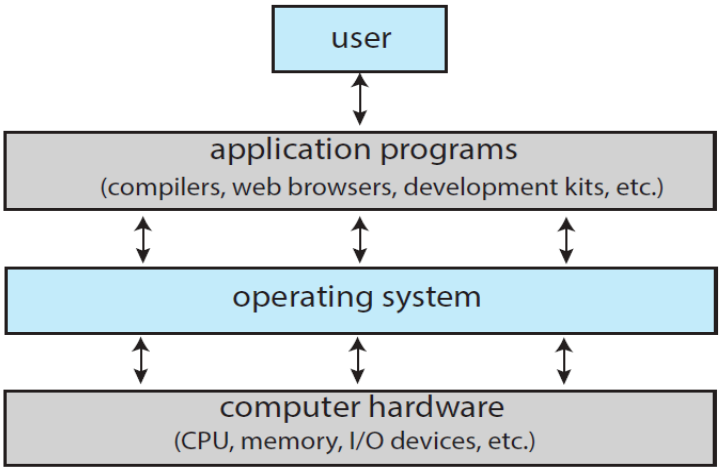
Ch-1

>>User view fig.



How people use a computer depends on the interface. Many use laptops or desktops with a monitor, keyboard, and mouse—made for one person to use fully. The operating system focuses on ease of use, with some attention to speed and security, but not on sharing resources.

>>Relation between device,device driver and device controller:

A **device** is a real piece of hardware, like a printer, keyboard, or hard drive. To use the device, the system needs help from two important parts: the **device controller** and the **device driver**.The **device controller** is a small hardware part that controls how the device works. It sends and receives data between the device and the computer. It also has a small memory area called a **buffer** to hold data for a short time.The **device driver** is software. It knows how to talk to the device controller. Even if devices are different, the driver gives the operating system a standard way to control them.This teamwork makes sure everything runs smoothly. Without drivers and controllers, the OS wouldn't know how to use hardware devices properly.

>>What is an interrupt?

An interrupt is a signal sent from a device (like a keyboard, printer, or hard disk) to the CPU to let it know that the device has finished its task. For example, after reading data or completing a write, the device sends an interrupt to inform the CPU.

Importance of interrupt in OS:

1. Interrupts help the CPU know when a device has finished its task.
2. They allow the CPU to do other work instead of waiting for I/O to finish.
3. Interrupts make the system faster and more efficient by handling tasks only when needed.
4. They help in better coordination between hardware and software.

>>Storage Structure

Storage structure refers to the different types of memory used in a computer to store programs and data. It includes both fast, temporary memory and slower, permanent memory.

Volatile Storage

- Example: Main memory (RAM).
- Fast but loses data when power is off.
- Used to run programs while the computer is on.

Non-Volatile Storage

- Example: EEPROM, firmware.
- Keeps data even when power is off.
- Stores important programs like the bootstrap program.

>>Difference between Multiprogramming and Multi tasking

- **Multiprogramming:**
    - Runs multiple programs by keeping them in memory.
    - CPU switches to another program only when the current one waits (like for I/O).
    - Switching happens less often to keep CPU busy.
- **Multitasking:**
    - A faster version of multiprogramming.
    - CPU switches between programs very quickly.
    - Gives users a fast response and feels like tasks run at the same time.

>>Mode Switching / Dual Mode

- Computers have two modes: **User mode** and **Kernel mode**.
- **User mode** is for normal programs and apps. They can't do everything to keep the computer safe.
- **Kernel mode** is for the operating system. It can do anything, like control hardware.
- **Mode switching** means changing between user mode and kernel mode when needed.
- This keeps the computer safe by stopping apps from doing dangerous things.
- Dual mode ensures system security and stability by controlling access to important resources.

