CSA0401 OPERATING SYSTEM

6.ROUND ROBIN SCHEDULING

#include <stdio.h>

int main() {

int i, n, time = 0, remain, flag = 0;

int quantum;

int wt = 0, tat = 0;

int at[10], bt[10], rt[10];

printf("Enter Total Process:\t ");

scanf("%d", &n);

remain = n;

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

printf("Enter Burst Time for Process P%d:\t", i + 1);

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

at[i] = 0; // Assuming arrival time is 0 for all processes

rt[i] = bt[i];

}

printf("Enter Time Quantum:\t");

scanf("%d", &quantum);

printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");

for(time = 0, i = 0; remain != 0;) {

if(rt[i] > 0 && rt[i] <= quantum) {

time += rt[i];

rt[i] = 0;

flag = 1;

} else if(rt[i] > 0) {

rt[i] -= quantum;

time += quantum;

}

if(rt[i] == 0 && flag == 1) {

remain--;

printf("P[%d]\t|\t%d\t|\t%d\n", i + 1, time - at[i], time - at[i] - bt[i]);

wt += time - at[i] - bt[i];

tat += time - at[i];

flag = 0;

}

i = (i + 1) % n;

}

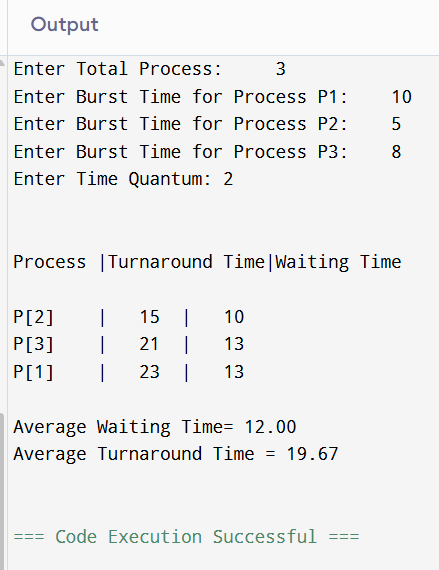
printf("\nAverage Waiting Time= %.2f\n", (float)wt / n);

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

return 0;

}

OUT PUT:



7. INTER PROCESS COMMUNICATION

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

int main() {

int fd[2]; // fd[0] for reading, fd[1] for writing

pid\_t pid;

char write\_msg[] = "Hello from parent!";

char read\_msg[100];

// Create the pipe

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

perror("pipe");

exit(EXIT\_FAILURE);

}

// Fork a child process

pid = fork();

if (pid < 0) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid > 0) {

// Parent process

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

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

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

} else {

// Child process

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

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

printf("Child received: %s\n", read\_msg);

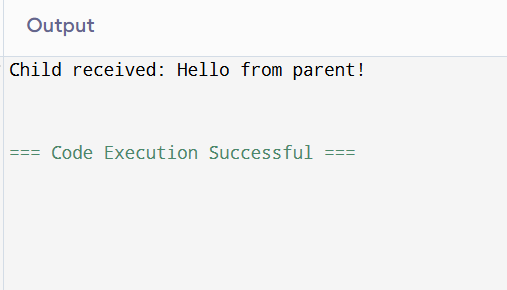
close(fd[0]); // Close read end after reading

}

return 0;

}

OUTPUT:



8.DINING -PHILOSOPHERS PROBLEM

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

#define N 5 // Number of philosophers

sem\_t forks[N]; // Semaphores for forks

sem\_t room; // Semaphore to limit the number of philosophers trying to pick up forks

pthread\_t philosophers[N];

int phil\_ids[N];

void\* philosopher(void\* num) {

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

while (1) {

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

sleep(1); // Thinking

sem\_wait(&room); // Enter room

sem\_wait(&forks[id]); // Pick up left fork

sem\_wait(&forks[(id + 1) % N]); // Pick up right fork

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

sleep(2); // Eating

sem\_post(&forks[(id + 1) % N]); // Put down right fork

sem\_post(&forks[id]); // Put down left fork

sem\_post(&room); // Leave room

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

sleep(1); // Thinking again

}

return NULL;

}

int main() {

int i;

