📝 Operating System Notes (Based on Your Syllabus)

Session 1: Introduction to OS

1. What is an Operating System (OS)?

- **Definition:** An OS is a system software that acts as an interface between **user and hardware**.
- **It controls hardware resources** and provides an environment for applications to run.

2. Difference between OS and Application Software

Feature	Operating System	Application Software
Purpose	Manages hardware & resources	Performs specific tasks for users
Examples	Windows, Linux, macOS	MS Word, VLC Player, Chrome
Dependency	Hardware-dependent	Runs on top of OS
Function	Resource allocation, scheduling	Task-specific like editing, browsing

3. Why OS is Hardware Dependent?

- Designed for specific CPU architecture & device drivers.
- Example: Windows designed for x86/x64, iOS for Apple devices.

4. Components of OS

1. **Kernel** – Core, manages CPU, memory, I/O.
2. **Shell** – Interface between user & kernel.

3. **File System** – Organizes and stores data.
4. **Device Drivers** – Help OS communicate with hardware.
5. **System Utilities** – Provide tools (disk mgmt, task manager).

5. Basic Computer Organization for OS

- **CPU** (executes instructions).
- **Memory** (RAM, Cache).
- **I/O Devices** (keyboard, monitor, disk).
- **System Bus** (data transfer).

6. Types of OS

Type	Example	Features
Desktop OS	Windows, Linux	GUI, multitasking
Mobile OS	Android, iOS	Touch-based, lightweight
Server OS	Ubuntu Server, Windows Server	Handles multiple clients, networking
Embedded OS	RTOS, VxWorks	Small footprint, runs on chips
Real Time OS (RTOS)	QNX, FreeRTOS	Predictable response, used in robotics

7. Functions of OS

- Process management
- Memory management
- File system mgmt
- I/O mgmt
- Security & protection
- User interface

8. User & Kernel Space

- **User Space:** Applications run here.
- **Kernel Space:** OS core runs here.
- **Modes:**
  - **User Mode** – Limited access, safe.
  - **Kernel Mode** – Full hardware access.

9. Interrupts & System Calls

- **Interrupts:** Signals that stop CPU to handle urgent tasks (keyboard press, I/O).
- **System Calls:** Interface for user programs to request kernel services (e.g., `read()`, `write()`).

✅ **Quick Revision – Session 1**

- OS = Interface between user & hardware.
- Kernel vs Shell.
- OS Types: Desktop, Mobile, Server, Embedded, RTOS.
- Functions: Process, Memory, File, I/O mgmt.
- User space ↔ Kernel space, Interrupts, System calls.

Session 2: Introduction to Linux

1. Working Basics of File System

- **Linux File System is hierarchical (tree-like).**
- **Root directory `/`** is at the top.

- Common directories:
    - `/home` → user files
    - `/bin` → essential binaries
    - `/etc` → configuration files
    - `/var` → log files
    - `/dev` → device files
    - `/tmp` → temporary files

2. Basic Commands

Command	Usage
`ls`	List files & directories
`pwd`	Print current directory
`cd`	Change directory
`touch file.txt`	Create empty file
`cat file.txt`	Display file content
`cp a.txt b.txt`	Copy file
`mv a.txt b.txt`	Rename/Move file
`rm file.txt`	Remove file
`mkdir new`	Create directory
`rmdir dir`	Remove empty directory

Operators:

- `>` → Redirect output to file (overwrite).

- `>>` → Append output to file.
- `<` → Input redirection.
- `|` (pipe) → Output of one command → input of another.

Example:

ls -l | grep ".txt"

(Find only `.txt` files).

---

3. File Permissions

- Every file has **3 permissions**:
  - **r (read), w (write), x (execute)**
- For **3 categories of users**:
  - **Owner, Group, Others**

Example:

-rwxr-xr-- 1 user group file.txt

(Owner → rwx, Group → r-x, Others → r--)

Change Permissions:

- `chmod 755 file` → rwxr-xr-x
- `chmod u+x file` → Add execute permission to user
- `chown user:group file` → Change ownership

Access Control List (ACL):

- More fine-grained permissions.

