**1. Write a c Program to create a process using fork.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

pid\_t pid;

// Create a new process

pid = fork();

// Check for errors

if (pid < 0) {

fprintf(stderr, "Fork failed.\n");

return 1;

}

// Child process

if (pid == 0) {

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

printf("Parent's PID: %d\n", getppid());

// Child process-specific code here

}

// Parent process

else {

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

printf("Child's PID: %d\n", pid);

// Parent process-specific code here

}

return 0;

}

**2. Simulation of FCFS CPU Scheduling algorithm.**

#include <stdio.h>

// Structure to represent a process

typedef struct {

int processId;

int arrivalTime;

int burstTime;

} Process;

// Function to calculate waiting time and turnaround time for each process

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

int i, currentTime = 0;

int waitingTime = 0, turnaroundTime = 0;

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

// Calculate waiting time for the current process

if (currentTime > processes[i].arrivalTime)

waitingTime = currentTime - processes[i].arrivalTime;

else

waitingTime = 0;

// Calculate turnaround time for the current process

turnaroundTime = waitingTime + processes[i].burstTime;

// Update current time

currentTime += processes[i].burstTime;

// Print process details

printf("Process ID: %d\tArrival Time: %d\tBurst Time: %d\tWaiting Time: %d\tTurnaround Time: %d\n",

processes[i].processId, processes[i].arrivalTime, processes[i].burstTime, waitingTime, turnaroundTime);

}

}

int main() {

int n, i;

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

scanf("%d", &n);

// Create an array of processes

Process processes[n];

// Read process details from the user

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

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

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

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

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

processes[i].processId = i + 1;

}

// Calculate waiting time and turnaround time for each process

calculateTimes(processes, n);

return 0;

}

3. Simulation of SJF CPU Scheduling algorithm.

#include <stdio.h>

// Structure to represent a process

typedef struct {

int processId;

int arrivalTime;

int burstTime;

int waitingTime;

int turnaroundTime;

int isCompleted;

} Process;

// Function to calculate waiting time and turnaround time for each process

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

int i, currentTime = 0, completed = 0;

float totalWaitingTime = 0, totalTurnaroundTime = 0;

while (completed != n) {

int shortestJob = -1;

int shortestBurstTime = 9999;

// Find the process with the shortest burst time among the arrived processes

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

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

if (processes[i].burstTime < shortestBurstTime) {

shortestBurstTime = processes[i].burstTime;

shortestJob = i;

}

}

}

if (shortestJob == -1) {

currentTime++;

continue;

}

// Update waiting time and turnaround time for the completed process

processes[shortestJob].waitingTime = currentTime - processes[shortestJob].arrivalTime;

processes[shortestJob].turnaroundTime = processes[shortestJob].waitingTime + processes[shortestJob].burstTime;

totalWaitingTime += processes[shortestJob].waitingTime;

totalTurnaroundTime += processes[shortestJob].turnaroundTime;

processes[shortestJob].isCompleted = 1;

completed++;

currentTime += processes[shortestJob].burstTime;

}

// Print process details

printf("Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].processId, processes[i].arrivalTime,

processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);

}

// Print average waiting time and average turnaround time

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

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

}

int main() {

int n, i;

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

scanf("%d", &n);

// Create an array of processes

Process processes[n];

// Read process details from the user

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

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

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

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

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

processes[i].processId = i + 1;

processes[i].isCompleted = 0;

}

// Calculate waiting time and turnaround time for each process

calculateTimes(processes, n);

return 0;

}

4. Simulation of RR CPU Scheduling algorithm.

#include <stdio.h>

// Structure to represent a process

typedef struct {

int processId;

int arrivalTime;

int burstTime;

int remainingTime;

int waitingTime;

int turnaroundTime;

} Process;

