**Operating System**

**AI3002**

**List of Programs for Practical ESE Exam -----------------------------------------------------------------------------------------------**

**Show the execution of Basic and Advanced Linux commands. (any 5 commands from each type)**

Basic Linux Commands

1. \*\*`pwd` (Print Working Directory)\*\*

Displays the current directory.

```bash

$ pwd

/home/user/documents

```

2. \*\*`ls` (List Files and Directories)\*\*

Lists files and directories in the current directory.

```bash

$ ls

file1.txt file2.txt directory1

```

3. \*\*`cd` (Change Directory)\*\*

Changes the working directory.

```bash

$ cd /home/user/downloads

$ pwd

/home/user/downloads

```

4. \*\*`mkdir` (Make Directory)\*\*

Creates a new directory.

```bash

$ mkdir new\_folder

$ ls

new\_folder

```

5. \*\*`rm` (Remove File)\*\*

Deletes a file.

```bash

$ rm file1.txt

$ ls

file2.txt directory1

```

---

### \*\*Advanced Linux Commands\*\*

1. \*\*`find` (Search for Files and Directories)\*\*

Searches files or directories matching specific criteria.

```bash

$ find /home/user -name "\*.txt"

/home/user/file2.txt

```

2. \*\*`grep` (Search for Patterns in Text)\*\*

Searches for a pattern in files.

```bash

$ grep "hello" file2.txt

hello world

```

3. \*\*`tar` (Archive Files)\*\*

Compresses files into an archive.

```bash

$ tar -cvf archive.tar file2.txt directory1

$ ls

archive.tar

```

4. \*\*`chmod` (Change File Permissions)\*\*

Modifies file permissions.

```bash

$ chmod 755 file2.txt

$ ls -l

-rwxr-xr-x 1 user user 0 Nov 16 10:00 file2.txt

```

5. \*\*`top` (Monitor System Processes)\*\*

Displays system processes and resource usage.

```bash

$ top

```

**Write a shell Script to find the sum of digits of the entered number.**

#!/bin/bash

# Prompt user to enter a number

echo "Enter a number:"

read number

# Initialize sum to 0

sum=0

# Loop through each digit

while [ $number -gt 0 ]; do

# Extract the last digit

digit=$((number % 10))

# Add the digit to the sum

sum=$((sum + digit))

# Remove the last digit

number=$((number / 10))

done

# Display the result

echo "The sum of the digits is: $sum"

Make it executable:  
chmod +x sum\_of\_digits.sh

Run the script:  
./sum\_of\_digits.sh

**Write a shell Script to sort array elements in Descending order.**

#!/bin/bash

# Prompt user to enter the array elements

echo "Enter the elements of the array separated by space:"

read -a array

# Get the length of the array

n=${#array[@]}

# Bubble sort to sort the array in descending order

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

for ((j = 0; j < n - i - 1; j++)); do

if [ ${array[j]} -lt ${array[j+1]} ]; then

# Swap elements

temp=${array[j]}

array[j]=${array[j+1]}

array[j+1]=$temp

fi

done

done

# Display the sorted array

echo "Array elements in descending order:"

echo "${array[@]}"

**Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the First Come and First Serve (FCFS) Scheduling. (enter suitable number of processes, CPU burst, and Arrival Time)**

**#include <iostream>**

**#include <vector>**

**#include <iomanip>**

**using namespace std;**

**// Function to calculate Turnaround Time (TAT) and Waiting Time (WT)**

**void calculateTimes(const vector<int>& processes, const vector<int>& burstTime, const vector<int>& arrivalTime, vector<int>& tat, vector<int>& wt) {**

