- **`ls`**: Lists files and directories in the current directory. The `-l` option shows detailed information including permissions, number of links, owner, group, size, and timestamp.

- **`pwd`**: Stands for "print working directory." Displays the absolute path of the current directory you are in.

- **`cd`**: Stands for "change directory." Changes the current directory to the one specified. For example, `cd /home/user` moves to the `/home/user` directory.

- **`mkdir`**: Stands for "make directory." Creates a new directory with the name provided. For example, `mkdir new_folder` creates a directory named `new_folder`.

- **`mv`**: Stands for "move." Moves or renames files and directories. For example, `mv file.txt /home/user` moves `file.txt` to `/home/user`.

- **`cp`**: Stands for "copy." Copies files or directories. For example, `cp file.txt copy_file.txt` duplicates `file.txt` to `copy_file.txt`.

- **`rm`**: Stands for "remove." Deletes files or directories. For example, `rm file.txt` removes `file.txt`. Use with caution as it doesn't move files to a trash/recycle bin.

- **`touch`**: Creates an empty file or updates the timestamp of an existing file. For example, `touch newfile.txt` creates an empty file named `newfile.txt`.

- **`ln`**: Stands for "link." Creates hard or symbolic links to files. For example, `ln -s /path/to/file linkname` creates a symbolic link named `linkname` pointing to `/path/to/file`.

- **`clear`**: Clears the terminal screen.

- **`cat`**: Concatenates and displays the contents of files. For example, `cat file.txt` prints the contents of `file.txt` to the terminal.

- **`echo`**: Prints text to the terminal. For example, `echo "Hello, World!"` outputs "Hello, World!" to the terminal.

- **`less`**: Allows viewing of the contents of a file one screen at a time, with the ability to scroll forward and backward. For example, `less file.txt` opens `file.txt` in this view.

- **`man`**: Displays the manual page for a command. For example, `man ls` shows the documentation for the `ls` command.

- **`whoami`**: Prints the username of the current user.

- **`uname`**: Displays system information. The `-a` option shows all available information including kernel version and system architecture.

- **`tar`**: Stands for "tape archive." Used for creating and manipulating archive files. For example, `tar -czvf archive.tar.gz /path/to/directory` creates a compressed archive of a directory.

- **`grep`**: Searches for a specified pattern within files. For example, `grep "search_term" file.txt` searches for "search_term" in `file.txt`.

- **`head`**: Displays the first few lines of a file. The `-n` option specifies the number of lines. For example, `head -n 10 file.txt` shows the first 10 lines of `file.txt`.

- **`tail`**: Displays the last few lines of a file. The `-n` option specifies the number of lines. For example, `tail -n 10 file.txt` shows the last 10 lines of `file.txt`.

- **`diff`**: Compares files line by line and displays differences. For example, `diff file1.txt file2.txt` shows the differences between `file1.txt` and `file2.txt`.

- **`comm`**: Compares two sorted files line by line and outputs three columns: lines unique to the first file, lines unique to the second file, and lines common to both files. For example, `comm file1.txt file2.txt`.

- **`cmp`**: Compares two files byte by byte and reports the first difference found. For example, `cmp file1.txt file2.txt`.

- **`sort`**: Sorts lines of text files. For example, `sort file.txt` sorts the lines in `file.txt` in ascending order.

- **`zip`**: Compresses files into a zip archive. For example, `zip archive.zip file.txt` creates a zip file named `archive.zip` containing `file.txt`.

- **`export`**: Sets environment variables. For example, `export PATH=$PATH:/new/path` adds `/new/path` to the `PATH` environment variable.

- **`unzip`**: Extracts files from a zip archive. For example, `unzip archive.zip` extracts the contents of `archive.zip`.

- **`ssh`**: Stands for "secure shell." Connects to a remote machine securely. For example, `ssh user@hostname` connects to `hostname` with the username `user`.

- **`ps`**: Displays information about active processes. The `aux` options show detailed information about all running processes.

- **`service`**: Used to manage system services. For example, `service apache2 restart` restarts the Apache2 service.