- Example: `setfacl -m u:john:rwx file.txt`

---

4. Network Commands

Command	Usage
`ping google.com`	Check connectivity
`ftp server`	File transfer
`sftp user@host`	Secure file transfer
`ssh user@host`	Remote login
`telnet host`	Old remote login (less secure)
`finger user`	Info about user

---

5. System Variables

- **PS1** → Primary prompt string (`$` or `#`).
  - Example: `PS1="MyLinux> "` → Changes shell prompt.
- **PS2** → Secondary prompt (for multi-line input, default `>`).
- **echo $HOME`** → Displays home directory.

---

6. Shell Programming

What is a Shell?

- Interface between **user** and **kernel**.
- Types of shells:

- `sh` → Bourne Shell
- `bash` → Bourne Again Shell (most used)
- `csh` → C Shell
- `ksh` → Korn Shell

Shell Variables

- `name="Prasad"`
- `echo $name`

Wildcard Symbols

- `*` → Matches zero or more characters.
- `?` → Matches single character.
- `[abc]` → Matches any one character from set.

Shell Meta Characters

- `;` → Separate multiple commands.
- `&&` → Run next only if first succeeds.
- `||` → Run next only if first fails.

Command Line Arguments

- `$0` → Script name
- `$1, $2`… → Arguments
- `$#` → Number of arguments
- `$@` → All arguments

Input/Output

- `read var` → Take input
- `echo $var` → Print value

✅ **Quick Revision – Session 2**

- Linux file system = tree-like, root `/`.
- Commands: `ls, cd, pwd, touch, cat, cp, mv, rm`.
- Operators: `>`, `>>`, `<`, `|`.
- File permissions: `chmod, chown, ACL`.
- Network commands: `ssh, ftp, sftp, telnet`.
- Shell types: `bash, sh, csh, ksh`.
- Variables: `$0, $1, $#, $@`.
- Wildcards: `* ? [ ]`.

📑 Differences – Session 1 & 2

1. OS vs Application Software

Feature	Operating System	Application Software
Purpose	Manages hardware, provides platform	Solves specific user tasks
Dependency	Hardware dependent	Runs on OS
Examples	Windows, Linux, Android	Word, VLC, Chrome
Layer	Directly above hardware	Above OS

---

2. User Space vs Kernel Space

Feature	User Space	Kernel Space
Runs	Applications & utilities	Core OS (kernel)
Access	Limited (no direct hardware access)	Full hardware access
Mode	User mode	Kernel mode
Example	Browser, Editor	Process scheduler, Memory manager

---

3. Process vs Thread (from Session 4 but relevant early)

Feature	Process	Thread
Definition	Independent program in execution	Lightweight unit of a process
Memory	Has its own memory	Shares memory of parent process
Overhead	More overhead	Less overhead
Example	Running `chrome.exe`	Multiple tabs inside Chrome

---

4. Linux vs Windows OS

Feature	Linux	Windows
Source	Open-source	Proprietary
Security	More secure (permissions, sudo)	More malware-prone
Cost	Free	Paid
Usage	Servers, Developers	Desktops, Enterprises
Customization	Highly customizable	Limited

---

5. Shell vs Kernel

Feature	Shell	Kernel

Role	Interface between user & kernel	Core of OS
Access	User interaction (commands)	Hardware management
Example	bash, sh, csh	Linux kernel, Windows NT kernel

---

6. Absolute vs Relative Path (Linux FS)

Path Type	Example	Meaning
Absolute	`/home/prasad/file.txt`	Starts from root `/`
Relative	`./file.txt` or `../file.txt`	Relative to current directory

---

7. File Permissions (User vs Group vs Others)

Category	Symbol	Who?
User	u	File owner
Group	g	Users in same group
Others	o	All other users

📝 Session 3: Shell Programming

---

1. Decision Making in Shell

if-else

if [condition]
then
  commands

```
else
  commands
fi
```

Example:

```
num=5
if [ $num -gt 0 ]
then
  echo "Positive number"
else
  echo "Negative number"
fi
```

Nested if-else

```
if [ condition1 ]
then
  ...
elif [ condition2 ]
then
  ...
else
  ...
fi
```

---

test Command

- Used to evaluate conditions.

