# Explore Linux Commands (At Least 20 commands)

1  pwd Command

2.  mkdir Command

3.  rmdir Command

4.  ls Command - The ls command is used to display a list of content of a directory.

5.  cd Command - The cd command is used to change the current directory.

6.  touch Command - The touch command is used to create empty files. We can create multiple empty files by

executing it once. Syntax:  touch <file name>

7.  cat Command - It can be used to create a file, display content of the file

8.  rm Command  - The rm command is used to remove a file.

9.  cp Command - The cp command is used to copy a file or directory.
cp <existing file name> <new file name>

10.  mv Command - The mv command is used to move a file or a directory form one location to another location.  mv <file name> <directory path>

11. head Command - It displays the first 10 lines of a file.  head <file name>

12. tail Command - It displays the last ten lines of the file content. tail <file name>

13. tac Command  -  The tac command is the reverse of cat command.
tac <file name>

14.  id Command  -  display the user ID (UID) and group ID (GID).

15.  comm Command - used to compare two files or streams. comm <file1>< file2>

16. tee command  --  cat<filename> | tee <new File> |  cat or tac |.....

17. tr Command -  lower case to upper case.  Cat file_name | tr [a-z] [z-a]

18. uniq Command -  cat<file name> | uniq

19. wc Command - count the lines, words, and characters in a file. wc <file name>

20. od Command - The od command is used to display the content of a file in different s, such as hexadecimal,
octal, and ASCII characters.
Syntax:
1. od -b <fileName>  // Octal format
2. od -t x1 <fileName>  // Hexa decimal format
3. od -c <fileName>  // ASCII character format

21. sort Command - used to sort files in alphabetical order.  sort <file name>

22. locate Command - used to search a file by file name.  locate<file name>

23. date Command - Syntax:  date

24. cal Command  Syntax:  cal  cal -A3  cal -B3
25. time Command  ---  time
26. sleep Command -- Syntax:  sleep <time u want>

# Questions on changing permissions of files and folders

pwd

mkdir make a directory

cd change directory

touch / cat make file

ls list of content

ls -l view permissions for ALL

ls -l <file name> view permissions

## CHANGING PERMISSION

Relative Permission

## chmod category +/- Permission <file name> (syntax)

eg – chmod u+x file1

chmod a+rwx file1

| Category | Operation | Permission |
|---|---|---|
| u – user<br>g -group<br>o -others<br>a -All(ugo) | + --- Assign permission<br>- --- Removes permission<br>= --- Assigns absolute permission | r – Read permission<br>w – Write permission<br>x – Execute permission |

**Absolute permission**

| Octal | Binary | Permission |
|---|---|---|
| 0 | 000 | --- |
| 1 | 001 | --x |
| 2 | 010 | -w- |
| 3 | 011 | -wx |
| 4 | 100 | r-- |
| 5 | 101 | r-x |
| 6 | 110 | rw- |
| 7 | 111 | rwx |

Chmod ugo <file name>

[in place of ugo we put the number corresponding to the permission to be given ]

**Write Linux shell script to perform following task:**

**1) Display current shell, home directory, operating system type, current working directory**

```
echo "Current Shell: $SHELL"
echo "Home Directory: $HOME"
echo "Operating System Type: $(uname -s)"
echo "Current Working Directory: $(pwd)"
```

**2) Write Linux shell script to Display even number between range 1 to 100.**

---

```
#!/bin/bash

echo "Number :"
read num;
for (( i = 1; i >= $num; i++ ))
do
  if [ $(($i % 2)) -eq 0 ]
  then
      echo $i
  fi
done
```

## OR

```
#!/bin/bash

echo "Number :"
read num;
for i in $num
do
  if [ $(($i % 2)) -eq 0 ]
  then
      echo $i
  fi
done
```

—----------

# What is vi editor? How to save files in vi editor.

The Vi editor is a text editor commonly found on Unix and Unix-like systems such as Linux. It's known for its efficiency and versatility in editing text files directly from the terminal. Vi has two main modes: command mode and insert mode.
Here's how to save files in Vi:

**Open a File in Vi:**

To open a file named example.txt in Vi, you can use the command:
Syntax - vi example.txt
Save Changes and Exit:
After editing, you'll need to save your changes and exit Vi. Here's how:
**Save Changes**: **Press Esc to switch from insert mode to command mode**. Then, **type :w and press Enter**. This command saves the changes but does not exit Vi.
**Save Changes and Exit:** If you want to **save changes and exit Vi simultaneously, press Esc to switch to command mode, then type :wq and press Enter.**
**Exit Without Saving**: If you want to exit Vi without saving changes, press Esc to switch to command mode, then type **:q!** and press Enter.