- **`kill`**: Sends signals to processes, usually to terminate them. For example, `kill 1234` sends a termination signal to the process with PID 1234.

- **`killall`**: Sends signals to processes by name. For example, `killall firefox` terminates all instances of Firefox.

- **`mount`**: Mounts a filesystem to a directory. For example, `mount /dev/sda1 /mnt` mounts the filesystem on `/dev/sda1` to the `/mnt` directory.

- **`df`**: Displays disk space usage for file systems. The `-h` option shows the output in a human-readable format.

- **`chmod`**: Changes file permissions. For example, `chmod 755 script.sh` sets the permissions of `script.sh` to read, write, and execute for the owner, and read and execute for others.

- **`chown`**: Changes the owner and group of a file. For example, `chown user:user file.txt` changes the owner and group of `file.txt` to `user`.

- **`ifconfig`**: Displays or configures network interfaces. For example, `ifconfig eth0` shows network configuration for the `eth0` interface.

- **`traceroute`**: Shows the path packets take to reach a network destination. For example, `traceroute google.com` traces the route to `google.com`.

- **`wget`**: Non-interactively downloads files from the web. For example, `wget http://example.com/file.zip` downloads `file.zip` from `example.com`.

- **`ufw`**: Stands for "Uncomplicated Firewall." Manages firewall rules. For example, `ufw enable` activates the firewall.

- **`iptables`**: Configures IP packet filter rules. For example, `iptables -L` lists all current iptables rules.

- **`apt`**: A package management tool for Debian-based systems. For example, `apt update` updates the package lists.

- **`pacman`**: A package manager for Arch-based systems. For example, `pacman -Syu` updates the system’s packages.

- **`yum`**: A package manager for Red Hat-based systems. For example, `yum update` updates packages.

- **`rpm`**: Manages RPM packages. For example, `rpm -ivh package.rpm` installs an RPM package.

- **`sudo`**: Executes commands with superuser privileges. For example, `sudo apt install package` installs a package with elevated permissions.

- **`alias`**: Creates shortcuts for commands. For example, `alias ll='ls -la'` creates a shortcut `ll` for `ls -la`.

- **`dd`**: Copies and converts files at a low level. For example, `dd if=/dev/sda of=/dev/sdb bs=4M` copies data from `/dev/sda` to `/dev/sdb` with a block size of 4MB.

- **`dmesg`**: Displays kernel ring buffer messages. For example, `dmesg | grep error` filters kernel messages for errors.

- **`whereis`**: Locates the binary, source, and manual page files for a command. For example, `whereis ls` shows the locations of the `ls` command's binary, source, and man page.

- **`whatis`**: Provides a brief description of a command. For example, `whatis ls` gives a one-line description of the `ls` command.

- **`top`**: Displays a real-time view of system processes, including their CPU and memory usage. It updates the display regularly and allows interaction to manage processes.

- **`passwd`**: Changes a user’s password. Running `passwd` prompts you to enter a new password for the current user. To change another user’s password, you need superuser privileges (e.g., `sudo passwd username`).

- **`useradd`**: Creates a new user account. For example, `useradd newuser` adds a user named `newuser` to the system. This command typically requires superuser privileges to execute.

IMP THINGS

User Space vs. Kernel Space

1. **User Space:**
    - **Definition:** This is the memory area where user applications and processes run.
    - **Access:** User space applications run with restricted access to system resources to ensure stability and security. They cannot directly access hardware or system memory.
    - **Example:** Applications like web browsers, word processors, and games run in user space.
2. **Kernel Space:**
    - **Definition:** This is the memory area where the operating system kernel runs and has direct access to hardware and system resources.
    - **Access:** Kernel space has full access to system resources and hardware, which allows it to manage resources, handle system calls, and execute privileged operations.
    - **Example:** The kernel manages hardware devices, memory management, and system calls.

Interrupts

- **Definition:** Interrupts are signals sent to the processor to indicate that an event needs immediate attention. They interrupt the current execution flow to execute a special routine (interrupt service routine or ISR).
- **Types:**
    - **Hardware Interrupts:** Generated by hardware devices (e.g., keyboard, network card) to signal that they need processing.
    - **Software Interrupts:** Generated by software or applications to request system services (e.g., system calls).

