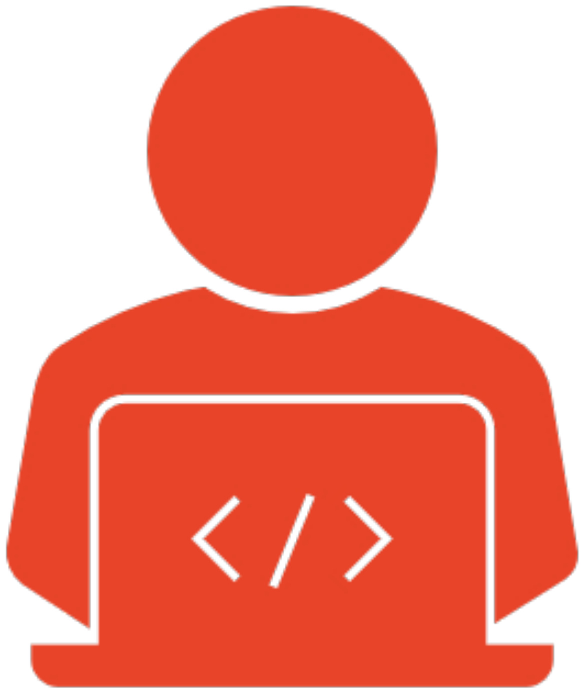
Understandi ng Threading

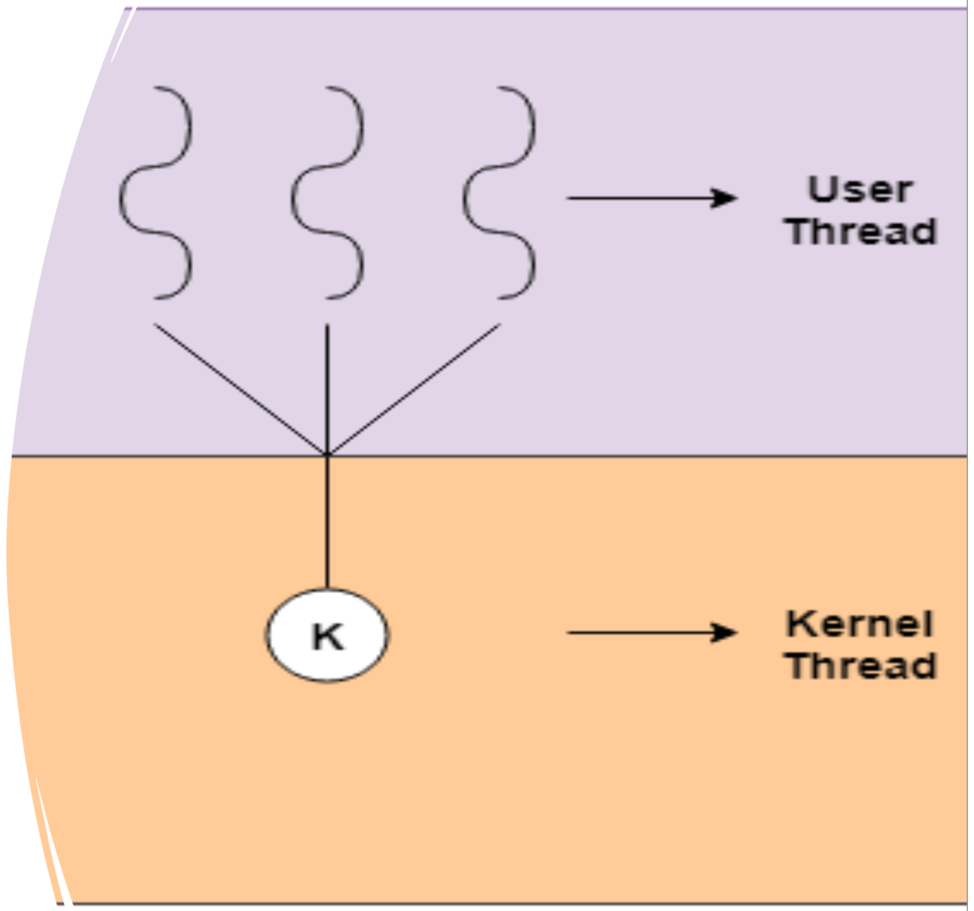
Introduction to Threads

- **Definition:** A thread refers to a single sequential flow of activities being executed in a process
- **Shares:** information like data segment, code segment files, etc. with its peer threads.
- **Contain:** its registers, stack, counter, etc.
- **Significance:** Simplifies code and increases efficiency.



