1. Any shell scripting program.

#!/bin/bash

# Function for if-else statement

if\_else\_demo() {

read -p "Enter your age: " age

if (( age < 18 )); then

echo "You are a minor."

elif (( age <= 65 )); then

echo "You are an adult."

else

echo "You are a senior."

fi

}

# Function for for loop to print numbers from start to end

for\_loop\_demo() {

read -p "Enter the start number: " start

read -p "Enter the end number: " end

for (( i=start; i<=end; i++ )); do

echo "Number $i"

done

}

# Function to check if a number is prime

is\_prime() {

local num=$1

if (( num <= 1 )); then

return 1

fi

for (( i=2; i\*i<=num; i++ )); do

if (( num % i == 0 )); then

return 1

fi

done

return 0

}

# Function for while loop demonstration (prime numbers)

while\_loop\_demo() {

read -p "Enter a start number: " start

read -p "Enter an end number: " end

while (( start <= end )); do

if is\_prime $start; then

echo "Prime number: $start"

fi

((start++))

done

}

# Function to calculate factorial of a number

factorial() {

local num=$1

local factorial=1

local i=1

until (( i > num )); do

factorial=$(( factorial \* i ))

((i++))

done

echo $factorial

}

# Function for until loop demonstration (factorial)

until\_loop\_demo() {

read -p "Enter a number to calculate factorial: " num

result=$(factorial $num)

echo "Factorial of $num is $result"

}

# Display the menu

while true; do

echo "Menu:"

echo "1) If-Else Statement"

echo "2) For Loop"

echo "3) While Loop"

echo "4) Until Loop"

echo "5) Exit"

read -p "Choose an option: " choice

case $choice in

1)

if\_else\_demo

;;

2)

for\_loop\_demo

;;

3)

while\_loop\_demo

;;

4)

until\_loop\_demo

;;

5)

echo "Exiting..."

break

;;

\*)

echo "Invalid option, please try again."

;;

esac

done

2. Write a program demonstrating use of different system calls.

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <stdlib.h>

#include <string.h>

void process\_related() {

int choice;

pid\_t pid;

printf("\nProcess Related System Calls:\n");

printf("1. fork()\n");

printf("2. exit()\n");

printf("3. wait()\n");

printf("4. exec()\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

pid = fork();

if (pid == 0) {

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

exit(0);

} else {

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

}

break;

case 2:

printf("Exiting with status 0...\n");

exit(0);

break;

case 3:

pid = fork();

if (pid == 0) {

printf("Child process. PID = %d\n", getpid());

exit(0);

} else {

wait(NULL);

printf("Child process finished. Parent PID = %d\n", getpid());

}

break;

case 4:

pid = fork();

if (pid == 0) {

printf("Child process executing ls command:\n");

execl("/bin/ls", "ls", NULL);

exit(0);

} else {

wait(NULL);

printf("ls command executed by child process.\n");

}

break;

default:

printf("Invalid choice.\n");

break;

}

}

void communication\_related() {

int choice;

int fd[2];

char write\_msg[20] = "Hello, World!";

char read\_msg[20];

pid\_t pid;

printf("\nCommunication Related System Calls:\n");

printf("1. pipe()\n");

printf("2. FIFO (named pipe)\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

if (pipe(fd) == -1) {

perror("Pipe failed");

exit(1);

}

pid = fork();

if (pid == 0) {

close(fd[0]); // Close unused read end

write(fd[1], write\_msg, strlen(write\_msg)+1);

close(fd[1]);

} else {

close(fd[1]); // Close unused write end

read(fd[0], read\_msg, sizeof(read\_msg));

printf("Received message: %s\n", read\_msg);

close(fd[0]);

}

break;

case 2:

mkfifo("/tmp/myfifo", 0666);

pid = fork();

if (pid == 0) {

int fd = open("/tmp/myfifo", O\_WRONLY);

write(fd, write\_msg, strlen(write\_msg)+1);

close(fd);

exit(0);

} else {

int fd = open("/tmp/myfifo", O\_RDONLY);

read(fd, read\_msg, sizeof(read\_msg));

printf("Received message: %s\n", read\_msg);

close(fd);

}

unlink("/tmp/myfifo");

break;

default:

printf("Invalid choice.\n");

break;

}

}

void file\_related() {

int choice;

int fd;

char buffer[100];

printf("\nFile Related System Calls:\n");

printf("1. open(), write(), close()\n");

printf("2. link(), stat(), unlink()\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

fd = open("example.txt", O\_WRONLY | O\_CREAT, 0644);

if (fd == -1) {

perror("Error opening file");

exit(1);

}

write(fd, "Hello, World!\n", 14);

close(fd);

printf("File written and closed successfully.\n");

fd = open("example.txt", O\_RDONLY);

read(fd, buffer, 100);

printf("File content: %s", buffer);

close(fd);

break;

case 2:

link("example.txt", "example\_link.txt");

struct stat file\_stat;

stat("example\_link.txt", &file\_stat);

printf("File size: %ld bytes\n", file\_stat.st\_size);

unlink("example\_link.txt");

printf("Link removed.\n");

break;

default:

printf("Invalid choice.\n");

break;

}

}

void info\_related() {

int choice;

printf("\nInformation Related System Calls:\n");

printf("1. alarm()\n");

printf("2. sleep()\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

printf("Setting an alarm for 5 seconds...\n");

alarm(5);

sleep(6); // To show the alarm trigger

break;

case 2:

printf("Sleeping for 3 seconds...\n");

sleep(3);

printf("Woke up after 3 seconds.\n");

break;

default:

printf("Invalid choice.\n");

break;

}

}

int main() {

int choice;

while(1) {

printf("\nMenu:\n");

printf("1. Process Related\n");

printf("2. Communication Related\n");

printf("3. File Related\n");

printf("4. Information Related\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch(choice) {

case 1:

process\_related();

break;

case 2:

communication\_related();

break;

case 3:

file\_related();

break;

case 4:

info\_related();

break;

case 5:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice.\n");

break;

}

}

return 0;

}

3. Implement multithreading for Matrix Operations using Pthreads

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

// Define the dimensions of the matrices

#define M 3

#define N 3

#define P 3

// Matrices

int A[M][N];

int B[N][P];

int C[M][P];

// Structure to pass data to threads

typedef struct {

int row;

int col;

} thread\_data\_t;

// Thread function to perform matrix multiplication for a specific element

void\* multiply(void\* arg) {

thread\_data\_t\* data = (thread\_data\_t\*) arg;

int row = data->row;

int col = data->col;

