OS-II

slip 1
Q1.Q.1) Write a C Menu driven Program to implement following functionality
a) Accept Available
b) Display Allocation, Max
c) Display the contents of need matrix
d) Display Available
Process Allocation Max Available
A B C A B C A B C
P0 2 3 2 9 7 5 3 3 2
P1 4 0 0 5 2 2
P2 5 0 4 1 0 4
P3 4 3 3 4 4 4
P4 2 2 4 6 5 5

ans:

```
#include <stdio.h>

#define MAX_PROCESS 5
#define MAX_RESOURCES 3

int allocation[MAX_PROCESS][MAX_RESOURCES];
int max[MAX_PROCESS][MAX_RESOURCES];
int need[MAX_PROCESS][MAX_RESOURCES];
int available[MAX_RESOURCES];

void displayAllocationMaxAvailable() {
    printf("\nProcess Allocation Max Available\n");
    printf("A B C A B C A B C\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", allocation[i][j]);
        }
        printf(" ");
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", max[i][j]);
        }
        printf(" ");
        if (i == 0) {
            for (int j = 0; j < MAX_RESOURCES; j++) {
                printf("%d ", available[j]);
            }
        }
        printf("\n");
    }
}

void displayNeed() {
```

```
    printf("\nNeed Matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}

int main() {
    printf("Enter available resources (A B C): ");
    scanf("%d %d %d", &available[0], &available[1], &available[2]);

    printf("Enter allocation matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    printf("Enter max matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }

    int choice;
    do {
        printf("\nMenu:\n");
        printf("1. Display Allocation, Max, Available\n");
        printf("2. Display Need Matrix\n");
        printf("3. Display Available Resources\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                displayAllocationMaxAvailable();
                break;
            case 2:
                displayNeed();
                break;
            case 3:
                printf("\nAvailable Resources: %d %d %d\n", available[0],
```

```
available[1], available[2]);
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a number between 1 to 4.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Q.2 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

Ans:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int total_blocks, *request_string, head_position, total_head_movements = 0;

    // Accepting input from the user
    printf("Enter total number of disk blocks: ");
    scanf("%d", &total_blocks);

    request_string = (int *)malloc(total_blocks * sizeof(int));

    printf("Enter disk request string:\n");
    for (int i = 0; i < total_blocks; i++) {
        scanf("%d", &request_string[i]);
    }

    printf("Enter current head position: ");
    scanf("%d", &head_position);

    // FCFS Algorithm
    printf("\nOrder of serving requests:\n");
    for (int i = 0; i < total_blocks; i++) {
        printf("%d ", request_string[i]);
        total_head_movements += abs(head_position - request_string[i]);
        head_position = request_string[i];
```

```
    }

    // Displaying total head movements
    printf("\nTotal number of head movements: %d\n", total_head_movements);

    free(request_string);
    return 0;
}
```

Slip 2

Q.1 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

ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100

// Node structure to represent a block
struct Block {
    int block_number;
    bool allocated;
    struct Block *next;
};

// Function to initialize the disk blocks
void initializeDisk(struct Block *disk[], int n) {
    for (int i = 0; i < n; i++) {
        disk[i] = (struct Block *)malloc(sizeof(struct Block));
        disk[i]->block_number = i;
        disk[i]->allocated = false;
        disk[i]->next = NULL;
    }
}

// Function to randomly mark some blocks as allocated
```

```
void randomlyAllocate(struct Block *disk[], int n) {
    srand(time(NULL));
    int num_allocated = rand() % (n / 2) + 1; // Randomly allocate up to half of
the blocks
    for (int i = 0; i < num_allocated; i++) {
        int block_index = rand() % n;
        if (!disk[block_index]->allocated) {
            disk[block_index]->allocated = true;
        }
    }
}

// Function to display the bit vector
void showBitVector(struct Block *disk[], int n) {
    printf("\nBit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]->allocated);
    }
    printf("\n");
}

// Function to create a new file
void createNewFile(struct Block *disk[], int n) {
    int start_block;
    printf("Enter the starting block number for the new file: ");
    scanf("%d", &start_block);
    if (start_block < 0 || start_block >= n) {
        printf("Invalid starting block number!\n");
        return;
    }
    if (disk[start_block]->allocated) {
        printf("Block %d is already allocated!\n", start_block);
        return;
    }

    struct Block *current = disk[start_block];
    while (current->next != NULL) {
        current = current->next;
    }
    current->allocated = true;
    printf("File created successfully!\n");
}

// Function to display the directory
void showDirectory(struct Block *disk[], int n) {
    printf("\nDirectory:\n");
    printf("Block\tAllocated\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\n", i, disk[i]->allocated);
    }
```

```
}

int main() {
    struct Block *disk[MAX_BLOCKS];
    int n, choice;

    printf("Enter the number of blocks in the disk: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX_BLOCKS) {
        printf("Invalid number of blocks!\n");
        return 1;
    }

    initializeDisk(disk, n);
    randomlyAllocate(disk, n);

    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(disk, n);
                break;
            case 2:
                createNewFile(disk, n);
                break;
            case 3:
                showDirectory(disk, n);
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a number between 1 to 4.\n");
        }
    } while (choice != 4);

