# Threads

Chapter 4

# Scope

- Overview
- Multicore Programming
- Multithreading Models
- Operating System Examples

# Objectives

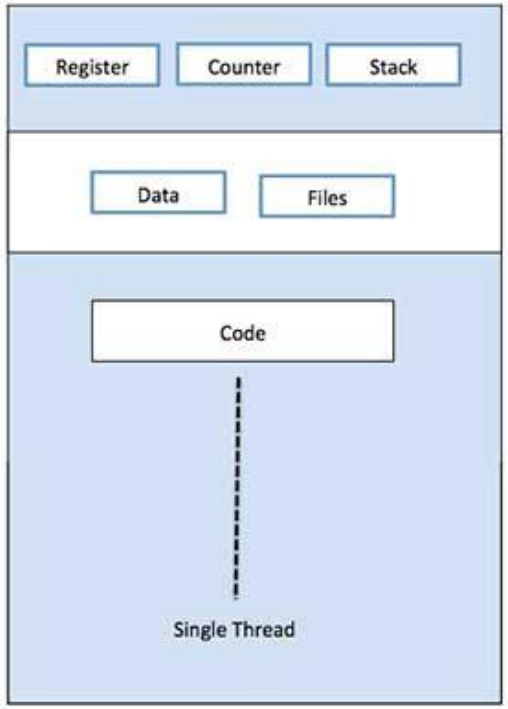
- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To cover operating system support for threads in Windows and Linux

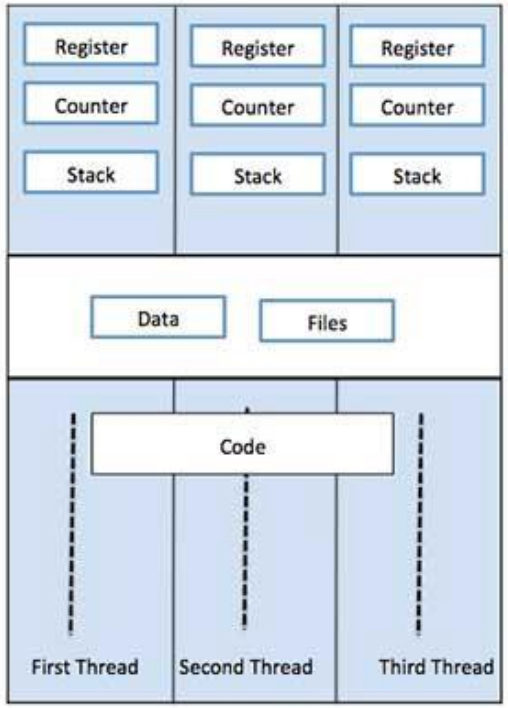
# Motivation

- 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
- Why use threads?
  - Process creation is heavy-weight while thread creation is light-weight
  - Can simplify code, increase efficiency
- Kernels are generally multithreaded

