**SAVEETHA SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES COMPUTER SCIENCE AND ENGINEERING PROGRAMME**

**CSA04 OPERATING SYSTEMS**

**LIST OF PROGRAMS**

1. Consider a system with 4 processes and 3 resources with the given resource matrices.

Claim matrix Allocation matrix

3 2 2 1 0 0

6 1 3 6 1 2

3 1 4 2 1 1

4 2 2 0 0 2

The resource vector is [9,3,6]. Write a C program to determine if the system is in safe or unsafe state.

#include <stdio.h>

#define MAX\_PROCESSES 4

#define MAX\_RESOURCES 3

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

int isSafe(int available[MAX\_RESOURCES], int max[MAX\_PROCESSES][MAX\_RESOURCES], int allocation[MAX\_PROCESSES][MAX\_RESOURCES]) {

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

int finish[MAX\_PROCESSES] = {0}; // Initialization: No process is finished

// Calculate the need matrix

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

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

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

}

}

// Try to find a safe sequence

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

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

if (!finish[i]) {

int safe = 1; // Assume the process is safe

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

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

safe = 0; // Unsafe state

break;

}

}

if (safe) {

// Process i can be completed

finish[i] = 1;

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

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

}

}

}

}

}

// Check if all processes are finished

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

if (!finish[i]) {

return 0; // Unsafe state

}

}

return 1; // Safe state

}

int main() {

int claim[MAX\_PROCESSES][MAX\_RESOURCES] = {{3, 2, 2}, {6, 1, 3}, {3, 1, 4}, {4, 2, 2}};

int allocation[MAX\_PROCESSES][MAX\_RESOURCES] = {{1, 0, 0}, {6, 1, 2}, {2, 1, 1}, {0, 0, 2}};

int available[MAX\_RESOURCES] = {9, 3, 6};

if (isSafe(available, claim, allocation)) {

printf("The system is in a safe state.\n");

} else {

printf("The system is in an unsafe state.\n");

}

return 0;

}

1. Write a C program to illustrate the FIFO method of page replacement and determine the number of page faults for the following test case:

No of page frames: 3; Page reference sequence: 4, 1, 2, 4, 3, 2, 1 and 5.

#include <stdio.h>

// Function to check if a page is present in the page frames

int isPagePresent(int page, int frames[], int numFrames) {

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

if (frames[i] == page) {

return 1; // Page is present

}

}

return 0; // Page is not present

}

// Function to find the index of the oldest page in the page frames

int findOldestPageIndex(int page, int frames[], int numFrames) {

int oldestIndex = 0;

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

if (frames[i] < frames[oldestIndex]) {

oldestIndex = i;

}

}

return oldestIndex;

}

// Function to simulate the FIFO page replacement algorithm

int fifoPageReplacement(int pages[], int numPages, int numFrames) {

int frames[numFrames];

int pageFaults = 0;

// Initialize frames to -1 (indicating an empty frame)

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

frames[i] = -1;

}

// Simulate page references

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

int page = pages[i];

if (!isPagePresent(page, frames, numFrames)) {

// Page fault, replace the oldest page

int replaceIndex = findOldestPageIndex(page, frames, numFrames);

frames[replaceIndex] = page;

++pageFaults;

// Print the current state of page frames after a page fault

printf("Page Fault #%d: [", pageFaults);

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

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

if (j < numFrames - 1) {

printf(", ");

}

}

printf("]\n");

}

}

return pageFaults;

}

int main() {

int numFrames = 3;

int numPages = 8;

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

printf("Page Reference Sequence: [");

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

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

if (i < numPages - 1) {

printf(", ");

}

}

printf("]\n");

int pageFaults = fifoPageReplacement(pageReferenceSequence, numPages, numFrames);

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

return 0;

}

1. Write a program to compute the average waiting time and average turnaround time based on Non Preemptive Shortest-Job-First Scheduling for the following process with the given CPU burst times, ( and the assumption that all jobs arrive at the same time.)

Process Burst Time

P1 6

P2 8

P3 7

P4 3

#include <stdio.h>

// Structure to represent a process

struct Process {

char id; // Process ID

int burstTime; // Burst time of the process

};

// Function to perform Non-Preemptive Shortest-Job-First Scheduling

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

// Sorting the processes based on burst time

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

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

if (processes[j].burstTime > processes[j + 1].burstTime) {

// Swap processes[j] and processes[j + 1]

struct Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

// Function to calculate average waiting time and average turnaround time

void calculateAverageTimes(struct Process processes[], int n, float \*averageWaitingTime, float \*averageTurnaroundTime) {

int waitingTime = 0; // Cumulative waiting time

int turnaroundTime = 0; // Cumulative turnaround time

// Calculate waiting time and turnaround time for each process

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

turnaroundTime += processes[i].burstTime;

waitingTime += turnaroundTime - processes[i].burstTime;

printf("Process %c:\n", processes[i].id);

printf(" Waiting Time: %d\n", waitingTime - processes[i].burstTime);

printf(" Turnaround Time: %d\n", turnaroundTime);

}

// Calculate averages

\*averageWaitingTime = (float)waitingTime / n;

\*averageTurnaroundTime = (float)turnaroundTime / n;

}

int main() {

// Given processes and their burst times

struct Process processes[] = {

{'P1', 6},

{'P2', 8},

{'P3', 7},

{'P4', 3},

};

int numProcesses = sizeof(processes) / sizeof(processes[0]);

// Perform Non-Preemptive Shortest-Job-First Scheduling

sjfScheduling(processes, numProcesses);

// Calculate average waiting time and average turnaround time

float averageWaitingTime, averageTurnaroundTime;

calculateAverageTimes(processes, numProcesses, &averageWaitingTime, &averageTurnaroundTime);

// Display the results

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

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

return 0;

}

1. Write a C program to implement the first-fit algorithm for memory management.

Test Case:

Memory partitions: 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order) Show the outcome for the test case with first-fit algorithms to place the processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)

#include <stdio.h>

#define NUM\_PARTITIONS 6

// Function to allocate memory using the first-fit algorithm

void firstFit(int memory[], int numMemoryPartitions, int processSize[], int numProcesses) {

int allocation[numProcesses];

// Initialize allocation array to -1 (indicating not allocated)

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

allocation[i] = -1;

}

// Allocate memory for each process using the first-fit algorithm

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

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

if (memory[j] >= processSize[i]) {

// Allocate memory in the first partition that fits the process

allocation[i] = j;

memory[j] -= processSize[i];

break;

}

}

}

// Display the outcome

printf("\nOutcome for First-Fit Algorithm:\n");

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

if (allocation[i] != -1) {

printf("Process %d: Allocated to Partition %d\n", i + 1, allocation[i] + 1);

} else {

printf("Process %d: Not Allocated\n", i + 1);

}

}

// Display the final state of memory partitions

printf("\nFinal State of Memory Partitions:\n");

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

printf("Partition %d: %d KB\n", i + 1, memory[i]);

}

}

int main() {

int memoryPartitions[NUM\_PARTITIONS] = {300, 600, 350, 200, 750, 125};

int numProcesses = 5;

int processSize[] = {115, 500, 358, 200, 375};

printf("Memory Partitions:\n");

for (int i = 0; i < NUM\_PARTITIONS; ++i) {

printf("Partition %d: %d KB\n", i + 1, memoryPartitions[i]);

}

printf("\nProcesses to Allocate:\n");

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

printf("Process %d: %d KB\n", i + 1, processSize[i]);

}

firstFit(memoryPartitions, NUM\_PARTITIONS, processSize, numProcesses);

return 0;

}

5. Write a program to compute the average waiting time and turnaround time based on Preemptive shortest remaining processing time first (SRPT) algorithm for the following set of processes, with the arrival times and the CPU-burst times given in milliseconds

Process Arrival Time Burst Time

P1 0 5

P2 1 3

P3 2 3

P4 4 1

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int processID;

int arrivalTime;

int burstTime;

int remainingTime;

int startTime;

int endTime;

int waitingTime;

int turnaroundTime;

} Process;

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

int currentTime = 0;

int completedProcesses = 0;

while (completedProcesses < n) {

int shortestRemainingTime = -1;

int shortestIndex = -1;

int im1;

for ( im1 = 0; im1 < n; im1++) {

if (processes[im1].arrivalTime <= currentTime && processes[im1].remainingTime > 0) {

if (shortestRemainingTime == -1 || processes[im1].remainingTime < shortestRemainingTime) {

shortestRemainingTime = processes[im1].remainingTime;

shortestIndex = im1;

