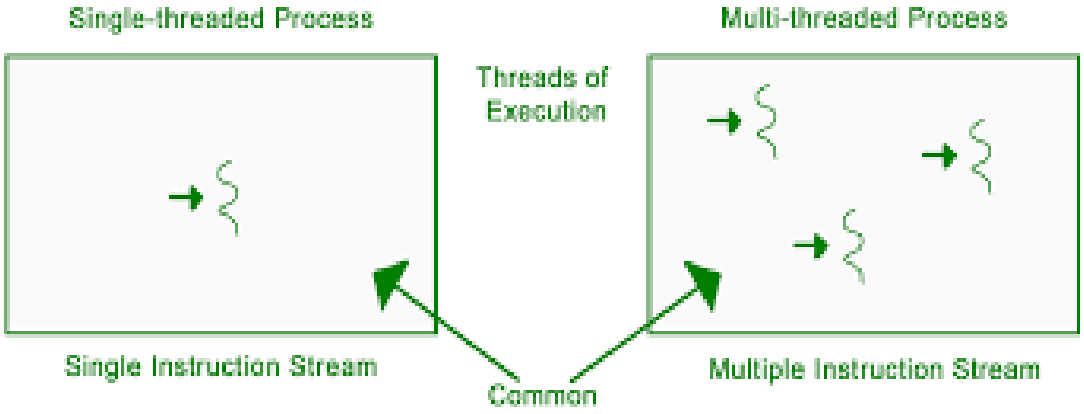
Multithreading

What is multithreading?

- Multithreading in an operating system is the ability of a program or operating system to allow multiple tasks to be executed simultaneously
- It is a type of parallelization that splits work into multiple threads, which are then processed in parallel by different CPU cores



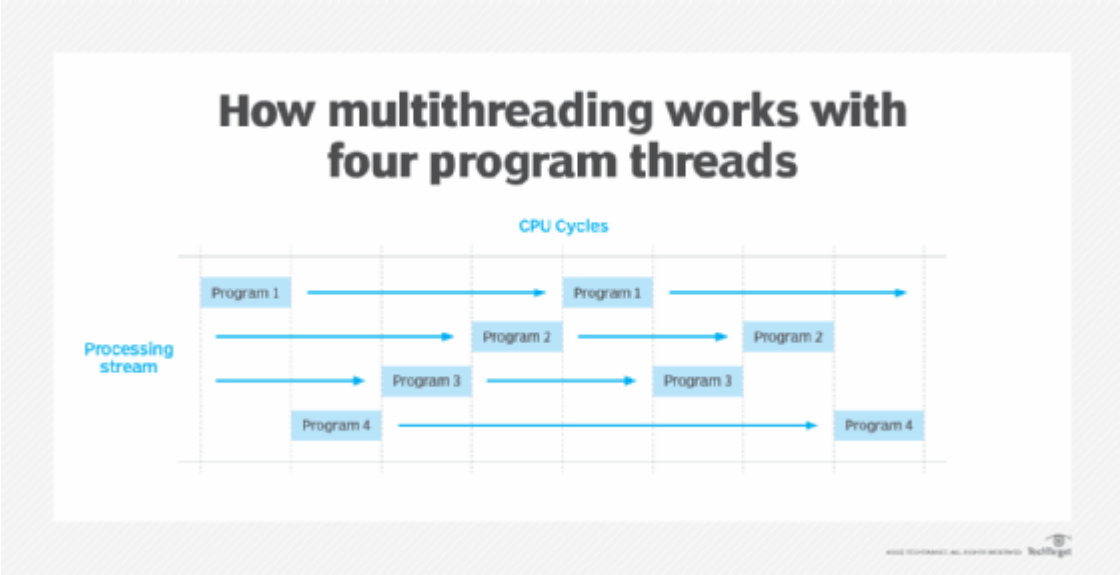
Single Thread and Multi Thread Process

Advantages of multithreading

- ✓ Multiple users: Multithreading allows multiple users to access a program or system service simultaneously, without requiring multiple copies of the program
- ✓ Multiple requests: Multithreading can handle multiple requests from the same user
- ✓ Faster processing: Multithreading saves time by splitting work into multiple threads that are processed in parallel