**int n = processes.size();**

**vector<int> completionTime(n);**

**// Calculate Completion Time**

**completionTime[0] = arrivalTime[0] + burstTime[0];**

**for (int i = 1; i < n; i++) {**

**completionTime[i] = max(completionTime[i - 1], arrivalTime[i]) + burstTime[i];**

**}**

**// Calculate Turnaround Time and Waiting Time**

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

**tat[i] = completionTime[i] - arrivalTime[i];**

**wt[i] = tat[i] - burstTime[i];**

**}**

**}**

**int main() {**

**int n;**

**cout << "Enter the number of processes: ";**

**cin >> n;**

**vector<int> processes(n), burstTime(n), arrivalTime(n), tat(n), wt(n);**

**cout << "Enter Arrival Time and Burst Time for each process:\n";**

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

**cout << "Process " << i + 1 << ":\n";**

**cout << "Arrival Time: ";**

**cin >> arrivalTime[i];**

**cout << "Burst Time: ";**

**cin >> burstTime[i];**

**processes[i] = i + 1;**

**}**

**// Calculate TAT and WT**

**calculateTimes(processes, burstTime, arrivalTime, tat, wt);**

**// Display results**

**cout << "\nProcess\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n";**

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

**cout << "P" << processes[i] << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i]**

**<< "\t\t" << tat[i] << "\t\t" << wt[i] << '\n';**

**}**

**// Calculate and display average TAT and WT**

**double avgTAT = 0, avgWT = 0;**

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

**avgTAT += tat[i];**

**avgWT += wt[i];**

**}**

**avgTAT /= n;**

**avgWT /= n;**

**cout << "\nAverage Turnaround Time: " << fixed << setprecision(2) << avgTAT << endl;**

**cout << "Average Waiting Time: " << fixed << setprecision(2) << avgWT << endl;**

**return 0;**

**}**

**Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the Shortest Job First (Preemptive and Non-Preemptive) Scheduling. (enter suitable number of processes, CPU burst, and**

