Advance C Programming

Module 2 -Assignment

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.

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

// Function to check if a number is prime
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;
}

// Thread A: Sum of first N prime numbers
void* threadA(void* arg) {
    int N = *(int*)arg;
    int sum = 0, count = 0, num = 2;
    while (count < N) {
        if (is_prime(num)) {
            sum += num;
            count++;
```

```
        }
        num++;
    }
    printf("Sum of first %d prime numbers: %d\n", N, sum);
    return NULL;
}
// Thread B: Print every 2 seconds
void* threadB(void* arg) {
    for (int i = 0; i < 50; i++) { // 100 seconds / 2 = 50 times
        printf("Thread 1 running\n");
        sleep(2);
    }
    return NULL;
}
// Thread C: Print every 3 seconds
void* threadC(void* arg) {
    for (int i = 0; i < 33; i++) { // 100 seconds / 3 ≈ 33 times
        printf("Thread 2 running\n");
        sleep(3);
    }
    return NULL;
}
int main() {
    pthread_t t1, t2, t3;
    int N;

    printf("Enter the value of N: ");
    scanf("%d", &N);
```

```
    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);
    printf("All threads completed.\n");
    return 0;
}
```

```
Enter the value of N: 5
Sum of first 5 prime numbers: 28
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
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 1 running
Thread 2 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 1 running
Thread 2 running
```

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.

```
#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. Ignoring termination request.\n");

   keep_running = 0; // Can be used to gracefully stop threads if needed

}


// Function to check if a number is prime

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;

}


// Thread A: Sum of first N prime numbers

void* threadA(void* arg) {
```

```
    int N = *(int*)arg;
    int sum = 0, count = 0, num = 2;
    clock_t start = clock();

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

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

    printf("Thread A: Sum of first %d prime numbers is %d (Time: %.2f sec)\n", N, sum,
time_taken);
    return NULL;
}

// Thread B: Prints every 2 seconds for 100 seconds
void* threadB(void* arg) {
    clock_t start = clock();
    time_t t_start = time(NULL);
    while (difftime(time(NULL), t_start) < 100 && keep_running) {
        printf("Thread 1 running\n");
        sleep(2);
    }
```

```
    clock_t end = clock();
    printf("Thread B completed (Time: %.2f sec)\n", (double)(end - start) / CLOCKS_PER_SEC);
    return NULL;
}

// Thread C: Prints every 3 seconds for 100 seconds
void* threadC(void* arg) {
    clock_t start = clock();
    time_t t_start = time(NULL);
    while (difftime(time(NULL), t_start) < 100 && keep_running) {
        printf("Thread 2 running\n");
        sleep(3);
    }
    clock_t end = clock();
    printf("Thread C completed (Time: %.2f sec)\n", (double)(end - start) / CLOCKS_PER_SEC);
    return NULL;
}

int main() {
    signal(SIGINT, handle_sigint); // Handle Ctrl+C gracefully

    pthread_t tA, tB, tC;
    int N;

    printf("Enter the value of N (for prime sum): ");
    scanf("%d", &N);

    pthread_create(&tA, NULL, threadA, &N);
```

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

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

    printf("All threads have completed execution.\n");
    return 0;
}
```

```
Enter the value of N (for prime sum): 3
Thread A: Sum of first 3 prime numbers is 10 (Time: 0.00 sec)
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 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 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
```

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

Areas for exploration,

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

1. Child Process – fork()

- fork() is a system call in UNIX/Linux used to **create a new process** (child) from the current process (parent).
- After fork(), both parent and child run the same program independently.
- Returns:
    - 0 in the child process,
    - Process ID (PID) of the child in the parent,
    - -1 on failure.
- **Use case**: Running separate processes in parallel.

2. Handling Common Signals

- Signals are **asynchronous notifications** sent to a process (e.g., SIGINT, SIGTERM, SIGKILL).
- Can be handled using signal() or sigaction().
- Common signals:
    - SIGINT: interrupt from keyboard (Ctrl+C)
    - SIGTERM: termination request
    - SIGKILL: forceful termination (cannot be caught)
    - SIGSEGV: segmentation fault
- You can define custom handlers to clean up or ignore termination.

3. Exploring Different Kernel Crashes

- Kernel crashes (also called **kernel panics**) occur when the OS encounters an unrecoverable error.

- Causes:
    - Null pointer dereference
    - Hardware failure
    - Unsafe kernel module operations
- Tools to explore:
    - dmesg (kernel log messages)
    - Crash dumps (/var/crash)
- Programmers must avoid unsafe operations in kernel-space code to prevent crashes.

4. Time Complexity

- Describes the **computational cost** of an algorithm as input size increases.
- Big-O Notation:
    - O(1) – constant time
    - O(log N) – logarithmic
    - O(N) – linear
    - O(N^2) – quadratic
- Helps compare algorithms and optimize performance.

5. Locking Mechanisms – mutex / spinlock

- Used in **multithreaded programs** to avoid race conditions.

mutex (Mutual Exclusion)

- A locking mechanism that **blocks** other threads until the lock is released.
- Good for general-purpose synchronization.

spinlock

- A low-level lock that **spins in a loop** until the lock is available.
- Does **not sleep**, so it's faster but CPU-intensive.
- Used when lock hold time is expected to be short.