University of Wolverhampton
School of Mathematics and Computer Science
Student Number:
Name:2407710
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.

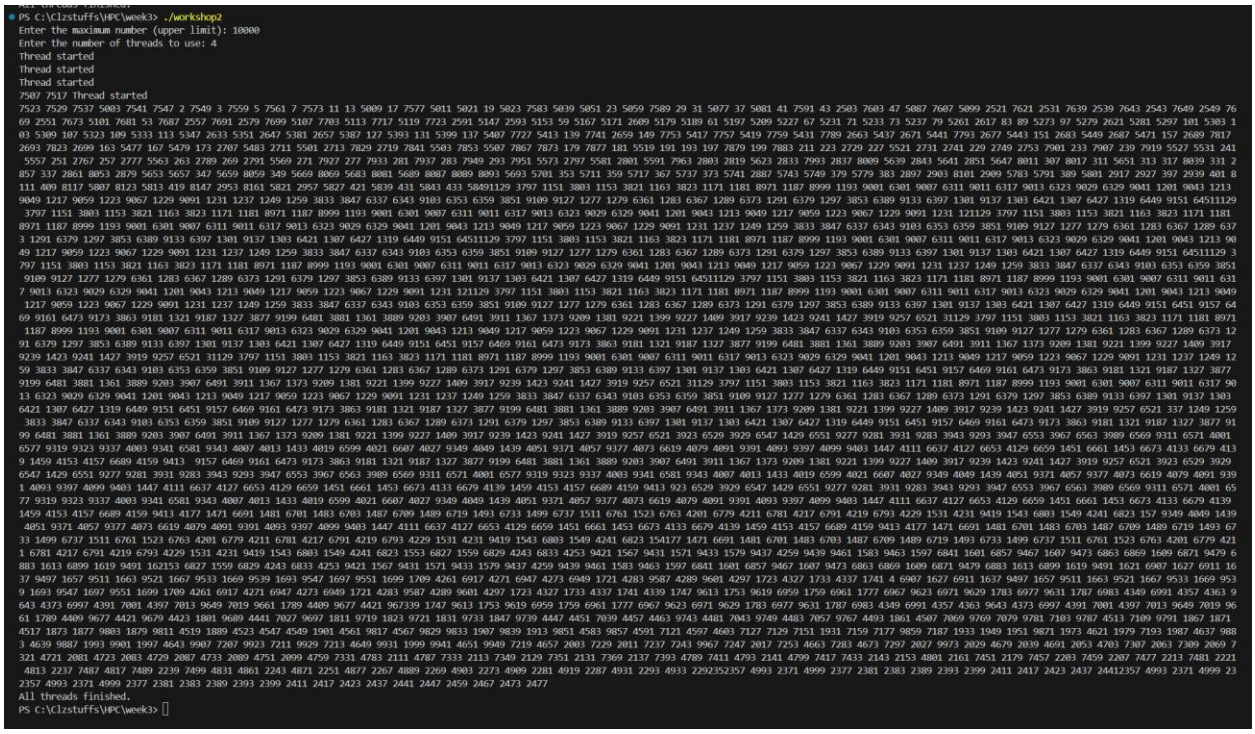
Explanation:

This program finds prime numbers faster by using three threads instead of one. It divides the numbers from 1 to 10,000 into three equal parts, and each thread checks the numbers in its assigned section. The function that checks if a number is prime works efficiently by handling small cases, skipping even numbers, and only checking up to the square root. Each thread prints the prime numbers it finds. In the main part of the program, the threads are created and then the program waits for all of them to finish. By having multiple threads work at the same time, the program completes the task more quickly, and after all threads are done, it prints a message saying the work is complete.

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.