**Arrival Time)**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**#include <iomanip>**

**#include<climits>**

**using namespace std;**

**// Function to calculate TAT and WT for Non-Preemptive SJF**

**void nonPreemptiveSJF(vector<int> processes, vector<int> burstTime, vector<int> arrivalTime) {**

**int n = processes.size();**

**vector<int> tat(n), wt(n), completionTime(n);**

**vector<bool> visited(n, false);**

**int currentTime = 0, completed = 0;**

**while (completed < n) {**

**int shortest = -1;**

**int minBurstTime = INT\_MAX;**

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

**if (!visited[i] && arrivalTime[i] <= currentTime && burstTime[i] < minBurstTime) {**

**shortest = i;**

**minBurstTime = burstTime[i];**

**}**

**}**

**if (shortest != -1) {**

**currentTime += burstTime[shortest];**

**completionTime[shortest] = currentTime;**

**tat[shortest] = completionTime[shortest] - arrivalTime[shortest];**

**wt[shortest] = tat[shortest] - burstTime[shortest];**

**visited[shortest] = true;**

**completed++;**

**} else {**

**currentTime++;**

**}**

**}**

**cout << "\nNon-Preemptive SJF Scheduling:\n";**

**cout << "Process\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n";**

**double avgTAT = 0, avgWT = 0;**

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

**cout << "P" << processes[i] << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i]**

**<< "\t\t" << tat[i] << "\t\t" << wt[i] << '\n';**

**avgTAT += tat[i];**

**avgWT += wt[i];**

**}**

**cout << "\nAverage Turnaround Time: " << fixed << setprecision(2) << avgTAT / n << endl;**

**cout << "Average Waiting Time: " << fixed << setprecision(2) << avgWT / n << endl;**

**}**

**// Function to calculate TAT and WT for Preemptive SJF**

**void preemptiveSJF(vector<int> processes, vector<int> burstTime, vector<int> arrivalTime) {**

**int n = processes.size();**

**vector<int> remainingTime = burstTime;**

**vector<int> tat(n), wt(n), completionTime(n);**

**int currentTime = 0, completed = 0;**

**int shortest = -1;**

**while (completed < n) {**

**shortest = -1;**

**int minRemainingTime = INT\_MAX;**

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

**if (arrivalTime[i] <= currentTime && remainingTime[i] > 0 && remainingTime[i] < minRemainingTime) {**

**shortest = i;**

**minRemainingTime = remainingTime[i];**

**}**

**}**

**if (shortest != -1) {**

**currentTime++;**

**remainingTime[shortest]--;**

**if (remainingTime[shortest] == 0) {**

**completionTime[shortest] = currentTime;**

**tat[shortest] = completionTime[shortest] - arrivalTime[shortest];**

**wt[shortest] = tat[shortest] - burstTime[shortest];**

**completed++;**

**}**

**} else {**

**currentTime++;**

**}**

**}**

**cout << "\nPreemptive SJF Scheduling:\n";**

**cout << "Process\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n";**

**double avgTAT = 0, avgWT = 0;**

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

**cout << "P" << processes[i] << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i]**

**<< "\t\t" << tat[i] << "\t\t" << wt[i] << '\n';**

**avgTAT += tat[i];**

**avgWT += wt[i];**

**}**

**cout << "\nAverage Turnaround Time: " << fixed << setprecision(2) << avgTAT / n << endl;**

**cout << "Average Waiting Time: " << fixed << setprecision(2) << avgWT / n << endl;**

**}**

**// Menu-driven program**

**int main() {**

**int n;**

**cout << "Enter the number of processes: ";**

**cin >> n;**

**vector<int> processes(n), burstTime(n), arrivalTime(n);**

**cout << "Enter Arrival Time and Burst Time for each process:\n";**

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

**cout << "Process " << i + 1 << ":\n";**

**cout << "Arrival Time: ";**

**cin >> arrivalTime[i];**

**cout << "Burst Time: ";**

**cin >> burstTime[i];**

**processes[i] = i + 1;**

**}**

**int choice;**

**do {**

**cout << "\nMenu:\n";**

**cout << "1. Non-Preemptive SJF Scheduling\n";**

**cout << "2. Preemptive SJF Scheduling\n";**

**cout << "3. Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**nonPreemptiveSJF(processes, burstTime, arrivalTime);**

**break;**

**case 2:**

**preemptiveSJF(processes, burstTime, arrivalTime);**

**break;**

**case 3:**

**cout << "Exiting the program.\n";**

**break;**

**default:**

**cout << "Invalid choice. Please try again.\n";**

**}**

**} while (choice != 3);**

**return 0;**

**}**

**Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the Priority (Preemptive and Non-Preemptive) Scheduling. (enter suitable number of processes, CPU burst, and Arrival Time)**

**#include <iostream>**

**#include <vector>**

**#include <iomanip>**

**#include <algorithm>**

**#include <climits>**

**using namespace std;**

**// Function to calculate TAT and WT for Non-Preemptive Priority Scheduling**

**void nonPreemptivePriority(vector<int> processes, vector<int> burstTime, vector<int> arrivalTime, vector<int> priority) {**

**int n = processes.size();**

**vector<int> tat(n), wt(n), completionTime(n);**

**vector<bool> visited(n, false);**

**int currentTime = 0, completed = 0;**

**while (completed < n) {**

**int highestPriority = -1;**

**int maxPriority = INT\_MIN; // Start with the lowest possible value**

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

**if (!visited[i] && arrivalTime[i] <= currentTime && priority[i] > maxPriority) {**

**highestPriority = i;**

**maxPriority = priority[i];**

**}**

**}**

**if (highestPriority != -1) {**

**currentTime += burstTime[highestPriority];**

**completionTime[highestPriority] = currentTime;**

**tat[highestPriority] = completionTime[highestPriority] - arrivalTime[highestPriority];**

**wt[highestPriority] = tat[highestPriority] - burstTime[highestPriority];**

**visited[highestPriority] = true;**

**completed++;**

**} else {**

**currentTime++;**

**}**

**}**

**cout << "\nNon-Preemptive Priority Scheduling:\n";**

**cout << "Process\tArrival Time\tBurst Time\tPriority\tTurnaround Time\tWaiting Time\n";**

**double avgTAT = 0, avgWT = 0;**

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

**cout << "P" << processes[i] << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i] << "\t\t" << priority[i]**

**<< "\t\t" << tat[i] << "\t\t" << wt[i] << '\n';**

**avgTAT += tat[i];**

**avgWT += wt[i];**

**}**

**cout << "\nAverage Turnaround Time: " << fixed << setprecision(2) << avgTAT / n << endl;**

**cout << "Average Waiting Time: " << fixed << setprecision(2) << avgWT / n << endl;**

**}**

**// Function to calculate TAT and WT for Preemptive Priority Scheduling**

**void preemptivePriority(vector<int> processes, vector<int> burstTime, vector<int> arrivalTime, vector<int> priority) {**

