Solutions for Practical Session N°4

Monitor-Based Synchronization in C on Linux (POSIX)

Objective: To practice monitor-based synchronization by implementing safe thread coordination in C using POSIX mutexes and condition variables on Linux.

POSIX Mutex and Condition Variable Operations:

Primitive / Operation	Description	Purpose
pthread_mutex_t myMutex;	Declare a mutex variable.	Prepare a mutex for protecting shared resources.
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;	Static initialization of a mutex.	Ready-to-use mutex with default attributes.
pthread_mutex_init (&myMutex, NULL);	Dynamic initialization of a mutex.	Initialize a mutex at runtime with optional attributes.
pthread_mutex_destroy (&myMutex);	Destroy a mutex.	Release resources associated with the mutex.
pthread_cond_t myCondition;	Declare a condition variable.	Prepare a condition variable for signaling.
pthread_cond_t myCondition = PTHREAD_COND_INITIALIZER;	Static initialization of a condition variable.	Ready-to-use condition variable with default attributes.
pthread_cond_init (&myCondition, NULL);	Dynamic initialization of a condition variable.	Initialize a condition variable at runtime with optional attributes.
pthread_cond_destroy (&myCondition);	Destroy a condition variable.	Release resources associated with the condition variable.
pthread_mutex_lock (&mutex)	Lock the mutex for exclusive access.	Protect shared data from concurrent access.
pthread_mutex_unlock (&mutex)	Unlock the mutex to allow other threads to acquire it.	End of critical section; signal that shared resources are available.
pthread_cond_wait (&conditionVariable, &mutex)	Wait on the condition variable; unlocks mutex while waiting and re-acquires it on wake.	Suspend a thread until a condition is true.
pthread_cond_signal (&conditionVariable)	Wake up one waiting thread.	Notify a thread that the condition may now be satisfied.
pthread_cond_broadcast (&conditionVariable)	Wake up all waiting threads.	Notify all waiting threads.

Important Note:

When using `pthread_cond_wait()`, always wrap it in a `while` loop to recheck the condition after waking. This ensures correctness in case of spurious wakeups or multiple waiting threads. Using `if` alone may work in simple scenarios, but it is unsafe and not recommended.

Exercise 1: (Positive Shared Counter)

Consider a shared integer counter that is initially zero. Two types of processes can access the counter: incrementers, which increase its value, and decrementers, which decrease its value. A decrementer may only execute if the counter is greater than zero; otherwise, it must wait until the counter becomes positive. Implement a C program under POSIX that synchronize the two processes using monitors.

SOLUTION

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

// Shared counter
int counter = 0;

// Mutex and condition variable
pthread_mutex_t counterMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t counterCond = PTHREAD_COND_INITIALIZER;

// Incrementer thread function
void* incrementer(void* arg) {
    while (1) {
        pthread_mutex_lock(&counterMutex);

        counter++; // increment the counter
        printf("Incrementer: counter = %d\n", counter);

        // Signal the decrementer that counter > 0
        pthread_cond_signal(&counterCond);

        pthread_mutex_unlock(&counterMutex);

        sleep(1); // simulate work
    }
    return NULL;
}

// Decrementer thread function
void* decrementer(void* arg) {
    while (1) {
        pthread_mutex_lock(&counterMutex);

        // Wait until counter > 0
        while (counter == 0) {
            pthread_cond_wait(&counterCond, &counterMutex);
        }

        counter--; // decrement the counter
```

```
        printf("Decrementer: counter = %d\n", counter);

        pthread_mutex_unlock(&counterMutex);

        sleep(1); // simulate work
    }
    return NULL;
}

int main() {
    pthread_t incThread, decThread;

    // Create one incrementer and one decrementer
    pthread_create(&incThread, NULL, incrementer, NULL);
    pthread_create(&decThread, NULL, decrementer, NULL);

    // Wait for threads to finish (they won't because of infinite loop)
    pthread_join(incThread, NULL);
    pthread_join(decThread, NULL);

