1. Write a program to implement an address book with options given

below: a) Create address book. b) View address book. c) Insert a record.

d) Delete a record. e) Modify a record. f) Exit

RUN COMMAND: terminal in file location

chmod +x addressbook.sh

./addressbook.sh

#!/bin/bash

ADDRESS\_BOOK="address\_book.txt"

add\_entry() {

echo "Enter Name:"

read name

echo "Enter Email:"

read email

echo "Enter Mobile Number:"

read mobile

echo "Enter Address:"

read address

echo "Enter Other Important Fields (separated by commas):"

read other\_fields

echo "$name,$email,$mobile,$address,$other\_fields" >> "$ADDRESS\_BOOK"

echo "Entry added successfully!"

}

display\_entries() {

if [ ! -s "$ADDRESS\_BOOK" ]; then

echo "Address book is empty."

else

cat "$ADDRESS\_BOOK"

fi

}

search\_entries() {

echo "Enter the name to search:"

read name

grep -i "$name" "$ADDRESS\_BOOK"

}

update\_entry() {

echo "Enter the name of the entry to update:"

read name

entry=$(grep -i "$name" "$ADDRESS\_BOOK")

if [ -z "$entry" ]; then

echo "No entry found with that name."

return

fi

echo "Current entry: $entry"

echo "Enter new Email:"

read email

echo "Enter new Mobile Number:"

read mobile

echo "Enter new Address:"

read address

echo "Enter new Other Important Fields (separated by commas):"

read other\_fields

sed -i.bak "/$name/d" "$ADDRESS\_BOOK"

echo "$name,$email,$mobile,$address,$other\_fields" >> "$ADDRESS\_BOOK"

echo "Entry updated successfully!"

}

delete\_entry() {

echo "Enter the name of the entry to delete:"

read name

grep -i "$name" "$ADDRESS\_BOOK"

if [ $? -eq 0 ]; then

sed -i.bak "/$name/d" "$ADDRESS\_BOOK"

echo "Entry deleted successfully!"

else

echo "No entry found with that name."

fi

}

while true; do

echo "Address Book Menu"

echo "1. Add Entry"

echo "2. Display All Entries"

echo "3. Search Entry"

echo "4. Update Entry"

echo "5. Delete Entry"

echo "6. Exit"

echo "Enter your choice:"

read choice

case $choice in

1) add\_entry ;;

2) display\_entries ;;

3) search\_entries ;;

4) update\_entry ;;

5) delete\_entry ;;

6) exit ;;

\*) echo "Invalid choice. Please enter a number between 1 and 6." ;;

Esac

Done

2. Write a Program to implement multithreading for Matrix Operations

using Pthreads.

RUN COMMAND: chmod +x run\_matrix\_mult.sh

./ run\_matrix\_mult.sh

run\_matrix\_mult.sh

#!/bin/bash

gcc -pthread -o matrix\_mult matrix\_mult.c

./matrix\_mult

matrix\_mult.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define MAX 4 // Define the size of the matrices (4x4 for example)

int matA[MAX][MAX];

int matB[MAX][MAX];

int matC[MAX][MAX];

// Structure to pass data to threads

typedef struct {

int row;

int col;

} MatrixCell;

void \*multiply(void \*arg) {

MatrixCell \*cell = (MatrixCell \*)arg;

int sum = 0;

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

sum += matA[cell->row][k] \* matB[k][cell->col];

}

matC[cell->row][cell->col] = sum;

pthread\_exit(0);

}

int main() {

pthread\_t threads[MAX][MAX];

MatrixCell cell[MAX][MAX];

// Take input for matA

printf("Enter elements of matrix A (%dx%d):\n", MAX, MAX);

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

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

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

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

}

}

// Take input for matB

printf("\nEnter elements of matrix B (%dx%d):\n", MAX, MAX);

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

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

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

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

}

}

// Create threads for matrix multiplication

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

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

cell[i][j].row = i;

cell[i][j].col = j;

pthread\_create(&threads[i][j], NULL, multiply, &cell[i][j]);

}

}

// Wait for all threads to finish

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

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

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

}

}

// Print the result matrix matC

printf("\nResult matrix:\n");

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

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

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

}

printf("\n");

}

return 0;

}

3. Process control system calls: The demonstration of FORK and WAIT

system calls along with zombie and orphan states. Implement the

program in which the main program accepts the integers to be sorted.