**Here's a summary of the commands:**

i: Switch to insert mode (for editing text).
Esc: Switch back to command mode (from insert mode).
:w: Save changes (without exiting Vi).
:wq: Save changes and exit Vi.
:q!: Exit Vi without saving changes.

# Write a shell script to FIND THE GIVEN NUMBER IS EVEN OR ODD

```
echo “---------Even or odd in shell script----------”
echo -n “enter a number: ”
read t
echo -n “Result: ”
if [‘expr $t %2’ == 0]
then
        echo “$t is even.”
else
       echo “$t id odd.”
fi
```

## What is a shell script?

A shell script is a list of commands in a computer program that is run by the Unix shell which is a command line interpreter. A shell script is a plain text file containing a series of commands that are interpreted and executed by a shell interpreter. Shell scripts are commonly used in Unix, Linux.

## Write a shell script to display the addition of two numbers.

```
echo “enter first number - ”
read num1
echo “enter second number - ”
read num2
sum=$(($num1+$num2))

echo “addition is  $sum”
```

---

# State types of shell. what is the default shell.

SHELL is a program which provides the interface between the user and an operating system. A Shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input.

Common types of shells:

Bash (Bourne Shell): Bash is the most common shell used in Linux systems. It is the default shell for most Linux distributions and provides a wide range of features and capabilities, including command-line editing, command history, and job control. (The Bourne shell (sh) is the most popular Unix shell. The default prompt of Bourne shell is $.)

2.] Korn Shell (ksh): Korn shell is another popular shell that provides many of the same features as Bash, along with some additional features such as better error messages and a more powerful scripting language. The Korn shell (ksh) is **backward compatible** with the Bourne shell and includes many features of C shell.

3.] C Shell (csh): C shell (csh) provides a few advantages over the Bourne shell. It provides history features and aliasing of commands. The default prompt of the C shell is %.

4.] Bourne Again Shell (bash) is a free shell replacement for the Bourne shell. The Korn and Bourne Again Shell are supersets of Bourne shell.