    // Free allocated memory
    for (int i = 0; i < n; i++) {
        free(disk[i]);
    }

    return 0;
```

```
}
```

Q.2 Write an MPI program to calculate sum of randomly generated 1000 numbers (stored in array) on a cluster

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int local_sum = 0, global_sum = 0;
    int data[ARRAY_SIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed random number generator differently on each process
    srand(rank);

    // Generate random numbers on each process
    for (int i = 0; i < ARRAY_SIZE; i++) {
        data[i] = rand() % 100;
    }

    // Calculate local sum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        local_sum += data[i];
    }

    // Reduce local sums to get the global sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Global Sum: %d\n", global_sum);
    }

    MPI_Finalize();
    return 0;
}
```

Slip 3

Q.1 Write a C program to simulate Banker’s algorithm for the purpose of deadlock

avoidance. Consider the following snapshot of system, A, B, C and D is the resource type.

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

a) Calculate and display the content of need matrix?
b) Is the system in safe state? If display the safe sequence.

ans:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

int available[MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
bool finish[MAX_PROCESSES];

// Function to initialize the system state
void initializeSystem() {
    // Initialize the allocation, max, and need matrices
    int allocationArray[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 0, 1, 2},
        {1, 0, 0, 0},
        {1, 3, 5, 4},
        {0, 6, 3, 2},
        {0, 0, 1, 4}
    };

    int maxArray[MAX_PROCESSES][MAX_RESOURCES] = {
        {1, 5, 2, 0},
        {1, 7, 5, 0},
        {2, 3, 5, 6},
        {0, 6, 5, 2},
        {0, 6, 5, 6}
    };

    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            allocation[i][j] = allocationArray[i][j];
            max[i][j] = maxArray[i][j];
            need[i][j] = max[i][j] - allocation[i][j];
        }
```

```
        finish[i] = false;
    }

    // Initialize the available resources
    printf("Enter available resources (A B C D): ");
    scanf("%d %d %d %d", &available[0], &available[1], &available[2],
&available[3]);
}

// Function to find a safe sequence if it exists
bool findSafeSequence(int safeSequence[]) {
    int work[MAX_RESOURCES];
    for (int i = 0; i < MAX_RESOURCES; i++) {
        work[i] = available[i];
    }

    int count = 0;
    while (count < MAX_PROCESSES) {
        bool found = false;
        for (int i = 0; i < MAX_PROCESSES; i++) {
            if (!finish[i]) {
                bool canExecute = true;
                for (int j = 0; j < MAX_RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        canExecute = false;
                        break;
                    }
                }
                if (canExecute) {
                    for (int j = 0; j < MAX_RESOURCES; j++) {
                        work[j] += allocation[i][j];
                    }
                    safeSequence[count++] = i;
                    finish[i] = true;
                    found = true;
                }
            }
        }
        if (!found) {
            return false; // No safe sequence exists
        }
    }
    return true; // Safe sequence found
}

int main() {
    initializeSystem();

    // Display the content of the need matrix
    printf("\nNeed Matrix:\n");
```

```
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    // Check if the system is in a safe state and display the safe sequence if it
exists
    int safeSequence[MAX_PROCESSES];
    if (findSafeSequence(safeSequence)) {
        printf("\nSystem is in safe state.\nSafe Sequence: ");
        for (int i = 0; i < MAX_PROCESSES; i++) {
            printf("P%d ", safeSequence[i]);
        }
        printf("\n");
    } else {
        printf("\nSystem is not in safe state.\n");
    }

    return 0;
}
```

Q.2 Write an MPI program to calculate sum and average of randomly generated 1000 numbers (stored in array) on a cluster

ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int local_sum = 0, global_sum = 0;
    double local_average = 0.0, global_average = 0.0;
    int data[ARRAY_SIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed random number generator differently on each process
    srand(rank);

    // Generate random numbers on each process
    for (int i = 0; i < ARRAY_SIZE; i++) {
```

```
        data[i] = rand() % 100;
    }

    // Calculate local sum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        local_sum += data[i];
    }

    // Reduce local sums to get the global sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Calculate local average
    local_average = (double)local_sum / ARRAY_SIZE;

    // Reduce local averages to get the global average
    MPI_Reduce(&local_average, &global_average, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        global_average /= size;
        printf("Global Sum: %d\n", global_sum);
        printf("Global Average: %.2f\n", global_average);
    }

    MPI_Finalize();
    return 0;
}
```

Slip 4

Q.1 Implement the Menu driven Banker's algorithm for accepting Allocation, Max from user.
a) Accept Available
b) Display Allocation, Max
c) Find Need and display It,
d) Display Available
Consider the system with 3 resources types A,B, and C with 7,2,6 instances respectively.
Consider the following snapshot:

Process	Allocation			Request		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	4	0	0	5	2	2
P2	5	0	4	1	0	4
P3	4	3	3	4	4	4
P4	2	2	4	6	5	5

