

# 4. Computer Architectures

Services and Functions of the Operating System

# Summary

- Process Management
- Main Memory Management
- File System Management
- Peripheral Device Management
- Secondary Memory Management
- Memory Hierarchy
- Protection and Security Management
- HCIs and Application Management

# Process Management/1

A **process** refers to an **instance** of a **running application** or **program**. On a computer or device with a **multi-tasking Operating System (OS)**, numerous processes can run **simultaneously**, whether they belong to different applications or the same one. The OS manages these processes through various activities, including:

- **Creating** and **terminating** processes
- **Suspending** and **resuming** process execution
- **Synchronizing** and **facilitating** communication among processes
- **Managing** deadlocks

Processes can make **system calls** to access **OS services** for their own management, which include:

- **Executing** other processes (exec)
- **Replicating** a running process (fork)
- **Sending signals** between processes (wait/signal)
- **Terminating** a process (kill/terminate)

# Process Management/2

Using the **fork** system call, the OS can **create additional processes** based on its configuration, starting with the system's initial boot process.

This results in a **parent/child** tree hierarchy among the running processes on the computer. The **root process**, often referred to as **systemd** in modern Linux distributions or **init** in other UNIX-like systems, is responsible for system initialization and is the **ancestor of all other processes**.

Each process is identified by a unique **Process ID (PID)**, a **Parent Process ID (PPID)**, which references the PID of its parent process, and a **User ID (USERID)**, which indicates the user that is running the process.

```
tecmin@ubuntu:~$  
tecmin@ubuntu:~$ pstree -p  
systemd(1)─ModemManager(1156)─{ModemManager}(1197)  
└─{ModemManager}(1200)  
├─NetworkManager(990)─{NetworkManager}(1072)  
└─{NetworkManager}(1124)  
├─VGAuthService(916)  
├─accounts-daemon(979)─{accounts-daemon}(1033)  
└─{accounts-daemon}(1121)  
├─acpid(980)  
└─apache2(1382)─apache2(1383)─{apache2}(1393)  
└─{apache2}(1395)  
└─{apache2}(1397)  
└─{apache2}(1398)  
└─{apache2}(1400)
```

```

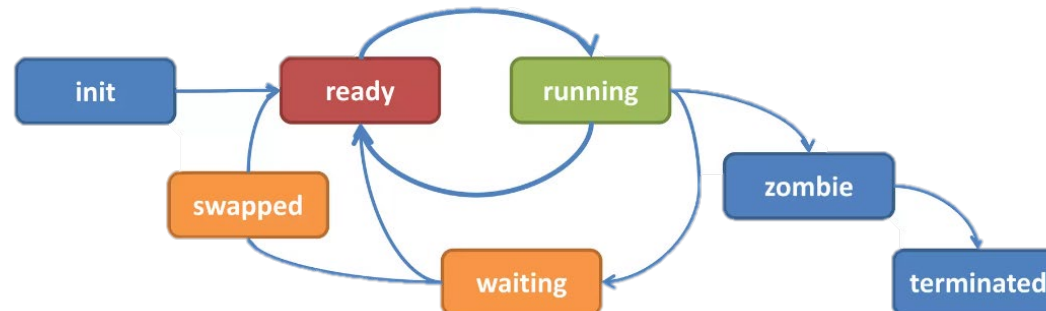
tecmint@ubuntu:~$ ps -ef
UID          PID     PPID  C  STIME TTY          TIME CMD
root          1        0  0  Jan20 ?        00:00:04 /sbin/init auto noprompt spl
root          2        0  0  Jan20 ?        00:00:00 [kthreadd]
root          3        2  0  Jan20 ?        00:00:00 [rcu_gp]
root          4        2  0  Jan20 ?        00:00:00 [rcu_par_gp]
root          5        2  0  Jan20 ?        00:00:00 [slub_flushwq]
root          6        2  0  Jan20 ?        00:00:00 [netns]
root          8        2  0  Jan20 ?        00:00:00 [kworker/0:0H-events_highpri]
root         10        2  0  Jan20 ?        00:00:00 [mm_percpu_wq]
root         11        2  0  Jan20 ?        00:00:00 [rcu_tasks_rude_]
root         12        2  0  Jan20 ?        00:00:00 [rcu_tasks_trace]
root         13        2  0  Jan20 ?        00:00:00 [ksoftirqd/0]
root         14        2  0  Jan20 ?        00:00:13 [rcu_sched]
root         15        2  0  Jan20 ?        00:00:00 [migration/0]

