Introduction to IT Systems for Beginners

Maciej Gowin @ CoderBrother

Operating Systems

What is an Operating System?

An operating system (OS) manages hardware and software resources and provides services to applications.

Popular Operating Systems:

- **Unix:** A multi-user, multitasking OS primarily used in servers and workstations.
- **Linux:** Open-source OS, Unix-based, used in a variety of systems from servers to desktops.
- **Windows:** Proprietary OS developed by Microsoft, dominant in personal computers.
- **macOS:** Unix-based OS developed by Apple for its Mac computers.

Operating Systems: Key Features

OS	Unix	Linux	Windows	macOS
Developer	AT&T Bell Labs (originally)	Community-driven (e.g., Linus Torvalds)	Microsoft	Apple Inc.
Source Code	Proprietary or open (varies by version)	Open-source (free to modify and distribute)	Proprietary	Proprietary
User Interface	Command-line (GUI available on some)	Command-line and Graphical (e.g., GNOME, KDE)	Graphical (GUI)	Graphical (GUI, Aqua interface)
Primary Use	Servers, workstations, large systems	Servers, desktops, embedded systems	Personal computers, gaming, enterprise	Personal computers, creative work
File System	UFS, ZFS, and others	EXT, BTRFS, XFS, and others	NTFS, FAT32	APFS (latest), HFS+

Operating Systems: Comparison

Criteria	Unix	Linux	Windows	macOS
Cost	Typically licensed, paid	Free and open-source	Paid (licensed)	Paid (comes with Mac devices)
Customization	Limited customization	Highly customizable	Limited customization	Minimal customization
Security	Highly secure, used in critical systems	Very secure, frequent updates	Moderate security, targeted by malware	Secure, less targeted than Windows
Software Support	Limited user-friendly software	Strong community and growing commercial support	Extensive software, gaming support	Focused on Apple ecosystem, creative tools
Performance	High stability, used in servers	Efficient, runs on diverse hardware	Heavy on system resources but widely supported	High performance, optimized for Apple hardware

Operating Systems: Use Cases

- **Unix**
  - Enterprise environments, academic institutions, servers, and mainframes.
- **Linux**
  - Servers, cloud computing, software development, desktops for tech enthusiasts.
- **Windows**
  - Personal computing, enterprise environments, gaming, and productivity.
- **macOS**
  - Graphic design, video editing, software development, creative professionals.

Operating Systems: Differences

- **Unix**
  - Older, stable, and widely used in servers and mainframes.
- **Linux**
  - Flexible, free, and highly customizable, with a strong presence in servers and desktops.
- **Windows**
  - User-friendly, supports a vast amount of software, and dominant in personal computing.
- **macOS**
  - Secure, optimized for Mac hardware, popular in creative industries.

Linux Distribution

A **Linux distribution** (or distro) is a version of the Linux operating system that includes **the Linux kernel** along with additional software like desktop environments, package managers, and system utilities.

Why are there Multiple Distributions?

Linux is **open-source**, so different communities and organizations create their own versions, customizing them for specific use cases (e.g., desktop use, servers, security).

Popular Linux Distributions

1. Ubuntu

- Focus: User-friendly, beginner-friendly, and widely supported.
- Use Case: Desktops, servers, cloud computing.
- Package Manager: APT (Debian-based).

2. Fedora

- Focus: Cutting-edge features, stability.
- Use Case: Developers, Red Hat ecosystem.
- Package Manager: DNF (RPM-based).

Popular Linux Distributions

3. Debian:

- Focus: Stability, open-source philosophy.
- Use Case: Servers, desktops with long-term support.
- Package Manager: APT (Debian-based).

4. CentOS/RHEL (Red Hat Enterprise Linux):

- Focus: Enterprise-level stability, security.
- Use Case: Enterprise servers, corporate environments.
- Package Manager: YUM/DNF (RPM-based).

Popular Linux Distributions

5. Arch Linux:

- Focus: Customization, simplicity, and user control.
- Use Case: Power users, those who want full control over their OS.
- Package Manager: Pacman.

What is the Linux Command Line?

The **Linux Command Line** (also called shell or console) is a text-based interface used to interact with the operating system.

What is the Linux Command Line?

What is a Shell?

- The shell is a program that takes commands from the user and instructs the operating system to execute them.
- Common shells: Bash, Zsh.

Why Use the Command Line?

- More control over the system.
- Efficient for automating tasks and managing files.
- Used in server administration, scripting, and development.

