University of Wolverhampton

School of Mathematics and Computer Science

Student Number: 2408842

Name: Samrajya Neupane

6CS005 High Performance Computing Week 3 Workshop

Revision on Multithreading

Tasks – Multithreading

You will need to refer to the Week 3 lecture sides in order to complete these tasks.

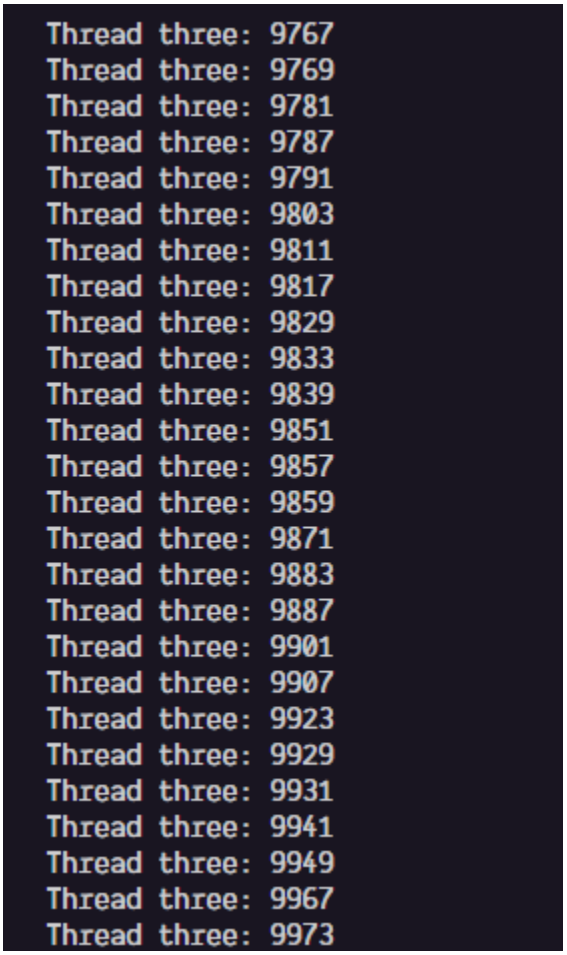
1. Write a multithreaded C program to print out all the prime numbers between 1 to 10000. Use exactly 3 threads.

Code:

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

void *threadOne(void *p)
{
    for (int i = 2; i < 3000; i++)
    {
        int isPrime = 1;
        for (int j = 2; j * j <= i; j++)
        {
            if (i % j == 0)
            {
                isPrime = 0; // not a prime
                break;
            }
        }

        if (isPrime)
        {
            printf("Thread one: %d\n", i);
        }
    }
}

void *threadTwo(void *p)
{

    for (int i = 3000; i < 6000; i++)
    {
        int isPrime = 1;
        for (int j = 2; j * j <= i; j++)
        {
            if (i % j == 0)
            {
                isPrime = 0;
                break;
            }
        }
```

```c
void *threadThree(void *p)
{
    for (int i = 6000; i <= 10000; i++)
    {
        int isPrime = 1;
        for (int j = 2; j * j <= i; j++)
        {
            if (i % j == 0)
            {
                isPrime = 0;
                break;
            }
        }
        if (isPrime)
        {
            printf("Thread three: %d\n", i);
        }
    }
}

void main()
{
    pthread_t thread1, thread2, thread3;