}

}

}

if (shortestIndex == -1) {

currentTime++;

} else {

processes[shortestIndex].remainingTime--;

currentTime++;

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

processes[shortestIndex].endTime = currentTime;

processes[shortestIndex].turnaroundTime = processes[shortestIndex].endTime - processes[shortestIndex].arrivalTime;

processes[shortestIndex].waitingTime = processes[shortestIndex].turnaroundTime - processes[shortestIndex].burstTime;

completedProcesses++;

}

}

}

}

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

float totalWaitingTime = 0, totalTurnaroundTime = 0;

int i;

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

float averageWaitingTime = totalWaitingTime / n;

float averageTurnaroundTime = totalTurnaroundTime / n;

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

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

}

int main() {

Process processes[] = {

{1, 0, 5, 5, 0, 0, 0, 0},

{2, 1, 3, 3, 0, 0, 0, 0},

{3, 2, 3, 3, 0, 0, 0, 0},

{4, 4, 1, 1, 0, 0, 0, 0}

};

int numProcesses = sizeof(processes) / sizeof(processes[0]);

printf("Preemptive Shortest Remaining Processing Time First (SRPT) Scheduling Algorithm\n\n");

srpt(processes, numProcesses);

printf("Process\tWaiting Time\tTurnaround Time\n");

int i;

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

printf("P%d\t%d\t\t%d\n", processes[i].processID, processes[i].waitingTime, processes[i].turnaroundTime);

}

calculateAverageTimes(processes, numProcesses);

return 0;

}

6. Write a C program to implement the deadlock detection algorithm for a system with 3 processes and 3 resource instances and the resource matrices are given below.

Max Matrix Allocation Matrix

3 6 8 3 3 3

4 3 3 2 0 3

3 4 4 1 2 4

The number of available resources is [1,2,0]. Determine if the system is in a deadlock state and identify the deadlocked processes.

#include <stdio.h>

#define MAX\_PROCESSES 3

#define MAX\_RESOURCES 3

void detectDeadlock(int maxMatrix[MAX\_PROCESSES][MAX\_RESOURCES], int allocationMatrix[MAX\_PROCESSES][MAX\_RESOURCES], int availableResources[MAX\_RESOURCES]) {

int marked[MAX\_PROCESSES] = {0};

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

int im2,j;

for ( im2 = 0; im2 < MAX\_PROCESSES; im2++) {

for ( j = 0; j < MAX\_RESOURCES; j++) {

needMatrix[im2][j] = maxMatrix[im2][j] - allocationMatrix[im2][j];

}

}

int found;

do {

found = 0;

int i;

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

if (!marked[i]) {

int j;

for (j = 0; j < MAX\_RESOURCES; j++) {

if (needMatrix[i][j] > availableResources[j]) {

break;

}

}

if (j == MAX\_RESOURCES) {

marked[i] = 1;

found = 1;

int k;

for ( k = 0; k < MAX\_RESOURCES; k++) {

availableResources[k] += allocationMatrix[i][k];

}

}

}

}

} while (found);

int deadlockDetected = 0;

printf("Deadlocked Processes: ");

for ( im2 = 0; im2 < MAX\_PROCESSES; im2++) {

if (!marked[im2]) {

printf("P%d ", im2 + 1);

deadlockDetected = 1;

}

}

if (deadlockDetected) {

printf("\nDeadlock Detected!\n");

} else {

printf("\nNo Deadlock Detected.\n");

}

}

int main() {

// Given matrices

int maxMatrix[MAX\_PROCESSES][MAX\_RESOURCES] = {

{3, 6, 8},

{4, 3, 3},

{3, 4, 4}

};

int allocationMatrix[MAX\_PROCESSES][MAX\_RESOURCES] = {

{3, 3, 3},

{2, 0, 3},

{1, 2, 4}

};

int availableResources[MAX\_RESOURCES] = {1, 2, 0};

// Perform deadlock detection and identify deadlocked processes

detectDeadlock(maxMatrix, allocationMatrix, availableResources);

return 0;

}

7. Write a C program to illustrate the page replacement method where the current least recently used element is replaced and determine the number of page faults for the following test case:

No. of page frames: 3; Page reference sequence 1,2,3,2,1,5,2,1,6,2,5,6,3,1,3,6,1,2,4 and 3.

#include <stdio.h>

#define MAX\_PAGE\_FRAMES 3

int findIndex(int array[], int size, int element) {

int im3;

for ( im3 = 0; im3 < size; im3++) {

if (array[im3] == element) {

return im3;

}

}

return -1;

}

int lruPageReplacement(int pageReferenceSequence[], int n, int pageFrames) {

int pageFaults = 0;

int pageFrame[MAX\_PAGE\_FRAMES] = {-1};

int i;

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

int pageIndex = findIndex(pageFrame, pageFrames, pageReferenceSequence[i]);

if (pageIndex == -1) {

pageFaults++;

int lruIndex = -1;

int lruTime = n;

int j;

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

int index = findIndex(pageReferenceSequence, n, pageFrame[j]);

if (index < lruTime) {

lruTime = index;

lruIndex = j;

}

}

pageFrame[lruIndex] = pageReferenceSequence[i];

}

}

return pageFaults;

}

int main() {

int pageFrames = 3;

int pageReferenceSequence[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3, 6, 1, 2, 4, 3};

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

int pageFaults = lruPageReplacement(pageReferenceSequence, n, pageFrames);

printf("Number of Page Faults using LRU: %d\n", pageFaults);

return 0;

}

8. Write a C program to simulate FCFS disk scheduling algorithm and execute your program and find the average head movement with the following test case:

No of tracks 5; Track position:55 58 60 70 18

#include <stdio.h>

#include <stdlib.h>

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

return abs(a - b);

}

void fcfs\_disk\_scheduling(int tracks[], int n, int initial\_position) {

int total\_head\_movement = 0;

int current\_position = initial\_position;

int im4;

for ( im4 = 0; im4 < n; im4++) {

int movement = abs\_diff(current\_position, tracks[im4]);

printf("Move from %d to %d: %d\n", current\_position, tracks[im4], movement);

total\_head\_movement += movement;

current\_position = tracks[im4];

}

printf("Total Head Movement: %d\n", total\_head\_movement);

printf("Average Head Movement: %.2f\n", (float) total\_head\_movement / n);

}

int main() {

int tracks[] = {55, 58, 60, 70, 18};

int num\_tracks = sizeof(tracks) / sizeof(tracks[0]);

int initial\_position = 50;

fcfs\_disk\_scheduling(tracks, num\_tracks, initial\_position);

return 0;

}

9. Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8-time units. All processes arrive at time zero. Write a program to compute the average waiting time and average turnaround time based on First Come First Serve scheduling

#include <stdio.h>

// Structure to represent a process

struct Process {

int id; // Process ID

int burstTime; // Burst time of the process

};

// Function to calculate average waiting time and average turnaround time

void calculateAverageTimes(struct Process processes[], int n, float \*averageWaitingTime, float \*averageTurnaroundTime) {

int waitingTime = 0; // Cumulative waiting time

int turnaroundTime = 0; // Cumulative turnaround time

// Calculate waiting time and turnaround time for each process

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

waitingTime += turnaroundTime;

turnaroundTime += processes[i].burstTime;

printf("Process %d:\n", processes[i].id);

printf(" Waiting Time: %d\n", waitingTime);

printf(" Turnaround Time: %d\n", turnaroundTime);

}

// Calculate averages

\*averageWaitingTime = (float)waitingTime / n;

\*averageTurnaroundTime = (float)turnaroundTime / n;

}

int main() {

// Given processes and their burst times

struct Process processes[] = {

{0, 2},

{1, 4},

{2, 8},

};

int numProcesses = sizeof(processes) / sizeof(processes[0]);

// Calculate average waiting time and average turnaround time

float averageWaitingTime, averageTurnaroundTime;

calculateAverageTimes(processes, numProcesses, &averageWaitingTime, &averageTurnaroundTime);

// Display the results

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

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

return 0;

}

10. Consider the following process table with number of processes that contains allocation field (for showing the number of resources of type: A, B and C allocated to each process in the table), max field (for showing the maximum number of resources of type: A, B, and C that can be allocated to each process). Write a program to calculate the entries of need matrix using the formula: (Need)i = (Max)i - (Allocation)i