Commonly Used Linux Shells

1. Bash (Bourne Again Shell)

- The default shell on most Linux distributions.
- Known for its simplicity and power in scripting.

2. Zsh (Z Shell)

- Advanced features like auto-completion and customization.
- Popular with developers and power users.

Shell: Basic Linux Commands

- `ls` - list directory contents
- `cd` - change directory
- `cp` - copy files
- `mv` - move or rename files
- `rm` - remove files or directories
- `mkdir` - create a new directory
- `cat` - display file content
- `pwd` - show current directory
- `df` - display disk space usage
- `top` - show running processes

Shell: Sample usage

```
gowinm:~> mkdir test
gowinm:~> cd test
gowinm:~/test> touch file1
gowinm:~/test> touch file2
gowinm:~/test> ls
file1   file2
gowinm:~/test> rm file1
gowinm:~/test> ls
file2
gowinm:~/test> pwd
/Users/gowinm/test
gowinm:~/test>
```

Using the Shell for Automation

Shell Scripting

- Scripts allow you to automate repetitive tasks using the command line.
- A script is simply a file containing a series of commands.

Basic Shell Script Example

```
#!/bin/bash
echo "Hello, World!"
```

Benefits of Automation:

- Save time on tasks like backups, file organization, and system maintenance.
- Useful in system administration, DevOps, and development environments.

Key Benefits of the Linux Command Line

- **Efficiency**
  - Perform tasks faster with keyboard commands rather than clicking through menus.
- **Scripting & Automation**
  - Automate repetitive tasks with shell scripts.
- **System Control**
  - Greater control over system processes, file permissions, and network settings.
- **Powerful Tools**
  - Access to tools like grep (search), awk (text processing), and sed (stream editor).

What is a Process in an Operating System?

A process is an instance of a program that is currently being executed by the operating system.

Key Attributes

- **Own Memory Space:** Each process operates in its own isolated memory space.
- **System Resources:** Each process is allocated resources like CPU, memory, and I/O devices.
- **Independence:** Processes run independently of each other; they don’t share memory or data.

What is a Process in an Operating System?

Use Case Example

- Running different programs simultaneously, like a web browser and a word processor.

Process Lifecycle

Creation	The OS creates a new process.
Execution	The process runs and performs tasks.
Waiting	The process waits for resources (e.g., I/O operations).
Termination	The process finishes and is removed from the system.

What is Process Scheduling?

Process Scheduling is the activity of the operating system to **determine which process should be executed** by the CPU at any given time.

Goals

- Efficient utilization of the CPU.
- Ensuring fair execution for all processes.
- Minimizing response time and maximizing throughput.

Components:

- Scheduler: Decides the order of process execution.
- Dispatcher: Loads the selected process onto the CPU.

Process States and Scheduling

Process States

New	The process is being created.
Ready	The process is waiting to be assigned to the CPU.
Running	The process is being executed on the CPU.
Waiting	The process is waiting for some event (e.g., I/O).
Terminated	The process has finished execution.

Process States and Scheduling

Transition Between States:

Ready → Running	The scheduler picks a process from the ready queue.
Running → Waiting	The process needs to wait for I/O.
Running → Ready	If a process is preempted or a higher-priority process arrives.
Waiting → Ready	I/O completes, and the process is ready to run.

Scheduling Algorithms

- **First-Come, First-Served (FCFS)**
  - Processes are scheduled in the order they arrive.
- **Shortest Job First (SJF)**
  - The process with the shortest execution time is scheduled next.
- **Round Robin (RR)**
  - Each process gets a fixed time slice to execute before the next process is scheduled.
- **Priority Scheduling**
  - Processes are scheduled based on their priority.

Process Execution and Context Switching

- **Context Switching**
  - The process of storing the state of a running process and loading the state of the next process to be executed.
  - Occurs when the CPU switches from one process to another.
- **Context Switching Steps**
  - Save the current process state (registers, program counter, etc.).
  - Load the next process state.
  - Resume execution of the next process.
- **Context Switching Overhead**
  - Involves time and system resources to switch between processes.

What is a Thread in an Operating System?

A thread is the smallest unit of **execution within a process**. It is a **"lightweight process"** that shares the same resources (like memory) as other threads in the same process.

Key Attributes

- **Shared Memory:** Threads in the same process share memory and data with each other.
- **Lightweight:** Threads are easier to create and manage than full processes because they share resources.
- **Multithreading:** A single process can run multiple threads concurrently, allowing tasks to be performed in parallel.

What is a Thread in an Operating System?

Use Case Example

- A web browser using multiple threads to handle separate tabs or background tasks.

