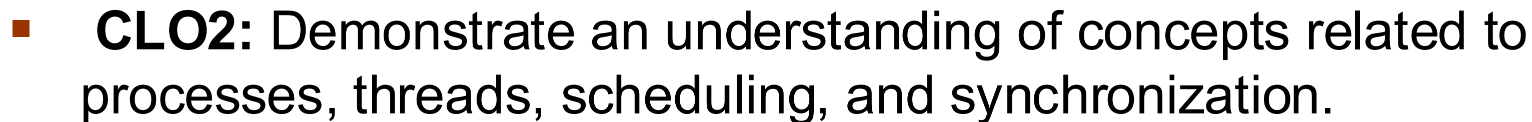
Chapter 4: Threads & Concurrency

- **CLO1**: Describe the objectives, major components, and functions of a modern operating system and recognize the concept of parallel processing.



- **CLO2:** Demonstrate an understanding of concepts related to processes, threads, scheduling, and synchronization.

Silberschatz, Galvin and Gagne ©2018

Outline of this Topic

In this Topic, we will cover the following:

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Operating System Examples

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Describe how the Windows and Linux operating systems represent threads
- Describe multithreaded applications using the Pthreads, Java, and Windows threading APIs

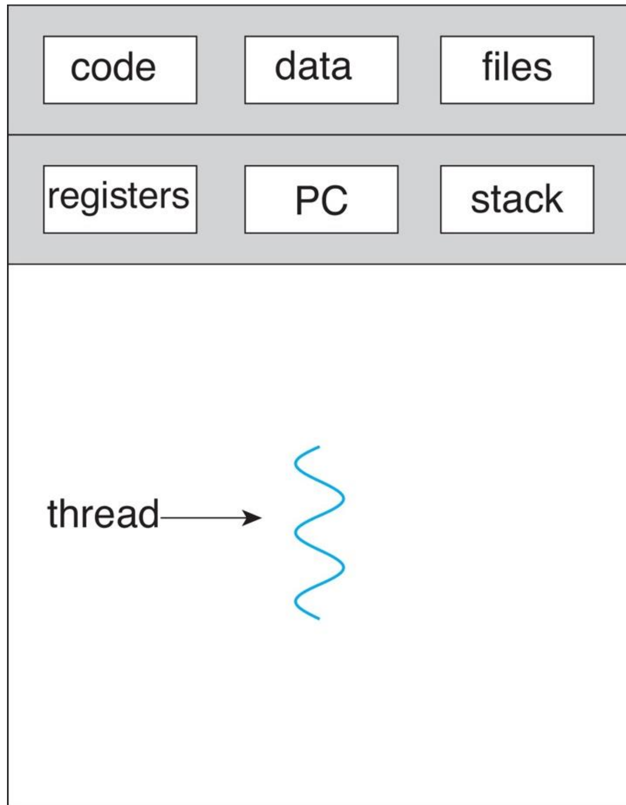
Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