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Allocation | Max | Availble |
|  | A B C | A B C | A B C |
| P0 | 1 1 2 | 5 4 4 | 3 2 1 |
| P1 | 2 1 2 | 4 3 3 |  |
| P2 | 3 0 1 | 9 1 3 |  |
| P3 | 0 2 0 | 8 6 4 |  |
| P4 | 1 1 2 | 2 2 3 |  |

#include <stdio.h>

#define NUM\_PROCESSES 5

#define NUM\_RESOURCES 3

// Function to calculate the entries of the need matrix

void calculateNeedMatrix(int allocation[][NUM\_RESOURCES], int max[][NUM\_RESOURCES], int need[][NUM\_RESOURCES]) {

for (int i = 0; i < NUM\_PROCESSES; ++i) {

for (int j = 0; j < NUM\_RESOURCES; ++j) {

// Need = Max - Allocation

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

}

}

}

int main() {

int allocation[NUM\_PROCESSES][NUM\_RESOURCES] = {

{1, 1, 2},

{2, 1, 2},

{3, 0, 1},

{0, 2, 0},

{1, 1, 2},

};

int max[NUM\_PROCESSES][NUM\_RESOURCES] = {

{5, 4, 4},

{4, 3, 3},

{9, 1, 3},

{8, 6, 4},

{2, 2, 3},

};

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

// Calculate the entries of the need matrix

calculateNeedMatrix(allocation, max, need);

// Display the Need matrix

printf("Need Matrix:\n");

printf(" A B C\n");

for (int i = 0; i < NUM\_PROCESSES; ++i) {

printf("P%d", i);

for (int j = 0; j < NUM\_RESOURCES; ++j) {

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

}

printf("\n");

}

return 0;

}

11. Write a C program to create 4 child processes. In the first child process, print the odd numbers. In the second child process print the even numbers. In the third child process print the multiple of 3. In the fourth child process print the multiples of 5. Print the process id for each of the processes.

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

void printOddNumbers() {

printf("Odd Numbers: ");

for (int i = 1; i <= 10; i += 2) {

printf("%d ", i);

}

printf("\n");

}

void printEvenNumbers() {

printf("Even Numbers: ");

for (int i = 2; i <= 10; i += 2) {

printf("%d ", i);

}

printf("\n");

}

void printMultiplesOfThree() {

printf("Multiples of 3: ");

for (int i = 3; i <= 30; i += 3) {

printf("%d ", i);

}

printf("\n");

}

void printMultiplesOfFive() {

printf("Multiples of 5: ");

for (int i = 5; i <= 50; i += 5) {

printf("%d ", i);

}

printf("\n");

}

int main() {

// Create four child processes

for (int i = 0; i < 4; ++i) {

pid\_t pid = fork();

if (pid == 0) { // Child process

switch (i) {

case 0:

printf("Child Process %d (PID: %d): ", i + 1, getpid());

printOddNumbers();

break;

case 1:

printf("Child Process %d (PID: %d): ", i + 1, getpid());

printEvenNumbers();

break;

case 2:

printf("Child Process %d (PID: %d): ", i + 1, getpid());

printMultiplesOfThree();

break;

case 3:

printf("Child Process %d (PID: %d): ", i + 1, getpid());

printMultiplesOfFive();

break;

}

return 0; // Terminate child process

} else if (pid < 0) {

fprintf(stderr, "Error in forking process\n");

return 1;

}

}

// Wait for all child processes to complete

for (int i = 0; i < 4; ++i) {

wait(NULL);

}

return 0;

}

12. Write a C program to implement the best-fit algorithm and allocate the memory block to each process.

Test Case:

Memory partitions: 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order),

Show the outcome for the test case with the best-fit algorithms to place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)

#include <stdio.h>

#define MAX\_PARTITIONS 6

#define MAX\_PROCESSES 5

// Function to allocate memory using best-fit algorithm

void bestFit(int partitions[], int m, int processes[], int n) {

int allocation[MAX\_PROCESSES];

// Initialize all allocations to -1, indicating unallocated

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

allocation[i] = -1;

}

// Iterate through each process and find the best-fit block

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

int bestFitIndex = -1;

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

if (partitions[j] >= processes[i]) {

if (bestFitIndex == -1 || partitions[j] < partitions[bestFitIndex]) {

bestFitIndex = j;

}

}

}

// If a suitable block is found, allocate the process to that block

if (bestFitIndex != -1) {

allocation[i] = bestFitIndex;

partitions[bestFitIndex] -= processes[i];

}

}

// Display the memory allocation

printf("\nProcess\t\tMemory Block\n");

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

printf("%d\t\t", i + 1);

if (allocation[i] != -1) {

printf("%d KB\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

int main() {

// Memory partitions

int partitions[MAX\_PARTITIONS] = {300, 600, 350, 200, 750, 125};

// Processes

int processes[MAX\_PROCESSES] = {115, 500, 358, 200, 375};

// Display initial memory partitions

printf("Initial Memory Partitions:\n");

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

printf("%d KB ", partitions[i]);

}

// Allocate memory using best-fit algorithm

bestFit(partitions, MAX\_PARTITIONS, processes, MAX\_PROCESSES);

return 0;

}

13. Write a C program to implement single-level directory system. In which all the files are placed in one directory and there are no sub directories.

Test Case: Create one directory with the name of CSE and Add 3 files(A,B,C) in to that directory

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_FILES 3

#define MAX\_FILENAME\_LENGTH 20

#define MAX\_DIRECTORY\_NAME\_LENGTH 20

// Structure to represent a file

typedef struct {

char filename[MAX\_FILENAME\_LENGTH];

} File;

// Structure to represent a directory

typedef struct {

char dirname[MAX\_DIRECTORY\_NAME\_LENGTH];

File files[MAX\_FILES];

int fileCount;

} Directory;

// Function to create a directory

Directory createDirectory(const char \*dirname) {

Directory dir;

strcpy(dir.dirname, dirname);

dir.fileCount = 0;

return dir;

}

// Function to add a file to a directory

void addFileToDirectory(Directory \*dir, const char \*filename) {

if (dir->fileCount < MAX\_FILES) {

strcpy(dir->files[dir->fileCount].filename, filename);

dir->fileCount++;

printf("File %s added to directory %s.\n", filename, dir->dirname);

} else {

printf("Directory %s is full. Cannot add more files.\n", dir->dirname);

}

}

// Function to display the contents of a directory

void displayDirectory(const Directory \*dir) {

printf("Directory: %s\n", dir->dirname);

printf("Files:\n");

for (int i = 0; i < dir->fileCount; i++) {

printf("- %s\n", dir->files[i].filename);

}

}

int main() {

// Create a directory named "CSE"

Directory cseDirectory = createDirectory("CSE");

// Add files A, B, and C to the "CSE" directory

addFileToDirectory(&cseDirectory, "A");

addFileToDirectory(&cseDirectory, "B");

addFileToDirectory(&cseDirectory, "C");

// Display the contents of the "CSE" directory

displayDirectory(&cseDirectory);

return 0;

}

14. Write a C program to illustrate the page replacement method where the page which is not in demand for the longest future time is replaced by the new page and determine the number of page faults for the following test case:

No. of page frames: 3; Page reference sequence 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0 and 1.

#include<stdio.h>

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter page reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

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

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

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

faults++;

frames[j] = pages[i];

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

flag3 =0;

for(j = 0; j < no\_of\_frames; ++j){

temp[j] = -1;

for(k = i + 1; k < no\_of\_pages; ++k){

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

temp[j] = k;

break;

}

}

}

for(j = 0; j < no\_of\_frames; ++j){

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

pos = j;

flag3 = 1;

break;

}

}

if(flag3 ==0){

max = temp[0];

pos = 0;

for(j = 1; j < no\_of\_frames; ++j){

if(temp[j] > max){

max = temp[j];

pos = j;

}

}

}

frames[pos] = pages[i];

faults++;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

printf("%d\t", frames[j]);

}

}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

15.Write a C program to simulate FCFS disk scheduling algorithms and execute your program and find out and print the average head movement for the following test case.

