UNIT-1

EXPERIMENT-1

Aim: To study and perform the installation of the Linux (Ubuntu) operating system on a Windows machine using dual-boot or virtualization techniques, ensuring a functional environment for learning and practicing Linux-based software development and system administration.

Description: Linux is a free, open-source operating system based on Unix, created by Linus Torvalds

in 1991. It powers everything from servers and supercomputers to smartphones and IoT devices.

Known for its stability, security, and flexibility, Linux is widely used in tech industries. Different distributions (like Ubuntu, Fedora, and Debian) make it adaptable for various needs

File and Directory Commands:

pwd →Print working directory (shows current location).

ls →List files and directories.

ls -l →Detailed list (permissions, size, owner).

ls -a →Show hidden files too.

cd <directory> →Change directory.

cd .. →Go back one level.

mkdir <dirname> →Create new directory.

rmdir <dirname> →Remove empty directory.

rm -r <dirname> →Remove directory and contents.

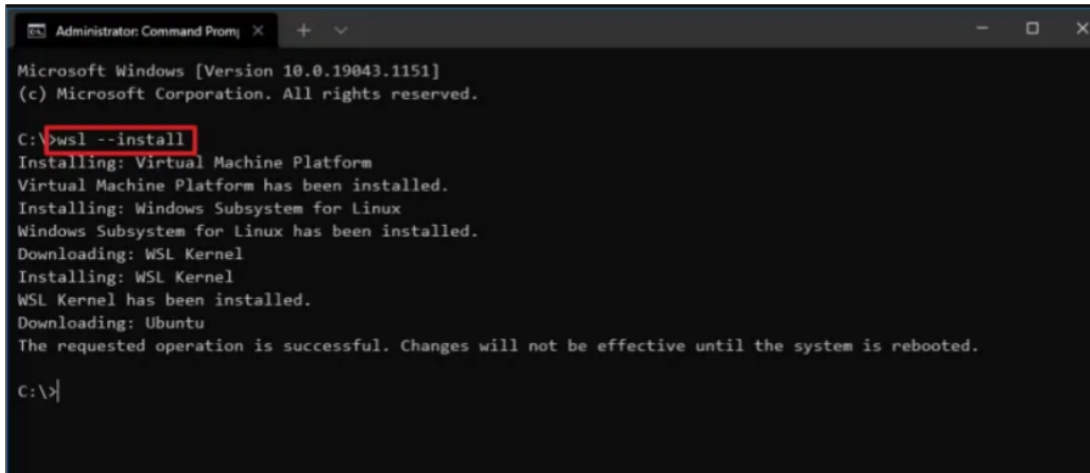exit →Logout or close terminal.

Steps to Install Linux (Ubuntu) in Windows (via WSL):

1. Open **PowerShell (Admin)** and run:

2. wsl –install
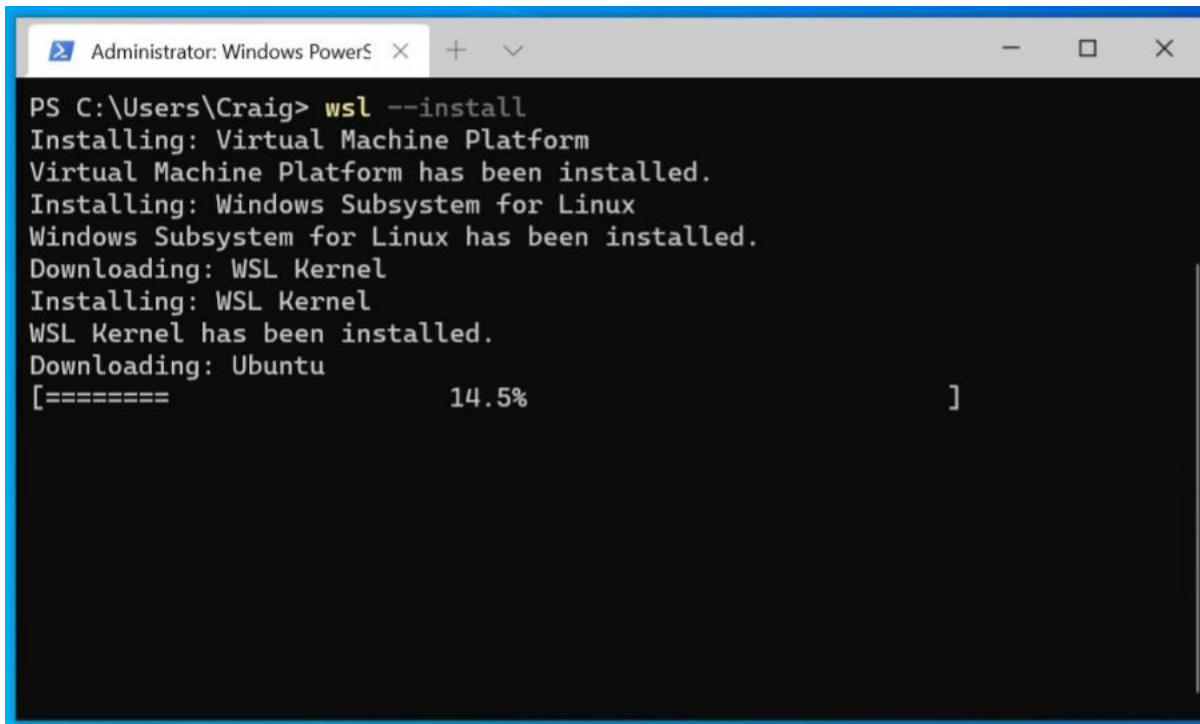
3. Wait for installation and select Ubuntu as your preferred distribution.