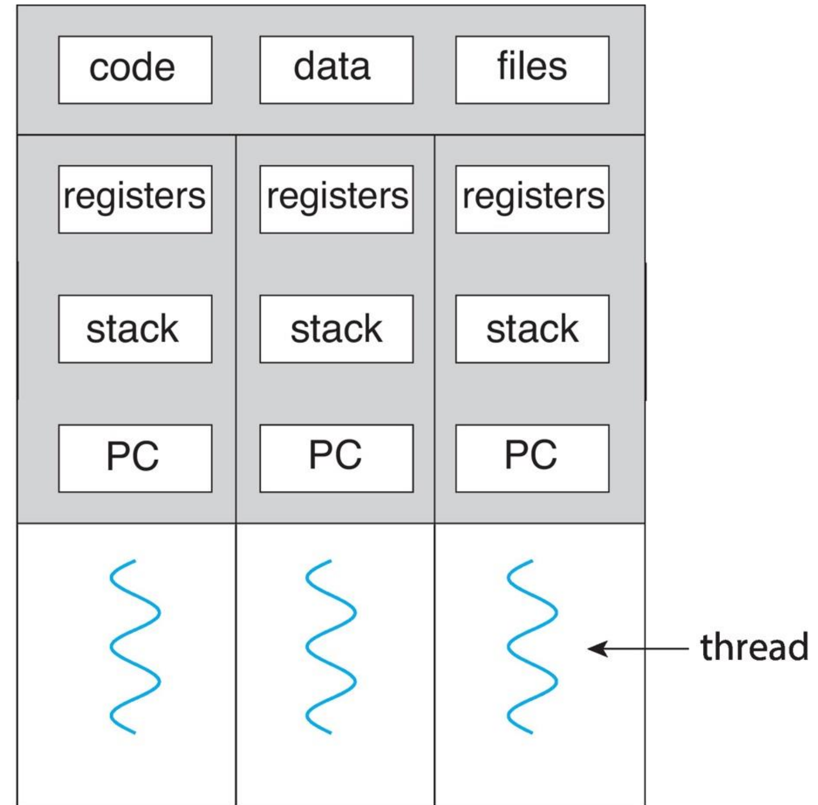
Threads

- *Heavyweight process*: Traditional processes that can have only one thread
- *Multi-threaded processes:* A process that includes multiple threads that share the resources allocated to the process
- *lightweight process:* another name for A *thread,* is a basic unit of CPU utilization; it is comprised of:
  - A thread ID
  - program counter
  - register set
  - stack space
- A thread shares with other threads its:
  - code section
  - data section
  - O/S resources

Single and Multithreaded Processes



single-threaded process



multithreaded process

Examples of Using Threads

- Threads are important for any application with multiple tasks that can be run with separate threads of control.
- A web server could produce a thread for each client
  - Serve clients with multiple threads concurrently .
  - less overhead to use multiple threads than to use multiple processes.
- A Word processor may have separate threads for:
  - Reading user input
  - Display graphics
  - Performing Spell & grammar check

Multithreaded Server Architecture

- Server will create a separate thread ------→listens for client requests.
- When a request is made, rather than creating another process, ---------->the server will create a new thread to service the request & resume listening for additional requests.

Benefits

- Benefits of threads are:
  - **Responsiveness:**
    - Threads allow a program to continue running even if part of it is blocked/ performing a lengthy operation. e.g: a multithreaded web browser can allow user interaction in 1 thread while loading an image in another thread.
  - **Resource Sharing:**
    - threads share memory and resources of the process they belong to.
  - **Economy:**
    - Allocating memory and resources to a process is costly- threads share resources of the process.
    - Threads are faster to create and to switch between them than processes .(i.e. in Solaris, creating P is 30 times slower than creating thread).
  - **Scalability- Utilization of Multiprocessor architectures:**
    - Multiple threads can run in parallel on different CPUs - A single threaded process can run only on 1 CPU no matter how many are available.
- Thread management could be done at either
  - User level
  - Kernel level