No of tracks:9; Track position:55 58 60 70 18 90 150 160 184

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

// logic for FCFS disk scheduling

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

printf("Total head moment is %d",TotalHeadMoment);

return 0;

}

16. Write a program to compute the average waiting time and average turnaround time based on First Come First Serve for the following process with the given CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process Burst Time

P1 10

P2 15

P3 25

#include <stdio.h>

#define MAX\_PROCESSES 3

// Function to calculate average waiting time and average turnaround time

void calculateAvgTimes(int burstTimes[], int n, float \*avgWaitingTime, float \*avgTurnaroundTime) {

int waitingTime[MAX\_PROCESSES], turnaroundTime[MAX\_PROCESSES];

// Calculate waiting time for each process

waitingTime[0] = 0; // Waiting time for the first process is always 0

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

waitingTime[i] = waitingTime[i - 1] + burstTimes[i - 1];

}

// Calculate turnaround time for each process

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

turnaroundTime[i] = waitingTime[i] + burstTimes[i];

}

// Calculate average waiting time and average turnaround time

\*avgWaitingTime = 0;

\*avgTurnaroundTime = 0;

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

\*avgWaitingTime += waitingTime[i];

\*avgTurnaroundTime += turnaroundTime[i];

}

\*avgWaitingTime /= n;

\*avgTurnaroundTime /= n;

}

int main() {

// Process burst times

int burstTimes[MAX\_PROCESSES] = {10, 15, 25};

// Calculate average waiting time and average turnaround time

float avgWaitingTime, avgTurnaroundTime;

calculateAvgTimes(burstTimes, MAX\_PROCESSES, &avgWaitingTime, &avgTurnaroundTime);

// Display the results

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

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

return 0;

}

17. Write a program to compute the average waiting time and average turnaround time based on Round Robin scheduling for the following process with the given CPU burst times and quantum time slots 4 ms, ( and the assumption that all jobs arrive at the same time.)

Process Burst Time

P1 24

P2 3

P3 3

#include <stdio.h>

#define MAX\_PROCESSES 3

// Function to calculate average waiting time and average turnaround time

void calculateAvgTimes(int burstTimes[], int n, int quantum, float \*avgWaitingTime, float \*avgTurnaroundTime) {

int remainingTime[MAX\_PROCESSES], waitingTime[MAX\_PROCESSES], turnaroundTime[MAX\_PROCESSES];

// Initialize remaining time to burst time

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

remainingTime[i] = burstTimes[i];

}

int currentTime = 0;

// Keep track of whether a process has been fully executed

int executed[MAX\_PROCESSES] = {0};

// Perform round robin scheduling

while (1) {

int allProcessesExecuted = 1;

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

if (remainingTime[i] > 0) {

allProcessesExecuted = 0;

if (remainingTime[i] > quantum) {

// Process executes for the time quantum

currentTime += quantum;

remainingTime[i] -= quantum;

} else {

// Process completes execution

currentTime += remainingTime[i];

waitingTime[i] = currentTime - burstTimes[i];

remainingTime[i] = 0;

executed[i] = 1;

}

}

}

if (allProcessesExecuted)

break;

}

// Calculate turnaround time for each process

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

turnaroundTime[i] = waitingTime[i] + burstTimes[i];

}

// Calculate average waiting time and average turnaround time

\*avgWaitingTime = 0;

\*avgTurnaroundTime = 0;

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

\*avgWaitingTime += waitingTime[i];

\*avgTurnaroundTime += turnaroundTime[i];

}

\*avgWaitingTime /= n;

\*avgTurnaroundTime /= n;

}

int main() {

// Process burst times

int burstTimes[MAX\_PROCESSES] = {24, 3, 3};

// Time quantum

int quantum = 4;

// Calculate average waiting time and average turnaround time

float avgWaitingTime, avgTurnaroundTime;

calculateAvgTimes(burstTimes, MAX\_PROCESSES, quantum, &avgWaitingTime, &avgTurnaroundTime);

// Display the results

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

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

return 0;

}

18. Write a program for solving the producer consumer problem with the following scenario: The producer should produce data only when the buffer is not full. Data can only be consumed by the consumer if and only if the memory buffer is not empty.

Test Case:

Buffer Size: 3

Consume an item in the beginning and show that the buffer is EMPTY

Produce 4 items and show that the buffer is FULL

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 3

// Buffer and semaphores

int buffer[BUFFER\_SIZE];

sem\_t empty, full;

pthread\_mutex\_t mutex;

// Index for producer and consumer

int in = 0, out = 0;

// Function prototypes

void \*producer(void \*arg);

void \*consumer(void \*arg);

void printBufferStatus();

int main() {

// Initialize semaphores and mutex

sem\_init(&empty, 0, BUFFER\_SIZE);

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

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

// Create producer and consumer threads

pthread\_t producerThread, consumerThread;

pthread\_create(&producerThread, NULL, producer, NULL);

pthread\_create(&consumerThread, NULL, consumer, NULL);

// Join threads

pthread\_join(producerThread, NULL);

pthread\_join(consumerThread, NULL);

// Clean up

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

void \*producer(void \*arg) {

for (int i = 0; i < 4; i++) {

// Wait if the buffer is full

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

// Produce item

buffer[in] = i + 1;

in = (in + 1) % BUFFER\_SIZE;

printBufferStatus();

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

}

return NULL;

}

void \*consumer(void \*arg) {

// Wait for the buffer to be non-empty

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

// Consume item

int item = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

return NULL;

}

void printBufferStatus() {

printf("Producer: Produced item. Buffer: [ ");

for (int i = 0; i < BUFFER\_SIZE; i++) {

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

}

printf("]\n");

}

19. Write a C program to create two threads to access shared memory which is an integer in a synchronized fashion using semaphore. In the first thread print the doubled the integer data after reading from the shared memory. In the second thread, print the five times of the integer data after reading from the shared memory

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

// Shared data

int sharedData = 0;

// Semaphores

sem\_t semaphore1, semaphore2;

// Function prototypes

void \*thread1(void \*arg);

void \*thread2(void \*arg);

int main() {

// Initialize semaphores

sem\_init(&semaphore1, 0, 0);

sem\_init(&semaphore2, 0, 0);

// Create threads

pthread\_t threadID1, threadID2;

pthread\_create(&threadID1, NULL, thread1, NULL);

pthread\_create(&threadID2, NULL, thread2, NULL);

// Wait for both threads to finish

pthread\_join(threadID1, NULL);

pthread\_join(threadID2, NULL);

// Clean up

sem\_destroy(&semaphore1);

sem\_destroy(&semaphore2);

return 0;

}

void \*thread1(void \*arg) {

// Access shared memory (read)

sem\_wait(&semaphore1);

// Print doubled value

printf("Thread 1: Doubled value: %d\n", sharedData \* 2);

// Release semaphore2 to allow thread2 to proceed

sem\_post(&semaphore2);

return NULL;

}

void \*thread2(void \*arg) {

// Access shared memory (read)

sem\_wait(&semaphore2);

// Print five times value

printf("Thread 2: Five times value: %d\n", sharedData \* 5);

return NULL;

}

20. Write a C program to implement the worst-fit algorithm and allocate the memory block to each process.

Test Case:

Memory partitions: 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order),

Show the outcome for the test case with the worst-fit algorithms to place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)

#include <stdio.h>

#define MAX\_PARTITIONS 6

#define MAX\_PROCESSES 5

// Function to allocate memory using worst-fit algorithm

void worstFit(int partitions[], int m, int processes[], int n) {

int allocation[MAX\_PROCESSES];

// Initialize all allocations to -1, indicating unallocated

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

allocation[i] = -1;

}

// Iterate through each process and find the worst-fit block

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

int worstFitIndex = -1;

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

if (partitions[j] >= processes[i]) {

if (worstFitIndex == -1 || partitions[j] > partitions[worstFitIndex]) {

worstFitIndex = j;

}

}

}

// If a suitable block is found, allocate the process to that block

if (worstFitIndex != -1) {

allocation[i] = worstFitIndex;

partitions[worstFitIndex] -= processes[i];

}

}

// Display the memory allocation

printf("\nProcess\t\tMemory Block\n");

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

printf("%d\t\t", i + 1);

if (allocation[i] != -1) {

printf("%d KB\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

int main() {

// Memory partitions

int partitions[MAX\_PARTITIONS] = {300, 600, 350, 200, 750, 125};

// Processes

int processes[MAX\_PROCESSES] = {115, 500, 358, 200, 375};

// Display initial memory partitions

printf("Initial Memory Partitions:\n");

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

printf("%d KB ", partitions[i]);

}

// Allocate memory using worst-fit algorithm

worstFit(partitions, MAX\_PARTITIONS, processes, MAX\_PROCESSES);

return 0;

}

21. Write a program to simulate the Dining Philosopher problem and verify your output with the following test case:

No of Philosophers: 5

THINKING – When philosopher doesn’t want to gain access to either fork.