System Calls

- **Definition:** System calls are mechanisms that allow user-space applications to request services from the kernel. They provide an interface between user space and kernel space.
- **Examples:** Common system calls include `read()`, `write()`, `open()`, and `close()`, which handle file operations; `fork()` and `exec()` for process management.
- **Process:**
    1. **User Space Request:** An application makes a system call to perform an operation.
    2. **Context Switch:** The operating system performs a context switch from user mode to kernel mode to handle the request.
    3. **Execution:** The kernel executes the system call and performs the requested operation.
    4. **Return:** Control is returned to user space with the result of the operation.

Context Switching

- **Definition:** Context switching is the process of saving the state of a currently running process and loading the state of another process. This is crucial for multitasking.
- **When It Occurs:** Context switching happens during system calls, interrupts, and process scheduling.

--list of common Linux filesystem directories and their purposes.

- **`/bin` (User Binaries):**
  - Contains essential command binaries that are required for basic system operation and maintenance. These binaries are available to all users and include commands like `ls`, `cp`, and `mv`.
- **`/sbin` (System Binaries):**
  - Contains essential system binaries used for system administration tasks. These binaries are typically used by the system administrator and include commands like `shutdown`, `fsck`, and `mount`.
- **`/etc` (Configuration Files):**
  - Contains system-wide configuration files and shell scripts that are used to configure the system and applications. Examples include `passwd`, `fstab`, and `hosts`.
- **`/dev` (Device Files):**
  - Contains device files that represent hardware devices and virtual devices. For example, `/dev/sda` represents a disk drive, and `/dev/tty` represents a terminal.
- **`/proc` (Process Information):**
  - A virtual filesystem that provides information about running processes and kernel parameters. For example, `/proc/cpuinfo` provides information about the CPU, and `/proc/uptime` shows system uptime.
- **`/var` (Variable Files):**
  - Contains files that are expected to grow in size, such as log files, mail spools, and temporary files. Examples include `/var/log` for log files and `/var/spool` for queued mail.
- **`/tmp` (Temporary Files):**
  - Used for storing temporary files that are created by various applications and processes. Files in this directory are typically deleted on reboot.

- **`/usr` (User Programs):**
  - Contains user-related programs and data. It includes subdirectories such as `/usr/bin` for user binaries, `/usr/lib` for libraries, and `/usr/share` for shared data.
- **`/home` (User Home Directories):**
  - Contains the home directories of all users. Each user has a subdirectory under `/home` where their personal files and configuration settings are stored.
- **`/boot` (Boot Loader Files):**
  - Contains files needed for the boot process, including the Linux kernel, initial RAM disk, and bootloader configuration files. Examples include `vmlinuz` and `initrd`.
- **`/opt` (Optional Apps):**
  - Used for installing optional application software packages. Applications installed here are usually not part of the core distribution and may be third-party applications.
- **`/lib` (System Libraries):**
  - Contains essential shared libraries and kernel modules needed by system binaries. Libraries in this directory are used by programs in `/bin` and `/sbin`.

--Other topics of linux

Access Control List (ACL)

Access Control Lists (ACLs) are used to specify permissions for different users or groups on a file or directory in addition to the standard owner/group/others permissions. ACLs provide finer-grained access control.

Basic ACL Commands:

1. **View ACLs:**

   ```
   bash
   Copy code
   getfacl filename
   ```

2. **Set ACLs:**

   ```
   bash
   Copy code
   setfacl -m u:username:permissions filename
   ```

   - Example: `setfacl -m u:john:rw filename` grants read and write permissions to user `john`.
3. **Remove ACLs:**

```
bash
Copy code
setfacl -x u:username filename
```

   - Example: `setfacl -x u:john filename` removes ACL for user `john`.
4. **Set Default ACLs (for directories):**

```
bash
Copy code
setfacl -d -m u:username:permissions directoryname
```

   - Example: `setfacl -d -m u:john:rw directoryname` sets default ACLs for new files in the directory.