# Write a shell script to DISPLAY HELLO WORLD.

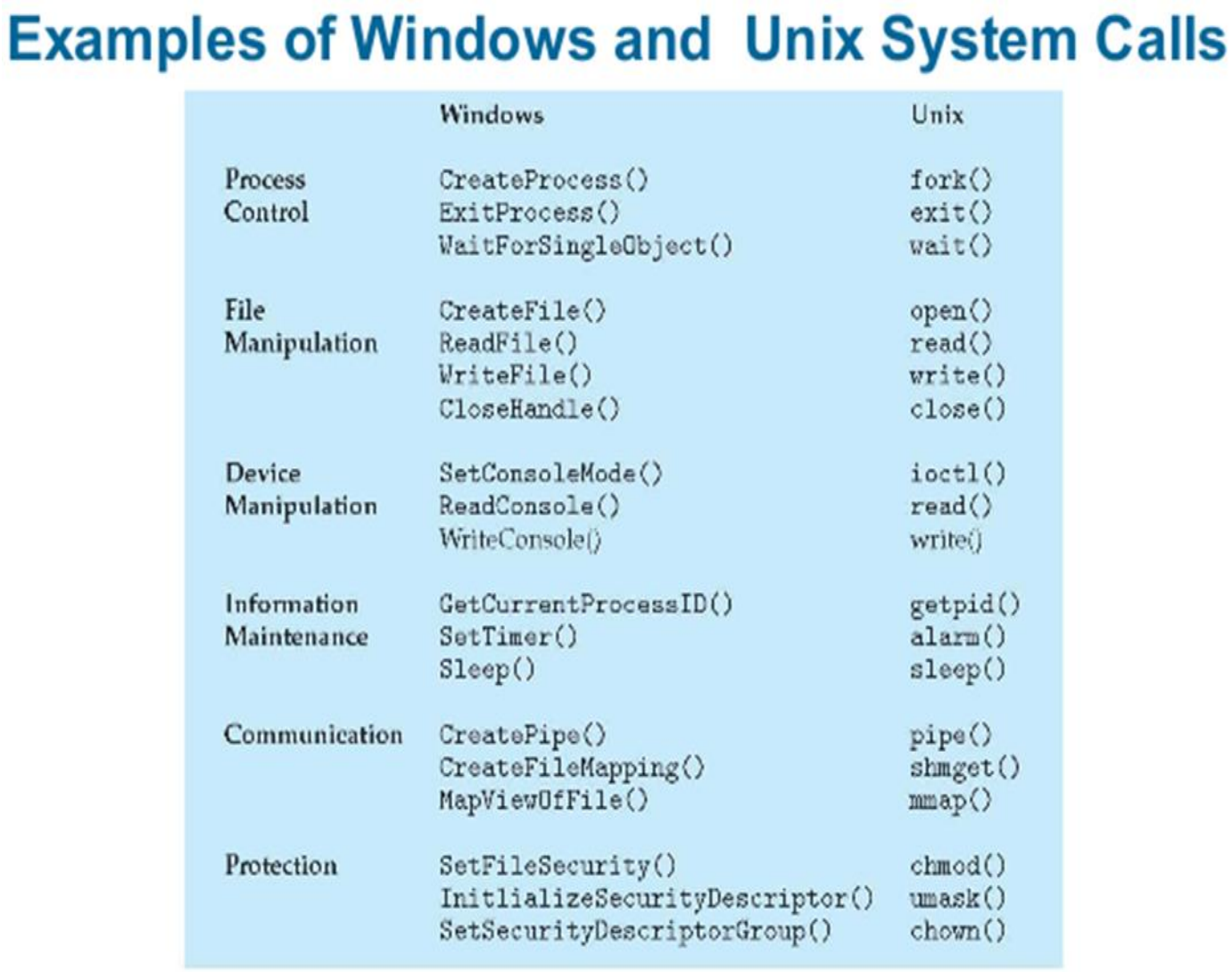
```
echo "Hello World"
```

## Write a shell script to display today's date, last 3 months calendar, current working directory and current shell.

```
echo "Today's Date: $(date +%Y-%m-%d)"
echo "Last 3 Months Calendar:"
cal -3
echo "Current Working Directory: $(pwd)"
echo "Current Shell: $SHELL"
```

## What is a system call? List types of Linux system calls and explain fork system calls in detail.

A system call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.




fork() in C

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork () call (parent process). After a new child process is created, both processes will execute the next instruction following the fork () system call. A child process uses the same pc (program counter), same CPU registers, same open files which use in the parent process. It takes no parameters and returns an integer value. Below are different values returned by fork (). Negative Value: creation of a child process was unsuccessful. Zero: Returned to the newly created child process. Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

## Write C code to create one parent process and child process.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
printf("MY NAME IS JHALAK DEDHIA\n");
pid_t pid=fork();
if(pid < 0)
{
printf("Unable to create process\n");
}
else if(pid == 0)
{
printf("child process PROCESS_ID %d and parent PROCESS_ID %d\n", getpid(), getppid());
}
else if(pid > 0)
{
printf("Parent Process PROCESS_ID %d\n", getpid());
}
return 0;
}
```

## Write C code to demonstrate the used of execv() system call.

```
oem@22D20310:~$ cd JHALAK
oem@22D20310:~/JHALAK$ ls
abc  def  file1.c  file2.c  file3.c  file4.c  ghi  jkl
oem@22D20310:~/JHALAK$ gedit exforc1.c
^Z
[1]+  Stopped                 gedit exforc1.c
oem@22D20310:~/JHALAK$ gedit exforc2.c
```

```
//exforc1.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(int argc, char *argv[])
{
printf("PID of exforc1 = %d\n", getpid());
char *args[] = ("Hello", "SY", "AI-DS", NULL);
execv("./exforc2", args);
```