C[row][col] = 0;

// Perform the matrix multiplication for a single element

for (int k = 0; k < N; k++) {

C[row][col] += A[row][k] \* B[k][col];

}

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[M \* P];

thread\_data\_t thread\_data[M \* P];

// Initialize matrices A and B

printf("Matrix A:\n");

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

for (int j = 0; j < N; j++) {

A[i][j] = rand() % 10;

printf("%d ", A[i][j]);

}

printf("\n");

}

printf("\nMatrix B:\n");

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

for (int j = 0; j < P; j++) {

B[i][j] = rand() % 10; // Random numbers between 0 and 9

printf("%d ", B[i][j]);

}

printf("\n");

}

int thread\_idx = 0;

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

for (int j = 0; j < P; j++) {

thread\_data[thread\_idx].row = i;

thread\_data[thread\_idx].col = j;

pthread\_create(&threads[thread\_idx], NULL, multiply, (void\*)&thread\_data[thread\_idx]);

thread\_idx++;

}

}

// Join all threads

for (int i = 0; i < M \* P; i++) {

pthread\_join(threads[i], NULL);

}

// Display the result matrix C

printf("\nResultant Matrix C (A \* B):\n");

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

for (int j = 0; j < P; j++) {

printf("%d ", C[i][j]);

}

printf("\n");

}

return 0;

}

.

4. Implementation of Classical problems (reader writer) using Threads and Mutex

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

// Shared data and synchronization variables

int shared\_data = 0;

int reader\_count = 0;

pthread\_mutex\_t resource\_mutex; // Mutex for exclusive access to shared data

pthread\_mutex\_t reader\_count\_mutex; // Mutex for protecting reader count

int max\_operations = 5; // Number of iterations for readers and writers

// Reader function

void\* reader(void\* arg) {

int reader\_id = \*((int\*)arg);

free(arg);

int read\_count = 0;

while (read\_count < max\_operations) {

// Entry section for reader

pthread\_mutex\_lock(&reader\_count\_mutex);

reader\_count++;

if (reader\_count == 1) {

pthread\_mutex\_lock(&resource\_mutex); // First reader locks the resource

}

pthread\_mutex\_unlock(&reader\_count\_mutex);

// Reading section

printf("Reader %d is reading the shared data: %d\n", reader\_id, shared\_data);

// Exit section for reader

pthread\_mutex\_lock(&reader\_count\_mutex);

reader\_count--;

if (reader\_count == 0) {

pthread\_mutex\_unlock(&resource\_mutex); // Last reader unlocks the resource

}

pthread\_mutex\_unlock(&reader\_count\_mutex);

read\_count++; // Increment read count

}

return NULL;

}

// Writer function

void\* writer(void\* arg) {

int writer\_id = \*((int\*)arg);

free(arg);

int write\_count = 0;

while (write\_count < max\_operations) {

pthread\_mutex\_lock(&resource\_mutex); // Writer locks the resource

shared\_data++;

printf("Writer %d updated shared data to: %d\n", writer\_id, shared\_data);

pthread\_mutex\_unlock(&resource\_mutex); // Writer releases the resource

write\_count++; // Increment write count

}

return NULL;

}

int main() {

pthread\_t readers[5], writers[2];

pthread\_mutex\_init(&resource\_mutex, NULL);

pthread\_mutex\_init(&reader\_count\_mutex, NULL);

// Create reader threads

for (int i = 0; i < 5; i++) {

int\* id = malloc(sizeof(int));

\*id = i + 1;

pthread\_create(&readers[i], NULL, reader, id);

}

// Create writer threads

for (int i = 0; i < 2; i++) {

int\* id = malloc(sizeof(int));

\*id = i + 1;

pthread\_create(&writers[i], NULL, writer, id);

}

// Wait for threads to complete

for (int i = 0; i < 5; i++) {

pthread\_join(readers[i], NULL);

}

for (int i = 0; i < 2; i++) {

pthread\_join(writers[i], NULL);

}

// Destroy mutexes

pthread\_mutex\_destroy(&resource\_mutex);

pthread\_mutex\_destroy(&reader\_count\_mutex);

return 0;

}

5. Implementation of Classical problems( producer consumer) using Threads and Mutex

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int count = 0;

pthread\_mutex\_t mutex;

pthread\_cond\_t not\_full, not\_empty;

void\* producer(void\* arg) {

int item;

for (int i = 0; i < 10; i++) {

item = rand() % 100;

pthread\_mutex\_lock(&mutex);

while (count == BUFFER\_SIZE)

pthread\_cond\_wait(&not\_full, &mutex);

buffer[count++] = item;

printf("Producer produced: %d\n", item);

pthread\_cond\_signal(&not\_empty);

pthread\_mutex\_unlock(&mutex);

// Removed sleep(1); // No sleep for faster execution

}

return NULL;

}

void\* consumer(void\* arg) {

int item;

for (int i = 0; i < 10; i++) {

pthread\_mutex\_lock(&mutex);

while (count == 0)

pthread\_cond\_wait(&not\_empty, &mutex);

item = buffer[--count];

printf("Consumer consumed: %d\n", item);

pthread\_cond\_signal(&not\_full);

pthread\_mutex\_unlock(&mutex);

// Removed sleep(1); // No sleep for faster execution

}

return NULL;

}

int main() {

pthread\_t prod\_thread, cons\_thread;

pthread\_mutex\_init(&mutex, NULL);

pthread\_cond\_init(&not\_full, NULL);

pthread\_cond\_init(&not\_empty, NULL);

pthread\_create(&prod\_thread, NULL, producer, NULL);

pthread\_create(&cons\_thread, NULL, consumer, NULL);

pthread\_join(prod\_thread, NULL);

pthread\_join(cons\_thread, NULL);

pthread\_mutex\_destroy(&mutex);

pthread\_cond\_destroy(&not\_full);

pthread\_cond\_destroy(&not\_empty);

return 0;

}

6. Implementation of Classical problems (reader writer) using Threads and Semaphore. .(reader writer, producer consumer, dining philosopher)

uu

7. Implementation of Classical problems (producer consumer,) using Threads and Semaphore.

import java.util.LinkedList;

import java.util.concurrent.Semaphore;

class ProducerConsumer {

private final LinkedList<Integer> buffer = new LinkedList<>();

private final int capacity = 5; // Size of the buffer

private final Semaphore mutex; // Mutual exclusion for buffer access

private final Semaphore empty; // Semaphore to count empty slots in buffer

private final Semaphore full; // Semaphore to count full slots in buffer

public ProducerConsumer() {

mutex = new Semaphore(1); // Binary semaphore for mutual exclusion

empty = new Semaphore(capacity); // Counting semaphore for empty slots

full = new Semaphore(0); // Counting semaphore for full slots

}

// Producer class

class Producer implements Runnable {

@Override

public void run() {

int item = 0; // Item to be produced

while (true) {

try {

empty.acquire(); // Wait if buffer is full

mutex.acquire(); // Protect buffer access

// Critical Section for Producer (Inserting item)

buffer.add(item);