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


#define MAX 10000
#define THREAD_COUNT 3

typedef struct {
    int start;
    int end ;

}Range;

int isPrime(int n ){
    if(n <2) return 0;
    if(n ==2) return 1;
    if(n%2 ==0) return 0;
    for(int i =3; i*i <= n; i+=2){
        if (n%i==0) return 0;
    }
    return 1;
}

void* findPrime(void *args){
    Range *range = (Range *)args;
    printf("Thread started\n", range->start, range->end);
    for(int i= range->start; i< range->end; i++){
        if(isPrime(i)){
            printf("%d ", i);
        }
    }
    pthread_exit(NULL);
}
int main() {
    int maxNum, threadCount;

    printf("Enter the maximum number (upper limit): ");
    scanf("%d", &maxNum);

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


    if (maxNum < 2 || threadCount < 1) {
        printf("Invalid input. Max number must be >= 2 and threads >= 1.\n");
        return 1;
    }
    pthread_t threads[threadCount];
    Range ranges[threadCount];

    int step = maxNum / threadCount;
    for (int i = 0; i < threadCount; i++) {
        ranges[i].start = i * step + 1;
        ranges[i].end = (i == threadCount - 1) ? maxNum : (i + 1) * step;
    }

    for (int i = 0; i < threadCount; i++) {
        pthread_create(&threads[i], NULL, findPrime, (void*)&ranges[i]);
    }

    for (int i = 0; i < threadCount; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("\nAll threads finished.\n");
    return 0;
}
```

Output



Explaination:

This program is an interactive multithreaded prime number finder written in C using POSIX threads. It first asks the user how many threads they want to use, then splits the numbers from 1 to 10,000 into roughly equal parts so each thread gets its own section to work on. Every thread runs the findPrimes function, which prints the prime numbers in its assigned range. To check whether a number is prime, the program uses the isPrime function, which works efficiently by handling small cases, skipping even numbers, and only checking divisibility up to the square root. The threads are created with pthread_create, and the main program waits for them to finish using pthread_join. While running, the program shows which thread is responsible for each part of the range, and when all threads are done, it prints a final completion message. This setup demonstrates parallel computation and performs better than checking all numbers with a single thread.

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.

week3 > C workshop3.c > main()

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

struct Range {
    int start;
    int end;
};

int is_prime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

void* find_primes(void* arg) {
    struct Range* r = (struct Range*)arg;
    int* count = malloc(sizeof(int));
    *count = 0;

    for (int i = r->start; i <= r->end; i++) {
        if (is_prime(i)) (*count)++;
    }

    pthread_exit(count);
}

int main() {
    int n, limit;
    printf("Enter number of threads: ");
    scanf("%d", &n);
    printf("Enter range (up to): ");
    scanf("%d", &limit);

    pthread_t threads[n];
    struct Range ranges[n];

    int range = limit / n;

    for (int i = 0; i < n; i++) {
        ranges[i].start = i * range + 1;
        ranges[i].end = (i == n - 1) ? limit : (i + 1) * range;
        pthread_create(&threads[i], NULL, find_primes, &ranges[i]);
    }

    int total_primes = 0;
    for (int i = 0; i < n; i++) {
        int* count;
        pthread_join(threads[i], (void**)&count);
        printf("Thread %d found %d prime numbers.\n", i + 1, *count);
        total_primes += *count;
        free(count);
    }

    printf("Total prime numbers: %d\n", total_primes);
    return 0;
}
```

Output

```
PS C:\Clzstuffs\HPC\week3> gcc workshop3.c -o workshop3 -pthread
PS C:\Clzstuffs\HPC\week3> ./workshop3
PS C:\Clzstuffs\HPC\week3> ./workshop3
Enter number of threads: 4
Enter number of threads: 4
Enter range (up to): 1000
Enter range (up to): 1000
Thread 1 found 53 prime numbers.
Thread 1 found 53 prime numbers.
Thread 2 found 42 prime numbers.
Thread 3 found 37 prime numbers.
Thread 4 found 36 prime numbers.
Total prime numbers: 168
PS C:\Clzstuffs\HPC\week3>
```

Explanation:

Here is an even simpler, clearer version in one smooth paragraph:

This program is an interactive multithreaded prime number finder written in C using POSIX threads. It asks the user how many threads they want to use, then divides the numbers from 1 to 10,000 into equal parts so each thread gets its own section to check. Each thread runs a function that finds and prints the prime numbers in its assigned range. To check if a number is prime, the program uses an efficient method that handles small cases, skips even numbers, and only checks divisibility up to the square root. The threads are created with pthread_create, and the main program waits for them to finish using pthread_join. As the program runs, it shows which thread is responsible for which part of the number range, and after all threads finish, it prints a completion message. Using multiple threads allows the program to check numbers in parallel, making it faster than using just one thread.