```
printf("Back to ex1.c");
return 0;
}

//exforc2.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/walt.h>
int main(int argc, char *argv[])
{
printf("We are in exforc2.c\n");
printf("PID for exforc2.c %d\n", getpid());
}
```

---

---

## List Linux system calls used for process creation and management.

Linux system calls used for process creation and management:

**fork():** Creates a new process by duplicating the calling process.

**exec():** Replaces the current process image with a new process image.

**wait():** Suspends the calling process until one of its child processes terminates.

**exit():** Terminates the calling process and returns an exit status to the parent process.

**getpid():** Returns the process ID (PID) of the calling process.

**getppid():** Returns the parent process ID (PPID) of the calling process.

## Write a C program to create one parent and 3 child process.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
printf("Parent process\n");
for (int i = 0; i < 3; i++) {
if (fork() == 0) {
printf("Child process %d with PID %d\n", i + 1, getpid());
exit(0);
}}
for (int i = 0; i < 3; i++) {
wait(NULL);
}
printf("Parent process exiting\n");
return 0;
}
```

# OR

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main() {
printf("my name is jhalak\n");
```

```
fork();
fork();
fork();
printf("roll no 14 \n");
}
```

## Write a C program to demonstrate the used of wait() and exit() system call

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include <sys/types.h>
#include<sys/wait.h>
int main()
{
printf("MY NAME IS JHALAK DEDHIA\n");
pid_t pid=fork();
printf("MY ROLL NO IS 14\n");
if(pid==0)
{
printf("Child Process\n");
exit(EXIT_SUCCESS);
}
else if(pid > 0)
{
printf("Parent Process\n");
printf("Parent PROCESS_ID %d\n", getpid());
printf("waiting for child to terminate\n");
wait(NULL);
printf("Child got Terminated\n");
}
else
{
printf("Unable to create process");
}
return 0;
}
```

## Which system call is used to create the process?

The system call used to create a new process in Unix-like operating systems (such as Linux) is fork(). When you call fork(), the operating system creates a new process, which is a copy of the calling process (the parent process). After the fork() call, there are two identical processes running concurrently: the parent process and the child process.

## Write a program to create parent and child processes and get process ID.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
printf("MY NAME IS JHALAK DEDHIA\n");
pid_t pid=fork();
if(pid < 0)
{
printf("Unable to create process\n");
}
else if(pid == 0)
{
printf("child process PROCESS_ID %d and parent PROCESS_ID %d\n", getpid(), getppid());
}
else if(pid > 0)
{
printf("Parent Process PROCESS_ID %d\n", getpid());
}
return 0;
}
```

---

# What is semaphore?

Semaphore : A semaphore S is an integer variable that can be accessed only through two standard operations :

● wait() - The wait() operation reduces the value of semaphore by 1

● signal() - The signal() operation increases its value by 1.

```
wait(S){
while(S<=0);   // busy waiting
S--;
}
signal(S){
S++;
}
```

Semaphores are of two types:

● Binary Semaphore – This is similar to mutex lock but not the same thing.  It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

● Counting Semaphore – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

# Write a C program to simulate producer and consumer problems using semaphores.

```
#include<stdio.h>
#include<stdlib.h>
int full = 0;
int empty = 8;
int mutex = 1;
int x = 0;
int wait(int*s)
{
return(*s)--;
}int signal(int*s)
{
return(*s)++;
}void producer()
{
wait(&empty);
wait(&mutex);
x++;
```

```
printf("Producer Producess Item %d\n",x);
signal(&mutex);
signal(&full);
}void consumer()
{
wait(&full);
wait(&mutex);
x--;
printf("Consumer Consumes Item %d\n",x);
signal(&mutex);
signal(&empty);
}void main()
{
int ch;
printf("Name: Jhalak Dedhia\nRoll No: 14");
printf("\n Menu: \n 1. Producer() \n 2. Consumer() \n 3.Exit \n");
while(1)
{
printf("\nEnter your choice: ");
scanf("%d",&ch);
switch(ch)
{
case 1: if(mutex==1 && empty!=0){
   producer();
   }else{
   printf("Bufer is Full \n ");
   }  break;
case 2: if (mutex==1 && full!=0){
   consumer();
   }else{
   printf("Buffer is Empty \n ");
   }break;
case 3:
   exit(0);
   break;
defalut: printf("Invalid choice \n");
}}}
```

## List types of scheduling algorithms.

- First-Come, First-Served (FCFS): Processes are executed in the order they arrive. It's a non-preemptive algorithm where the CPU is allocated to the first process in the ready queue until it completes or gets blocked.
- Shortest Job Next (SJN) or Shortest Job First (SJF): The process with the shortest burst time is selected for execution. It can be either preemptive (Shortest Remaining Time Next - SRTN) or non-preemptive.
- Round Robin (RR): Each process is given a fixed time slice (time quantum) to execute on the CPU. After the time slice expires, the process is preempted, and the CPU is allocated to the next process in the ready queue. It's commonly used in time-sharing systems.
- Priority Scheduling: Processes are assigned priorities, and the CPU is allocated to the process with the highest priority. It can be preemptive (Priority Preemptive Scheduling) or non-preemptive (Priority Non-Preemptive Scheduling).

## Write a C program to simulate the CPU scheduling algorithm First Come First Serve (FCFS).