**int n = processes.size();**

**vector<int> remainingTime = burstTime;**

**vector<int> tat(n), wt(n), completionTime(n);**

**int currentTime = 0, completed = 0;**

**int highestPriority = -1;**

**while (completed < n) {**

**highestPriority = -1;**

**int maxPriority = INT\_MIN; // Start with the lowest possible value**

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

**if (arrivalTime[i] <= currentTime && remainingTime[i] > 0 && priority[i] > maxPriority) {**

**highestPriority = i;**

**maxPriority = priority[i];**

**}**

**}**

**if (highestPriority != -1) {**

**currentTime++;**

**remainingTime[highestPriority]--;**

**if (remainingTime[highestPriority] == 0) {**

**completionTime[highestPriority] = currentTime;**

**tat[highestPriority] = completionTime[highestPriority] - arrivalTime[highestPriority];**

**wt[highestPriority] = tat[highestPriority] - burstTime[highestPriority];**

**completed++;**

**}**

**} else {**

**currentTime++;**

**}**

**}**

**cout << "\nPreemptive Priority Scheduling:\n";**

**cout << "Process\tArrival Time\tBurst Time\tPriority\tTurnaround Time\tWaiting Time\n";**

**double avgTAT = 0, avgWT = 0;**

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

**cout << "P" << processes[i] << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i] << "\t\t" << priority[i]**

**<< "\t\t" << tat[i] << "\t\t" << wt[i] << '\n';**

**avgTAT += tat[i];**

**avgWT += wt[i];**

**}**

**cout << "\nAverage Turnaround Time: " << fixed << setprecision(2) << avgTAT / n << endl;**

**cout << "Average Waiting Time: " << fixed << setprecision(2) << avgWT / n << endl;**

**}**

**// Menu-driven program**

**int main() {**

**int n;**

**cout << "Enter the number of processes: ";**

**cin >> n;**

**vector<int> processes(n), burstTime(n), arrivalTime(n), priority(n);**

**cout << "Enter Arrival Time, Burst Time, and Priority for each process:\n";**

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

**cout << "Process " << i + 1 << ":\n";**

**cout << "Arrival Time: ";**

**cin >> arrivalTime[i];**

**cout << "Burst Time: ";**

**cin >> burstTime[i];**

**cout << "Priority (higher number = higher priority): ";**

**cin >> priority[i];**

**processes[i] = i + 1;**

**}**

**int choice;**

**do {**

**cout << "\nMenu:\n";**

**cout << "1. Non-Preemptive Priority Scheduling\n";**

**cout << "2. Preemptive Priority Scheduling\n";**

**cout << "3. Exit\n";**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**nonPreemptivePriority(processes, burstTime, arrivalTime, priority);**

**break;**

**case 2:**

**preemptivePriority(processes, burstTime, arrivalTime, priority);**

**break;**

**case 3:**

**cout << "Exiting the program.\n";**

**break;**

**default:**

**cout << "Invalid choice. Please try again.\n";**

**}**

**} while (choice != 3);**

**return 0;**

**}**

**Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the Round Robin Scheduling. (enter suitable number of processes, CPU burst, and Arrival Time)**