Ans:

```
#include <stdio.h>
```

```
#define MAX_PROCESS 5
#define MAX_RESOURCES 3

int allocation[MAX_PROCESS][MAX_RESOURCES];
int max[MAX_PROCESS][MAX_RESOURCES];
int need[MAX_PROCESS][MAX_RESOURCES];
int available[MAX_RESOURCES] = {7, 2, 6};

// Function to accept allocation matrix from the user
void acceptAllocation() {
    printf("Enter Allocation Matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }
}

// Function to accept max matrix from the user
void acceptMax() {
    printf("Enter Max Matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

// Function to display allocation and max matrices
void displayAllocationMax() {
    printf("\nAllocation Matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", allocation[i][j]);
        }
        printf("\n");
    }

    printf("\nMax Matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\n");
```

```
    }
}

// Function to calculate and display the need matrix
void displayNeed() {
    printf("\nNeed Matrix:\n");
    for (int i = 0; i < MAX_PROCESS; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}

// Function to display the available resources
void displayAvailable() {
    printf("\nAvailable Resources: ");
    for (int i = 0; i < MAX_RESOURCES; i++) {
        printf("%d ", available[i]);
    }
    printf("\n");
}

int main() {
    int choice;
    do {
        printf("\nMenu:\n");
        printf("1. Accept Allocation\n");
        printf("2. Accept Max\n");
        printf("3. Display Allocation, Max\n");
        printf("4. Find and Display Need\n");
        printf("5. Display Available Resources\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                acceptAllocation();
                break;
            case 2:
                acceptMax();
                break;
            case 3:
                displayAllocationMax();
                break;
            case 4:
                displayNeed();
                break;
```

```
            case 5:
                displayAvailable();
                break;
            case 6:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a number between 1 to 6.\n");
        }
    } while (choice != 6);

    return 0;
}
```

Q.2 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.
86, 147, 91, 170, 95, 130, 102, 70
Starting Head position= 125
Direction: Left

ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort the disk request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (request_string[j] > request_string[j+1]) {
                int temp = request_string[j];
                request_string[j] = request_string[j+1];
                request_string[j+1] = temp;
            }
        }
    }
}

int main() {
    int total_blocks, *request_string, head_position, total_head_movements = 0;
    char direction;

    // Accepting input from the user
    printf("Enter total number of disk blocks: ");
    scanf("%d", &total_blocks);
```

```
request_string = (int *)malloc(total_blocks * sizeof(int));

printf("Enter disk request string:\n");
for (int i = 0; i < total_blocks; i++) {
    scanf("%d", &request_string[i]);
}

printf("Enter starting head position: ");
scanf("%d", &head_position);

printf("Enter direction (L for left, R for right): ");
scanf(" %c", &direction);

// Sorting the request string
sort(request_string, total_blocks);

// Adding the starting head position to the request string
request_string[total_blocks] = head_position;
total_blocks++;

// SCAN Algorithm
if (direction == 'L') { // Left direction
    for (int i = 0; i < total_blocks; i++) {
        if (request_string[i] >= head_position) {
            total_head_movements += abs(head_position - request_string[i]);
            head_position = request_string[i];
        }
    }
    for (int i = total_blocks - 1; i >= 0; i--) {
        if (request_string[i] < head_position) {
            total_head_movements += abs(head_position - request_string[i]);
            head_position = request_string[i];
        }
    }
} else if (direction == 'R') { // Right direction
    for (int i = total_blocks - 1; i >= 0; i--) {
        if (request_string[i] <= head_position) {
            total_head_movements += abs(head_position - request_string[i]);
            head_position = request_string[i];
        }
    }
    for (int i = 0; i < total_blocks; i++) {
        if (request_string[i] > head_position) {
            total_head_movements += abs(head_position - request_string[i]);
            head_position = request_string[i];
        }
    }
} else {
    printf("Invalid direction! Please enter L for left or R for right.\n");
```

```
        return 1;
    }

    // Displaying the list of requests in the order they are served
    printf("\nList of requests served:\n");
    for (int i = 0; i < total_blocks - 1; i++) {
        printf("%d -> ", request_string[i]);
    }
    printf("%d\n", request_string[total_blocks - 1]);

    // Displaying the total number of head movements
    printf("Total number of head movements: %d\n", total_head_movements);

    free(request_string);
    return 0;
}
```

Slip 5

Q.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

Ans:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int available[MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];

// Function to accept the number of instances for each resource type
void acceptResourceInstances(int n) {
    printf("Enter number of instances for each resource type:\n");
    for (int i = 0; i < n; i++) {
        printf("Resource %d: ", i);
        scanf("%d", &available[i]);
    }
}

// Function to accept allocation matrix from the user for each process
void acceptAllocation(int m, int n) {
    printf("Enter Allocation Matrix:\n");
    for (int i = 0; i < m; i++) {
```

```
        printf("Process %d: ", i);
        for (int j = 0; j < n; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }
}

// Function to accept max matrix from the user for each process
void acceptMax(int m, int n) {
    printf("Enter Max Matrix:\n");
    for (int i = 0; i < m; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < n; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

// Function to display the contents of the need matrix
void displayNeed(int m, int n) {
    printf("\nNeed Matrix:\n");
    for (int i = 0; i < m; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < n; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}

// Function to check if a given request of a process can be granted immediately
bool checkRequest(int process, int request[], int n) {
    for (int i = 0; i < n; i++) {
        if (request[i] > need[process][i] || request[i] > available[i]) {
            return false;
        }
    }
    return true;
}

int main() {
    int m, n;
    printf("Enter number of processes: ");
    scanf("%d", &m);
    printf("Enter number of resource types: ");
    scanf("%d", &n);

    acceptResourceInstances(n);
    acceptAllocation(m, n);
```

```
    acceptMax(m, n);
    displayNeed(m, n);

    // Check if a given request of a process can be granted immediately
    int process;
    printf("\nEnter process number to check request: ");
    scanf("%d", &process);
    int request[MAX_RESOURCES];
    printf("Enter request for process %d: ", process);
    for (int i = 0; i < n; i++) {
        scanf("%d", &request[i]);
    }
    if (checkRequest(process, request, n)) {
        printf("Request can be granted immediately.\n");
    } else {
        printf("Request cannot be granted immediately.\n");
    }

    return 0;
}
```