    // Cleanup (not reached in infinite loop)
    pthread_mutex_destroy(&counterMutex);
    pthread_cond_destroy(&counterCond);

    return 0;
}
```

Exercise 2

For each of the following problems (covered in **chapter 2**), write a **C program** using **monitor principle**, to simulate its behavior:

1- The **Rendezvous Problem:** one RDV and N processes

2- Producers/Consumers: assume the program ends after a fixed number of iterations (10 productions)

SOLUTION

1- Rendezvous Problem

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

#define N 4  // Number of other processes (excluding RDV)

// Shared monitor variables
pthread_mutex_t monitorMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t rdvCond = PTHREAD_COND_INITIALIZER;
pthread_cond_t othersCond = PTHREAD_COND_INITIALIZER;

int count = 0;  // Number of processes that have reached the rendezvous

// RDV process function
void* rdvProcess(void* arg) {
    pthread_mutex_lock(&monitorMutex);
```

```c
    // Wait until all N other processes arrive
    while (count < N) {
        pthread_cond_wait(&rdvCond, &monitorMutex);
    }

    printf("RDV process: all processes have arrived!\n");

    // Wake up all other blocked processes
    pthread_cond_broadcast(&othersCond);

    pthread_mutex_unlock(&monitorMutex);
    return NULL;
}

// Other processes function
void* otherProcess(void* arg) {
    int id = *((int*)arg);

    pthread_mutex_lock(&monitorMutex);

    count++;
    printf("Process %d reached the rendezvous (count=%d)\n", id, count);

    // If this is the last process, signal RDV
    if (count == N) {
        pthread_cond_signal(&rdvCond);
    }

    // Wait until RDV process wakes them up
    else pthread_cond_wait(&othersCond, &monitorMutex);

    printf("Process %d is proceeding after RDV\n", id);

    pthread_mutex_unlock(&monitorMutex);
    return NULL;
}

int main() {
    pthread_t rdvThread;
    pthread_t otherThreads[N];
    int ids[N];

    // Create RDV thread
    pthread_create(&rdvThread, NULL, rdvProcess, NULL);

    // Create other N threads
    for (int i = 0; i < N; i++) {
        ids[i] = i+1;
        pthread_create(&otherThreads[i], NULL, otherProcess, &ids[i]);
    }

    // Wait for all threads (they will exit after one rendezvous)
    pthread_join(rdvThread, NULL);
    for (int i = 0; i < N; i++)
        pthread_join(otherThreads[i], NULL);

    // Cleanup
    pthread_mutex_destroy(&monitorMutex);
    pthread_cond_destroy(&rdvCond);
    pthread_cond_destroy(&othersCond);

    return 0;
}
```

2- Producers/Consumers

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

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2
#define NUM_ITERATIONS 10

// Shared buffer
int buffer[BUFFER_SIZE];
int count = 0;      // number of items in buffer
int in = 0;         // next write index
int out = 0;        // next read index

// Monitor variables
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t notFull = PTHREAD_COND_INITIALIZER;
pthread_cond_t notEmpty = PTHREAD_COND_INITIALIZER;

// Producer function
void* producer(void* arg) {
    int id = *((int*)arg);
      int i;
    for (i = 0; i < NUM_ITERATIONS; i++) {
        pthread_mutex_lock(&mutex);

        // Wait if buffer is full
        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&notFull, &mutex);
        }

        // Produce an item
        int item = i + 1 + id * 100; // unique item for demo
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        printf("Producer %d produced item %d (count=%d)\n", id, item, count);

        // Signal consumers
        pthread_cond_signal(&notEmpty);

        pthread_mutex_unlock(&mutex);
        usleep(100000); // simulate work
    }
    return NULL;
}

// Consumer function
void* consumer(void* arg) {
    int id = *((int*)arg);
      int i;
    for (i = 0; i < NUM_ITERATIONS; i++) {
        pthread_mutex_lock(&mutex);

        // Wait if buffer is empty
        while (count == 0) {
            pthread_cond_wait(&notEmpty, &mutex);
        }

        // Consume an item
```

```
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        printf("Consumer %d consumed item %d (count=%d)\n", id, item, count);

        // Signal producers
        pthread_cond_signal(&notFull);

        pthread_mutex_unlock(&mutex);
        usleep(150000); // simulate work
    }
    return NULL;
}

int main() {
    pthread_t producers[NUM_PRODUCERS];
    pthread_t consumers[NUM_CONSUMERS];
    int ids[NUM_PRODUCERS > NUM_CONSUMERS ? NUM_PRODUCERS : NUM_CONSUMERS];

    // Create producer threads
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        ids[i] = i + 1;
        pthread_create(&producers[i], NULL, producer, &ids[i]);
    }

    // Create consumer threads
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        ids[i] = i + 1;
        pthread_create(&consumers[i], NULL, consumer, &ids[i]);
    }

    // Join all threads
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }
    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
    }

    // Cleanup
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&notFull);
    pthread_cond_destroy(&notEmpty);

    return 0;
}
```