`pwd`: **P**rint **W**orking **D**irectory. `ls`: **L**ist directory contents. `cd <dir>`: **C**hange **D**irectory. `man <cmd>`: View the command **man**ual. `sudo <cmd>`: Run a command with root privileges.
`mkdir <dir>`: **M**a**k**e a new **dir**ectory. `touch <file>`: Creates a new, empty file. `cp <src> <dest>`: **C**o**p**y files. Use `-i` flag to prevent accidental overwrites. `mv <src> <dest>`: **M**o**v**e or rename files. `rm <file>`: **R**e**m**ove (delete) files. Use `-r` to remove directories recursively. `cat <file>`: Display file contents. `less <file>`: View file contents one page at a time. `nano` / `vi`: Terminal-based text editors.
find: Searches for files by name, type, size, etc.
`find <path> [expressions]`
- `-name "*.txt"`: Find by filename pattern.
- `-type f`: Find files (`d` for directories).
- `-size +1M`: Find files larger than 1MB.
- `-delete`: Deletes any files found.

grep: Searches inside files for lines matching a pattern (regex).
`grep [opts] <pattern> [file]`
- `-c`: Prints only a **c**ount of matching lines.
- `-r`: Searches **r**ecursively through directories.
- `-l`: Prints on**l**y filenames containing matches.
- `-i`: Performs a case-**i**nsensitive search.

Redirection & Pipes: `> file`: Redirects output, **overwriting** 'file'. `>> file`: Redirects output, **appending** to 'file'. `2> file`: Redirects standard **error**. `2>/dev/null`: Discards all error messages. **Pipe** '`|`': Chains commands. Sends output of the left command as input to the right command.
Shell Environment & Scripting:
Variables: `$USER`, `$PATH`, `$HOME`. View with `env`.
Expansion: `*` (glob), `$(date)` (command sub), `lec_01..13` (brace).
Scripting: Automates commands.
- **Shebang**: Must be first line, e.g., `#/bin/bash`.
- **Execution**: `chmod +x script.sh`, then `./script.sh`.
- **Arguments**: `$1`, `$2`, `$#` (count), `$@` (all).

If-Else Syntax:
```
if [[ condition ]]; then
  # commands...
fi
```
For Loop Syntax:
```
for var in item1; do
  # commands using $var...
done
```
Complex Chained Commands:
1. Count all Python files on the system:
```
sudo find / -type f -name "*.py"
2>/dev/null | wc -l
```
- `find /`: Searches the entire filesystem.
- `-name "*.py"`: Finds files ending in `.py`.
- `2>/dev/null`: Silences permission errors.
- `| wc -l`: Pipes the file list to 'wc' to count the lines.

2. Find the 5 largest files in `/usr/bin`:
```
find /usr/bin -type f | xargs du -h | sort
-rh | head -n 5
```
- `find ...`: Lists all files in `/usr/bin`.
- `| xargs du -h`: Calculates disk usage for each file.
- `| sort -rh`: Sorts reverse (**r**), human-readable (**h**).
- `| head -n 5`: Shows the top 5 results.

Permissions:
Use `ls -l` to view permissions and `chmod` to change them for **u**ser, **g**roup, and **o**thers (**r**ead, **w**rite, e**x**ecute).
- `chmod u+x file`: Adds execute permission for the user.
- `chmod g-w file`: Removes write permission for the group.
- `chmod o=r file`: Sets others' permission to read-only.

Other Useful Commands:
`wc`: Word, line, character count.
`xargs`: Build/execute commands from stdin.
`du`: Report disk usage.
`sort`: Sort lines of text.
`head`/`tail`: Show start/end of a file.
Parallel Concepts:
Processes & Threads:
- **Process:** Program in execution, isolated address space.
- **Thread:** Execution unit within a process, shared address space.
- **Context Switch:** OS saves/restores state to swap processes; costly.

Parallelism & Performance:
- **Thread-Level Parallelism:** Threads run on different CPU cores.
- **Python GIL:** A lock that prevents multiple native threads from executing Python bytecodes at once. Limits CPU-bound parallelism but is released during I/O.
- **Bottlenecks: Compute-bound** (CPU limit), **Memory-bound** (RAM limit), **I/O-bound** (disk limit).