    pthread_create(&thread1, NULL, threadOne, NULL);
    pthread_create(&thread2, NULL, threadTwo, NULL);
    pthread_create(&thread3, NULL, threadThree, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
}
```

Output:



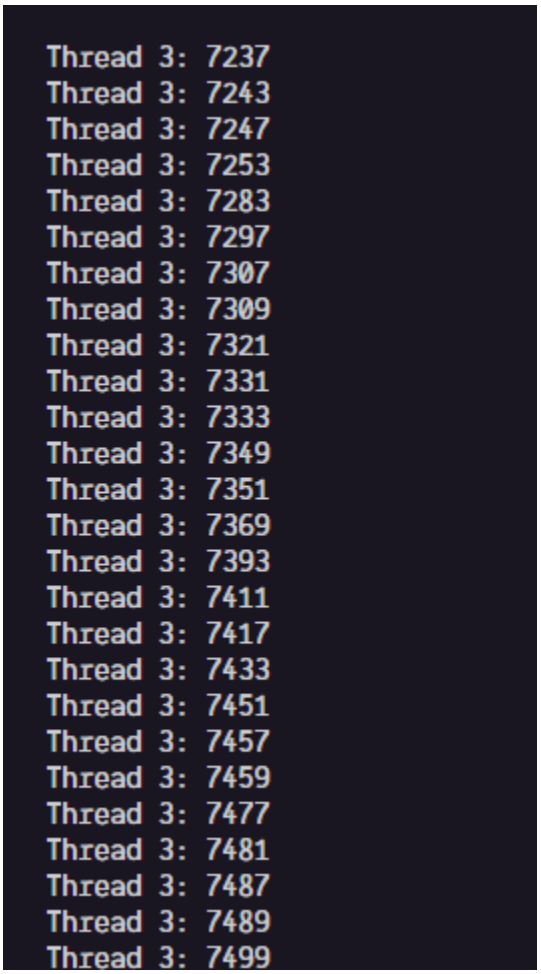
2. Convert this program to prompt the user for a number and then to create the number of threads the user has specified to find the prime numbers.

Code:

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

typedef struct
{
    int start;
    int end;
    int id;
} ThreadData;

void *isPrime(void *arg)
{
    ThreadData *data = (ThreadData *)arg;

    int start = data->start;
    int end = data->end;
    int id = data->id;

    for (int i = start; i <= end; i++)
    {
        int isPrime = 1;

        for (int j = 2; j * j <= i; j++)
        {
            if (i % j == 0)
            {
                isPrime = 0;
                break;
            }
        }

        if (isPrime)
        {
            printf("Thread %d: %d\n", id, i);
        }
    }
}
```

```c
void main()
{
    int n;

    printf("Enter the number of threads: ");
    scanf("%d", &n);

    pthread_t threads[n];
    ThreadData data[n];

    int start = 2;
    int range = 10000 / n;

    for (int i = 0; i < n; i++)
    {
        data[i].start = start;
        data[i].end = (i == n - 1) ? 10000 : start + range - 1;
        data[i].id = i + 1;

        pthread_create(&threads[i], NULL, isPrime, &data[i]);

        start += range;
    }

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

Output:



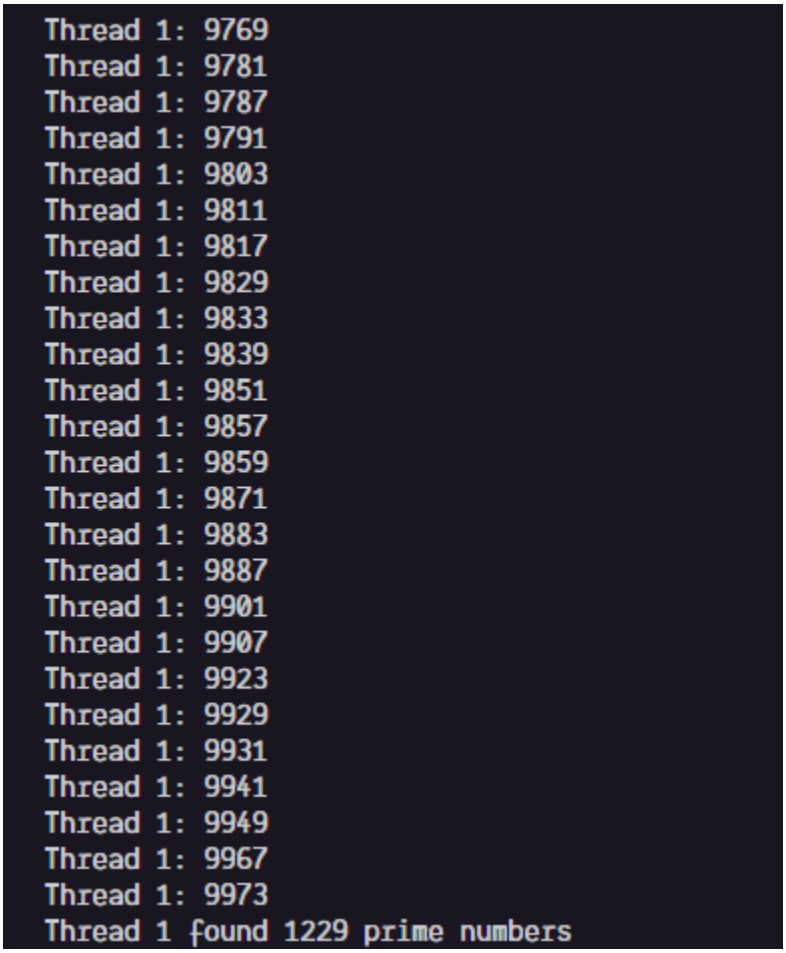
3. Convert the program in (2) so that each thread returns the number of prime numbers that it has found using pthread_exit() and for main program to print out the number of prime number that each thread has found.

Code:

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

typedef struct
{
    int start;
    int end;
    int id;
} ThreadData;

void *isPrime(void *arg)
{
    ThreadData *data = (ThreadData *)arg;
    long count = 0; // number of primes found

    for (int i = data->start; i <= data->end; i++)
    {
        if (i < 2)
            continue; // skip 0 and 1

        int isPrime = 1;
        for (int j = 2; j * j <= i; j++)
        {
            if (i % j == 0)
            {
                isPrime = 0;
                break;
            }
        }

        if (isPrime)
        {
            printf("Thread %d: %d\n", data->id, i);
            count++;
        }
    }

    pthread_exit((void *)count);
```

```c
int n;
printf("Enter the number of threads: ");
scanf("%d", &n);

if (n <= 0)
{
    printf("Invalid number of threads.\n");
    return 1;
}

pthread_t threads[n];
ThreadData data[n];

int start = 2;
int range = 10000 / n;

for (int i = 0; i < n; i++)
{
    data[i].start = start;
    data[i].end = (i == n - 1) ? 10000 : start + range - 1;
    data[i].id = i + 1;

    pthread_create(&threads[i], NULL, isPrime, &data[i]);
    start += range;
}

void *retval;

for (int i = 0; i < n; i++)
{
    pthread_join(threads[i], &retval);
    long count = (long)retval; // long because void pointer is also 8 byte
    printf("Thread %d found %ld prime numbers\n", i + 1, count);
}

return 0;
```