HUNGRY – When philosopher wants to enter the critical section.

EATING – When philosopher has got both the forks, i.e., he has entered the section.

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating

(state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

sem\_t room;

sem\_t chopstick[5];

void \* philosopher(void \*);

void eat(int);

int main()

{

int i,a[5];

pthread\_t tid[5];

sem\_init(&room,0,4);

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

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

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

a[i]=i;

pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]);

}

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

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

}

void \* philosopher(void \* num)

{

int phil=\*(int \*)num;

sem\_wait(&room);

printf("\nPhilosopher %d has entered room",phil);

sem\_wait(&chopstick[phil]);

sem\_wait(&chopstick[(phil+1)%5]);

eat(phil);

sleep(2);

printf("\nPhilosopher %d has finished eating",phil);

sem\_post(&chopstick[(phil+1)%5]);

sem\_post(&chopstick[phil]);

sem\_post(&room);

}

void eat(int phil)

{

printf("\nPhilosopher %d is eating",phil);

}

22. Write a C program to implement the two-level directory system.

Test Case:

3 user directories have to be created with name of user1, user2, user3 and also to create 3 files with user1 directory,2 files with user2 and user3 directory

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_USERS 3

#define MAX\_FILES\_PER\_USER 3

// Structure to represent a file

typedef struct {

char filename[20];

} File;

// Structure to represent a user directory

typedef struct {

char username[20];

File files[MAX\_FILES\_PER\_USER];

int fileCount;

} UserDirectory;

// Function to create a user directory

UserDirectory createUserDirectory(const char \*username) {

UserDirectory userDir;

strcpy(userDir.username, username);

userDir.fileCount = 0;

return userDir;

}

// Function to add a file to a user directory

void addFileToUserDirectory(UserDirectory \*userDir, const char \*filename) {

if (userDir->fileCount < MAX\_FILES\_PER\_USER) {

strcpy(userDir->files[userDir->fileCount].filename, filename);

userDir->fileCount++;

printf("File %s added to directory %s.\n", filename, userDir->username);

} else {

printf("Directory %s is full. Cannot add more files.\n", userDir->username);

}

}

int main() {

// Create user directories

UserDirectory userDirs[MAX\_USERS];

userDirs[0] = createUserDirectory("user1");

userDirs[1] = createUserDirectory("user2");

userDirs[2] = createUserDirectory("user3");

// Add files to user directories

addFileToUserDirectory(&userDirs[0], "file1\_user1");

addFileToUserDirectory(&userDirs[0], "file2\_user1");

addFileToUserDirectory(&userDirs[0], "file3\_user1");

addFileToUserDirectory(&userDirs[1], "file1\_user2");

addFileToUserDirectory(&userDirs[1], "file2\_user2");

addFileToUserDirectory(&userDirs[2], "file1\_user3");

addFileToUserDirectory(&userDirs[2], "file2\_user3");

return 0;

}

23. Write a C program to simulate SCAN disk scheduling algorithms. and execute your program and find out and print the average head movement for the following test case.

No of tracks:5; Track position:55 58 60 70 18

#include <stdio.h>

#include <stdlib.h>

// Function to perform SCAN disk scheduling

void scanDisk(int tracks[], int numTracks, int initialPosition) {

int totalHeadMovement = 0;

// Sort the tracks in ascending order

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

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

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

// Swap tracks if out of order

int temp = tracks[j];

tracks[j] = tracks[j + 1];

tracks[j + 1] = temp;

}

}

}

// Find the index of the initial position

int initialIndex = -1;

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

if (tracks[i] >= initialPosition) {

initialIndex = i;

break;

}

}

// SCAN to the right

printf("Head movement order (SCAN to the right):\n");

for (int i = initialIndex; i < numTracks; i++) {

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

totalHeadMovement += abs(tracks[i] - initialPosition);

initialPosition = tracks[i];

}

// SCAN to the left

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

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

totalHeadMovement += abs(tracks[i] - initialPosition);

initialPosition = tracks[i];

}

// Display the average head movement

printf("\nAverage Head Movement: %.2f\n", (float)totalHeadMovement / numTracks);

}

int main() {

// Test case

int tracks[] = {55, 58, 60, 70, 18};

int numTracks = sizeof(tracks) / sizeof(tracks[0]);

int initialPosition = 50; // Initial head position

// Execute SCAN disk scheduling algorithm

scanDisk(tracks, numTracks, initialPosition);

return 0;

}

24. Write a C Program to find the maximum size of a file that can be stored in the below file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks.

Test Case:

● Check that the start blocks and the required contiguous blocks are free.

● If free, allocate those blocks to the file.

● If not free, find the next available contiguous blocks.

#include <stdio.h>

#include <math.h>

#define BLOCK\_SIZE 8192 // 8 KB

#define POINTER\_SIZE 4 // 4 bytes

// Function to calculate the maximum file size

long long calculateMaxFileSize() {

// Total number of direct blocks

int directBlocks = 12;

// Number of pointers in a single indirect block

int pointersInSingleIndirect = BLOCK\_SIZE / POINTER\_SIZE;

// Number of pointers in a double indirect block

int pointersInDoubleIndirect = pointersInSingleIndirect \* pointersInSingleIndirect;

// Number of pointers in a triple indirect block

int pointersInTripleIndirect = pointersInDoubleIndirect \* pointersInSingleIndirect;

// Calculate the maximum file size

long long maxFileSize = 0;

maxFileSize += directBlocks \* BLOCK\_SIZE; // Direct blocks

maxFileSize += pointersInSingleIndirect \* BLOCK\_SIZE; // Single indirect block

maxFileSize += pointersInDoubleIndirect \* BLOCK\_SIZE \* BLOCK\_SIZE; // Double indirect block

maxFileSize += pointersInTripleIndirect \* BLOCK\_SIZE \* BLOCK\_SIZE \* BLOCK\_SIZE; // Triple indirect block

return maxFileSize;

}

int main() {

// Calculate the maximum file size

long long maxFileSize = calculateMaxFileSize();

// Display the result

printf("Maximum File Size: %lld bytes\n", maxFileSize);

return 0;

}

25. Write the C program to Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold in a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory.

Test Cases:

a. The block is added at the beginning.

b. The block is added in the middle.

c. The block is added at the end.

#include <stdio.h>

// Function to calculate disk I/O operations for contiguous allocation

int contiguousAllocation(int blockPosition, int currentSize, int newSize) {

if (newSize > 100) {

printf("Error: File size exceeds the total number of blocks.\n");

return -1;

}

if (blockPosition < 1 || blockPosition > currentSize + 1) {

printf("Error: Invalid block position.\n");

return -1;

}

// Contiguous allocation requires reading the block and updating the file control block

return 2;

}

// Function to calculate disk I/O operations for linked allocation

int linkedAllocation(int blockPosition, int currentSize, int newSize) {

if (newSize > 100) {

printf("Error: File size exceeds the total number of blocks.\n");

return -1;

}

if (blockPosition < 1 || blockPosition > currentSize + 1) {

printf("Error: Invalid block position.\n");

return -1;

}

// Linked allocation requires updating the pointers in the existing blocks

return currentSize;

}

// Function to calculate disk I/O operations for indexed (single-level) allocation

int indexedAllocation(int blockPosition, int currentSize, int newSize) {

if (newSize > 100) {

printf("Error: File size exceeds the total number of blocks.\n");

return -1;

}

if (blockPosition < 1 || blockPosition > currentSize + 1) {

printf("Error: Invalid block position.\n");

return -1;

}

// Indexed allocation requires reading the index block and updating it

return 2;

}

int main() {

int currentSize = 100;

int newSize = 0;

// Test Case a: Add block at the beginning

newSize = currentSize + 1;

printf("Contiguous Allocation - Add block at the beginning: %d disk I/O operations\n", contiguousAllocation(1, currentSize, newSize));

printf("Linked Allocation - Add block at the beginning: %d disk I/O operations\n", linkedAllocation(1, currentSize, newSize));

printf("Indexed Allocation - Add block at the beginning: %d disk I/O operations\n\n", indexedAllocation(1, currentSize, newSize));

// Test Case b: Add block in the middle

newSize = currentSize + 1;

printf("Contiguous Allocation - Add block in the middle: %d disk I/O operations\n", contiguousAllocation(currentSize / 2, currentSize, newSize));

printf("Linked Allocation - Add block in the middle: %d disk I/O operations\n", linkedAllocation(currentSize / 2, currentSize, newSize));

printf("Indexed Allocation - Add block in the middle: %d disk I/O operations\n\n", indexedAllocation(currentSize / 2, currentSize, newSize));

// Test Case c: Add block at the end

newSize = currentSize + 1;

printf("Contiguous Allocation - Add block at the end: %d disk I/O operations\n", contiguousAllocation(currentSize + 1, currentSize, newSize));

printf("Linked Allocation - Add block at the end: %d disk I/O operations\n", linkedAllocation(currentSize + 1, currentSize, newSize));

printf("Indexed Allocation - Add block at the end: %d disk I/O operations\n", indexedAllocation(currentSize + 1, currentSize, newSize));

return 0;

}

26. Write a program to compute the average waiting time and average turnaround time based on Priority scheduling for the following process with the given CPU burst times (and the assumption that all jobs arrive at the same time.)

