**S20**

Write a simulation program for disk scheduling using SCAN algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.

33, 99, 142, 52, 197, 79, 46, 65

Start Head Position: 72 Direction: User defined

#include <stdio.h>

#include <stdlib.h>

// Function to sort an array

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

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

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

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

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

// Function to find the index of the disk request just before the head position

int findPrevious(int arr[], int n, int head) {

int previous = 0;

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

if (arr[i] < head) {

previous = i;

} else {

break;

}

}

return previous;

}

// Function to simulate SCAN disk scheduling algorithm

void scan(int arr[], int n, int start, int head) {

int totalHeadMovements = 0;

int direction;

// Finding the direction of movement

if (start < head) {

direction = 1; // Moving towards higher block numbers

} else {

direction = -1; // Moving towards lower block numbers

}

// Sorting the disk request array

sort(arr, n);

// Finding the index of the disk request just before the head position

int previousIndex = findPrevious(arr, n, head);

// Serving requests in the order of SCAN algorithm

printf("Order of service: %d ", head);

totalHeadMovements += abs(head - start); // Moving from start to head

// Scan towards higher block numbers

if (direction == 1) {

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

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

totalHeadMovements += abs(head - arr[i]);

head = arr[i];

}

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

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

totalHeadMovements += abs(head - arr[i]);

head = arr[i];

}

}

// Scan towards lower block numbers

else {

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

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

totalHeadMovements += abs(head - arr[i]);

head = arr[i];

}

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

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

totalHeadMovements += abs(head - arr[i]);

head = arr[i];

}

}

printf("\nTotal head movements: %d\n", totalHeadMovements);

}

int main() {

int n, start, head;

// Accepting input from the user

printf("Enter the total number of disk blocks: ");

scanf("%d", &n);

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

printf("Enter the disk request string: ");

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

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

}

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

scanf("%d", &start);

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

scanf("%d", &head);

// Simulating SCAN algorithm

scan(requests, n, start, head);

free(requests);

return 0;

}

* 1. Write an MPI program to find the max number from randomly generated 1000 numbers (stored in array) on a cluster (Hint: Use MPI\_Reduce)

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define ARRAY\_SIZE 1000

int main(int argc, char \*argv[]) {

int rank, size;

int \*data = NULL;

int local\_max = 0, global\_max = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Each process generates its local random numbers

data = (int \*)malloc(ARRAY\_SIZE \* sizeof(int));

if (data == NULL) {

printf("Memory allocation failed\n");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

srand(rank + 1); // Seed based on rank

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

data[i] = rand() % 1000; // Generate random numbers between 0 and 999

}

// Calculate local maximum

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

if (data[i] > local\_max) {

local\_max = data[i];

}

}

// Reduce local maximum to global maximum

MPI\_Reduce(&local\_max, &global\_max, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

// Print results from rank 0

if (rank == 0) {

printf("Global maximum number is: %d\n", global\_max);

}

free(data);

MPI\_Finalize();

return 0;

}

**S21**

* 1. Write a simulation program for disk scheduling using FCFS algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.

55, 58, 39, 18, 90, 160, 150, 38, 184

Start Head Position: 50

#include <stdio.h>

#include <stdlib.h>

// Function to calculate the total head movements

int calculateHeadMovements(int request[], int n, int start) {

int totalHeadMovements = abs(start - request[0]);

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

totalHeadMovements += abs(request[i] - request[i - 1]);

}

return totalHeadMovements;

}

int main() {

int n, start;

// Accepting input from the user

printf("Enter the total number of disk blocks: ");

scanf("%d", &n);

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

printf("Enter the disk request string: ");

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

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

}

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

scanf("%d", &start);

// Displaying the list of requests in the order in which they are served

printf("Order of service: ");

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

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

}

printf("\n");

// Calculating total head movements

int totalHeadMovements = calculateHeadMovements(request, n, start);

printf("Total head movements: %d\n", totalHeadMovements);

free(request);

return 0;

}

Write an MPI program to calculate sum of all even randomly generated 1000 numbers (stored in array) on a cluster