```

# Process Life Cycle

Each **process**, during its **life cycle**, can be in one of the following states:

1. **Init**: The initial load state of the process in memory. The program is set in an '**execution state**' inside the computer, the main process is created, and the required memory (RAM) is allocated.
2. **Ready**: The process is loaded into memory (RAM) in a '**ready to run**' state, waiting for the CPU allocation.
3. **Running**: The process **is being executed** by the CPU.
4. **Waiting**: The process is **suspended**, awaiting an event (e.g., feedback from a device).
5. **Swapped**: The process, while awaiting an event, has been placed inside the **virtual memory** (i.e., hard drive) and is pending recovery to primary memory to be executed.
6. **Zombie**: The process **has completed its execution** but remains in memory (i.e., it has a PID), waiting for its parent process to release it permanently.
7. **Terminated**: The process is **terminated**, and the OS deallocates the previously assigned memory (RAM).



# Main Memory Management

Main memory includes the **CPU registers**, the **CPU cache**, and the **Random Access Memory (RAM)**.

The OS has several responsibilities for managing the main memory:

- **Allocating and deallocating memory:** The OS assigns memory to processes as needed and releases it when it is no longer required.
- **Isolating memory segments:** In multi-tasking systems, the OS ensures that different processes have separate memory spaces to prevent conflicts and unauthorized access.
- **Address mapping:** The OS manages the mapping between logical memory addresses (used by processes) and physical memory addresses (actual locations in RAM).
- **Managing paging and virtual memory:** The OS uses paging to move portions of a program between primary memory (RAM) and secondary storage (swap space) to optimize the use of available memory and extend the effective amount of memory.

# Address Mapping

**Isolation:** Each process (like an application or a background service) running on a computer is given its own **virtual address space**. This space is essentially a set of memory addresses that the process can use to access memory. The key point is that this space is **private** to the process—no other process can see or access this space directly. This isolation protects processes from interfering with each other, which enhances **security** and **stability**.

**Consistency:** To the process itself, this virtual address space appears as a **continuous** and **consistent range of addresses** starting from zero upwards. It doesn't matter how the physical memory (RAM) is structured or how much physical memory is actually available.

**Mapping:** When a process requests access to a memory address within its virtual address space, it doesn't access the physical memory directly. Instead, the address it uses is a virtual address. The operating system, with the help of the **memory management unit (MMU)**, translates this virtual address into a physical memory address where the data is actually stored in RAM.

The **system calls** that the processes can invoke to obtain the **memory management services**, from the operating system, are the following:

- **malloc**, **calloc**, **realloc**, for the dynamic allocation of memory blocks;
- **free**, for the deallocation of the previously allocated memory blocks.

# Address Mapping - Advantages

- **Security:** By isolating each process's memory, the system prevents one process from accidentally or maliciously interfering with another process's data.
- **Flexibility:** Processes can be given more address space than the actual physical memory available on the system. For example, a process might be allowed to use 4 GB of addresses even if the computer only has 2 GB of physical RAM.
- **Efficiency:** The operating system can manage memory more effectively. It can allocate physical memory where needed and use techniques like paging (storing parts of the virtual memory on disk when RAM is full) to optimize the use of available physical memory.



# Virtual Memory and Paging

**Virtual memory** is a system where the **OS** uses both **physical RAM** and a **portion of the hard disk** called the **swap space** (or **paging file**) to simulate a much larger pool of memory. This system allows each process to have access to a private virtual address space, which is mapped to physical memory and disk as needed.

