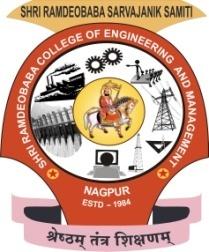
****

**SHRI RAMDEOBABA COLLEGE OF ENGINEERING AND MANAGEMENT**

**Department of Computer Science & Engineering**

**Session: 2022-23**

**OPERATING SYSTEM**

**ASSIGNMENT**

**IV Semester, B.E.**

**Shift II**

**Group Members**

**Maitri Markandeywar (07)**

**Rishabh Jain (54)**

**Course Co-ordinator**

**Prof.Heena Agrawal**

**STATEMENT 22 B**

**Consider a server program running in an online market place firm. The program receives buy and sell orders for one type of commodity from external clients. For every buy or sell request received by the server, the main process spawns a new buy or sell thread. We require that every buy thread waits until a sell thread arrives, and vice versa. A matched pair of buy and sell threads will both return a response to the clients and exit. You may assume that all buy/sell requests are identical to each other, so that any buy thread can be matched with any sell thread. The code executed by the buy thread is shown below (the code of the sell thread would be symmetric). You have to write the synchronization logic that must be run at the start of the execution of the thread to enable it to wait for a matching sell thread to arrive (if none exists already). Once the threads are matched, you may assume that the function completeBuy() takes care of the application logic for exchanging information with the matching thread, communicating with the client, andfinishing the transaction. You may use any synchronization technique of your choice.**

**APPROACH:**

**This code implements a simple multi-threaded program for matching buy and sell orders. Here's a summarized explanation of the logic:**

**1. The code uses semaphores (`buy\_sem` and `sell\_sem`) and a mutex (`mutex`) for synchronization between the buy and sell threads.**

**2. The `buy\_thread` function represents a thread that wants to buy. It waits for a sell thread to be available by calling `sem\_wait(&sell\_sem)`. Once matched with a sell thread, it prints a message indicating the match.**

**3. To ensure thread safety, the code acquires a mutex lock (`pthread\_mutex\_lock(&mutex)`) before calling the `completeBuy` function, which simulates the completion of the buy operation. Once completed, the mutex lock is released (`pthread\_mutex\_unlock(&mutex)`).**

**4. The `sell\_thread` function represents a thread that wants to sell. It waits for a buy thread to be available by calling `sem\_wait(&buy\_sem)`. Once matched with a buy thread, it prints a message indicating the match.**

**5. The main function initializes semaphores and mutex, creates multiple buy and sell threads, and waits for their completion using `pthread\_join`. After all threads have completed, it destroys the semaphores and mutex.**

**In summary, the code sets up a matching system where buy and sell threads wait for each other using semaphores. The mutex ensures that the `completeBuy` function is called safely by only one thread at a time.**

**INPUT CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**sem\_t buy\_sem;**

**sem\_t sell\_sem;**

**pthread\_mutex\_t mutex;**

**void\* buy\_thread(void\* arg);**

**void\* sell\_thread(void\* arg);**

**void completeBuy(int thread\_id);**

**void\* buy\_thread(void\* arg) {**

**int thread\_id = (int)arg;**

**sem\_wait(&sell\_sem);**

**printf("Buy thread %d matched with a sell thread.\n", thread\_id);**

**pthread\_mutex\_lock(&mutex);**

**completeBuy(thread\_id);**

**pthread\_mutex\_unlock(&mutex);**

**sem\_post(&buy\_sem);**

**pthread\_exit(NULL);**

**}**

**void\* sell\_thread(void\* arg) {**

**int thread\_id = (int)arg;**

**sem\_wait(&buy\_sem);**

**printf("Sell thread %d matched with a buy thread.\n", thread\_id);**

**pthread\_mutex\_lock(&mutex);**

**pthread\_mutex\_unlock(&mutex);**

**sem\_post(&sell\_sem);**

**pthread\_exit(NULL);**

**}**

**void completeBuy(int thread\_id) {**

**printf("Buy thread %d completed.\n", thread\_id);**

**}**

**int main() {**

**int num\_orders = 5;**

**sem\_init(&buy\_sem, 0, num\_orders);**

**sem\_init(&sell\_sem, 0, num\_orders);**

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

**pthread\_t buy\_threads[num\_orders];**

**pthread\_t sell\_threads[num\_orders];**

**int buy\_thread\_ids[num\_orders];**

**int sell\_thread\_ids[num\_orders];**

**for (int i = 0; i < num\_orders; i++) {**

**buy\_thread\_ids[i] = i + 1;**

**pthread\_create(&buy\_threads[i], NULL, buy\_thread, &buy\_thread\_ids[i]);**

**}**

**for (int i = 0; i < num\_orders; i++) {**

**sell\_thread\_ids[i] = i + 1;**

**pthread\_create(&sell\_threads[i], NULL, sell\_thread, &sell\_thread\_ids[i]);**

**}**

**for (int i = 0; i < num\_orders; i++) {**

**pthread\_join(buy\_threads[i], NULL);**

**}**

**for (int i = 0; i < num\_orders; i++) {**

**pthread\_join(sell\_threads[i], NULL);**

**}**

**sem\_destroy(&buy\_sem);**

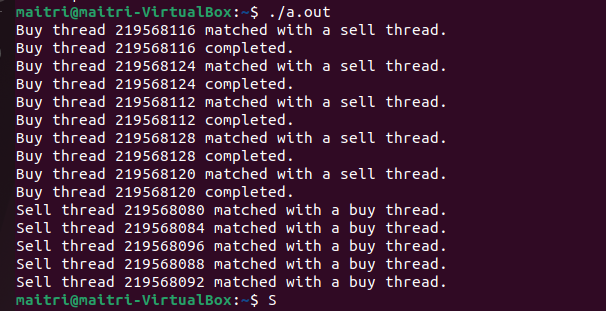
**sem\_destroy(&sell\_sem);**

**pthread\_mutex\_destroy(&mutex);**

**return 0;**

**}**

**OUTPUT:**

****