Main program uses the FORK system call to create a new process

called a child process. Parent process sorts the integers using a sorting

algorithm and waits for the child process using WAIT system call to sort

the integers using any sorting algorithm. Also demonstrate zombie and

orphan states.

RUN COMMAND:

run\_zombie\_orphan.sh

#!/bin/bash

gcc -pthread -o zombie\_orphan zombie\_orphan.c

./ zombie\_orphan

zombie\_orphan.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

void bubbleSort(int arr[], int n) {

int i, j, temp;

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

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

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

temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

int main() {

int n, i;

pid\_t pid;

printf("Enter the number of integers to sort: ");

scanf("%d", &n);

int arr[n];

printf("Enter the integers: ");

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

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

}

pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0) {

printf("Child Process: Sorting in child process...\n");

bubbleSort(arr, n);

printf("Child Process: Sorted array: ");

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

printf("%d ", arr[i]);

}

printf("\n");

printf("Child Process: Sleeping for 5 seconds...\n");

sleep(5);

printf("Child Process: Exiting...\n");

exit(0);

}

else {

printf("Parent Process: Waiting for child process to complete...\n");

wait(NULL);

printf("Parent Process: Child process completed. Now sorting in parent process...\n");

bubbleSort(arr, n);

printf("Parent Process: Sorted array: ");

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

printf("%d ", arr[i]);

}

printf("\n");

printf("Parent Process: Sleeping for 5 seconds to demonstrate the zombie state of the child

process...\n");

sleep(5);

printf("Parent Process: Exiting...\n");

}

return 0;

}

4. Implementation of Classical Process Synchronization problems using

Threads and Semaphore, Mutex. Application to demonstrate: producerconsumer problem

run\_producer\_consumer.sh

#!/bin/bash

gcc -pthread -o producer\_consumer producer\_consumer.c

./ producer\_consumer

producer\_consumer.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

int \*buffer;

int buffer\_size, in = 0, out = 0;

int item\_count;

sem\_t empty, full;

pthread\_mutex\_t mutex;

int produce\_item() {

return rand() % 100;

}

void consume\_item(int item) {

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

}

void\* producer(void\* arg) {

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

int item;

for (int i = 0; i < item\_count; i++) {

item = produce\_item();

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

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

in = (in + 1) % buffer\_size;

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

sleep(1);

}

return NULL;

}

void\* consumer(void\* arg) {

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

int item;

for (int i = 0; i < item\_count; i++) {

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

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

out = (out + 1) % buffer\_size;

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

sleep(1);

}

return NULL;

}

int main() {

int producers\_count, consumers\_count;

printf("Enter the buffer size: ");

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

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

scanf("%d", &producers\_count);

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

scanf("%d", &consumers\_count);

printf("Enter the number of items to produce/consume: ");

scanf("%d", &item\_count);

buffer = (int \*)malloc(buffer\_size \* sizeof(int));

sem\_init(&empty, 0, buffer\_size);

sem\_init(&full, 0, 0);

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

pthread\_t producers[producers\_count], consumers[consumers\_count];

int producer\_ids[producers\_count], consumer\_ids[consumers\_count];

for (int i = 0; i < producers\_count; i++) {

producer\_ids[i] = i + 1;

pthread\_create(&producers[i], NULL, producer, (void \*)&producer\_ids[i]);

}

for (int i = 0; i < consumers\_count; i++) {

consumer\_ids[i] = i + 1;

pthread\_create(&consumers[i], NULL, consumer, (void \*)&consumer\_ids[i]);

}

for (int i = 0; i < producers\_count; i++) {

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

}

for (int i = 0; i < consumers\_count; i++) {

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

}

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

free(buffer);

return 0;

