Operating Systems (CSEN 283)

Project 3 Report

Name - Sri Sai Saketh Chillapalli

ID - 07700000089

---

1 Introduction:

This report presents an analysis of a multi-threaded C program designed to demonstrate thread syn-chronization using mutexes. The program consists of multiple threads that execute concurrently, each attempting to access a shared resource (the variable CurrentID). The goal of the program is to ensure that each thread can only access the shared resource when it is their turn.

2 Explanation

The program creates multiple threads, each identified by a unique ID. These threads execute the thread-Function, which simulates a scenario where each thread needs to wait for its turn to access a shared resource. The shared resource is represented by the variable CurrentID. The threadFunction locks a mutex before checking if it's the thread's turn to access the resource. If it's not the thread's turn (CurrentID != myID), the thread prints a message indicating that it's not their turn and increments the notMyTurnCount array for statistical purposes. If it is the thread's turn, the thread prints a message indicating that it's their turn, updates the CurrentID, and unlocks the mutex.

3 Testing

The program was tested to ensure correct behaviour under concurrent execution. The program was com-piled using the following command:

gcc project3.c -o output.out -lpthread

After compiling program was executed and the output was saved in output.txt file using following command:

./output.out > output.txt

Various test cases were employed to verify that:

- Threads wait for their turn to access the shared resource.
- Threads increment their respective notMyTurnCount only when it's not their turn.
- The program terminates gracefully after all threads have completed their execution.

4 Code

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5
#define NUM_PRINTS 5
int CurrentID = 1;
int notMyTurnCount[NUM_THREADS];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* threadFunction(void* arg) {
   int myID = *((int*) arg);
   int prints = 0;
   while (prints < NUM_PRINTS) {
       pthread_mutex_lock(&mutex);
       if (CurrentID != myID) {
           printf("Not My Turn! ThreadID: %d\n", myID);
           notMyTurnCount[myID - 1]++;
           pthread_mutex_unlock(&mutex);
       } else {
           printf("My Turn! ThreadID: %d\n", myID);
           CurrentID++;
           if (CurrentID == 6)
               CurrentID = 1;
           prints++;
           pthread_mutex_unlock(&mutex);
       }
   }
   return NULL;
}
int main() {
   pthread_t threads[NUM_THREADS];
   int threadIDs[NUM_THREADS];
   for (int i = 0; i < NUM_THREADS; i++) {
       threadIDs[i] = i + 1;
       notMyTurnCount[i] = 0;
       pthread_create(&threads[i], NULL, threadFunction, &threadIDs[i]);
   }
   for (int i = 0; i < NUM_THREADS; i++) {
```

```
        pthread_join(threads[i], NULL);
    }
    printf("\nNumber of 'Not My Turn!' prints for each thread:\n");
    for (int i = 0; i < NUM_THREADS; i++) {
        printf("Thread %d: %d\n", i + 1, notMyTurnCount[i]);
    }
    return 0;
}
```

5 Results

Upon testing, the program demonstrated correct synchronization among threads. Each thread correctly waited for its turn to access the shared resource, as evidenced by the output messages. The statistical analysis also confirmed that threads

6 Conclusion

The program successfully demonstrates the use of mutexes for thread synchronization in a multi-threaded environment. Mutexes ensure that threads can safely access shared resources without interfering with each other. Understanding and implementing mutex-based synchronization mechanisms are crucial for developing robust multi-threaded applications.