```
#include <stdio.h>
int main() {
int n, atat, q = 0;
printf("Enter number of Processes: ");
scanf("%d", &n);
int p[n], a[n], bt[n], c[n];
int d = 0;
printf("Enter process ID: ");
for (int i = 0; i < n; i++) {
scanf("%d", &p[i]);
}
printf("Enter Burst Time: ");
for (int j = 0; j < n; j++) {
scanf("%d", &bt[j]);
d = d + bt[j];
}
printf("Final ct: %d\n", d);
for (int k = 0; k < n; k++) {
a[0] = 0;
a[k + 1] = a[k] + bt[k];
c[k] = a[k + 1];
printf("\nStart time of process %d: %d", p[k], a[k]);
}
```

```
for (int l = 0; l < n; l++) {
printf("\nEnd time of process %d: %d", p[l], c[l]);
}
c[n] = d;
printf("\n");
for (int s = 0; s < n; s++) {
printf("\nWaiting time is: %d", a[s]);
printf("\nTAT of process %d: %d", p[s], c[s]);
q = q + c[s];
}
atat = q / n;
printf("\nATAT of process: %d\n", atat);
return 0;
}
```

---

# Elaborate FCFS scheduling algorithm.

The First-Come, First-Served (FCFS) scheduling algorithm is one of the simplest scheduling algorithms used in operating systems. It is a non-preemptive scheduling algorithm, meaning once a process starts execution, it continues until it completes or voluntarily gives up the CPU. In the FCFS scheduling algorithm, the job that arrived first in the ready queue is allocated to the CPU and then the job that came second, and so on. We can say that the ready queue acts as a FIFO (First In First Out) queue thus the arriving jobs/processes are placed at the end of the queue.

**How FCFS Scheduling Works:**

**Arrival Order**: Processes are executed in the order they arrive in the ready queue. The process that arrives first is executed first.

**Non-Preemptive**: Once a process starts executing, it continues until it finishes, or it performs an I/O operation and goes into the waiting state.

**Ready Queue**: Processes that are ready for execution but waiting for the CPU are placed in the ready queue.

**CPU Allocation:** The CPU is allocated to the first process in the ready queue. It continues executing that process until it completes or performs I/O.

**Completion:** When a process completes its execution (or performs I/O), the next process in the ready queue is selected for execution.

**Characteristics of FCFS Scheduling**:

**Simple**: FCFS is easy to implement and understand, making it suitable for basic scheduling scenarios.

**Fairness:** It provides fairness in the sense that every process gets a chance to execute based on its arrival time. However, it may not be optimal in terms of overall system performance.

**Non-Preemptive Nature**: Since FCFS is non-preemptive, it may not be suitable for time-critical or interactive systems where responsiveness is crucial.

**Advantages:**

- Simple to implement and understand.
- No starvation (every process eventually gets CPU time).
- Fairness in terms of arrival order.

**Disadvantages:**

- Poor performance in terms of average waiting time, especially for long processes (convoy effect).
- Not suitable for time-critical or interactive systems due to its non-preemptive nature.

## To write a c program to simulate the CPU scheduling algorithm Shortest Job First(SJF)