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define ARRAY\_SIZE 1000

int main(int argc, char \*argv[]) {

int rank, size;

int \*data = NULL;

int local\_sum = 0, global\_sum = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Each process generates its local random numbers

data = (int \*)malloc(ARRAY\_SIZE \* sizeof(int));

if (data == NULL) {

printf("Memory allocation failed\n");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

srand(rank + 1); // Seed based on rank

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

data[i] = rand() % 1000; // Generate random numbers between 0 and 999

}

// Calculate local sum of even numbers

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

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

local\_sum += data[i];

}

}

// Reduce local sums to global sum

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// Print results from rank 0

if (rank == 0) {

printf("Global sum of even numbers is: %d\n", global\_sum);

}

free(data);

MPI\_Finalize();

return 0;

}

S22

Write an MPI program to calculate sum of all odd randomly generated 1000 numbers (stored in array) on a cluster.

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define ARRAY\_SIZE 1000

int main(int argc, char \*argv[]) {

int rank, size;

int \*data = NULL;

int local\_sum = 0, global\_sum = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Each process generates its local random numbers

data = (int \*)malloc(ARRAY\_SIZE \* sizeof(int));

if (data == NULL) {

printf("Memory allocation failed\n");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

srand(rank + 1); // Seed based on rank

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

data[i] = rand() % 1000; // Generate random numbers between 0 and 999

}

// Calculate local sum of odd numbers

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

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

local\_sum += data[i];

}

}

// Reduce local sums to global sum

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// Print results from rank 0

if (rank == 0) {

printf("Global sum of odd numbers is: %d\n", global\_sum);

}

free(data);

MPI\_Finalize();

return 0;

}

* 1. Write a program to simulate Sequential (Contiguous) file allocation method. Assume disk with n number of blocks. Give value of n as input. Randomly mark some block as allocated and accordingly maintain the list of free blocks Write menu driver program with menu options as mentioned below and implement each option
     + Show Bit Vector
     + Delete already created file
     + Exit

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to initialize the bit vector (disk) with random allocation

void initializeBitVector(int bitVector[], int n) {

srand(time(NULL));

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

bitVector[i] = rand() % 2; // Randomly mark block as allocated (1) or free (0)

}

}

// Function to display the bit vector (disk)

void showBitVector(int bitVector[], int n) {

printf("Bit Vector (Disk):\n");

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

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

}

printf("\n");

}

// Function to delete a file by marking its allocated blocks as free

void deleteFile(int bitVector[], int n) {

int startBlock;

printf("Enter the starting block of the file to delete (0 to %d): ", n - 1);

scanf("%d", &startBlock);

if (startBlock < 0 || startBlock >= n) {

printf("Invalid block number.\n");

return;

}

if (bitVector[startBlock] == 0) {

printf("No file exists starting from block %d.\n", startBlock);

return;

}

// Mark allocated blocks as free

while (startBlock < n && bitVector[startBlock] == 1) {

bitVector[startBlock] = 0;

startBlock++;

}

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

}

int main() {

int n;

printf("Enter the number of blocks on the disk: ");

scanf("%d", &n);

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

if (bitVector == NULL) {

printf("Memory allocation failed\n");

return 1;

}

// Initialize bit vector with random allocation

initializeBitVector(bitVector, n);

int choice;

do {

printf("\nMenu:\n");

printf("1. Show Bit Vector\n");

printf("2. Delete already created file\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

showBitVector(bitVector, n);

break;

case 2:

deleteFile(bitVector, n);

break;

case 3:

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

break;

default:

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

}

} while (choice != 3);

free(bitVector);

return 0;

}

**S23**

* 1. Consider a system with ‘m’ processes and ‘n’ resource types. Accept number of instances for every resource type. For each process accept the allocation and maximum requirement matrices. Write a program to display the contents of need matrix and to check if the given request of a process can be granted immediately or not

#include <stdio.h>

#include <stdlib.h>

// Function to allocate memory for a matrix

int \*\*allocateMatrix(int m, int n) {

int \*\*matrix = (int \*\*)malloc(m \* sizeof(int \*));

if (matrix == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

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

matrix[i] = (int \*)malloc(n \* sizeof(int));

if (matrix[i] == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

}

return matrix;