// Function to calculate waiting time and turnaround time for each process

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

int i, currentTime = 0, completed = 0;

int totalWaitingTime = 0, totalTurnaroundTime = 0;

while (completed != n) {

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

// If the process has already completed, skip to the next process

if (processes[i].remainingTime == 0)

continue;

// Execute the process for the time quantum or until it completes

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

currentTime += processes[i].remainingTime;

processes[i].remainingTime = 0;

completed++;

processes[i].waitingTime = currentTime - processes[i].arrivalTime - processes[i].burstTime;

processes[i].turnaroundTime = currentTime - processes[i].arrivalTime;

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

} else {

currentTime += timeQuantum;

processes[i].remainingTime -= timeQuantum;

}

}

}

// Print process details

printf("Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].processId, processes[i].arrivalTime,

processes[i].burstTime, processes[i].waitingTime, processes[i].turnaroundTime);

}

// Print average waiting time and average turnaround time

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

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

}

int main() {

int n, i, timeQuantum;

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

scanf("%d", &n);

printf("Enter the time quantum: ");

scanf("%d", &timeQuantum);

// Create an array of processes

Process processes[n];

// Read process details from the user

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

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

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

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

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

processes[i].processId = i + 1;

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

processes[i].waitingTime = 0;

processes[i].turnaroundTime = 0;

}

// Calculate waiting time and turnaround time for each process

calculateTimes(processes, n, timeQuantum);

return 0;

}

**5. Simulation of deadlock (in the below program the output will no deadlock detected)**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

bool isDeadlock(int processes, int resources, int allocation[MAX\_PROCESSES][MAX\_RESOURCES], int request[MAX\_PROCESSES][MAX\_RESOURCES]) {

int i, j, k;

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

bool needMoreResources;

bool safeState = false;

int work[MAX\_RESOURCES];

int available[MAX\_RESOURCES];

// Calculate the available resources

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

int sum = 0;

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

sum += allocation[j][i];

}

available[i] = sum;

}

// Check if all processes are finished

while (true) {

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

if (!finished[i]) {

needMoreResources = false;

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

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

needMoreResources = true;

break;

}

}

if (!needMoreResources) {

// Release the allocated resources

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

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

}

finished[i] = true;

safeState = true;

}

}

}

// If no processes can be finished, break the loop

if (!safeState) {

break;

}

safeState = false;

}

// Check if all processes are finished

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

if (!finished[i]) {

return true; // Deadlock detected

}

}

return false; // No deadlock detected

}

int main() {

int processes, resources;

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

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

// Sample input

processes = 5;

resources = 3;

int allocationMatrix[5][3] = {

{1, 2, 2},

{1, 0, 1},

{2, 2, 1},

{0, 1, 0},

{0, 0, 2}

};

int requestMatrix[5][3] = {

{1, 1, 2},

{0, 0, 1},

{1, 0, 0},

{0, 0, 0},

{0, 0, 2}

};

// Assign the sample input to the respective variables

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

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

allocation[i][j] = allocationMatrix[i][j];

request[i][j] = requestMatrix[i][j];

}

}

// Call the deadlock detection function

bool deadlockDetected = isDeadlock(processes, resources, allocation, request);

// Display the output

if (deadlockDetected) {

printf("Deadlock detected.\n");

} else {

printf("No deadlock detected.\n");

}

return 0;

}

**6. Write a Program For Inter Process Communication using Pipe.**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#define BUFFER\_SIZE 256

int main() {

int pipefd[2];

pid\_t pid;

char buffer[BUFFER\_SIZE];

// Create the pipe

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

perror("Pipe creation failed");

exit(EXIT\_FAILURE);

}

// Fork a child process

pid = fork();

if (pid == -1) {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

// Child process

close(pipefd[1]); // Close the write end of the pipe

// Read from the pipe

read(pipefd[0], buffer, BUFFER\_SIZE);

printf("Child process received message: %s\n", buffer);

close(pipefd[0]); // Close the read end of the pipe

} else {

// Parent process

close(pipefd[0]); // Close the read end of the pipe

char message[] = "Hello from parent process!";