```
#include <stdio.h>
int main() {
int n, i, j, temp;
```

```
float avg_wt, avg_tat, total_wt = 0, total_tat = 0;
printf("Enter the number of processes: ");
scanf("%d", &n);
int burst_time[n], process_id[n], waiting_time[n], turnaround_time[n];
printf("Enter Burst Time for each process:\n");
for (i = 0; i < n; i++) {
printf("P%d: ", i + 1);
scanf("%d", &burst_time[i]);
process_id[i] = i + 1;
}for (i = 0; i < n - 1; i++) {
for (j = i + 1; j < n; j++) {
if (burst_time[i] > burst_time[j]) {
temp = burst_time[i];
burst_time[i] = burst_time[j];
burst_time[j] = temp;
temp = process_id[i];
process_id[i] = process_id[j];
process_id[j] = temp;
}}}
waiting_time[0] = 0; // Waiting time for the first process is 0
for (i = 1; i < n; i++) {
waiting_time[i] = 0;
for (j = 0; j < i; j++)
waiting_time[i] += burst_time[j];
total_wt += waiting_time[i];
}
for (i = 0; i < n; i++) {
turnaround_time[i] = burst_time[i] + waiting_time[i];
total_tat += turnaround_time[i];
}
avg_wt = total_wt / n;
avg_tat = total_tat / n;
printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (i = 0; i < n; i++) {
printf("P%d\t%d\t\t%d\t\t%d\n", process_id[i], burst_time[i], waiting_time[i],turnaround_time[i]);
}
printf("\nAverage Waiting Time = %.2f\n", avg_wt);
printf("Average Turnaround Time = %.2f\n", avg_tat);
return 0;   }
```

# What is Deadlock? How to avoid deadlock.

When two or more processes try to access the critical section at the same time and they fail to access simultaneously or stuck while accessing the critical section then this condition is known as Deadlock.

The deadlock has the following characteristics:

- Mutual Exclusion
- Hold and Wait
- No preemption
- Circular wait

**Avoiding Deadlock:**

To avoid deadlock, several strategies and techniques can be employed:

**Resource Allocation:** Use resource allocation strategies such as Banker's Algorithm that ensure processes acquire resources in a safe sequence to avoid circular waits.

**Avoid Hold and Wait:** Require processes to acquire all necessary resources before execution starts, reducing the chances of holding resources while waiting for others.

**Preemption:** Allow preemption of resources in critical situations. If a process cannot acquire a resource, it may release previously acquired resources to prevent deadlock.

**Resource Ordering**: Establish a global order for acquiring resources and require processes to follow this order to avoid circular waits.

**Timeouts:** Implement timeout mechanisms where processes release resources if they cannot acquire all necessary resources within a specified time.

**Resource Sharing**: Encourage resource sharing and minimize resource holding times to reduce the likelihood of deadlock.

**Avoid Circular Wait:** Design resource allocation and locking mechanisms to prevent circular dependencies among processes.

# Write C code to implement banker’s Algorithm.

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10
int available[MAX_RESOURCES];
int maximum [MAX_PROCESSES] [MAX_RESOURCES];
int allocation [MAX_PROCESSES] [MAX_RESOURCES];
int need [MAX_PROCESSES] [MAX_RESOURCES];
int num_processes, num_resources;
int safe_sequence [MAX_PROCESSES];
void initialize() {
printf("Enter the number of resources: ");
scanf("%d", &num_processes);
```

```
printf("Enter the available resources: \n");
scanf("%d", &num_resources);
for (int i = 0; i < num_resources; i++) {
scanf("%d", &available[i]);
}
printf("Enter the maximum resources required for each process: \n");
for (int i = 0; i < num_processes; i++) {
printf("For process %d: ", 1);
for (int j = 0; j < num_resources; j++) {
scanf("%d", &maximum[1][j]);
need[i][j] = maximum[i][j];
}}}

int is_safe() {
int work [MAX_RESOURCES];
int finish [MAX_PROCESSES] = {0};
for (int i = 0; i < num_resources; i++) {
work[i] = available[i];
}
int count = 0;
while (count < num_processes) {
int found = 0;
for (int i = 0; i < num_processes; i++){
if (!finish[1]) {
int j;
for (j=0; j < num_resources; j++) {
if (need[i][j] > work[j]) {
break;
}
}
if (j == num_resources) {
for (int k; k < num_resources; k++) {
work[k] += allocation[i][k];
}
finish[1] = 1;
safe_sequence[count++] = i;
found = 1;
}}}
if (!found) {
```

```
return 0; // unsafe state
}
}
return 1; // safe state
}
void display_need_matrix() {
printf("Need Matrix:\n");
for (int i=0; i < num_processes; i++) {
printf("Process %d: ", 1);
for (int j=0; j < num_resources; j++) {
printf("%d", need [1][j]);
}
printf("\n");
}
}
int main(){
initialize();
if (is_safe()) {
printf("Safe state\n");
display_need_matrix();
printf("Safe sequence: ");
for (int i=0;i<num_processes; i++) {
printf("%d", safe_sequence[1]);
}
printf("\n");

} else
{ printf("Unsafe state\n");
}
return 0;
}
```

---

# List types of page replacement techniques.

Page replacement techniques are used in virtual memory management systems to decide which page to evict from memory when a page fault occurs and a new page needs to be loaded. Here are some common types of page replacement techniques:

**Optimal Page Replacement (OPT):**

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Evicts the page that will not be used for the longest period of time in the future.

