Details of Linux commands top, ps,kill,wait,sleep,exit,nice with example.

1. top – Display Linux Tasks in Real-Time

Definition:
top provides a dynamic, real-time view of the running system. It shows processes, CPU usage, memory usage, and system load.

Syntax:

top [options]

Common Options:

- -u <username> → Show processes of a specific user
- -p <pid> → Monitor specific process ID
- -n <num> → Number of iterations to display before exiting
- -d <seconds> → Delay time between updates

Example:

top # Default view of all processes

top -u root # Show only root user's processes

top -p 1234 # Monitor process with PID 1234

top -n 5 -d 2 # Refresh every 2s, run 5 iterations

---

2. ps – Report a Snapshot of Current Processes

Definition:
ps gives a one-time snapshot of processes currently running.

Syntax:

ps [options]

Common Options:

- ps → Shows processes for the current shell
- ps -e or ps -A → All processes
- ps -f → Full format (UID, PID, PPID, CMD, etc.)
- ps aux → All processes in BSD style (commonly used)
- ps -u <user> → Show processes of a user

Example:

```
ps            # Current shell processes
ps -ef        # Full-format all processes
ps aux | grep python   # Find processes with 'python'
ps -u root    # Show root's processes
```

---

3. kill – Send Signal to a Process

Definition:
kill is used to stop or send a signal to a process.

Syntax:

```
kill [options] <PID>
```

Common Signals:

- -l → List all signals
- -9 → SIGKILL (force kill)
- -15 → SIGTERM (default, terminate gracefully)
- -STOP → Pause process
- -CONT → Resume paused process

Example:

```
ps -ef | grep firefox     # Find PID of Firefox
kill 1234                 # Terminate process 1234
kill -9 1234              # Forcefully kill process 1234
kill -STOP 1234           # Pause process
kill -CONT 1234           # Resume process
```

---

4. wait – Wait for a Process to Finish

Definition:
wait waits for background processes to complete before continuing.

Syntax:

```
wait [PID]
```

Example:

```
sleep 5 &    # Run sleep in background
wait         # Waits until sleep finishes
echo "Done"  # Printed after wait completes
```

With PID:

```
sleep 10 & pid=$!   # Store background PID
wait $pid
echo "Process $pid finished"
```

---

5. sleep – Delay Execution

Definition:
sleep pauses execution for a specified amount of time.

Syntax:

```
sleep NUMBER[SUFFIX]
```

Suffix:

- s → seconds (default)
- m → minutes
- h → hours
- d → days

Example:

```
sleep 5       # Sleep 5 seconds
sleep 2m      # Sleep 2 minutes
sleep 1h      # Sleep 1 hour
```

---

6. exit – Exit a Shell or Script

Definition:
exit is used to terminate a shell session or script, optionally returning a status code.

Syntax:

```
exit [N]
```

Example:

```
exit 0    # Exit with success
```

exit 1 # Exit with error code 1

Inside a script:

#!/bin/bash

echo "Start"

exit 1

echo "This won't run"

---

7. nice – Start Process with Modified Priority

Definition:
nice starts a process with a specified priority (default is 0). Lower values = higher priority, higher values = lower priority.

Range: **-20 (highest priority) → 19 (lowest priority)**.

Syntax:

nice -n <priority> command

Example:

nice -n 10 sleep 60 # Run sleep with lower priority

nice -n -5 ./myprogram # Run with higher priority (root required)

Check priorities:

ps -o pid,ni,comm -p <pid>

OSY Practical 1: System call commands in Linux such as fork(), exec(), getpid, pipe, exit, open, close, stat, uname.

