University of Wolverhampton

School of Mathematics and Computer Science

Student Number: 2407710

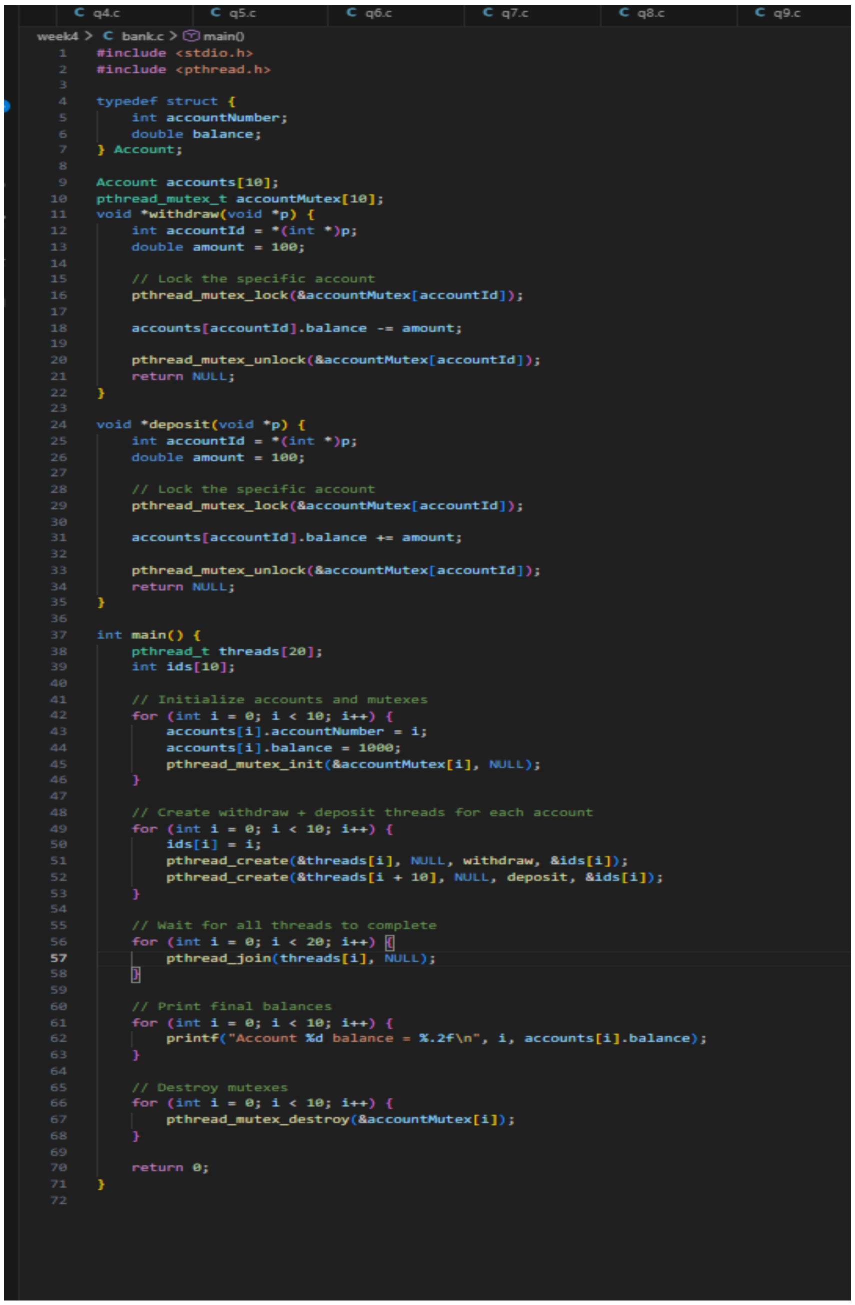
Name: Dhadkan K.C.