}

5. Implementation of Classical Process Synchronization problems using

mutex. Application to demonstrate: Reader- Writer problem.

run\_reader\_writer.sh

#!/bin/bash

gcc -pthread -o reader\_writer reader\_writer.c

./ reader\_writer

reader\_writer.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <semaphore.h>

#include <fcntl.h>

#include <sys/stat.h>

#define MAX\_READERS 5

sem\_t readSemaphore;

sem\_t writeSemaphore;

int readCount = 0;

void bubbleSort(int arr[], int n) {

int i, j, temp;

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

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

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

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

void reader(int id, int arr[], int n) {

sem\_wait(&readSemaphore);

readCount++;

if (readCount == 1) {

sem\_wait(&writeSemaphore);

}

sem\_post(&readSemaphore);

printf("Reader %d: Reading the array: ", id);

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

printf("%d ", arr[i]);

}

printf("\n");

sem\_wait(&readSemaphore);

readCount--;

if (readCount == 0) {

sem\_post(&writeSemaphore);

}

sem\_post(&readSemaphore);

}

void writer(int arr[], int n) {

sem\_wait(&writeSemaphore);

printf("Writer: Sorting the array...\n");

bubbleSort(arr, n);

printf("Writer: Sorted array: ");

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

printf("%d ", arr[i]);

}

printf("\n");

sem\_post(&writeSemaphore);

}

int main() {

int n, i;

printf("Enter the number of integers to sort: ");

scanf("%d", &n);

int arr[n];

printf("Enter the integers: ");

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

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

}

sem\_init(&readSemaphore, 1, 1);

sem\_init(&writeSemaphore, 1, 1);

for (i = 0; i < MAX\_READERS; i++) {

pid\_t pid = fork();

if (pid == 0) {

reader(i + 1, arr, n);

exit(0);

}

}

pid\_t writerPid = fork();

if (writerPid == 0) {

writer(arr, n);

exit(0);

}

for (i = 0; i < MAX\_READERS; i++) {

wait(NULL);

}

wait(NULL);

sem\_destroy(&readSemaphore);

sem\_destroy(&writeSemaphore);

printf("All processes have completed.\n");

return 0;

}

6. Write a program to compute the finish time, turnaround time and

waiting time for the following algorithms: First come First serve b)

Shortest Job First (Preemptive and Non-Preemptive) c) Priority

(Preemptive and Non Preemptive) d) Round robin

FCFS

#include <stdio.h>

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int completion\_time;

int waiting\_time;

int turnaround\_time;

} Process;

void sortProcessesByArrival(Process processes[], int n) {

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

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

if (processes[j].arrival\_time > processes[j + 1].arrival\_time) {

Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

}

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

int current\_time = 0;

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

if (current\_time < processes[i].arrival\_time) {

current\_time = processes[i].arrival\_time;

}

processes[i].completion\_time = current\_time + processes[i].burst\_time;

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

current\_time = processes[i].completion\_time;

}

}

void calculateAverages(Process processes[], int n, double \*avg\_wt, double \*avg\_tat) {

int total\_wt = 0, total\_tat = 0;

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

total\_wt += processes[i].waiting\_time;

total\_tat += processes[i].turnaround\_time;

}

\*avg\_wt = (double)total\_wt / n;

\*avg\_tat = (double)total\_tat / n;

}

int main() {

int n;

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

scanf("%d", &n);

Process processes[n];

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

printf("Enter Process ID, Arrival Time, and Burst Time for process %d: ", i + 1);

scanf("%d %d %d", &processes[i].pid, &processes[i].arrival\_time, &processes[i].burst\_time);

}

\

sortProcessesByArrival(processes, n);

calculateTimes(processes, n);

double avg\_wt, avg\_tat;

calculateAverages(processes, n, &avg\_wt, &avg\_tat);

printf("\nProcess ID | Arrival Time | Burst Time | Completion Time | Waiting Time | Turnaround

Time\n");

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

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

processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time,

processes[i].completion\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage Waiting Time: %.2f\n", avg\_wt);

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

return 0;