// Write to the pipe

write(pipefd[1], message, strlen(message) + 1);

close(pipefd[1]); // Close the write end of the pipe

}

return 0;

}

**7. Write a Program to simulate FIFO page Replacement algorithm.**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

void printPageFrames(int frames[], int frameCount) {

printf("Page Frames: ");

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

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

}

printf("\n");

}

bool isPagePresent(int frames[], int frameCount, int pageNumber) {

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

if (frames[i] == pageNumber) {

return true;

}

}

return false;

}

int getPageFaults(int referenceString[], int referenceStringLength, int frameCount) {

int frames[MAX\_FRAMES] = { 0 };

int pageFaults = 0;

int currentIndex = 0;

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

int pageNumber = referenceString[i];

if (!isPagePresent(frames, frameCount, pageNumber)) {

frames[currentIndex] = pageNumber;

currentIndex = (currentIndex + 1) % frameCount;

pageFaults++;

}

printPageFrames(frames, frameCount);

}

return pageFaults;

}

int main() {

int referenceString[] = { 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 };

int referenceStringLength = sizeof(referenceString) / sizeof(referenceString[0]);

int frameCount = 3;

printf("Reference String: ");

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

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

}

printf("\n");

int pageFaults = getPageFaults(referenceString, referenceStringLength, frameCount);

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

return 0;

}

**8. Write a Program to simulate LRU page Replacement algorithm.**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

void printPageFrames(int frames[], int frameCount) {

printf("Page Frames: ");

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

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

}

printf("\n");

}

bool isPagePresent(int frames[], int frameCount, int pageNumber) {

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

if (frames[i] == pageNumber) {

return true;

}

}

return false;

}

int getLRUPage(int frames[], int frameCount, int pageReference[], int referenceIndex) {

int lruIndex = -1;

int farthestIndex = referenceIndex;

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

int frameNumber = frames[i];

for (int j = referenceIndex - 1; j >= 0; j--) {

if (frameNumber == pageReference[j]) {

if (j < farthestIndex) {

farthestIndex = j;

lruIndex = i;

}

break;

}

}

}

return lruIndex;

}

int getPageFaults(int pageReference[], int referenceStringLength, int frameCount) {

int frames[MAX\_FRAMES] = { 0 };

int pageFaults = 0;

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

int pageNumber = pageReference[i];

if (!isPagePresent(frames, frameCount, pageNumber)) {

int lruIndex = getLRUPage(frames, frameCount, pageReference, i);

frames[lruIndex] = pageNumber;

pageFaults++;

}

printPageFrames(frames, frameCount);

}

return pageFaults;

}

int main() {

int pageReference[] = { 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 };

int referenceStringLength = sizeof(pageReference) / sizeof(pageReference[0]);

int frameCount = 3;

printf("Page Reference: ");

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

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

}

printf("\n");

int pageFaults = getPageFaults(pageReference, referenceStringLength, frameCount);

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

return 0;

}

**9.Write a Program to simulate Optimal page Replacement algorithm.**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define MAX\_FRAMES 10

void printPageFrames(int frames[], int frameCount) {

printf("Page Frames: ");

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

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

}

printf("\n");

}

bool isPagePresent(int frames[], int frameCount, int pageNumber) {

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

if (frames[i] == pageNumber) {

return true;

}

}

return false;

}

int getNextReferenceIndex(int pageReference[], int referenceIndex, int referenceStringLength, int pageNumber) {

for (int i = referenceIndex + 1; i < referenceStringLength; i++) {

if (pageReference[i] == pageNumber) {

return i;

}

}

return INT\_MAX;

}

int getPageFaults(int pageReference[], int referenceStringLength, int frameCount) {

int frames[MAX\_FRAMES] = { 0 };

int pageFaults = 0;