Types of Threads

The two main types of threads are user-level threads and kernel-level threads.



User - Level Threads

- User-level threads are implemented by users.
- The kernel is unaware of the existence of user-level threads.
- The kernel treats user-level threads as single-threaded processes.
- User-level threads are small and much faster than kernel-level threads.
- Represented by: Program counter (PC), Stack, Registers, and Small process control block
- No kernel involvement in synchronization for user-level threads.

Kernel-level threads

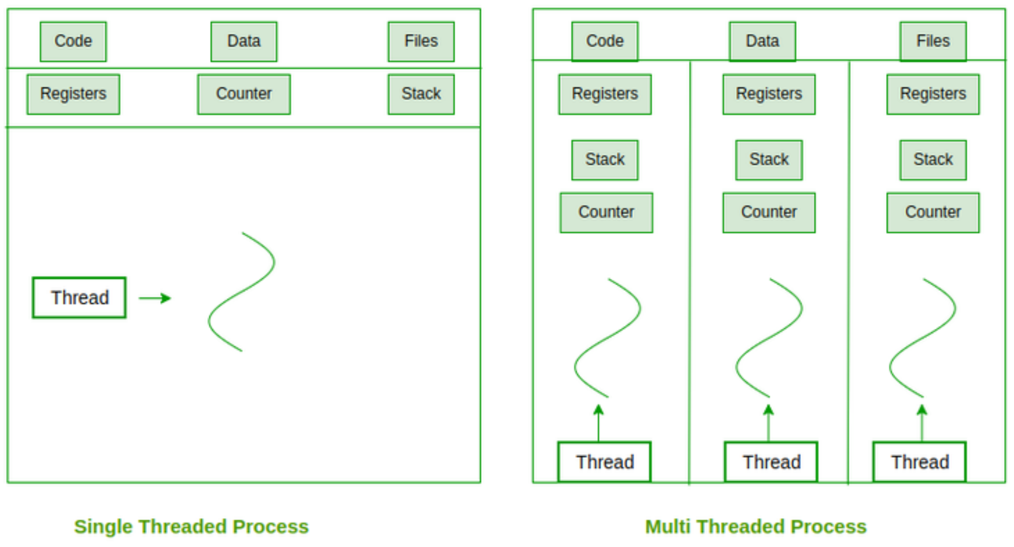
- Kernel-level threads are handled by the operating system directly.
- Thread management is done by the kernel.
- The kernel manages context information for both the process and its threads.
- Kernel-level threads are slower than user-level threads due to kernel management.

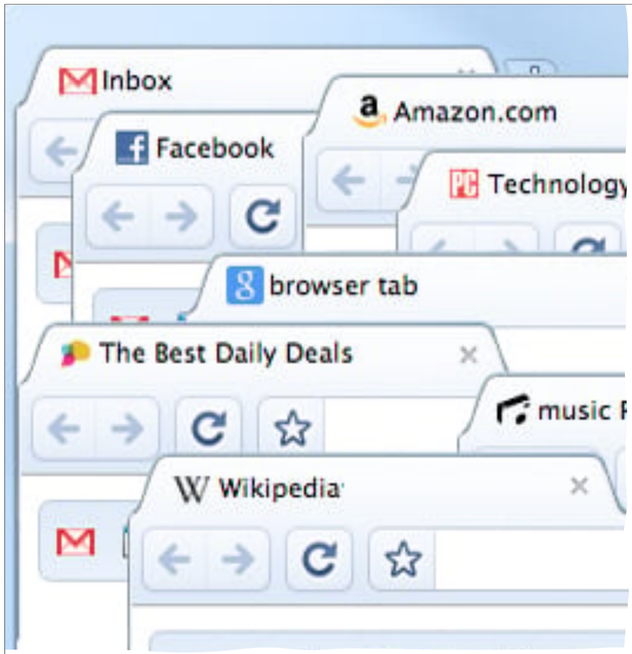
Advantages and Disadvantages of User-level & Kernel-Level Threads