System.out.println("Produced: " + item);

item++; // Increment item to produce new one

mutex.release(); // Release buffer access

full.release(); // Signal that there's a full slot now

// Simulate production time

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e.getMessage());

}

}

}

}

// Consumer class

class Consumer implements Runnable {

@Override

public void run() {

while (true) {

try {

full.acquire(); // Wait if buffer is empty

mutex.acquire(); // Protect buffer access

// Critical Section for Consumer (Removing item)

int item = buffer.removeFirst();

System.out.println("Consumed: " + item);

mutex.release(); // Release buffer access

empty.release(); // Signal that there's an empty slot now

// Simulate consumption time

Thread.sleep(1500);

} catch (InterruptedException e) {

System.out.println(e.getMessage());

}

}

}

}

public static void main(String[] args) {

ProducerConsumer pc = new ProducerConsumer();

// Creating producer and consumer threads

Thread producer1 = new Thread(pc.new Producer());

Thread producer2 = new Thread(pc.new Producer());

Thread producer3 = new Thread(pc.new Producer());

Thread consumer1 = new Thread(pc.new Consumer());

Thread consumer2 = new Thread(pc.new Consumer());

// Start threads

producer1.start();

producer2.start();

producer3.start();

consumer1.start();

consumer2.start();

}

}

8. Implementation of Classical problems (dining philosopher) using Threads and Semaphore.

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

#define N 5

sem\_t forks[N];

void \*philosopher(void \*arg) {

int id = \*((int \*)arg);

for (int i = 0; i < 3; i++) {

printf("Philosopher %d is thinking.\n", id);

usleep(100000);

sem\_wait(&forks[id]);

sem\_wait(&forks[(id + 1) % N]);

printf("Philosopher %d is eating.\n", id);

usleep(100000);

sem\_post(&forks[id]);

sem\_post(&forks[(id + 1) % N]);

printf("Philosopher %d finished eating.\n", id);

usleep(100000);

}

return NULL;

}

int main() {

pthread\_t philosophers[N];

int ids[N];

for (int i = 0; i < N; i++)

sem\_init(&forks[i], 0, 1);

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

ids[i] = i;

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

for (int i = 0; i < N; i++)

pthread\_join(philosophers[i], NULL);

for (int i = 0; i < N; i++)

sem\_destroy(&forks[i]);

return 0;

}

9. Write a program to compute the finish time, turnaround time and waiting time for the First come First serve

#include <stdio.h>

struct Process {

int pid; // Process ID

int arrivalTime;

int burstTime;

int finishTime;

int turnAroundTime;

int waitingTime;

};

void calculateTimes(struct Process processes[], int n) {

int currentTime = 0;

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

// If the process arrives after the current time, update the current time to the arrival time of the process

if (currentTime < processes[i].arrivalTime) {

currentTime = processes[i].arrivalTime;

}

// Finish time is the current time plus the burst time

processes[i].finishTime = currentTime + processes[i].burstTime;

// Turnaround time is finish time minus arrival time

processes[i].turnAroundTime = processes[i].finishTime - processes[i].arrivalTime;

// Waiting time is turnaround time minus burst time

processes[i].waitingTime = processes[i].turnAroundTime - processes[i].burstTime;

// Update current time to finish time of the current process

currentTime = processes[i].finishTime;

}

}

void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");

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

printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",

processes[i].pid,

processes[i].arrivalTime,

processes[i].burstTime,

processes[i].finishTime,

processes[i].turnAroundTime,

processes[i].waitingTime);

}

float totalTurnAroundTime = 0, totalWaitingTime = 0;

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

totalTurnAroundTime += processes[i].turnAroundTime;

totalWaitingTime += processes[i].waitingTime;

}

printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);

printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

processes[i].pid = i + 1;

printf("Enter arrival time and burst time for process %d: ", processes[i].pid);

scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);

}

// Sort processes by arrival time (FCFS scheduling)

for (int i = 0; i < n - 1; i++) {

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

if (processes[i].arrivalTime > processes[j].arrivalTime) {

struct Process temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

calculateTimes(processes, n);

displayResults(processes, n);

return 0;

}

10. Write a program to compute the finish time, turnaround time and waiting time for the

Shortest Job First (Preemptive and Non Preemptive)

#include <stdio.h>

#include <stdbool.h>

struct Process {

int pid; // Process ID

int arrivalTime;

int burstTime;

int remainingTime; // For preemptive SJF

int finishTime;

int turnAroundTime;

int waitingTime;

bool isCompleted;

};

// Function for Non-Preemptive SJF

void sjfNonPreemptive(struct Process processes[], int n) {

int currentTime = 0, completed = 0;

while (completed < n) {

int minIndex = -1;

int minBurstTime = 1e9;

// Select the process with the smallest burst time that has arrived

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

if (!processes[i].isCompleted && processes[i].arrivalTime <= currentTime &&

processes[i].burstTime < minBurstTime) {

minBurstTime = processes[i].burstTime;

minIndex = i;

}

}

if (minIndex == -1) {

currentTime++;

} else {

// Calculate the finish time, turnaround time, and waiting time

processes[minIndex].finishTime = currentTime + processes[minIndex].burstTime;

processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;

processes[minIndex].waitingTime = processes[minIndex].turnAroundTime - processes[minIndex].burstTime;

processes[minIndex].isCompleted = true;

currentTime = processes[minIndex].finishTime;

completed++;

}

}

}

// Function for Preemptive SJF

void sjfPreemptive(struct Process processes[], int n) {

int currentTime = 0, completed = 0;

int minIndex = -1;

int minRemainingTime = 1e9;

while (completed < n) {

minIndex = -1;

minRemainingTime = 1e9;

// Select the process with the smallest remaining time that has arrived

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

if (processes[i].arrivalTime <= currentTime && !processes[i].isCompleted &&

processes[i].remainingTime < minRemainingTime) {

minRemainingTime = processes[i].remainingTime;

minIndex = i;

}

}

if (minIndex != -1) {

processes[minIndex].remainingTime--;

currentTime++;