4. After setup, launch Ubuntu from the Start Menu.

5. Create a new Linux username and password.

6. Test installation by running basic commands in Ubuntu.

Some Basic Commands in Ubuntu:

1. **pwd**

o   Shows the current location in the filesystem.

o   Example:

o   pwd

o   /home/student

2. **man pwd**

o   Opens the manual page for the pwd command.

o   Example:

o   man pwd

3. **mkdir**

o   Creates a new directory.

o   Example:

o   mkdir myfolder

4. **ls**

o   Lists files and directories.

o   Example:

o   ls

o   Desktop Documents myfolder

5. **cat > file**

o   Creates a new text file and adds content.

o   Example:

o   cat > notes.txt

o   (type text, press Ctrl+D to save)

6. **vi file**

o   Opens file in *vi editor*.

o   Example:

o   vi notes.txt

o   Inside vi:

▪   Press **i** →insert mode.

▪   Press **Esc** →then type :wq →save and quit.

```
Try 'mkdir --help' for more information.
hi@Shannu44-laptop:~/s$ cd
hi@Shannu44-laptop:~$ ls
s
hi@Shannu44-laptop:~$ cat> x.txt
Welcome Linux^C
hi@Shannu44-laptop:~$ vi x.txt

[5]+  Stopped                 vi x.txt
hi@Shannu44-laptop:~$ cat> y.txt
Hello
^C
hi@Shannu44-laptop:~$ cp y.txt x.txt
hi@Shannu44-laptop:~$ vi y.txt

[6]+  Stopped                 vi y.txt
hi@Shannu44-laptop:~$ vi x.txt
hi@Shannu44-laptop:~$ clear
```

```
hi@Shannu44-laptop:~/s$ pwd
/home/hi/s
hi@Shannu44-laptop:~/s$ man pwd

[4]+  Stopped                 man pwd
hi@Shannu44-laptop:~/s$ mkdir
mkdir: missing operand
Try 'mkdir --help' for more information.
hi@Shannu44-laptop:~/s$ cd
hi@Shannu44-laptop:~$ ls
s
hi@Shannu44-laptop:~$ cat> x.txt
Welcome Linux^C
hi@Shannu44-laptop:~$ vi x.txt

[5]+  Stopped                 vi x.txt
hi@Shannu44-laptop:~$ |
```
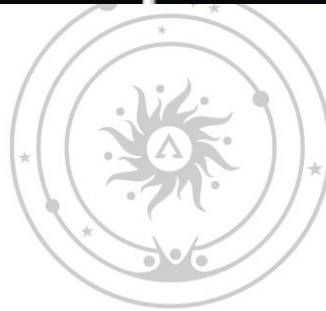
```
Hello
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"y.txt" 1L, 6B
```
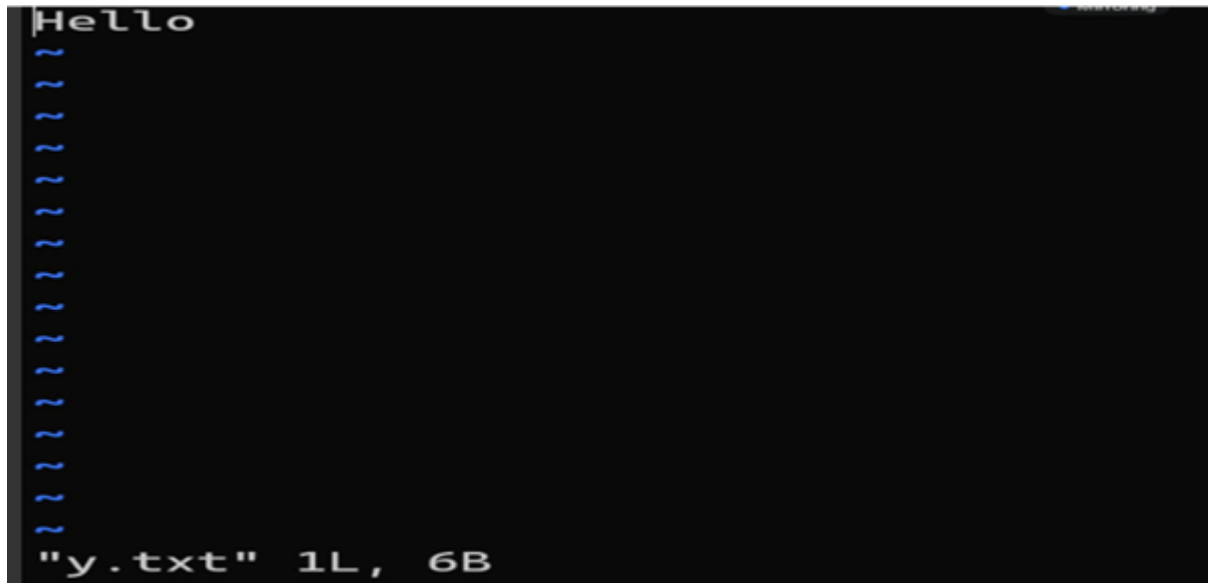
Conclusion: Linux (Ubuntu) was successfully installed on Windows using WSL. Basic commands for file and directory management were executed, demonstrating the ability to work with the Linux terminal environment.
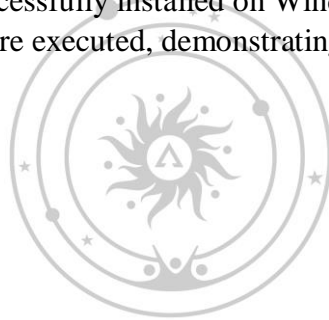
EXPERIMENT-2

Aim: To demonstrate the creation of a child process using the fork() system call in Linux.