More on multithreading

- Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer
- Multithreading can also handle multiple requests from the same user
- Each user request for a program or system service is tracked as a thread with a separate identity
- As programs work on behalf of the initial thread request and are interrupted by other requests, the work status of the initial request is tracked until the work is completed. In this context, a user can also be another program
- Fast central processing unit (CPU) speed and large memory capacities are needed for multithreading
- The single processor executes pieces, or threads, of various programs so fast, it appears the computer is handling multiple requests simultaneously



Multithreading vs. multitasking vs. multiprocessing

- **Multithreading** in an operating system is the ability of a program or operating system to allow multiple tasks to be executed simultaneously
- **Multitasking** is a computer's ability to execute two or more concurrent programs. Multithreading makes multitasking possible when it breaks programs into smaller, executable threads. Each thread has the programming elements needed to execute the main program, and the computer executes each thread one at a time.
- **Multiprocessing** uses more than one CPU to speed up overall processing and supports multitasking.

What is a thread?

- A **thread** is a path that is followed during a program's execution
- Majority of programs written nowadays run as a single thread
- Ex: a program is not capable of reading keystrokes while making drawings. These tasks cannot be executed by the program at the same time
- This problem can be solved through multitasking so that two or more tasks can be executed simultaneously
- //Hand analogy//

How does multithreading work?

- Processor Handling: The processor can execute only one instruction at a time, but it switches between different threads so fast that it gives the illusion of simultaneous execution.
- Thread Synchronization: Each thread is like a separate task within a program. They share resources and work together smoothly, ensuring programs run efficiently.
- Efficient Execution: Threads in a program can run independently or wait for their turn to process, making programs faster and more responsive.
- Programming Considerations: Programmers need to be careful about managing threads to avoid problems like conflicts or situations where threads get stuck waiting for each other.

Two types of multitasking

1. Processor-based
2. Thread-based

- Processor-based multitasking is managed by the OS, however, multitasking through multithreading can be controlled by the programmer to some extent
- **a process and a thread**:
- A process is a program being executed. A process can be further divided into independent units known as threads.
- A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process

What is posix?

- POSIX, or Portable Operating System Interface, is a set of standards that ensure compatibility between different operating systems, especially those based on Unix

- IEEE Computer Society POSIX specification includes:
  - A standard interface and environment
  - A command interpreter (shell)
  - Common utility programs
  - Fundamental services for building POSIX-compliant applications
  - Standard semantics and syntax to help developers write portable applications

Current version: IEEE Std 1003.1-2024.

POSIX and Linux

- Linux is seen as “partially POSIX compatible” when it comes to the file system, mainly because it doesn't provide the isolation feature that POSIX requires for IO operations
- POSIX does not specify a kernel interface; It does specify the system interface, various tools, and extensions to the C standard, which could exist on top of any kernel

More on POSIX

- POSIX defines its standards in terms of the C language. Therefore, **programs are portable to other operating systems at the source code level**.
- We can also implement it in any standardized language
- The POSIX C API adds more functions on top of the ANSI C Standard for a number of aspects like:
  - ✓File operations
  - ✓Processes, threads, shared memory, and scheduling parameters
  - ✓Networking
  - ✓Memory management
  - ✓Regular expressions

Can we create OS threads in Linux?