Network Commands

1. **Telnet:**
   - **Purpose:** Used to connect to remote servers and issue commands.
   - **Basic Usage:** `telnet hostname port`
   - **Example:** `telnet example.com 80` connects to `example.com` on port `80`.
2. **FTP (File Transfer Protocol):**
   - **Purpose:** Used for transferring files between computers over a network.
   - **Basic Usage:** `ftp hostname`
   - **Example:** `ftp ftp.example.com` connects to `ftp.example.com`.
3. **SSH (Secure Shell):**
   - **Purpose:** Provides secure access to a remote server over an unsecured network.[Also used to take control of remote server/machine]
   - **Basic Usage:** `ssh username@hostname`
   - **Example:** `ssh user@example.com` connects to `example.com` as `user`.
4. **SFTP (Secure File Transfer Protocol):**
   - **Purpose:** Provides secure file transfer over SSH.
   - **Basic Usage:** `sftp username@hostname`
   - **Example:** `sftp user@example.com` connects to `example.com` for file transfers.
5. **Finger:**
   - **Purpose:** Displays information about users on a remote system.
   - **Basic Usage:** `finger username@hostname`
   - **Example:** `finger user@example.com` shows information about `user` on `example.com`.

System Variables

System variables (also known as environment variables) are used to define system-wide settings and behavior in the shell. Here are some common ones:

1. **`PS1` (Prompt String 1):**
   - **Purpose:** Defines the primary prompt string in the terminal.
   - **Default Example:** `\u@\h:\w\$` which shows `username@hostname:current_directory$`.
   - **Set Example:**

```
bash
Copy code
export PS1="\u@\h:\w\$ "
```

2. **`PS2` (Prompt String 2):**
   - **Purpose:** Defines the secondary prompt string, used when a command is continued.
   - **Default Example:** `>`.
   - **Set Example:**

```
bash
Copy code
export PS2="continue> "
```

3. **`PATH`:**
   - **Purpose:** Defines the directories where the system looks for executable files.
   - **Default Example:** `/usr/local/bin:/usr/bin:/bin`
   - **Set Example:**

```
bash
Copy code
export PATH="$PATH:/new/directory"
```

4. **`HOME`:**
   - **Purpose:** Defines the home directory of the current user.
   - **Default Example:** `/home/username`
   - **Set Example:**

```
bash
Copy code
export HOME="/new/home/directory"
```

5. **`LANG`:**
   - **Purpose:** Defines the system's language and locale settings.
   - **Default Example:** `en_US.UTF-8`
   - **Set Example:**

```
bash
Copy code
export LANG="en_US.UTF-8"
```

Setting System Variables

To set environment variables:

- **Temporarily:** Use the `export` command in the terminal. It will last until the terminal session is closed.

```
bash
Copy code
export VARIABLE_NAME=value
```

- **Permanently:** Add the `export` command to your shell configuration file (e.g., `~/.bashrc`, `~/.bash_profile`, `~/.zshrc`), and then reload the file or restart the terminal.

```
bash
Copy code
echo 'export VARIABLE_NAME=value' >> ~/.bashrc
source ~/.bashrc
```

DAY-2

Process Management

Process management is a crucial aspect of operating systems, handling the execution of processes (programs in execution) efficiently and ensuring smooth multitasking.

User Mode vs. Kernel Mode

- **User Mode**: In this mode, the processor restricts access to hardware and critical system resources. Applications run in user mode, and any attempt to access restricted resources triggers a mode switch to kernel mode.
- **Kernel Mode**: This mode has full access to all system resources, including hardware and memory. The operating system runs in kernel mode to perform privileged tasks. System calls are the primary method for transitioning from user mode to kernel mode.

System Calls

System calls provide the interface between a running program and the operating system. They allow user-level processes to request services from the kernel.

Types of System Calls

1. **File-Related Calls**:
   - `read()`: Reads data from a file.
   - `write()`: Writes data to a file.
   - `delete()`: Deletes a file.
   - `open()`: Opens a file.
   - `close()`: Closes a file.
   - `create()`: Creates a new file.
2. **Process-Related Calls**:
   - `fork()`: Creates a new process by duplicating an existing one.
   - `exit()`: Terminates a process.
   - `wait()`: Makes a process wait for the completion of another process.
   - `exec()`: Replaces the process's memory space with a new program.
   - `getpid()`: Retrieves the process ID.
   - `kill()`: Sends a signal to a process.
3. **Device-Related Calls**:
   - `read()`: Reads data from a device.