// Initialize semaphores

sem\_init(&room, 0, N - 1); // Allow up to N-1 philosophers to enter room

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

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

phil\_ids[i] = i;

}

// Create philosopher threads

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

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

}

// Wait for philosopher threads to finish (they won't in this infinite loop)

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

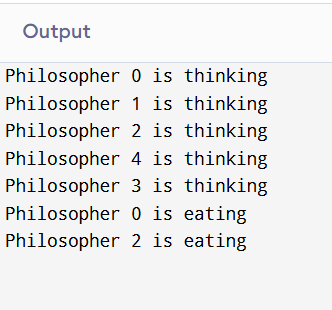
pthread\_join(philosophers[i], NULL);

}

return 0;

}

OUTPUT:



9.BANKER’S ALGORITHM

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define SIZE 5

int buffer[SIZE];

int in = 0, out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void \*producer(void \*arg) {

int item;

while (1) {

item = rand() % 100; // Produce an item

sem\_wait(&empty); // Wait for empty slot

pthread\_mutex\_lock(&mutex); // Lock the buffer

buffer[in] = item;

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

in = (in + 1) % SIZE;

pthread\_mutex\_unlock(&mutex); // Unlock the buffer

sem\_post(&full); // Signal that item is added

sleep(1); // Simulate production time

}

return NULL;

}

void \*consumer(void \*arg) {

int item;

while (1) {

sem\_wait(&full); // Wait for available item

pthread\_mutex\_lock(&mutex); // Lock the buffer

item = buffer[out];

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

out = (out + 1) % SIZE;

pthread\_mutex\_unlock(&mutex); // Unlock the buffer

sem\_post(&empty); // Signal that slot is free

sleep(2); // Simulate consumption time

}

return NULL;

}

int main() {

pthread\_t prod, cons;

// Initialize semaphores and mutex

sem\_init(&empty, 0, SIZE);

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

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

// Create producer and consumer threads

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

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

// Wait for threads to finish (they won't in this infinite loop example)

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

// Cleanup (unreachable here)

sem\_destroy(&empty);

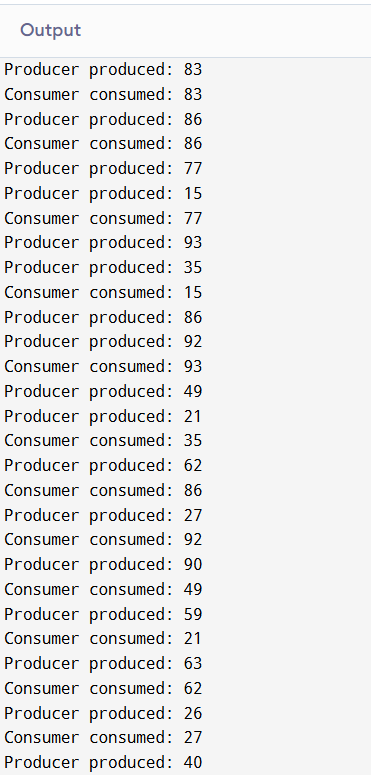
sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

    return 0;

}

OUTPUT:



10. PRODUCER CONSUMER PROBLEM

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void \*producer(void \*arg) {

int item;

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

item = rand() % 100;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

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

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

sleep(1);

}

pthread\_exit(NULL);

}

void \*consumer(void \*arg) {

int item;

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

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

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

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

sleep(1);

}

pthread\_exit(NULL);

}

int main() {

pthread\_t prod, cons;

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

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

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

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

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

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

OUTPUT:



11. MULTITHREADING

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 5

void\* print\_hello(void\* threadid) {

long tid = (long)threadid;

printf("Hello from thread %ld\n", tid);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

for(t = 0; t < NUM\_THREADS; t++) {

printf("Creating thread %ld\n", t);

rc = pthread\_create(&threads[t], NULL, print\_hello, (void \*)t);

if (rc) {

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

// Wait for all threads to complete

for(t = 0; t < NUM\_THREADS; t++) {

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

}

printf("Main thread exiting.\n");

return 0;

}

OUTPUT:



12.FIFO PAGING

#include <stdio.h>

#include <stdlib.h>

int main() {

int i, j, n, frames, page\_faults = 0, flag, index = 0;

int \*pages, \*memory;

