1.

1. #include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

#include <stdbool.h>

#include <time.h>

volatile sig\_atomic\_t keepRunning = 1;

// Signal handler to prevent termination on Ctrl+C

void sigint\_handler(int sig) {

printf("\nSIGINT caught! Termination is blocked. Program continues running.\n");

}

// Prime number checker

bool is\_prime(int num) {

if (num < 2) return false;

for (int i = 2; i \* i <= num; i++) {

if (num % i == 0) return false;

}

return true;

}

// Thread A: Sum of first N prime numbers

void\* threadA\_func(void\* arg) {

int N = (int)arg;

int sum = 0, count = 0, i = 2;

time\_t start = time(NULL);

while (count < N) {

if (is\_prime(i)) {

sum += i;

count++;

}

i++;

}

time\_t end = time(NULL);

printf("Thread A: Sum of first %d prime numbers = %d\n", N, sum);

printf("Thread A completed in %ld seconds.\n", end - start);

pthread\_exit(NULL);

}

// Thread B: Prints every 2 seconds

void\* threadB\_func(void\* arg) {

time\_t start = time(NULL);

while (time(NULL) - start < 100) {

printf("Thread 1 running\n");

sleep(2);

}

printf("Thread B completed.\n");

pthread\_exit(NULL);

}

// Thread C: Prints every 3 seconds

void\* threadC\_func(void\* arg) {

time\_t start = time(NULL);

while (time(NULL) - start < 100) {

printf("Thread 2 running\n");

sleep(3);

}

printf("Thread C completed.\n");

pthread\_exit(NULL);

}

int main() {

// Setup SIGINT handler

signal(SIGINT, sigint\_handler);

int N;

printf("Enter the value of N (for prime numbers): ");

scanf("%d", &N);

pthread\_t threadA, threadB, threadC;

// Start all threads

pthread\_create(&threadA, NULL, threadA\_func, &N);

pthread\_create(&threadB, NULL, threadB\_func, NULL);

pthread\_create(&threadC, NULL, threadC\_func, NULL);

// Wait for all threads to complete

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

pthread\_join(threadC, NULL);

printf("All threads finished execution.\n");

return 0;

}

2.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

#include <stdbool.h>

#include <time.h>

volatile sig\_atomic\_t keepRunning = 1;

// Signal handler to prevent termination on Ctrl+C, Ctrl+Z, etc.

void sigint\_handler(int sig) {

switch (sig) {

case SIGINT:

printf("\nSIGINT (Ctrl+C) caught! Termination blocked. Program continues...\n");

break;

case SIGTERM:

printf("\nSIGTERM caught! Termination blocked.\n");

break;

case SIGTSTP:

printf("\nSIGTSTP (Ctrl+Z) caught! Termination blocked.\n");

break;

default:

printf("\nUnknown signal caught.\n");

}

}

// Prime number checker function

bool is\_prime(int num) {

if (num < 2) return false;

for (int i = 2; i \* i <= num; i++) {

if (num % i == 0) return false;

}

return true;

}

// Thread A: Sum of first N prime numbers

void\* threadA\_func(void\* arg) {

int N = (int)arg;

int sum = 0, count = 0, i = 2;

time\_t start = time(NULL); // Start time tracking

while (count < N) {

if (is\_prime(i)) {

sum += i;

count++;

}

i++;

}

time\_t end = time(NULL); // End time tracking

printf("Thread A: Sum of first %d prime numbers = %d\n", N, sum);

printf("Thread A completed in %ld seconds.\n", end - start);

pthread\_exit(NULL);

}

// Thread B: Prints "Thread 1 running" 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);

}

printf("Thread B completed.\n");

pthread\_exit(NULL);

}

// Thread C: Prints "Thread 2 running" 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);

}

printf("Thread C completed.\n");

pthread\_exit(NULL);