Output:



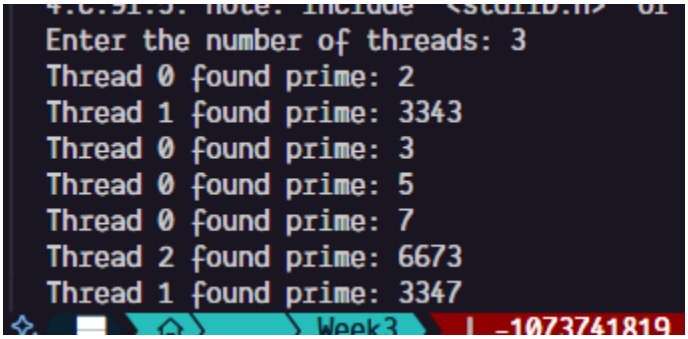
4. Convert the program in (3) to use pthread_cancel() to cancel all threads as soon as the 5th prime number has been found.

Code:

```c
}

int main()
{
    int n;
    printf("Enter the number of threads: ");
    scanf("%d", &n);

    threads = malloc(n * sizeof(pthread_t));
    if (!threads)
        return 1;

    ThreadData data[n];

    int start = 2;
    int range = 10000 / n;

    for (int i = 0; i < n; i++)
    {
        data[i].start = start;
        data[i].end = (i == n - 1) ? 10000 : start + range - 1;
        data[i].id = i;

        pthread_create(&threads[i], NULL, isPrime, &data[i]);
        start += range;
    }

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

    printf("Total primes found (stopped at 5th prime): %d\n", totalPrimes);

    free(threads);
    return 0;
}
```