Example:

```
 if test -f "file.txt"
then
  echo "File exists"
fi
```

-

---

case Statement

```
case $variable in
```

```
  pattern1) commands ;;
  pattern2) commands ;;
  *) default ;;
esac
```

Example:

```
echo "Enter a letter:"
read ch
case $ch in
  [a-z]) echo "Lowercase" ;;
  [A-Z]) echo "Uppercase" ;;
  *) echo "Other" ;;
esac
```

---

2. Loops in Shell

while Loop

```
while [ condition ]
do
  commands
done
```

Example:

```
i=1
while [ $i -le 5 ]
do
  echo "Count: $i"
  i=$((i+1))
done
```

---

until Loop

- Opposite of `while`. Runs until condition becomes **true**.

```
i=1
until [ $i -gt 5 ]
do
  echo "Value: $i"
```

```
  i=$((i+1))
done
```

---

for Loop

```
for var in 1 2 3 4 5
do
  echo $var
done
```

Example with files:

```
for file in *.txt
do
  echo "File: $file"
done
```

---

3. Regular Expressions (Regex)

Used with tools like `grep`, `sed`, `awk`.

Examples:

- `^word` → starts with word
- `word$` → ends with word
- `.` → any single char
- `*` → zero or more chars
- `[0-9]` → digits
- `[a-z]` → lowercase letters

Example:

```
grep "^[A-Z]" file.txt   # Lines starting with capital letter
```

---

4. Arithmetic Expressions

Two ways:

Using `$(( ))`

```
a=10
b=20
sum=$((a+b))
echo "Sum = $sum"
```

Using `expr`

```
expr 5 + 3
expr 10 \* 2
```

---

5. More Examples

Example 1: Factorial

```
echo "Enter a number:"
read n
fact=1
for (( i=1; i<=n; i++ ))
do
  fact=$((fact*i))
done
echo "Factorial = $fact"
```

Example 2: Even / Odd

```
echo "Enter a number:"
read n
if [ $((n%2)) -eq 0 ]
then
  echo "Even"
else
  echo "Odd"
fi
```

✅ Quick Revision – Session 3

- **if, if-else, nested if, case** → decision making.
- **while, until, for** → looping constructs.
- **Regex** → powerful search (e.g., `^`, `$`, `*`, `[ ]`).
- **Arithmetic** → `$((a+b))`, `expr`.
- Shell scripts use `read`, `echo`, arguments `$1, $2...`.

📝 Sessions 4 & 5: Processes

---

1. What is a Process?

- A **program in execution** is called a process.
- Contains: program code, data, stack, registers, program counter, etc.

---

2. Preemptive vs Non-Preemptive Processes

Feature	Preemptive	Non-Preemptive
Definition	CPU can be taken away from process	Once CPU allocated, process keeps it until completion
Example	Round Robin, Priority (preemptive)	FCFS, SJF (non-preemptive)
Response	Better response time	Poor response time
Complexity	Complex	Simple

---

3. Process vs Thread

Feature	Process	Thread

Definition	Independent program in execution	Lightweight unit of a process
Memory	Own memory space	Shares memory of process
Overhead	High (context switching)	Low
Example	Running Chrome app	Multiple tabs inside Chrome

---

4. Process Management & Life Cycle

Process States:

1. **New** → Process created
2. **Ready** → Waiting for CPU
3. **Running** → Executing on CPU
4. **Waiting/Blocked** → Waiting for I/O or event
5. **Terminated** → Finished execution

👉 Diagram:

```
New → Ready → Running → Terminated
       ↑          ↓
    Waiting ← I/O request
```

---

5. Schedulers

- **Long-term Scheduler** → controls admission (new → ready).
- **Medium-term Scheduler** → suspends/resumes processes.
- **Short-term Scheduler** → decides which process gets CPU next (ready → running).