Q.2 Write an MPI program to find the max number from randomly generated 1000 numbers
(stored in array) on a cluster (Hint: Use MPI_Reduce)

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int local_max = 0, global_max = 0;
    int data[ARRAY_SIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed random number generator differently on each process
    srand(rank);

    // Generate random numbers on each process
    for (int i = 0; i < ARRAY_SIZE; i++) {
        data[i] = rand() % 1000;
    }
```

```
    // Find local maximum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        if (data[i] > local_max) {
            local_max = data[i];
        }
    }

    // Reduce local maximum to get the global maximum
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Global Max: %d\n", global_max);
    }

    MPI_Finalize();
    return 0;
}
```

Slip 6

Q.1 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

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100

// Node structure to represent a block
struct Block {
    int block_number;
    bool allocated;
    struct Block *next;
};

// Function to initialize the disk blocks
void initializeDisk(struct Block *disk[], int n) {
    for (int i = 0; i < n; i++) {
```

```
        disk[i] = (struct Block *)malloc(sizeof(struct Block));
        disk[i]->block_number = i;
        disk[i]->allocated = false;
        disk[i]->next = NULL;
    }
}

// Function to randomly mark some blocks as allocated
void randomlyAllocate(struct Block *disk[], int n) {
    srand(time(NULL));
    int num_allocated = rand() % (n / 2) + 1; // Randomly allocate up to half of
the blocks
    for (int i = 0; i < num_allocated; i++) {
        int block_index = rand() % n;
        if (!disk[block_index]->allocated) {
            disk[block_index]->allocated = true;
        }
    }
}

// Function to display the bit vector
void showBitVector(struct Block *disk[], int n) {
    printf("\nBit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]->allocated);
    }
    printf("\n");
}

// Function to create a new file
void createNewFile(struct Block *disk[], int n) {
    int start_block;
    printf("Enter the starting block number for the new file: ");
    scanf("%d", &start_block);
    if (start_block < 0 || start_block >= n) {
        printf("Invalid starting block number!\n");
        return;
    }
    if (disk[start_block]->allocated) {
        printf("Block %d is already allocated!\n", start_block);
        return;
    }

    struct Block *current = disk[start_block];
    while (current->next != NULL) {
        current = current->next;
    }
    current->allocated = true;
    printf("File created successfully!\n");
}
```

```
// Function to display the directory
void showDirectory(struct Block *disk[], int n) {
    printf("\nDirectory:\n");
    printf("Block\tAllocated\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\n", i, disk[i]->allocated);
    }
}

int main() {
    struct Block *disk[MAX_BLOCKS];
    int n, choice;

    printf("Enter the number of blocks in the disk: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX_BLOCKS) {
        printf("Invalid number of blocks!\n");
        return 1;
    }

    initializeDisk(disk, n);
    randomlyAllocate(disk, n);

    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(disk, n);
                break;
            case 2:
                createNewFile(disk, n);
                break;
            case 3:
                showDirectory(disk, n);
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a number between 1 to 4.\n");
        }
```

```
    } while (choice != 4);

    // Free allocated memory
    for (int i = 0; i < n; i++) {
        free(disk[i]);
    }

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-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..
80, 150, 60,135, 40, 35, 170
Starting Head Position: 70
Direction: Right

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort the disk request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (request_string[j] > request_string[j+1]) {
                int temp = request_string[j];
                request_string[j] = request_string[j+1];
                request_string[j+1] = temp;
            }
        }
    }
}

// Function to simulate disk scheduling using C-SCAN algorithm
void cScan(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;

    // Sort the request string
    sort(request_string, n);

    // Determine the end of the disk
    int end_position = 199; // Total number of blocks - 1

    // If the direction is left, set the end position to 0
```

```
    if (direction == 'L') {
        end_position = 0;
    }

    // Move the head to the start position
    int current_position = start_position;

    // Scan towards the end of the disk
    if (direction == 'R') {
        // Scan to the right until reaching the end of the disk
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position && request_string[i] <=
end_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        total_head_movements += abs(end_position - current_position);
        printf("%d -> ", end_position);

        // Scan back to the beginning of the disk
        total_head_movements += end_position; // Add the head movement from end to
0
        printf("0 -> ");
        current_position = 0;
    } else if (direction == 'L') {
        // Scan to the left until reaching the beginning of the disk
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position && request_string[i] >=
end_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        total_head_movements += abs(end_position - current_position);
        printf("%d -> ", end_position);

        // Scan back to the end of the disk
        total_head_movements += (199 - end_position); // Add the head movement from
0 to end
        printf("199 -> ");
        current_position = 199;
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
```

```
    int request_string[] = {80, 150, 60, 135, 40, 35, 170};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 70;
    char direction = 'R'; // Right direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    cScan(request_string, n, start_position, direction);

    return 0;
}
```

Slip 7

Q.1 Consider the following snapshot of the system.