// If process is completed

if (processes[minIndex].remainingTime == 0) {

processes[minIndex].finishTime = currentTime;

processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;

processes[minIndex].waitingTime = processes[minIndex].turnAroundTime - processes[minIndex].burstTime;

processes[minIndex].isCompleted = true;

completed++;

}

} else {

currentTime++;

}

}

}

void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");

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

printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",

processes[i].pid,

processes[i].arrivalTime,

processes[i].burstTime,

processes[i].finishTime,

processes[i].turnAroundTime,

processes[i].waitingTime);

}

float totalTurnAroundTime = 0, totalWaitingTime = 0;

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

totalTurnAroundTime += processes[i].turnAroundTime;

totalWaitingTime += processes[i].waitingTime;

}

printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);

printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

int n, choice;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

processes[i].pid = i + 1;

printf("Enter arrival time and burst time for process %d: ", processes[i].pid);

scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);

processes[i].remainingTime = processes[i].burstTime;

processes[i].isCompleted = false;

}

printf("Choose Scheduling:\n1. Non-Preemptive SJF\n2. Preemptive SJF\n");

scanf("%d", &choice);

if (choice == 1) {

sjfNonPreemptive(processes, n);

} else if (choice == 2) {

sjfPreemptive(processes, n);

} else {

printf("Invalid choice!\n");

return 0;

}

displayResults(processes, n);

return 0;

}

11. Write a program to compute the finish time, turnaround time and waiting time for the

Priority (Preemptive and Non Preemptive)

#include <stdio.h>

#include <stdbool.h>

struct Process {

int pid; // Process ID

int arrivalTime;

int burstTime;

int remainingTime; // For preemptive scheduling

int priority;

int finishTime;

int turnAroundTime;

int waitingTime;

bool isCompleted;

};

// Function for Non-Preemptive Priority Scheduling

void priorityNonPreemptive(struct Process processes[], int n) {

int currentTime = 0, completed = 0;

while (completed < n) {

int minIndex = -1;

int highestPriority = 1e9;

// Select the process with the highest priority that has arrived

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

if (!processes[i].isCompleted && processes[i].arrivalTime <= currentTime &&

processes[i].priority < highestPriority) {

highestPriority = processes[i].priority;

minIndex = i;

}

}

if (minIndex == -1) {

currentTime++;

} else {

// Calculate the finish time, turnaround time, and waiting time

processes[minIndex].finishTime = currentTime + processes[minIndex].burstTime;

processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;

processes[minIndex].waitingTime = processes[minIndex].turnAroundTime - processes[minIndex].burstTime;

processes[minIndex].isCompleted = true;

currentTime = processes[minIndex].finishTime;

completed++;

}

}

}

// Function for Preemptive Priority Scheduling

void priorityPreemptive(struct Process processes[], int n) {

int currentTime = 0, completed = 0;

while (completed < n) {

int minIndex = -1;

int highestPriority = 1e9;

// Select the process with the highest priority that has arrived

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

if (processes[i].arrivalTime <= currentTime && !processes[i].isCompleted &&

processes[i].priority < highestPriority) {

highestPriority = processes[i].priority;

minIndex = i;

}

}

if (minIndex != -1) {

// Process one unit of the burst time

processes[minIndex].remainingTime--;

currentTime++;

// If process is completed

if (processes[minIndex].remainingTime == 0) {

processes[minIndex].finishTime = currentTime;

processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;

processes[minIndex].waitingTime = processes[minIndex].turnAroundTime - processes[minIndex].burstTime;

processes[minIndex].isCompleted = true;

completed++;

}

} else {

currentTime++;

}

}

}

void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tPriority\tFinish\tTurnaround\tWaiting\n");

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

printf("%d\t%d\t%d\t%d\t\t%d\t%d\t\t%d\n",

processes[i].pid,

processes[i].arrivalTime,

processes[i].burstTime,

processes[i].priority,

processes[i].finishTime,

processes[i].turnAroundTime,

processes[i].waitingTime);

}

float totalTurnAroundTime = 0, totalWaitingTime = 0;

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

totalTurnAroundTime += processes[i].turnAroundTime;

totalWaitingTime += processes[i].waitingTime;

}

printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);

printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

int n, choice;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

processes[i].pid = i + 1;

printf("Enter arrival time, burst time, and priority for process %d: ", processes[i].pid);

scanf("%d %d %d", &processes[i].arrivalTime, &processes[i].burstTime, &processes[i].priority);

processes[i].remainingTime = processes[i].burstTime;

processes[i].isCompleted = false;

}

printf("Choose Scheduling:\n1. Non-Preemptive Priority\n2. Preemptive Priority\n");

scanf("%d", &choice);

if (choice == 1) {

priorityNonPreemptive(processes, n);

} else if (choice == 2) {

priorityPreemptive(processes, n);

} else {

printf("Invalid choice!\n");

return 0;

}

displayResults(processes, n);

return 0;

}

12. Write a program to compute the finish time, turnaround time and waiting time for the

Round robin

#include <stdio.h>

struct Process {

int pid; // Process ID

int arrivalTime; // Arrival time

int burstTime; // Burst time

int remainingTime; // Remaining burst time

int finishTime; // Finish time

int turnAroundTime; // Turnaround time

int waitingTime; // Waiting time

};

void roundRobin(struct Process processes[], int n, int quantum) {

int currentTime = 0;

int completed = 0;

int timeQuantum = quantum;

while (completed < n) {

int done = 1;

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

// Check if process has remaining time and has arrived

if (processes[i].remainingTime > 0 && processes[i].arrivalTime <= currentTime) {

done = 0;

// If remaining time is less than or equal to time quantum, process will finish

if (processes[i].remainingTime <= timeQuantum) {

currentTime += processes[i].remainingTime;

processes[i].finishTime = currentTime;

processes[i].turnAroundTime = processes[i].finishTime - processes[i].arrivalTime;

processes[i].waitingTime = processes[i].turnAroundTime - processes[i].burstTime;

processes[i].remainingTime = 0;

completed++;

} else {

// Process runs for the time quantum

processes[i].remainingTime -= timeQuantum;

currentTime += timeQuantum;

}

}

}

// If all processes are done

if (done) {

currentTime++;

}

}

}

void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");

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

printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",

processes[i].pid,

processes[i].arrivalTime,

processes[i].burstTime,

processes[i].finishTime,

processes[i].turnAroundTime,

processes[i].waitingTime);

}

float totalTurnAroundTime = 0, totalWaitingTime = 0;

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

totalTurnAroundTime += processes[i].turnAroundTime;

totalWaitingTime += processes[i].waitingTime;