Python code

fork()

```
import os

pid = os.fork()

if pid == 0:
    print("Child process")
else:
    print(f"Parent process, child PID: {pid}")
```

exec()

```
import os

print("Before exec")
os.execl("/bin/ls", "ls", "-l")
# This line will not execute unless execl fails
print("After exec (won't be printed)")
```

getpid()

```
import os

print(f"My PID is {os.getpid()}")
```

pipe()

```
import os

r, w = os.pipe()
```

```
pid = os.fork()
if pid == 0:
    os.close(r)  # Close read end
    os.write(w, b"Hello from child")
    os.close(w)
else:
    os.close(w)  # Close write end
    msg = os.read(r, 100)
    os.close(r)
    print(f"Parent received: {msg.decode()}")
```

---

exit()

```
import sys

print("Exiting with status 1")
sys.exit(1)
```

---

open(), close()

```
import os

fd = os.open("test.txt", os.O_CREAT | os.O_WRONLY)
os.write(fd, b"Hello, file!\n")
os.close(fd)
```

---

stat()

```
import os

file_stat = os.stat("test.txt")
print(f"File size: {file_stat.st_size} bytes")
```

---

uname()

```
import platform

print(f"System: {platform.system()}")
print(f"Node name: {platform.node()}")
print(f"Release: {platform.release()}")
print(f"Version: {platform.version()}")
print(f"Machine: {platform.machine()}")
```

Process related commands in Linux - top, ps, kill, wait, sleep, nice, renice,bg,fg.

Top

```
import subprocess

# Launch top (press 'q' to quit manually)
subprocess.run(["top"])
```

---

ps (list processes)

```
import subprocess

# Show all processes
subprocess.run(["ps", "-aux"])
```

---

sleep

```
import subprocess

# Sleep for 10 seconds
proc = subprocess.Popen(["sleep", "10"])
print(f"Started sleep process with PID: {proc.pid}")
```

---

wait

```
import subprocess

proc = subprocess.Popen(["sleep", "5"])
print(f"Waiting for process {proc.pid} to finish...")
proc.wait()
print("Process finished.")
```

---

kill

```
import os, signal, subprocess, time
```

```
# Start a process
proc = subprocess.Popen(["sleep", "30"])
print(f"Started process PID: {proc.pid}")

time.sleep(2)  # give time to run

# Kill the process
os.kill(proc.pid, signal.SIGKILL)
print(f"Process {proc.pid} killed.")
```

---

nice

```
import subprocess

# Run sleep with nice value 10 (lower priority)
subprocess.run(["nice", "-n", "10", "sleep", "5"])
```

---

renice

```
import subprocess

# Replace <PID> with your process ID
pid = "1234"
subprocess.run(["renice", "-n", "5", "-p", pid])
```

---

bg

```
import subprocess

# Run a process in background
proc = subprocess.Popen(["sleep", "15"])
print(f"Running in background with PID: {proc.pid}")
```

---

fg

```
import subprocess


# Foreground (blocks until finished)
subprocess.run(["sleep", "5"])
print("Foreground process finished.")
```

Process-Related Commands in Linux **execute these commands directly in a Linux terminal** without Python

```
top                              //press q for quit.
**********
ps        # shows your processes
ps -aux   # shows all processes with details
**********
sleep 10   # sleeps for 10 seconds
**********


sleep 10 &
wait
echo "All background processes finished"
**********
ps -aux           # find PID
kill 1234         # replace 1234 with PID
kill -9 1234      # force kill
**********
nice -n 10 sleep 60   # run sleep with nice value 10
**********
renice -n 5 -p 1234   # change priority of PID 1234 to 5
**********
sleep 100   # press Ctrl+Z to stop
```

```
bg          # sends it to background
**********
fg   # brings last job
fg %1  # bring job with ID 1 to foreground
**********
```

Practical 3 Commands for Sending Messages to Logged-in Users -who, cat, wall, write, mesg. (type following commands on trminal)

Who

cat message.txt | wall # send file content to all users

cat message.txt | write user1 # send file content to specific user

wall "System will go down for maintenance at 9 PM."

write username

write student

Hello, are you free for a meeting?

mesg y //allow message

mesg n //deny message

Practical 3 b List Processes Attached to a Shared Memory Segment: ipcs.

ipcs -m //specifically list shared memory segments

ipcs -m -i <shmid> //To see which processes are attached to a particular shared memory ID (shmid) ipcs -m -i 65536