Process Burst Time Priority

P1 30 2

P2 5 1

P3 12 3

#include <stdio.h>

// Process structure

struct Process {

int burstTime;

int priority;

};

// Function to calculate average waiting time and average turnaround time

void calculateAvgTimes(struct Process processes[], int n, float \*avgWaitingTime, float \*avgTurnaroundTime) {

int waitingTime[n], turnaroundTime[n];

// Calculate waiting time for each process

waitingTime[0] = 0;

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

waitingTime[i] = waitingTime[i - 1] + processes[i - 1].burstTime;

}

// Calculate turnaround time for each process

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

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

}

// Calculate average waiting time and average turnaround time

\*avgWaitingTime = 0;

\*avgTurnaroundTime = 0;

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

\*avgWaitingTime += waitingTime[i];

\*avgTurnaroundTime += turnaroundTime[i];

}

\*avgWaitingTime /= n;

\*avgTurnaroundTime /= n;

}

int main() {

// Processes

struct Process processes[] = {

{30, 2}, // P1

{5, 1}, // P2

{12, 3} // P3

};

// Number of processes

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

// Calculate average waiting time and average turnaround time

float avgWaitingTime, avgTurnaroundTime;

calculateAvgTimes(processes, n, &avgWaitingTime, &avgTurnaroundTime);

// Display the results

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

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

return 0;

}

27. Write a program for semaphore signaling mechanism where a process can signal a process that is waiting on a semaphore.

Test Case:

number of instances: 4

Number of Processes: 4 (P1, P2, P3, P4) all are calling wait operation on S

Show the response when another process P5 wants the same resource.

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t semaphore;

void \*process(void \*processID) {

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

// Process is waiting on the semaphore

printf("Process P%d is waiting on the semaphore.\n", id);

sem\_wait(&semaphore);

// Critical section

printf("Process P%d is in the critical section.\n", id);

sleep(2); // Simulate some work in the critical section

// Process is signaling the semaphore

printf("Process P%d is signaling the semaphore.\n", id);

sem\_post(&semaphore);

return NULL;

}

int main() {

// Initialize semaphore with 4 instances

sem\_init(&semaphore, 0, 4);

// Create threads for processes P1 to P4

pthread\_t threads[4];

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

for (int i = 0; i < 4; i++) {

pthread\_create(&threads[i], NULL, process, &processIDs[i]);

}

// Wait for processes P1 to P4 to finish

for (int i = 0; i < 4; i++) {

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

}

// Process P5 wants the same resource

printf("\nProcess P5 wants the same resource.\n");

sem\_wait(&semaphore);

printf("Process P5 is in the critical section.\n");

sleep(2); // Simulate some work in the critical section

printf("Process P5 is signaling the semaphore.\n");

sem\_post(&semaphore);

// Clean up

sem\_destroy(&semaphore);

return 0;

}

28. Write a C program to create a file using the system call. Read the data from the user and write the same in the file. Also, Read the data from the file and print the same in the console.

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

// File descriptor

int fd;

// Create a file

fd = open("example.txt", O\_CREAT | O\_WRONLY | O\_TRUNC, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

perror("Error creating file");

exit(EXIT\_FAILURE);

}

// Read data from the user

char buffer[100];

printf("Enter data to write to the file: ");

fgets(buffer, sizeof(buffer), stdin);

// Write data to the file

if (write(fd, buffer, sizeof(buffer)) == -1) {

perror("Error writing to file");

close(fd);

exit(EXIT\_FAILURE);

}

// Close the file

close(fd);

// Open the file for reading

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

if (fd == -1) {

perror("Error opening file for reading");

exit(EXIT\_FAILURE);

}

// Read data from the file and print to console

printf("\nReading data from the file:\n");

ssize\_t bytesRead;

while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {

write(STDOUT\_FILENO, buffer, bytesRead);

}

// Close the file

close(fd);

return 0;

}

29.Write a C program to implement the first-fit algorithm for memory management.

Test Case:

Memory partitions: 40 KB, 10 KB, 30 KB, 60 KB, (in order) Show the outcome for the test case with first-fit algorithms to place the processes of size 100 KB.50 KB.30 KB ,120 KB,35 KB (in order)

#include <stdio.h>

#define MAX\_PARTITIONS 4

#define MAX\_PROCESSES 5

// Function to allocate memory using first-fit algorithm

void firstFit(int partitions[], int m, int processes[], int n) {

int allocation[MAX\_PROCESSES];

// Initialize all allocations to -1, indicating unallocated

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

allocation[i] = -1;

}

// Iterate through each process and find the first-fit block

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

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

if (partitions[j] >= processes[i]) {

allocation[i] = j;

partitions[j] -= processes[i];

break;

}

}

}

// Display the memory allocation

printf("\nProcess\t\tMemory Block\n");

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

printf("%d\t\t", i + 1);

if (allocation[i] != -1) {

printf("%d KB\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

int main() {

// Memory partitions

int partitions[MAX\_PARTITIONS] = {40, 10, 30, 60};

// Processes

int processes[MAX\_PROCESSES] = {100, 50, 30, 120, 35};

// Display initial memory partitions

printf("Initial Memory Partitions:\n");

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

printf("%d KB ", partitions[i]);

}

// Allocate memory using first-fit algorithm

firstFit(partitions, MAX\_PARTITIONS, processes, MAX\_PROCESSES);

return 0;

}

30. Write a C Program to create two threads and print even numbers with one thread and odd numbers with another thread.

#include <stdio.h>

#include <pthread.h>

// Function to print even numbers

void \*printEven(void \*arg) {

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

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

printf("Even Thread: %d\n", i);

}

pthread\_exit(NULL);

}

// Function to print odd numbers

void \*printOdd(void \*arg) {

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

for (int i = 1; i <= n; i += 2) {

printf("Odd Thread: %d\n", i);

}

pthread\_exit(NULL);

}

int main() {

int maxNumber;

// Get the maximum number from the user

printf("Enter the maximum number: ");

scanf("%d", &maxNumber);

// Create threads

pthread\_t evenThread, oddThread;

// Create thread for even numbers

pthread\_create(&evenThread, NULL, printEven, &maxNumber);

// Create thread for odd numbers

pthread\_create(&oddThread, NULL, printOdd, &maxNumber);

// Wait for threads to finish

pthread\_join(evenThread, NULL);

pthread\_join(oddThread, NULL);

return 0;

}

31. Write a C program to simulate CSCAN disk scheduling algorithms and execute your program and find out and print the average head movement for the following test case.

No of tracks:5; Track position:55 58 60 70 18

#include <stdio.h>

#include <stdlib.h>

// Function to perform CSCAN disk scheduling

void cscanDisk(int tracks[], int numTracks, int initialPosition) {

int totalHeadMovement = 0;

// Sort the tracks in ascending order

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

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

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

// Swap tracks if out of order

int temp = tracks[j];

tracks[j] = tracks[j + 1];

tracks[j + 1] = temp;

}

}

}

// Find the index of the initial position

int initialIndex = -1;

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

if (tracks[i] >= initialPosition) {

initialIndex = i;

break;

}

}

// CSCAN to the right

printf("Head movement order (CSCAN to the right):\n");

for (int i = initialIndex; i < numTracks; i++) {

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

totalHeadMovement += abs(tracks[i] - initialPosition);

initialPosition = tracks[i];

}

// Move to the end

printf("0 "); // Move to track 0

totalHeadMovement += abs(0 - initialPosition);

initialPosition = 0;

// CSCAN to the right from track 0

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

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

totalHeadMovement += abs(tracks[i] - initialPosition);

initialPosition = tracks[i];

}

// Display the average head movement

printf("\nAverage Head Movement: %.2f\n", (float)totalHeadMovement / numTracks);

}

int main() {

// Test case

int tracks[] = {55, 58, 60, 70, 18};

int numTracks = sizeof(tracks) / sizeof(tracks[0]);

int initialPosition = 50; // Initial head position

// Execute CSCAN disk scheduling algorithm

cscanDisk(tracks, numTracks, initialPosition);

return 0;

}

32. Write a C program to simulate SCAN disk scheduling algorithms and execute your program and find out and print the average head movement for the following test case.