}

printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);

printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);

}

int main() {

int n, quantum;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

processes[i].pid = i + 1;

printf("Enter arrival time and burst time for process %d: ", processes[i].pid);

scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);

processes[i].remainingTime = processes[i].burstTime;

}

printf("Enter the time quantum: ");

scanf("%d", &quantum);

roundRobin(processes, n, quantum);

displayResults(processes, n);

return 0;

}

13. Write a program to check whether given system is in safe state or not using Banker’s Deadlock Avoidance algorithm.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int processes, resources;

int available[MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

void calculateNeed() {

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

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

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

}

}

}

bool isSafeState() {

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES] = {false};

int safeSequence[MAX\_PROCESSES];

int count = 0;

// Initialize work as a copy of available resources

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

work[i] = available[i];

}

while (count < processes) {

bool found = false;

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

if (!finish[i]) {

bool canAllocate = true;

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

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

canAllocate = false;

break;

}

}

if (canAllocate) {

for (int k = 0; k < resources; k++) {

work[k] += allocation[i][k];

}

safeSequence[count++] = i;

finish[i] = true;

found = true;

}

}

}

if (!found) {

printf("System is not in a safe state.\n");

return false;

}

}

printf("System is in a safe state.\nSafe sequence is: ");

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

printf("P%d ", safeSequence[i]);

}

printf("\n");

return true;

}

int main() {

printf("Enter the number of processes: ");

scanf("%d", &processes);

printf("Enter the number of resources: ");

scanf("%d", &resources);

printf("Enter the available resources:\n");

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

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

}

printf("Enter the maximum resource matrix:\n");

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

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

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

}

}

printf("Enter the allocation matrix:\n");

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

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

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

}

}

calculateNeed();

isSafeState();

return 0;

}

14. Write a program for Deadlock detection algorithm

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int processes, resources;

int available[MAX\_RESOURCES];

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int request[MAX\_PROCESSES][MAX\_RESOURCES];

void deadlockDetection() {

bool finish[MAX\_PROCESSES] = {false};

int work[MAX\_RESOURCES];

// Initialize work as a copy of available resources

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

work[i] = available[i];

}

bool deadlock = false;

int deadlockedProcesses[MAX\_PROCESSES];

int deadlockedCount = 0;

for (int count = 0; count < processes; count++) {

bool found = false;

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

if (!finish[i]) {

bool canProceed = true;

// Check if the process's request can be satisfied

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

if (request[i][j] > work[j]) {

canProceed = false;

break;

}

}

// If the request can be satisfied, allocate resources temporarily

if (canProceed) {

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

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

}

finish[i] = true;

found = true;

}

}

}

// If no process could proceed in this round, break out

if (!found) {

break;

}

}

// Check for processes still marked as unfinished

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

if (!finish[i]) {

deadlockedProcesses[deadlockedCount++] = i;

deadlock = true;

}

}

if (deadlock) {

printf("System is in a deadlock state.\n");

printf("Deadlocked processes: ");

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

printf("P%d ", deadlockedProcesses[i]);

}

printf("\n");

} else {

printf("System is not in a deadlock state.\n");

}

}

int main() {

printf("Enter the number of processes: ");

scanf("%d", &processes);

printf("Enter the number of resources: ");

scanf("%d", &resources);

printf("Enter the available resources:\n");

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

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

}

printf("Enter the allocation matrix:\n");

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

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

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

}

}

printf("Enter the request matrix:\n");

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

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

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

}

}

deadlockDetection();

return 0;

}

15. Write a program to calculate the number of page faults for a reference string for the FIFO page replacement algorithms:

#include <stdio.h>

#define MAX\_FRAMES 10

int isPageInFrames(int frames[], int frameCount, int page) {

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

if (frames[i] == page) {

return 1; // Page found in frames

}

}

return 0; // Page not found

}

int main() {

int frameCount, pageCount;

int pageFaults = 0;

int nextFrameToReplace = 0; // To keep track of which frame to replace next

// Input: Number of frames and number of pages in the reference string

printf("Enter the number of frames: ");

scanf("%d", &frameCount);

printf("Enter the number of pages in the reference string: ");

scanf("%d", &pageCount);

int pages[pageCount];

printf("Enter the reference string (space-separated): ");

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

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

}

int frames[MAX\_FRAMES];

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

frames[i] = -1; // Initialize frames as empty

}

// Processing each page in the reference string

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

int currentPage = pages[i];

// Check if the current page is already in the frames

if (!isPageInFrames(frames, frameCount, currentPage)) {

// Page fault occurs as the page is not in frames

frames[nextFrameToReplace] = currentPage; // Replace the page at nextFrameToReplace

pageFaults++; // Increment page faults

nextFrameToReplace = (nextFrameToReplace + 1) % frameCount; // Move to the next frame

}

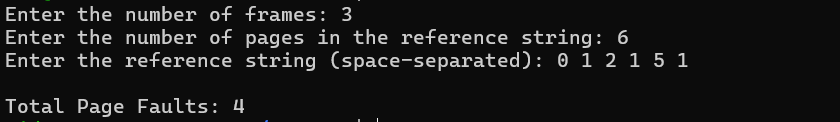
}

printf("\nTotal Page Faults: %d\n", pageFaults);

return 0;

}

Output:-



16. Write a program to calculate the number of page faults for a reference string for the LRU page replacement algorithms:

#include <stdio.h>

#define MAX\_FRAMES 10

int findLRU(int frames[], int time[], int frameCount) {

int min = time[0], minIndex = 0;

for (int i = 1; i < frameCount; i++) {

if (time[i] < min) {

min = time[i];

minIndex = i;

}

}

return minIndex; // Return the index of the LRU page

}

int isPageInFrames(int frames[], int frameCount, int page) {

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

if (frames[i] == page) {

return 1; // Page found in frames

}

}

return 0; // Page not found

}

int main() {

int frameCount, pageCount;

int pageFaults = 0;

// Input: Number of frames and number of pages in the reference string

printf("Enter the number of frames: ");

scanf("%d", &frameCount);

printf("Enter the number of pages in the reference string: ");

scanf("%d", &pageCount);

int pages[pageCount];

printf("Enter the reference string (space-separated): ");

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

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

}

int frames[MAX\_FRAMES];

int time[MAX\_FRAMES]; // Array to keep track of the last used time of each frame

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

frames[i] = -1; // Initialize frames as empty

time[i] = 0; // Initialize the last used time

}

// Processing each page in the reference string

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

int currentPage = pages[i];