6CS005 High Performance Computing Week 4 Workshop

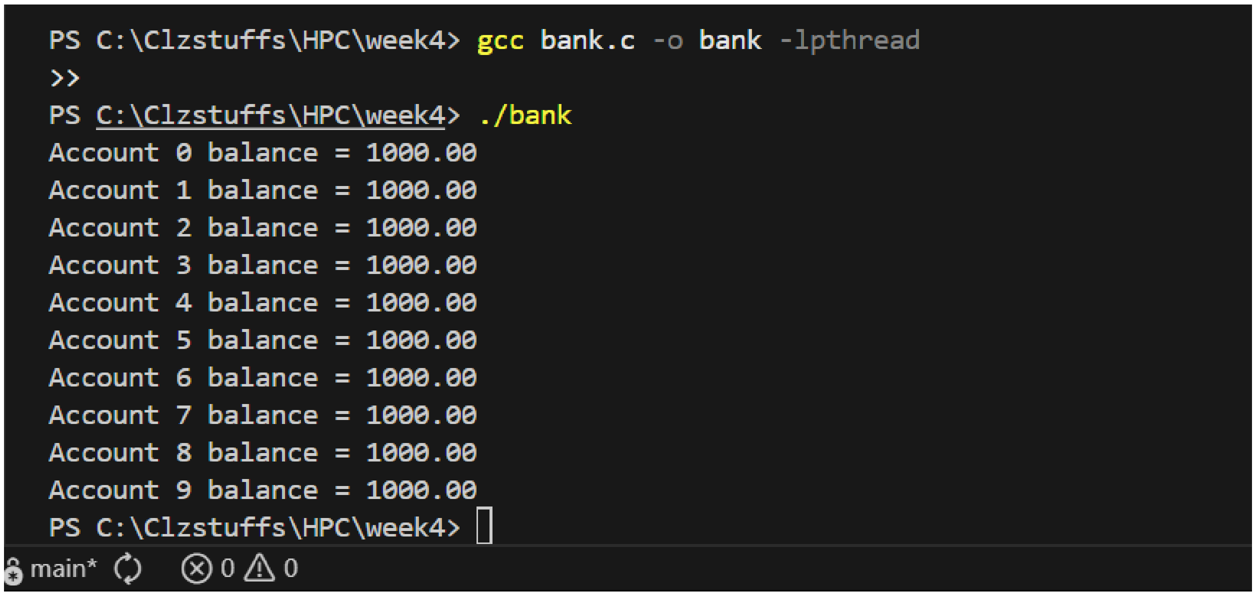
Revision on Multithreading

Tasks – Multithreading

5.In a banking system, multiple users can access and modify their accounts concurrently. Each user has a balance, and they can deposit or withdraw money. Modify the code below to use a **mutex** to prevent race conditions during balance updates and ensure that multiple users can't simultaneously update the balance of the same account.