Proces
Allocation Max Available
A B C D A B C D A B C D
P0 2 0 0 1 4 2 1 2 3 3 2 1
P1 3 1 2 1 5 2 5 2
P2 2 1 0 3 2 3 1 6
P3 1 3 1 2 1 4 2 4
P4 1 4 3 2 3 6 6 5

Using Resource –Request algorithm to Check whether the current system is in safe state
or not

Ans:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

int allocation[MAX_PROCESSES][MAX_RESOURCES] = {
    {2, 0, 0, 1},
    {3, 1, 2, 1},
    {2, 1, 0, 3},
    {1, 3, 1, 2},
    {1, 4, 3, 2}
};

int max[MAX_PROCESSES][MAX_RESOURCES] = {
    {4, 2, 1, 2},
```

```
    {5, 2, 5, 2},
    {2, 3, 1, 6},
    {1, 4, 2, 4},
    {3, 6, 6, 5}
};

int available[MAX_RESOURCES] = {3, 3, 2, 2};
bool finish[MAX_PROCESSES] = {false};

// Function to check if the system is in a safe state
bool isSafeState() {
    int work[MAX_RESOURCES];
    bool tempFinish[MAX_PROCESSES];

    // Initialize work array
    for (int i = 0; i < MAX_RESOURCES; i++) {
        work[i] = available[i];
    }

    // Initialize temporary finish array
    for (int i = 0; i < MAX_PROCESSES; i++) {
        tempFinish[i] = finish[i];
    }

    bool found;
    do {
        found = false;
        for (int i = 0; i < MAX_PROCESSES; i++) {
            if (!tempFinish[i]) {
                bool canAllocate = true;
                for (int j = 0; j < MAX_RESOURCES; j++) {
                    if (max[i][j] - allocation[i][j] > work[j]) {
                        canAllocate = false;
                        break;
                    }
                }
                if (canAllocate) {
                    // Simulate allocation
                    for (int j = 0; j < MAX_RESOURCES; j++) {
                        work[j] += allocation[i][j];
                    }
                    tempFinish[i] = true;
                    found = true;
                }
            }
        }
    } while (found);

    // Check if all processes are finished
    for (int i = 0; i < MAX_PROCESSES; i++) {
```

```
        if (!tempFinish[i]) {
            return false;
        }
    }

    return true;
}

int main() {
    if (isSafeState()) {
        printf("The current system is in a safe state.\n");
    } else {
        printf("The current system is not in a safe state.\n");
    }

    return 0;
}
```

Q.2 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.
82, 170, 43, 140, 24, 16, 190
Starting Head Position: 50
Direction: Right

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort the disk request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (request_string[j] > request_string[j+1]) {
                int temp = request_string[j];
                request_string[j] = request_string[j+1];
                request_string[j+1] = temp;
            }
        }
    }
}

// Function to simulate disk scheduling using SCAN algorithm
void scan(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;
```

```
    // Sort the request string
    sort(request_string, n);

    // Find the end of the disk
    int end_position = 199; // Total number of blocks - 1

    // Move the head to the start position
    int current_position = start_position;

    // Scan towards the end of the disk
    if (direction == 'R') {
        // Scan to the right until reaching the end of the disk
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position && request_string[i] <=
end_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        total_head_movements += abs(end_position - current_position);
        printf("%d -> ", end_position);
    } else if (direction == 'L') {
        // Scan to the left until reaching the beginning of the disk
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position && request_string[i] >= 0) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        total_head_movements += abs(current_position); // Add the head movement
from 0 to current position
        printf("0 -> ");
        current_position = 0;
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {82, 170, 43, 140, 24, 16, 190};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 50;
    char direction = 'R'; // Right direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
```

```
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    scan(request_string, n, start_position, direction);

    return 0;
}
```

slip 8

Q.1 Write a program to simulate 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 above and implement each option.

- Show Bit Vector
- Create New File
- Show Directory
- Exit

ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100

// Node structure to represent a block
struct Block {
    int block_number;
    bool allocated;
};

// Function to initialize the disk blocks
void initializeDisk(struct Block disk[], int n) {
    for (int i = 0; i < n; i++) {
        disk[i].block_number = i;
        disk[i].allocated = false;
    }
}

// Function to randomly mark some blocks as allocated
void randomlyAllocate(struct Block disk[], int n) {
```

```
    srand(time(NULL));
    int num_allocated = rand() % (n / 2) + 1; // Randomly allocate up to half of
the blocks
    for (int i = 0; i < num_allocated; i++) {
        int block_index = rand() % n;
        if (!disk[block_index].allocated) {
            disk[block_index].allocated = true;
        }
    }
}

// Function to display the bit vector
void showBitVector(struct Block disk[], int n) {
    printf("\nBit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i].allocated);
    }
    printf("\n");
}