// Check if the current page is already in the frames

if (!isPageInFrames(frames, frameCount, currentPage)) {

// Page fault occurs as the page is not in frames

int lruIndex = findLRU(frames, time, frameCount); // Find the index of the LRU page

frames[lruIndex] = currentPage; // Replace the LRU page with the current page

pageFaults++; // Increment page faults

}

// Update the time of the current page

for (int j = 0; j < frameCount; j++) {

if (frames[j] == currentPage) {

time[j] = i; // Update the last used time for the current page

break;

}

}

}

printf("\nTotal Page Faults: %d\n", pageFaults);

return 0;

}

17. Write a program to calculate the number of page faults for a reference string for the Optimal page replacement algorithms:

#include <stdio.h>

#define MAX\_FRAMES 10

int findOptimal(int frames[], int frameCount, int pages[], int pageCount, int currentIndex) {

int farthest = currentIndex, indexToReplace = -1;

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

int j;

for (j = currentIndex; j < pageCount; j++) {

if (frames[i] == pages[j]) {

if (j > farthest) {

farthest = j;

indexToReplace = i;

}

break;

}

}

// If the frame is never going to be used again

if (j == pageCount) {

return i; // Replace this frame

}

}

// If all pages are used in the future, replace the one that is used the farthest in the future

return (indexToReplace != -1) ? indexToReplace : 0;

}

int isPageInFrames(int frames[], int frameCount, int page) {

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

if (frames[i] == page) {

return 1; // Page found in frames

}

}

return 0; // Page not found

}

int main() {

int frameCount, pageCount;

int pageFaults = 0;

// Input: Number of frames and number of pages in the reference string

printf("Enter the number of frames: ");

scanf("%d", &frameCount);

printf("Enter the number of pages in the reference string: ");

scanf("%d", &pageCount);

int pages[pageCount];

printf("Enter the reference string (space-separated): ");

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

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

}

int frames[MAX\_FRAMES];

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

frames[i] = -1; // Initialize frames as empty

}

// Processing each page in the reference string

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

int currentPage = pages[i];

// Check if the current page is already in the frames

if (!isPageInFrames(frames, frameCount, currentPage)) {

// Page fault occurs as the page is not in frames

int indexToReplace = findOptimal(frames, frameCount, pages, pageCount, i); // Find the optimal frame to replace

frames[indexToReplace] = currentPage; // Replace the optimal frame with the current page

pageFaults++; // Increment page faults

}

}

printf("\nTotal Page Faults: %d\n", pageFaults);

return 0;

}

18. Write a program to simulate FCFS disk scheduling. Calculate total seek time.Print accepted input and output in tabular format

#include <stdio.h>

#include <stdlib.h>

void FCFS(int head, int requests[], int n) {

int total\_seek\_time = 0;

int current\_position = head;

printf("\nFCFS Disk Scheduling\n");

printf("Initial Head Position: %d\n", head);

printf("\nRequest\tCurrent Head Position\tSeek Time\n");

printf("-------\t----------------------\t---------\n");

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

int seek\_time = abs(requests[i] - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[i], current\_position, seek\_time);

current\_position = requests[i];

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, head;

// Input number of requests

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

// Input the disk requests

printf("Enter the disk requests:\n");

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

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

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

}

// Input initial head position

printf("Enter the initial head position: ");

scanf("%d", &head);

// Call the FCFS function to calculate seek time

FCFS(head, requests, n);

return 0;

}

19. Write a program to simulate SSTF disk scheduling. Calculate total seek time.Print accepted input and output in tabular format

#include <stdio.h>

#include <stdlib.h>

void SSTF(int head, int requests[], int n) {

int total\_seek\_time = 0;

int completed[n]; // Array to track completed requests

for (int i = 0; i < n; i++) completed[i] = 0; // Mark all requests as incomplete initially

printf("\nSSTF Disk Scheduling\n");

printf("Initial Head Position: %d\n", head);

printf("\nRequest\tCurrent Head Position\tSeek Time\n");

printf("-------\t----------------------\t---------\n");

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

int closest = -1;

int min\_seek\_time = 9999;

// Find the closest request

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

if (!completed[j]) { // Only consider uncompleted requests

int seek\_time = abs(requests[j] - head);

if (seek\_time < min\_seek\_time) {

min\_seek\_time = seek\_time;

closest = j;

}

}

}

// Move the head to the closest request

int seek\_time = abs(requests[closest] - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[closest], head, seek\_time);

head = requests[closest];

completed[closest] = 1; // Mark this request as completed

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, head;

// Input number of requests

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

// Input the disk requests

printf("Enter the disk requests:\n");

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

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

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

}

// Input initial head position

printf("Enter the initial head position: ");

scanf("%d", &head);

// Call the SSTF function to calculate seek time

SSTF(head, requests, n);

return 0;

}

20. Write a program to simulate SCAN disk scheduling. Calculate total seek time.Print accepted input and output in tabular format

#include <stdio.h>

#include <stdlib.h>

void SCAN(int head, int requests[], int n, int disk\_size, int direction) {

int total\_seek\_time = 0;

int temp, i, j;

// Sort the request array

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

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

if (requests[j] > requests[j + 1]) {

temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// Find the position of the head in the sorted request array

int pos = 0;

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

if (head < requests[i]) {

pos = i;

break;

}

}

printf("\nSCAN Disk Scheduling\n");

printf("Initial Head Position: %d\n", head);

printf("Direction: %s\n", direction == 1 ? "Right" : "Left");

printf("\nRequest\tCurrent Head Position\tSeek Time\n");

printf("-------\t----------------------\t---------\n");

// Serve requests based on the direction

if (direction == 1) { // Moving towards the right

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

int seek\_time = abs(requests[i] - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[i], head, seek\_time);

head = requests[i];

}

// Go to the end of the disk if necessary

if (head != disk\_size - 1) {

int seek\_time = abs((disk\_size - 1) - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", disk\_size - 1, head, seek\_time);

head = disk\_size - 1;

}

// Reverse direction and continue serving requests to the left

for (i = pos - 1; i >= 0; i--) {

int seek\_time = abs(requests[i] - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[i], head, seek\_time);

head = requests[i];

}

} else { // Moving towards the left

for (i = pos - 1; i >= 0; i--) {

int seek\_time = abs(requests[i] - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[i], head, seek\_time);

head = requests[i];

}

// Go to the start of the disk if necessary

if (head != 0) {

int seek\_time = abs(head - 0);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", 0, head, seek\_time);

head = 0;

}

// Reverse direction and continue serving requests to the right

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

int seek\_time = abs(requests[i] - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[i], head, seek\_time);

head = requests[i];