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

int pageNumber = pageReference[i];

if (!isPagePresent(frames, frameCount, pageNumber)) {

int farthestIndex = -1;

int pageToReplace = -1;

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

if (!isPagePresent(pageReference, referenceStringLength, frames[j])) {

pageToReplace = j;

break;

}

int nextReferenceIndex = getNextReferenceIndex(pageReference, i, referenceStringLength, frames[j]);

if (nextReferenceIndex == INT\_MAX) {

pageToReplace = j;

break;

}

if (nextReferenceIndex > farthestIndex) {

farthestIndex = nextReferenceIndex;

pageToReplace = j;

}

}

frames[pageToReplace] = pageNumber;

pageFaults++;

}

printPageFrames(frames, frameCount);

}

return pageFaults;

}

int main() {

int pageReference[] = { 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 };

int referenceStringLength = sizeof(pageReference) / sizeof(pageReference[0]);

int frameCount = 3;

printf("Page Reference: ");

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

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

}

printf("\n");

int pageFaults = getPageFaults(pageReference, referenceStringLength, frameCount);

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

return 0;

}

**10. Write a Program for Continuous File allocation method.**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_BLOCKS 100

void printDiskStatus(bool disk[], int diskSize) {

printf("Disk Status: ");

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

printf("%c ", disk[i] ? '#' : '.');

}

printf("\n");

}

bool allocateFile(bool disk[], int diskSize, int startBlock, int fileSize) {

if (startBlock + fileSize > diskSize) {

return false; // Not enough space on disk

}

for (int i = startBlock; i < startBlock + fileSize; i++) {

if (disk[i]) {

return false; // Block already allocated

}

}

for (int i = startBlock; i < startBlock + fileSize; i++) {

disk[i] = true;

}

return true; // File allocation successful

}

void deallocateFile(bool disk[], int diskSize, int startBlock, int fileSize) {

for (int i = startBlock; i < startBlock + fileSize; i++) {

disk[i] = false;

}

}

int main() {

bool disk[MAX\_BLOCKS] = { false };

int diskSize = sizeof(disk) / sizeof(disk[0]);

int option;

int startBlock, fileSize;

while (true) {

printf("\nMenu:\n");

printf("1. Allocate File\n");

printf("2. Deallocate File\n");

printf("3. Print Disk Status\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &option);

switch (option) {

case 1:

printf("Enter starting block number: ");

scanf("%d", &startBlock);

printf("Enter file size: ");

scanf("%d", &fileSize);

if (allocateFile(disk, diskSize, startBlock, fileSize)) {

printf("File allocated successfully.\n");

} else {

printf("File allocation failed. Not enough space or blocks already allocated.\n");

}

break;

case 2:

printf("Enter starting block number: ");

scanf("%d", &startBlock);

printf("Enter file size: ");

scanf("%d", &fileSize);

deallocateFile(disk, diskSize, startBlock, fileSize);

printf("File deallocated successfully.\n");

break;

case 3:

printDiskStatus(disk, diskSize);

break;

case 4:

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

return 0;

default:

printf("Invalid option. Please try again.\n");

}

}

}

**11. Write a Program for Linked File allocation method**.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_BLOCKS 100

struct FileBlock {

int blockNumber;

struct FileBlock\* nextBlock;

};

void printDiskStatus(bool allocatedBlocks[], int diskSize) {

printf("Disk Status: ");

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

printf("%c ", allocatedBlocks[i] ? '#' : '.');

}

printf("\n");

}

struct FileBlock\* allocateFile(struct FileBlock\* file, bool allocatedBlocks[], int diskSize, int fileSize) {

struct FileBlock\* newBlock = NULL;

int allocatedBlocksCount = 0;

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

if (!allocatedBlocks[i]) {

if (file == NULL) {

file = (struct FileBlock\*)malloc(sizeof(struct FileBlock));

file->blockNumber = i;

file->nextBlock = NULL;

newBlock = file;

