# Laboratory Assignments

**Subject: Design Principles of Operating Systems**

# Subject code: CSE 3249

**Assignment 5: Implementation of synchronization using semaphore:**

**Objective of this Assignment:**

* To implement the concept of multi-threading in a process.
* To learn the use of semaphore i.e., to control access to shared resources.

## Producer-Consumer problem

**Problem:** Write a C program to implement the producer-consumer program where:

* + Producer generates integers from 1 to 100.
  + Consumer processes the numbers.

Requirements:

* + Use a shared buffer with a maximum size of 10.
  + Use semaphores and mutex to ensure thread-safe access to the buffer.
  + Print the number that producer is producing and consumer is consuming.
  + Both producer and consumer will continue for 20 iterations

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

// Constants

#define BUFFER\_SIZE 10

#define ITERATIONS 20

// Shared buffer and synchronization primitives

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty\_slots, full\_slots;

pthread\_mutex\_t mutex;

// Producer function

void\* producer(void\* arg) {

for (int i = 1; i <= ITERATIONS; i++) {

// Wait for an empty slot

sem\_wait(&empty\_slots);

// Lock the buffer

pthread\_mutex\_lock(&mutex);

// Produce item

buffer[in] = i;

printf("Producer produced: %d\n", buffer[in]);

in = (in + 1) % BUFFER\_SIZE;

// Unlock the buffer

pthread\_mutex\_unlock(&mutex);

// Signal that a new item is available

sem\_post(&full\_slots);

// Simulate production time

usleep(100000);

}

pthread\_exit(NULL);

}

// Consumer function

void\* consumer(void\* arg) {

for (int i = 1; i <= ITERATIONS; i++) {

// Wait for an available item

sem\_wait(&full\_slots);

// Lock the buffer

pthread\_mutex\_lock(&mutex);

// Consume item

int item = buffer[out];

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

out = (out + 1) % BUFFER\_SIZE;

// Unlock the buffer

pthread\_mutex\_unlock(&mutex);

// Signal that a slot is now empty

sem\_post(&empty\_slots);

// Simulate consumption time

usleep(150000);

}

pthread\_exit(NULL);

}

int main() {

// Initialize semaphores and mutex

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

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

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

// Create producer and consumer threads

pthread\_t producer\_thread, consumer\_thread;

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

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

// Wait for threads to finish

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

// Destroy semaphores and mutex

sem\_destroy(&empty\_slots);

sem\_destroy(&full\_slots);

pthread\_mutex\_destroy(&mutex);

printf("Producer-Consumer program completed.\n");

return 0;

}

## Alternating Numbers with Two Threads

**Problem:** Write a program to print 1, 2, 3 … upto 20. Create threads where two threads print numbers alternately.

* + **Thread A** prints odd numbers: 1, 3, 5 ...
  + **Thread B** prints even numbers: 2, 4, 6 ...

## Requirements:

* + Use semaphores to control the order of execution of the threads.
  + Ensure no race conditions occur.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

// Semaphores for synchronization

sem\_t sem\_odd, sem\_even;

// Function for thread A (prints odd numbers)

void\* print\_odd(void\* arg) {

for (int i = 1; i <= 19; i += 2) {

sem\_wait(&sem\_odd); // Wait for permission to print

printf("Thread A (Odd): %d\n", i);

sem\_post(&sem\_even); // Signal Thread B to print

}

pthread\_exit(NULL);

}

// Function for thread B (prints even numbers)

void\* print\_even(void\* arg) {

for (int i = 2; i <= 20; i += 2) {

sem\_wait(&sem\_even); // Wait for permission to print

printf("Thread B (Even): %d\n", i);

sem\_post(&sem\_odd); // Signal Thread A to print

}

pthread\_exit(NULL);

}

int main() {

// Initialize semaphores

sem\_init(&sem\_odd, 0, 1); // Start with odd numbers

sem\_init(&sem\_even, 0, 0); // Even thread starts blocked

// Create threads

pthread\_t threadA, threadB;

pthread\_create(&threadA, NULL, print\_odd, NULL);

pthread\_create(&threadB, NULL, print\_even, NULL);

// Wait for threads to finish

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

// Destroy semaphores

sem\_destroy(&sem\_odd);

sem\_destroy(&sem\_even);

printf("Alternating Numbers program completed.\n");

return 0;

}

## Alternating Characters

**Problem:** Write a program to create two threads that print characters (A and B) alternately such as ABABABABA…. upto 20. Use semaphores to synchronize the threads.

* + **Thread A** prints A.
  + **Thread B** prints B.