Practical no 4 Write a C/Python program to calculate average waiting time and Turnaround Time of n processes with First Come First Serve (FCFS) CPU scheduling algorithm.

```
def fcfs(processes, burst_time):
    n = len(processes)
    wait_time = [0] * n
    tat = [0] * n

    # Waiting time calculation
    for i in range(1, n):
        wait_time[i] = burst_time[i - 1] + wait_time[i - 1]

    # Turnaround time calculation
    for i in range(n):
        tat[i] = burst_time[i] + wait_time[i]

    # Average waiting time & turnaround time
    avg_wt = sum(wait_time) / n
    avg_tat = sum(tat) / n

    # Print process details
    print("Process\tBurst Time\tWaiting Time\tTurnaround Time")
    for i in range(n):
        print(f"P{processes[i]}\t{burst_time[i]}\t\t{wait_time[i]}\t\t{tat[i]}")

    print(f"\nAverage Waiting Time: {avg_wt:.2f}")
    print(f'Average Turnaround Time: {avg_tat:.2f}")
```

```
# Driver Code
if __name__ == "__main__":
    n = int(input("Enter number of processes: "))
    processes = [i+1 for i in range(n)]
    burst_time = []

    for i in range(n):
        bt = int(input(f"Enter Burst Time for Process P{i+1}: "))
        burst_time.append(bt)

    print("\n--- FCFS Scheduling ---")
    fcfs(processes, burst_time)

//
/**/
Enter number of processes: 3
Enter Burst Time for Process P1: 5
Enter Burst Time for Process P2: 8
Enter Burst Time for Process P3: 12
```

FCFS Scheduling

Process	Burst Time	Waiting Time	Turnaround Time
P1	5	0	5
P2	8	5	13
P3	12	13	25

Average Waiting Time: 6.00

Average Turnaround Time: 14.33

/**/

Practical 5 Write a C/Python program to calculate average waiting time and Turnaround Time of n processes with Shortest Job First (SJF) CPU scheduling algorithm.

```
# Shortest Job First (SJF) Scheduling in Python (Non-preemptive)

def sjf(processes, burst_time):
    n = len(processes)

    # Sort processes by Burst Time
    sorted_processes = sorted(zip(processes, burst_time), key=lambda x: x[1])
    processes, burst_time = zip(*sorted_processes)

    wait_time = [0] * n
    tat = [0] * n

    # Waiting time calculation
    for i in range(1, n):
        wait_time[i] = burst_time[i-1] + wait_time[i-1]

    # Turnaround time calculation
    for i in range(n):
        tat[i] = burst_time[i] + wait_time[i]

    # Averages
    avg_wt = sum(wait_time) / n
    avg_tat = sum(tat) / n
```

```
    # Print details
    print("Process\tBurst Time\tWaiting Time\tTurnaround Time")
    for i in range(n):
        print(f"P{processes[i]}\t{burst_time[i]}\t\t{wait_time[i]}\t\t{tat[i]}")

    print(f"\nAverage Waiting Time: {avg_wt:.2f}")
    print(f"Average Turnaround Time: {avg_tat:.2f}")
# Driver Code
if __name__ == "__main__":
    n = int(input("Enter number of processes: "))
    processes = [i+1 for i in range(n)]
    burst_time = []

    for i in range(n):
        bt = int(input(f"Enter Burst Time for Process P{i+1}: "))
        burst_time.append(bt)

    print("\n--- Shortest Job First (SJF) Scheduling ---")
    sjf(processes, burst_time)

//
/**/
Enter number of processes: 4
Enter Burst Time for Process P1: 6
Enter Burst Time for Process P2: 8
Enter Burst Time for Process P3: 7
Enter Burst Time for Process P4: 3
```

Shortest Job First (SJF) Scheduling

Process	Burst Time	Waiting Time	Turnaround Time
P4	3	0	3
P1	6	3	9
P3	7	9	16
P2	8	16	24