**#include <iostream>**

**#include <vector>**

**#include <iomanip>**

**#include <queue>**

**using namespace std;**

**// Function to calculate Turnaround Time and Waiting Time using Round Robin Scheduling**

**void roundRobin(vector<int> processes, vector<int> burstTime, vector<int> arrivalTime, int timeQuantum) {**

**int n = processes.size();**

**vector<int> remainingTime = burstTime;**

**vector<int> completionTime(n), tat(n), wt(n);**

**queue<int> readyQueue;**

**int currentTime = 0;**

**int completed = 0;**

**vector<bool> visited(n, false);**

**// Add processes to ready queue as per their arrival time**

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

**if (arrivalTime[i] <= currentTime && !visited[i]) {**

**readyQueue.push(i);**

**visited[i] = true;**

**}**

**}**

**while (!readyQueue.empty() || completed < n) {**

**if (!readyQueue.empty()) {**

**int currentProcess = readyQueue.front();**

**readyQueue.pop();**

**if (remainingTime[currentProcess] > 0) {**

**int timeToRun = min(remainingTime[currentProcess], timeQuantum);**

**currentTime += timeToRun;**

**remainingTime[currentProcess] -= timeToRun;**

**// If the process is completed**

**if (remainingTime[currentProcess] == 0) {**

**completionTime[currentProcess] = currentTime;**

**tat[currentProcess] = completionTime[currentProcess] - arrivalTime[currentProcess];**

**wt[currentProcess] = tat[currentProcess] - burstTime[currentProcess];**

**completed++;**

**}**

**}**

**// Add newly arrived processes to the queue**

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

**if (arrivalTime[i] <= currentTime && !visited[i]) {**

**readyQueue.push(i);**

**visited[i] = true;**

**}**

**}**

**// If the process is not completed, add it back to the ready queue**

**if (remainingTime[currentProcess] > 0) {**

**readyQueue.push(currentProcess);**

**}**

**} else {**

**currentTime++; // No process is ready, move time forward**

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

**if (arrivalTime[i] <= currentTime && !visited[i]) {**

**readyQueue.push(i);**

**visited[i] = true;**

**}**

**}**

**}**

**}**

**// Output results**

**cout << "\nRound Robin Scheduling:\n";**

**cout << "Process\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n";**

**double avgTAT = 0, avgWT = 0;**

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

**cout << "P" << processes[i] << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i]**

**<< "\t\t" << tat[i] << "\t\t" << wt[i] << '\n';**

**avgTAT += tat[i];**

**avgWT += wt[i];**

**}**

**cout << "\nAverage Turnaround Time: " << fixed << setprecision(2) << avgTAT / n << endl;**

**cout << "Average Waiting Time: " << fixed << setprecision(2) << avgWT / n << endl;**

**}**

**// Main function**

**int main() {**

**int n, timeQuantum;**

**cout << "Enter the number of processes: ";**

**cin >> n;**

**vector<int> processes(n), burstTime(n), arrivalTime(n);**

**cout << "Enter Arrival Time and Burst Time for each process:\n";**

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

**cout << "Process " << i + 1 << ":\n";**

**cout << "Arrival Time: ";**

**cin >> arrivalTime[i];**

**cout << "Burst Time: ";**

**cin >> burstTime[i];**

**processes[i] = i + 1;**

**}**

**cout << "Enter Time Quantum: ";**

**cin >> timeQuantum;**

**roundRobin(processes, burstTime, arrivalTime, timeQuantum);**

**return 0;**

**}**

**Write a program to demonstrate any 5 system calls.**

**#include <iostream>**

**#include <unistd.h> // For fork, getpid, getppid**

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

**#include <sys/stat.h>**

**#include <fcntl.h> // For open**

**#include <cstring> // For strlen**

**#include <cstdlib> // For exit**

**#include <sys/wait.h> // For wait**

**using namespace std;**

**int main() {**

**cout << "Demonstrating 5 system calls:\n";**

**// 1. Fork system call**

**cout << "\n1. Fork System Call:\n";**

**pid\_t pid = fork();**

**if (pid < 0) {**

**cerr << "Fork failed!" << endl;**

**exit(1);**

**} else if (pid == 0) {**

**// Child process**

**cout << "In Child Process (PID: " << getpid() << ", Parent PID: " << getppid() << ")\n";**

**exit(0);**

**} else {**

**// Parent process**

**cout << "In Parent Process (PID: " << getpid() << "), Child PID: " << pid << endl;**

**wait(NULL); // Wait for the child process to complete**

**cout << "Child process completed.