ADVANCE C PROGRAMMING TRAINING MODULE 2 ASSIGNMENT SOLUTION

-BY SAKTHI KUMAR S

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.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.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: Calculate sum of first N prime numbers
void* threadA_func(void* arg) {
    int N = *(int*)arg;
    int count = 0, num = 2, sum = 0;
    while (count < N) {
        if (is_prime(num)) {
            sum += num;
            count++;
        }
        num++;
    }
    printf("Thread A: Sum of first %d prime numbers is %d\n", N, sum);
    pthread_exit(NULL);
}
// Thread B: Print message every 2 seconds for 100 seconds
void* threadB_func(void* arg) {
    time_t start = time(NULL);
    while (time(NULL) - start < 100) {
        printf("Thread 1 running\n");
        sleep(2);
    }
    pthread_exit(NULL);
```

```c
}
// Thread C: Print message every 3 seconds for 100 seconds
void* threadC_func(void* arg) {
    time_t start = time(NULL);
    while (time(NULL) - start < 100) {
        printf("Thread 2 running\n");
        sleep(3);
    }
    pthread_exit(NULL);
}

int main() {
    int N;
    printf("Enter value for N: ");
    scanf("%d", &N);

    pthread_t threadA, threadB, threadC;

    // Create threads
    pthread_create(&threadA, NULL, threadA_func, &N);
    pthread_create(&threadB, NULL, threadB_func, NULL);
    pthread_create(&threadC, NULL, threadC_func, NULL);

    // Wait for threads to finish
    pthread_join(threadA, NULL);
    pthread_join(threadB, NULL);
    pthread_join(threadC, NULL);

    return 0;
}
```

OUTPUT:

```
Enter value for N: 5
Thread A: Sum of first 5 prime numbers is 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 2 running
Thread 1 running
Thread 1 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

PROGRAM:

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

// Global control flag for threads
volatile sig_atomic_t keep_running = 1;

// Signal handler for SIGINT (Ctrl + C)
void sigint_handler(int sig) {
    printf("\nSIGINT received. Ignoring and continuing execution...\n");
    keep_running = 0;
}

// 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: Computes sum of first N prime numbers
void* threadA_func(void* arg) {
    int N = *(int*)arg;
    int count = 0, num = 2, sum = 0;

    printf("Thread A: Starting computation of prime numbers...\n");

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

    if (keep_running)
```

```c
        printf("Thread A: Sum of first %d prime numbers is %d\n", N, sum);
    else
        printf("Thread A: Interrupted. Partial sum = %d\n", sum);

    pthread_exit(NULL);
}

// Thread B: Prints message every 2 seconds for 100 seconds
void* threadB_func(void* arg) {
    time_t start = time(NULL);
    printf("Thread B: Started...\n");

    while (keep_running && time(NULL) - start < 100) {
        printf("Thread 1 running\n");
        sleep(2);
    }

    printf("Thread B: Finished.\n");
    pthread_exit(NULL);
}

// Thread C: Prints message every 3 seconds for 100 seconds
void* threadC_func(void* arg) {
    time_t start = time(NULL);
    printf("Thread C: Started...\n");

    while (keep_running && time(NULL) - start < 100) {
        printf("Thread 2 running\n");
        sleep(3);
    }

    printf("Thread C: Finished.\n");
    pthread_exit(NULL);
}

int main() {
    int N;
    pthread_t threadA, threadB, threadC;

    // Set up signal handler
    signal(SIGINT, sigint_handler);

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

    // Start real-time tracking
    time_t begin = time(NULL);
```

```c
    // Create threads
    pthread_create(&threadA, NULL, threadA_func, &N);
    pthread_create(&threadB, NULL, threadB_func, NULL);
    pthread_create(&threadC, NULL, threadC_func, NULL);

    // Wait for threads to finish
    pthread_join(threadA, NULL);
    pthread_join(threadB, NULL);
    pthread_join(threadC, NULL);

    // End time and calculate duration
    time_t end = time(NULL);
    double elapsed_time = difftime(end, begin);
    printf("Main: Total execution time = %.2f seconds\n", elapsed_time);

    return 0;
}
```

OUTPUT:

```
Enter the value of N (for prime sum): 10
Thread B: Started...
Thread 1 running
Thread C: Started...
Thread 2 running
Thread A: Starting computation of prime numbers...
Thread A: Sum of first 10 prime numbers is 129
Thread 1 running
Thread 2 running
Thread 1 running
Thread 2 running
Thread 1 running
Thread 1 running
Thread 2 running
Thread 1 running
^C
SIGINT received. Ignoring and continuing execution...
Thread C: Finished.
Thread B: Finished.
Main: Total execution time = 12.00 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

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

Areas for exploration,

1. Child Process – fork()

- fork() is a system call in Unix/Linux that creates a new process by duplicating the calling process.
- The new process is called a child, and the original is the parent.
- After a successful fork():
  - The parent receives the child's PID.
  - The child receives 0.
- If return value is **> 0 → Parent**, **== 0 → Child**, **< 0 → Error (fork failed)**
- Both processes continue executing from the point where fork() was called, but independently.

Example:

```
pid_t pid = fork();

   if (pid == 0) {
       printf("Child Process\n");
   } else if (pid > 0) {
       printf("Parent Process, Child PID = %d\n",
pid);
   } else {
       printf("Process Failed\n");
   }
```

2. Handling Common Signals

- Signals are asynchronous notifications sent to a process to notify it of events (e.g., termination request, invalid memory access).

Common Signals:

- SIGINT - Ctrl+C (interrupt)
- SIGTERM - Terminate gracefully
- SIGKILL - Force kill (cannot be caught)
- SIGSEGV - Invalid memory access (segmentation fault)
- SIGFPE- Floating Point Exception

- SIGILL- Illegal Instruction
- SIGABRT- Abort

Signal Handling: using the signal() or sigaction() function.

Example:

```
void handler(int sig) {
   printf("Caught signal %d\n", sig);
}
signal(SIGINT, handler);
```

This allows program to intercept and handle signals instead of terminating unexpectedly.

3. Exploring Different Kernel Crashes

Kernel crashes (also called kernel panics) occur when the Linux kernel encounters a fatal error it cannot recover from.

- Common Causes: Faulty drivers, Null pointer dereference in kernel space, Buffer overflows, Race conditions, Stack overflows, Deadlocks
- Symptoms: System freeze or reboot and Panic logs in /var/log/kern.log or dmesg
- Tools for Analysis: dmesg: Check kernel ring buffer, kdump: Capture kernel crash dumps, gdb with kernel symbols: Debug the crash
- Preventive Measures: By Using proper locking mechanism, by Validating memory access and by Testing kernel modules thoroughly

4. Time Complexity

Time complexity measures how the runtime of an algorithm increases with input size. Analysing Time complexity helps in selecting the most efficient algorithm for large input sizes and optimizing program performance.

Common Notations:

- O(1) – Constant time
- O(log n) – Logarithmic time
- O(n) – Linear time
- O(n^2) – Quadratic time
- O(2^n) – Exponential time

Example:
A loop running n times is O(n).
A nested loop on n elements is $O(n^2)$.

5. Mutex/Spinlock:

Feature	Mutex	Spinlock
Waiting Behaviour	Puts thread to sleep if lock is unavailable	Busy-waits (keeps checking until lock is free)
CPU Usage While Waiting	Low (since thread sleeps)	High (uses CPU cycles while spinning)
Context Switching	Yes (can cause overhead)	No (remains in same context)
Lock Hold Duration	Suitable for longer critical sections	Suitable for very short critical sections
Fairness	Can be fair (depends on implementation)	May cause starvation (no queuing)
Power Consumption	Lower	Higher (CPU is active while spinning)
Performance	Better for multitasking systems	Better for real-time or low-latency tasks
Use Case Example	File access, I/O operations	Low-level kernel locks, short ops in SMP systems