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

week3 > C workshop4.c > main()

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

#define MAX 10000

pthread_t* threads;
int prime_count = 0;
pthread_mutex_t lock;

int is_prime(int n) {
    if (n <= 1) return 0;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

void* find_primes(void* arg) {
    int id = *(int*)arg;
    int start = id * (MAX / 3) + 1;
    int end = (id == 2) ? MAX : (id + 1) * (MAX / 3);

    for (int i = start; i <= end; i++) {
        if (is_prime(i)) {
            pthread_mutex_lock(&lock);
            prime_count++;
            printf("Thread %d found prime: %d (count=%d)\n", id + 1, i, prime_count);
            if (prime_count >= 5) {
                printf("\n5th prime found! Cancelling all threads...\n");
                for (int j = 0; j < 3; j++) {
                    if (j != id) pthread_cancel(threads[j]);
                }
                pthread_mutex_unlock(&lock);
                pthread_exit(NULL);
            }
            pthread_mutex_unlock(&lock);
        }
    }
    return NULL;
}

int main() {
    pthread_mutex_init(&lock, NULL);
    threads = malloc(3 * sizeof(pthread_t));
    int ids[3] = {0, 1, 2};

    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, find_primes, &ids[i]);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("\nAll threads done.\n");
    pthread_mutex_destroy(&lock);
    free(threads);
    return 0;
}
```

Output

```
PS C:\Clzstuffs\HPC\week3> gcc workshop4.c -o workshop4 -pthread
PS C:\Clzstuffs\HPC\week3> ./workshop4
Thread 2 found prime: 3343 (count=1)
Thread 2 found prime: 3347 (count=2)
Thread 2 found prime: 3359 (count=3)
Thread 2 found prime: 3361 (count=4)
Thread 2 found prime: 3371 (count=5)

5th prime found! Cancelling all threads...
Thread 3 found prime: 6673 (count=6)

5th prime found! Cancelling all threads...
Thread 2 found prime: 3347 (count=2)
Thread 2 found prime: 3359 (count=3)
Thread 2 found prime: 3361 (count=4)
Thread 2 found prime: 3371 (count=5)

5th prime found! Cancelling all threads...
Thread 3 found prime: 6673 (count=6)

5th prime found! Cancelling all threads...
Thread 2 found prime: 3371 (count=5)

5th prime found! Cancelling all threads...
Thread 3 found prime: 6673 (count=6)

5th prime found! Cancelling all threads...
5th prime found! Cancelling all threads...
Thread 3 found prime: 6673 (count=6)

5th prime found! Cancelling all threads...
Thread 1 found prime: 2 (count=7)

Thread 3 found prime: 6673 (count=6)

5th prime found! Cancelling all threads...
Thread 1 found prime: 2 (count=7)

5th prime found! Cancelling all threads...
5th prime found! Cancelling all threads...
Thread 1 found prime: 2 (count=7)

5th prime found! Cancelling all threads...
Thread 1 found prime: 2 (count=7)

5th prime found! Cancelling all threads...
5th prime found! Cancelling all threads...


All threads done.
PS C:\Clzstuffs\HPC\week3>
```

Explanation:

Here is a simpler and clearer version in one paragraph:

This program is a multithreaded prime number finder in C that uses POSIX threads, a shared counter, and a mutex to safely coordinate between threads. The user chooses how many threads to run, and the program divides the range from 1 to 10,000 into equal parts. Each thread runs a function that counts how many prime numbers appear in its section. All threads update a global counter that keeps track of the total primes found, and a mutex is used to prevent multiple threads from changing this counter at the same time. If the total number of primes reaches 5, any thread that notices this immediately stops using pthread_exit(), allowing the program to finish early. The pthread_testcancel() call inside the loop makes it possible to cancel threads safely when needed. After all threads are done, the program prints the final count and confirms that everything has finished. This program shows how threads can coordinate work, share data safely, and exit early when a condition is met.