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

scanf("%d", &n);

pages = (int \*)malloc(n \* sizeof(int));

printf("Enter the page reference string:\n");

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

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

}

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

scanf("%d", &frames);

memory = (int \*)malloc(frames \* sizeof(int));

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

memory[i] = -1;

}

printf("\nPage\tFrames\n");

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

flag = 0;

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

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

flag = 1;

break;

}

}

if(flag == 0) {

memory[index] = pages[i];

index = (index + 1) % frames;

page\_faults++;

}

printf("%d\t", pages[i]);

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

if(memory[j] != -1)

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

else

printf("- ");

}

printf("\n");

}

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

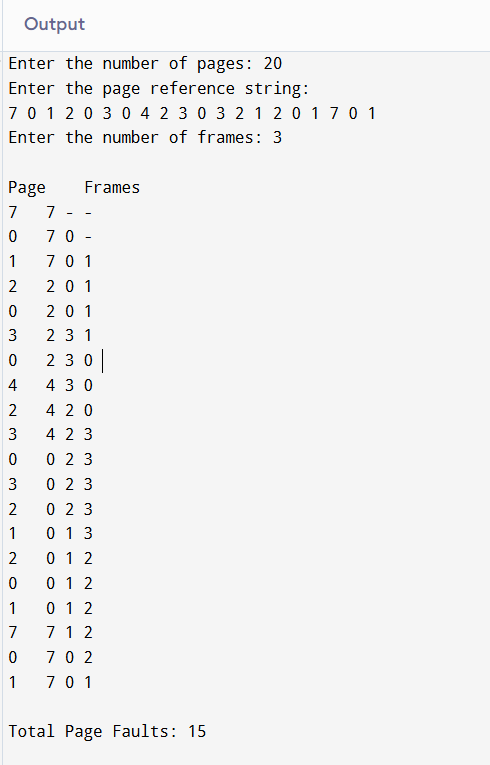
free(pages);

free(memory);

return 0;

}

OUTPUT:



13.LRU PAGING

#include <stdio.h>

int findLRU(int time[], int n) {

int i, minimum = time[0], pos = 0;

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

if(time[i] < minimum) {

minimum = time[i];

pos = i;

}

}

return pos;

}

int main() {

int no\_of\_frames, no\_of\_pages, frames[10], pages[30];

int counter = 0, time[10];

int flag1, flag2, i, j, pos, faults = 0;

printf("Enter number of frames: ");

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

printf("Enter number of pages: ");

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

printf("Enter the reference string:\n");

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

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

}

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

frames[i] = -1;

}

printf("\nPage\tFrames\n");

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

flag1 = flag2 = 0;

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

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

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0) {

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

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

counter++;

faults++;

frames[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

if(flag2 == 0) {

pos = findLRU(time, no\_of\_frames);

counter++;

faults++;

frames[pos] = pages[i];

time[pos] = counter;

}

printf("%d\t", pages[i]);

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

if(frames[j] != -1)

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

else

printf("- ");

}

printf("\n");

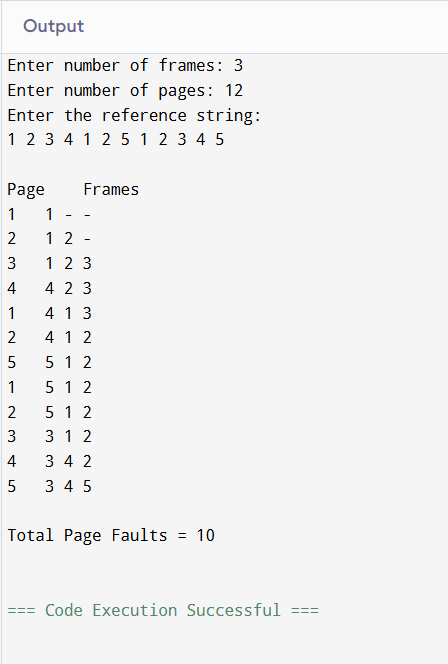
}

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

return 0;

}

OUTPUT:



14. OPTIMAL PAGING

#include <stdio.h>

int findOptimal(int pages[], int frames[], int current, int totalPages, int numFrames) {

int farthest = current;

int index = -1;