---

6. CPU Scheduling Algorithms

(i) First Come First Serve (FCFS)

- Non-preemptive.
- Processes served in order of arrival.
- Like a queue.

👉 Example:
Processes: P1=5, P2=3, P3=6 (burst times).

Gantt Chart: | P1 | P2 | P3 |
Waiting Time = (0 + 5 + 8)/3 = 13/3 = 4.33
Turnaround Time = (5 + 8 + 14)/3 = 27/3 = 9

(ii) Shortest Job First (SJF)

- Process with smallest burst time first.
- Can be **preemptive** (SRTF) or non-preemptive.

👉 Example: Burst times = P1=8, P2=7, P3=2, P4=4

- Order (SJF) = P3 → P4 → P2 → P1

(iii) Priority Scheduling

- Each process has priority.
- Higher priority executed first.
- Can be preemptive or non-preemptive.

(iv) Round Robin (RR)

- Preemptive.
- Time slice (quantum) is fixed.

- Processes get CPU for fixed time in cyclic order.

👉 Example:
Burst times P1=10, P2=5, P3=8, Quantum=4

- Cycle 1: P1(4), P2(4), P3(4)
- Cycle 2: P1(6), P2(1), P3(4)
- Cycle 3: P1(2), P3(0)…

(v) Multilevel Queue Scheduling

- Ready queue divided into multiple queues (foreground, background).
- Each has own scheduling.

---

7. Belady's Anomaly

- In some page replacement algorithms (like FIFO), **increasing number of frames increases page faults** instead of decreasing.

---

8. Process Creation in Unix/Linux

- `fork()` → Creates new child process.
- `exec()` → Replaces process with new program.
- `waitpid()` → Parent waits for child to finish.