}

SFJ-Preemptive

#include <stdio.h>

#include <limits.h>

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int remaining\_time;

int completion\_time;

int waiting\_time;

int turnaround\_time;

} Process;

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

int current\_time = 0, completed = 0, min\_index;

int min\_remaining\_time = INT\_MAX;

int finish\_time, check = 0;

while (completed != n) {

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

if (processes[i].arrival\_time <= current\_time && processes[i].remaining\_time > 0 &&

processes[i].remaining\_time < min\_remaining\_time) {

min\_remaining\_time = processes[i].remaining\_time;

min\_index = i;

check = 1;

}

}

if (check == 0) {

current\_time++;

continue;

}

processes[min\_index].remaining\_time--;

min\_remaining\_time = processes[min\_index].remaining\_time;

if (min\_remaining\_time == 0) {

min\_remaining\_time = INT\_MAX;

}

if (processes[min\_index].remaining\_time == 0) {

completed++;

check = 0;

finish\_time = current\_time + 1;

processes[min\_index].completion\_time = finish\_time;

processes[min\_index].turnaround\_time = finish\_time - processes[min\_index].arrival\_time;

processes[min\_index].waiting\_time = processes[min\_index].turnaround\_time -

processes[min\_index].burst\_time;

}

current\_time++;

}

}

void calculateAverages(Process processes[], int n, double \*avg\_wt, double \*avg\_tat) {

int total\_wt = 0, total\_tat = 0;

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

total\_wt += processes[i].waiting\_time;

total\_tat += processes[i].turnaround\_time;

}

\*avg\_wt = (double)total\_wt / n;

\*avg\_tat = (double)total\_tat / n;

}

int main() {

int n;

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

scanf("%d", &n);

Process processes[n];

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

printf("Enter Process ID, Arrival Time, and Burst Time for process %d: ", i + 1);

scanf("%d %d %d", &processes[i].pid, &processes[i].arrival\_time, &processes[i].burst\_time);

processes[i].remaining\_time = processes[i].burst\_time;

}

calculateTimes(processes, n);

double avg\_wt, avg\_tat;

calculateAverages(processes, n, &avg\_wt, &avg\_tat);

printf("\nProcess ID | Arrival Time | Burst Time | Completion Time | Waiting Time | Turnaround Time\n");

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

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

processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time,

processes[i].completion\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage Waiting Time: %.2f\n", avg\_wt);

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

return 0;

}

SJFS- Non Preemptive

#include <stdio.h>

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int completion\_time;

int waiting\_time;

int turnaround\_time;

} Process;

void sortProcessesByBurstTime(Process processes[], int n, int current\_time) {

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

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

if ((processes[j].burst\_time > processes[j + 1].burst\_time) && (processes[j + 1].arrival\_time <=

current\_time)) {

Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

}

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

int current\_time = 0, completed = 0;

while (completed != n) {

int min\_index = -1;

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

if (processes[i].arrival\_time <= current\_time && processes[i].completion\_time == 0) {

if (min\_index == -1 || processes[i].burst\_time < processes[min\_index].burst\_time) {

min\_index = i;

}

}

}

if (min\_index != -1) {

current\_time += processes[min\_index].burst\_time;

processes[min\_index].completion\_time = current\_time;

processes[min\_index].turnaround\_time = processes[min\_index].completion\_time -

processes[min\_index].arrival\_time;

processes[min\_index].waiting\_time = processes[min\_index].turnaround\_time -

processes[min\_index].burst\_time;

completed++;

} else {

current\_time++;

}

}

}

void calculateAverages(Process processes[], int n, double \*avg\_wt, double \*avg\_tat) {

int total\_wt = 0, total\_tat = 0;

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

total\_wt += processes[i].waiting\_time;

total\_tat += processes[i].turnaround\_time;

}

\*avg\_wt = (double)total\_wt / n;

\*avg\_tat = (double)total\_tat / n;

}

int main() {

int n;

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

scanf("%d", &n);

Process processes[n];