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

int j;

for (j = current + 1; j < totalPages; j++) {

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

if (j > farthest) {

farthest = j;

index = i;

}

break;

}

}

if (j == totalPages) {

return i;

}

}

return (index == -1) ? 0 : index;

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int numFrames = 3;

int totalPages = sizeof(pages) / sizeof(pages[0]);

int frames[numFrames];

int pageFaults = 0;

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

frames[i] = -1;

}

printf("Page Reference String: ");

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

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

}

printf("\n");

printf("\nPage\tFrames\n");

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

int page = pages[i];

int found = 0;

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

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

int replaceIndex = findOptimal(pages, frames, i, totalPages, numFrames);

frames[replaceIndex] = page;

pageFaults++;

}

printf("%d\t", page);

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

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

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

} else {

printf("- ");

}

}

printf("\n");

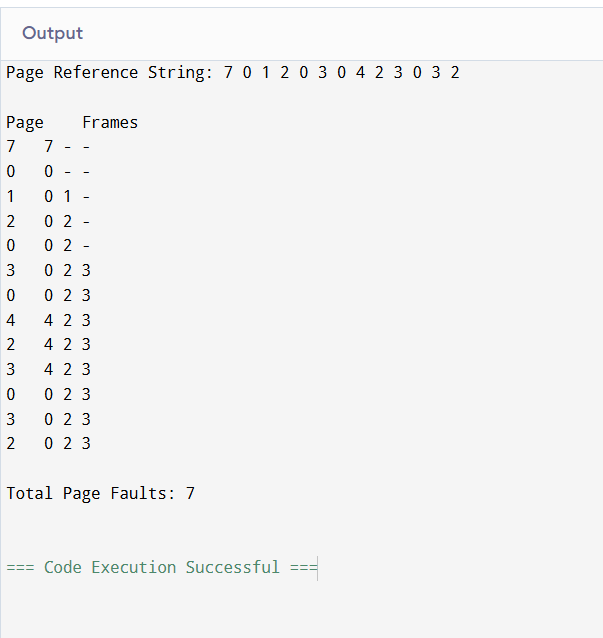
}

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

return 0;

}

OUTPUT:



15.SEQUENTIAL FILE ALLOCATION

#include <stdio.h>

#define MAX\_FILES 10

#define MAX\_BLOCKS 20

int disk[MAX\_BLOCKS] = {0}; // 0 indicates free, 1 indicates occupied

void allocateFile(int fileNum, int numBlocks) {

int startBlock = -1;

for (int i = 0; i < MAX\_BLOCKS - numBlocks + 1; i++) {

int j;

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

if (disk[i + j] == 1) {

break;

}

}

if (j == numBlocks) {

startBlock = i;

break;

}

}

if (startBlock != -1) {

printf("File%d allocated from block %d to %d\n", fileNum, startBlock, startBlock + numBlocks - 1);

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

disk[i] = 1;

}

} else {

printf("Not enough contiguous space to allocate File%d\n", fileNum);

}

}

void deallocateFile(int fileNum, int startBlock, int numBlocks) {

printf("Deallocating File%d from block %d to %d\n", fileNum, startBlock, startBlock + numBlocks - 1);

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

disk[i] = 0;

}

}

void displayDisk() {

printf("\nDisk Blocks: ");

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

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

}

printf("\n");

}

int main() {

int numFiles, fileNum, numBlocks, startBlock;

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

scanf("%d", &numFiles);

for (fileNum = 1; fileNum <= numFiles; fileNum++) {

printf("\nEnter the number of blocks for File%d: ", fileNum);

scanf("%d", &numBlocks);

allocateFile(fileNum, numBlocks);

displayDisk();

}

printf("\nEnter the file number to deallocate: ");

scanf("%d", &fileNum);

printf("Enter the starting block and number of blocks for File%d: ", fileNum);

scanf("%d %d", &startBlock, &numBlocks);