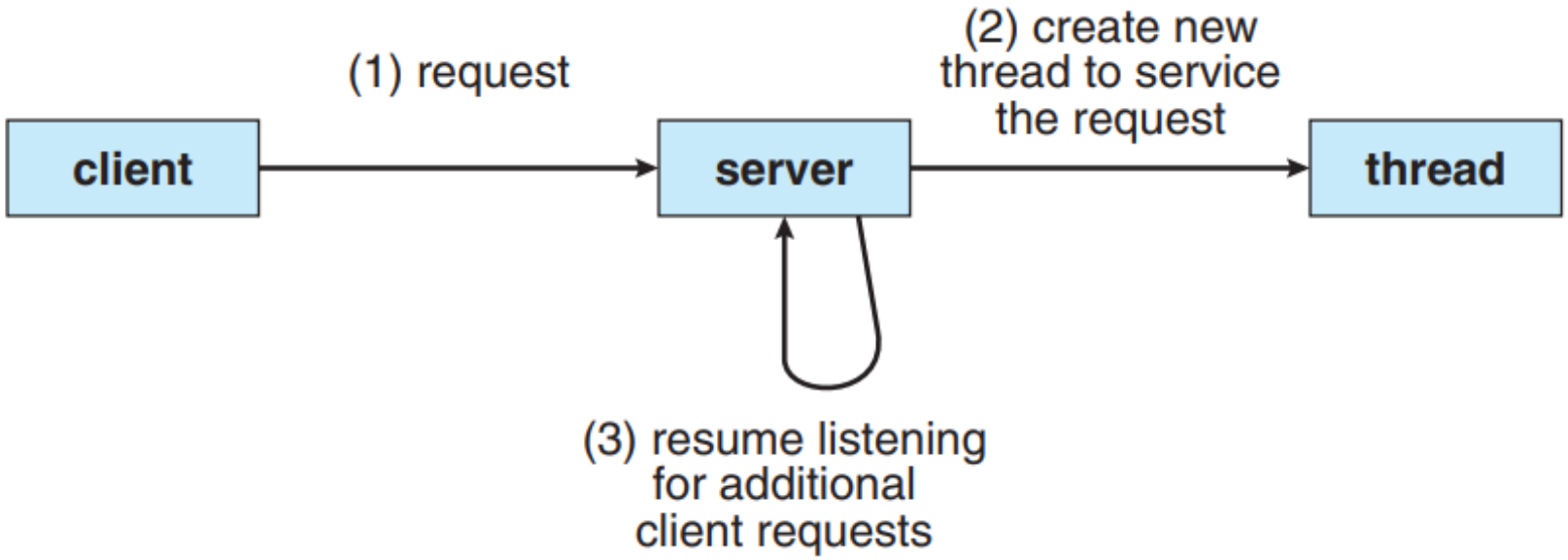
# Threads

- A **thread** is a basic unit of **CPU utilization**;
  - comprises a thread ID, a program counter, a register set, and a stack.
  - It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals
- Examples:
  - A web browser might have one thread display images or text while another thread retrieves data from the network
  - A web server, where multiple threads works on multiple requests sent by clients.


Register
Counter
Stack
Data
Files
Code
Single Thread


Register
Counter
Stack
Register
Counter
Stack
Register
Counter
Stack
Data
Files
Code
First Thread
Second Thread
Third Thread

# Threads




# Multithreaded Server Architecture

# Benefits

**Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces

**Resource Sharing** – threads share resources of process, easier than shared memory or message passing

**Economy** – cheaper than process creation, thread switching lower overhead than context switching

**Scalability** – process can take advantage of multiprocessor architectures

# Multicore Programming

**Multicore or multiprocessor systems putting pressure on programmers**

- Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow parallel execution
- Challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

**Parallelism implies a system can perform more than one task simultaneously**

**Concurrency supports more than one task making progress**

- Single processor / core, scheduler providing concurrency

# Multicore Programming (Cont.)

**Types of parallelism**

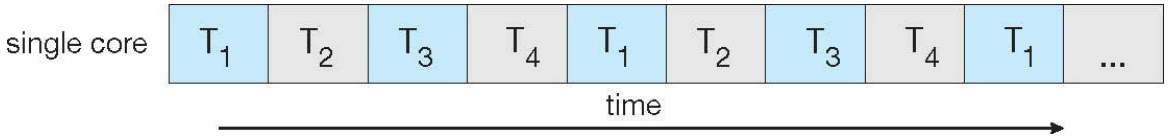
- **Data parallelism – distributes subsets of the same data across multiple cores, same operation on each**
- **Task parallelism – distributing threads across cores, each thread performing unique operation**

**As the number of threads grows, so does architectural support for threading**

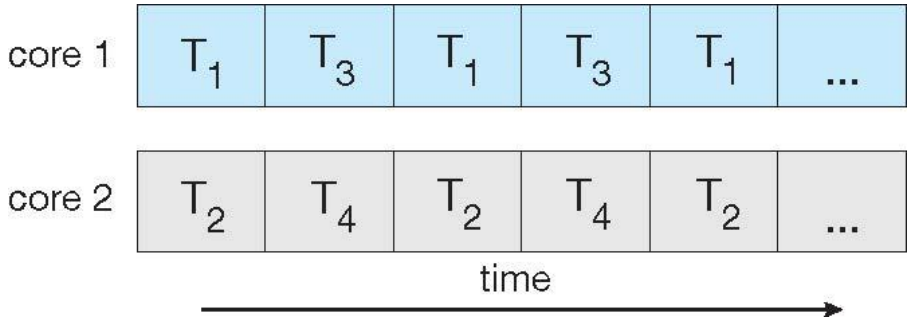
- **CPUs have cores as well as hardware threads**
- **Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core**