No of tracks:9

Track position:55 58 60 70 18 90 150 160 184

#include <stdio.h>

#include <stdlib.h>

// Function to perform SCAN disk scheduling

void scanDisk(int tracks[], int numTracks, int initialPosition) {

int totalHeadMovement = 0;

// Sort the tracks in ascending order

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

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

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

// Swap tracks if out of order

int temp = tracks[j];

tracks[j] = tracks[j + 1];

tracks[j + 1] = temp;

}

}

}

// Find the index of the initial position

int initialIndex = -1;

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

if (tracks[i] >= initialPosition) {

initialIndex = i;

break;

}

}

// SCAN to the right

printf("Head movement order (SCAN to the right):\n");

for (int i = initialIndex; i < numTracks; i++) {

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

totalHeadMovement += abs(tracks[i] - initialPosition);

initialPosition = tracks[i];

}

// Move to the end

printf("184 "); // Move to track 184

totalHeadMovement += abs(184 - initialPosition);

initialPosition = 184;

// SCAN to the left from track 184

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

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

totalHeadMovement += abs(tracks[i] - initialPosition);

initialPosition = tracks[i];

}

// Display the average head movement

printf("\nAverage Head Movement: %.2f\n", (float)totalHeadMovement / numTracks);

}

int main() {

// Test case

int tracks[] = {55, 58, 60, 70, 18, 90, 150, 160, 184};

int numTracks = sizeof(tracks) / sizeof(tracks[0]);

int initialPosition = 50; // Initial head position

// Execute SCAN disk scheduling algorithm

scanDisk(tracks, numTracks, initialPosition);

return 0;

}

33. Write a C program using the wait system call to synchronize the parent process and child process. In the parent process print the prime numbers. In the child process generate the Fibonacci series.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

// Function to check if a number is prime

int isPrime(int num) {

if (num < 2) {

return 0; // Not a prime number

}

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

if (num % i == 0) {

return 0; // Not a prime number

}

}

return 1; // Prime number

}

// Function to generate Fibonacci series

void generateFibonacci(int n) {

int a = 0, b = 1, nextTerm;

printf("Fibonacci Series: ");

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

printf("%d ", a);

nextTerm = a + b;

a = b;

b = nextTerm;

}

printf("\n");

}

int main() {

pid\_t childPid;

int status;

// Fork a child process

childPid = fork();

if (childPid < 0) {

perror("Fork failed");

exit(EXIT\_FAILURE);

} else if (childPid == 0) {

// Child process

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

int n = 10; // Number of Fibonacci series terms

generateFibonacci(n);

} else {

// Parent process

wait(&status); // Wait for the child process to finish

printf("Parent process (PID: %d)\n", getpid());

// Print prime numbers

printf("Prime Numbers: ");

for (int i = 2; i <= 20; i++) {

if (isPrime(i)) {

printf("%d ", i);

}

}

printf("\n");

}

return 0;

}

34. Write a C program to implement the worst-fit algorithm and allocate the memory block to each process.

Test Case:

Memory partitions: 40 KB, 10 KB. 30 KB, .60 KB (in order),

Show the outcome for the test case with the worst-fit algorithms to place processes of size 100 KB.50 KB.30 KB .120 KB,35 KB (in order)

#include <stdio.h>

#define MAX\_PARTITIONS 4

#define MAX\_PROCESSES 5

// Function to allocate memory using worst-fit algorithm

void worstFit(int partitions[], int m, int processes[], int n) {

int allocation[MAX\_PROCESSES];

// Initialize all allocations to -1, indicating unallocated

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

allocation[i] = -1;

}

// Iterate through each process and find the worst-fit block

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

int worstFitIndex = -1;

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

if (partitions[j] >= processes[i]) {

if (worstFitIndex == -1 || partitions[j] > partitions[worstFitIndex]) {

worstFitIndex = j;

}

}

}

// Allocate the process to the worst-fit block

if (worstFitIndex != -1) {

allocation[i] = worstFitIndex;

partitions[worstFitIndex] -= processes[i];

}

}

// Display the memory allocation

printf("\nProcess\t\tMemory Block\n");

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

printf("%d\t\t", i + 1);

if (allocation[i] != -1) {

printf("%d KB\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

int main() {

// Memory partitions

int partitions[MAX\_PARTITIONS] = {40, 10, 30, 60};

// Processes

int processes[MAX\_PROCESSES] = {100, 50, 30, 120, 35};

// Display initial memory partitions

printf("Initial Memory Partitions:\n");

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

printf("%d KB ", partitions[i]);

}

// Allocate memory using worst-fit algorithm

worstFit(partitions, MAX\_PARTITIONS, processes, MAX\_PROCESSES);

return 0;

}

35. Write a C program to simulate the sequential file allocation in a very simple file system with a disk of 16 blocks, each block is of 1 KB size and first 8 blocks (0 to 7) are allocated to the “iNodes” and can’t be used by the file system. Blocks available for allocation are from block 8 to block 16. Minimum file size is 1 KB. Check that the start blocks and the required contiguous blocks are free. If free, allocate those blocks to the file. If not free, find the next available contiguous blocks.

Test Case: If there are not enough contiguous blocks available for a file, the program must exit ()

#include <stdio.h>

#include <stdlib.h>

#define TOTAL\_BLOCKS 16

#define INODE\_BLOCKS 8

#define FIRST\_DATA\_BLOCK 8

// Function to allocate sequential blocks for a file

void allocateBlocks(int startBlock, int numBlocks) {

// Check if the blocks are available

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

if (i >= TOTAL\_BLOCKS) {

printf("Error: Not enough contiguous blocks available for the file.\n");

exit(EXIT\_FAILURE);

}

}

// Allocate blocks to the file

printf("Allocated Blocks: ");

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

printf("%d ", i);

}

printf("\n");

}

int main() {

int fileSize; // File size in KB

int startBlock; // Starting block for the file

int numBlocks; // Number of blocks required for the file

// Disk initialization

printf("Disk Initialization:\n");

printf("Total Blocks: %d\n", TOTAL\_BLOCKS);

printf("iNode Blocks: %d\n", INODE\_BLOCKS);

printf("Data Blocks: %d to %d\n", FIRST\_DATA\_BLOCK, TOTAL\_BLOCKS - 1);

printf("\n");

// Get file information from the user

printf("Enter the file size in KB (minimum 1 KB): ");

scanf("%d", &fileSize);

// Minimum file size check

if (fileSize < 1) {

printf("Error: Minimum file size is 1 KB.\n");

exit(EXIT\_FAILURE);

}

// Get starting block and number of blocks required for the file

printf("Enter the starting block for the file (between %d and %d): ", FIRST\_DATA\_BLOCK, TOTAL\_BLOCKS - 1);

scanf("%d", &startBlock);

printf("Enter the number of blocks required for the file: ");

scanf("%d", &numBlocks);

// Sequential file allocation

allocateBlocks(startBlock, numBlocks);

return 0;

}

36. Write a C program to simulate SSTF disk scheduling algorithm and execute your program and find the average head movement with the following test case:

No of tracks 5; Track position:55 58 60 70 18

#include <stdio.h>

#include <stdlib.h>

// Function to find the index of the nearest track

int findNearestTrack(int tracks[], int numTracks, int currentPosition) {

int minDistance = abs(tracks[0] - currentPosition);

int index = 0;

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

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

if (distance < minDistance) {

minDistance = distance;

index = i;

}

}

return index;

}

// Function to perform SSTF disk scheduling

void sstfDisk(int tracks[], int numTracks, int initialPosition) {

int totalHeadMovement = 0;

int currentPosition = initialPosition;

printf("Head movement order (SSTF):\n");

