Multithreading

Parallel Programming: Multiple process running

Background process: ./&app1

LWT: Light weighted threads/process: Shell attach threads to application

Concurrency: perform multiple tasks(functions) at a time(simultaneously) or handle multiple operations within overlapping time frames

Concurrency is necessary for:

1. Responsiveness:

It allows application to remain responsive to user even when performing long tasks in the background

For e.g.: webserver can handle multiple client requests concurrently without bocking other users

1. Utilizing multi core Processors:

Execute multiple instructions at a time

1. Efficient Resource utilization

Avoid idle times to better utilize available system resources.

1. Scalability

Efficient and effective usage

1. Improved throughput

By overlapping tasks especially in systems where tasks don’t always require CPU like I/O bound

Concurrency in Programming handled by:

1. Multithreading:

* Multiple threads within a single process are created to perform different tasks concurrently.
* Threads share same memory space which allows efficient communication but requires careful synchronization to prevent race conditions

1. Multiprocessing:

* Separate processes are created to handle different tasks concurrently.
* Each process has its own memory space. Which is safer but requires inter process communication methods like pipes or shared memory for collaboration

E.g.: Kho Kho

* POSIX threads(pthreads) creates and manages threads within a process

1. Asynchronous Programming

* Non-blocking functions are used to handle tasks such as I/O operations without blocking the main thread allowing other operations to proceed concurrently.
* This is common languages like Python, Java Script
* Fork() 🡪creates separate process with individual memory spaces in multiprocessing
* Concurrency is dealing with multiple tasks at once or having multiple tasks in progress
* Parallelism is executing multiple tasks truly simultaneously achieved by multiple CPU cores

Atomic Operations: Enables simple, thread-safe operations without full mutex overhead

Semaphores: Controls access to shared resources by multiple threads

* Each thread has its own stack, registers, program counter
* Threads within the same process can easily communicate
* Threads are lighter in terms of system process than full resources
* Traditional View of Process:

*Process= Process context +code, data and stack*

* Alternate view of Process:

*Process: thread+ code, data and kernel context*

* A process with multiple threads

Each thread has its own logical control flow

Each thread shares the same code data and kernel context

Each thread has its own thread id

Threads and Process similarities

* Each has its own logical control flow
* Each can run concurrently
* Each is context switched

Threads and Process differences

* Threads share same data and process do not
* Threads are less expensive than process
* Shares memory while process do not
* Threads are fast to create and lightweight
* Thread terminates when function finishes by exit call while process terminate independently.
* Threads are easy to communicate while in process we need inter process communication.

Pthreads

**Creating and reaping threads:** pthread\_create, pthread\_join

**Determining thread id**: pthread\_self

**Terminating threads**: pthread\_cancel, pthread\_exit

Exit(terminates all threads), return( terminates current thread)

**Synchronizing access to shared variables**: pthread\_mutex\_init, pthread\_mutex\_[un]lock

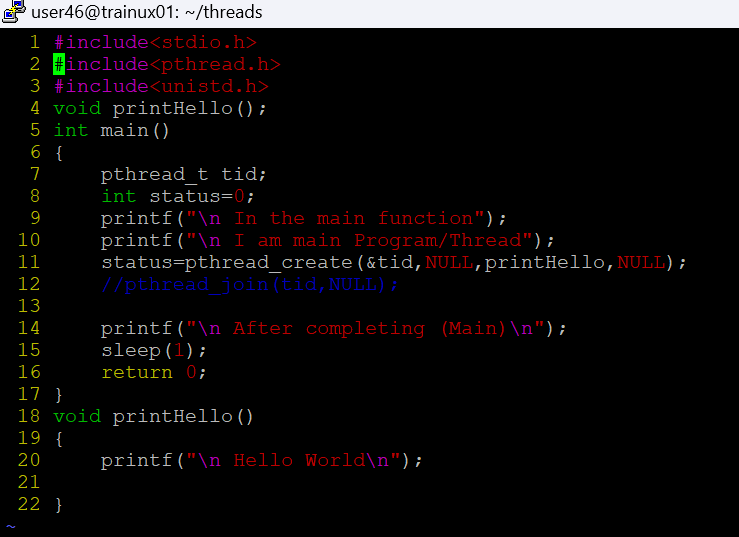
pthread\_cond\_init, pthread\_cond\_[timed]wait

pthread\_create returns 0 if created on error returns error number

pthread\_join: it is used to wait and execute the thread completely

Without joining we can use sleep to wait for sometime to get the thread executed

To compile threads: gcc t1.c -lpthread



When we pass the arguments to the function like structure or pointer we need to type cast it to void \* type and the function should receive void \* type

If it is a linear search we can reduce time by threads.

We can use mutex when there is a race condition

Pthread\_mutex\_t lock 🡪global declaration

In main initialize the mutex: Pthread\_mutex\_init(&lock);

And when there’s a shared resource there, we can use mutex lock Pthread\_mutex\_lock(&lock);

And unlock it when it is removed: Pthread\_mutex\_unlock (&lock);

And destroy the mutex : Pthread\_mutex\_destroy(&lock);

Why should we destroy the mutex

Why do while has a semicolon?

Because for thr first time it should execute irrespective of any condition like printf and other

Why structures have a semicolon?

Because they need to be bunched together at once