Parallel Programming Strategies:
- **Decomposition:** Break problem into sub-tasks (e.g., **Map-Reduce**, **Embarrassingly Parallel**).
- **Work Scheduling:** Assign tasks. **Static** (pre-defined) vs. **Dynamic** (at runtime).
- **Orchestration:** Communication. **Message Passing** (separate memory) vs. **Shared Memory** (common memory).

Performance Formulas: **Speed-up (S)**: Measures gain from parallelization.

$$S = \frac{T_s}{T_p}$$

Amdahl's Law: Max speed-up, limited by serial portion $(1 - f)$. Let f be the parallelizable fraction and N be the number of workers.

$$S \leq \frac{1}{(1-f) + \frac{f}{N}}$$

Roofline Model: Upper performance bound based on hardware.

Perf. $\leq$ min(Peak Perf., AI×Peak Mem. Bandwidth)

Performance Analysis: **perf**: Powerful Linux profiler. Gathers hardware (CPU cycles, cache misses) and software events to find bottlenecks.
Arithmetic Intensity (AI): Ratio of computational work per byte of memory accessed. High AI is compute-bound (good); low AI is memory-bound (bad).

$$\text{AI} = \frac{\text{Total Floating-Point Operations (FLOPS)}}{\text{Total Memory Traffic (Bytes)}}$$

Ex 1: Dot Product `func(float* a, float* b, int n)`
- **FLOPS**: $2n$ (1 mult, 1 add per iteration).
- **Memory**: Total Traffic $= n \times (4 + 4) = 8n$ bytes (read two 4-byte floats per iteration).
- **AI**: $\frac{2n}{8n} = \frac{1}{4}$ FLOP/byte. **Memory-bound**. Caches do not help as there is no data reuse.

Ex 2: Matrix Multiplication `C[i][j] += A[i][k] * B[k][j]`
- **FLOPS**: Total FLOPS $= n \times n \times n \times 2 = 2n^3$ (2 ops in 3 nested loops).
- **Memory (Smart Compiler)**: `C[i][j]` is kept in a register. Read A (n^3 times), read B (n^3 times), write C (n^2 times). Traffic $\approx (n^3 + n^3 + n^2) \times 4 \approx 8n^3$ bytes for large n.
- **Memory (Naive Compiler)**: `C[i][j]` is read/written in every `k` iteration. Solution notes $3n^3$ memory accesses. Traffic $= 3n^3 \times 4 = 12n^3$ bytes
- **AI (Smart Compiler)**: For large n, $\frac{2n^3}{8n^3} = \frac{1}{4}$ FLOP/byte. **Memory-bound**.
- **AI (Naive Compiler)**: AI $= \frac{2n^3}{12n^3} = \frac{1}{6}$ FLOP/byte
- **Note**: High data reuse potential. Caching (e.g., tiling) can dramatically increase effective AI.

Multi-Core Architecture:
Memory Access Architectures:
- **UMA (Uniform Memory Access):** All cores have the same latency and bandwidth when accessing any part of the main memory. This is common in consumer-grade computers (with ≤ 16 cores) and is easier to program for.
- **NUMA (Non-Uniform Memory Access):** Different cores have different latencies to different parts of memory. A core can access its 'local' memory faster than memory attached to another core. This is common in high-end servers (with ≥ 16 cores).

Cache Coherence: In a multi-core system, hardware ensures that if one core modifies data in its cache, all other cores see that updated data, maintaining a consistent view of memory. This process is automatic but has a performance cost.
GPU vs. CPU A CPU is like a few highly intelligent brains, good at complex, latency-sensitive tasks. A GPU is like a massive number of less sophisticated brains, excelling at simple, repetitive tasks that can be done in parallel (SIMD on steroids). GPUs have much higher memory bandwidth ($\sim$ 1 TB/s vs $\sim$ 50 GB/s) but lower clock speeds ($\sim$ 2 GHz) than CPUs. **GPU Components**:

- **Streaming Multiprocessor (SM):** An SM is a key component of a GPU, roughly analogous to a CPU core but designed for massive parallelism.
- **Streaming Processor (SP):** These are the 'cores' within an SM. They contain execution units but lack complex control logic or private caches.