\n";**

**}**

**// 2. getpid system call**

**cout << "\n2. getpid and getppid System Calls:\n";**

**cout << "Current Process ID: " << getpid() << endl;**

**cout << "Parent Process ID: " << getppid() << endl;**

**// 3. open and write system calls**

**cout << "\n3. open and write System Calls:\n";**

**const char \*filename = "example.txt";**

**int fd = open(filename, O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR);**

**if (fd < 0) {**

**cerr << "Failed to open file!" << endl;**

**exit(1);**

**}**

**const char \*message = "Hello, this is a system call demonstration.\n";**

**write(fd, message, strlen(message)); // Write to the file**

**cout << "Data written to file: " << filename << endl;**

**close(fd); // Close the file**

**// 4. read system call**

**cout << "\n4. read System Call:\n";**

**fd = open(filename, O\_RDONLY);**

**if (fd < 0) {**

**cerr << "Failed to open file for reading!" << endl;**

**exit(1);**

**}**

**char buffer[1024];**

**int bytesRead = read(fd, buffer, sizeof(buffer) - 1); // Read file content**

**if (bytesRead < 0) {**

**cerr << "Failed to read file!" << endl;**

**exit(1);**

**}**

**buffer[bytesRead] = '\0'; // Null-terminate the string**

**cout << "Contents of the file:\n" << buffer << endl;**

**close(fd);**

**// 5. execvp system call**

**cout << "\n5. execvp System Call:\n";**

**pid = fork(); // Create another child process**

**if (pid == 0) {**

**// In child process**

**char \*args[] = {(char \*)"ls", (char \*)"-l", NULL};**

**execvp("ls", args); // Execute the `ls -l` command**

**cerr << "Exec failed!" << endl;**

**exit(1);**

**} else if (pid > 0) {**

**wait(NULL); // Parent waits for the child**

**cout << "Execvp executed successfully.\n";**

**} else {**

**cerr << "Fork failed for execvp demonstration!" << endl;**

**}**

**return 0;**

**}**

**Write a program to Implement multithreading for Matrix Operations using Pthreads (any one operation).**

**#include <iostream>**

**#include <pthread.h>**

**#include <vector>**

**using namespace std;**

**// Matrices and dimensions**

**vector<vector<int>> matA, matB, matC;**

**int rowsA, colsA, rowsB, colsB;**

**// Structure to hold row and column for each thread**

**struct ThreadData {**

**int row;**

**int col;**

**};**

**// Thread function to compute a single element in the result matrix**

**void\* multiplyElement(void\* arg) {**

**ThreadData\* data = (ThreadData\*)arg;**

**int row = data->row;**

**int col = data->col;**

**matC[row][col] = 0;**

**for (int i = 0; i < colsA; i++) {**

**matC[row][col] += matA[row][i] \* matB[i][col];**

**}**

**pthread\_exit(0);**

**}**

**int main() {**

**// Input dimensions of Matrix A**

**cout << "Enter the dimensions of Matrix A (rows and columns): ";**

**cin >> rowsA >> colsA;**

**// Input dimensions of Matrix B**

**cout << "Enter the dimensions of Matrix B (rows and columns): ";**

**cin >> rowsB >> colsB;**

**// Check if multiplication is possible**

**if (colsA != rowsB) {**

**cerr << "Error: Number of columns in Matrix A must equal number of rows in Matrix B for multiplication.\n";**

**return 1;**

**}**

**// Initialize matrices**

**matA.resize(rowsA, vector<int>(colsA));**

**matB.resize(rowsB, vector<int>(colsB));**

**matC.resize(rowsA, vector<int>(colsB));**

**// Input elements for Matrix A**

**cout << "Enter elements of Matrix A:\n";**

**for (int i = 0; i < rowsA; i++) {**

**for (int j = 0; j < colsA; j++) {**

**cin >> matA[i][j];**

**}**

**}**

**// Input elements for Matrix B**

**cout << "Enter elements of Matrix B:\n";**

**for (int i = 0; i < rowsB; i++) {**

**for (int j = 0; j < colsB; j++) {**

**cin >> matB[i][j];**

**}**

**}**

**// Create threads**

**vector<vector<pthread\_t>> threads(rowsA, vector<pthread\_t>(colsB));**

**vector<vector<ThreadData>> threadData(rowsA, vector<ThreadData>(colsB));**

**// Create a thread for each element in the result matrix**

**for (int i = 0; i < rowsA; i++) {**

**for (int j = 0; j < colsB; j++) {**

**threadData[i][j] = {i, j};**

**pthread\_create(&threads[i][j], nullptr, multiplyElement, &threadData[i][j]);**

**}**

**}**

**// Join all threads**

**for (int i = 0; i < rowsA; i++) {**

**for (int j = 0; j < colsB; j++) {**

**pthread\_join(threads[i][j], nullptr);**

**}**

**}**

**// Display the result matrix**

**cout << "\nResultant Matrix (C = A \* B):\n";**

**for (int i = 0; i < rowsA; i++) {**

**for (int j = 0; j < colsB; j++) {**

**cout << matC[i][j] << " ";**

**}**

**cout << endl;**

**}**

**return 0;**

**}**

**Write a program to check whether a given system is in a safe state or not using Banker’s Deadlock Avoidance algorithm (assume suitable data).**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**// Function to check if the system is in a safe state**