// Function to create a new file
void createNewFile(struct Block disk[], int n) {
    int start_block;
    printf("Enter the starting block number for the new file: ");
    scanf("%d", &start_block);
    if (start_block < 0 || start_block >= n) {
        printf("Invalid starting block number!\n");
        return;
    }
    if (disk[start_block].allocated) {
        printf("Block %d is already allocated!\n", start_block);
        return;
    }

    int length;
    printf("Enter the length of the file: ");
    scanf("%d", &length);
    if (start_block + length > n) {
        printf("File cannot fit in the disk!\n");
        return;
    }

    for (int i = start_block; i < start_block + length; i++) {
        if (disk[i].allocated) {
            printf("Block %d is already allocated!\n", i);
            return;
        }
    }

    for (int i = start_block; i < start_block + length; i++) {
```

```
        disk[i].allocated = true;
    }

    printf("File created successfully!\n");
}

// Function to display the directory
void showDirectory(struct Block disk[], int n) {
    printf("\nDirectory:\n");
    printf("Block\tAllocated\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\n", i, disk[i].allocated);
    }
}

int main() {
    struct Block disk[MAX_BLOCKS];
    int n, choice;

    printf("Enter the number of blocks in the disk: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX_BLOCKS) {
        printf("Invalid number of blocks!\n");
        return 1;
    }

    initializeDisk(disk, n);
    randomlyAllocate(disk, n);

    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(disk, n);
                break;
            case 2:
                createNewFile(disk, n);
                break;
            case 3:
                showDirectory(disk, n);
                break;
            case 4:
```

```
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a number between 1 to 4.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Q.2 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.
186, 89, 44, 70, 102, 22, 51, 124
Start Head Position: 70

ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Function to calculate absolute difference
int absDiff(int a, int b) {
    return abs(a - b);
}

// Function to simulate disk scheduling using SSTF algorithm
void sstf(int *request_string, int n, int start_position) {
    int total_head_movements = 0;
    bool visited[n];

    // Initialize visited array
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    // Current position of the head
    int current_position = start_position;

    printf("Order of service: %d -> ", current_position);

    for (int i = 0; i < n; i++) {
        int min_distance = __INT_MAX__;
        int next_request = -1;
```

```
        // Find the request with the shortest seek time
        for (int j = 0; j < n; j++) {
            if (!visited[j]) {
                int distance = absDiff(request_string[j], current_position);
                if (distance < min_distance) {
                    min_distance = distance;
                    next_request = j;
                }
            }
        }

        // Mark the next request as visited
        visited[next_request] = true;

        // Move the head to the next request
        current_position = request_string[next_request];

        // Update total head movements
        total_head_movements += min_distance;

        // Print the next request
        printf("%d -> ", current_position);
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {186, 89, 44, 70, 102, 22, 51, 124};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 70;

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);

    printf("\nOrder of service:\n");
    sstf(request_string, n, start_position);

    return 0;
}
```

slip 9

Q.1. Consider the following snapshot of system, A, B, C, D is the resource type.

Proces

s

Allocation Max Available
A B C D A B C D A B C D
P0 0 0 1 2 0 0 1 2 1 5 2 0
P1 1 0 0 0 1 7 5 0
P2 1 3 5 4 2 3 5 6
P3 0 6 3 2 0 6 5 2
P4 0 0 1 4 0 6 5 6
Using Resource –Request algorithm to Check whether the current system is in safe state
or not

ans:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

int allocation[MAX_PROCESSES][MAX_RESOURCES] = {
    {0, 0, 1, 2},
    {1, 0, 0, 0},
    {1, 3, 5, 4},
    {0, 6, 3, 2},
    {0, 0, 1, 4}
};

int max[MAX_PROCESSES][MAX_RESOURCES] = {
    {0, 0, 1, 2},
    {1, 7, 5, 0},
    {2, 3, 5, 6},
    {0, 6, 5, 2},
    {0, 6, 5, 6}
};

int available[MAX_RESOURCES] = {1, 5, 2, 0};
bool finish[MAX_PROCESSES] = {false};

// Function to check if the system is in a safe state
bool isSafeState() {
    int work[MAX_RESOURCES];
    bool tempFinish[MAX_PROCESSES];

    // Initialize work array
    for (int i = 0; i < MAX_RESOURCES; i++) {
        work[i] = available[i];
    }

    // Initialize temporary finish array
    for (int i = 0; i < MAX_PROCESSES; i++) {
        tempFinish[i] = finish[i];
```

```
    }

    bool found;
    do {
        found = false;
        for (int i = 0; i < MAX_PROCESSES; i++) {
            if (!tempFinish[i]) {
                bool canAllocate = true;
                for (int j = 0; j < MAX_RESOURCES; j++) {
                    if (max[i][j] - allocation[i][j] > work[j]) {
                        canAllocate = false;
                        break;
                    }
                }
                if (canAllocate) {
                    // Simulate allocation
                    for (int j = 0; j < MAX_RESOURCES; j++) {
                        work[j] += allocation[i][j];
                    }
                    tempFinish[i] = true;
                    found = true;
                }
            }
        }
    } while (found);

    // Check if all processes are finished
    for (int i = 0; i < MAX_PROCESSES; i++) {
        if (!tempFinish[i]) {
            return false;
        }
    }

    return true;
}

int main() {
    if (isSafeState()) {
        printf("The current system is in a safe state.\n");
    } else {
        printf("The current system is not in a safe state.\n");
    }

    return 0;
}
```