>>How virtualization works

Virtualization is a technology that lets one physical computer act like many computers. It does this by dividing the computer's hardware into several **virtual machines (VMs)**. Each virtual machine works like a real computer and can have its own **operating system**, like Windows or Linux.These virtual machines can run **at the same time**, and users can easily **switch between them**. It feels like using many computers, but it's really just one.Virtualization is different from **emulation**. Emulation copies different types of hardware and is usually **slower**. Virtualization is **faster** because it runs directly on the computer's real hardware (same CPU).This technology is useful because it lets **many programs or users share one computer**, even if it was made for only one person.

Ch-2

>>Why Applications Are Operating-System Specific?

Applications are usually made for one operating system because each OS uses different system calls, file formats, and ways of working with hardware. Programs made for one OS won't run on another without changes. Also, different CPUs use different instructions, which makes it harder. Some apps work on many systems using interpreters (like Python) or virtual machines (like Java), but these methods can be slower. Even if the same app is available on different OSs, developers often have to write and test separate versions.So, because of these differences, most applications are OS-specific.

>>What is a System Call?

A **system call** is a way for programs to request services from the operating system. It acts as a bridge between the user program and the OS.

>>Types of System Calls

System calls are mainly divided into **six types**:

1. **Process control** – For creating, ending, and managing processes.
2. **File management** – For creating, deleting, reading, or writing files.
3. **Device management** – To control hardware devices.

4. **Information maintenance** – For getting or setting system data like time, date, memory, etc.
5. **Communication** – For sending messages or data between processes or systems.
6. **Protection** – For controlling access and permissions to system resources.

>>Operating System Services

9 types of OS services:

1.Program Execution – Runs programs and ends them when done.

2.I/O Operations – Manages input and output devices like keyboard, mouse, printer.

3.File System Management – Lets you create, open, save, and delete files.

4.Communication – Helps programs share information with each other or over a network.

5.Error Detection – Finds and fixes errors in hardware or programs.

6.Resource Allocation – Shares CPU, memory, and devices among programs.

7.Logging – Keeps records of system activities and events.

8.Protection – Keeps system safe from unauthorized access.

9.User Interface – Lets users interact with the computer.

Ch-3

>>Process Control Block (PCB)

A Process Control Block (PCB) is a data structure used by the operating system to store information about a process. It acts as the "identity card" for a process.

How PCB is Used:

* The OS creates a PCB when a process starts.

* During context switching, the OS saves the current process's state in its PCB and loads the state of the next process from its PCB.

* Used for scheduling,resource allocation, and process tracking.

>>Process States

As a process runs, it changes its state based on what it's doing. The main process states are:

- New – The process is being created.
- Running – The process is currently using the CPU.
- Waiting – It is waiting for something (like input/output to finish).
- Ready – It is ready to run but waiting for CPU time.
- Terminated – It has finished running.

Only one process can run on a CPU core at a time. Others must wait in the Ready or Waiting state.

>>Variable

A variable in an operating system is a named storage location used to hold data while a program or the system is running.

Types of Variables in OS:

1. **User-level Variables:**
    - Used inside programs.
    - Temporary and specific to the program.

2. **System-level Variables:**
    - Used by the OS or shell (like Linux or Windows).
    - Examples: PATH, HOME, USERNAME.

3. **Environment Variables:**
    - Special variables that affect how processes run.
    - Example: PATH helps the OS find where programs are stored.

>>Interprocess Communication (IPC)

Interprocess Communication (IPC) is the way **two or more processes** running at the same time **communicate and share data** with each other.

Two Types of IPC:

1. **Message Passing:**
    - Processes send and receive messages.
    - No shared memory.
    - Simple and good for processes on different systems.
2. **Shared Memory:**
    - Processes use the same memory space to share data.
    - Faster but needs control to avoid conflicts.

>>How Multiprocessor Architecture Works

Multiprocessor architecture uses **two or more CPUs (processors)** in a single computer system. These CPUs share the same memory and work **together to perform tasks faster**.Each processor can work on **different parts of a task** or even on **different tasks at the same time**. The operating system controls how work is divided and makes sure the processors don't interfere with each other.