} else {

newBlock->nextBlock = (struct FileBlock\*)malloc(sizeof(struct FileBlock));

newBlock = newBlock->nextBlock;

newBlock->blockNumber = i;

newBlock->nextBlock = NULL;

}

allocatedBlocks[i] = true;

allocatedBlocksCount++;

if (allocatedBlocksCount == fileSize) {

return file; // File allocation complete

}

}

}

// Not enough space on disk

// Deallocate the partially allocated file

while (file != NULL) {

int blockNumber = file->blockNumber;

allocatedBlocks[blockNumber] = false;

struct FileBlock\* temp = file;

file = file->nextBlock;

free(temp);

}

return NULL;

}

void deallocateFile(struct FileBlock\* file, bool allocatedBlocks[]) {

while (file != NULL) {

int blockNumber = file->blockNumber;

allocatedBlocks[blockNumber] = false;

struct FileBlock\* temp = file;

file = file->nextBlock;

free(temp);

}

}

int main() {

bool allocatedBlocks[MAX\_BLOCKS] = { false };

int diskSize = sizeof(allocatedBlocks) / sizeof(allocatedBlocks[0]);

struct FileBlock\* file = NULL;

int option;

int fileSize;

while (true) {

printf("\nMenu:\n");

printf("1. Allocate File\n");

printf("2. Deallocate File\n");

printf("3. Print Disk Status\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &option);

switch (option) {

case 1:

printf("Enter file size: ");

scanf("%d", &fileSize);

file = allocateFile(file, allocatedBlocks, diskSize, fileSize);

if (file != NULL) {

printf("File allocated successfully.\n");

} else {

printf("File allocation failed. Not enough space on disk.\n");

}

break;

case 2:

if (file == NULL) {

printf("No file to deallocate.\n");

} else {

deallocateFile(file, allocatedBlocks);

file = NULL;

printf("File deallocated successfully.\n");

}

break;

case 3:

printDiskStatus(allocatedBlocks, diskSize);

break;

case 4:

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

return 0;

default:

printf("Invalid option. Please try again.\n");

}

}

}

**12. Write a Program for Indexed File allocation methods.**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_BLOCKS 100

struct FileBlock {

int blockNumber;

bool allocated;

};

void printDiskStatus(bool allocatedBlocks[], int diskSize) {

printf("Disk Status: ");

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

printf("%c ", allocatedBlocks[i] ? '#' : '.');

}

printf("\n");

}

void allocateFile(struct FileBlock\* indexBlock, bool allocatedBlocks[], int diskSize, int indexBlockSize, int fileSize) {

int allocatedBlocksCount = 0;

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

int blockNumber = indexBlock[i].blockNumber;

if (!allocatedBlocks[blockNumber]) {

allocatedBlocks[blockNumber] = true;

allocatedBlocksCount++;

}

if (allocatedBlocksCount == fileSize) {

break; // File allocation complete

}

}

}

void deallocateFile(struct FileBlock\* indexBlock, bool allocatedBlocks[], int indexBlockSize) {

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

int blockNumber = indexBlock[i].blockNumber;

if (allocatedBlocks[blockNumber]) {

allocatedBlocks[blockNumber] = false;

}

}

}

int main() {

bool allocatedBlocks[MAX\_BLOCKS] = { false };

int diskSize = sizeof(allocatedBlocks) / sizeof(allocatedBlocks[0]);

int indexBlockSize;

printf("Enter index block size: ");

scanf("%d", &indexBlockSize);

struct FileBlock indexBlock[indexBlockSize];

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

printf("Enter block number for index %d: ", i);

scanf("%d", &indexBlock[i].blockNumber);

indexBlock[i].allocated = false;