# Concurrency vs. Parallelism

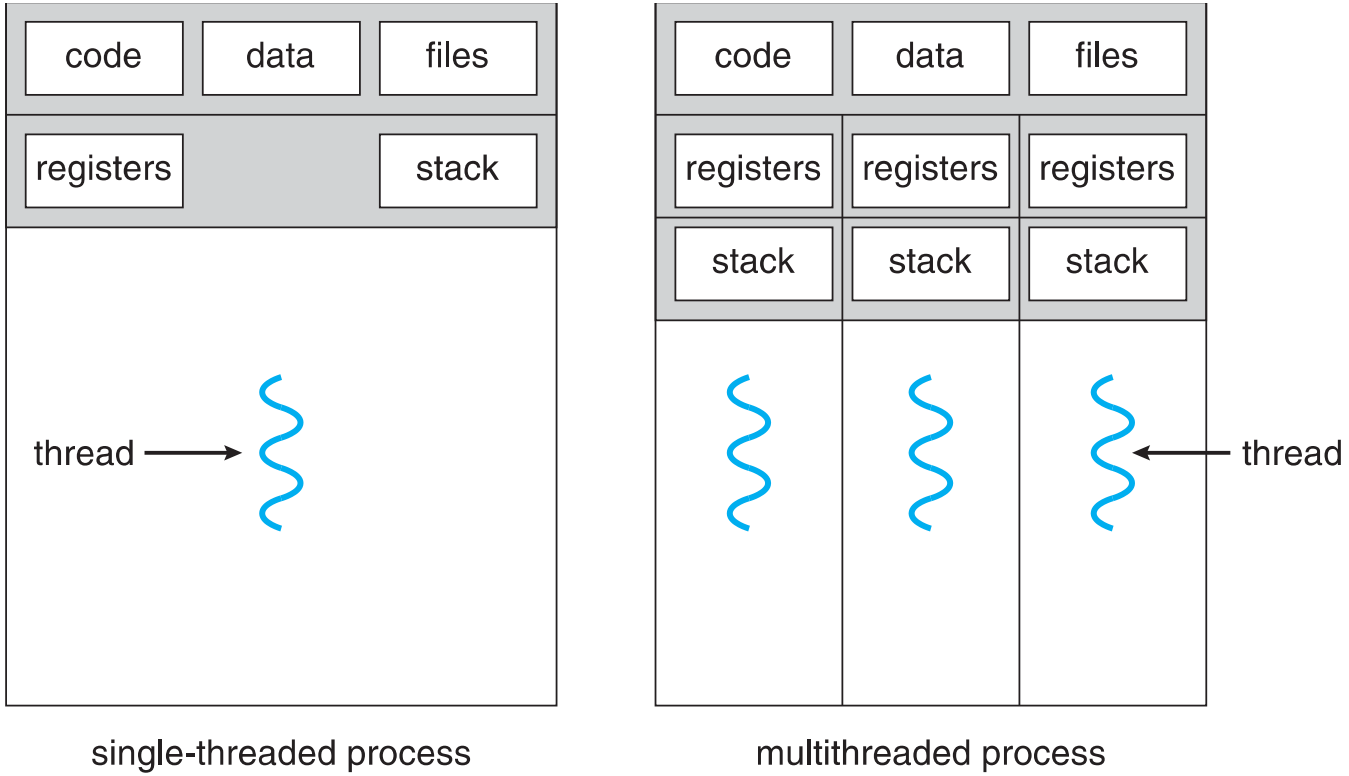
- **Concurrent execution on single-core system**