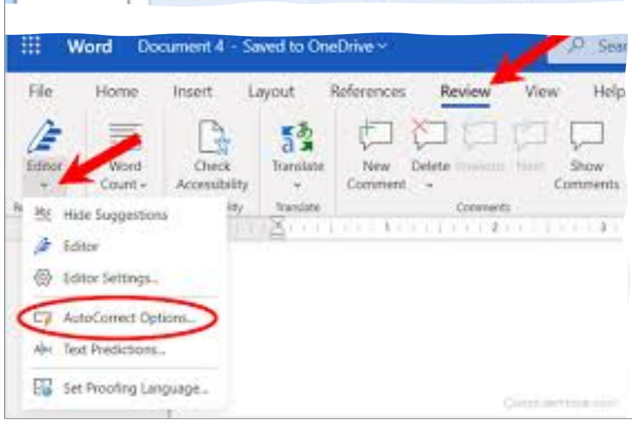
ASPECT	USER-LEVEL THREADS	KERNEL-LEVEL THREADS
Advantages	- Faster	- Only blocks the calling thread
	- More resource-efficient	- Kernel manages scheduling and resources
	- Quicker context switching	- Easier synchronization
	- Simpler to implement	- Easier to debug
Disadvantages	- Blocks entire process on system call	- Slower
	- Kernel is unaware	- Less resource-efficient
	- Complex synchronization	- Slower context switching

Multi-Threading

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.





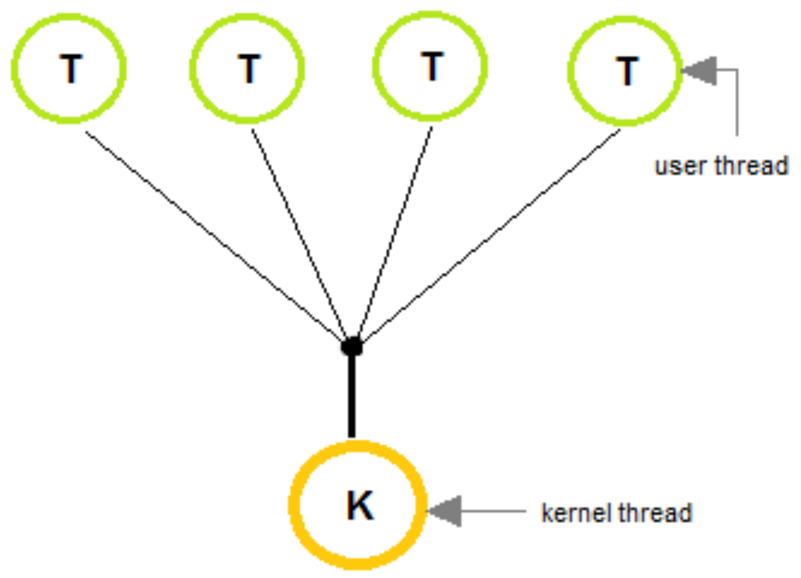


Example of multi-threading

- In a browser, each tab can be a different thread.
- MS Word employs multiple threads: one for text formatting, another for processing inputs, etc.

Multi-Threading Models

- **Many-to-One:** Many user-level threads mapped to a single kernel thread.
- **One-to-One:** Each user-level thread maps to a kernel thread.
- **Many-to-Many:** Many user-level threads mapped to many kernel threads.

