

**Department of Computer Engineering**

**Course: Operating Systems**

**Course Code: BTECCE21502**

**Mini-Project -Report**

**PART - C**

**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 C: Implementation for Solving Classical Synchronization Problems**

We have implemented solutions to two classic synchronization problems: the Producer-Consumer problem and the Dining Philosophers problem.

**Producer-Consumer Problem:**

**Problem Statement:** The Producer-Consumer Problem involves two types of processes, producers and consumers, sharing a common, finite-size buffer as a shared resource. Producers produce data and place it in the buffer, and consumers consume data from the buffer. The problem is to ensure that producers do not add data to a full buffer, and consumers do not remove data from an empty buffer. It requires proper coordination between producers and consumers using synchronization mechanisms.

**Algorithm for Producer-Consumer Problem Implementation:**

**Step 1: Initialization**

1.1. Include the necessary header files:

* **#include <stdio.h>**
* **#include <pthread.h>**
* **#include <semaphore.h>**
* **#include <stdlib.h>**
* **#include <unistd.h>**

1.2. Define constants:

* **BUFFER\_SIZE**: The size of the buffer (e.g., 5).

1.3. Declare global variables:

* **buffer[BUFFER\_SIZE]**: An array representing the shared buffer.
* **empty**: A semaphore to track the number of empty slots in the buffer.
* **full**: A semaphore to track the number of filled slots in the buffer.
* **mutex**: A mutex for protecting access to the shared buffer.

**Step 2: Producer Function**

2.1. Define the **producer** function with the following steps:

* Initialize a variable, **item**, to store the produced item.
* Repeat the following for a predetermined number of iterations (e.g., 10 times):
  + Generate a random item to produce (e.g., **item = rand() % 100**).
  + Wait for an available slot in the buffer (decrement **empty** semaphore) using **sem\_wait**.
  + Lock the mutex to protect the critical section (access to the buffer) using **pthread\_mutex\_lock**.
  + Add the produced item to the buffer, ensuring it cycles through the buffer slots (e.g., **buffer[i % BUFFER\_SIZE] = item**).
  + Print a message indicating the item produced.
  + Unlock the mutex to release the critical section (using **pthread\_mutex\_unlock**).
  + Signal that a slot in the buffer is filled (increment **full** semaphore) using **sem\_post**.
  + Simulate some work using **usleep** to introduce delays.

**Step 3: Consumer Function**

3.1. Define the **consumer** function with the following steps:

* Initialize a variable, **item**, to store the consumed item.
* Repeat the following for a predetermined number of iterations (e.g., 10 times):
  + Wait for a filled slot in the buffer (decrement **full** semaphore) using **sem\_wait**.
  + Lock the mutex to protect the critical section (access to the buffer) using **pthread\_mutex\_lock**.
  + Retrieve and consume an item from the buffer, ensuring it cycles through the buffer slots (e.g., **item = buffer[i % BUFFER\_SIZE]**).
  + Print a message indicating the item consumed.
  + Unlock the mutex to release the critical section (using **pthread\_mutex\_unlock**).
  + Signal that a slot in the buffer is now empty (increment **empty** semaphore) using **sem\_post**.
  + Simulate some work using **usleep** to introduce delays.

**Step 4: Main Function**

4.1. In the **main** function, declare two **pthread\_t** variables for the producer and consumer threads.

4.2. Initialize synchronization primitives:

* Initialize the **empty** semaphore to the buffer size (**sem\_init(&empty, 0, BUFFER\_SIZE)**).
* Initialize the **full** semaphore to 0 (**sem\_init(&full, 0, 0)**).
* Initialize the **mutex** (**pthread\_mutex\_init(&mutex, NULL)**).

4.3. Create producer and consumer threads:

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

4.4. Wait for the producer and consumer threads to finish using **pthread\_join**.

4.5. Clean up synchronization primitives:

* Destroy the **empty** semaphore using **sem\_destroy(&empty)**.
* Destroy the **full** semaphore using **sem\_destroy(&full)**.
* Destroy the **mutex** using **pthread\_mutex\_destroy(&mutex)**.