    - `ioctl()`: Stands for input/output control; it manipulates the underlying device parameters.
4. **Information-Related Calls**:
    - `getpid()`: Returns the process ID.
    - `gettime()`: Fetches the current time.
    - `sysdata()`: Retrieves system-related information.
5. **Communication-Related Calls**:
    - `wait()`: Waits for a signal or event.
    - `signal()`: Sends signals to processes.
    - `status()`: Checks the status of a process or communication channel.

Process Structure

A process in an operating system is a program in execution, which comprises the following segments:

1. **Code Segment**:
    - Contains the executable code or instructions that the CPU executes.
2. **Data Segment**:
    - Contains the global variables, static variables, and dynamically allocated memory used during execution.
3. **Information Segment**:
    - Stores metadata about the process, such as process ID, open files, and other system-related data.
4. **Memory Segment**:
    - **Heap**: For dynamic memory allocation.
    - **Stack**: For managing function calls and local variables.

Process Life Cycle

1. **New State**:
    - The process is being created. At this stage, the operating system is setting up the necessary resources for the process, such as memory, process control blocks (PCB), and scheduling parameters.
2. **Ready State**:
    - The process is prepared to execute and is waiting in the **ready queue**. It has all the resources it needs except for the CPU. The process will move from the ready state to the running state when the CPU becomes available.
3. **Running State**:
    - The process is currently being executed by the CPU. In this state, the instructions within the process are being executed. If the process needs to wait for a resource (like I/O), it will transition to the wait state.
4. **Wait State** (Blocked State):
    - The process is unable to continue executing because it is waiting for some external event to occur, such as I/O completion or the availability of a resource. Once the event occurs, the process will move back to the ready state.

5. **Terminated State**:
    - The process has finished execution, either by completing its task or by being terminated by the system or another process. In this state, the operating system performs cleanup tasks, such as releasing resources and updating process tables.

Transitions Between States:

- **New → Ready**: After creation, a process moves to the ready state.
- **Ready → Running**: The scheduler selects a process from the ready queue to execute, moving it to the running state.
- **Running → Wait**: If a process needs to wait for an event or resource, it transitions to the wait state.
- **Wait → Ready**: Once the event occurs or the resource becomes available, the process moves back to the ready state.
- **Running → Terminated**: When a process completes its execution or is terminated, it transitions to the terminated state.

This life cycle, with the inclusion of the wait state, provides a more comprehensive view of how a process progresses through different stages in an operating system.

Day-3

1. Convey Effect (Convoy Effect)

- **Definition**: The Convoy Effect occurs when a set of shorter processes gets delayed behind a longer process in a non-preemptive scheduling system like First-Come, First-Served (FCFS).
- **Impact**: This leads to poor CPU utilization and increased waiting times for shorter processes, as they must wait for the longer process to complete before they can execute.

2. Starvation

- **Definition**: Starvation is a situation where a process is perpetually delayed from being scheduled for execution due to the continuous arrival of higher-priority processes.
- **Cause**: Commonly occurs in priority scheduling algorithms where lower-priority processes may never get CPU time if higher-priority processes keep arriving.

3. Round Robin (RR) Scheduling Disadvantages

- **Definition**: Round Robin (RR) is a preemptive scheduling algorithm where each process is assigned a fixed time slice (quantum) to execute.
- **Disadvantages**:
    - **High Context Switching Overhead**: Frequent switching between processes can lead to increased overhead, especially if the time quantum is too small.
    - **Poor Performance for Longer Processes**: Long processes may take much longer to complete due to being repeatedly preempted and placed at the back of the queue.

- **Selection of Quantum Size**: Choosing the optimal time quantum is challenging. If it's too small, it results in too much context switching; if it's too large, it behaves like FCFS.