}

// Function to free memory allocated for a matrix

void freeMatrix(int \*\*matrix, int m) {

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

free(matrix[i]);

}

free(matrix);

}

// Function to display the contents of a matrix

void displayMatrix(int \*\*matrix, int m, int n) {

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

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

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

}

printf("\n");

}

}

// Function to calculate the need matrix

void calculateNeedMatrix(int \*\*allocation, int \*\*maxRequirement, int \*\*need, int m, int n) {

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

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

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

}

}

}

// Function to check if the request can be granted immediately

int checkRequest(int \*\*allocation, int \*\*maxRequirement, int \*\*need, int \*available, int process, int resource, int \*request) {

// Check if the request is greater than the need

if (request[resource] > need[process][resource]) {

return 0;

}

// Check if the request is greater than the available resources

if (request[resource] > available[resource]) {

return 0;

}

// Check if the request can be granted without exceeding the maximum requirement

if (request[resource] > maxRequirement[process][resource]) {

return 0;

}

return 1; // Request can be granted

}

int main() {

int m, n;

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

scanf("%d", &m);

printf("Enter the number of resource types: ");

scanf("%d", &n);

// Allocate memory for matrices

int \*\*allocation = allocateMatrix(m, n);

int \*\*maxRequirement = allocateMatrix(m, n);

int \*\*need = allocateMatrix(m, n);

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

// Accept number of instances for every resource type

printf("Enter the number of instances for each resource type:\n");

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

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

}

// Accept allocation matrix

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

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

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

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

}

}

// Accept maximum requirement matrix

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

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

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

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

}

}

// Calculate need matrix

calculateNeedMatrix(allocation, maxRequirement, need, m, n);

// Display need matrix

printf("Need matrix:\n");

displayMatrix(need, m, n);

// Accept request

int process, resource;

printf("Enter the process number requesting for a resource: ");

scanf("%d", &process);

printf("Enter the resource number: ");

scanf("%d", &resource);

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

printf("Enter the request for each resource type:\n");

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

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

}

// Check if the request can be granted

if (checkRequest(allocation, maxRequirement, need, available, process, resource, request)) {

printf("Request can be granted immediately.\n");

} else {

printf("Request cannot be granted immediately.\n");

}

// Free allocated memory

freeMatrix(allocation, m);

freeMatrix(maxRequirement, m);

freeMatrix(need, m);

free(available);

free(request);

return 0;

}

* 1. Write a simulation program for disk scheduling using SSTF algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.

24, 90, 133, 43, 188, 70, 37, 55

Start Head Position: 58

#include <stdio.h>

#include <stdlib.h>

// Function to calculate the absolute difference between two numbers

int absDiff(int a, int b) {

return abs(a - b);

}

// Function to find the index of the nearest request to the current head position

int findNearestRequest(int request[], int n, int head) {

int minDistance = absDiff(head, request[0]);

int nearestIndex = 0;

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

int distance = absDiff(head, request[i]);

if (distance < minDistance) {

minDistance = distance;

nearestIndex = i;

}

}

return nearestIndex;

}

// Function to simulate SSTF disk scheduling algorithm

void sstf(int request[], int n, int start, int head) {

int totalHeadMovements = 0;

int currentPosition = start;

printf("Order of service: ");

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

int nearestIndex = findNearestRequest(request, n, currentPosition);

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

totalHeadMovements += absDiff(currentPosition, request[nearestIndex]);

currentPosition = request[nearestIndex];

request[nearestIndex] = -1; // Mark the request as serviced

}

printf("\nTotal head movements: %d\n", totalHeadMovements);

}

int main() {

int n, start;

// Accepting input from the user

printf("Enter the total number of disk blocks: ");

scanf("%d", &n);

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

printf("Enter the disk request string: ");

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

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

}

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

scanf("%d", &start);

// Simulating SSTF algorithm

sstf(request, n, start, start);

free(request);

return 0;

}

S24

Write an MPI program to calculate sum of all odd randomly generated 1000 numbers (stored in array) on a cluster

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define ARRAY\_SIZE 1000

int main(int argc, char \*argv[]) {

int rank, size;

int \*data = NULL;

int local\_sum = 0, global\_sum = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Calculate local sum for each process

int chunk\_size = ARRAY\_SIZE / size;

int remainder = ARRAY\_SIZE % size;