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

printf("Enter Process ID, Arrival Time, and Burst Time for process %d: ", i + 1);

scanf("%d %d %d", &processes[i].pid, &processes[i].arrival\_time, &processes[i].burst\_time);

processes[i].completion\_time = 0; // Initialize completion time to 0

}

calculateTimes(processes, n);

double avg\_wt, avg\_tat;

calculateAverages(processes, n, &avg\_wt, &avg\_tat);

printf("\nProcess ID | Arrival Time | Burst Time | Completion Time | Waiting Time | Turnaround Time\n");

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

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

processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time,

processes[i].completion\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

printf("\nAverage Waiting Time: %.2f\n", avg\_wt);

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

return 0;

}

Priority- Preemptive

#include <stdio.h>

#include <stdbool.h>

typedef struct {

int pid;

int burst\_time;

int arrival\_time;

int remaining\_time;

int priority;

int completion\_time;

int waiting\_time;

int turnaround\_time;

bool completed;

} Process;

void priority\_preemptive(Process processes[], int n) {

int current\_time = 0;

int completed = 0;

while (completed < n) {

int idx = -1;

int min\_priority = 99999;

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

if (processes[i].arrival\_time <= current\_time && !processes[i].completed && processes[i].priority <

min\_priority) {

min\_priority = processes[i].priority;

idx = i;

}

}

if (idx != -1) {

processes[idx].remaining\_time--;

current\_time++;

if (processes[idx].remaining\_time == 0) {

processes[idx].completion\_time = current\_time;

processes[idx].turnaround\_time = processes[idx].completion\_time - processes[idx].arrival\_time;

processes[idx].waiting\_time = processes[idx].turnaround\_time - processes[idx].burst\_time;

processes[idx].completed = true;

completed++;

}

} else {

current\_time++;

}

}

}

void print\_table(Process processes[], int n) {

printf("\nPID\tBurst Time\tArrival Time\tPriority\tCompletion Time\tTurnaround Time\tWaiting Time\n");

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

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

processes[i].pid,

processes[i].burst\_time,

processes[i].arrival\_time,

processes[i].priority,

processes[i].completion\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

}

void calculate\_avg\_times(Process processes[], int n) {

float total\_tat = 0, total\_wt = 0;

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

total\_tat += processes[i].turnaround\_time;

total\_wt += processes[i].waiting\_time;

}

printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n);

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

}

int main() {

int n;

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

scanf("%d", &n);

Process processes[n];

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

printf("Enter arrival time, burst time, and priority for process %d: ", i + 1);

scanf("%d %d %d", &processes[i].arrival\_time, &processes[i].burst\_time, &processes[i].priority);

processes[i].pid = i + 1;

processes[i].remaining\_time = processes[i].burst\_time;

processes[i].completed = false;

}

priority\_preemptive(processes, n);

print\_table(processes, n);

calculate\_avg\_times(processes, n);

return 0;

}

Priority- Non Preemptive

#include <stdio.h>

typedef struct {

int pid;

int burst\_time;

int arrival\_time;

int priority;

int completion\_time;

int waiting\_time;

int turnaround\_time;

} Process;

void sort\_by\_priority\_and\_arrival(Process processes[], int n) {

Process temp;

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

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

if (processes[j].priority > processes[j + 1].priority ||

(processes[j].priority == processes[j + 1].priority && processes[j].arrival\_time > processes[j +

1].arrival\_time)) {

temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

}

void priority\_non\_preemptive(Process processes[], int n) {

int current\_time = 0;

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

if (processes[i].arrival\_time > current\_time) {

current\_time = processes[i].arrival\_time;

}

processes[i].completion\_time = current\_time + processes[i].burst\_time;

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

current\_time = processes[i].completion\_time;

}

}

void print\_table(Process processes[], int n) {

printf("\nPID\tBurst Time\tArrival Time\tPriority\tCompletion Time\tTurnaround Time\tWaiting Time\n");

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

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

processes[i].pid,

processes[i].burst\_time,

processes[i].arrival\_time,

processes[i].priority,

processes[i].completion\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

}

void calculate\_avg\_times(Process processes[], int n) {

float total\_tat = 0, total\_wt = 0;