- **Parallelism on a multi-core system**



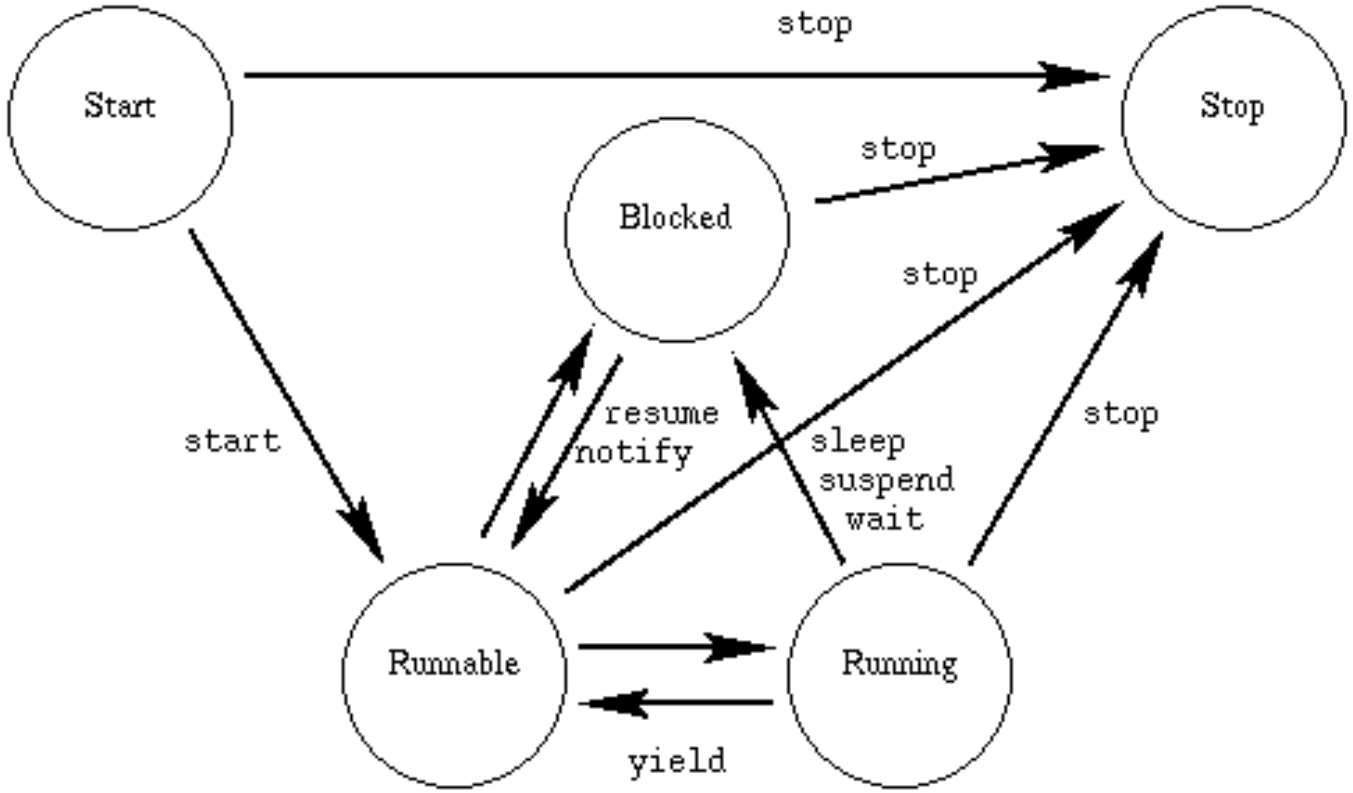
# Single and Multithreaded Processes




single-threaded process

multithreaded process

# Thread States




Source: https://www.w3resource.com/java-tutorial/java-threadclass-methods-and-threadstates.php

# User Threads and Kernel Threads

- **User threads –**
  - Managed by user-level threads library
  - Three primary thread libraries:
    - POSIX Pthreads
    - Windows threads
    - Java threads
- **Kernel threads –**
  - Supported by the Kernel
  - Examples – virtually all general-purpose operating systems, including:
    - Windows
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X