## Requirements:

* + Use semaphores to control the order of execution of the threads.
  + Ensure no race conditions occur.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

// Semaphores for synchronization

sem\_t sem\_A, sem\_B;

// Function for thread A (prints 'A')

void\* print\_A(void\* arg) {

for (int i = 0; i < 10; i++) { // Print 10 'A's

sem\_wait(&sem\_A); // Wait for permission to print

printf("A");

fflush(stdout); // Ensure immediate output

sem\_post(&sem\_B); // Signal Thread B to print

}

pthread\_exit(NULL);

}

// Function for thread B (prints 'B')

void\* print\_B(void\* arg) {

for (int i = 0; i < 10; i++) { // Print 10 'B's

sem\_wait(&sem\_B); // Wait for permission to print

printf("B");

fflush(stdout); // Ensure immediate output

sem\_post(&sem\_A); // Signal Thread A to print

}

pthread\_exit(NULL);

}

int main() {

// Initialize semaphores

sem\_init(&sem\_A, 0, 1); // Start with Thread A

sem\_init(&sem\_B, 0, 0); // Thread B starts blocked

// Create threads

pthread\_t threadA, threadB;

pthread\_create(&threadA, NULL, print\_A, NULL);

pthread\_create(&threadB, NULL, print\_B, NULL);

// Wait for threads to finish

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

// Destroy semaphores

sem\_destroy(&sem\_A);

sem\_destroy(&sem\_B);

printf("\nAlternating Characters program completed.\n");

return 0;

}

1. **Countdown and Countup**

**Problem**: Write a program create two threads where:

* + **Thread A** counts down from 10 to 1.
  + **Thread B** counts up from 1 to 10.

Both threads should alternate execution.

## Requirements:

* + Use semaphores to control the order of execution of the threads.
  + Ensure no race conditions occur.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

// Semaphores for synchronization

sem\_t sem\_A, sem\_B;

// Function for Thread A (counts down from 10 to 1)

void\* countdown(void\* arg) {

for (int i = 10; i >= 1; i--) {

sem\_wait(&sem\_A); // Wait for permission to execute

printf("Thread A (Countdown): %d\n", i);

sem\_post(&sem\_B); // Signal Thread B to execute

}

pthread\_exit(NULL);

}

// Function for Thread B (counts up from 1 to 10)

void\* countup(void\* arg) {

for (int i = 1; i <= 10; i++) {

sem\_wait(&sem\_B); // Wait for permission to execute

printf("Thread B (Countup): %d\n", i);

sem\_post(&sem\_A); // Signal Thread A to execute

}

pthread\_exit(NULL);

}

int main() {

// Initialize semaphores

sem\_init(&sem\_A, 0, 1); // Thread A starts first

sem\_init(&sem\_B, 0, 0); // Thread B starts blocked

// Create threads

pthread\_t threadA, threadB;

pthread\_create(&threadA, NULL, countdown, NULL);

pthread\_create(&threadB, NULL, countup, NULL);

// Wait for threads to finish

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

// Destroy semaphores

sem\_destroy(&sem\_A);

sem\_destroy(&sem\_B);

printf("Countdown and Countup program completed.\n");

return 0;

}

## Sequence Printing using Threads

**Problem:** Write a program that creates three threads: Thread A, Thread B, and Thread C. The threads must print numbers in the following sequence: A1, B2, C3, A4, B5, C6 … upto 20 numbers.

* + **Thread A** prints A1, A4, A7, …
  + **Thread B** prints B2, B5, B8, …
  + **Thread C** prints C3, C6, C9, ...

## Requirements:

* + Use semaphores to control the order of execution of the threads.
  + Ensure no race conditions occur.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

// Semaphores for synchronization

sem\_t sem\_A, sem\_B, sem\_C;

// Shared variable to keep track of the current number

int current\_number = 1;

// Function for Thread A

void\* thread\_A(void\* arg) {

while (current\_number <= 20) {

sem\_wait(&sem\_A); // Wait for Thread A's turn

if (current\_number <= 20) {

printf("A%d\n", current\_number);

current\_number++;

}

sem\_post(&sem\_B); // Signal Thread B to execute

}

pthread\_exit(NULL);

}

// Function for Thread B

void\* thread\_B(void\* arg) {

while (current\_number <= 20) {

sem\_wait(&sem\_B); // Wait for Thread B's turn

if (current\_number <= 20) {

printf("B%d\n", current\_number);

current\_number++;

}

sem\_post(&sem\_C); // Signal Thread C to execute

}

pthread\_exit(NULL);

}

// Function for Thread C

void\* thread\_C(void\* arg) {

while (current\_number <= 20) {

sem\_wait(&sem\_C); // Wait for Thread C's turn

if (current\_number <= 20) {

printf("C%d\n", current\_number);

current\_number++;

}

sem\_post(&sem\_A); // Signal Thread A to execute

}

pthread\_exit(NULL);

}

int main() {

// Initialize semaphores

sem\_init(&sem\_A, 0, 1); // Thread A starts first

sem\_init(&sem\_B, 0, 0); // Thread B starts blocked

sem\_init(&sem\_C, 0, 0); // Thread C starts blocked

// Create threads

pthread\_t threadA, threadB, threadC;

pthread\_create(&threadA, NULL, thread\_A, NULL);

pthread\_create(&threadB, NULL, thread\_B, NULL);

pthread\_create(&threadC, NULL, thread\_C, NULL);

// Wait for threads to finish

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

pthread\_join(threadC, NULL);

// Destroy semaphores

sem\_destroy(&sem\_A);

sem\_destroy(&sem\_B);

sem\_destroy(&sem\_C);

printf("Sequence printing program completed.\n");

return 0;

}