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

typedef struct
{
    int start;
    int end;
    int id;
} ThreadData;

pthread_t *threads;                                 // Array of threads
int totalPrimes = 0;                                // Shared counter
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; // Mutex to protect totalPrimes

void *isPrime(void *arg)
{
    ThreadData *data = (ThreadData *)arg;

    for (int i = data->start; i <= data->end; i++)
    {
        if (i < 2)
            continue;

        int prime = 1;
        for (int j = 2; j * j <= i; j++)
        {
            if (i % j == 0)
            {
                prime = 0;
                break;
            }
        }

        if (prime)
        {
            printf("Thread %d found prime: %d\n", data->id, i);
```

Output:



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.

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

Account accounts[10];

void *withdraw(void *p) {
    int accountId = *(int *)p;
    double amount = 100;  // Withdraw amount
    accounts[accountId].balance -= amount;
}

void *deposit(void *p) {
    int accountId = *(int *)p;
```

```
    double amount = 100;  // Deposit amount
    accounts[accountId].balance += amount;
}

int main() {
    pthread_t threads[20];
    int ids[10];
    // Create multiple threads to simulate transactions on the same 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]);
    }
    for (int i = 0; i < 20; i++) {
        pthread_join(threads[i], NULL);
    }
    // Print final balance
    for (int i = 0; i < 10; i++) {
        printf("Account %d balance = %.2f\n", i, accounts[i].balance);
    }
}
```

Code:

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

typedef struct
{
    int accountNumber;
    double balance;
    pthread_mutex_t lock; // Mutex for each account
} Account;

Account accounts[10];

void *withdraw(void *p)
{
    int accountId = *(int *)p;
    double amount = 100.0; // Withdraw amount

    pthread_mutex_lock(&accounts[accountId].lock); // Lock before modifyi
    accounts[accountId].balance -= amount;
    printf("Withdrawn %.2f from Account %d | New Balance: %.2f\n",
           amount, accountId, accounts[accountId].balance);
    pthread_mutex_unlock(&accounts[accountId].lock); // Unlock after modi
    return NULL;
}

void *deposit(void *p)
{
    int accountId = *(int *)p;
    double amount = 100.0; // Deposit amount

    pthread_mutex_lock(&accounts[accountId].lock);
    accounts[accountId].balance += amount;
    printf("Deposited %.2f to Account %d | New Balance: %.2f\n",
           amount, accountId, accounts[accountId].balance);
    pthread_mutex_unlock(&accounts[accountId].lock);
    return NULL;
}
```

```
    pthread_mutex_unlock(&accounts[accountId].lock);
    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 = 1000.0; // Initial balance
        pthread_mutex_init(&accounts[i].lock, NULL);
        ids[i] = i;
    }

    // Create multiple threads to simulate concurrent transactions
    for (int i = 0; i < 10; 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 balance
    printf("\nFinal Account Balances:\n");
    for (int i = 0; i < 10; i++)
    {
        printf("Account %d balance = %.2f\n", i, accounts[i].balance);
        pthread_mutex_destroy(&accounts[i].lock); // Clean up mutex
    }