Average Waiting Time: 7.00

Average Turnaround Time: 13.00

/**/

Practical 6 Write a C/Python program to calculate average waiting time and Turnaround Time of n processes with Priority CPU scheduling algorithm.

```
# Priority CPU Scheduling (Non-Preemptive)

def priority_scheduling(processes, burst_time, priority):
    n = len(processes)

    # Sort processes by priority (lower number = higher priority)
    sorted_processes = sorted(
        zip(processes, burst_time, priority),
        key=lambda x: x[2]
    )

    wt = [0] * n  # Waiting times
    tat = [0] * n  # Turnaround times

    # Calculate Waiting Time
    for i in range(1, n):
        wt[i] = wt[i - 1] + sorted_processes[i - 1][1]

    # Calculate Turnaround Time
    for i in range(n):
        tat[i] = wt[i] + sorted_processes[i][1]

    # Calculate averages
    avg_wt = sum(wt) / n
    avg_tat = sum(tat) / n
```

```
    # Print table
    print("\nPriority Scheduling Results")
    print("-------------------------------------------------------------")
    print("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time")
    print("-------------------------------------------------------------")
    for i in range(n):
        p, bt, pr = sorted_processes[i]
        print(f"P{p}\t\t{bt}\t\t{pr}\t\t{wt[i]}\t\t{tat[i]}")
    print("-------------------------------------------------------------")
    print(f"Average Waiting Time: {avg_wt:.2f}")
    print(f"Average Turnaround Time: {avg_tat:.2f}")


# ------------------- Main Program -------------------
if __name__ == "__main__":
    n = int(input("Enter number of processes: "))

    processes = []
    burst_time = []
    priority = []

    for i in range(n):
        processes.append(i + 1)
        bt = int(input(f"Enter Burst Time for Process P{i+1}: "))
        pr = int(input(f"Enter Priority for Process P{i+1} (lower = higher priority): "))
        burst_time.append(bt)
        priority.append(pr)
```

priority_scheduling(processes, burst_time, priority)

Output:

Enter number of processes: 3

Enter Burst Time for Process P1: 10

Enter Priority for Process P1 (lower = higher priority): 3

Enter Burst Time for Process P2: 5

Enter Priority for Process P2 (lower = higher priority): 1

Enter Burst Time for Process P3: 8

Enter Priority for Process P3 (lower = higher priority): 2

Priority Scheduling Results

Process	Burst Time	Priority	Waiting Time	Turnaround Time
P2	5	1	0	5
P3	8	2	5	13
P1	10	3	13	23

Average Waiting Time: 6.00

Average Turnaround Time: 13.67

Practical 7 Write a Python program to calculate average waiting time and Turnaround Time of n processes with Round Robin (RR) CPU scheduling algorithm.

```
# Round Robin CPU Scheduling


def round_robin(processes, burst_time, quantum):
    n = len(processes)
    rem_bt = burst_time[:]  # Remaining burst times
    wt = [0] * n            # Waiting times
    tat = [0] * n           # Turnaround times
    t = 0                   # Current time

    # Loop until all processes are done
    while True:
        done = True
        for i in range(n):
            if rem_bt[i] > 0:
                done = False
                if rem_bt[i] > quantum:
                    # Process executes for time quantum
                    t += quantum
                    rem_bt[i] -= quantum
                else:
                    # Process finishes execution
                    t += rem_bt[i]
                    wt[i] = t - burst_time[i]
                    rem_bt[i] = 0
        if done:
```

```
            break

    # Calculate Turnaround Times
    for i in range(n):
        tat[i] = burst_time[i] + wt[i]

    # Calculate averages
    avg_wt = sum(wt) / n
    avg_tat = sum(tat) / n

    # Print table
    print("\nRound Robin Scheduling Results")
    print("-------------------------------------------------------------")
    print("Process\tBurst Time\tWaiting Time\tTurnaround Time")
    print("-------------------------------------------------------------")
    for i in range(n):
        print(f"P{processes[i]}\t\t{burst_time[i]}\t\t{wt[i]}\t\t{tat[i]}")
    print("-------------------------------------------------------------")
    print(f"Average Waiting Time: {avg_wt:.2f}")
    print(f"Average Turnaround Time: {avg_tat:.2f}")



# ------------------- Main Program -------------------
if __name__ == "__main__":
    n = int(input("Enter number of processes: "))

    processes = []
    burst_time = []
```

```
    for i in range(n):
        processes.append(i + 1)
        bt = int(input(f"Enter Burst Time for Process P{i+1}: "))
        burst_time.append(bt)

    quantum = int(input("Enter Time Quantum: "))

    round_robin(processes, burst_time, quantum)
```