4. Arrival Time

- **Definition**: The arrival time is the time at which a process arrives in the ready queue and is ready to be executed by the CPU.
- **Importance**: It is used in scheduling algorithms to determine the order of process execution, especially in non-preemptive scheduling.

5. Response Time

- **Definition**: Response time is the time elapsed from when a process enters the ready queue until it gets its first time on the CPU.
- **Formula**: `Response Time = First CPU Execution Time - Arrival Time`
- **Importance**: A lower response time is generally desirable as it indicates that a process has started executing shortly after arrival.

6. Completion Time

- **Definition**: Completion time is the time at which a process finishes its execution and leaves the CPU.
- **Formula**: `Completion Time = Time at which process completes`
- **Importance**: It helps in calculating other metrics like turnaround time.

7. Waiting Time

- **Definition**: Waiting time is the total time a process spends waiting in the ready queue before it gets executed by the CPU.
- **Formula**: `Waiting Time = Turnaround Time - Burst Time`
- **Importance**: Reducing waiting time is a key objective in scheduling algorithms to improve process efficiency.

8. Turnaround Time

- **Definition**: Turnaround time is the total time taken for a process to complete its execution, from the time it arrives in the ready queue until it completes.
- **Formula**: `Turnaround Time = Completion Time - Arrival Time`
- **Importance**: Turnaround time measures how quickly a system can process a task, making it a critical performance metric.

Summary of Relationships:

- **Turnaround Time** = Completion Time - Arrival Time
- **Waiting Time** = Turnaround Time - Burst Time
- **Response Time** = First CPU Execution Time - Arrival Time

Shared Pages and Reentrant Code

Shared Pages:

- **Definition**: Shared pages in a paging system refer to the practice of allowing multiple processes to share the same physical memory pages. This is particularly useful when the same code or data is used by multiple processes, as it reduces the overall memory usage and increases efficiency.
- **Usage**:
    - **Shared Libraries**: Shared pages are commonly used for shared libraries, where multiple programs can use the same code without each program needing its own copy in memory.
    - **Inter-Process Communication (IPC)**: Shared pages can also facilitate communication between processes by allowing them to share data in memory.
    - **Text Segments**: In many operating systems, the text segment (containing the executable code) of a program can be shared among processes, particularly if they are running the same program.
- **Benefits**:
    - **Memory Efficiency**: Reduces the amount of physical memory required, as common code and data are stored only once.
    - **Faster Execution**: Reduces loading time since shared code is already in memory.
    - **Simplified Updates**: Updating shared code (e.g., libraries) in memory automatically benefits all processes using that code.

Reentrant Code:

- **Definition**: Reentrant code (or pure code) is code that can be safely executed by multiple processes or threads simultaneously without interfering with each other. It does not modify itself or rely on shared, modifiable data.
- **Characteristics**:
    - **No Self-Modification**: Reentrant code does not change during execution. It doesn't modify global or static variables, and any variables used by the code are passed as parameters or are local to the function.
    - **Thread-Safe**: Because reentrant code doesn’t rely on shared, modifiable data, it is inherently thread-safe.
- **Example**:
    - **Library Functions**: Many standard library functions (e.g., mathematical functions like `sqrt()`) are reentrant because they operate only on their input parameters and do not modify any global state.
- **Importance**:
    - **Shared Execution**: Reentrant code is critical for systems where the same code needs to be executed by multiple processes simultaneously, such as in multitasking operating systems or in systems with multiple threads.
    - **Memory Sharing**: Since reentrant code doesn’t modify itself, it can be shared across different processes using shared pages, further optimizing memory usage.

Throttling

Throttling refers to the practice of controlling the rate at which a process or resource is used, typically to prevent overuse, maintain performance, or manage resource consumption.