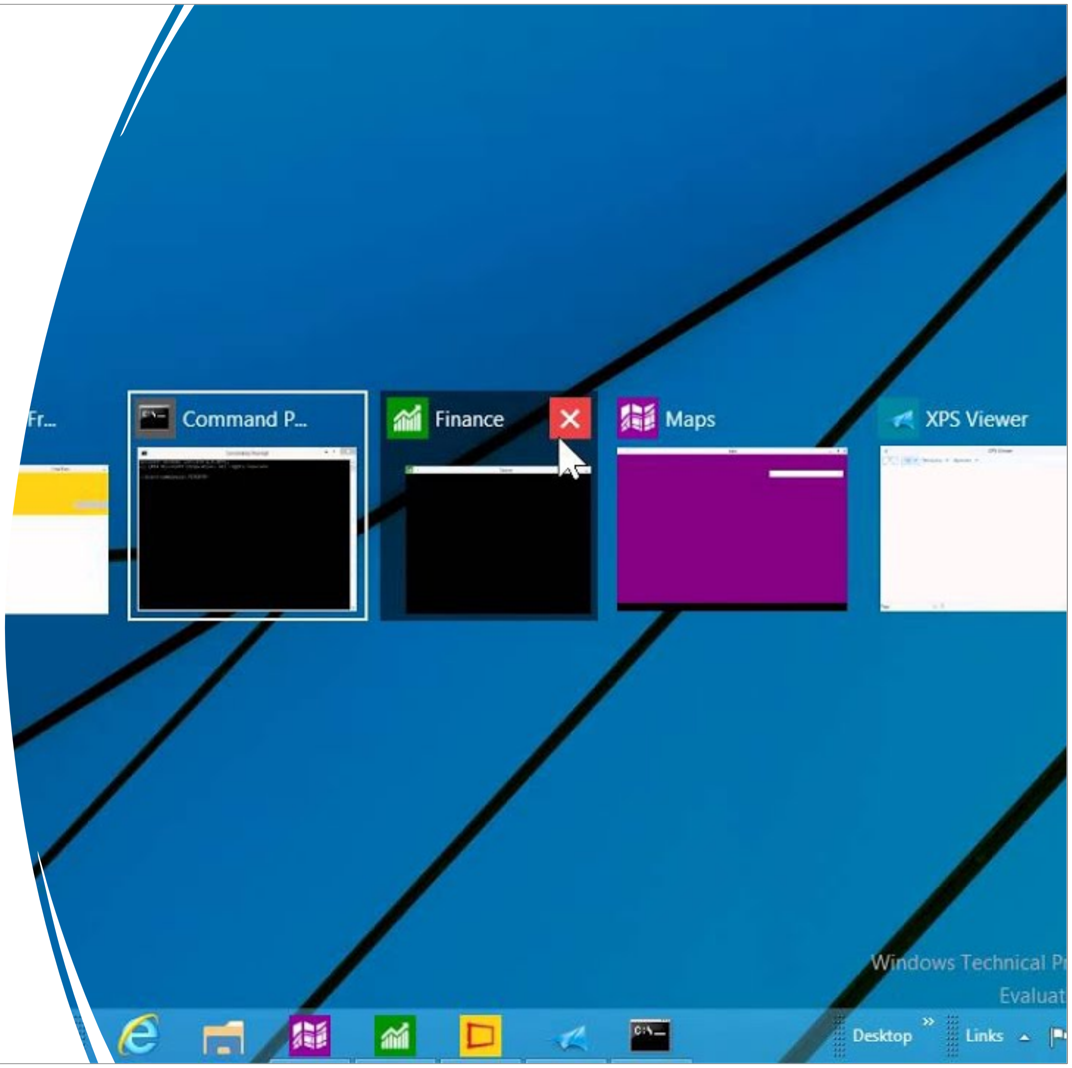


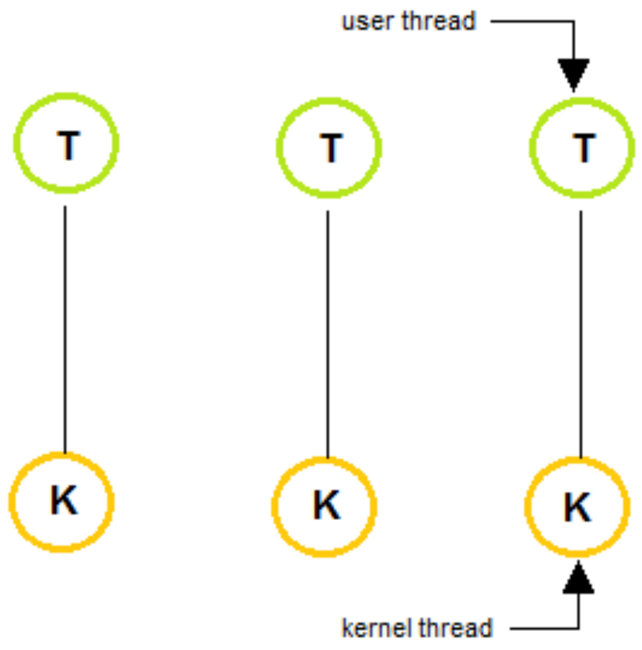
Many-to-One Model

- **Description:** Many user-level threads mapped to a single kernel thread.
- **Advantage:** Simple to implement.
- **Disadvantage:** If the kernel thread blocks, all user-level threads are blocked.

eXAMPLE

Imagine you are working on your computer. You can have several programs open at the same time—like a word processor, a web browser, and a music player. All of these programs are running under your user account. So, one user can have many applications running.