**bool isSafeState(vector<vector<int>>& allocation, vector<vector<int>>& max, vector<int>& available, int resources) {**

**int processes = allocation.size();**

**// Calculate the Need matrix**

**vector<vector<int>> need(processes, vector<int>(resources));**

**for (int i = 0; i < processes; i++) {**

**for (int j = 0; j < resources; j++) {**

**need[i][j] = max[i][j] - allocation[i][j];**

**}**

**}**

**// Initialize work and finish arrays**

**vector<int> work = available;**

**vector<bool> finish(processes, false);**

**vector<int> safeSequence;**

**// Keep checking until all processes are either finished or deadlocked**

**int finishedProcesses = 0;**

**while (finishedProcesses < processes) {**

**bool found = false;**

**for (int i = 0; i < processes; i++) {**

**if (!finish[i]) {**

**// Check if process can execute**

**bool canExecute = true;**

**for (int j = 0; j < resources; j++) {**

**if (need[i][j] > work[j]) {**

**canExecute = false;**

**break;**

**}**

**}**

**// If process can execute, mark it as finished and update work**

**if (canExecute) {**

**for (int j = 0; j < resources; j++) {**

**work[j] += allocation[i][j];**

**}**

**finish[i] = true;**

**safeSequence.push\_back(i);**

**finishedProcesses++;**

**found = true;**

**}**

**}**

**}**

**// If no process can execute in this round, the system is not in a safe state**

**if (!found) {**

**cout << "System is not in a safe state.\n";**

**return false;**

**}**

**}**

**// Print the safe sequence**

**cout << "System is in a safe state.\nSafe Sequence: ";**

**for (int i : safeSequence) {**

**cout << "P" << i + 1 << " ";**

**}**

**cout << endl;**

**return true;**

**}**

**int main() {**

**int processes, resources;**

**// Input number of processes and resources**

**cout << "Enter the number of processes: ";**

**cin >> processes;**

**cout << "Enter the number of resources: ";**

**cin >> resources;**

**// Input total available resources**

**vector<int> totalAvailable(resources);**

**cout << "Enter the total number of each type of resource: ";**

**for (int i = 0; i < resources; i++) {**

**cin >> totalAvailable[i];**

**}**

**// Input Allocation matrix**

**vector<vector<int>> allocation(processes, vector<int>(resources));**

**cout << "Enter the Allocation matrix (resources allocated to each process):\n";**

**for (int i = 0; i < processes; i++) {**

**for (int j = 0; j < resources; j++) {**

**cin >> allocation[i][j];**

**}**

**}**

**// Input Maximum Need matrix**

**vector<vector<int>> max(processes, vector<int>(resources));**

**cout << "Enter the Maximum Need matrix (maximum resources each process may require):\n";**

**for (int i = 0; i < processes; i++) {**

**for (int j = 0; j < resources; j++) {**

**cin >> max[i][j];**

**}**

**}**

**// Calculate the Available resources vector**

**vector<int> available(resources);**

**for (int i = 0; i < resources; i++) {**

**int allocated = 0;**

**for (int j = 0; j < processes; j++) {**

**allocated += allocation[j][i];**

**}**

**available[i] = totalAvailable[i] - allocated;**

**}**

**// Check if the system is in a safe state**

**isSafeState(allocation, max, available, resources);**

**return 0;**

**}**

**Write a program to calculate the number of page faults for a reference string (input any suitable reference string) using First In First Out (FIFO) page replacement algorithms.**

**#include <iostream>**

**#include <vector>**

**#include <queue>**

**#include <unordered\_set>**

**using namespace std;**

**// Function to calculate page faults using FIFO**

**int calculatePageFaultsFIFO(vector<int> referenceString, int frameSize) {**

**unordered\_set<int> pages; // To store the pages in memory**

**queue<int> fifoQueue; // To maintain the order of pages**

**int pageFaults = 0;**

**for (int page : referenceString) {**

**// If the page is not in memory**

**if (pages.find(page) == pages.end()) {**

**// If there is space in memory**

**if (pages.size() < frameSize) {**

**pages.insert(page);**

**fifoQueue.push(page);**

**} else {**

**// Replace the oldest page using FIFO**

**int oldestPage = fifoQueue.front();**

**fifoQueue.pop();**

**pages.erase(oldestPage);**

**pages.insert(page);**

**fifoQueue.push(page);**

**}**

**pageFaults++; // Increment page fault count**

**}**

**}**

**return pageFaults;**

**}**

**// Main function**

**int main() {**

**int n, frameSize;**

**cout << "Enter the number of pages in the reference string: ";**

**cin >> n;**

**vector<int> referenceString(n);**

**cout << "Enter the reference string (space-separated integers): ";**

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

**cin >> referenceString[i];**

**}**

**cout << "Enter the frame size: ";**

**cin >> frameSize;**

**int pageFaults = calculatePageFaultsFIFO(referenceString, frameSize);**

**cout << "\nNumber of page faults using FIFO: " << pageFaults << endl;**

**return 0;**

**}**

**Write a program to calculate the number of page faults for a reference string (input any suitable reference string) using the Least Recently Used (LRU) page replacement algorithms.**

