Assignment 04

2023PCS0034

Mohammad Faisal Sayed

1) Non – Deterministic behaviour of code

The code provided demonstrates non-deterministic behavior, which means its outcome can vary from one execution to another. This unpredictability arises due to a race condition, a common issue in multithreaded programs.

In this scenario, two threads, "Thread A" and "Thread B," are created to execute the `mythread` function concurrently. Inside `mythread`, each thread enters a loop where it increases a `counter` variable by 1,000,000 times.

The trouble lies in the fact that both threads access and modify the counter variable simultaneously without any coordination. This can lead to interference between them, causing unexpected results.

While the counter variable is marked as volatile, which prevents the compiler from optimizing its operations, it doesn't protect against race conditions. The order in which these threads are scheduled and execute is managed by the operating system's scheduler and can vary each time you run the program. Consequently, the final value of `counter` becomes dependent on how the instructions from both threads interleave, making the program's output unpredictable.

To ensure consistent and deterministic behavior, it's essential to implement proper synchronization mechanisms, such as mutexes or locks, to control access to shared resources. These mechanisms prevent data races and provide the necessary coordination between threads, ensuring that the program behaves predictably regardless of its execution context.

2) Code for deterministic Behaviour

```
/*

We added a pthread_mutex_t named counter_mutex to protect access to the sharedCounter variable. Before each thread increments the sharedCounter, it locks the mutex using pthread_mutex_lock(), and after incrementing, it unlocks the mutex using pthread_mutex_unlock(). This ensures that only one thread can modify the sharedCounter at a time, making the program's output deterministic.

*/

#include <stdio.h>

#include <pthread.h>

// A shared counter variable

static int sharedCounter = 0;

// Mutex for protecting sharedCounter

static pthread_mutex_t counterMutex = PTHREAD_MUTEX_INITIALIZER;


void *mythread(void *threadName)

{

    char *name = (char *)threadName;

    printf("%s: begin\n", name);

    for (int i = 0; i < 1000000; ++i)

    {

        // Lock the mutex before accessing the sharedCounter

        pthread_mutex_lock(&counterMutex);

        sharedCounter++;

        // Unlock the mutex after modifying the sharedCounter

        pthread_mutex_unlock(&counterMutex);

    }

    printf("%s: done\n", name);

    return NULL;

}

int main()

{

    pthread_t thread1, thread2;

    printf("main: begin (sharedCounter = %d)\n", sharedCounter);
    // Create two threads

    pthread_create(&thread1, NULL, mythread, "Thread A");

    pthread_create(&thread2, NULL, mythread, "Thread B");

    // Join both threads to wait for them to finish

    pthread_join(thread1, NULL);

    pthread_join(thread2, NULL);

    printf("main: done with both (sharedCounter = %d)\n", sharedCounter);

    return 0;

}
```

3) CPU Scheduling

CPU scheduling is a vital part of operating systems that determines the order in which processes are executed on a CPU. Various algorithms are employed to optimize this process based on different criteria. This document provides a concise overview of nine CPU scheduling algorithms, their functionalities, and key concepts associated with them.

Functionalities

1. **Nine Implemented Algorithms**: The system supports nine CPU scheduling algorithms.
2. **Variable Burst Times**: Each process can have different CPU Burst Time and I/O Burst Time.
3. **Gantt and Timeline Charts**: Visualization tools for the scheduled processes.
4. **Context Switching Time**: Measurement of time spent on switching between processes.
5. **Time Log Animation**: Animated representation of the execution timeline.
6. **Round Robin Comparison**: Compare Round Robin Algorithm for different time quantum values.
7. **Performance Metrics Comparison**: Compare all algorithms based on Average Completion Time, Turn Around Time, Waiting Time, and Response Time.

Scheduling Algorithms

First Come First Serve (FCFS)

- **Type**: Non-Preemptive
- **Description**: Processes are executed in the order they arrive in the ready queue.

Shortest Job First (SJF)

- **Type**: Non-Preemptive
- **Description**: The shortest job in the ready queue is executed first.

Shortest Remaining Job First (SRJF)

- **Type**: Preemptive
- **Description**: The shortest remaining job in the ready queue is executed first.

Longest Job First (LJF)

- **Type**: Non-Preemptive
- **Description**: The longest job in the ready queue is executed first.

Longest Remaining Job First (LRJF)

- **Type**: Preemptive
- **Description**: The longest remaining job in the ready queue is executed first.

Priority Non-Preemptive (PNP)

- **Type**: Non-Preemptive
- **Description**: The highest-priority job in the ready queue is executed first.

Priority Preemptive (PP)

- **Type**: Preemptive
- **Description**: The highest-priority job in the ready queue is executed first, and it can be preempted.

Round Robin (RR)

- **Type**: Preemptive
- **Description**: Jobs in the ready queue are given a fixed time quantum for execution.

Highest Response Ratio Next (HRRN)

- **Type**: Non-Preemptive
- **Description**: Jobs with the highest response ratio in the ready queue are executed first.

States in CPU Scheduler

- **Remain**: Processes yet to arrive.
- **Ready**: Processes ready for execution.
- **Running**: The current process executing on the CPU.
- **Block**: Processes waiting for I/O operations.
- **Terminate**: Processes that have completed all CPU and I/O tasks.