Output

Enter number of processes: 3

Enter Burst Time for Process P1: 24

Enter Burst Time for Process P2: 3

Enter Burst Time for Process P3: 3

Enter Time Quantum: 4

Round Robin Scheduling Results

Process	Burst Time	Waiting Time	Turnaround Time
P1	24	6	30
P2	3	4	7
P3	3	7	10

Average Waiting Time: 5.67

Average Turnaround Time: 15.67

Practical 10 Write a Python program on First In First Out (FIFO) Page Replacement algorithm.

```
# FIFO Page Replacement Algorithm


def fifo_page_replacement(pages, capacity):
    frame = []
    page_faults = 0

    print("\nFIFO Page Replacement Simulation")
    print("-------------------------------------------------")
    print("Pages\tFrames (after insertion/replacement)\tPage Fault")

    for page in pages:
        if page not in frame:
            if len(frame) < capacity:
                frame.append(page)
            else:
                frame.pop(0)  # remove oldest page
                frame.append(page)
            page_faults += 1
            fault = "Yes"
        else:
            fault = "No"

        print(f"{page}\t\t{frame}\t\t\t{fault}")

    print("-------------------------------------------------")
    print(f"Total Page Faults: {page_faults}")
```

```
# ------------------- Main Program -------------------
if __name__ == "__main__":
    n = int(input("Enter number of pages: "))
    pages = []
    for i in range(n):
        x = int(input(f"Enter page {i+1}: "))
        pages.append(x)

    capacity = int(input("Enter number of frames: "))

    fifo_page_replacement(pages, capacity)
```

Output

```
Enter number of pages: 12
Enter page 1: 1
Enter page 2: 3
Enter page 3: 0
Enter page 4: 3
Enter page 5: 5
Enter page 6: 6
Enter page 7: 3
Enter page 8: 2
Enter page 9: 4
Enter page 10: 3
Enter page 11: 0
```

Enter page 12: 3

Enter number of frames: 3

FIFO Page Replacement Simulation

Pages	Frames (after insertion/replacement)	Page Fault
1	[1]	Yes
3	[1, 3]	Yes
0	[1, 3, 0]	Yes
3	[1, 3, 0]	No
5	[3, 0, 5]	Yes
6	[0, 5, 6]	Yes
3	[5, 6, 3]	Yes
2	[6, 3, 2]	Yes
4	[3, 2, 4]	Yes
3	[3, 2, 4]	No
0	[2, 4, 0]	Yes
3	[4, 0, 3]	Yes

Total Page Faults: 9

Practical 11 Write a C/Python program on Least Recently Used (LRU) Page Replacement algorithm.

```
# LRU Page Replacement Algorithm


def lru_page_replacement(pages, capacity):
    frame = []
    page_faults = 0

    print("\nLRU Page Replacement Simulation")
    print("-------------------------------------------------")
    print("Pages\tFrames (after insertion/replacement)\tPage Fault")

    for i in range(len(pages)):
        page = pages[i]

        if page not in frame:
            if len(frame) < capacity:
                frame.append(page)
            else:
                # find the least recently used page
                lru_index = -1
                lru_page = None
                for f in frame:
                    if f not in pages[:i]:
                        lru_page = f
                        break
                    idx = len(pages[:i]) - 1 - pages[:i][::-1].index(f)
                    if lru_index == -1 or idx < lru_index:
```

```
                    lru_index = idx
                    lru_page = f
                frame.remove(lru_page)
                frame.append(page)

            page_faults += 1
            fault = "Yes"
        else:
            fault = "No"

        print(f"{page}\t\t{frame}\t\t\t{fault}")

    print("-------------------------------------------------")
    print(f"Total Page Faults: {page_faults}")


# -------------------- Main Program --------------------
if __name__ == "__main__":
    n = int(input("Enter number of pages: "))
    pages = []
    for i in range(n):
        x = int(input(f"Enter page {i+1}: "))
        pages.append(x)

    capacity = int(input("Enter number of frames: "))

    lru_page_replacement(pages, capacity)
```