This system is useful because it makes the computer **faster**, helps it **run many programs at once**, and if one CPU fails, the others can still work.

>>Cooperating Processes

Cooperating processes are processes that **work together** and can **share data or information** to achieve a common goal.

Example:

Think of a word processor and a spell checker running at the same time. The spell checker process needs to read the document the word processor is working on and highlight mistakes. They cooperate by sharing data.

>>How Cooperating Process Works:

Processes share information by using ways like shared memory or sending messages. They work together so one process can use data made or changed by another process. This teamwork helps the system run faster and better.

>>Problems of Cooperating Process :

- **Data inconsistency:**If they don't share carefully, data can get mixed up.
- **Deadlock:**Sometimes, they can get stuck waiting for each other forever.
- **Security:**Sharing data can cause security risks if not protected

>>Fork codes(how it works and output)

[from pdf exercises]

Code-1

```
#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int value = 5;
```

```
int main(){

pid t pid;

pid = fork();

if (pid == 0) { /* child process */

value += 15;

return 0;

}

else if (pid > 0) { /* parent process */

wait(NULL);

printf("PARENT: value = %d",value); /* LINE A */

return 0;

}

}
```

What output will be at Line A?

Ans:

- fork() creates a new child process. Both parent and child have separate copies of the variables.
- Initially, `value = 5`.
- In the child process: `value` is increased by 15, so `value = 20` inside the child only.
- In the parent process: `value` remains `5` because the parent and child have separate memory spaces.
- The parent waits for the child to finish with `wait(NULL)`.
- Then the parent prints `value`.

Final output at LINE A: PARENT: value = 5

Code-2

```
#include <stdio.h>

#include <unistd.h>

int main(){

/* fork a child process */

fork();

/* fork another child process */

fork();

/* and fork another */

fork();

return 0;

}
```

How many processes are created?

Ans:

Step-by-step:

1. Initially, there is **1 process** (the original).
2. After the first `fork()`:The 1 process splits into **2 processes**.
3. After the second `fork()`:Both of the 2 processes fork, so **2 × 2 = 4 processes** now.
4. After the third `fork()`:All 4 processes fork, so **4 × 2 = 8 processes** now.

Total processes created: 8

Code-3

```
#include <stdio.h>
```

```
#include <unistd.h>

int main(){

int i;

for (i = 0; i < 4; i++)

fork();

return 0;

}
```

How many processes are created?

Ans:

- The loop runs 4 times.

Step-by-step process count:

- After 1st `fork()`: 1 × 2 = 2 processes.
- After 2nd `fork()`: 2 × 2 = 4 processes.
- After 3rd `fork()`: 4 × 2 = 8 processes.
- After 4th `fork()`: 8 × 2 = 16 processes.

Total processes created: 16

[Extra from chatgpt]

Code-1

```
#include <stdio.h>

#include <unistd.h>

int main() {

    fork();

    printf("Hello\n");
```

```
    return 0;
}
```

How many times will "Hello" be printed? What will be the output?

Ans:

`fork()` creates 1 child process.

Now 2 processes (parent + child) run `printf("Hello\n");`.

Output: Hello

Hello

Code-2

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid == 0)
        printf("Child process\n");
    else
        printf("Parent process\n");
    return 0;
}
```

What will be printed and how many times?

Ans:

- fork() creates a child process.
- In the **child**, pid == 0, so it prints: Child process.

- In the **parent**, pid > 0, so it prints: Parent process

Output:

Parent process

Child process

Code-3

```
#include <stdio.h>
#include <unistd.h>
int main() {
    fork();
    fork();
    printf("Hi\n");
    return 0;
}
```

Q: How many times will "Hi" be printed? How many processes are created?

Ans:

- First fork() → now 2 processes
- Second fork() → both 2 processes fork again → now 4 processes

Each of the 4 processes prints Hi.

Output:

Hi

Hi

Hi

Hi