// Allocate memory for local data

data = (int \*)malloc((chunk\_size + (rank < remainder ? 1 : 0)) \* sizeof(int));

if (data == NULL) {

printf("Memory allocation failed\n");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

// Generate random numbers for each process

srand(rank + 1); // Seed based on rank

for (int i = 0; i < chunk\_size + (rank < remainder ? 1 : 0); i++) {

data[i] = rand() % 1000; // Generate random numbers between 0 and 999

}

// Calculate local sum of odd numbers

for (int i = 0; i < chunk\_size + (rank < remainder ? 1 : 0); i++) {

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

local\_sum += data[i];

}

}

// Reduce local sums to global sum

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// Print results from rank 0

if (rank == 0) {

printf("Global sum of odd numbers is: %d\n", global\_sum);

}

free(data);

MPI\_Finalize();

return 0;

}

* 1. Write a C program to simulate Banker’s algorithm for the purpose of deadlock avoidance.The following snapshot of system, A, B, C and D are the resource type.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Proces s | Allocation | | | Max | | | Available | | |
|  | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P1 | 2 | 0 | 0 | 2 | 0 | 2 |  |  |  |
| P2 | 3 | 0 | 3 | 0 | 0 | 0 |  |  |  |
| P3 | 2 | 1 | 1 | 1 | 0 | 0 |  |  |  |
| P4 | 0 | 0 | 2 | 0 | 0 | 2 |  |  |  |

* + 1. Calculate and display the content of need matrix?

Is the system in safe state? If display the safe sequence.

#include <stdio.h>

#define NUM\_PROCESSES 5

#define NUM\_RESOURCES 3

int available[NUM\_RESOURCES] = {0};

int allocation[NUM\_PROCESSES][NUM\_RESOURCES] = {{0}};

int max[NUM\_PROCESSES][NUM\_RESOURCES] = {{0}};

int need[NUM\_PROCESSES][NUM\_RESOURCES] = {{0}};

int finish[NUM\_PROCESSES] = {0};

// Function to calculate the need matrix

void calculateNeed() {

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

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

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

}

}

}

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

int isSafe() {

int work[NUM\_RESOURCES];

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

work[i] = available[i];

}

int safeSequence[NUM\_PROCESSES];

int count = 0;

while (count < NUM\_PROCESSES) {

int found = 0;

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

if (!finish[i]) {

int j;

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

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

break;

}

}

if (j == NUM\_RESOURCES) {

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

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

}

safeSequence[count++] = i;

finish[i] = 1;

found = 1;

}

}

}

if (!found) {

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

return 0;

}

}

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

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

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

}

printf("\n");

return 1;

}

int main() {

// Input allocation matrix

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

{0, 1, 0},

{2, 0, 0},

{3, 0, 3},

{2, 1, 1},

{0, 0, 2}

};

// Input max matrix

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

{0, 0, 0},

{2, 0, 2},

{0, 0, 0},

{1, 0, 0},

{0, 0, 2}

};

// Input available resources

int availableInput[NUM\_RESOURCES] = {0, 0, 0};

// Copy input data to global arrays

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

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

allocation[i][j] = allocationInput[i][j];

max[i][j] = maxInput[i][j];

}

}

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

available[i] = availableInput[i];

}

// Calculate and display the need matrix

calculateNeed();

printf("Need matrix:\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");

}

// Check if the system is in a safe state

isSafe();

return 0;

}

S25

* 1. Write a simulation program for disk scheduling using LOOK algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.