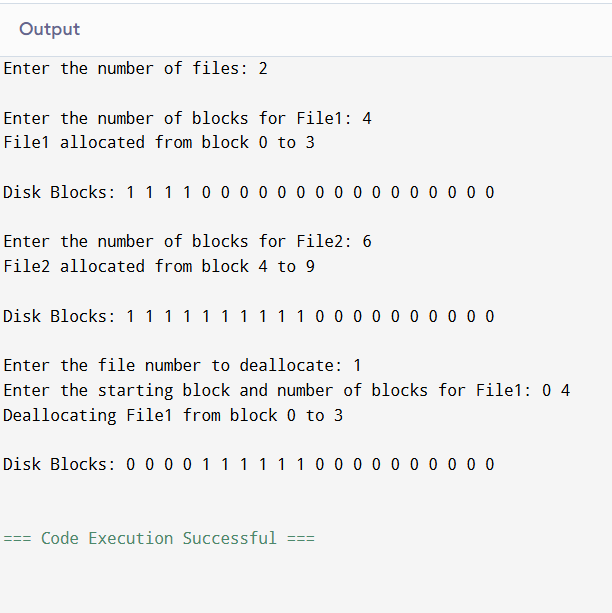
deallocateFile(fileNum, startBlock, numBlocks);

displayDisk();

return 0;

}

Output:



16.FIRST FIT STRATEGY

#include <stdio.h>

#define MAX\_BLOCKS 10

#define MAX\_PROCESSES 10

int main() {

int blockSize[MAX\_BLOCKS], processSize[MAX\_PROCESSES];

int blockCount, processCount;

int allocation[MAX\_PROCESSES];

int i, j;

// Input number of blocks and their sizes

printf("Enter the number of memory blocks: ");

scanf("%d", &blockCount);

printf("Enter the size of each block:\n");

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

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

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

}

// Input number of processes and their sizes

printf("\nEnter the number of processes: ");

scanf("%d", &processCount);

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

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

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

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

allocation[i] = -1; // Initialize allocation as -1 (not allocated)

}

// First Fit Allocation

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

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

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

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

// Display allocation results

printf("\nProcess No.\tProcess Size\tBlock No.\n");

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

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

if(allocation[i] != -1)

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

else

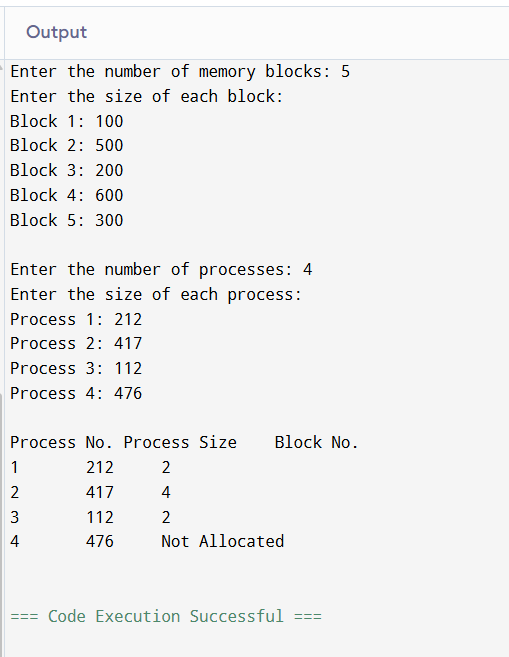
printf("Not Allocated\n");

}

return 0;

}

OUTPUT:



17.FIRST COME FIRST SERVED DISK SCHEDULING

#include <stdio.h>

#include <stdlib.h>

int main() {

int n, i, head, seek = 0, diff;

int queue[100];

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

scanf("%d", &n);

printf("Enter the sequence of disk I/O requests:\n");

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

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

}

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

scanf("%d", &head);

printf("\nSequence of head movements:\n");

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

diff = abs(queue[i] - head);

seek += diff;

printf("Move from %d to %d with seek %d\n", head, queue[i], diff);

head = queue[i];