}

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, head, disk\_size, direction;

// Input number of requests

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

// Input the disk requests

printf("Enter the disk requests:\n");

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

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

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

}

// Input initial head position

printf("Enter the initial head position: ");

scanf("%d", &head);

// Input the size of the disk

printf("Enter the disk size: ");

scanf("%d", &disk\_size);

// Input direction (1 for right, 0 for left)

printf("Enter the initial direction (1 for right, 0 for left): ");

scanf("%d", &direction);

// Call the SCAN function to calculate seek time

SCAN(head, requests, n, disk\_size, direction);

return 0;

}

21. Write a program to simulate C-SCAN disk scheduling. Calculate total seek time.Print accepted input and output in tabular format

#include <stdio.h>

#include <stdlib.h>

void CSCAN(int head, int requests[], int n, int disk\_size) {

int total\_seek\_time = 0;

int temp, i, j;

// Sort the request array

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

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

if (requests[j] > requests[j + 1]) {

temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// Find the position of the head in the sorted request array

int pos = 0;

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

if (head < requests[i]) {

pos = i;

break;

}

}

printf("\nC-SCAN Disk Scheduling\n");

printf("Initial Head Position: %d\n", head);

printf("\nRequest\tCurrent Head Position\tSeek Time\n");

printf("-------\t----------------------\t---------\n");

// Move towards the end of the disk serving requests on the right

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

int seek\_time = abs(requests[i] - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[i], head, seek\_time);

head = requests[i];

}

// Jump to the beginning of the disk

if (head != disk\_size - 1) {

int seek\_time = abs((disk\_size - 1) - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", disk\_size - 1, head, seek\_time);

head = 0; // Jump to 0 (beginning of the disk)

printf("%d\t\t%d\t\t\t%d\n", 0, disk\_size - 1, head);

}

// Continue serving requests from the start to the head position

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

int seek\_time = abs(requests[i] - head);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t\t%d\n", requests[i], head, seek\_time);

head = requests[i];

}

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int n, head, disk\_size;

// Input number of requests

printf("Enter the number of disk requests: ");

scanf("%d", &n);

int requests[n];

// Input the disk requests

printf("Enter the disk requests:\n");

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

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

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

}

// Input initial head position

printf("Enter the initial head position: ");

scanf("%d", &head);

// Input the size of the disk

printf("Enter the disk size: ");

scanf("%d", &disk\_size);

// Call the CSCAN function to calculate seek time

CSCAN(head, requests, n, disk\_size);

return 0;

}

22. Write a program for following 1) zombie process 2),orphan processes 3)sum of even numbers of an array in parent and odd numbers of an array in child process

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

void create\_zombie\_process() {

pid\_t pid = fork();

if (pid > 0) {

printf("Parent process (PID: %d) created a zombie process (PID: %d)\n", getpid(), pid);

sleep(5); // Wait to keep the child as a zombie for a while

} else if (pid == 0) {

printf("Child process (PID: %d) exiting to become a zombie\n", getpid());

exit(0); // Child exits immediately, making it a zombie

} else {

perror("Fork failed for zombie process");

}

}

void create\_orphan\_process() {

pid\_t pid = fork();

if (pid > 0) {

printf("Parent process (PID: %d) exiting, creating an orphan process\n", getpid());

exit(0); // Parent exits, making the child an orphan

} else if (pid == 0) {

sleep(5); // Keep the child alive to observe orphan status

printf("Orphan child process (PID: %d) now adopted by init process (PPID: %d)\n", getpid(), getppid());

} else {

perror("Fork failed for orphan process");

}

}

void calculate\_even\_odd\_sum(int arr[], int n) {

pid\_t pid = fork();

if (pid > 0) { // Parent process calculates the sum of even numbers

int even\_sum = 0;

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

if (arr[i] % 2 == 0) {

even\_sum += arr[i];

}

}

wait(NULL); // Wait for child to finish

printf("Parent process (PID: %d) - Sum of even numbers: %d\n", getpid(), even\_sum);

} else if (pid == 0) { // Child process calculates the sum of odd numbers

int odd\_sum = 0;

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

if (arr[i] % 2 != 0) {

odd\_sum += arr[i];

}

}

printf("Child process (PID: %d) - Sum of odd numbers: %d\n", getpid(), odd\_sum);

exit(0); // Exit child process

} else {

perror("Fork failed for even-odd sum calculation");

}

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int n = sizeof(arr) / sizeof(arr[0]);

// Task 1: Create a zombie process

create\_zombie\_process();

sleep(6); // Wait for zombie to be cleared

// Task 2: Create an orphan process

create\_orphan\_process();

sleep(6); // Wait to observe orphan process

// Task 3: Calculate sum of even and odd numbers using parent and child process

calculate\_even\_odd\_sum(arr, n);

return 0;