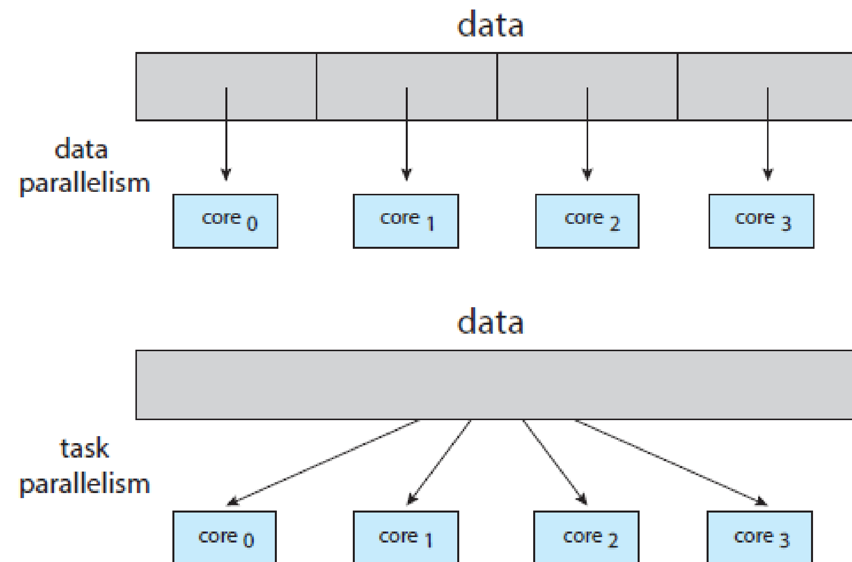
Multicore Programming

- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - **Dividing activities-** area of the application that can be divided into separate tasks.
  - **Balance-** ensure that tasks perform equal work or equal value
  - **Data splitting-** data accessed by the tasks must be divided to run on separate cores
  - **Data dependency-** 1 task depend on data from another, the execution synchronized
  - **Testing and debugging-** When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications

Multicore Programming *cont.*

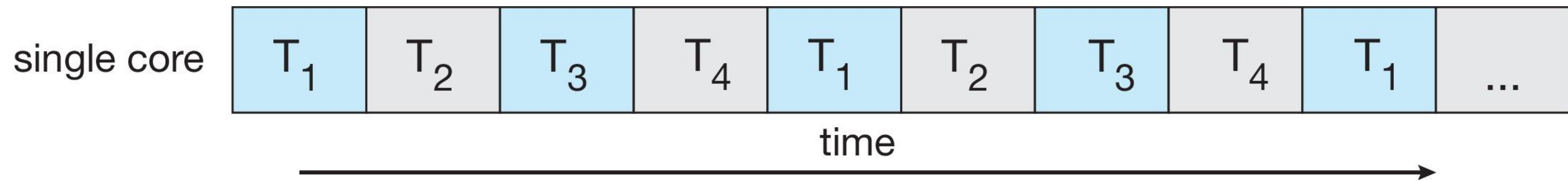
Models of multiprogramming:

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task by allowing all of them to make progress
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, **same** operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing **unique** operation

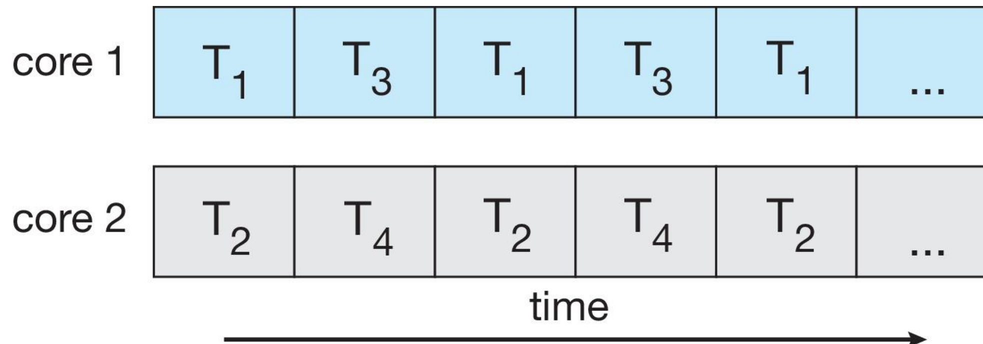


Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**
  - Concurrency means that execution of threads will be interleaved over time



- **Parallelism on a multi-core system:**
  - Concurrency means that execution of threads can run in parallel because the system can assign a separate thread to each core.



User Threads

- A thread is managed by an application (by user-level threads library) not in the kernel
- Scheduled by the thread library
- Fast to create and manage: all thread creation and scheduling are done in user space without the need for kernel intervention
- Any user-level thread performing a blocking system call will cause the entire process to block if the kernel is single threaded.
- Examples of three primary thread libraries