GPU Hierarchy:
- **Warp:** A group of 32 threads that all execute the same instruction at the same time on different data (a model called SIMT - Single Instruction, Multiple Threads).
- **Thread Block:** A group of warps that execute on the same SM and can share data through its fast shared memory.
- **Grid:** A group of all the thread blocks that a programmer launches to run a particular task.

Branch Divergence: In a GPU, if threads within the same warp take different paths in an if-else statement (branching), the warp must execute both paths sequentially. This can significantly reduce performance.
Shared-Memory Computing Fundamentals: Shared-memory parallel computing is a model where multiple workers, known as **threads**, operate within a single process. All threads share access to the same main memory, though they might have their own private "workspace". This approach is also called **"multithreading"**.
Race Conditions: This is a major issue where the final result of a program depends on the non-deterministic order in which threads are executed. This often happens because common operations are not **atomic**, meaning they are not a single, indivisible step. For example, an increment operation (`counter++`) is actually three steps:
- Load the value from memory into a register.
- Increment the value in the register.
- Write the new value back to memory.

Memory Models & Reordering CPUs can reorder instructions to improve performance, as long as the final result is the same for a single thread. In multi-core systems, this can cause unexpected behavior.
- **Weak Memory Model**: Allows aggressive reordering for higher performance.
- **Strong Memory Model**: Does not allow such reordering, which is less efficient but safer.

Synchronization Primitives To prevent race conditions and coordinate threads, we use synchronization mechanisms. **Locks / Mutexes**: A **mutex** (short for mutual exclusion) ensures that only one thread can access a "critical section" of code at a time.
- **Main Operations**:
  - `acquire()` or `lock()`: A thread calls this to gain exclusive access. If another thread holds the lock, this thread is blocked (put to sleep) until the lock is released.
  - `release()` or `unlock()`: The thread that acquired the lock calls this to release it, allowing other waiting threads to proceed.
- **Key Feature**: Mutexes are **ownership-based**; the same thread that locks it must be the one to unlock it.

Semaphores: A semaphore is essentially a counter with atomic operations used for signaling between threads.
- **Main Operations**:
  - `acquire()` or `wait()`: Decrements the counter. If the counter goes below zero, the thread blocks.

- `release()` or `signal()`: Increments the counter and wakes up a waiting thread if any exist.

- **Key Feature**: Semaphores are **signal-based**. Unlike a mutex, any thread can call `release()` (signal), regardless of which thread called `acquire()` (wait). They are excellent for managing access to a pool of resources, like in the **producer-consumer problem**.

Condition Variables: Condition variables allow threads to wait for more complex, user-defined conditions to become true. They are always used in conjunction with a mutex.

- **Main Operations**:
  - `wait()`: Atomically releases the associated mutex and blocks the thread until it is notified.
  - `notify()`: Wakes up one waiting thread.
  - `notify_all()`: Wakes up all waiting threads.

Barriers: A barrier is a simple mechanism that blocks a group of threads until all of them have reached the barrier. This is useful for synchronizing the start time of an operation across multiple threads.

- **Main Operation**:
  - `wait()`: The calling thread blocks here until the required number of threads (`parties`) have also called `wait()`.

Common Problems & Solutions:
Producer-Consumer Problem: A classic scenario where "producer" threads add items to a shared buffer and "consumer" threads remove them. Problems include producers adding to a full buffer (**overflow**) or consumers taking from an empty one (**underflow**).
Deadlocks: A situation where two or more threads are blocked forever, each waiting for a resource held by the other.

- **Example**: Thread 1 locks A then tries to lock B, while Thread 2 locks B then tries to lock A.

- **Avoidance**: The simplest way to avoid deadlocks is to enforce a **global lock ordering**. All threads must acquire locks in the same specified order.

Python Implementation Tools: Python provides modules for managing threads and synchronization. `threading` **Module**: This module provides low-level, fine-grained control over threads and includes implementations for all the synchronization primitives discussed (Locks, Semaphores, etc.).