Q.2 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. [15]
176, 79, 34, 60, 92, 11, 41, 114
Starting Head Position: 65
Direction: Left

ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort the disk request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (request_string[j] > request_string[j+1]) {
                int temp = request_string[j];
                request_string[j] = request_string[j+1];
                request_string[j+1] = temp;
            }
        }
    }
}

// Function to simulate disk scheduling using LOOK algorithm
void look(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;

    // Sort the request string
    sort(request_string, n);

    // Move the head to the start position
    int current_position = start_position;

    printf("Order of service: %d -> ", current_position);

    if (direction == 'L') {
        // Scan to the left
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    } else if (direction == 'R') {
        // Scan to the right
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position) {
```

```
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {176, 79, 34, 60, 92, 11, 41, 114};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 65;
    char direction = 'L'; // Left direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    look(request_string, n, start_position, direction);

    return 0;
}
```

Slip 10

Q.1 Write an MPI program to calculate sum and average of randomly generated 1000 numbers (stored in array) on a cluster

ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int array[ARRAY_SIZE];
    int sum = 0;
    double average;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    // Generate random numbers
    srand(time(NULL) + rank); // Ensure different seeds for each process
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = rand() % 1000; // Generate random numbers between 0 and 999
    }

    // Calculate local sum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += array[i];
    }

    // Reduce to get global sum
    MPI_Reduce(&sum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        average = (double) sum / (ARRAY_SIZE * size);
        printf("Sum: %d\n", sum);
        printf("Average: %.2f\n", average);
    }

    MPI_Finalize();

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-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: Left

ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort the disk request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (request_string[j] > request_string[j+1]) {
                int temp = request_string[j];
                request_string[j] = request_string[j+1];
                request_string[j+1] = temp;
            }
        }
```

```
    }
}

// Function to simulate disk scheduling using C-SCAN algorithm
void cscan(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;
    int end_position = 199; // Total number of blocks - 1
    int current_position = start_position;

    // Sort the request string
    sort(request_string, n);

    printf("Order of service: ");

    if (direction == 'L') {
        // Scan to the left
        // Move the head to the end of the disk
        total_head_movements += abs(end_position - current_position);
        printf("%d -> ", end_position);
        current_position = end_position;

        // Move the head to the beginning of the disk
        total_head_movements += abs(current_position);
        printf("0 -> ");
        current_position = 0;

        // Scan from the beginning to the start position
        for (int i = 0; i < n; i++) {
            if (request_string[i] <= start_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    } else if (direction == 'R') {
        // Scan to the right
        // Move the head to the beginning of the disk
        total_head_movements += abs(current_position);
        printf("%d -> ", current_position);
        current_position = 0;

        // Move the head to the end of the disk
        total_head_movements += abs(end_position - current_position);
        printf("%d -> ", end_position);
        current_position = end_position;

        // Scan from the end to the start position
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] >= start_position) {
                total_head_movements += abs(request_string[i] - current_position);
```

```
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {33, 99, 142, 52, 197, 79, 46, 65};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 72;
    char direction = 'L'; // Left direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    cscan(request_string, n, start_position, direction);

    return 0;
}
```

Slip 11

Q.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			

Implement the following Menu.

a) Accept Available

b) Display Allocation, Max

c) Display the contents of need matrix

d) Display Available

Ans:

```
#include <stdio.h>
```

```
#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int allocation[MAX_PROCESSES][MAX_RESOURCES] = {
    {0, 1, 0},
    {2, 0, 0},
    {3, 0, 3},
    {2, 1, 1},
    {0, 0, 2}
};

int max[MAX_PROCESSES][MAX_RESOURCES] = {
    {0, 0, 0},
    {2, 0, 2},
    {0, 0, 0},
    {1, 0, 0},
    {0, 0, 2}
};

int available[MAX_RESOURCES] = {0, 0, 0};
int need[MAX_PROCESSES][MAX_RESOURCES];

// Function to calculate the need matrix
void calculateNeedMatrix() {
    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];
        }
    }
}

// Function to accept available resources
void acceptAvailable() {
    printf("Enter the available resources for A, B, and C: ");
    scanf("%d %d %d", &available[0], &available[1], &available[2]);
}

// Function to display allocation and maximum matrices
void displayAllocationMax() {
    printf("\nAllocation Matrix:\n");
    printf("   A  B  C\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%2d ", allocation[i][j]);
        }
        printf("\n");
    }
```

```
    printf("\nMaximum Matrix:\n");
    printf("   A  B  C\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%2d ", max[i][j]);
        }
        printf("\n");
    }
}

// Function to display the need matrix
void displayNeedMatrix() {
    printf("\nNeed Matrix:\n");
    printf("   A  B  C\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%2d ", need[i][j]);
        }
        printf("\n");
    }
}

// Function to display available resources
void displayAvailable() {
    printf("\nAvailable Resources:\n");
    printf("A: %d\n", available[0]);
    printf("B: %d\n", available[1]);
    printf("C: %d\n", available[2]);
}

int main() {
    calculateNeedMatrix();

    char choice;
    do {
        printf("\nMenu:\n");
        printf("a) Accept Available\n");
        printf("b) Display Allocation, Max\n");
        printf("c) Display the contents of need matrix\n");
        printf("d) Display Available\n");
        printf("e) Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);

        switch(choice) {
            case 'a':
                acceptAvailable();
                break;
```

```
            case 'b':
                displayAllocationMax();
                break;
            case 'c':
                displayNeedMatrix();
                break;
            case 'd':
                displayAvailable();
                break;
            case 'e':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while(choice != 'e');

    return 0;
}
```