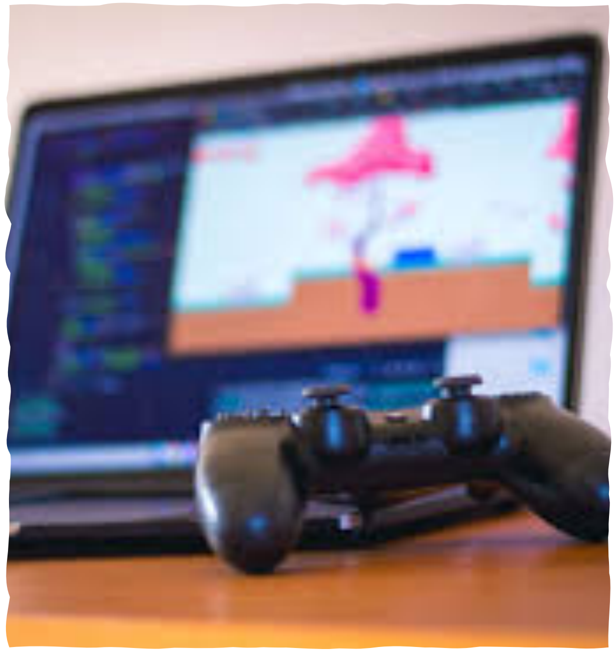




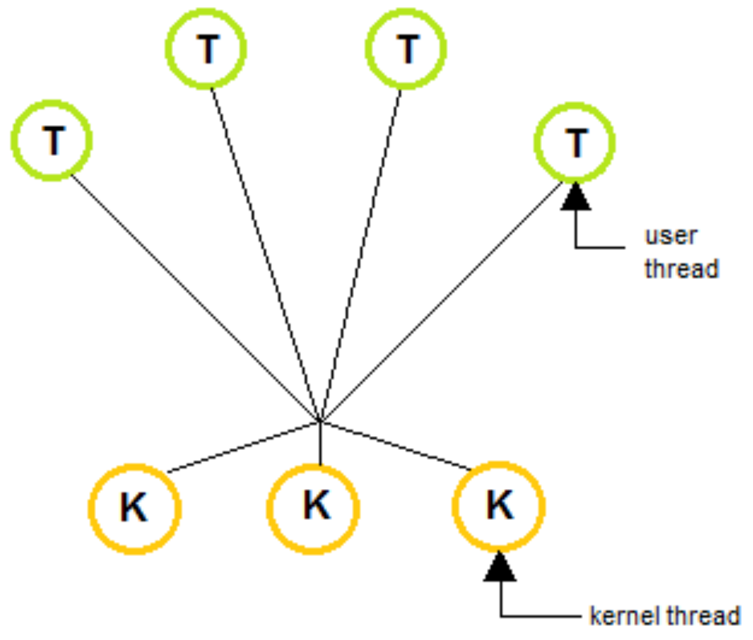
One-to-One Model

- **Description:** Each user-level thread maps to a kernel thread.
- **Advantage:** More concurrency as other threads can run if one thread is blocked.
- **Disadvantage:** Creates overhead due to large number of kernel threads.

EXAMPLE



Think of a process as a task or job your computer is doing, like running a game or a web browser. Each of these tasks has a unique “ID card” called a PCB that keeps track of everything about that task, like how much memory it’s using or what it’s doing right now. So, one process gets one PCB.



Many-to-Many Model

- **Description:** Many user-level threads mapped to many kernel threads.
- **Advantage:** Combines the benefits of both models, allowing better resource utilization.
- **Disadvantage:** More complex to implement.

EXAMPLE

Think of files on your computer (like documents or pictures). Different users can have different kinds of access to these files—some might be able to read them, some might be able to write to them, and some might not have access at all. And, each user can have different access to many files. So, many files can be accessed by many users in various ways.