Description: The fork() system call is used to create a new process by duplicating the calling

process.

The original process is the **parent**, and the newly created one is the **child**.

Both processes execute the same code but have different process IDs.

fork() returns:

o   > 0 →Parent process (child's PID is returned)

o   0 →Child process

o   < 0 →Error in creating process

Algorithm:

1. Start program execution.

2. Call fork() to create a new process.

3. If fork() returns 0, it's the child process.

4. If fork() returns a positive value, it's the parent process.

5. Print process ID for both parent and child.
6. End program.

Program:

```
#include <stdio.h>

#include <unistd.h>

int main() {
int pid;
pid = fork();
if (pid < 0) {

printf("Fork failed!\n");

} else if (pid == 0) {

printf("This is the child process. PID = %d\n", getpid());

} else {

printf("This is the parent process. PID = %d, Child PID = %d\n", getpid(), pid);

}
```

return 0;

}

Output:

```
This is the parent process. PID = 1851, Child PID = 1852
This is the child process. PID = 1852
```

Conclusion: The fork() system call was successfully demonstrated. A parent process and a child process were created, each with unique process IDs.

UNIT-2

EXPERIMENT-1A

Aim: To implement the First-Come, First-Served (FCFS) CPU scheduling algorithm.

Description:

FCFS is the simplest CPU scheduling algorithm.

Processes are executed in the order they arrive in the ready queue.

It is non-preemptive.

Average waiting time is highly affected by the order of process arrival.

Algorithm:

1. Input the number of processes and their burst times.

2. Sort processes by arrival order (if given).

3. Compute waiting time for each process:

$WT[i] = CT[i] - AT[i] - BT[i]$

4. Compute turnaround time:

$TAT[i] = WT[i] + BT[i]$

5. Display Gantt chart, waiting times, turnaround times, and averages.

Program:

```c
#include <stdio.h>

int main() {

int n, bt[20], wt[20], tat[20], i;

float avg_wt = 0, avg_tat = 0;
printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter Burst Time for each process:\n");

for (i = 0; i < n; i++) {

printf("P%d: ", i+1);

scanf("%d", &bt[i]);

}

wt[0] = 0;
```

```
for (i = 1; i < n; i++) {

wt[i] = wt[i-1] + bt[i-1];

}

for (i = 0; i < n; i++) {

tat[i] = wt[i] + bt[i];

avg_wt += wt[i];

avg_tat += tat[i];

}

printf("\nProcess\tBT\tWT\tTAT\n");

for (i = 0; i < n; i++) {

printf("P%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);

}

printf("\nAverage Waiting Time = %.2f", avg_wt/n);
printf("\nAverage Turnaround Time = %.2f\n", avg_tat/n);

return 0;

}
```

Output:

```
Enter number of processes: 3
Enter Burst Time for each process:
P1: 5
P2: 3
P3: 8


Process BT  WT   TAT
P1  5   0    5
P2  3   5    8
P3  8   8    16


Average Waiting Time = 4.33
Average Turnaround Time = 9.67
```

Conclusion:

The FCFS scheduling algorithm was successfully implemented and analyzed.

EXPERIMENT-1B

Aim: To implement the Shortest Job First (SJF) CPU scheduling algorithm.

Description:

SJF selects the process with the smallest burst time first.

It reduces average waiting time compared to FCFS.

Can be preemptive or non-preemptive (here, non-preemptive).

Algorithm:

1. Input processes and burst times.

2. Sort processes in ascending order of burst time.

3. Compute waiting time and turnaround time.

4. Display results with averages.

Program:

```
#include <stdio.h>

int main() {
int n, bt[20], wt[20], tat[20], p[20], i, j, temp;

float avg_wt = 0, avg_tat = 0;
printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter Burst Time for each process:\n");

for (i = 0; i < n; i++) {

printf("P%d: ", i+1);

scanf("%d", &bt[i]);
p[i] = i+1;
}
for (i = 0; i < n-1; i++) {
        for (j = i+1; j < n; j++) {
                if (bt[i] > bt[j]) {
                        temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
                        temp = p[i]; p[i] = p[j]; p[j] = temp;
                }
        }
}
wt[0] = 0;
for (i = 1; i < n; i++) {
```
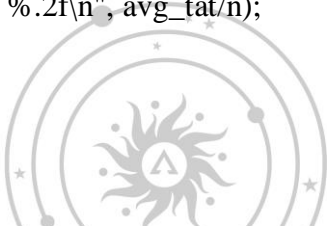
```
        wt[i] = wt[i-1] + bt[i-1];
}
for (i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
        avg_wt += wt[i];
        avg_tat += tat[i];
}
```

printf("\nProcess\tBT\tWT\tTAT\n");

for (i = 0; i < n; i++) {

printf("P%d\t%d\t%d\t%d\n", p[i], bt[i], wt[i], tat[i]);

}

printf("\nAverage Waiting Time = %.2f", avg_wt/n);

printf("\nAverage Turnaround Time = %.2f\n", avg_tat/n);

return 0;

}

Output:

```
Enter number of processes: 3
Enter Burst Time for each process:
P1: 2
P2: 5
P3: 8

Process BT  WT  TAT
P1  2   0   2
P2  5   2   7
P3  8   7   15

Average Waiting Time = 3.00
Average Turnaround Time = 8.00
```

Conclusion:

The SJF scheduling algorithm was successfully implemented and analyzed.

