Advanced C Programming - Assignment 2

Name	T K Gowtham
Email ID	gowthamkamalasekar@gmail.com
College	VIT Chennai

1. Write a C program to define 3 different threads with the following purposes where N is the input
    - Thread A - To run a loop and return the sum of first N prime numbers
    - Thread B & C - should run in parallel. One prints "Thread 1 running" every 2 seconds, and the other prints "Thread 2 running" every 3 seconds for 100 seconds.

Code :

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

int N;

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

```c
void* threadA(void* arg) {
    int count = 0, num = 2, sum = 0;
    while (count < N) {
        if (is_prime(num)) {
            sum += num;
            count++;
        }
        num++;
    }
    printf("Sum of first %d prime numbers: %d\n", N, sum);
    return NULL;
}

void* threadB(void* arg) {
    time_t start = time(NULL);
    while (time(NULL) - start < 100) {
        printf("Thread 1 running\n");
        sleep(2);
    }
    return NULL;
}

void* threadC(void* arg) {
    time_t start = time(NULL);
    while (time(NULL) - start < 100) {
        printf("Thread 2 running\n");
        sleep(3);
    }
    return NULL;
}

int main() {
    pthread_t tA, tB, tC;

    printf("Enter N: ");
    scanf("%d", &N);

    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);
    pthread_create(&tC, NULL, threadC, NULL);
```

```
    pthread_join(tA, NULL);
    pthread_join(tB, NULL);
    pthread_join(tC, NULL);

    return 0;
}
```

Output :

```
PS C:\Users\gowth\OneDrive\Desktop\EmbedUR\C Programming\Advanced> cd "c:\Users\gowth\OneDrive\
Desktop\EmbedUR\C Programming\Advanced\Assingment 2\" ; if ($?) { gcc Q1.c -o Q1 } ; if ($?) {
.\Q1 }
Enter N: 100
Sum of first 100 prime numbers: 24133
Thread 1 running
Thread 2 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 1 running
Thread 2 running
```

2. In the above program,

   - add signal handling for SIGINT (etc) and prevent termination.
   - Convert the above threads to individual functions and note down the time taken and the flow of execution.

Code :

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

volatile sig_atomic_t keep_running = 1;

void handle_sigint(int sig) {
    printf("\nSIGINT received. Preventing termination. Press Ctrl+\\ to
force quit.\n");
}

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

void* threadA(void* arg) {
    int N = *((int*)arg);
    int count = 0, num = 2, sum = 0;

    clock_t start = clock();

    while (count < N) {
        if (is_prime(num)) {
            sum += num;
            count++;
```

```
        }
        num++;
    }

    clock_t end = clock();
    double duration = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Thread A: Sum of first %d primes = %d\n", N, sum);
    printf("Thread A completed in %.2f seconds.\n", duration);
    pthread_exit(NULL);
}

void* threadB(void* arg) {
    time_t start = time(NULL);
    while (keep_running && time(NULL) - start < 100) {
        printf("Thread B running\n");
        sleep(2);
    }
    pthread_exit(NULL);
}

void* threadC(void* arg) {
    time_t start = time(NULL);
    while (keep_running && time(NULL) - start < 100) {
        printf("Thread C running\n");
        sleep(3);
    }
    pthread_exit(NULL);
}

int main() {
    signal(SIGINT, handle_sigint);

    int N;
    printf("Enter N: ");
    scanf("%d", &N);

    pthread_t t1, t2, t3;

    time_t overall_start = time(NULL);
```

```c
    pthread_create(&t1, NULL, threadA, &N);
    pthread_create(&t2, NULL, threadB, NULL);
    pthread_create(&t3, NULL, threadC, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    time_t overall_end = time(NULL);
    printf("All threads completed in %ld seconds.\n", overall_end -
overall_start);

    return 0;
}
```

Output :