Provides the lowest possible page-fault rate but requires future knowledge of page accesses, which is generally not feasible.

**FIFO (First-In, First-Out):** This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue.

Evicts the oldest page in memory, based on the order of arrival into memory.

Simple and easy to implement but suffers from the "Belady's anomaly," where increasing the number of frames can lead to more page faults.

**LRU (Least Recently Used):**

Evicts the page that has not been used for the longest period of time. In this algorithm, page will be replaced which is least recently used.

# Write C code for one of the page replacement algorithms.

```
#include <stdio.h>
int findPageFaults(int pages[], int n, int numFrames) {
int frames[numFrames];
int faults =0;
int frameIndex =0;
int i, j;
for (i =0; i < numFrames; i++)
frames[i] = -1;
for (i =0; i < n; i++) {
int pageFound =0;
for (j =0; j < numFrames; j++) {
if (frames[j] == pages[i]) {
pageFound =1;
break;
}}
if (!pageFound) {
```

```
faults++;
frames[frameIndex] = pages[i];
frameIndex = (frameIndex +1) % numFrames;
}
printf("Page %d -> Frames: ", pages[i]);
for (j =0; j < numFrames; j++) {
printf("%d ", frames[j]);
}
printf("\n");
}
return faults;
}
int main() {
int n, numFrames;
printf("Enter the number of pages: ");
scanf("%d", &n);
int pages[n];
printf("Enter the pages: ");
for (int i =0; i < n; i++) {
scanf("%d", &pages[i]);
}
printf("Enter the number of frames: ");
scanf("%d", &numFrames);
int faults = findPageFaults(pages, n, numFrames);
printf("\n Total Page Faults: %d\n", faults);
int hit = n - faults;
printf("\n Total Page Hits: %d\n", hit);
return 0;
}
```

## Define memory management.

In a multiprogramming computer, the Operating System resides in a part of memory, and the rest is used by multiple processes. The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

**Why Memory Management is Required?**

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

## Write a C program to implement fixed memory partitioning. Take a example and show Internal fragmentation.

```
#include <stdio.h>
#define MAX_PARTITIONS 10 // Maximum number of partitions
int main() {
int n;
printf("Enter the total number of partitions including OS (maximum %d): ",
MAX_PARTITIONS);
scanf("%d", &n);
if (n > MAX_PARTITIONS) {
printf("Error: Number of partitions exceeds maximum allowed.\n");
return 1; // Exit with error code
}
int total [MAX_PARTITIONS], process [MAX_PARTITIONS], internal [MAX_PARTITIONS],
total_internal = 0;
int os_size;
printf("Enter the size of the operating system: ");
scanf("%d", &os_size);
for (int i = 0; i < n; i++) {
printf("Enter the size of partition %d: ", i + 1);
scanf("%d", &total[i]);
}
for (int i = 0; i < n; i++) {
printf("Enter the size of process %d: ", i + 1);
scanf("%d", &process[i]);
}
```

```
// Calculate internal fragmentation for each partition
for (int i = 0; i < n; i++) {
internal[i] = total[i] - process[i];
total_internal += internal[i];
}
// Calculate total memory used by processes
int total_used_memory = 0;
for (int i = 0; i < n; i++) {
total_used_memory += process[i];
}
// Calculate external fragmentation
int total_external_fragmentation = 0;
if (total_internal < total_used_memory) {
total_external_fragmentation = total_used_memory - total_internal;
}
printf("\nPartition\tTotal Size\tProcess Size\tInternal Fragmentation\n");
for (int i=0; i<n; i++) {
printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, total [i], process [i], internal [i]);
}
printf("\nTotal Used Memory %d\n", total_used_memory);
printf("\nTotal Internal Fragmentation: %d\n", total_internal);
printf("Total External Fragmentation: %d\n", total_external_fragmentation);
return 0;
}
```

---