**#include <iostream>**

**#include <vector>**

**#include <unordered\_map>**

**#include<climits>**

**using namespace std;**

**// Function to calculate page faults using LRU**

**int calculatePageFaultsLRU(vector<int> referenceString, int frameSize) {**

**unordered\_map<int, int> pageMap; // To store page and its last used time**

**int pageFaults = 0;**

**int currentTime = 0;**

**for (int page : referenceString) {**

**// Check if the page is in memory**

**if (pageMap.find(page) == pageMap.end()) {**

**// Page not in memory, so page fault occurs**

**if (pageMap.size() < frameSize) {**

**// If there is space in memory, add the page**

**pageMap[page] = currentTime;**

**} else {**

**// Find the least recently used page**

**int lruPage = -1, minTime = INT\_MAX;**

**for (auto &entry : pageMap) {**

**if (entry.second < minTime) {**

**minTime = entry.second;**

**lruPage = entry.first;**

**}**

**}**

**// Remove the least recently used page**

**pageMap.erase(lruPage);**

**// Add the new page**

**pageMap[page] = currentTime;**

**}**

**pageFaults++; // Increment page fault counter**

**} else {**

**// If page is already in memory, update its last used time**

**pageMap[page] = currentTime;**

**}**

**currentTime++;**

**}**

**return pageFaults;**

**}**

**// Main function**

**int main() {**

**int n, frameSize;**

**cout << "Enter the number of pages in the reference string: ";**

**cin >> n;**

**vector<int> referenceString(n);**

**cout << "Enter the reference string (space-separated integers): ";**

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

**cin >> referenceString[i];**

**}**

**cout << "Enter the frame size: ";**

**cin >> frameSize;**

**int pageFaults = calculatePageFaultsLRU(referenceString, frameSize);**

**cout << "\nNumber of page faults using LRU: " << pageFaults << endl;**

**return 0;**

**}**

**Write a program to calculate the number of page faults for a reference string (input any suitable reference**

**string) using the Optimal page replacement algorithms.**

**#include <iostream>**

**#include <vector>**

**#include <unordered\_set>**

**#include <unordered\_map>**

**using namespace std;**

**// Function to calculate page faults using Optimal Page Replacement**

**int calculatePageFaultsOptimal(vector<int> referenceString, int frameSize) {**

**unordered\_set<int> pages; // To store the pages in memory**

**int pageFaults = 0;**

**for (int i = 0; i < referenceString.size(); i++) {**

**int page = referenceString[i];**

**// If the page is not already in memory**

**if (pages.find(page) == pages.end()) {**

**// Page fault occurs**

**if (pages.size() < frameSize) {**

**// If there is space in memory, add the page**

**pages.insert(page);**

**} else {**

**// If no space, replace a page using the Optimal strategy**

**int farthest = i, pageToReplace = -1;**

**for (int p : pages) {**

**bool found = false;**

**for (int j = i + 1; j < referenceString.size(); j++) {**

**if (referenceString[j] == p) {**

**if (j > farthest) {**

**farthest = j;**

**pageToReplace = p;**

**}**

**found = true;**

**break;**

**}**

**}**

**if (!found) { // If page is not used in the future**

**pageToReplace = p;**

**break;**

**}**

**}**

**// Remove the page to be replaced and add the new page**

**pages.erase(pageToReplace);**

**pages.insert(page);**

**}**

**pageFaults++;**

**}**

**}**

**return pageFaults;**

**}**

**// Main function**

**int main() {**

**int n, frameSize;**

**cout << "Enter the number of pages in the reference string: ";**

**cin >> n;**

**vector<int> referenceString(n);**

**cout << "Enter the reference string (space-separated integers): ";**

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

**cin >> referenceString[i];**

**}**

**cout << "Enter the frame size: ";**

**cin >> frameSize;**

**int pageFaults = calculatePageFaultsOptimal(referenceString, frameSize);**

**cout << "\nNumber of page faults using Optimal Page Replacement: " << pageFaults << endl;**

**return 0;**

**}**