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

int nearestIndex = findNearestTrack(tracks, numTracks, currentPosition);

printf("%d ", tracks[nearestIndex]);

totalHeadMovement += abs(tracks[nearestIndex] - currentPosition);

currentPosition = tracks[nearestIndex];

tracks[nearestIndex] = -1; // Mark the track as visited

}

// Display the average head movement

printf("\nAverage Head Movement: %.2f\n", (float)totalHeadMovement / numTracks);

}

int main() {

// Test case

int tracks[] = {55, 58, 60, 70, 18};

int numTracks = sizeof(tracks) / sizeof(tracks[0]);

int initialPosition = 50; // Initial head position

// Execute SSTF disk scheduling algorithm

sstfDisk(tracks, numTracks, initialPosition);

return 0;

}

37. Write a C program to illustrate the Optimal method of page replacement and determine the number of page faults for the following test case:

No of page frames: 3; Page reference sequence: 4, 1, 2, 4, 3, 2, 1 and 5.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGE\_FRAMES 10

// Function to check if a page is present in the page frames

int isPagePresent(int pageFrames[], int numFrames, int page) {

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

if (pageFrames[i] == page) {

return 1; // Page is present

}

}

return 0; // Page is not present

}

// Function to find the optimal page to replace

int findOptimalPage(int pageFrames[], int numFrames, int pageReferences[], int referenceLength, int currentPosition) {

int farthestIndex = -1;

int farthestDistance = -1;

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

int nextPage = pageFrames[i];

int futureIndex = currentPosition;

while (futureIndex < referenceLength) {

if (pageReferences[futureIndex] == nextPage) {

if (futureIndex > farthestDistance) {

farthestDistance = futureIndex;

farthestIndex = i;

}

break;

}

futureIndex++;

}

if (futureIndex == referenceLength) {

return i; // If a page will not be referenced in the future, replace it

}

}

return farthestIndex;

}

// Function to perform Optimal page replacement

void optimalPageReplacement(int pageReferences[], int referenceLength, int numPageFrames) {

int pageFrames[MAX\_PAGE\_FRAMES] = {0};

int pageFaults = 0;

printf("Page Reference Sequence: ");

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

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

}

printf("\n");

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

printf("\nStep %d:\n", i + 1);

int currentPage = pageReferences[i];

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

if (!isPagePresent(pageFrames, numPageFrames, currentPage)) {

pageFaults++;

printf("Page %d is not present in the frames. Page Fault!\n", currentPage);

// Find the index to replace

int replaceIndex = findOptimalPage(pageFrames, numPageFrames, pageReferences, referenceLength, i + 1);

printf("Replacing Page %d with Page %d\n", pageFrames[replaceIndex], currentPage);

pageFrames[replaceIndex] = currentPage;

} else {

printf("Page %d is already present in the frames. No Page Fault.\n", currentPage);

}

// Display the current state of page frames

printf("Page Frames: ");

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

printf("%d ", pageFrames[j]);

}

printf("\n");

}

// Display the total number of page faults

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

}

int main() {

// Test case

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

int referenceLength = sizeof(pageReferences) / sizeof(pageReferences[0]);

int numPageFrames = 3;

// Execute Optimal page replacement algorithm

optimalPageReplacement(pageReferences, referenceLength, numPageFrames);

return 0;

}

38. Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8-time units. All processes arrive at time zero. Write a program to compute the average waiting time and average turnaround time based on Shortest Job First Scheduling.

#include <stdio.h>

#include <stdlib.h>

// Process structure

typedef struct {

int processId;

int burstTime;

int waitingTime;

int turnaroundTime;

} Process;

// Function to swap two processes

void swap(Process \*a, Process \*b) {

Process temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to perform Shortest Job First scheduling

void sjfScheduling(Process processes[], int numProcesses) {

// Sort processes based on burst time

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

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

if (processes[j].burstTime > processes[j + 1].burstTime) {

swap(&processes[j], &processes[j + 1]);

}

}

}

// Calculate waiting time and turnaround time

processes[0].waitingTime = 0;

processes[0].turnaroundTime = processes[0].burstTime;

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

processes[i].waitingTime = processes[i - 1].waitingTime + processes[i - 1].burstTime;

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

}

}

// Function to display process details and compute averages

void displayAndComputeAverages(Process processes[], int numProcesses) {

float totalWaitingTime = 0, totalTurnaroundTime = 0;

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

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

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

// Compute averages

float avgWaitingTime = totalWaitingTime / numProcesses;

float avgTurnaroundTime = totalTurnaroundTime / numProcesses;

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

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

}

int main() {

// Number of processes

int numProcesses = 3;

// Process details

Process processes[numProcesses];

processes[0].processId = 0;

processes[0].burstTime = 2;

processes[1].processId = 1;

processes[1].burstTime = 4;

processes[2].processId = 2;

processes[2].burstTime = 8;

// Perform Shortest Job First scheduling

sjfScheduling(processes, numProcesses);

// Display process details and compute averages

displayAndComputeAverages(processes, numProcesses);

return 0;

}

39. Write a C program to simulate LOOK disk scheduling algorithms. and execute your program and find out and print the average head movement for the following test case.

No of tracks:5; Track position:55 58 60 70 18

#include <stdio.h>

#include <stdlib.h>

// Function to perform LOOK disk scheduling

void lookDisk(int tracks[], int numTracks, int initialPosition) {

int totalHeadMovement = 0;

int currentPosition = initialPosition;

int direction = 1; // 1 for moving towards higher track numbers, -1 for moving towards lower track numbers

printf("Head movement order (LOOK):\n");

while (1) {

printf("%d ", currentPosition);

int found = 0;

// Move towards higher track numbers

if (direction == 1) {

for (int i = currentPosition + 1; i < numTracks; i++) {

if (tracks[i] != -1) {

totalHeadMovement += abs(tracks[i] - currentPosition);

currentPosition = tracks[i];

tracks[i] = -1; // Mark the track as visited

found = 1;

break;

}

}

}

// Move towards lower track numbers

else {

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

if (tracks[i] != -1) {

totalHeadMovement += abs(tracks[i] - currentPosition);

currentPosition = tracks[i];

tracks[i] = -1; // Mark the track as visited

found = 1;

break;

}

}

}

// If no track found in the current direction, change direction

if (!found) {

direction = -direction;

}

// Break if all tracks are visited

int allVisited = 1;

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

if (tracks[i] != -1) {

allVisited = 0;

break;

}

}

if (allVisited) {

break;

}

}

// Display the average head movement

printf("\nAverage Head Movement: %.2f\n", (float)totalHeadMovement / numTracks);

}

int main() {

// Test case

int tracks[] = {55, 58, 60, 70, 18};

int numTracks = sizeof(tracks) / sizeof(tracks[0]);

int initialPosition = 50; // Initial head position

// Execute LOOK disk scheduling algorithm

lookDisk(tracks, numTracks, initialPosition);

return 0;

}

40. Write a C program to simulate CLOOK disk scheduling algorithms. and execute your program and find out and print the average head movement for the following test case.

No of tracks:5; Track position:55 58 60 70 18

#include <stdio.h>

#include <stdlib.h>

// Function to perform CLOOK disk scheduling

void clookDisk(int tracks[], int numTracks, int initialPosition) {

int totalHeadMovement = 0;

int currentPosition = initialPosition;

printf("Head movement order (CLOOK):\n");

// Sort tracks in ascending order

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

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

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

int temp = tracks[j];

tracks[j] = tracks[j + 1];

tracks[j + 1] = temp;

}

}

}

// Find the index of the initial position

int initialIndex = 0;

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

if (tracks[i] >= initialPosition) {

initialIndex = i;

break;

}

}

// Move towards higher track numbers

for (int i = initialIndex; i < numTracks; i++) {

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

totalHeadMovement += abs(tracks[i] - currentPosition);

currentPosition = tracks[i];

}

// Move towards lower track numbers

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

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

totalHeadMovement += abs(tracks[i] - currentPosition);

currentPosition = tracks[i];

}

// Display the average head movement

printf("\nAverage Head Movement: %.2f\n", (float)totalHeadMovement / numTracks);

}

int main() {

// Test case

int tracks[] = {55, 58, 60, 70, 18};

int numTracks = sizeof(tracks) / sizeof(tracks[0]);

int initialPosition = 50; // Initial head position

// Execute CLOOK disk scheduling algorithm

clookDisk(tracks, numTracks, initialPosition);

return 0;

}