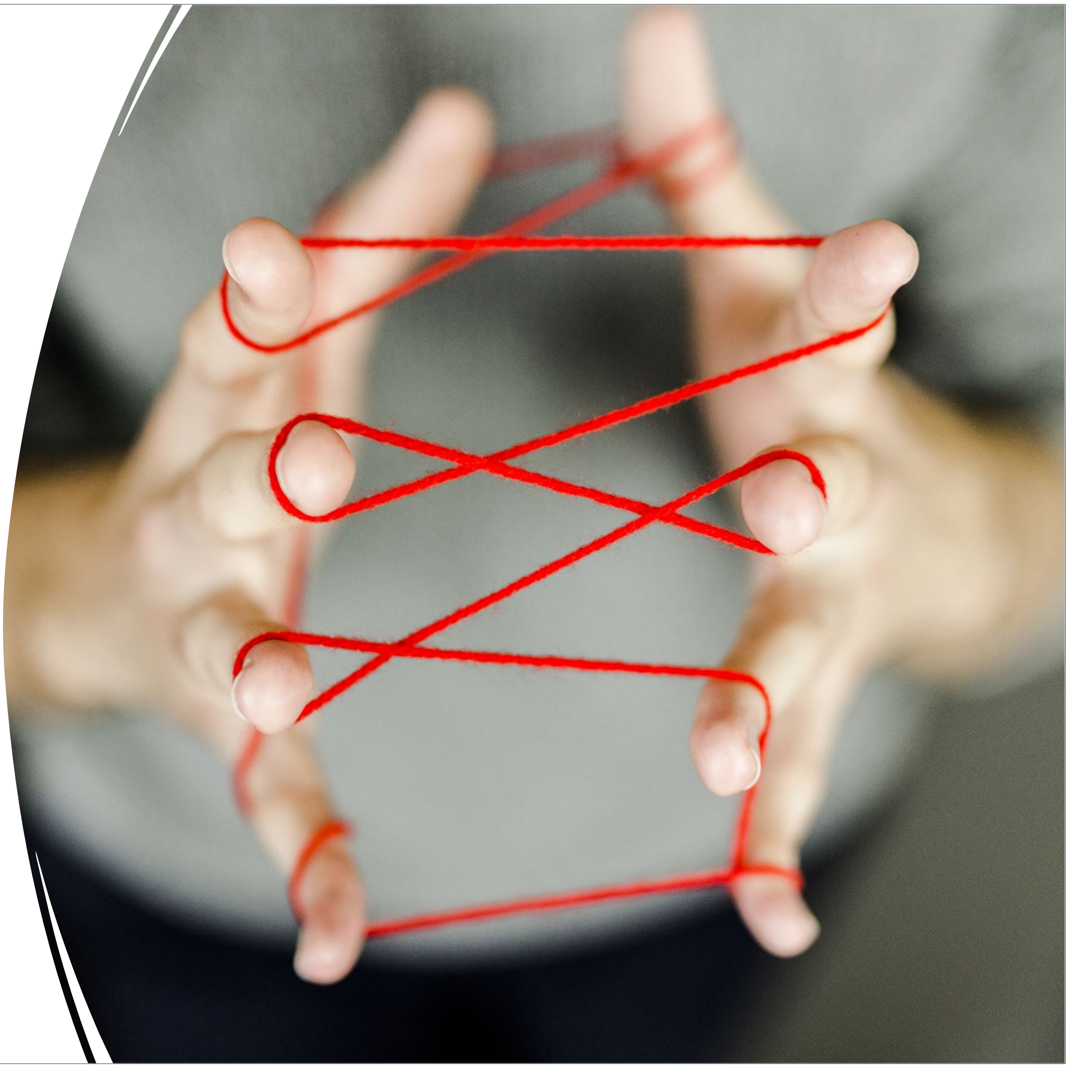
Thread Libraries

PURPOSE: PROVIDE PROGRAMMERS WITH AN API FOR CREATING AND MANAGING

EXAMPLES: POSIX PTHREADS, JAVA THREADS, WINDOWS THREADS.

Issues in Multi-Threading

Thread cancellation, signal handling (synchronous/ asynchronous), handling thread-specific data, and scheduler activations



Thread Cancellation

Definition: The process of terminating a thread before it completes execution.