EXPERIMENT-2A

Aim: To simulate the CPU Scheduling using the **Priority Scheduling algorithm** in C++ and analyze process execution order based on priority.

Description: CPU Scheduling is the process of selecting a process from the ready queue and allocating CPU to it.

Priority Scheduling assigns a priority value to each process.

The CPU is allocated to the process with the **highest priority** (lowest priority number is usually considered highest priority).

If two processes have the same priority, they are scheduled according to their arrival order (or FCFS).

Algorithm/Steps:
1. Start the program.

2. Input the number of processes.

3. For each process, input:

o   Burst Time

o   Priority

4. Sort processes based on their priority (highest priority first).

5. Calculate:

o   Waiting Time (WT) for each process

o   Turnaround Time (TAT) for each process
6. Compute average WT and TAT.

7. Display the scheduling order and results.
8. End program.

Program:

#include <iostream>

#include <algorithm>
using namespace std;
struct Process {
int pid, bt, priority, wt, tat;
};

// Compare function for sorting based on priority

```cpp
bool compare(Process a, Process b) {
return (a.priority < b.priority);
}
int main() {
int n;
cout << "Enter number of processes: ";
cin >> n;
Process p[n];
for(int i = 0; i < n; i++) {
        p[i].pid = i + 1;
                cout << "Enter Burst Time and Priority for Process " << p[i].pid << ": ";
                cin >> p[i].bt >> p[i].priority;
}
// Sort processes by priority
sort(p, p + n, compare);

// Calculate Waiting Time and Turnaround Time

p[0].wt = 0;
for(int i = 1; i < n; i++) {
p[i].wt = p[i-1].wt + p[i-1].bt;
}

float avgWT = 0, avgTAT = 0;

for(int i = 0; i < n; i++) {
p[i].tat = p[i].wt + p[i].bt;
        avgWT += p[i].wt;
        avgTAT += p[i].tat;
}
avgWT /= n;
avgTAT /= n;
cout << "\nProcess\tBT\tPriority\tWT\tTAT\n";
for(int i = 0; i < n; i++) {
        cout << p[i].pid << "\t" << p[i].bt << "\t" << p[i].priority
                << "\t\t" << p[i].wt << "\t" << p[i].tat << endl;
}

cout << "\nAverage Waiting Time = " << avgWT;

cout << "\nAverage Turnaround Time = " << avgTAT << endl;

return 0;
}
```
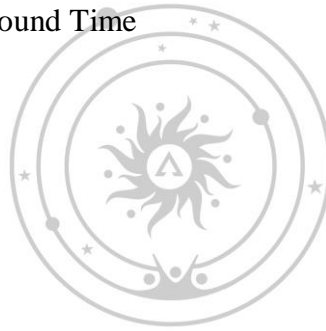
Output:

```
Enter number of processes: 3
Enter Burst Time and Priority for Process 1: 10 2
Enter Burst Time and Priority for Process 2: 5 1
Enter Burst Time and Priority for Process 3: 8 3

Process BT  Priority    WT   TAT
2   5   1         0    5
1   10  2         5    15
3   8   3         15   23

Average Waiting Time = 6.66667
Average Turnaround Time = 14.3333
```

Conclusion: The CPU Scheduling using the **Priority Algorithm** was successfully simulated in C++. The program computes **Waiting Time** and **Turnaround Time** for each process and demonstrates how processes are scheduled based on their priority.
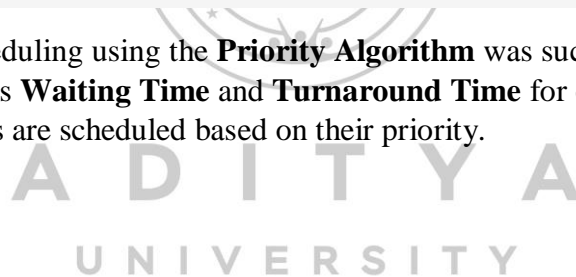
EXPERIMENT-2B

Aim: To simulate CPU Scheduling using the **Round Robin (RR) algorithm** in C++ and analyze

process execution order, waiting time, and turnaround time.

Description:

Round Robin (**RR**) is a **preemptive CPU scheduling algorithm**.

Each process is assigned a fixed **time quantum**.

The CPU executes a process for at most one quantum. If the process is not finished, it

is placed at the back of the ready queue.

This ensures **fairness** since every process gets CPU time in a cyclic order.

Algorithm/Steps:

1. Start the program.

2. Input the number of processes and their **Burst Times**.

3. Input the **Time Quantum**.

4. Initialize a queue for scheduling.

5. Execute each process for the given quantum:

o   If a process finishes before the quantum, mark it complete.

o   Otherwise, reduce its burst time and move it to the end of the queue.

6. Calculate **Waiting Time (WT)** and **Turnaround Time (TAT)** for each process.
7. Compute average WT and TAT.

8. Display process execution details.

9. End program.

Program:

```
#include <iostream>

#include <queue>
using namespace std;
struct Process {

int pid, bt, rt, wt, tat;
};
int main() {

int n, tq;
cout << "Enter number of processes: ";
```

```
cin >> n;

Process p[n];
for(int i = 0; i < n; i++) {

p[i].pid = i + 1;
cout << "Enter Burst Time for Process " << p[i].pid << ": ";
cin >> p[i].bt;

p[i].rt = p[i].bt; // remaining time

}

cout << "Enter Time Quantum: ";

cin >> tq;

int time = 0, completed = 0;

queue<int> q;
for(int i = 0; i < n; i++) q.push(i);
while(!q.empty()) {
int i = q.front();

q.pop();
if(p[i].rt > tq) {

time += tq;

p[i].rt -= tq;
q.push(i); // re-queue the process
} else {

time += p[i].rt;
p[i].wt = time - p[i].bt;
p[i].tat = time;

p[i].rt = 0;
        completed++;
} }

float avgWT = 0, avgTAT = 0;

cout << "\nProcess\tBT\tWT\tTAT\n";
for(int i = 0; i < n; i++) {
cout << p[i].pid << "\t" << p[i].bt
        << "\t" << p[i].wt << "\t" << p[i].tat << endl;
avgWT += p[i].wt;
avgTAT += p[i].tat;
}
```