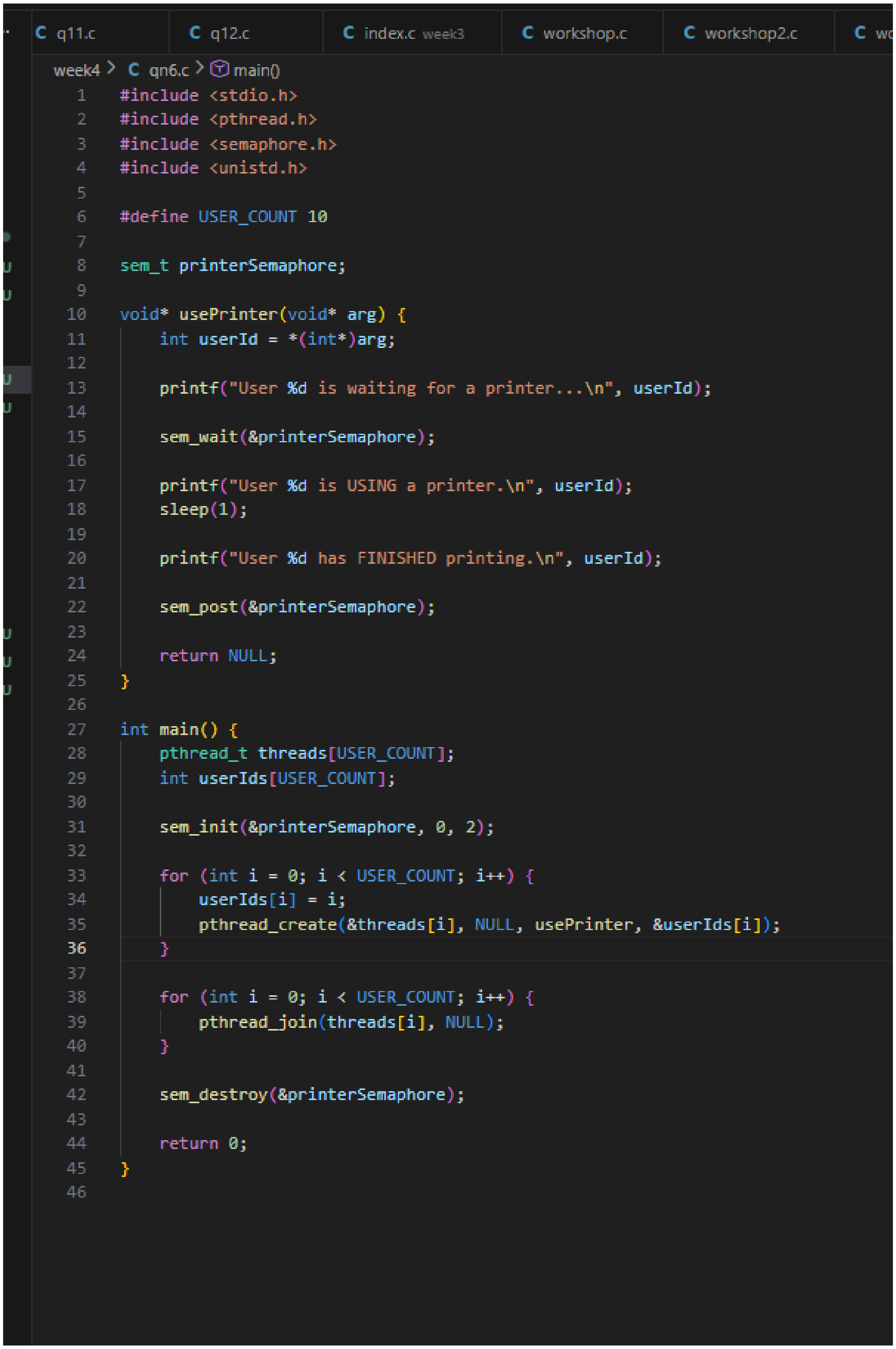
```c
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int accountNumber;
    double balance;
} Account;

Account accounts[10];
pthread_mutex_t accountMutex[10];
void *withdraw(void *p) {
    int accountId = *(int *)p;
    double amount = 100;

    // Lock the specific account
    pthread_mutex_lock(&accountMutex[accountId]);

    accounts[accountId].balance -= amount;

    pthread_mutex_unlock(&accountMutex[accountId]);
    return NULL;
}

void *deposit(void *p) {
    int accountId = *(int *)p;
    double amount = 100;

    // Lock the specific account
    pthread_mutex_lock(&accountMutex[accountId]);

    accounts[accountId].balance += amount;

    pthread_mutex_unlock(&accountMutex[accountId]);
    return NULL;
}

int main() {
    pthread_t threads[20];
    int ids[10];

    // Initialize accounts and mutexes
    for (int i = 0; i < 10; i++) {
        accounts[i].accountNumber = i;
        accounts[i].balance = 1000;
        pthread_mutex_init(&accountMutex[i], NULL);
    }

    // Create withdraw + deposit threads for each account
    for (int i = 0; i < 10; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, withdraw, &ids[i]);
        pthread_create(&threads[i + 10], NULL, deposit, &ids[i]);
    }

    // Wait for all threads to complete
    for (int i = 0; i < 20; i++) {
        pthread_join(threads[i], NULL);
    }

    // Print final balances
    for (int i = 0; i < 10; i++) {
        printf("Account %d balance = %.2f\n", i, accounts[i].balance);
    }

    // Destroy mutexes
    for (int i = 0; i < 10; i++) {
        pthread_mutex_destroy(&accountMutex[i]);
    }

    return 0;
}
```