👉 Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
  if(fork() == 0)
```

```
    printf("Child Process\n");
  else
    printf("Parent Process\n");
  return 0;
}
```

---

9. Parent, Child, Orphan & Zombie

- **Parent process** → Creates another process.
- **Child process** → Created by parent.
- **Orphan** → Parent finishes, child still running.
- **Zombie** → Child finished, parent has not read exit status.

✅ Quick Revision – Session 4 & 5

- Process = program in execution.
- Preemptive vs Non-preemptive.
- Process vs Thread.
- States: New, Ready, Running, Waiting, Terminated.
- Scheduling: FCFS, SJF, Priority, RR, Multilevel Queue.
- Belady's Anomaly = more frames → more page faults.
- System calls: fork(), exec(), waitpid().
- Orphan vs Zombie.

📝 Sessions 6 & 7: Memory Management

---

1. Types of Memories

- **Registers** – Fastest, inside CPU.
- **Cache** – Very fast, stores frequently used data.
- **Main Memory (RAM)** – Volatile, holds processes.
- **Secondary Storage (HDD/SSD)** – Permanent storage.

👉 OS manages memory because multiple processes compete for it.

---

2. Need of Memory Management

- Efficient utilization of RAM.
- Protection (no process can access others' memory).
- Fair allocation among processes.
- To support multitasking.

---

3. Memory Allocation

(a) Contiguous Allocation

- Each process gets a single contiguous block of memory.
- Simple but causes fragmentation.

(b) Dynamic Partitioning

- Memory divided dynamically depending on process size.

---

4. Allocation Strategies

Strategy	Explanation	Example
First Fit	Allocate first block large enough	Fast but may waste space
Best Fit	Allocate smallest block that fits	Less waste but slower
Worst Fit	Allocate largest block	Leaves large leftover space

---

5. Compaction

- Technique to remove **external fragmentation** by shifting processes to one side.

---

6. Fragmentation

Type	Definition	Example
Internal	Wasted space inside allocated block	Process needs 18KB, block is 20KB → 2KB wasted
External	Enough total free memory but not contiguous	3 free blocks of 10KB but process needs 25KB

---

7. Segmentation

- Memory divided into **logical segments** (Code, Data, Stack).
- Each segment has **base + limit** in a **Segment Table**.

👉 Example:

Segment Table
Code → Base 1000, Limit 400
Data → Base 1400, Limit 600

- Address = Segment no. + Offset.
- Hardware: **Segment Table Base Register (STBR)**, **Segment Table Length Register (STLR)**.

---

8. Paging

- Memory divided into **fixed-size frames**.
- Process divided into **pages** (same size as frames).
- **Page Table** maps pages → frames.

👉 Example:
Process of 10KB, Page size = 1KB → 10 pages.
Pages mapped to available frames in RAM.

Translation Lookaside Buffer (TLB)

- Special cache for page table entries.
- Speeds up address translation.

👉 Effective Access Time (EAT):

EAT = (Hit Ratio × (TLB Access + Memory Access))
+ (Miss Ratio × (TLB Access + 2 × Memory Access))

Dirty Bit

- Each page has **dirty bit**.
- If 1 → Page modified, must be written back to disk before replacement.
- If 0 → Page unmodified, no need to write.

Shared Pages & Reentrant Code

- **Shared Pages** → Common code (e.g., text editors) shared by processes.

- **Reentrant Code** → Code that can be executed by multiple processes without changing itself.

---

9. Throttling

- If processes demand more memory than available → OS **limits execution** (throttling) to prevent overload.

---

10. I/O Management

- OS manages communication between CPU and I/O devices.
- Uses **Interrupts**, **Buffers**, **Spooling**.

✅ **Quick Revision – Sessions 6 & 7**

- **Internal vs External fragmentation**.
- Allocation: **First Fit, Best Fit, Worst Fit**.
- **Compaction** = remove external fragmentation.
- **Segmentation** = logical division (Code, Data, Stack).
- **Paging** = fixed-size frames + pages, Page Table, TLB.
- **Dirty Bit** = Modified page check.
- **Shared pages & reentrant code**.
- **Throttling** = limiting processes due to memory shortage.

📝 Session 8: Virtual Memory

---

1. What is Virtual Memory?

- Virtual Memory = technique that gives an **illusion of a very large memory** by combining **RAM + Disk**.
- Allows programs larger than physical memory to execute.
- Implemented using **paging + demand paging**.

👉 Example: If RAM = 4GB but program needs 8GB → OS swaps part of program to disk and loads only required pages into RAM.

---

2. Demand Paging

- Instead of loading the entire process into memory, **only required pages** are loaded.