cout << "\nAverage Waiting Time = " << avgWT / n;

cout << "\nAverage Turnaround Time = " << avgTAT / n << endl; return 0;
}

Output:

```
Enter number of processes: 3
Enter Burst Time for Process 1: 10
Enter Burst Time for Process 2: 5
Enter Burst Time for Process 3: 8
Enter Time Quantum: 3

Process BT  WT  TAT
1   10  13  23
2   5   9   14
3   8   14  22

Average Waiting Time = 12
Average Turnaround Time = 19.6667
```

Conclusion: The Round Robin CPU Scheduling algorithm was successfully simulated in C++.

UNIT-3

EXPERIMENT-1

Aim: To simulate the **Banker's Algorithm** for deadlock avoidance in C++.

Description:

The Banker's Algorithm is used in operating systems to **avoid deadlocks** by allocating

resources safely.

It checks if a system is in a **safe state** before granting resource requests.

Algorithm/Steps:
1. Input number of processes and resources.

2. Input Allocation, Maximum, and Available matrices.
3. Compute Need = Max – Allocation.

4. Apply Banker's Algorithm to check if the system is in a safe state.
5. If a safe sequence exists, display it; otherwise, report unsafe state.

Program:

```cpp
#include <iostream>

using namespace std;
int main() {

int n, m;

cout << "Enter number of processes: ";

cin >> n;

cout << "Enter number of resources: ";

cin >> m;

int alloc[n][m], max[n][m], avail[m];

cout << "Enter Allocation Matrix:\n";

for(int i=0;i<n;i++) for(int j=0;j<m;j++) cin >> alloc[i][j];
cout << "Enter Maximum Matrix:\n";

for(int i=0;i<n;i++) for(int j=0;j<m;j++) cin >> max[i][j];
cout << "Enter Available Resources:\n";

for(int i=0;i<m;i++) cin >> avail[i];

int need[n][m], finish[n]={0}, safeSeq[n], work[m];
```

```
for(int i=0;i<m;i++) work[i]=avail[i];
for(int i=0;i<n;i++) for(int j=0;j<m;j++) need[i][j]=max[i][j]-alloc[i][j];

int count=0;

while(count<n){

bool found=false;

for(int i=0;i<n;i++){

if(!finish[i]){

int j;
for(j=0;j<m;j++) if(need[i][j]>work[j]) break;
if(j==m){
for(int k=0;k<m;k++) work[k]+=alloc[i][k];
safeSeq[count++]=i;

finish[i]=1;

found=true;

}
}

}
        if(!found){ cout<<"System is not in a safe state\n"; return 0; }
}
cout<<"System is in a safe state.\nSafe sequence: ";
for(int i=0;i<n;i++) cout<<"P"<<safeSeq[i]<<" ";
cout<<endl;
return 0;
}
```
Output:

```
Enter number of processes: 5
Enter number of resources: 3
Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Maximum Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Available Resources:
3 3 2
System is in a safe state.
Safe sequence: P1 P3 P4 P0 P2
```

Conclusion: Banker's Algorithm was successfully implemented to check safe and unsafe states for deadlock avoidance.

EXPERIMENT-2:

Aim: To write a C program that illustrates **two processes communicating using shared memory**.

Description:

Shared memory allows **two or more processes to communicate** by accessing a common memory space.

It is one of the fastest IPC mechanisms.

Algorithm/Steps:

1. Create a shared memory segment using shmget().

2. Attach it to the process using shmat().

3. One process writes data into shared memory.

4. Another process reads data from shared memory.

5. Detach and remove shared memory segment.

Program:

#include <stdio.h>

#include <sys/ipc.h>
#include <sys/shm.h>

```c
#include <string.h>
int main() {

key_t key = ftok("shmfile",65);

int shmid = shmget(key,1024,0666|IPC_CREAT);
char *str = (char*) shmat(shmid,(void*)0,0);

printf("Write Data: ");

fgets(str, 1024, stdin);

printf("Data written in memory: %s\n", str);

shmdt(str);

return 0;

}
```

Output:

```
Write Data: LUCKY
Data written in memory: LUCKY
```

Conclusion: Shared memory IPC was successfully demonstrated, showing inter-process data exchange.

EXPERIMENT-3:

Aim: To create a **thread using the pthreads library** in C and run its function.

Description:

    Threads allow concurrent execution within a process.

    Pthreads (pthread.h) provide APIs for creating and managing threads.

Algorithm/Steps:
    1. Include pthread.h.

2. Define a function to be executed by a thread.

3. Create a thread using pthread_create().

4. Wait for the thread to finish using pthread_join().

Program:

```c
#include <stdio.h>

#include <pthread.h>
void* threadFunc(void* arg) {

printf("Hello from thread!\n");
```

```
return NULL;

}
int main() {

pthread_t tid;
pthread_create(&tid, NULL, threadFunc, NULL);
pthread_join(tid, NULL);
printf("Thread execution completed.\n");

return 0;

}
```
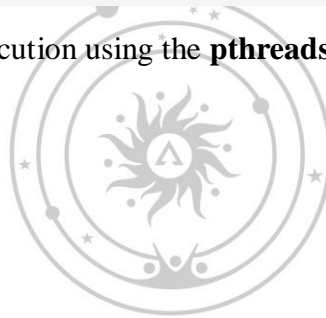
Output:

```
Hello from thread!
Thread execution completed.
```

Conclusion: Thread creation and execution using the **pthreads library** was successfully

demonstrated.