```
PS C:\Users\gowth\OneDrive\Desktop\EmbedUR\C Programming\Advanced\Assingment 2> cd "c:\Users\go
wth\OneDrive\Desktop\EmbedUR\C Programming\Advanced\Assingment 2\" ; if ($?) { gcc Q2.c -o Q2 }
 ; if ($?) { .\Q2 }
Enter N: 1000
Thread B running
Thread A: Sum of first 1000 primes = 3682913
Thread A completed in 0.00 seconds.
Thread C running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
```

```
Thread B running
Thread C running
Thread B running
Thread B running

SIGINT received. Preventing termination. Press Ctrl+\ to force quit.
Thread C running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
```

```
Thread C running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
Thread B running
Thread B running
Thread C running
All threads completed in 102 seconds.
PS C:\Users\gowth\OneDrive\Desktop\EmbedUR\C Programming\Advanced\Assingment 2>
```

3. Know about the following topics and explore them (Write a note on your understandings)

Areas for exploration,

- Child process - fork()
- Handing common signals
- Exploring different Kernel crashes
- Time complexity
- Locking mechanism - mutex/spinlock

a. Child process - fork() :
   - It creates a new process called child from the parent.
   - After fork(), both processes run independently from the same point.
   - It can return values such as 0 → inside child process, greater than 0 → inside the parent process, less than 0 → error.
   - It is primarily used in multitasking, servers, etc.
   - Syntax Example :

```
pid_t pid = fork();
if(pid == 0) printf("Child \n");
else printf("Parent \n");
```

b. Handling common signals
   - Signals are used to communicate with processes.
   - Common signals are
     - SIGINT → Ctrl + C
     - SIGTERM → Termination Request
     - SIGKILL → force kill (cannot be caught)
     - Using signal(SIGINT, handler_func); to handle.
     - sigaction() is more powerful than signal()
   - Syntax Example :

```
Void handler(int sig) {
        printf("singal caught is %d",sig);
}

signal(SIGINT, handler);
```

c. Exploring different Kernel Crashes :
   i. Kernel crashes are basically panic or segmentation faults in kernel space.
   ii. Common Causes of Kernel Crashes are Null pointer dereference, invalid memory access, stack overflow, bad device driver code.
   iii. Tools used for handling the kernel crashes are dmesg, journalctl, crash dumps(kexec, kdump), QEMU with debug symbols.

d. Time Complexity :
   i. It is the measure of how fast an algorithm grows with input size n.
   ii. Common time complexity are :
      1. O(n) → Linear
      2. O(1) → Constant
      3. O(log n) → logarithmic
      4. O(n^2) → quadratic
      5. O(2^n) → exponential
   iii. Efficiency may vary machine to machine for the same algorithm based on various parameters which will make finding the efficiency of an algorithm difficult.
   iv. In order to tackle this, we analyze the algorithm to approximate the efficiency for all machines in general.

e. Locking Mechanism - Mutex/Spinlock :

   i. Both prevent race case conditions i.e. when multiple threads are accessing shared memory.
   ii. Blocking :
      1. Mutex : If the lock is already taken by another thread, the current thread gets blocked (sleep) and waits for its turn. It does not consume CPU while waiting.
      2. Spinlock : If the lock is already taken the thread will keep checking in a loop (busy-wait) until it's free. So, it consumes CPU continuously during this wait.
      3. So Mutex is efficient for longer waits and spinlock is only good if the wait is shorter.
   iii. Use Cases :
      1. Mutex : Used in user space programs like normal C program with pthread, and it is suitable for long or unpredictable critical sections.
      2. Spinlock : mostly used in kernel space where context switches are expensive, and the critical section is very small and fast.
      3. We shouldn’t use spinlocks in normal apps unless you know exactly what the program is doing.
   iv. Overhead :
      1. Mutex : Less CPU usage because it sleeps instead of looping but takes time to wake up, so it is a bit slower if the lock is held very briefly.

2. Spinlock : High CPU usage because it spins, but responds instantly once the lock is released.