Process vs. Thread: Key Differences

Criteria	Process	Thread
Memory	Separate memory space	Shared memory within process
Creation	High overhead, resource-intensive	Lightweight, minimal overhead
Execution	Independent execution	Concurrent execution within a process

What is Multithreading?

Multithreading is a feature that allows multiple threads (smaller units of a process) to run concurrently within a single process, sharing the same resources like memory and data.

Key Concepts

- **Thread:** The smallest unit of execution in a program. Threads run within the same process and share resources.
- **Concurrency:** Multiple threads can run in overlapping time periods, providing more efficient use of the CPU.
- **Parallelism:** With multicore processors, threads can be executed truly in parallel on different CPU cores.

Advantages of Multithreading

- **Improved Performance:** Faster execution by dividing tasks across multiple threads.
- **Resource Sharing:** Threads share memory and resources, leading to less overhead than processes.
- **Responsiveness:** Applications remain responsive (e.g., a program can continue to run while performing background tasks like I/O).

Multithreading Use Cases

- **Web Browsers:** Each tab can run as a separate thread.
- **Video Games:** Different threads handle game logic, rendering, and physics simultaneously.
- **Servers:** Handle multiple requests concurrently by using threads for each connection.

Multithreading Challenges

- **Synchronization:** Managing access to shared resources between threads to avoid conflicts (e.g., using locks or semaphores).
- **Context Switching:** Frequent switching between threads can lead to performance overhead.

Common Issues in Multithreading

While multithreading improves performance by allowing concurrent execution, it also introduces challenges due to the shared use of resources like memory.

Key Multithreading Issues

Race Conditions

Occurs when multiple threads access and modify shared resources concurrently, and the final outcome depends on the order in which the threads execute.

Solution

- Locks/Mutexes: Ensure only one thread can access the shared resource at a time.

Key Multithreading Issues

Deadlocks

A situation where two or more threads are blocked forever, each waiting for a resource held by the other thread(s).

Solution

- Avoid Circular Wait: Use a fixed order for acquiring resources.
- Deadlock Detection: Implement algorithms to detect and recover from deadlocks.

Key Multithreading Issues

Starvation

Occurs when a thread is perpetually denied access to resources because other threads are given preference.

Solution

- Priority Inversion Control: Ensure that lower-priority threads are not indefinitely blocked by higher-priority threads.
- Fair Scheduling: Implement fair scheduling algorithms to ensure every thread gets a chance to execute.

Key Multithreading Issues

Thread Synchronization Issues

Threads must coordinate their actions when accessing shared data to avoid inconsistencies.
Inconsistent Data: If one thread modifies shared data while another thread is reading it, the data can become inconsistent.

Solution

- Locks and Mutexes: Prevent multiple threads from entering critical sections of code simultaneously.
- Semaphores: Use for signaling and managing access to resources in multi-threaded programs.

Application Operation on OS: From Code to CPU

1. Source Code

- The journey starts with source code written in high-level programming languages (e.g., C, Python, Java).
- This code is human-readable and contains the instructions for the application.

Application Operation on OS: From Code to CPU

2. Compilation/Interpretation

- Compiled Languages (e.g., C, C++):
  - Compiler translates the **source code** into **machine code** (binary instructions) specific to the CPU architecture.
- Interpreted Languages (e.g., Python, JavaScript):
  - An **interpreter** reads the code line by line and executes it, often using an intermediate bytecode (like in Java) or directly interpreting it.

Application Operation on OS: From Code to CPU

3. Executable Code

- The result of compilation is an **executable file** (e.g., .exe, .out) that contains machine code.
- The executable file is ready to be loaded into memory for execution.

Application Operation on OS: From Code to CPU

4. Loading into Memory

- The **Operating System (OS)** loads the executable code into the system's memory (RAM).
- The **loader** sets up the memory space for the program and prepares it for execution by the CPU.

Application Operation on OS: From Code to CPU

5. CPU Execution

- The OS sends the loaded instructions to the CPU:
    - **Fetch:** The CPU fetches the next instruction from memory.
    - **Decode:** The CPU decodes the binary instruction to understand what action to perform.
    - **Execute:** The CPU performs the operation (e.g., arithmetic, logical operations).
    - **Write Back:** Results are written back to memory or registers.

Application Operation on OS: From Code to CPU

6. System Resources

- The OS manages **system resources** such as memory, I/O devices, and CPU scheduling to ensure smooth execution of the program.
- The **Process Scheduler** allocates CPU time to different applications to ensure multitasking.