \*consumer(void \*arg) {

for (int i = 0; i < 5; i++) {

sem\_wait(&semaphore); // Wait for the producer

pthread\_mutex\_lock(&mutex);

shared\_resource--; // Decrement the shared resource

counter--;

do\_work(2);

if (counter == 0) {

pthread\_cond\_signal(&condition); // Signal the main thread

}

pthread\_mutex\_unlock(&mutex);

sleep(1);

}

return NULL;

}

int main() {

pthread\_t producer\_thread, consumer\_thread;

// Initialize synchronization primitives

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&semaphore, 0, 0);

pthread\_cond\_init(&condition, NULL);

// Create producer and consumer threads

pthread\_create(&producer\_thread, NULL, producer, NULL);

pthread\_create(&consumer\_thread, NULL, consumer, NULL);

// Wait for the consumer to finish

pthread\_mutex\_lock(&mutex);

while (counter > 0) {

pthread\_cond\_wait(&condition, &mutex);

}

pthread\_mutex\_unlock(&mutex);

// Clean up synchronization primitives

pthread\_mutex\_destroy(&mutex);

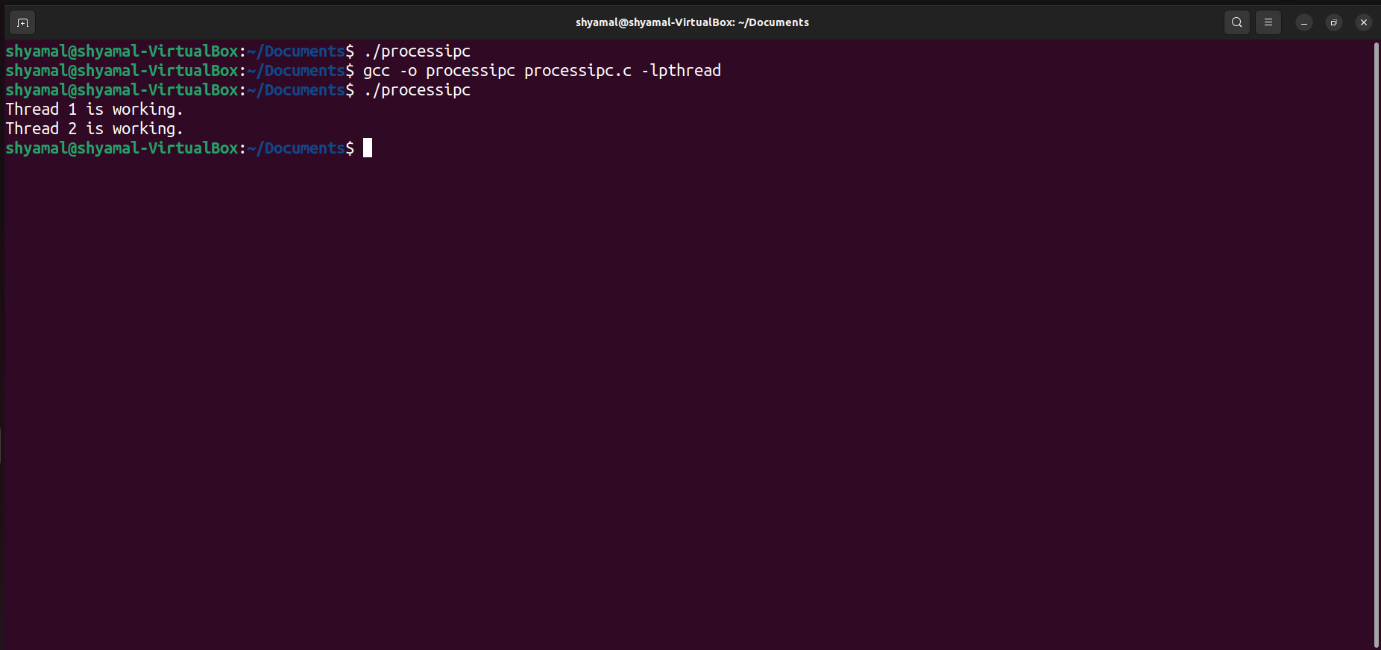
sem\_destroy(&semaphore);

pthread\_cond\_destroy(&condition);

return 0;

}

**Output:**

****