EXPERIMENT-4

Aim: To illustrate **concurrent execution of threads** using the pthreads library in C.

Description:

Multiple threads can run **concurrently** within the same process.

Each thread executes independently but shares memory.

Algorithm/Steps:

1. Define multiple functions for threads.

2. Create multiple threads using pthread_create().

3. Let each thread execute its function concurrently.
4. Use pthread_join() to synchronize thread completion.

Program:

```c
#include <stdio.h>

#include <pthread.h>
void* task1(void* arg) {

for(int i=0;i<5;i++) printf("Task 1 running\n");

return NULL;
}
void* task2(void* arg) {

for(int i=0;i<5;i++) printf("Task 2 running\n");

return NULL;
}
int main() {
pthread_t t1, t2;
pthread_create(&t1, NULL, task1, NULL);
pthread_create(&t2, NULL, task2, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);

printf("Both threads finished execution.\n");

return 0;
}
```
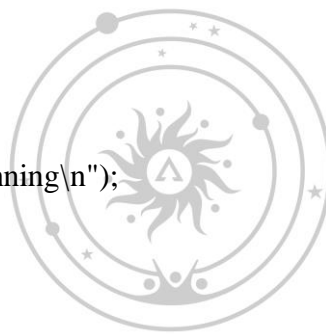
Output:

```
Task 1 running
Task 1 running
Task 1 running
Task 1 running
Task 1 running
Task 2 running
Task 2 running
Task 2 running
Task 2 running
Task 2 running
Both threads finished execution.
```

Conclusion: Concurrent execution of threads using **pthreads** was successfully simulated, showing independent task execution within a process.
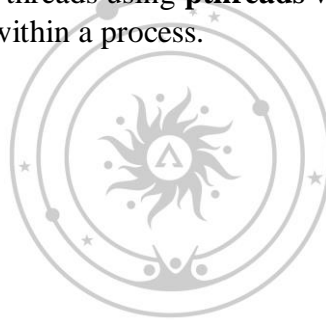
UNIT-4

EXPERIMENT-1

Aim: To simulate memory management using Multiprogramming with a Fixed Number of

Tasks (MFT).

Description: MFT divides the memory into fixed partitions. Each process is allocated exactly one partition. Internal fragmentation occurs when a process does not fully use its partition.

Algorithm/Steps:

1. Input total memory and partition size.

2. Calculate number of partitions = total memory / partition size.

3. Input process sizes.

4. Allocate each process to a partition if it fits.

5. Show internal fragmentation and unused memory.

Program:

```cpp
#include <iostream>

using namespace std;
int main() {
int ms, ps, nop, mp[10], n;
cout << "Enter total memory size: ";
cin >> ms;

cout << "Enter partition size: ";

cin >> ps;
n = ms / ps;

cout << "Total partitions: " << n << endl;

cout << "Enter number of processes: ";
cin >> nop;
for (int i = 0; i < nop; i++) {
cout << "Enter memory required for process " << i+1 << ": ";
cin >> mp[i];
}
for (int i = 0; i < nop && i < n; i++) {          if (mp[i] <= ps)

cout << "Process " << i+1 << " allocated in partition " << i+1
<< " with internal fragmentation: " << ps - mp[i] << endl;

else
```

cout << "Process " << i+1 << " not allocated (too large)." << endl; }

return 0;

}

Output:

```
Enter total memory size: 1000
Enter partition size: 200
Total partitions: 5
Enter number of processes: 3
Enter memory required for process 1: 180
Enter memory required for process 2: 20
Enter memory required for process 3: 250
Process 1 allocated in partition 1 with internal fragmentation: 20
Process 2 allocated in partition 2 with internal fragmentation: 180
Process 3 not allocated (too large).
```

Conclusion: MFT was simulated and internal fragmentation was observed when process sizes were smaller than the partition size.
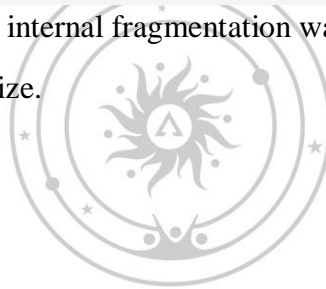
EXPERIMENT-3

Aim: To simulate the FIFO page replacement algorithm.

Description: FIFO replaces the oldest page in memory when a new page must be loaded and

no free frames are available.

Algorithm/Steps:
1. Input number of frames.

2. Input page reference string.

3. Load pages sequentially.

4. If page fault occurs and frames are full, replace the oldest page.

Program:

```
#include <iostream>

#include <queue>

#include <vector>

using namespace std;
int main() {
int n, f, page, faults = 0;
cout << "Enter number of frames: ";
cin >> f;
cout << "Enter number of pages: ";
cin >> n;
vector<int> pages(n);
cout << "Enter reference string: ";
for (int i = 0; i < n; i++) cin >> pages[i];
queue<int> q;
vector<int> frame;
for (int i = 0; i < n; i++) {
        page = pages[i];
        bool found = false;
        for (int x : frame) if (x == page) found = true;
        if (!found) {
                if (frame.size() < f) frame.push_back(page);
                else {

int victim = q.front(); q.pop();

for (int j = 0; j < frame.size(); j++)

if (frame[j] == victim) frame[j] = page;
```

```
        }
        q.push(page);
        faults++;
        }
        cout << "Frames: ";
        for (int x : frame) cout << x << " ";
        cout << endl;
    }
cout << "Total Page Faults: " << faults << endl;
return 0;
}
```

Output:

```
Enter number of frames: 3
Enter number of pages: 7
Enter reference string: 1 2 3 4 1 2 5
Frames: 1
Frames: 1 2
Frames: 1 2 3
Frames: 4 2 3
Frames: 4 1 3
Frames: 4 1 2
Frames: 5 1 2
Total Page Faults: 7
```

Conclusion: FIFO page replacement was simulated and the number of page faults was observed.