    return 0;
}
```

Output:

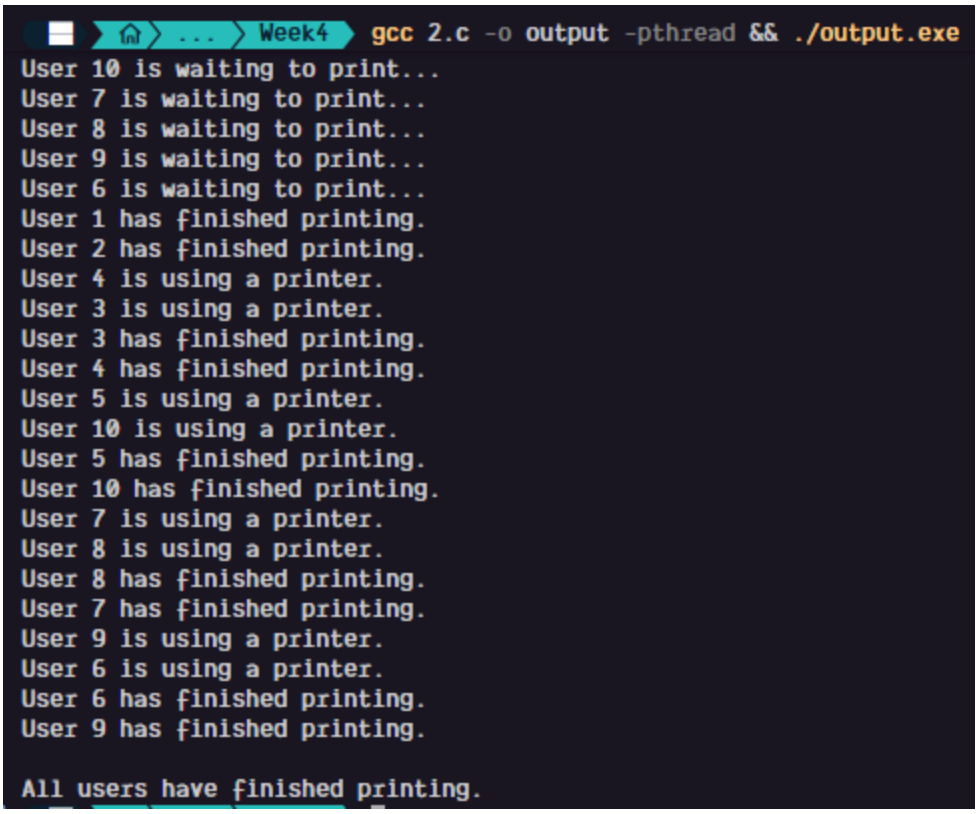
6. A printer is shared among multiple users. Each user can either print a document or wait if the printer is in use. There are only 2 printers in the office. Use **semaphores** to manage the printer access, ensuring that no more than 2 users can print at the same time.

Code:

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

#define NUM_USERS 10   // Number of users trying to print
#define NUM_PRINTERS 2 // Only 2 printers available

sem_t printer_sem; // Semaphore to control printer access

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

    printf("User %d is waiting to print...\n", userId);

    // Wait until a printer is available
    sem_wait(&printer_sem);

    printf("User %d is using a printer.\n", userId);
    sleep(rand() % 3 + 1); // Simulate printing time (1-3 seconds)
    printf("User %d has finished printing.\n", userId);

    // Release the printer (increment semaphore)
    sem_post(&printer_sem);

    return NULL;
}

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

    // Initialize semaphore with value = number of printers
    sem_init(&printer_sem, 0, NUM_PRINTERS);

    srand(time(NULL));

    // Create user threads
```

```c
{
    pthread_t threads[NUM_USERS];
    int userIds[NUM_USERS];

    // Initialize semaphore with value = number of printers
    sem_init(&printer_sem, 0, NUM_PRINTERS);

    srand(time(NULL));

    // Create user threads
    for (int i = 0; i < NUM_USERS; i++)
    {
        userIds[i] = i + 1;
        pthread_create(&threads[i], NULL, print_document, &userIds[i]);
    }

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

    // Destroy semaphore
    sem_destroy(&printer_sem);

    printf("\nAll users have finished printing.\n");
    return 0;
}
```

Output:


... Week4 gcc 2.c -o output -pthread && ./output.exe
User 10 is waiting to print...
User 7 is waiting to print...
User 8 is waiting to print...
User 9 is waiting to print...
User 6 is waiting to print...
User 1 has finished printing.
User 2 has finished printing.
User 4 is using a printer.
User 3 is using a printer.
User 3 has finished printing.
User 4 has finished printing.
User 5 is using a printer.
User 10 is using a printer.
User 5 has finished printing.
User 10 has finished printing.
User 7 is using a printer.
User 8 is using a printer.
User 8 has finished printing.
User 7 has finished printing.
User 9 is using a printer.
User 6 is using a printer.
User 6 has finished printing.
User 9 has finished printing.
All users have finished printing.