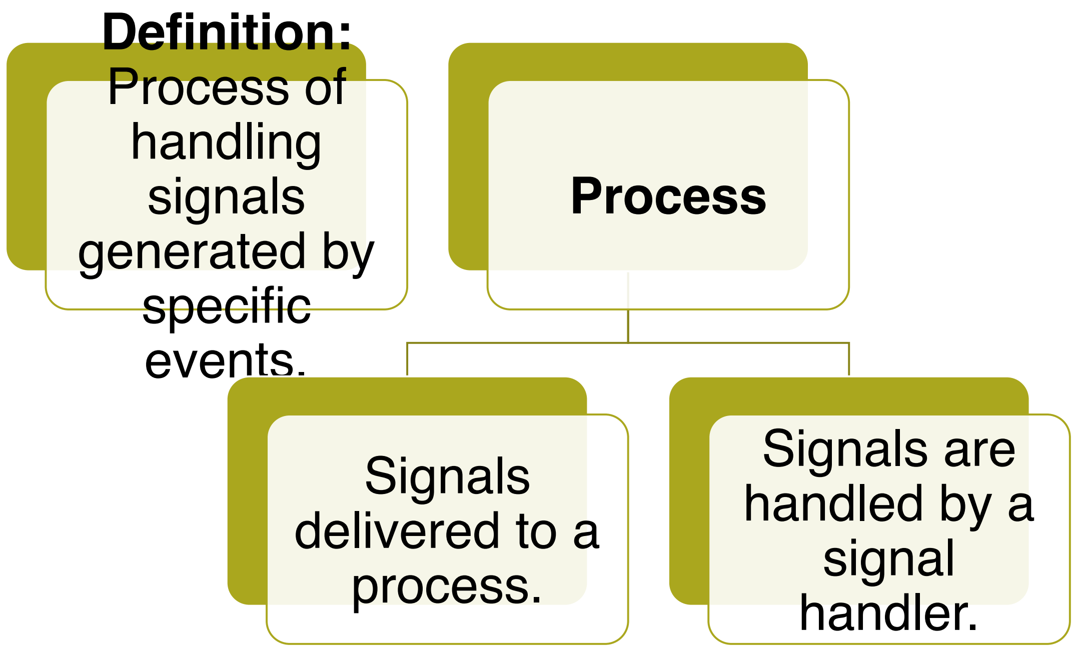
Asynchronous Cancellation

- **Definition**: Terminates the target thread immediately.
- **Use Case**: Required for immediate termination.

Deferred Cancellation

- **Definition**: Allows the target thread to periodically check if it should be canceled.
- **Use Case**: Ensures safe and orderly termination.

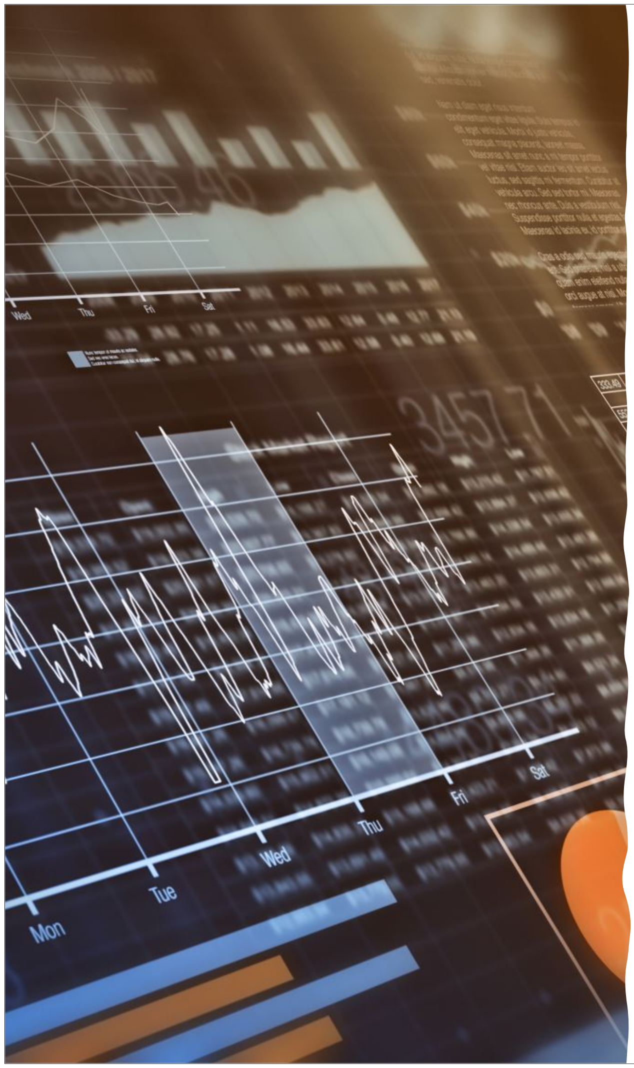
Signal Handling


Definition: Process of handling signals generated by specific events.
Process
Signals delivered to a process.
Signals are handled by a signal handler.

Handling thread-specific data

- Definition: Managing data that is unique to each thread in a multi-threaded application.
- Process

1. **Creation**: Allocate data storage for each thread.
2. **Access**: Provide mechanisms for each thread to access its own data.
3. **Cleanup**: Ensure thread-specific data is cleaned up when the thread terminates.



Scheduler Activations

•**Definition**: A mechanism that provides upcalls, enabling communication from the kernel to the thread library.

•**Purpose**: Allows the application to maintain the correct number of kernel threads.

•**Benefit**: Improves performance and resource management.

basic thread creation and execution

•**Define Functions**: Two functions, print_numbers and print_letters, are defined. Each function prints a sequence of items with a 1-second delay between prints.

•**Create Threads**: Two threads, t1 and t2, are created. t1 runs the print_numbers function, and t2 runs the print_letters function.

•**Start Threads**: Both threads are started using the start() method, which triggers the execution of the respective functions in parallel.