}

int option;

int fileSize;

while (true) {

printf("\nMenu:\n");

printf("1. Allocate File\n");

printf("2. Deallocate File\n");

printf("3. Print Disk Status\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &option);

switch (option) {

case 1:

printf("Enter file size: ");

scanf("%d", &fileSize);

allocateFile(indexBlock, allocatedBlocks, diskSize, indexBlockSize, fileSize);

printf("File allocated successfully.\n");

break;

case 2:

deallocateFile(indexBlock, allocatedBlocks, indexBlockSize);

printf("File deallocated successfully.\n");

break;

case 3:

printDiskStatus(allocatedBlocks, diskSize);

break;

case 4:

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

return 0;

default:

printf("Invalid option. Please try again.\n");

}

}

}

**13. Write a Program to simulate FCFS disk Scheduling Algorithm.**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void calculateSeekTime(int requestQueue[], int queueSize, int initialPosition) {

int totalSeekTime = 0;

int currentPosition = initialPosition;

printf("Seek Sequence: %d", currentPosition);

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

int distance = abs(requestQueue[i] - currentPosition);

totalSeekTime += distance;

currentPosition = requestQueue[i];

printf(" -> %d", currentPosition);

}

printf("\nTotal Seek Time: %d\n", totalSeekTime);

}

int main() {

int requestQueue[100];

int queueSize;

int initialPosition;

printf("Enter the size of the request queue: ");

scanf("%d", &queueSize);

printf("Enter the request queue elements:\n");

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

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

}

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

scanf("%d", &initialPosition);

calculateSeekTime(requestQueue, queueSize, initialPosition);

return 0;

}

**14. Write a Program to simulate SSTF disk Scheduling Algorithm.**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <stdbool.h>

int findClosestRequest(int requestQueue[], bool visited[], int currentPosition, int queueSize) {

int minDistance = INT\_MAX;

int closestRequest = -1;

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

if (!visited[i]) {

int distance = abs(requestQueue[i] - currentPosition);

if (distance < minDistance) {

minDistance = distance;

closestRequest = i;

}

}

}

return closestRequest;

}

void calculateSeekTime(int requestQueue[], int queueSize, int initialPosition) {

bool visited[100] = { false };

int totalSeekTime = 0;

int currentPosition = initialPosition;

printf("Seek Sequence: %d", currentPosition);

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

int closestRequest = findClosestRequest(requestQueue, visited, currentPosition, queueSize);

visited[closestRequest] = true;

int distance = abs(requestQueue[closestRequest] - currentPosition);

totalSeekTime += distance;

currentPosition = requestQueue[closestRequest];

printf(" -> %d", currentPosition);

}

printf("\nTotal Seek Time: %d\n", totalSeekTime);

}

int main() {

int requestQueue[100];

int queueSize;

int initialPosition;

printf("Enter the size of the request queue: ");

scanf("%d", &queueSize);

printf("Enter the request queue elements:\n");

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

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

}

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

scanf("%d", &initialPosition);

calculateSeekTime(requestQueue, queueSize, initialPosition);

return 0;

}

**15. Write a Program to simulate SCAN disk Scheduling Algorithm.**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void sortRequests(int requestQueue[], int queueSize) {

// Bubble sort the request queue in ascending order

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

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

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

// Swap the elements

int temp = requestQueue[j];

requestQueue[j] = requestQueue[j + 1];

requestQueue[j + 1] = temp;

}

}

}

}

void calculateSeekTime(int requestQueue[], int queueSize, int initialPosition, bool direction) {

int totalSeekTime = 0;

int currentPosition = initialPosition;

int index;

if (direction) {

// Find the index where the current position is present or the next higher position

for (index = 0; index < queueSize; index++) {

if (requestQueue[index] >= currentPosition) {

break;

}

}

} else {

// Find the index where the current position is present or the next lower position

for (index = queueSize - 1; index >= 0; index--) {

if (requestQueue[index] <= currentPosition) {

break;

}

}

}

printf("Seek Sequence: %d", currentPosition);