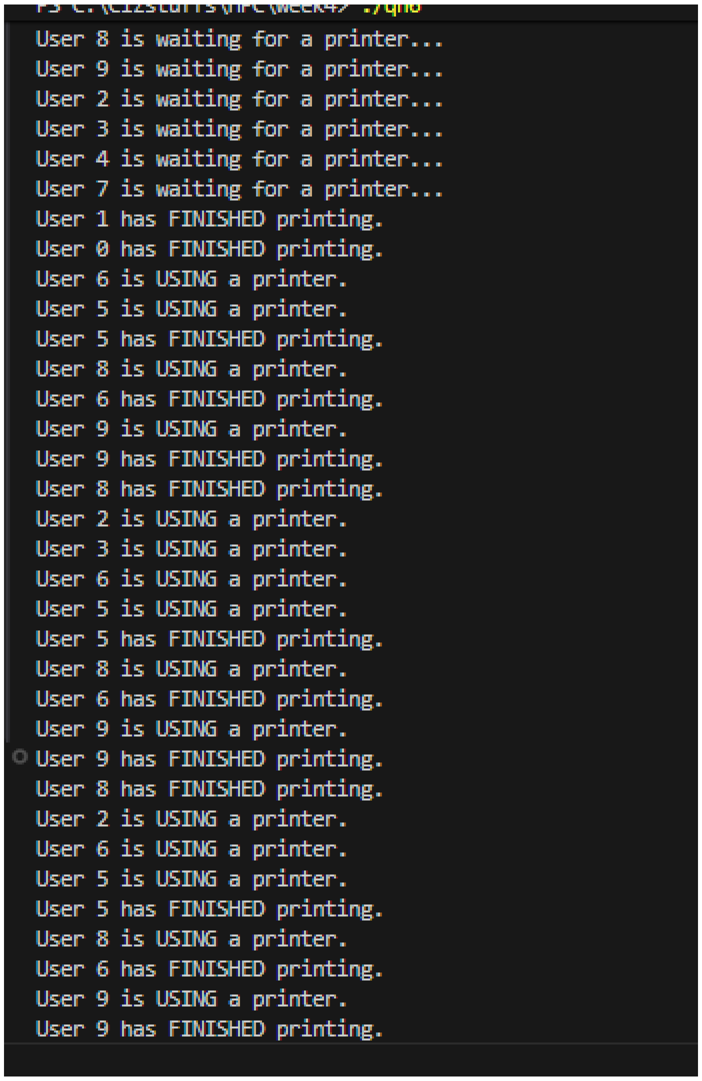
```
PS C:\Clzstuffs\HPC\week4> gcc bank.c -o bank -lpthread
>>
PS C:\Clzstuffs\HPC\week4> ./bank
Account 0 balance = 1000.00
Account 1 balance = 1000.00
Account 2 balance = 1000.00
Account 3 balance = 1000.00
Account 4 balance = 1000.00
Account 5 balance = 1000.00
Account 6 balance = 1000.00
Account 7 balance = 1000.00
Account 8 balance = 1000.00
Account 9 balance = 1000.00
PS C:\Clzstuffs\HPC\week4>
```

This program manages multiple bank accounts safely by using one mutex lock for each account to prevent race conditions. Since there are 10 different shared account resources, a separate mutex is created for each one, and each Account structure includes its own lock. In the withdraw and deposit functions, the mutex for the specific account is locked before updating the balance and unlocked immediately afterward, ensuring that only one thread can modify that account at a time. In the main function, 20 threads are created, and an array of 10 ids is used to identify each account. A loop initializes every account with an account number, a starting balance, and its corresponding mutex. In the next loop, 20 threads are started 10 running the withdraw function and 10 running the deposit function each working on the account indicated by its id. After that, another loop joins all 20 threads to ensure they finish executing. Finally, the program prints the final balance of each account. This setup ensures safe concurrent access by synchronizing balance updates and preventing multiple users from modifying the same account at the same time.