- **Valid-Invalid Bit** in Page Table:
    - `Valid` → page in memory
    - `Invalid` → page on disk

👉 Advantages: Less memory usage, faster program start.
👉 Disadvantage: Page faults.

---

3. Page Fault

- Occurs when process tries to access a page that is **not in RAM**.
- Steps:
    1. CPU generates page fault interrupt.
    2. OS checks if page is valid.
    3. If valid → load page from disk into RAM.

4. Update page table.
5. Restart instruction.

---

4. Page Replacement Algorithms

When a new page is needed but all frames are full → OS replaces one existing page.

(i) FIFO (First In First Out)

- Oldest page replaced first.
- **Simple but may cause Belady's Anomaly**.

👉 Example: Reference string = 7, 0, 1, 2, 0, 3, 0, 4 (3 frames)
Page Faults = 9

(ii) Optimal Page Replacement

- Replace the page that will not be used for the longest time.
- **Best performance but not practical** (future knowledge needed).

(iii) LRU (Least Recently Used)

- Replace page that was least recently used.
- Uses **stack or counters**.
- More practical, commonly used.

(iv) LFU (Least Frequently Used)

- Replace the page used least number of times.

(v) MFU (Most Frequently Used)

- Replace page with **highest usage count**.

5. Example Problem (FIFO vs LRU)

Reference string: **1, 3, 0, 3, 5, 6, 3** (Frames = 3)

FIFO:

Step: 1 → 3 → 0 → 3 → 5 → 6 → 3
Frames:
1 | 1 3 | 1 3 0 | 1 3 0 | 5 3 0 | 6 3 0 | 6 3 0
Page Faults = 5

LRU:

Step: 1 → 3 → 0 → 3 → 5 → 6 → 3
Frames:
1 | 1 3 | 1 3 0 | 1 3 0 | 5 3 0 | 5 6 0 | 5 6 3
Page Faults = 5

👉 Sometimes FIFO = LRU, sometimes LRU performs better.

✅ **Quick Revision – Session 8**

- Virtual Memory = illusion of large memory.
- Demand Paging = load pages only when needed.
- Page Fault = page not in RAM.
- Replacement Algorithms: **FIFO, Optimal, LRU, LFU, MFU**.
- **Belady's Anomaly** occurs in FIFO (more frames → more page faults).

📝 Session 9: Deadlock, Synchronization & Concurrency

---

1. What is Deadlock?

Deadlock occurs when **two or more processes wait for each other forever** → no progress.

4 Necessary Conditions (Coffman's):

1. **Mutual Exclusion** – only one process can use a resource at a time.
2. **Hold & Wait** – process holds one resource while waiting for another.
3. **No Preemption** – resources cannot be forcibly taken away.
4. **Circular Wait** – processes form a circular chain waiting for each other.

👉 If all 4 are true → Deadlock.

---

2. Deadlock Handling

- **Prevention** → Break one of the 4 conditions.
- **Avoidance** → Use algorithms like **Banker's Algorithm**.
- **Detection & Recovery** → Detect deadlock and kill/restart processes.
- **Ignore** → Used in many OS (like UNIX), assuming deadlocks are rare.

---

3. Deadlock vs Starvation

- **Deadlock** → Processes wait forever (cyclic dependency).
- **Starvation** → Process waits indefinitely due to low priority.
  👉 Deadlock = group problem | Starvation = individual process problem.

---

4. Semaphores

- Synchronization tool invented by **Dijkstra**.
- Used to manage access to shared resources.

Types:

1. **Binary Semaphore (Mutex)** → only 0/1, like a lock.
2. **Counting Semaphore** → integer value, allows multiple instances.

👉 Operations:

- `wait(S)` or `P(S)` → Decrement, block if < 0.
- `signal(S)` or `V(S)` → Increment, wake up a waiting process.

---

5. Mutex vs Semaphore

- **Mutex** = Lock (one owner at a time).
- **Semaphore** = Counter (multiple processes allowed).
  👉 Mutex is a special case of Semaphore (binary).

---

6. Producer-Consumer Problem (Bounded Buffer Problem)

- Example of **process synchronization** using semaphores.

Semaphores:

- `mutex = 1` → ensures mutual exclusion.

- `empty = n` → counts empty slots.
- `full = 0` → counts filled slots.

Producer:

```
wait(empty)
wait(mutex)
  add item to buffer
signal(mutex)
signal(full)
```

Consumer:

```
wait(full)
wait(mutex)
  remove item from buffer
signal(mutex)
signal(empty)
```

👉 This prevents **race conditions**.

---

7. Dining Philosophers Problem

- 5 philosophers sitting, need 2 chopsticks to eat.
- Demonstrates **deadlock, starvation, resource sharing**.
- Solution: Use **semaphores, monitors, or odd-even pickup strategy**.

✅ Quick Revision – Session 9

- Deadlock → 4 conditions (Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait).
- Handling → Prevention, Avoidance (Banker's), Detection & Recovery.
- Deadlock ≠ Starvation.
- Semaphore = counter, Mutex = lock.

- Producer-Consumer → classic synchronization example.
- Dining Philosophers → synchronization + deadlock problem.