// Traverse the request queue based on the direction

if (direction) {

for (int i = index; i < queueSize; i++) {

printf(" -> %d", requestQueue[i]);

totalSeekTime += abs(requestQueue[i] - currentPosition);

currentPosition = requestQueue[i];

}

// Move to the last track

totalSeekTime += abs(currentPosition - (queueSize - 1));

currentPosition = queueSize - 1;

printf(" -> %d", currentPosition);

// Traverse back to the initial position

for (int i = queueSize - 2; i >= index; i--) {

printf(" -> %d", requestQueue[i]);

totalSeekTime += abs(requestQueue[i] - currentPosition);

currentPosition = requestQueue[i];

}

} else {

for (int i = index; i >= 0; i--) {

printf(" -> %d", requestQueue[i]);

totalSeekTime += abs(requestQueue[i] - currentPosition);

currentPosition = requestQueue[i];

}

// Move to the first track

totalSeekTime += abs(currentPosition - 0);

currentPosition = 0;

printf(" -> %d", currentPosition);

// Traverse back to the initial position

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

printf(" -> %d", requestQueue[i]);

totalSeekTime += abs(requestQueue[i] - currentPosition);

currentPosition = requestQueue[i];

}

}

printf("\nTotal Seek Time: %d\n", totalSeekTime);

}

int main() {

int requestQueue[100];

int queueSize;

int initialPosition;

bool direction;

printf("Enter the size of the request queue: ");

scanf("%d", &queueSize);

printf("Enter the request queue elements:\n");

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

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

}

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

scanf("%d", &initialPosition);

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

scanf("%d", &direction);

// Sort the request queue in ascending order

sortRequests(requestQueue, queueSize);

calculateSeekTime(requestQueue, queueSize, initialPosition, direction);

return 0;

}

**16. Write a Program to simulate CSCAN disk Scheduling Algorithm.**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void sortRequests(int requestQueue[], int queueSize) {

// Bubble sort the request queue in ascending order

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

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

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

// Swap the elements

int temp = requestQueue[j];

requestQueue[j] = requestQueue[j + 1];

requestQueue[j + 1] = temp;

}

}

}

}

void calculateSeekTime(int requestQueue[], int queueSize, int initialPosition) {

int totalSeekTime = 0;

int currentPosition = initialPosition;

int index;

// Find the index where the current position is present or the next higher position

for (index = 0; index < queueSize; index++) {

if (requestQueue[index] >= currentPosition) {

break;

}

}

printf("Seek Sequence: %d", currentPosition);

// Traverse the request queue in the forward direction

for (int i = index; i < queueSize; i++) {

printf(" -> %d", requestQueue[i]);

totalSeekTime += abs(requestQueue[i] - currentPosition);

currentPosition = requestQueue[i];

}

// Move to the last track

totalSeekTime += abs(currentPosition - (queueSize - 1));

currentPosition = queueSize - 1;

printf(" -> %d", currentPosition);

// Move back to the first track

totalSeekTime += abs(currentPosition - 0);

currentPosition = 0;

printf(" -> %d", currentPosition);

// Traverse the request queue in the forward direction again

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

printf(" -> %d", requestQueue[i]);

totalSeekTime += abs(requestQueue[i] - currentPosition);

currentPosition = requestQueue[i];

}

printf("\nTotal Seek Time: %d\n", totalSeekTime);

}

int main() {

int requestQueue[100];

int queueSize;

int initialPosition;

printf("Enter the size of the request queue: ");

scanf("%d", &queueSize);

printf("Enter the request queue elements:\n");

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

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

}

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

scanf("%d", &initialPosition);

// Sort the request queue in ascending order

sortRequests(requestQueue, queueSize);

calculateSeekTime(requestQueue, queueSize, initialPosition);

return 0;

}

**17. C program that uses Linux system calls to remove directories**

recursively:

#include &lt;stdio.h&gt;

#include &lt;unistd.h&gt;