4.6. Return 0 to indicate successful execution.

This algorithm outlines the implementation of the Producer-Consumer Problem using pthreads and synchronization primitives, ensuring that producers and consumers coordinate and access the shared buffer safely.

**Implementation Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <stdlib.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

sem\_t empty, full;

pthread\_mutex\_t mutex;

void \*producer(void \*arg) {

int item;

for (int i = 0; i < 10; i++) {

item = rand() % 100; // Produce an item

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[i % BUFFER\_SIZE] = item; // Add item to buffer

printf("Produced: %d\n", item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

usleep(rand() % 100000); // Simulate some work

}

return NULL;

}

void \*consumer(void \*arg) {

int item;

for (int i = 0; i < 10; i++) {

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[i % BUFFER\_SIZE]; // Consume item from buffer

printf("Consumed: %d\n", item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

usleep(rand() % 100000); // Simulate some work

}

return NULL;

}

int main() {

pthread\_t producer\_thread, consumer\_thread;

sem\_init(&empty, 0, BUFFER\_SIZE);

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

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

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

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

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

sem\_destroy(&empty);

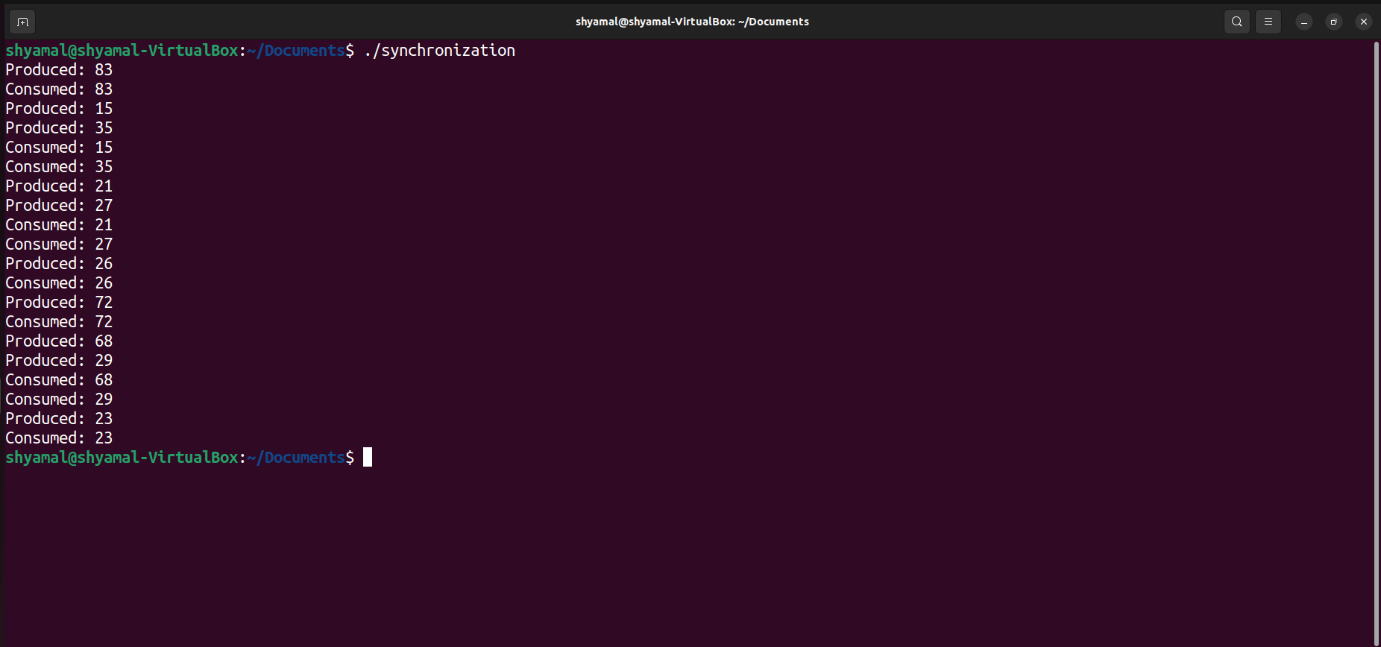
sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

**Output:**

****

**Dining Philosophers Problem:**

**Problem Statement:** The Dining Philosophers Problem is a classic example of deadlock, where a finite number of philosophers sit at a round table. Each philosopher thinks and eats, and to eat, they need two forks. The problem is to ensure that philosophers can eat without causing deadlock. Philosophers must acquire two forks to eat and release them when done.