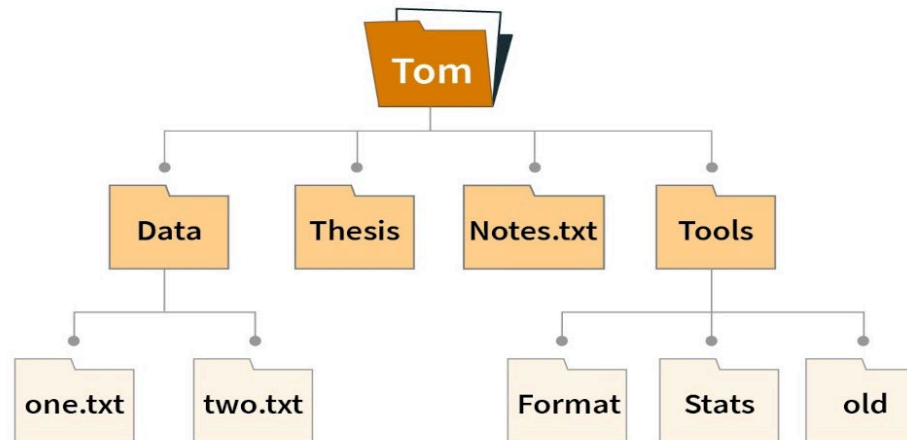
- **Paging:** The virtual memory is divided into blocks called pages. The corresponding blocks in physical memory are called **page frames**. Not all pages are loaded in physical memory at all times; some are kept on the disk in the swap space.
- **Page Table:** The OS maintains a page table for each process, which maps virtual pages to physical page frames. If a page is not in physical memory (a condition known as a "page fault"), the OS fetches it from the disk to a free page frame in physical memory, updating the table to reflect this.
- **Swap In/Swap Out:** When RAM fills up, the OS chooses less frequently used pages and moves them out to disk (swapping out) to make room for active pages that need to be loaded into RAM (swapping in).

# File System Management/1

The **file system** is an abstraction layer that the Operating System uses to **manage data** on **secondary storage** (e.g., hard drives and SSDs). This model is independent of the type and number of secondary storage devices.

The **basic element** of the file system is the **file**, which is a sequence of bytes stored on secondary storage. Unlike in-memory data structures, files persist beyond the process that created them and are often terminated with an **EOF (end of file)** marker, though this is more of a **logical concept** rather than a physical symbol stored in the file.

The **file system** provides an abstract framework for **organizing files** on secondary storage, typically in the form of a **hierarchical directory tree**, which includes **directories** and **subdirectories**.



# File System Management/2

The concept of a **directory** (or **folder**) is also an abstraction. Precisely, a directory is a file that contains **pointers to other files**, establishing a **parent-child** relationship within the file system.

**File names**, the set of characters allowed in file names, and the **meta-characters** used to indicate **file placement** within the file system are aspects defined by each specific file system model.

Examples:

“**C:\Tom\Data\one.txt**”: This is an absolute path that uniquely identifies a file located on the *C* drive in Microsoft Windows.

NOTE: that it is not case-sensitive and specifies the identifier of the physical drive where the file is stored.

“**~Tom/src/minimumSpanningTree.c**”: This path is typical in UNIX-like systems.

Note: It is case sensitive, uses the ‘*~username*’ convention to identify a user’s home directory, and the path is independent of the physical location of the file on a specific device.

# File System Management/3

Processes can invoke **system calls** to interact with the file system for various operations, including:

- **Creating** and **deleting** files
- **Opening** and **closing** files (**fopen** / **fclose**)
- **Reading** and **writing** files (**fget**/ **fread** / **fwrite**/ etc.)
- **Setting** file attributes (such as read-only, writable, executable)

The file system implements **protection mechanisms** to restrict access to files, ensuring only **authorized users** can interact with them. Additionally, it manages a **queue** of file access requests from processes.

In a **multi-user Operating System**, the file system tracks the ID of the **file owner** and defines **access rules** for other system users, ensuring appropriate access controls are enforced.