•**Wait for Completion**: The join() method is used to wait for both threads to finish their execution before the main program continues.

```python
import threading
import time

def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")
        time.sleep(1)

def print_letters():
    for letter in ['A', 'B', 'C', 'D', 'E']:
        print(f"Letter: {letter}")
        time.sleep(1)

# Creating threads
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

# Starting threads
t1.start()
t2.start()

# Waiting for threads to finish
t1.join()
t2.join()
```

What will the print_numbers function do when executed in a thread?

[✓ A. Print numbers from 1 to 5, each followed by a 1-second delay]

B. Print letters from A to E, each followed by a 1-second delay

C. Print numbers from 1 to 5 without any delay

D. Print numbers from 1 to 5, but with a 1-second delay between threads

What is the purpose of the t1.join() and t2.join() calls in the code?

A. To start the threads t1 and t2

[✓ B. To wait for the threads t1 and t2 to complete before the main program exits]

C. To ensure that t1 and t2 run concurrently

D. To terminate the threads t1 and t2 immediately

What will happen if time.sleep(1) is removed from both functions?

[✓ A. The output will be printed immediately without any delay]

B. The threads will not start at all

C. The threads will wait for 1 second before printing the output

D. The program will hang indefinitely

Activity

- Scan this QR code or click the link below to read the article titled ‘A Comparison of Process and Thread Creation’ by Martin Sysel.
- Then, be prepared to answer multiple-choice questions that the professor will ask in class.

What is the primary difference between processes and threads according to the article?

[✓ A.] Processes deal with resource ownership, while threads are units of execution.

B. Processes are lighter than threads.

C. Threads can exist without processes.

D. Processes cannot create threads.

In Linux, which system call is primarily responsible for creating both processes and threads?

A. exec()

B. fork()

[✓ C. clone()]

D. pthread_create()

How does the fork() function differ from pthread_create() in terms of resource sharing?

A. fork() shares more resources than pthread_create().

B. pthread_create() shares more resources than fork().

C. Both share the same amount of resources.

D. Neither shares any resources.

Which of the following is a key advantage of threads over processes?

A. Threads are safer and more secure.

[✓ B. Threads have faster inter-thread communication.]

C. Threads have their own address space.

D. Threads do not require synchronization.

In the context of this article, which statement is true about processes and threads in Linux?

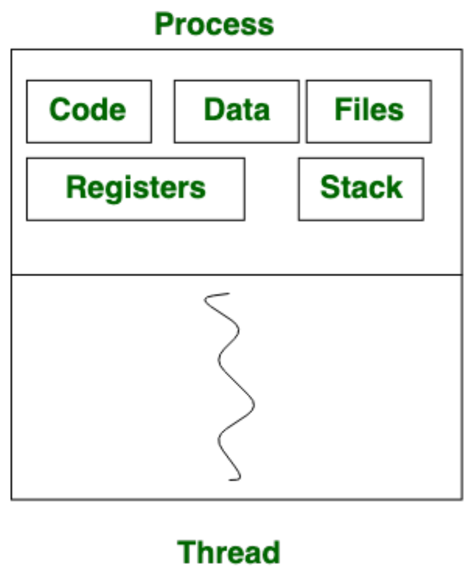
A. Processes are generally more lightweight than threads.

B. Threads and processes are both created using the exec() system call.

[✓ C.] Processes and threads are both considered tasks and are similarly scheduled.

D. Threads cannot share virtual memory with their parent process.

Process vs. Thread



Process

•**Memory**: Separate.

•**Overhead**: Higher.

•**Communication**: Uses IPC.

•**Creation**: Slower and more expensive.

•**Isolation**: Isolated from other processes.

Thread

•**Memory**: Shared.

•**Overhead**: Lower.

•**Communication**: Easy within process.

•**Creation**: Faster and cheaper.

•**Isolation**: Not isolated; affects other threads.

Example of a process

Applications like word processors or web browsers.

Example of a thread

Background tasks or operations within an application.

When to choose between using processes and threads

- Processes: Better isolation, security, and fault tolerance. Use when tasks are independent and require separate memory spaces. Examples: Web server handling each request in a separate process, Microservices architecture with separate processes for different services, and Database systems with each client connection in a separate process.

- Threads: Faster communication and resource sharing. Use when tasks are interdependent and can benefit from parallel execution. Example: Real-time data processing in a stock trading application, GUI applications with the main thread for UI and background threads for tasks, Parallel computation in scientific applications, and Web server with thread pools for handling multiple connections.