}

23. Write a shell script to perform following operations on student database.  
 a) Insert b) Delete c)Update d)Search

**#!/bin/bash**

**# File to store student records**

**DB\_FILE="students.txt"**

**# Function to display menu**

**show\_menu() {**

**echo "Choose an operation:"**

**echo "1) Insert"**

**echo "2) Delete"**

**echo "3) Update"**

**echo "4) Search"**

**echo "5) Exit"**

**}**

**# Function to insert a new student record**

**insert\_student() {**

**read -p "Enter Student ID: " id**

**read -p "Enter Student Name: " name**

**read -p "Enter Student Age: " age**

**read -p "Enter Student Grade: " grade**

**echo "$id|$name|$age|$grade" >> $DB\_FILE**

**echo "Student record inserted successfully."**

**}**

**# Function to delete a student record by ID**

**delete\_student() {**

**read -p "Enter Student ID to delete: " id**

**grep -v "^$id|" $DB\_FILE > temp.txt && mv temp.txt $DB\_FILE**

**echo "Student record deleted if ID was found."**

**}**

**# Function to update a student record by ID**

**update\_student() {**

**read -p "Enter Student ID to update: " id**

**if grep -q "^$id|" $DB\_FILE; then**

**grep -v "^$id|" $DB\_FILE > temp.txt**

**mv temp.txt $DB\_FILE**

**read -p "Enter new Name: " name**

**read -p "Enter new Age: " age**

**read -p "Enter new Grade: " grade**

**echo "$id|$name|$age|$grade" >> $DB\_FILE**

**echo "Student record updated successfully."**

**else**

**echo "Student ID not found."**

**fi**

**}**

**# Function to search for a student record by ID**

**search\_student() {**

**read -p "Enter Student ID to search: " id**

**grep "^$id|" $DB\_FILE && echo "Record found." || echo "Record not found."**

**}**

**# Main script loop**

**while true; do**

**show\_menu**

**read -p "Enter your choice: " choice**

**case $choice in**

**1) insert\_student ;;**

**2) delete\_student ;;**

**3) update\_student ;;**

**4) search\_student ;;**

**5) exit 0 ;;**

**\*) echo "Invalid choice, please try again." ;;**

**esac**

**done**

24. Write a program to read and copy the contents of file character by character, line by line.

**#!/bin/bash**

**# Input and output file names**

**read -p "Enter the source file name: " source\_file**

**read -p "Enter the destination file name: " destination\_file**

**# Check if the source file exists**

**if [ ! -f "$source\_file" ]; then**

**echo "Source file does not exist."**

**exit 1**

**fi**

**# Copy contents character by character**

**echo "Copying contents character by character..."**

**while IFS= read -r -n1 char; do**

**echo -n "$char" >> "$destination\_file"**

**done < "$source\_file"**

**# Adding a newline after character copy**

**echo "" >> "$destination\_file"**

**# Copy contents line by line**

**echo "Copying contents line by line..."**

**while IFS= read -r line; do**

**echo "$line" >> "$destination\_file"**

**done < "$source\_file"**

**echo "Contents copied successfully to $destination\_file."**

25. Write a program to load ALP program from input file to main memory.

**#!/bin/bash**

**# Simulated main memory file**

**MEMORY\_FILE="main\_memory.txt"**

**# Input ALP file**

**read -p "Enter the ALP input file name: " input\_file**

**# Check if the input ALP file exists**

**if [ ! -f "$input\_file" ]; then**

**echo "Input file does not exist."**

**exit 1**

**fi**

**# Clear the memory file before loading new content**

**> "$MEMORY\_FILE"**

**# Load ALP into simulated main memory**

**echo "Loading ALP program into main memory..."**

**line\_number=1**

**while IFS= read -r line; do**

**echo "Loading line $line\_number: $line"**

**echo "$line" >> "$MEMORY\_FILE"**

**((line\_number++))**

**done < "$input\_file"**

**echo "ALP program loaded into main memory (stored in $MEMORY\_FILE)."**

MOV AX, 1

ADD BX, AX

INT 21h

26. Write a program to check Opcode error in a given job and raise an interrupt.

**#!/bin/bash**

**# List of valid opcodes (example list)**

**VALID\_OPCODES=("MOV" "ADD" "SUB" "MUL" "DIV" "INT" "JMP" "CMP" "CALL" "RET")**

**# Input ALP file**

**read -p "Enter the ALP input file name: " input\_file**

**# Check if the input ALP file exists**

**if [ ! -f "$input\_file" ]; then**

**echo "Input file does not exist."**

**exit 1**

**fi**

**# Function to check opcode validity**

**check\_opcode() {**

**local opcode=$1**

**for valid\_opcode in "${VALID\_OPCODES[@]}"; do**

**if [[ "$opcode" == "$valid\_opcode" ]]; then**

**return 0 # Opcode is valid**

**fi**

**done**

**return 1 # Opcode is invalid**

**}**

**# Reading the input file line by line to check for opcode errors**

**echo "Checking for opcode errors..."**

**line\_number=1**

**error\_found=false**

**while IFS= read -r line; do**

**# Extract the first word (opcode) from the line**

**opcode=$(echo "$line" | awk '{print $1}')**

**# Check if the extracted opcode is valid**

**if ! check\_opcode "$opcode"; then**

**echo "Error: Invalid opcode '$opcode' found on line $line\_number."**

**error\_found=true**

**# Simulate raising an interrupt by printing a message**

**echo "Interrupt raised due to opcode error."**

**break**

**fi**

**((line\_number++))**

**done < "$input\_file"**

**if [ "$error\_found" = false ]; then**

**echo "No opcode errors found. Job is valid."**

**fi**

27. Write a program to check Oprand error in a given job and raise an interrupt.

**#!/bin/bash**

**# List of valid opcodes and expected operand count (example list)**

**declare -A VALID\_OPCODES**

**VALID\_OPCODES["MOV"]=2**

**VALID\_OPCODES["ADD"]=2**

**VALID\_OPCODES["SUB"]=2**

**VALID\_OPCODES["MUL"]=2**

**VALID\_OPCODES["DIV"]=2**

**VALID\_OPCODES["INT"]=1**

**VALID\_OPCODES["JMP"]=1**

**VALID\_OPCODES["CMP"]=2**

**VALID\_OPCODES["CALL"]=1**

**VALID\_OPCODES["RET"]=0**

**# Input ALP file**

**read -p "Enter the ALP input file name: " input\_file**

**# Check if the input ALP file exists**

**if [ ! -f "$input\_file" ]; then**

**echo "Input file does not exist."**

**exit 1**

**fi**

**# Function to check operand validity**

**check\_operand() {**

**local opcode=$1**

**local operand\_count=$2**

**if [[ -v VALID\_OPCODES["$opcode"] ]]; then**

**if [ "${VALID\_OPCODES["$opcode"]}" -ne "$operand\_count" ]; then**

**return 1 # Operand count is incorrect**

**fi**

**else**

**return 2 # Invalid opcode**

**fi**

**return 0 # Operand is valid**

**}**

**# Reading the input file line by line to check for operand errors**

**echo "Checking for operand errors..."**

**line\_number=1**

**error\_found=false**

**while IFS= read -r line; do**

**# Extract opcode and operands from the line**

**opcode=$(echo "$line" | awk '{print $1}')**

**operands=$(echo "$line" | cut -d' ' -f2-)**

**operand\_count=$(echo "$operands" | tr ',' ' ' | wc -w)**

**# Check if the opcode is valid and if the operand count is correct**

**if ! check\_operand "$opcode" "$operand\_count"; then**

**if [ $? -eq 1 ]; then**

**echo "Error: Operand count mismatch for opcode '$opcode' on line $line\_number."**

**error\_found=true**

**# Simulate raising an interrupt by printing a message**

**echo "Interrupt raised due to operand error."**

**break**

**elif [ $? -eq 2 ]; then**

**echo "Error: Invalid opcode '$opcode' found on line $line\_number."**

**error\_found=true**

**echo "Interrupt raised due to invalid opcode."**

**break**

**fi**

**fi**

**((line\_number++))**

**done < "$input\_file"**

**if [ "$error\_found" = false ]; then**

**echo "No operand errors found. Job is valid."**

**fi**