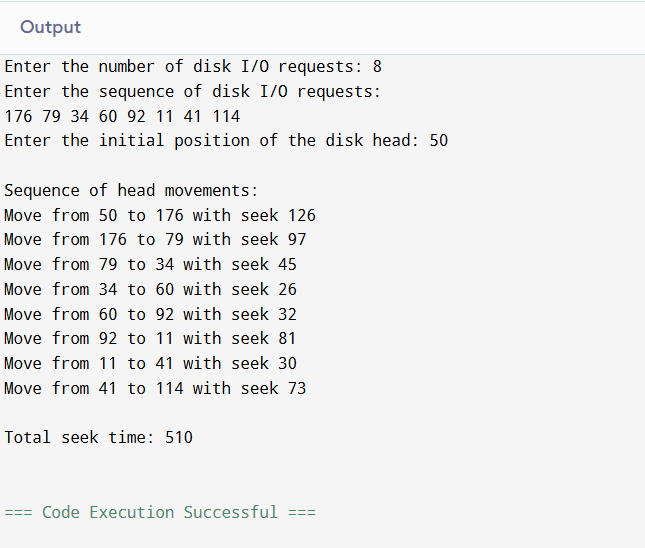
}

printf("\nTotal seek time: %d\n", seek);

return 0;

}

OUTPUT:



18.SCAND DISK SHEDULING

#include <stdio.h>

#include <stdlib.h>

int compare(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b);

}

int main() {

int n, head, i, j, seek = 0, size, direction;

int requests[100], left[100], right[100];

int left\_count = 0, right\_count = 0;

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

scanf("%d", &n);

printf("Enter the sequence of disk I/O requests:\n");

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

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

}

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

scanf("%d", &head);

printf("Enter the total disk size: ");

scanf("%d", &size);

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

scanf("%d", &direction);

// Separate requests into left and right of the head

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

if(requests[i] < head)

left[left\_count++] = requests[i];

else

right[right\_count++] = requests[i];

}

// Sort the left and right arrays

qsort(left, left\_count, sizeof(int), compare);

qsort(right, right\_count, sizeof(int), compare);

printf("\nSeek Sequence:\n");

if(direction == 0) {

// Move towards left

for(i = left\_count - 1; i >= 0; i--) {

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

seek += abs(head - left[i]);

head = left[i];

}

// If there are requests on the right

if(right\_count > 0) {

// Move to the beginning of the disk

seek += head; // head is at left[0]

head = 0;

printf("0 -> ");

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

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

seek += abs(head - right[i]);

head = right[i];

}

}

} else {

// Move towards right

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

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

seek += abs(head - right[i]);

head = right[i];

}

// If there are requests on the left

if(left\_count > 0) {

// Move to the end of the disk

seek += (size - 1 - head);

head = size - 1;

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

for(i = left\_count - 1; i >= 0; i--) {

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

seek += abs(head - left[i]);

head = left[i];

}

}

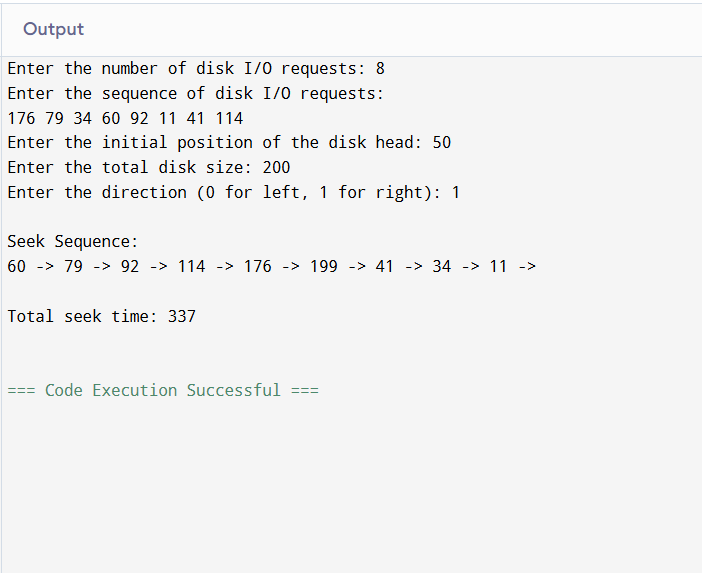
}

printf("\n\nTotal seek time: %d\n", seek);

return 0;

}

OUTPUT:



19.SINGLE LEVEL DIRECTORY

#include <stdio.h>

#include <string.h>

#define MAX\_FILES 100

#define MAX\_FILENAME\_LENGTH 20

typedef struct {

char dirName[50];

char fileNames[MAX\_FILES][MAX\_FILENAME\_LENGTH];

int fileCount;