EXPERIMENT-4

Aim: To simulate the LRU page replacement algorithm.

Description: Least Recently Used (LRU) replaces the page that has not been used for the

longest time.

Algorithm/Steps:

1. Input number of frames and reference string.

2. Keep track of last used time for each page.

3. Replace the page with the longest idle time when a page fault occurs.

Program:

#include <iostream>

#include <vector>

```
#include <unordered_map>
using namespace std;
int main() {
        int n, f, page, faults = 0;
        cout << "Enter number of frames: ";
        cin >> f;
        cout << "Enter number of pages: ";
        cin >> n;
        vector<int> pages(n);
        cout << "Enter reference string: ";
        for (int i = 0; i < n; i++) cin >> pages[i];
        vector<int> frame;
        unordered_map<int,int> lastUsed;
        for (int i = 0; i < n; i++) {
        page = pages[i];
        bool found = false;
        for (int x : frame) if (x == page) found = true;
        if (!found) {
                if (frame.size() < f) frame.push_back(page);

                else {
                int lru_index = 0;
                int min_last = i;
                for (int j = 0; j < frame.size(); j++) {
                        if (lastUsed[frame[j]] < min_last) {
                                min_last = lastUsed[frame[j]];
                                lru_index = j;
                                        }
```

```
                    }
              frame[lru_index] = page;
       }
faults++;
}
lastUsed[page] = i;
cout << "Frames: ";
for (int x : frame) cout << x << " ";

cout << endl;
}
cout << "Total Page Faults: " << faults << endl;
return 0;
}
```

Output:

```
Enter number of frames: 3
Enter number of pages: 7
Enter reference string: 1 2 3 4 1 2 5
Frames: 1
Frames: 1 2
Frames: 1 2 3
Frames: 4 2 3
Frames: 4 1 3
Frames: 4 1 2
Frames: 5 1 2
Total Page Faults: 7
```

Conclusion: LRU page replacement was simulated and the least recently used page was replaced on each fault.

EXPERIMENT-5: Simulate File Allocation Strategies

Aim:

To simulate file allocation strategies: (a) Sequential, (b) Indexed, (c) Linked.
Description:

Sequential: Files occupy consecutive blocks.

Indexed: An index block maintains pointers to all file blocks.

Linked: Each block contains a pointer to the next block.
Algorithm/Steps:

1. Input number of blocks and file sizes.

2. For Sequential, allocate continuous blocks.

3. For Indexed, allocate an index block with pointers.
4. For Linked, create chain of blocks.

Program(a):

```cpp
#include <iostream>

#include <vector>

using namespace std;
int main() {
int n, f, start, length;
cout << "Enter total number of disk blocks: ";
cin >> n;

vector<int> block(n, 0); // 0 = free, 1 = allocated

cout << "Enter number of files: ";

cin >> f;
for (int i = 0; i < f; i++) {

cout << "\nEnter starting block of file " << i+1 << ": ";

cin >> start;

cout << "Enter length of file " << i+1 << ": ";

cin >> length;
bool allocated = true;
        for (int j = start; j < start + length; j++) {
                if (j >= n || block[j] == 1) {
                                allocated = false;
                        break;
```

```
                    }
            }
        if (allocated) {
                for (int j = start; j < start + length; j++)
                        block[j] = 1;
                cout << "File " << i+1 << " allocated blocks: ";
                for (int j = start; j < start + length; j++)
                                cout << j << " ";
                cout << endl;
        } else {
                cout << "File " << i+1 << " cannot be allocated." << endl;
        }
    }
}
return 0;
}
```

Output:

```
Enter total number of disk blocks: 10
Enter number of files: 2

Enter starting block of file 1: 2
Enter length of file 1: 3
File 1 allocated blocks: 2 3 4

Enter starting block of file 2: 5
Enter length of file 2: 2
File 2 allocated blocks: 5 6
```

Program(b):

```
#include <iostream>

#include <vector>

using namespace std;
int main() {
int n, f, indexBlock, length;
cout << "Enter total number of disk blocks: ";
cin >> n;
vector<int> block(n, 0);

cout << "Enter number of files: ";
```

```
cin >> f;
for (int i = 0; i < f; i++) {

cout << "\nEnter index block for file " << i+1 << ": ";

cin >> indexBlock;
if (block[indexBlock] == 1) {
cout << "Index block already allocated. File cannot be stored.\n";

continue;

}
block[indexBlock] = 1;

cout << "Enter number of blocks needed for file " << i+1 << ": ";

cin >> length;
vector<int> allocated;
cout << "Enter block numbers: ";
for (int j = 0; j < length; j++) {
int b;
cin >> b;
if (b < n && block[b] == 0) {
        block[b] = 1;
            allocated.push_back(b);
} else {
            cout << "Block " << b << " already allocated or invalid.\n";

}
    }
    cout << "File " << i+1 << " stored using index block "
                << indexBlock << " -> ";
    for (int b : allocated) cout << b << " ";
    cout << endl;
}
return 0;
}
```
Output:

```
Enter total number of disk blocks: 20
Enter number of files: 2

Enter index block for file 1: 15
Enter number of blocks needed for file 1: 3
Enter block numbers: 1 2 3
File 1 stored using index block 15 -> 1 2 3

Enter index block for file 2: 15
Index block already allocated. File cannot be stored.
```

Program(c):

#include <iostream>

#include <vector>

using namespace std;
struct Node {
int block;
Node* next;
};
int main() {
int n, f, length;
cout << "Enter total number of disk blocks: ";
cin >> n;
vector<int> block(n, 0);

cout << "Enter number of files: ";

cin >> f;
for (int i = 0; i < f; i++) {

cout << "\nEnter number of blocks required for file " << i+1 << ": ";

cin >> length;
Node* head = nullptr;
Node* temp = nullptr;
cout << "Enter block numbers: ";
for (int j = 0; j < length; j++) {
int b;
cin >> b;
if (b < n && block[b] == 0) {   block[b] = 1;

```
Node* newNode = new Node{b, nullptr};
            if (!head) head = newNode;
            else temp->next = newNode;
            temp = newNode;
} else {

            cout << "Block " << b << " already allocated or invalid.\n";
}
}
cout << "File " << i+1 << " allocated blocks (linked): ";
temp = head;
while (temp) {
cout << temp->block << " -> ";

    temp = temp->next;            }
    cout << "NULL" << endl;     }
return 0; }
```

Output:

```
Enter total number of disk blocks: 20
Enter number of files: 2

Enter number of blocks required for file 1: 4
Enter block numbers: 10 12 1 8
File 1 allocated blocks (linked): 10 -> 12 -> 1 -> 8 -> NULL

Enter number of blocks required for file 2: 3
Enter block numbers: 1 5 6
Block 1 already allocated or invalid.
File 2 allocated blocks (linked): 5 -> 6 -> NULL
```

Conclusion: We successfully simulated **Sequential, Indexed, and Linked File Allocation** strategies in C++. Each strategy illustrated how files are stored in secondary memory and demonstrated issues like external fragmentation, indexing, and linked block traversal

UNIT-5

AIM: To compare how **Windows** and **Unix/Linux** operating systems implement **Secondary Storage Structures, System Protection, and System Security** through case study analysis.

DESCRIPTION / THEORY:

1. Secondary Storage Structure:

Disk Structure & Attachment:

o   Windows: NTFS, basic/dynamic disks, SATA/USB/iSCSI attachments.

o   Unix/Linux: ext4, XFS, ZFS, devices as /dev/sda, mounted with mount.

Disk Scheduling:

o   Windows: SCAN/Elevator algorithms, NCQ.

o   Linux: CFQ, Deadline, NOOP schedulers.

RAID Structure:

o   Windows: Storage Spaces (software RAID), hardware RAID controllers.

o   Linux: mdadm for RAID 0,1,5,6,10.

Stable Storage:

o   Windows: NTFS journaling, chkdsk, Volume Shadow Copy.

o   Linux: ext4 journaling, fsck, LVM snapshots.

2. System Protection:

Goals of Protection: Prevent misuse of CPU, memory, and files; ensure authorized access.

Principles and Domains:

o   Windows: Access Control Lists (ACLs), Active Directory domains.

o   Linux: User/Group/Other model, rwx permissions, sudo.

Access Matrix & Control:

o   Windows: ACLs implement access matrix.

o   Linux: Permission bits, chmod, chown, setfacl.

Revocation of Rights: Both OSs allow instant revocation, but require auditing for active sessions.

3. System Security:

Program Threats:

o   Windows: Viruses, ransomware, buffer overflows. Mitigation: Defender, Code

Signing.

o   Linux: Rootkits, privilege escalation. Mitigation: SELinux, AppArmor.

System & Network Threats:

o   Windows: Worms, phishing, DoS. Mitigation: Firewall, BitLocker, auto-

updates.

o   Linux: SSH brute force, DDoS. Mitigation: iptables/ufw, fail2ban, Snort.

ALGORITHM / STEPS (Case Study Approach)

1. Study secondary storage management in both Windows and Linux.

2. Compare protection models: ACLs vs. permission bits.

3. Analyze access matrix and revocation implementation.

4. Identify common program, system, and network threats in both OSs.

5. Compare mitigation strategies (firewalls, encryption, auditing tools).

6. Summarize similarities and differences.

COMPARISON:

Feature	Windows	Unix/Linux
File System	NTFS (journaling, EFS)	ext4, XFS, ZFS
Disk Scheduling	SCAN, NCQ	CFQ, Deadline, NOOP
RAID	Storage Spaces, Hardware RAID	mdadm, Hardware RAID
Stable Storage	Journaling, chkdsk, Shadow Copy	Journaling, fsck, LVM snapshots
Protection Model	ACLs, Active Directory	User/Group/Other, rwx, sudo
Access Control	ACLs on files and objects	chmod, chown, setfacl
Program Threats	Viruses, ransomware	Rootkits, privilege escalation
Network Threats	Worms, phishing, DoS	SSH brute force, DDoS
Security Tools	Defender, BitLocker, Firewall	SELinux, iptables, fail2ban, Snort

CONCLUSION: Windows is preferred in enterprise environments requiring integration and centralized management, while Unix/Linux dominates servers and critical systems due to its security,

flexibility, and stability. This case study highlights how different OS designs achieve the same fundamental goals of efficiency, protection, and security.