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

total\_tat += processes[i].turnaround\_time;

total\_wt += processes[i].waiting\_time;

}

printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n);

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

}

int main() {

int n;

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

scanf("%d", &n);

Process processes[n];

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

printf("Enter arrival time, burst time, and priority for process %d: ", i + 1);

scanf("%d %d %d", &processes[i].arrival\_time, &processes[i].burst\_time, &processes[i].priority);

processes[i].pid = i + 1;

}

sort\_by\_priority\_and\_arrival(processes, n);

priority\_non\_preemptive(processes, n);

print\_table(processes, n);

calculate\_avg\_times(processes, n);

return 0;

}

Round Robin

#include <stdio.h>

#include <stdbool.h>

typedef struct {

int pid;

int arrival\_time;

int burst\_time;

int remaining\_time;

int completion\_time;

int waiting\_time;

int turnaround\_time;

} Process;

void queueUpdate(int queue[], int n, int maxIndex) {

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

if (queue[i] == 0) {

queue[i] = maxIndex + 1;

break;

}

}

}

void queueMaintain(int queue[], int n) {

for (int i = 0; i < n - 1 && queue[i + 1] != 0; i++) {

int temp = queue[i];

queue[i] = queue[i + 1];

queue[i + 1] = temp;

}

}

void checkNewArrival(int timer, int arrival[], int n, int \*maxIndex, int queue[]) {

if (timer <= arrival[n - 1]) {

for (int j = \*maxIndex + 1; j < n; j++) {

if (arrival[j] <= timer) {

\*maxIndex = j;

queueUpdate(queue, n, \*maxIndex);

}

}

}

}

int main() {

int n, tq, timer = 0, maxIndex = 0;

float avgWait = 0, avgTT = 0;

printf("Enter the time quantum: ");

scanf("%d", &tq);

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

scanf("%d", &n);

Process processes[n];

int arrival[n], queue[n];

bool complete[n];

printf("Enter the arrival time of the processes: ");

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

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

processes[i].arrival\_time = arrival[i];

processes[i].pid = i + 1;

complete[i] = false;

queue[i] = 0;

}

printf("Enter the burst time of the processes: ");

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

scanf("%d", &processes[i].burst\_time);

processes[i].remaining\_time = processes[i].burst\_time;

}

while (timer < arrival[0])

timer++;

queue[0] = 1;

while (true) {

bool done = true;

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

if (processes[i].remaining\_time != 0) {

done = false;

break;

}

}

if (done)

break;

for (int i = 0; (i < n) && (queue[i] != 0); i++) {

int ctr = 0;

while ((ctr < tq) && (processes[queue[0] - 1].remaining\_time > 0)) {

processes[queue[0] - 1].remaining\_time -= 1;

timer += 1;

ctr++;

checkNewArrival(timer, arrival, n, &maxIndex, queue);

}

if ((processes[queue[0] - 1].remaining\_time == 0) && (complete[queue[0] - 1] == false)) {

processes[queue[0] - 1].completion\_time = timer;

processes[queue[0] - 1].turnaround\_time = timer - processes[queue[0] - 1].arrival\_time;

processes[queue[0] - 1].waiting\_time = processes[queue[0] - 1].turnaround\_time -

processes[queue[0] - 1].burst\_time;

complete[queue[0] - 1] = true;

}

queueMaintain(queue, n);

}

}

printf("\nPID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time,

processes[i].burst\_time,

processes[i].completion\_time, processes[i].turnaround\_time, processes[i].waiting\_time);

avgWait += processes[i].waiting\_time;

avgTT += processes[i].turnaround\_time;

}

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

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

return 0;

}

7. Write a program to check whether a given system is in safe state or

not using Banker’s Deadlock Avoidance algorithm.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

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

bool isSafeState(int processes[], int avail[], int max[][MAX\_RESOURCES], int alloc[][MAX\_RESOURCES], int

need[][MAX\_RESOURCES], int num\_processes, int num\_resources) {

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

int safeSequence[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

// Initialize work as available resources

for (int i = 0; i < num\_resources; i++) {

work[i] = avail[i];