Output

Enter number of pages: 12

Enter page 1: 1

Enter page 2: 3

Enter page 3: 0

Enter page 4: 3

Enter page 5: 5

Enter page 6: 6

Enter page 7: 3

Enter page 8: 2

Enter page 9: 4

Enter page 10: 3

Enter page 11: 0

Enter page 12: 3

Enter number of frames: 3

LRU Page Replacement Simulation

Pages	Frames (after insertion/replacement)	Page Fault
1	[1]	Yes
3	[1, 3]	Yes
0	[1, 3, 0]	Yes
3	[1, 3, 0]	No
5	[3, 0, 5]	Yes
6	[0, 5, 6]	Yes
3	[5, 6, 3]	Yes
2	[6, 3, 2]	Yes

4	[3, 2, 4]	Yes
3	[3, 2, 4]	No
0	[2, 4, 0]	Yes
3	[4, 0, 3]	Yes

Total Page Faults: 9

Practical 12 Write a Python program on sequential file allocation method.

```
# Sequential File Allocation in Python

def sequential_file_allocation(disk, file_name, start, length):
    n = len(disk)

    # Check if space is valid
    if start + length > n:
        print(f" Cannot allocate {file_name}: Out of disk bounds.")
        return

    # Check if required blocks are free
    for i in range(start, start + length):
        if disk[i] != -1:
            print(f"Cannot allocate {file_name}: Block {i} already allocated.")
            return

    # Allocate file
    for i in range(start, start + length):
        disk[i] = file_name
    print(f"File '{file_name}' allocated from Block {start} to Block {start + length - 1}.")



def display_disk(disk):
    print("\nDisk Status:")
    for i in range(len(disk)):
        print(f"Block {i}: {disk[i]}")
```

```
    print("-" * 40)

# ------------------ Main Program ------------------
if __name__ == "__main__":
    size = int(input("Enter total number of disk blocks: "))
    disk = [-1] * size   # -1 means free block

    while True:
        print("\nMenu: 1. Allocate File  2. Display Disk  3. Exit")
        choice = int(input("Enter choice: "))

        if choice == 1:
            file_name = input("Enter file name: ")
            start = int(input("Enter starting block: "))
            length = int(input("Enter length of file (in blocks): "))
            sequential_file_allocation(disk, file_name, start, length)

        elif choice == 2:
            display_disk(disk)

        elif choice == 3:
            print("Exiting program.")
            break

        else:
            print("Invalid choice. Try again.")
```

Output

Enter total number of disk blocks: 10

Menu: 1. Allocate File 2. Display Disk 3. Exit

Enter choice: 1

Enter file name: A

Enter starting block: 2

Enter length of file (in blocks): 4

File 'A' allocated from Block 2 to Block 5.

Menu: 1. Allocate File 2. Display Disk 3. Exit

Enter choice: 1

Enter file name: B

Enter starting block: 5

Enter length of file (in blocks): 3

Cannot allocate B: Block 5 already allocated.

Menu: 1. Allocate File 2. Display Disk 3. Exit

Enter choice: 2

Disk Status:

Block 0: -1

Block 1: -1

Block 2: A

Block 3: A

Block 4: A

Block 5: A

Block 6: -1

Block 7: -1

Block 8: -1

Block 9: -1

How to Execute program

1. **Open Terminal**
2. **Create a Python File**
3. **Type the Code**
4. **Save the File**
5. **Run the Python Program** (python3 program_name.py)