86, 147, 91, 170, 95, 130, 102, 70

Starting Head position= 125 Direction: User Defined

#include <stdio.h>

#include <stdlib.h>

// Function to sort an array in ascending order

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

int temp;

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

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

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

temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

// Function to simulate LOOK disk scheduling algorithm

void look(int request[], int n, int start, int direction) {

int totalHeadMovements = 0;

int currentPosition = start;

int currentIndex = -1;

// Sort the request array

sortArray(request, n);

// Find the index of current head position

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

if (request[i] >= start) {

currentIndex = i;

break;

}

}

printf("Order of service: ");

// Traverse the requests based on the direction

if (direction == 1) {

// Moving towards higher block numbers

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

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

totalHeadMovements += abs(currentPosition - request[i]);

currentPosition = request[i];

}

// Traverse back to lower block numbers if there are requests

if (currentIndex > 0) {

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

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

totalHeadMovements += abs(currentPosition - request[i]);

currentPosition = request[i];

}

}

} else {

// Moving towards lower block numbers

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

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

totalHeadMovements += abs(currentPosition - request[i]);

currentPosition = request[i];

}

// Traverse forward to higher block numbers if there are requests

if (currentIndex < n - 1) {

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

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

totalHeadMovements += abs(currentPosition - request[i]);

currentPosition = request[i];

}

}

}

printf("\nTotal head movements: %d\n", totalHeadMovements);

}

int main() {

int n, start, direction;

// Accepting input from the user

printf("Enter the total number of disk blocks: ");

scanf("%d", &n);

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

printf("Enter the disk request string: ");

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

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

}

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

scanf("%d", &start);

printf("Enter the direction (1 for towards higher block numbers, 0 for towards lower block numbers): ");

scanf("%d", &direction);

// Simulating LOOK algorithm

look(request, n, start, direction);

free(request);

return 0;

}

Write a program to simulate Linked file allocation method. Assume disk with n number of blocks. Give value of n as input. Randomly mark some block as allocated and accordingly maintain the list of free blocks Write menu driver program with menu options as mentioned below and implement each option.

* + - Show Bit Vector
    - Create New File
    - Show Directory

Exit

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_BLOCKS 100

typedef struct File {

int start\_block;

int length;

struct File \*next;

} File;

int allocated\_blocks[MAX\_BLOCKS] = {0};

File \*directory = NULL;

// Function to initialize the bit vector (disk) with random allocation

void initializeBitVector(int n) {

srand(time(NULL));

int num\_allocated = rand() % (n + 1); // Randomly choose the number of allocated blocks

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

int block = rand() % n; // Randomly choose a block

while (allocated\_blocks[block]) {

block = rand() % n; // Choose another block if it's already allocated

}

allocated\_blocks[block] = 1; // Mark block as allocated

}

}

// Function to display the bit vector (disk)

void showBitVector(int n) {

printf("Bit Vector (Disk):\n");

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

printf("%d ", allocated\_blocks[i]);

}

printf("\n");

}

// Function to create a new file

void createNewFile(int n) {

int start, length;

printf("Enter starting block number for the new file (0 to %d): ", n - 1);

scanf("%d", &start);

printf("Enter length of the new file: ");

scanf("%d", &length);

if (start < 0 || start >= n || length <= 0 || start + length > n) {

printf("Invalid input. File creation failed.\n");

return;

}

// Check if blocks are already allocated

for (int i = start; i < start + length; i++) {

if (allocated\_blocks[i]) {

printf("Blocks are already allocated. File creation failed.\n");

return;

}

}

// Allocate blocks for the file

for (int i = start; i < start + length; i++) {

allocated\_blocks[i] = 1;

}

// Update the directory

File \*new\_file = (File \*)malloc(sizeof(File));

if (new\_file == NULL) {

printf("Memory allocation failed. File creation failed.\n");

return;

}

new\_file->start\_block = start;

new\_file->length = length;

new\_file->next = directory;

directory = new\_file;

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

}

// Function to display the directory

void showDirectory() {

printf("Directory:\n");

File \*current = directory;

while (current != NULL) {

printf("Start Block: %d, Length: %d\n", current->start\_block, current->length);

current = current->next;

}

}

int main() {

int n;

printf("Enter the number of blocks on the disk: ");

scanf("%d", &n);

initializeBitVector(n);

int choice;

do {

printf("\nMenu:\n");

printf("1. Show Bit Vector\n");

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

printf("3. Show Directory\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

showBitVector(n);

break;

case 2:

createNewFile(n);

break;

case 3:

showDirectory();

break;

case 4:

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

break;

default:

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

}

} while (choice != 4);

// Free memory allocated for the directory

File \*current = directory;

while (current != NULL) {

File \*temp = current;

current = current->next;

free(temp);

}

return 0;

}