}

int count = 0;

// Find the processes that can finish

while (count < num\_processes) {

bool found = false;

for (int p = 0; p < num\_processes; p++) {

if (!finish[p]) {

int j;

for (j = 0; j < num\_resources; j++) {

if (need[p][j] > work[j])

break;

}

// If all resources for process p can be allocated

if (j == num\_resources) {

// Process p can finish

for (int k = 0; k < num\_resources; k++) {

work[k] += alloc[p][k];

}

safeSequence[count++] = p;

finish[p] = true;

found = true;

}

}

}

// If no process could be found in this iteration, system is not in a safe state

if (!found) {

return false;

}

}

// If all processes can finish, system is in a safe state

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

for (int i = 0; i < num\_processes; i++) {

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

}

printf("\n");

return true;

}

// Function to calculate the need matrix

void calculateNeed(int need[][MAX\_RESOURCES], int max[][MAX\_RESOURCES], int alloc[][MAX\_RESOURCES],

int num\_processes, int num\_resources) {

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

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

}

}

}

int main() {

int num\_processes, num\_resources;

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

scanf("%d", &num\_processes);

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

scanf("%d", &num\_resources);

int processes[MAX\_PROCESSES], avail[MAX\_RESOURCES], max[MAX\_PROCESSES][MAX\_RESOURCES],

alloc[MAX\_PROCESSES][MAX\_RESOURCES], need[MAX\_PROCESSES][MAX\_RESOURCES];

// Input the available resources

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

for (int i = 0; i < num\_resources; i++) {

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

}

// Input the maximum demand of each process

printf("Enter the maximum demand of each process: \n");

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

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

}

}

// Input the allocated resources for each process

printf("Enter the allocated resources for each process: \n");

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

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

}

}

// Calculate the need matrix

calculateNeed(need, max, alloc, num\_processes, num\_resources);

// Check if the system is in a safe state

if (!isSafeState(processes, avail, max, alloc, need, num\_processes, num\_resources)) {

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

}

return 0;

}

8. Write a program to calculate the number of page faults for a reference

string for the following page replacement algorithms: a. FIFO b)LRU

c)Optimal

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 10

// Function prototypes

int fifo(int pages[], int n, int frames);

int lru(int pages[], int n, int frames);

int optimal(int pages[], int n, int frames);

int main() {

int pages[MAX\_PAGES];

int n, frames;

// Input the number of pages

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

scanf("%d", &n);

// Input the reference string

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

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

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

}

// Input the number of frames

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

scanf("%d", &frames);

// Calculate page faults for each algorithm

int fifo\_faults = fifo(pages, n, frames);

int lru\_faults = lru(pages, n, frames);

int optimal\_faults = optimal(pages, n, frames);

// Display results

printf("\nPage Faults:\n");

printf("FIFO: %d\n", fifo\_faults);

printf("LRU: %d\n", lru\_faults);

printf("Optimal: %d\n", optimal\_faults);

return 0;

}

// FIFO Page Replacement Algorithm

int fifo(int pages[], int n, int frames) {

int page[frames];

int page\_faults = 0, index = 0, found, i;

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

page[i] = -1; // Initialize frames

}

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

found = 0;

// Check if page is already in frames

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

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

found = 1;

break;

}

}

// Page fault occurs

if (!found) {

page[index] = pages[i];

index = (index + 1) % frames; // Move to the next frame

page\_faults++;

}

}

return page\_faults;

}

// LRU Page Replacement Algorithm

int lru(int pages[], int n, int frames) {

int page[frames];

int page\_faults = 0, found, i, j, lru\_index, lru\_time[MAX\_FRAMES] = {0};

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

page[i] = -1; // Initialize frames

}

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

found = 0;

// Check if page is already in frames

for (j = 0; j < frames; j++) {

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

found = 1;

lru\_time[j] = i; // Update the last used time

break;