Types of Throttling:

1. **CPU Throttling**:
    - **Definition**: Involves reducing the CPU's speed or limiting the CPU time allocated to a process or application to reduce heat output, save power, or balance load across multiple processors.
    - **Usage**: Often used in mobile devices to prevent overheating or to extend battery life by reducing the CPU clock speed when full power is not needed.
2. **Network Throttling**:
    - **Definition**: Regulates the rate at which data is transferred over a network to prevent congestion, ensure fair bandwidth distribution, or comply with usage policies.
    - **Usage**: Internet service providers (ISPs) may throttle network speeds during peak usage times to manage network load and ensure fair access for all users.
3. **API Throttling**:
    - **Definition**: Limits the number of API requests a client can make to a server within a given timeframe to prevent overloading the server and to ensure fair usage among clients.
    - **Usage**: Common in cloud services, where APIs are accessed by many clients simultaneously, preventing any single client from overwhelming the service.
4. **I/O Throttling**:
    - **Definition**: Controls the rate at which input/output operations are performed on a storage device to prevent bottlenecks and to manage the load on the system.
    - **Usage**: Often used in disk-intensive applications, where too many concurrent I/O operations could degrade system performance.

Benefits of Throttling:

- **Resource Management**: Throttling helps manage limited resources efficiently, ensuring that no single process or user consumes an excessive amount.
- **Performance Stability**: By controlling the rate of resource usage, throttling can help maintain overall system performance and prevent degradation.
- **Fairness**: Throttling ensures that resources are distributed fairly among users or processes, preventing any single entity from monopolizing resources.
- **Protection**: Throttling protects systems from overloading and potential crashes by controlling the pace at which resources are consumed.

Drawbacks:

- **Reduced Performance**: While throttling helps maintain system stability, it can also lead to reduced performance for processes that are throttled.
- **Complexity**: Implementing throttling mechanisms can add complexity to system design and may require careful tuning to avoid negative impacts on user experience.

Summary

- **Shared Pages**: Enable multiple processes to share the same physical memory, improving efficiency and reducing memory usage. Reentrant code is key to safely sharing executable code between processes.
- **Throttling**: Controls the rate of resource usage to manage performance, prevent overloading, and ensure fair access across users or processes. It applies to various system resources, including CPU, network bandwidth, API requests, and I/O operations.

Preemptive Scheduling

Definition:
Preemptive scheduling allows the operating system to interrupt a currently running process in order to start or resume another process. The decision to preempt a process is typically based on a predefined criterion, such as process priority or time slice expiration.

Key Characteristics:

- **Preemption:** The running process can be interrupted by the OS to switch to a more important process or if its time slice expires.
- **Responsiveness:** More responsive to real-time needs since higher-priority processes can quickly get CPU time.
- **Overhead:** Context switching introduces overhead, as the state of the process must be saved and the state of the new process must be loaded.
- **Example Algorithms:** Round Robin (RR), Shortest Remaining Time First (SRTF), Priority Scheduling (with preemption), Multilevel Queue Scheduling (with preemption).

Use Case:
Suitable for systems requiring real-time processing or when response time is critical, such as operating systems running interactive applications.

Non-Preemptive Scheduling

Definition:
In non-preemptive scheduling, once a process starts executing, it cannot be interrupted until it finishes or voluntarily yields control (e.g., when it needs I/O or waits for an event).

Key Characteristics:

- **No Preemption:** A running process will hold the CPU until it finishes or blocks itself by waiting for input/output.
- **Simplicity:** Easier to implement since there is no need for context switching during execution.
- **Less Overhead:** No context switching during the execution of a process, leading to reduced overhead.
- **Example Algorithms:** First-Come, First-Served (FCFS), Shortest Job Next (SJN), Priority Scheduling (without preemption).

Use Case:
Suitable for batch processing systems or applications where tasks have predictable execution times, and response time is not as critical.

Comparison:

- **Efficiency:** Preemptive scheduling is generally more efficient in terms of CPU utilization and handling multiple tasks, but it introduces more complexity and overhead. Non-preemptive scheduling is simpler but can lead to inefficiencies, such as longer wait times for processes with lower priorities.
- **Context Switching:** Preemptive scheduling requires more context switches, leading to additional overhead, while non-preemptive scheduling minimizes context switching.
- **Fairness:** Preemptive scheduling is often considered more fair, as it allows high-priority tasks to receive CPU time sooner, whereas non-preemptive scheduling can lead to longer wait times for lower-priority tasks.