} Directory;

void createFile(Directory \*dir) {

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

printf("Directory is full. Cannot create more files.\n");

return;

}

char fileName[MAX\_FILENAME\_LENGTH];

printf("Enter the name of the file to create: ");

scanf("%s", fileName);

// Check for duplicate file names

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

if (strcmp(dir->fileNames[i], fileName) == 0) {

printf("File '%s' already exists in the directory.\n", fileName);

return;

}

}

strcpy(dir->fileNames[dir->fileCount], fileName);

dir->fileCount++;

printf("File '%s' created successfully.\n", fileName);

}

void deleteFile(Directory \*dir) {

if (dir->fileCount == 0) {

printf("Directory is empty. No files to delete.\n");

return;

}

char fileName[MAX\_FILENAME\_LENGTH];

printf("Enter the name of the file to delete: ");

scanf("%s", fileName);

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

if (strcmp(dir->fileNames[i], fileName) == 0) {

// Shift files to fill the gap

for (int j = i; j < dir->fileCount - 1; j++) {

strcpy(dir->fileNames[j], dir->fileNames[j + 1]);

}

dir->fileCount--;

printf("File '%s' deleted successfully.\n", fileName);

return;

}

}

printf("File '%s' not found in the directory.\n", fileName);

}

void searchFile(Directory \*dir) {

if (dir->fileCount == 0) {

printf("Directory is empty. No files to search.\n");

return;

}

char fileName[MAX\_FILENAME\_LENGTH];

printf("Enter the name of the file to search: ");

scanf("%s", fileName);

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

if (strcmp(dir->fileNames[i], fileName) == 0) {

printf("File '%s' found at position %d.\n", fileName, i + 1);

return;

}

}

printf("File '%s' not found in the directory.\n", fileName);

}

void displayFiles(Directory \*dir) {

if (dir->fileCount == 0) {

printf("Directory is empty. No files to display.\n");

return;

}

printf("Files in directory '%s':\n", dir->dirName);

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

printf("%d. %s\n", i + 1, dir->fileNames[i]);

}

}

int main() {

Directory dir;

dir.fileCount = 0;

printf("Enter the name of the directory: ");

scanf("%s", dir.dirName);

int choice;

do {

printf("\n--- Single-Level Directory Menu ---\n");

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

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

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

printf("4. Display Files\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

createFile(&dir);

break;

case 2:

deleteFile(&dir);

break;

case 3:

searchFile(&dir);

break;

case 4:

displayFiles(&dir);

break;

case 5:

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

break;

default:

printf("Invalid choice. Please select a valid option.\n");

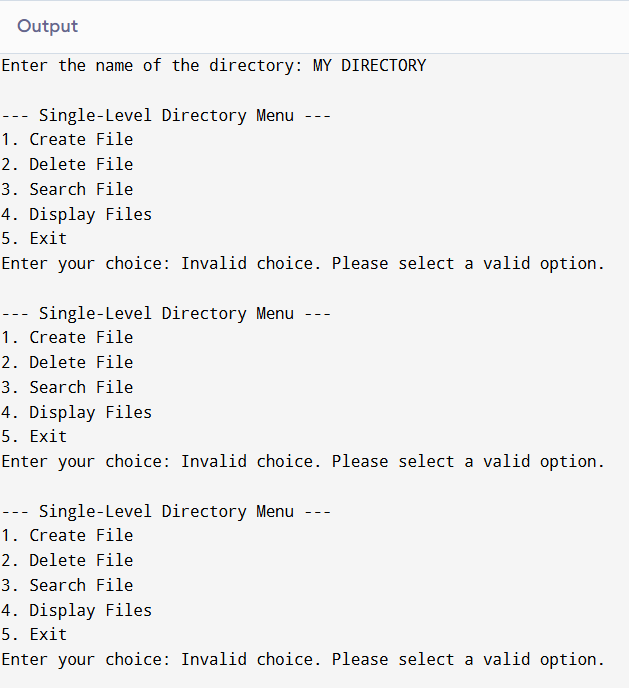
}

} while (choice != 5);

return 0;

}

OUTPUT:



20.TWO LEVEL DIRECTORY STRUCTURE

#include <stdio.h>

#include <string.h>

#define MAX\_USERS 10

#define MAX\_FILES 10

#define MAX\_NAME\_LEN 20

typedef struct {

char filename[MAX\_NAME\_LEN];