**Algorithm for Dining Philosophers Problem Implementation:**

**Step 1: Initialization**

1.1. Include the necessary header files:

* **#include <stdio.h>**
* **#include <pthread.h>**
* **#include <semaphore.h>**
* **#include <stdlib.h>**

1.2. Define constants:

* **NUM\_PHILOSOPHERS**: The number of philosophers (e.g., 5).

1.3. Declare global variables:

* **pthread\_t philosophers[NUM\_PHILOSOPHERS]**: An array to store thread IDs for philosophers.
* **sem\_t forks[NUM\_PHILOSOPHERS]**: An array of semaphores to represent forks.

**Step 2: Philosopher Function**

2.1. Define the **philosopher** function with the following steps:

* Initialize an integer variable **id** to represent the philosopher's ID.
* Initialize two integers, **left** and **right**, to represent the philosopher's left and right forks' indices.

2.2. Enter an infinite loop (philosopher's life cycle) with the following steps:

* Think: Print a message indicating that the philosopher is thinking.
* Pick up forks:
  + Use **sem\_wait** to acquire the left fork (**sem\_wait(&forks[left]**).
  + Use **sem\_wait** to acquire the right fork (**sem\_wait(&forks[right]**).
* Eat: Print a message indicating that the philosopher is eating.
* Put down forks:
  + Use **sem\_post** to release the left fork (**sem\_post(&forks[left]**).
  + Use **sem\_post** to release the right fork (**sem\_post(&forks[right]**).

**Step 3: Main Function**

3.1. Initialize forks:

* Use a loop to initialize the semaphores for each fork to 1 (indicating they are available for use):
  + **sem\_init(&forks[i], 0, 1)**.

3.2. Create philosopher threads:

* Use a loop to create threads for each philosopher, each executing the **philosopher** function.
  + Create a dynamic integer pointer (**int \*philosopher\_id**) and allocate memory for it.
  + Set the value of **\*philosopher\_id** to the philosopher's index (i).
  + Use **pthread\_create** to create the philosopher thread and pass the **philosopher\_id** to the thread.

3.3. Wait for philosopher threads to finish:

* Use a loop to wait for each philosopher thread to finish using **pthread\_join**.

3.4. Clean up:

* Use a loop to destroy the semaphores for each fork using **sem\_destroy(&forks[i]**).

3.5. Return 0 to indicate successful execution.

This algorithm outlines the implementation of the Dining Philosophers Problem using pthreads and semaphores. It ensures that philosophers can pick up and put down forks safely and without causing deadlocks.

**Implementation Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <stdlib.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_t philosophers[NUM\_PHILOSOPHERS];

sem\_t forks[NUM\_PHILOSOPHERS];

void \*philosopher(void \*arg) {

int id = \*(int \*)arg;

int left = id;

int right = (id + 1) % NUM\_PHILOSOPHERS;

while (1) {

// Thinking

printf("Philosopher %d is thinking\n", id);

// Pick up forks

sem\_wait(&forks[left]);

sem\_wait(&forks[right]);

// Eating

printf("Philosopher %d is eating\n", id);

// Put down forks

sem\_post(&forks[left]);

sem\_post(&forks[right]);

}

return NULL;

}

int main() {

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_init(&forks[i], 0, 1);

}

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

int \*philosopher\_id = (int \*)malloc(sizeof(int));

\*philosopher\_id = i;

pthread\_create(&philosophers[i], NULL, philosopher, philosopher\_id);

}

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_join(philosophers[i], NULL);

sem\_destroy(&forks[i]);

}

return 0;

}

**Output:**