6.A printer is shared among multiple users. Each user can either print a document or wait if the printer is in use. There are only 2 printers in the office. Use **semaphores** to manage the printer access, ensuring that no more than 2 users can print at the same time.



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

#define USER_COUNT 10

sem_t printerSemaphore;

void* usePrinter(void* arg) {
    int userId = *(int*)arg;

    printf("User %d is waiting for a printer...\n", userId);

    sem_wait(&printerSemaphore);

    printf("User %d is USING a printer.\n", userId);
    sleep(1);

    printf("User %d has FINISHED printing.\n", userId);

    sem_post(&printerSemaphore);

    return NULL;
}

int main() {
    pthread_t threads[USER_COUNT];
    int userIds[USER_COUNT];

    sem_init(&printerSemaphore, 0, 2);

    for (int i = 0; i < USER_COUNT; i++) {
        userIds[i] = i;
        pthread_create(&threads[i], NULL, usePrinter, &userIds[i]);
    }

    for (int i = 0; i < USER_COUNT; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&printerSemaphore);

    return 0;
}
```



```
PS C:\Lizstuffs\HPC\Week4> ./qn6
User 8 is waiting for a printer...
User 9 is waiting for a printer...
User 2 is waiting for a printer...
User 3 is waiting for a printer...
User 4 is waiting for a printer...
User 7 is waiting for a printer...
User 1 has FINISHED printing.
User 0 has FINISHED printing.
User 6 is USING a printer.
User 5 is USING a printer.
User 5 has FINISHED printing.
User 8 is USING a printer.
User 6 has FINISHED printing.
User 9 is USING a printer.
User 9 has FINISHED printing.
User 8 has FINISHED printing.
User 2 is USING a printer.
User 3 is USING a printer.
User 6 is USING a printer.
User 5 is USING a printer.
User 5 has FINISHED printing.
User 8 is USING a printer.
User 6 has FINISHED printing.
User 9 is USING a printer.
User 9 has FINISHED printing.
User 8 has FINISHED printing.
User 2 is USING a printer.
User 6 is USING a printer.
User 5 is USING a printer.
User 5 has FINISHED printing.
User 8 is USING a printer.
User 6 has FINISHED printing.
User 9 is USING a printer.
User 9 has FINISHED printing.
```

In the first few lines, I have included all the necessary header files to execute this program. I have defined number of users as 10 and number of printers as 2. I have then initialized a global semaphore called printer_sem which will control access to the printers. The thread function user_point extracts the user id first and then sem_wait() tries to get a printer. If printer is available, it continues to print but if no printer is available or value=0, the thread is blocked and then waits for a printer to be free. Then, it prints for a random time between 1-3 seconds using sleep() and after printing, sem_post(&printer_sem) releases the printer so someone else can use it. In the main function, the srand((unsigned)time(NULL)) makes sure that the sleep times are random. Then, I have created threads as equal to the number of users. I have also declared an array of ids that holds numbers 1-10 so we can pass unique IDs to each thread. Then, I have initialized the semaphore and passed the NUM_PRINTERS in the argument which sets the semaphore value to 2. This means that only two users can print at the same time. The for loop creates 10 threads and each one runs user_print(). The usleep(100000) is used to avoid messy overlapping output. Then, I have joined all the threads to make main program wait until every user thread has finished printing. I have used sem_destroy(&print_sem) to destroy the semaphore.