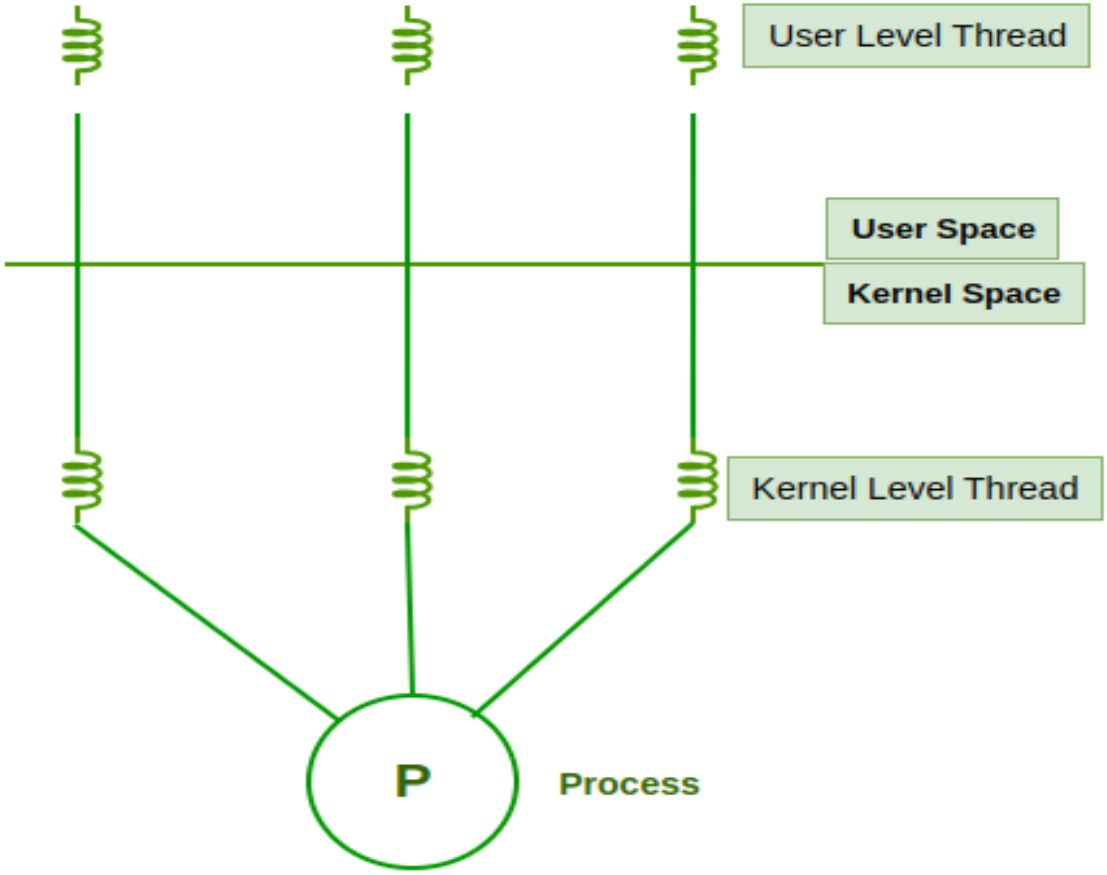
pthread_create() creates a new thread and takes three arguments:

A pointer to a pthread_t variable to store the thread ID

pthread_attr_t* attr

void (function)(void),void* param

User level threads and kernel level threads in OS



Thread creation and termination: ex


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

void *print_message_function( void *ptr );

main()
{
pthread_t thread1, thread2;
char *message1 = "Thread 1";
char *message2 = "Thread 2";
int iret1, iret2;

/* Create independent threads each of which will execute function */

iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

/* Wait till threads are complete before main continues. Unless we */
/* wait we run the risk of executing an exit which will terminate */
/* the process and all threads before the threads have completed. */

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);

printf("Thread 1 returns: %d\n",iret1);
printf("Thread 2 returns: %d\n",iret2);
exit(0);
}

void *print_message_function( void *ptr )
{
char *message;
message = (char *) ptr;
printf("%s \n", message);
}
```

Compile:

C compiler: cc -lpthread pthread1.c
or
C++ compiler: g++ -lpthread pthread1.c

Run: ./a.out
Results:
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0

Thread basics

- Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- **Threads in the same process share:**
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id
- **Each thread has a unique:**
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno
- pthread functions return "0" if OK.

https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html