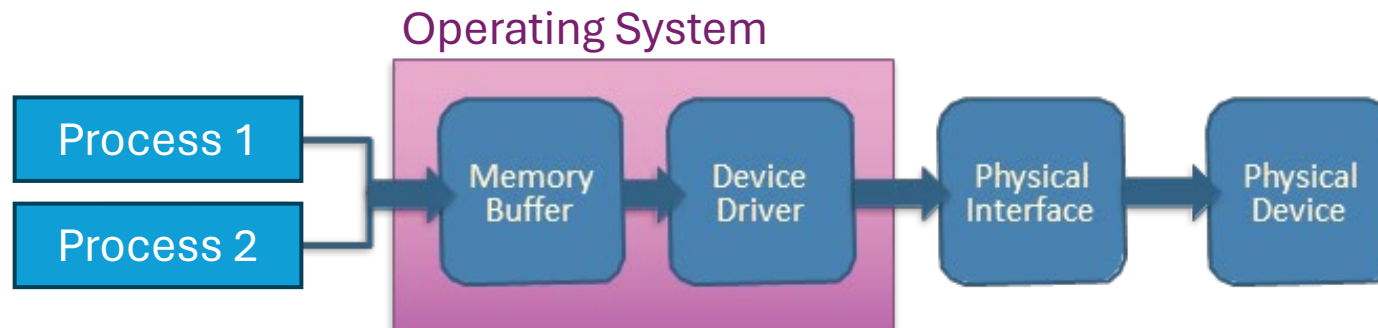
# Peripheral Device Management

The **OS** manages the **communication** with **peripheral units** and provides an **abstraction** (i.e., functions) that allows programs to utilize the **communication channel**.

Since **multiple programs** may **simultaneously** need **access** to a particular device (e.g., terminal output, keyboard input, or printer), the OS manages a **queue of requests** (**serialization**) to **avoid conflicts**.

The OS ensures efficient communication with specific peripheral devices. It uses a special type of memory called **buffer memory** to support the communication process. This buffer memory acts like a **parking area** where processes data directed to or coming from the device can be held until they are processed.

Interaction with peripheral devices is facilitated through a specific software module known as a **device driver**.



# Secondary Memory Management/1

**Secondary memory** consists of **persistent storage devices** that can retain recorded information even when the machine is turned off.

Due to physical and technological reasons, secondary memory has a much **higher access time** and **data transfer rate** compared to primary memory (which lacks mechanical components).

Typical secondary storage devices (or mass storage) include **magnetic hard drives**, **optical disks**, **solid-state drives (SSDs)**, **memory cards**, and more.



# Secondary Memory Management/2

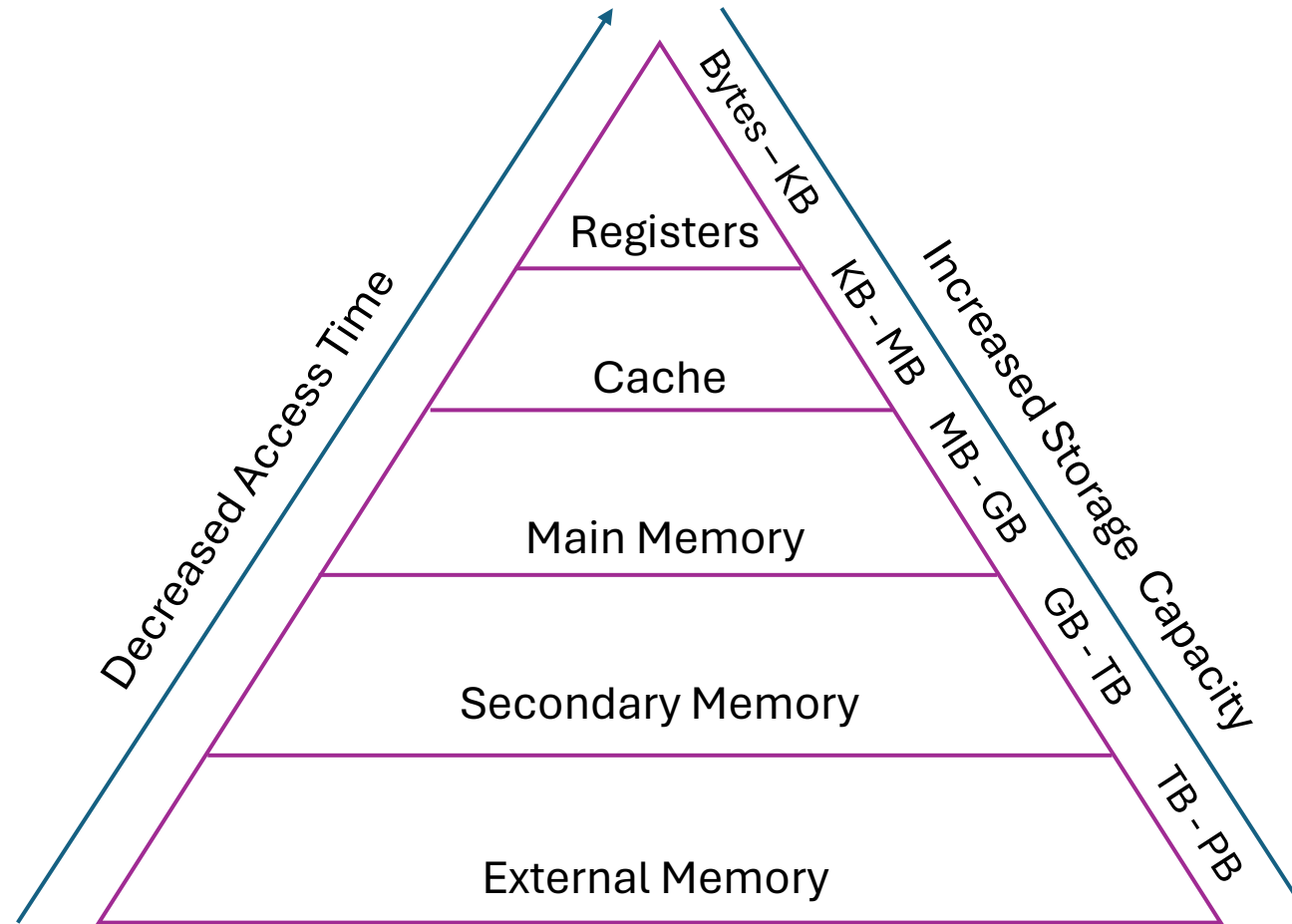
Typical **operations** performed by the **OS** on these devices include:

- **Allocating** and **deallocating** space for storing data (files)
- **Managing** free space within the mass memory unit
- **Optimizing**, **serializing**, and **scheduling** operations within the mass memory unit

File system management and secondary memory management are two distinct but closely related functions of the operating system.

The OS component that manages secondary memory makes the physical structure of the storage device transparent to programs, allowing the same system calls to be used for files stored on different devices.

# Memory Hierarchy





# Protection and Security Management

In a **multi-tasking** and **multi-user** system, the **OS** manages **resource protection** to ensure the **privacy** of **resources** and **users**.

The protection process is based on the following elements:

- **Authentication:** A procedure to verify the user's identity.
- **Authorization:** A procedure to determine a user's or process's rights to access a resource.



# Authorization

## **Linking Processes to User Permissions:**

- Each process in the system is associated with a specific user.
- Processes inherit the permissions of their associated user for accessing system resources.

## **High-Level Authorization for OS Processes:**

- OS processes are executed by users with the highest level of authorization, such root or administrator.

## **Resource Security Policies:**

- Resource security policies are based on rules that map resource access permissions to system users. For example, file access permissions dictate which users can read, write, or execute specific files.

## **Grouping Users for Simplified Authorization:**

To simplify authorization mapping, the OS allows users to be grouped into clusters or groups. Each member within a group inherits the group's authorizations.

# Authentication

A **multi-user OS** implements **login procedures** for system access.

The **login procedure** is crucial for two main reasons:

- It **authenticates** the user based on their credentials (i.e., username and password).
- It **verifies** whether the user is **authorized** to access the system.

The **login process** relies on a repository of **user credentials** and the definition of **user groups**, such as:

- The files **/etc/passwd** , **/etc/shadow** , and **/etc/group/** in UNIX-like systems.
- Other external systems of authentication and authorization.

## NOTE:


Username and password based authentication is the most common, although not the only or safest method.

# HCI and Applications Management

The **OS** provides a set of features to enable applications to create **user interaction** tools.

In general, a **User Interface (UI)** can be:

- **Alphanumeric Interface:** The operating system offers an abstract terminal model for presenting information on a screen that displays alphabetic and numeric characters. It also captures input through a keyboard.
- **Graphical User Interface (GUI):** The operating system provides features for programs to build a GUI using elements such as windows and icons. Input can be done using a mouse or touch-screen, in addition to the keyboard.



```
2. ~/nexmo/ (zsh)

nexmo/
> npm install -g nexmo-cli
/usr/local/Cellar/nvm/0.22.0/versions/v6.2.2/bin/nexmo -> /usr/local/Cellar/nvm/0.22.0/versions/v6.2.2/lib/node_modules/nexmo-cli/lib/bin.js
/usr/local/Cellar/nvm/0.22.0/versions/v6.2.2/lib
├─ nexmo-cli@0.0.16
├─ colors@1.1.2
├─ commander@2.9.0
├─ graceful-readlink@1.0.1
├─ ini@1.3.4
└─ nexmo@1.0.0-beta-4

nexmo/
> |
```