# Multithreading Models

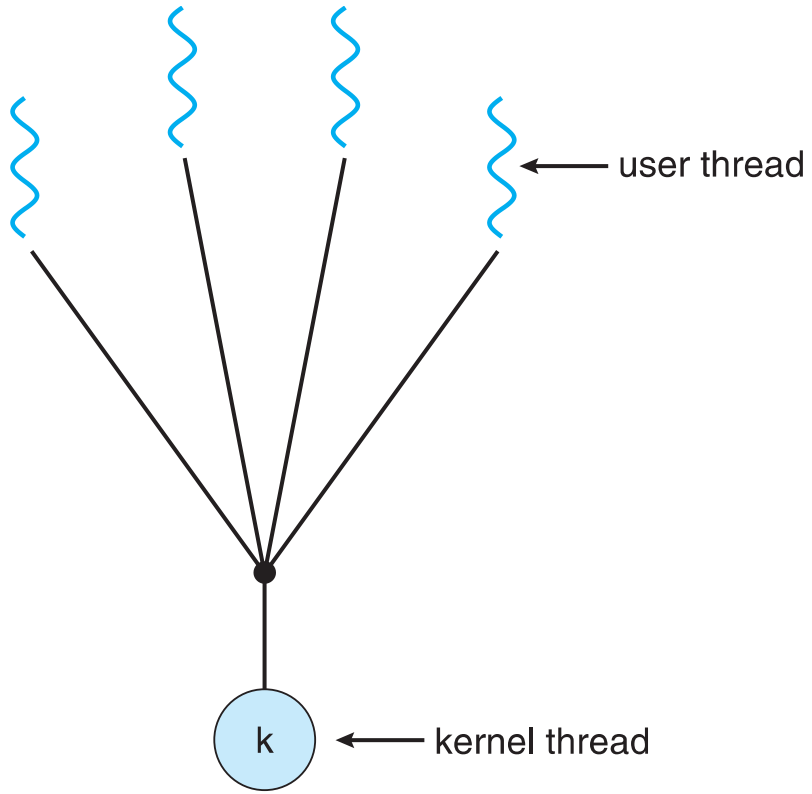
**A relationship must exist between user threads and kernel threads**

Three common ways of establishing such a relationship are:

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

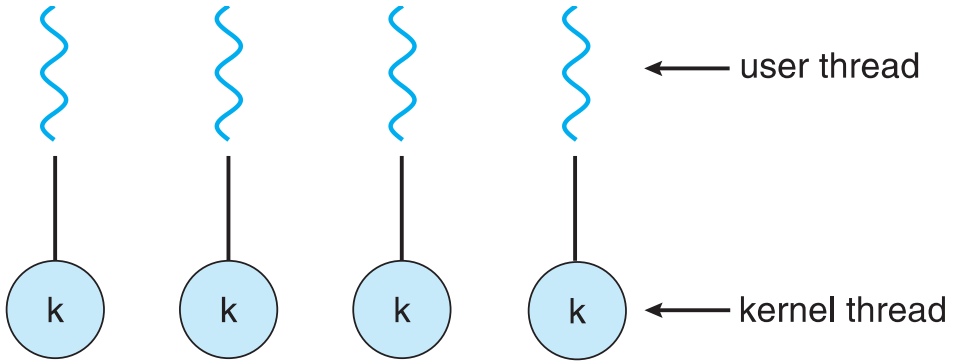
# Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
  - The entire process will block if a thread makes a blocking system call
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



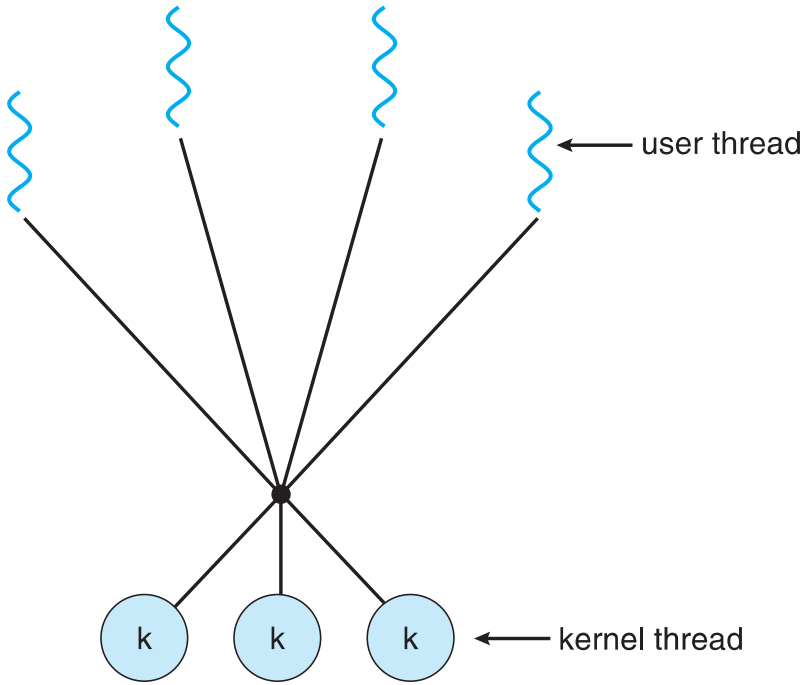
# One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