}

}

// Page fault occurs

if (!found) {

// Find the LRU page

lru\_index = 0;

for (j = 1; j < frames; j++) {

if (lru\_time[j] < lru\_time[lru\_index]) {

lru\_index = j;

}

}

page[lru\_index] = pages[i]; // Replace the LRU page

lru\_time[lru\_index] = i; // Update the last used time

page\_faults++;

}

}

return page\_faults;

}

// Optimal Page Replacement Algorithm

int optimal(int pages[], int n, int frames) {

int page[frames];

int page\_faults = 0, found, i, j, k, farthest;

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

page[i] = -1; // Initialize frames

}

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

found = 0;

// Check if page is already in frames

for (j = 0; j < frames; j++) {

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

found = 1;

break;

}

}

// Page fault occurs

if (!found) {

// If there is an empty frame

for (j = 0; j < frames; j++) {

if (page[j] == -1) {

page[j] = pages[i];

page\_faults++;

found = 1;

break;

}

}

// If all frames are full, replace the optimal page

if (!found) {

farthest = -1;

int index\_to\_replace = -1;

// Find the page that will not be used for the longest period of time

for (j = 0; j < frames; j++) {

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

if (page[j] == pages[k]) {

if (k > farthest) {

farthest = k;

index\_to\_replace = j;

}

break;

}

}

// If the page is not found in the future reference string, replace it

if (k == n) {

index\_to\_replace = j;

break;

}

}

page[index\_to\_replace] = pages[i]; // Replace the optimal page

page\_faults++;

}

}

}

return page\_faults;

}

9. Write a program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look

considering the initial head position moving away from the spindle

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 100

int abs\_diff(int a, int b) {

return abs(a - b);

}

// SSTF Algorithm

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

bool processed[MAX] = {false};

int current\_head = head;

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

printf("Order of requests: ");

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

int min\_distance = MAX;

int next\_index = -1;

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

if (!processed[j]) {

int distance = abs\_diff(current\_head, requests[j]);

if (distance < min\_distance) {

min\_distance = distance;

next\_index = j;

}

}

}

processed[next\_index] = true;

current\_head = requests[next\_index];

printf("%d ", current\_head);

}

printf("\n");

}

// SCAN Algorithm

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

int requests\_sorted[MAX + 1];

requests\_sorted[0] = head;

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

requests\_sorted[i + 1] = requests[i];

}

// Sort the requests including the head position

n++;

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

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

if (requests\_sorted[i] > requests\_sorted[j]) {

int temp = requests\_sorted[i];

requests\_sorted[i] = requests\_sorted[j];

requests\_sorted[j] = temp;

}

}

}

int pos;

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

if (requests\_sorted[i] == head) {

pos = i;

break;

}

}

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

printf("Order of requests: ");\

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

printf("%d ", requests\_sorted[i]);

}

printf("%d ", disk\_size - 1); // End of the disk

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

printf("%d ", requests\_sorted[i]);

}

printf("\n");

}

// C-LOOK Algorithm

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

int requests\_sorted[MAX];

// Copy requests and sort them

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

requests\_sorted[i] = requests[i];

}

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

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

if (requests\_sorted[i] > requests\_sorted[j]) {

int temp = requests\_sorted[i];

requests\_sorted[i] = requests\_sorted[j];

requests\_sorted[j] = temp;

}

}

}

int pos;

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

if (requests\_sorted[i] >= head) {

pos = i;

break;

}

}

printf("\nC-LOOK Disk Scheduling:\n");

printf("Order of requests: ");

// Process requests from the current position to the end

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

printf("%d ", requests\_sorted[i]);

}

// Process requests from the beginning to the position

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

printf("%d ", requests\_sorted[i]);

}

printf("\n");

}

int main() {

int n, head, disk\_size;

int requests[MAX];

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

scanf("%d", &n);

printf("Enter the disk size: ");

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

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

scanf("%d", &head);

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

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

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

}

// Call SSTF Algorithm

sstf(requests, n, head);

// Call SCAN Algorithm

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

// Call C-LOOK Algorithm

clook(requests, n, head);

return 0;

}