- **Typical Usage Pattern**:
  1. Define a `worker` function.
  2. Create a `threading.Thread` object, passing the `worker` as the target.
  3. Call `t.start()` to begin execution.
  4. Call `t.join()` to wait for the thread to complete.

`concurrent.futures` **Module** This is a higher-level interface for managing pools of threads or processes. It's generally preferred unless you need fine-grained control. It creates a pool of threads that can be reused for multiple tasks.

- **Typical Usage Pattern with** `ThreadPoolExecutor`:
  1. Define a `worker` function that processes data and returns a result.
  2. Use a `with ThreadPoolExecutor(...)` as executor: block to manage the thread pool's lifecycle.
  3. Use `executor.map(worker, data_chunks)` to apply the worker function to each item in an iterable and collect the results.

Example 1:Barrier The barrier ensures that the main thread (thread 0) only proceeds to calculate the final total sum *after* all other worker threads have finished calculating and storing their partial sums.

```python
import threading
from concurrent.futures import ThreadPoolExecutor
import numpy as np
import time

n_workers = 16
partial_sums = [None] * n_workers
final_answer = [0]

# len(data) is divisible by n_workers for simplicity
data = np.random.rand(n_workers * 1_000_000)
barrier = threading.Barrier(n_workers)  ## !

def worker(thread_id):
    start = thread_id * len(data)//n_workers ## !
    end = (thread_id + 1) * len(data)//n_workers ## !
    partial_sums[thread_id] = np.sum(data[start : end])
      ## !

    barrier.wait() ## !

    if thread_id == 0: ## !
        final_answer[0] = np.sum(partial_sums) ## !

def main():
    threads = []
    for i in range(n_workers):
        t = threading.Thread(target=worker, args=(i,))
        threads.append(t)
        t.start()

    for i, t in enumerate(threads):
        t.join()

if __name__ == "__main__":
    start = time.time()
    main()
    elapsed_parallel = time.time() - start
    print(f"Time taken in parallel: {elapsed_parallel} s
      ")

    start = time.time()
    np.sum(data)
    elapsed_serial = time.time() - start
    print(f"Time taken in serial: {elapsed_serial} s")
    print(f"Speed-up = {elapsed_serial/elapsed_parallel
      }")

    print(f"Serial sum: {np.sum(data)}")
    print(f"Parallel sum: {final_answer[0]}")
```

Example 2:Race Condition&The Lock To count all the prime numbers up to a large number N (8 million in this case) and to compare the execution time of three different approaches:Serial, Manual Threading, Thread Pool

```python
import threading
from concurrent.futures import ThreadPoolExecutor
import time
import numpy as np ## !

n_workers = 8
N = n_workers * 1_000_000
count = 0 # counter for number of primes
count_lock = threading.Lock() ## !
idx = np.random.permutation(N) ## !

def is_prime(n):
    if n < 2: ## !
        return False ## !
    if n == 2: ## !
        return True ## !
    if n % 2 == 0: ## !
        return False ## !
    for i in range(3, int(n**0.5) + 1, 2): ## !
        if n % i == 0: ## !
            return False ## !
    return True ## !

def worker(thread_id):
    global count  ## !

    data = (i for i in range(thread_id * (N//n_workers),
        (thread_id+1) * (N//n_workers))) ## !

    for n in data: ## !
        if is_prime(n): ## !
            with count_lock: ## !
                count += 1 ## !

    # for n in idx[thread_id * (N//n_workers): (
      thread_id+1) * (N//n_workers)]:
    #     if is_prime(n):
    #         with count_lock:
    #             count += 1

def serial():
    count = 0

    for i in range(N):
        if is_prime(i):
            count += 1

    return count

def manual_threading():
    threads = []
    for i in range(n_workers):
        t = threading.Thread(target=worker, args=(i,))
        threads.append(t)
        t.start()

    for t in threads:
        t.join()

def thread_pool():
    with ThreadPoolExecutor(max_workers=n_workers) as
      executor: ## !
        executor.map(worker, [i for i in range(n_workers
      )]) ## !

def main():
    start = time.time()
    manual_threading()
    manual_time = time.time() - start

    start = time.time()
    thread_pool()
    thread_pool_time = time.time() - start

    start = time.time()
    serial()
    serial_time = time.time() - start

    print(f"Time taken with manual threading: {
      manual_time} seconds")
    print(f"Time taken with using thread pool: {
      thread_pool_time} seconds")
    print(f"Serial time taken: {serial_time} seconds")

if __name__ == "__main__":
    main()
```