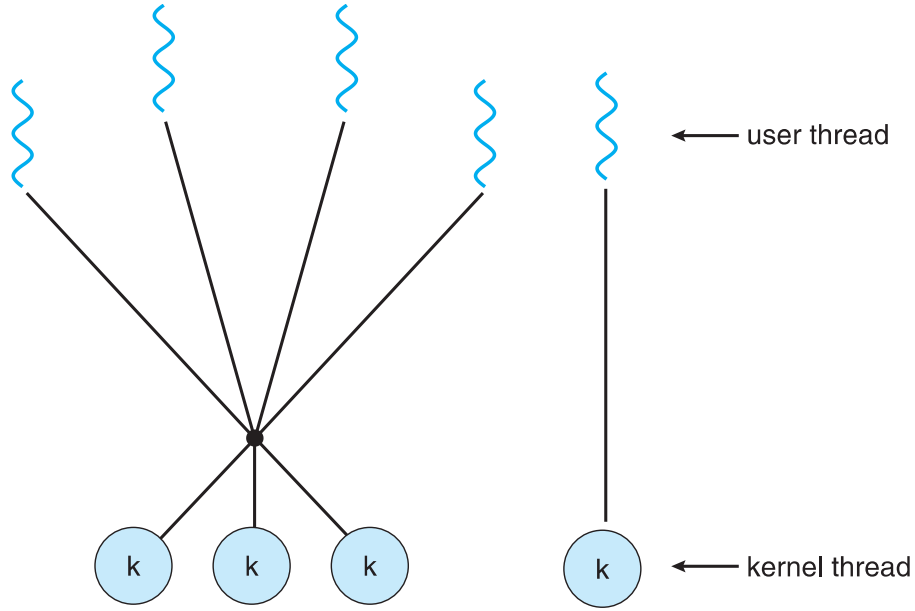
# Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the  operating system to create enough kernel threads
- Solaris prior to version 9
- Windows  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



# OS Threads Examples

- Windows Threads
- Linux Threads

# Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the **one-to-one** mapping, **kernel-level**
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

# Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through `clone()` system call
  - `clone()` allows a child task to share the address space of the parent task (process)

| flag | meaning |
|---|---|
| CLONE_FS | File-system information is shared. |
| CLONE_VM | The same memory space is shared. |
| CLONE_SIGHAND | Signal handlers are shared. |
| CLONE_FILES | The set of open files is shared. |

# Process vs. Thread

| S.N. | Process | Thread |
|---|---|---|
| 1 | Process is heavy weight or resource intensive. | Thread is light weight, taking lesser resources than a process. |
| 2 | Process switching needs interaction with operating system. | Thread switching does not need to interact with operating system. |
| 3 | In multiple processing environments, each process executes the same code but has its own memory and file resources. | All threads can share same set of open files, child processes. |
| 4 | If one process is blocked, then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, a second thread in the same task can run. |
| 5 | Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| 6 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

- https://medium.com/@demozeik/quest-02-03-multithreading-in-operating-system-bfa2d2194a83

# User-Level vs. Kernel-Level Thread

| S.N. | User-Level Threads | Kernel-Level Thread |
|---|---|---|
| 1 | User-level threads are faster to create and manage. | Kernel-level threads are slower to create and manage. |
| 2 | Implementation is by a thread library at the user level. | Operating system supports creation of Kernel threads. |
| 3 | User-level thread is generic and can run on any operating system. | Kernel-level thread is specific to the operating system. |
| 4 | Multi-threaded applications cannot take advantage of multiprocessing. | Kernel routines themselves can be multithreaded. |

https://medium.com/@demozeik/quest-02-03-multithreading-in-operating-system-bfa2d2194a83

# Homework

- Textbook, questions:
  - 4.1
  - 4.2
  - 4.4
  - 4.6
  - 4.8
  - 4.11
  - 4.15
- The homework will be evaluated next session. Be prepared!