}

int main() {

// Register signal handlers for SIGINT, SIGTERM, and SIGTSTP

signal(SIGINT, sigint\_handler);

signal(SIGTERM, sigint\_handler);

signal(SIGTSTP, sigint\_handler);

int N;

printf("Enter the value of N (for prime numbers): ");

scanf("%d", &N);

// Thread declarations

pthread\_t threadA, threadB, threadC;

// Start the threads

pthread\_create(&threadA, NULL, threadA\_func, &N);

pthread\_create(&threadB, NULL, threadB\_func, NULL);

pthread\_create(&threadC, NULL, threadC\_func, NULL);

// Wait for all threads to complete

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

pthread\_join(threadC, NULL);

printf("All threads finished execution.\n");

return 0;

}

3.

Child Process - fork()

- fork() is a fundamental system call in Unix/Linux used to create a new process (child) that is an almost exact replica of the parent process. This new process inherits the parent’s code, data, open file descriptors, and environment variables.

- The child process has a unique Process ID (PID) and a Parent Process ID (PPID) which is the PID of the parent. Despite sharing the same code segment, both processes have separate memory spaces (address spaces).

- The return value of `fork()` is different for the parent and the child:

- Parent Process: Receives the child's PID as the return value.

- Child Process: Receives 0, indicating it is the child.

- Practical Use Cases: Creating background tasks, implementing process pools, creating daemons, etc.

Handling Common Signals

- Signals are a form of inter-process communication in Unix/Linux that allow the operating system or other processes to interrupt the execution of a process.

- Signals can be categorized into two types:

- Synchronous Signals: Generated due to an error in the code (e.g., SIGFPE for divide by zero).

- Asynchronous Signals: Sent by other processes (e.g., SIGINT for interrupting using Ctrl+C).

- Signal handlers can be set using `signal()` for basic signal handling or `sigaction()` for more advanced control (signal mask, flags).

- Common Signals Explained:

- SIGINT: Interrupt signal, usually triggered by Ctrl+C.

- SIGTERM: Termination request, can be caught and handled.

- SIGKILL: Immediate termination, cannot be caught, blocked, or ignored.

Exploring Different Kernel Crashes

- Kernel crashes, also known as Kernel Panics, are severe system errors that force the system to halt or restart to prevent data corruption.

- Causes of Kernel Crashes include:

- Invalid memory access (NULL pointer dereference).

- Buffer overflows in kernel space.

- Hardware failures (CPU, RAM, disk errors).

- Faulty device drivers.

- Debugging Kernel Crashes involves:

- Viewing kernel logs using `dmesg` or `/var/log/kern.log`.

- Analyzing Core Dumps for memory state during the crash.

Time Complexity

- Time complexity is a computational metric that indicates how the execution time of an algorithm scales with the size of its input (n).

- The most common time complexities are:

- O(1) - Constant Time: Execution does not depend on input size (e.g., array access).

- O(log n) - Logarithmic Time: Efficient for large inputs, common in search algorithms (Binary Search).

- O(n) - Linear Time: Execution grows proportionally with input (Linear Search).

- O(n^2) - Quadratic Time: Common in nested loops (Bubble Sort).

- O(2^n) - Exponential Time: Rapidly increasing (Recursive Fibonacci).

- Understanding time complexity is critical for optimizing algorithms.

Locking Mechanism - Mutex vs Spinlock

- Mutex (Mutual Exclusion): A synchronization mechanism that ensures only one thread can access a critical section at a time. It is a blocking lock, meaning that a thread trying to acquire a locked mutex will be suspended until it is available.

- Spinlock: A non-blocking lock that causes a thread to continuously check if the lock is available (busy-waiting). It is efficient in scenarios where the lock is held for very short durations.

- Key Differences:

- Mutex is suitable for multi-core systems with threads that might be blocked for some time.

- Spinlock is ideal for fast, short operations on the same core where context switching would be costly.

- Use Cases:

- Mutex: Thread-safe file access, database connections.

- Spinlock: Low-level kernel synchronization (interrupts, fast data access).