  - POSIX *Pthreads*

  - Java threads

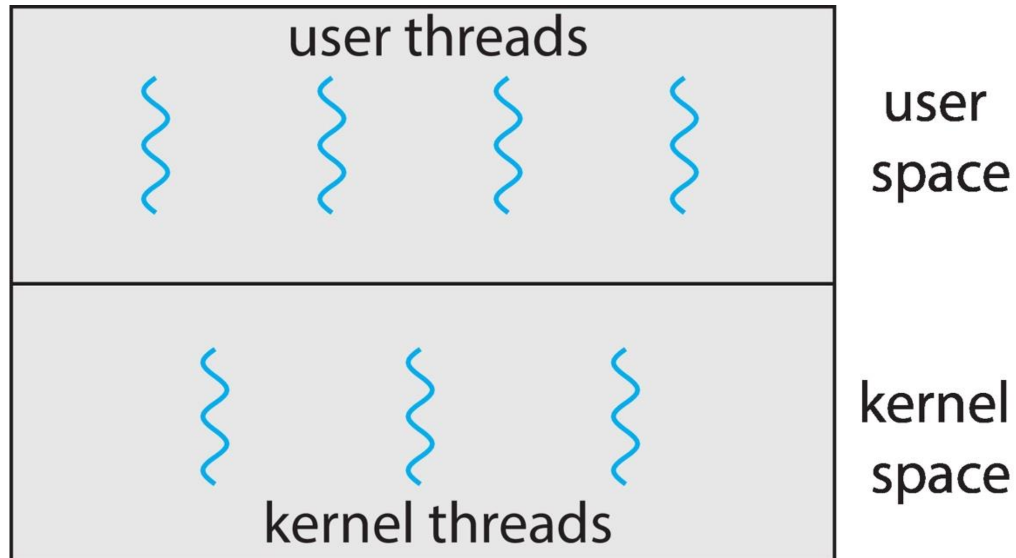
  - Win32 threads

Kernel Threads

- All thread management is done by the kernel itself
- Generally slower to create and manage than user threads
- If a thread is performing a blocking system, the kernel can schedule another thread in an application to be executed
- Kernel can schedule threads on different processors
- Kernel does creation, scheduling and management of threads.

- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads



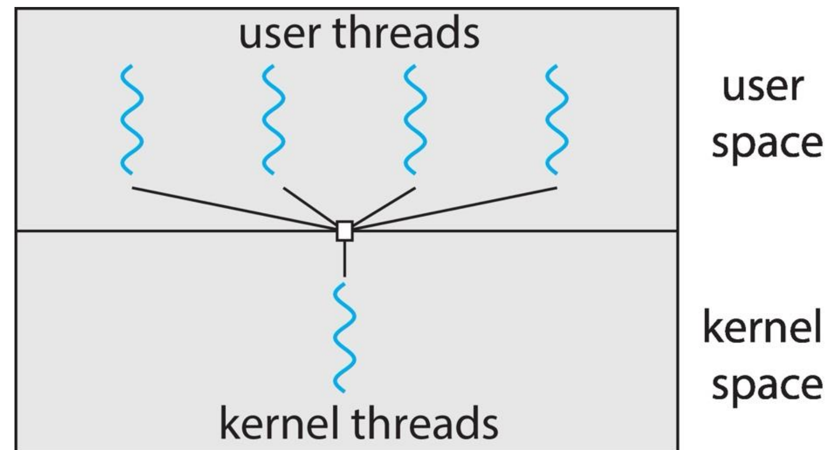
Multithreading Models

What is the relationship between the user & kernel threads:

- Many-to-One
- One-to-One
- Many-to-Many

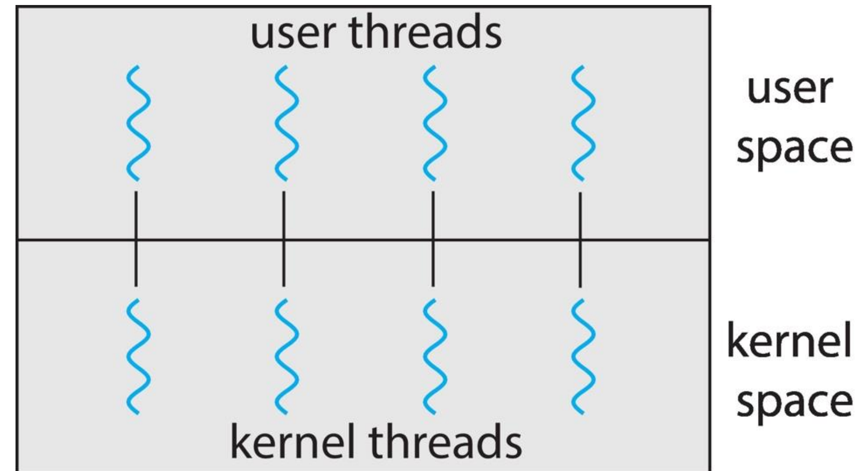
Many-to-One

- Many User threads are mapped onto a Single Kernel Thread.
- Used on Systems That Do Not Support Kernel Threads.
- Thread management is done outside the kernel--□ Efficient but
  - Entire process will block if a thread performs a blocking system call.
  - Only one thread can access the kernel at a time so multiple threads are unable to run in parallel on multiprocessors.
  - Few systems currently use this model
- Examples
  - Solaris Green Threads
  - GNU Portable Threads



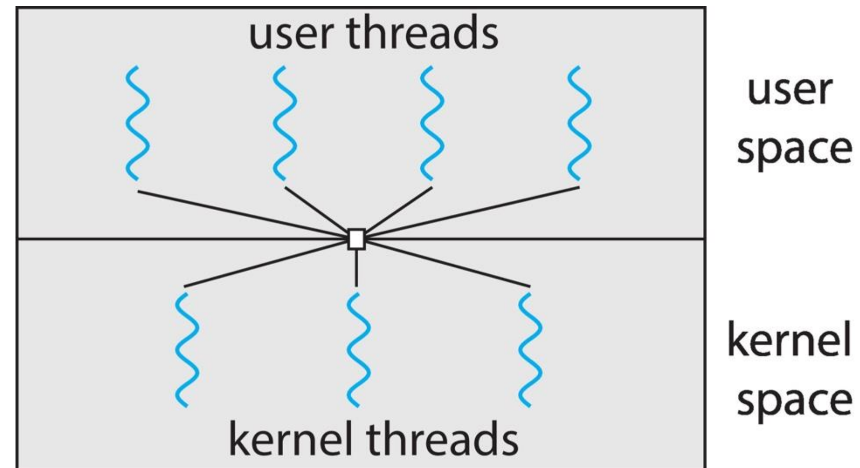
One-to-One

- Each User-Level Thread is mapped onto its own single Kernel Thread giving more concurrency than many-to-one **by allowing other threads to run** if a thread performs a blocking system call.
- Threads don't block each other.
- Thread management is done by the kernel--□Slower!
- Different threads can be run on different CPU's in an MP system.
- **Drawback**: overhead of creating so many kernel threads for an application so there is always a limit.
- Examples:
  - Windows
  - Linux



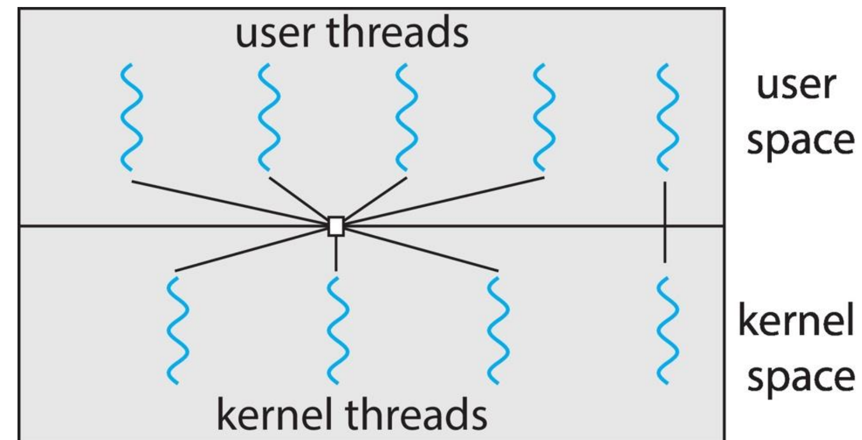
Many-to-Many

- Many user-level threads may be mapped onto a smaller or equal number of kernel threads.
- **Advantages**
- kernel threads run in parallel on a MP system
- If a thread performs a blocking system call, the kernel can schedule another thread for execution
- Examples of systems with combination of User and Kernel Level threads:
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



End of Chapter 4