#include &lt;dirent.h&gt;

#include &lt;sys/stat.h&gt;

#include &lt;sys/types.h&gt;

void removeDirectory(const char \*path) {

DIR \*dir = opendir(path);

struct dirent \*entry;

if (dir == NULL) {

perror(&quot;opendir&quot;);

return;

}

while ((entry = readdir(dir)) != NULL) {

if (strcmp(entry-&gt;d\_name, &quot;.&quot;) == 0 || strcmp(entry-&gt;d\_name, &quot;..&quot;) == 0) {

continue;

}

char childPath[PATH\_MAX];

snprintf(childPath, sizeof(childPath), &quot;%s/%s&quot;, path, entry-&gt;d\_name);

struct stat st;

if (stat(childPath, &amp;st) == -1) {

perror(&quot;stat&quot;);

continue;

}

if (S\_ISDIR(st.st\_mode)) {

removeDirectory(childPath);

} else {

if (unlink(childPath) == -1) {

perror(&quot;unlink&quot;);

continue;

}

}

}

closedir(dir);

if (rmdir(path) == -1) {

perror(&quot;rmdir&quot;);

}

}

int main() {

const char \*path = &quot;/path/to/directory&quot;;

removeDirectory(path);

return 0;

}

**18. C program to implement UNIX system call, CP and CAT.**

#include &lt;stdio.h&gt;

#include &lt;stdlib.h&gt;

#include &lt;fcntl.h&gt;

#include &lt;unistd.h&gt;

#define BUFFER\_SIZE 4096

void copyFile(const char \*source, const char \*destination) {

int source\_fd = open(source, O\_RDONLY);

if (source\_fd == -1) {

perror(&quot;open&quot;);

return;

}

int destination\_fd = open(destination, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (destination\_fd == -1) {

perror(&quot;open&quot;);

close(source\_fd);

return;

}

char buffer[BUFFER\_SIZE];

ssize\_t bytesRead;

while ((bytesRead = read(source\_fd, buffer, BUFFER\_SIZE)) &gt; 0) {

ssize\_t bytesWritten = write(destination\_fd, buffer, bytesRead);

if (bytesWritten == -1) {

perror(&quot;write&quot;);

break;

}

}

if (bytesRead == -1) {

perror(&quot;read&quot;);

}

close(source\_fd);

close(destination\_fd);

}

void concatenateFiles(const char \*files[], int numFiles, const char \*destination) {

int destination\_fd = open(destination, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (destination\_fd == -1) {

perror(&quot;open&quot;);

return;

}

char buffer[BUFFER\_SIZE];

for (int i = 0; i &lt; numFiles; i++) {

int source\_fd = open(files[i], O\_RDONLY);

if (source\_fd == -1) {

perror(&quot;open&quot;);

continue;

}

ssize\_t bytesRead;

while ((bytesRead = read(source\_fd, buffer, BUFFER\_SIZE)) &gt; 0) {

ssize\_t bytesWritten = write(destination\_fd, buffer, bytesRead);

if (bytesWritten == -1) {

perror(&quot;write&quot;);

break;

}

}

if (bytesRead == -1) {

perror(&quot;read&quot;);

}

close(source\_fd);

}

close(destination\_fd);

}

int main() {

// CP Command

const char \*sourceFile = &quot;source.txt&quot;;

const char \*destinationFile = &quot;destination.txt&quot;;

copyFile(sourceFile, destinationFile);

printf(&quot;File copied successfully.\n&quot;);

// CAT Command

const char \*files[] = {&quot;file1.txt&quot;, &quot;file2.txt&quot;, &quot;file3.txt&quot;};

int numFiles = sizeof(files) / sizeof(files[0]);

const char \*concatenatedFile = &quot;concatenated.txt&quot;;

concatenateFiles(files, numFiles, concatenatedFile);

printf(&quot;Files concatenated successfully.\n&quot;);

return 0;

}