Example 3:Files sort ssh Write a shell script titled 'organize.sh' that automatically organizes files in a directory based on their extensions and provides a summary report.

```bash
#!/bin/bash   !

# check if a directory is given
if [[ $# = 0 ]]; then
    target_directory="."
else
    target_directory=$1
fi

# validate target_directory
# use double quotes around $target_directory to handle
      any blank spaces
if [[ ! -e "$target_directory" ]]; then
    echo "Error: The given directory does not exist"
    exit 1
fi

if [[ ! -d "$target_directory" ]]; then
    echo "Error: The given path is not a directory"
    exit 1
fi

# check in case the directory is not readable
if [[ ! -r "$target_directory" ]]; then
    echo "Error: The given directory is not readable."
    exit 1
fi

# counters to keep track of various extensions
total_files=0
images_count=0
audio_count=0
videos_count=0
documents_count=0
scripts_count=0
others_count=0

# various extensions
image_ext="jpg jpeg png gif bmp"
audio_ext="mp3 wav"
video_ext="mp4 wmv mov mvi"
doc_ext="pdf doc docx txt md"
script_ext="sh py js php"

for file in "$target_directory"/*; do
    # skip if it is not a regular file
    if [[ ! -f "$file" ]]; then
        continue
    fi

    # skip if it is a hidden file
    if [[ "$(basename "$file")" == .* ]]; then
        continue
    fi

    total_files=$((total_files + 1))

    # check file type
    file_type=""

    for ext in $image_ext; do
        if [[ "$file" == *.$ext ]]; then
            file_type="image"
            break
        fi
    done

    for ext in $audio_ext; do
        if [[ "$file" == *.$ext ]]; then
            file_type="audio"
            break
        fi
    done

    for ext in $video_ext; do
        if [[ "$file" == *.$ext ]]; then
            file_type="video"
            break
        fi
    done

    for ext in $doc_ext; do
        if [[ "$file" == *.$ext ]]; then
            file_type="doc"
            break
        fi
    done

    for ext in $script_ext; do
        if [[ "$file" == *.$ext ]]; then
            file_type="script"
            break
        fi
    done

    if [[ -z "$file_type" ]]; then
        file_type="other"
    fi

    echo "$file_type"

    # moving files depending on their extensions
    # it is much better to use case statements here, but
      since I haven't covered it, oh well
    if [[ "$file_type" == "image" ]]; then
        mkdir -p "$target_directory"/Images
        mv "$file" "$target_directory"/Images
        images_count=$((images_count + 1))
    fi

    if [[ "$file_type" == "audio" ]]; then
        mkdir -p "$target_directory"/Audio
        mv "$file" "$target_directory"/Audio
        audio_count=$((audio_count + 1))
    fi

    if [[ "$file_type" == "video" ]]; then
        mkdir -p "$target_directory"/Videos
        mv "$file" "$target_directory"/Videos
        videos_count=$((videos_count + 1))
    fi

    if [[ "$file_type" == "doc" ]]; then
        mkdir -p "$target_directory"/Documents
        mv "$file" "$target_directory"/Documents
        documents_count=$((documents_count + 1))
    fi

    if [[ "$file_type" == "script" ]]; then
        mkdir -p "$target_directory"/Scripts
        mv "$file" "$target_directory"/Scripts
        scripts_count=$((scripts_count + 1))
    fi

    if [[ "$file_type" == "other" ]]; then
        mkdir -p "$target_directory"/Others
        mv "$file" "$target_directory"/Others
        others_count=$((others_count + 1))
    fi
done

# print out summary report
echo "Total number of files processed: $total_files"
echo "Number of image files moved: $images_count"
echo "Number of audio files moved: $audio_count"
echo "Number of video files moved: $videos_count"
echo "Number of document files moved: $documents_count"
echo "Number of script files moved: $scripts_count"
echo "Number of other files moved: $others_count"
```