} File;

typedef struct {

char username[MAX\_NAME\_LEN];

File files[MAX\_FILES];

int fileCount;

} UserDirectory;

UserDirectory directories[MAX\_USERS];

int userCount = 0;

int findUserIndex(char \*username) {

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

if (strcmp(directories[i].username, username) == 0)

return i;

}

return -1;

}

void createUserDirectory() {

if (userCount >= MAX\_USERS) {

printf("Maximum user limit reached.\n");

return;

}

char username[MAX\_NAME\_LEN];

printf("Enter username: ");

scanf("%s", username);

if (findUserIndex(username) != -1) {

printf("User directory already exists.\n");

return;

}

strcpy(directories[userCount].username, username);

directories[userCount].fileCount = 0;

userCount++;

printf("User directory '%s' created successfully.\n", username);

}

void createFile() {

char username[MAX\_NAME\_LEN], filename[MAX\_NAME\_LEN];

printf("Enter username: ");

scanf("%s", username);

int userIndex = findUserIndex(username);

if (userIndex == -1) {

printf("User directory not found.\n");

return;

}

if (directories[userIndex].fileCount >= MAX\_FILES) {

printf("Maximum file limit reached for user '%s'.\n", username);

return;

}

printf("Enter filename: ");

scanf("%s", filename);

// Check for duplicate file

for (int i = 0; i < directories[userIndex].fileCount; i++) {

if (strcmp(directories[userIndex].files[i].filename, filename) == 0) {

printf("File already exists in user directory.\n");

return;

}

}

strcpy(directories[userIndex].files[directories[userIndex].fileCount].filename, filename);

directories[userIndex].fileCount++;

printf("File '%s' created in user directory '%s'.\n", filename, username);

}

void deleteFile() {

char username[MAX\_NAME\_LEN], filename[MAX\_NAME\_LEN];

printf("Enter username: ");

scanf("%s", username);

int userIndex = findUserIndex(username);

if (userIndex == -1) {

printf("User directory not found.\n");

return;

}

printf("Enter filename to delete: ");

scanf("%s", filename);

for (int i = 0; i < directories[userIndex].fileCount; i++) {

if (strcmp(directories[userIndex].files[i].filename, filename) == 0) {

// Shift files

for (int j = i; j < directories[userIndex].fileCount - 1; j++) {

strcpy(directories[userIndex].files[j].filename, directories[userIndex].files[j + 1].filename);

}

directories[userIndex].fileCount--;

printf("File '%s' deleted from user directory '%s'.\n", filename, username);

return;

}

}

printf("File not found in user directory.\n");

}

void searchFile() {

char username[MAX\_NAME\_LEN], filename[MAX\_NAME\_LEN];

printf("Enter username: ");

scanf("%s", username);

int userIndex = findUserIndex(username);

if (userIndex == -1) {

printf("User directory not found.\n");

return;

}

printf("Enter filename to search: ");

scanf("%s", filename);

for (int i = 0; i < directories[userIndex].fileCount; i++) {

if (strcmp(directories[userIndex].files[i].filename, filename) == 0) {

printf("File '%s' found in user directory '%s'.\n", filename, username);

return;

}

}

printf("File not found in user directory.\n");

}

void displayDirectories() {

if (userCount == 0) {

printf("No user directories available.\n");

return;

}

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

printf("User Directory: %s\n", directories[i].username);

if (directories[i].fileCount == 0) {

printf(" No files.\n");

} else {

printf(" Files:\n");

for (int j = 0; j < directories[i].fileCount; j++) {

printf(" %s\n", directories[i].files[j].filename);

}

}

}

}

int main() {

int choice;

while (1) {

printf("\n--- Two-Level Directory Structure ---\n");

printf("1. Create User Directory\n");

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

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

printf("4. Search File\n");

printf("5. Display Directories\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

createUserDirectory();

break;

case 2:

createFile();

break;

case 3:

deleteFile();

break;

case 4:

searchFile();

break;

case 5:

displayDirectories();

break;

case 6:

printf("Exiting program.\n");

return 0;

default:

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

}

}

return 0;

}

OUTPUT:

