Module 2 Assignment

Task 1: C Program with 3 Threads (Prime sum, periodic messages)

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
int is_prime(int num) {
    if (num <= 1) 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;

    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);
}
void* threadB(void* arg) {
    int elapsed = 0;
    while (elapsed < 100) {
```

```
        printf("Thread 1 running\n");
        sleep(2);
        elapsed += 2;
    }
    pthread_exit(NULL);
}

void* threadC(void* arg) {
    int elapsed = 0;
    while (elapsed < 100) {
        printf("Thread 2 running\n");
        sleep(3);
        elapsed += 3;
    }
    pthread_exit(NULL);
}
int main() {
    int N;
    printf("Enter the value of N: ");
    scanf("%d", &N);
    pthread_t t1, t2, t3;
    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;
}
```

Code Snippet:

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

int is_prime(int num) {
    if (num <= 1) 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;

    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);
}

void* threadB(void* arg) {
    int elapsed = 0;
    while (elapsed < 100) {
        printf("Thread 1 running\n");
        sleep(2);
        elapsed += 2;
    }
    pthread_exit(NULL);
}

void* threadC(void* arg) {
    int elapsed = 0;
    while (elapsed < 100) {
        printf("Thread 2 running\n");
        sleep(3);
        elapsed += 3;
    }
    pthread_exit(NULL);
}

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

    pthread_t t1, t2, t3;

    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;
}
```

Output Snippet:

```
Enter the value of N: 20
Thread A: Sum of first 20 prime numbers is 639
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 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
```

Task 2: C Program with 3 Threads (Prime sum, periodic messages) + SIGINT

Code:

```
#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...\n");
    keep_running = 0;
}

int is_prime(int num) {
    if (num <= 1) 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 time_taken = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Thread A: Sum of first %d prime numbers is %d (time: %.2fs)\n", N, sum,
time_taken);
    pthread_exit(NULL);
}

void* threadB(void* arg) {
    int elapsed = 0;
    while (elapsed < 100 && keep_running) {
        printf("Thread 1 running\n");
        sleep(2);
        elapsed += 2;
    }
    pthread_exit(NULL);
}

void* threadC(void* arg) {
    int elapsed = 0;
    while (elapsed < 100 && keep_running) {
        printf("Thread 2 running\n");
        sleep(3);
        elapsed += 3;
    }
```

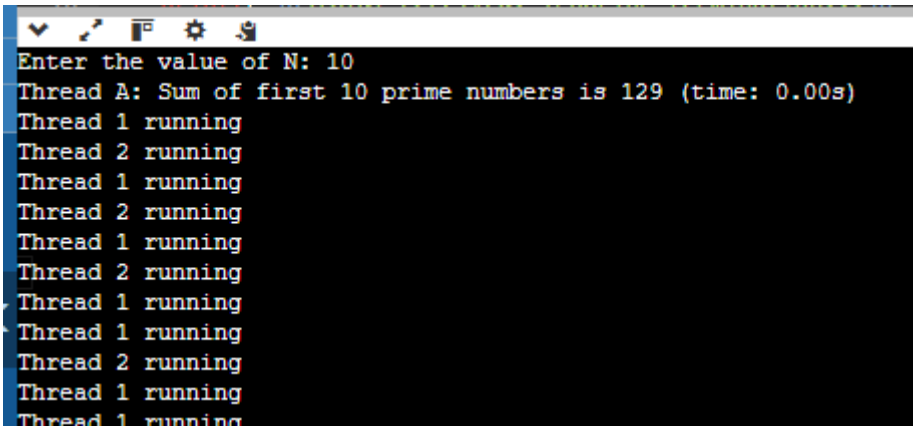
```
    pthread_exit(NULL);
}

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

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

    pthread_t t1, t2, t3;

    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;
}
```

Code Snippet:

main.c

```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. Ignoring termination...\n");
    keep_running = 0;
}
int is_prime(int num) {
    if (num <= 1) 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 time_taken = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Thread A: Sum of first %d prime numbers is %d (time: %.2fs)\n", N, sum, time_taken);
    pthread_exit(NULL);
}
void* threadB(void* arg) {
    int elapsed = 0;
    while (elapsed < 100 && keep_running) {
        printf("Thread 1 running\n");
        sleep(2);
        elapsed += 2;
    }
    pthread_exit(NULL);}
void* threadC(void* arg) {
    int elapsed = 0;
    while (elapsed < 100 && keep_running) {
        printf("Thread 2 running\n");
        sleep(3);
        elapsed += 3;
    }
    pthread_exit(NULL);}
int main() {
    signal(SIGINT, handle_sigint);
    int N;
    printf("Enter the value of N: ");
    scanf("%d", &N);
    pthread_t t1, t2, t3;
    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;
}
```

Output Snippet:



Task 3: Notes on Topics

1. Child Process – fork()

The fork() system call in C is used to create a new process by duplicating the current process. The new process is called the child, and the original is the parent.

- Syntax: pid_t pid = fork();
- Return Values:
  - > 0 → Parent process (returns child PID)
  - = 0 → Child process
  - < 0 → Error (process creation failed)

Example:

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

Use Cases: Creating background tasks, process pools, handling multitasking in UNIX.

2. Handling Common Signals

Signals are software interrupts delivered to a process to notify it of events.

- Common signals:
  - SIGINT: Interrupt from keyboard (Ctrl + C)
  - SIGTERM: Termination request
  - SIGKILL: Kill the process forcefully
  - SIGSEGV: Segmentation fault

Signal Handling in C:

```
#include <signal.h>
void handle_sigint(int sig) {
        printf("Caught SIGINT\n");
}
```

```
int main() {
        signal(SIGINT, handle_sigint);
        while (1);
}
```

Use Cases: Clean exit, releasing memory, stopping threads gracefully.

3. Exploring Kernel Crashes

Kernel crashes are serious errors that occur in the core of an operating system.

Causes:

- Illegal memory access
- Bad device drivers
- Stack overflow in kernel space
- Hardware failure

How to Explore:

- Use dmesg to view kernel logs.
- Check /var/log/kern.log or /var/log/syslog
- Enable kernel crash dumps with kdump in Linux

Debugging Tools:

- gdb
- crash
- Kernel logging with printk()

Preventive Practice: Write safe kernel modules, avoid null pointer dereferences.

4. Time Complexity

Time complexity expresses the runtime growth of an algorithm in terms of input size N.

- Big O Notation Examples:
    - O(1) – Constant (e.g., accessing an array element)
    - O(n) – Linear (e.g., for loop over N elements)
    - $O(n^2)$ – Quadratic (e.g., nested loops)

Use Case in Thread A (Prime Calculation):

```
while (count < N) {
// Check if num is prime
}
```

- Time Complexity: Roughly O(N√M) where M is the Nth prime candidate.
- Helps in optimizing code.
- Ensures scalability and better performance.

5. Locking Mechanism – mutex and spinlock

Feature	Mutex	Spinlock
Blocking	Yes	No (spins)
CPU Usage	Low	High
Context Switch	Yes	No
Best For	User space	Kernel / Low wait time

Locks are used to synchronize access to shared data in multi-threaded programs.

1. Mutex (Mutual Exclusion)
   - Uses blocking mechanism. If a thread tries to lock a mutex that's already locked, it waits (sleeps).
   - Safer for threads in user space.

Example:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);
// critical section
pthread_mutex_unlock(&lock);
```

2. Spinlock

- The thread actively waits in a loop (spins) until the lock is available.
- Used in kernel space or when wait time is expected to be very short.

Example:

```
#include <stdatomic.h>
atomic_flag lock = ATOMIC_FLAG_INIT;
while (atomic_flag_test_and_set(&lock)) {
        // spin
}
// critical section
atomic_flag_clear(&lock);
```