Q.2 Write an MPI program to find the min number from randomly generated 1000 numbers
(stored in array) on a cluster (Hint: Use MPI_Reduce)

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int array[ARRAY_SIZE];
    int local_min, global_min;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Generate random numbers
    srand(time(NULL) + rank); // Ensure different seeds for each process
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = rand() % 1000; // Generate random numbers between 0 and 999
    }

    // Calculate local minimum
    local_min = array[0];
    for (int i = 1; i < ARRAY_SIZE; i++) {
```

```
        if (array[i] < local_min) {
            local_min = array[i];
        }
    }

    // Reduce to get global minimum
    MPI_Reduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Minimum number: %d\n", global_min);
    }

    MPI_Finalize();

    return 0;
}
```

Slip 12

Q.1 Write an MPI program to calculate sum and average randomly generated 1000 numbers (stored in array) on a cluster.

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int array[ARRAY_SIZE];
    int local_sum = 0;
    double global_sum, global_average;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Generate random numbers
    srand(time(NULL) + rank); // Ensure different seeds for each process
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = rand() % 1000; // Generate random numbers between 0 and 999
    }

    // Calculate local sum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        local_sum += array[i];
    }
```

```
    // Reduce to get global sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        global_average = global_sum / (ARRAY_SIZE * size);
        printf("Sum: %.2f\n", global_sum);
        printf("Average: %.2f\n", global_average);
    }

    MPI_Finalize();

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-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.
23, 89, 132, 42, 187, 69, 36, 55
Start Head Position: 40
Direction: Right

Ans:

```
#include <stdio.h>
#include <stdlib.h>

void c_look(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;

    // Sort the request string
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (request_string[j] > request_string[j + 1]) {
                int temp = request_string[j];
                request_string[j] = request_string[j + 1];
                request_string[j + 1] = temp;
            }
        }
    }

    // Determine the index of the current position in the sorted array
    int current_index = 0;
    for (int i = 0; i < n; i++) {
        if (request_string[i] >= start_position) {
            current_index = i;
            break;
        }
```

```
    }

    printf("Order of service: ");
    if (direction == 'L') {
        // Move the head to the leftmost request
        for (int i = current_index; i >= 0; i--) {
            total_head_movements += abs(start_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            start_position = request_string[i];
        }

        // Move the head to the rightmost request
        for (int i = n - 1; i > current_index; i--) {
            total_head_movements += abs(start_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            start_position = request_string[i];
        }
    } else if (direction == 'R') {
        // Move the head to the rightmost request
        for (int i = current_index; i < n; i++) {
            total_head_movements += abs(start_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            start_position = request_string[i];
        }

        // Move the head to the leftmost request
        for (int i = 0; i < current_index; i++) {
            total_head_movements += abs(start_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            start_position = request_string[i];
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {23, 89, 132, 42, 187, 69, 36, 55};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 40;
    char direction = 'R'; // Right direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
```

```
    c_look(request_string, n, start_position, direction);

    return 0;
}
```

Slip 13

Q.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			

a) Calculate and display the content of need matrix?

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

Ans:

```
#include <stdio.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int allocation[MAX_PROCESSES][MAX_RESOURCES] = {
    {0, 1, 0},
    {2, 0, 0},
    {3, 0, 3},
    {2, 1, 1},
    {0, 0, 2}
};

int max[MAX_PROCESSES][MAX_RESOURCES] = {
    {0, 0, 0},
    {2, 0, 2},
    {0, 0, 0},
    {1, 0, 0},
    {0, 0, 2}
};

int available[MAX_RESOURCES] = {0, 0, 0};
int need[MAX_PROCESSES][MAX_RESOURCES];

void calculateNeedMatrix() {
    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];
```

```
        }
    }
}

int checkSafeState(int processes[], int n) {
    int work[MAX_RESOURCES];
    int finish[MAX_PROCESSES];

    // Initialize work and finish arrays
    for (int i = 0; i < MAX_RESOURCES; i++) {
        work[i] = available[i];
    }
    for (int i = 0; i < MAX_PROCESSES; i++) {
        finish[i] = 0;
    }

    // Find an index 'i' such that all resources for 'i' can be allocated
    int count = 0;
    int safe_sequence[MAX_PROCESSES];
    while (count < n) {
        int found = 0;
        for (int i = 0; i < n; i++) {
            if (finish[i] == 0) {
                int j;
                for (j = 0; j < MAX_RESOURCES; j++) {
                    if (need[processes[i]][j] > work[j]) {
                        break;
                    }
                }
                if (j == MAX_RESOURCES) {
                    for (int k = 0; k < MAX_RESOURCES; k++) {
                        work[k] += allocation[processes[i]][k];
                    }
                    safe_sequence[count++] = processes[i];
                    finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (found == 0) {
            printf("System is not in safe state.\n");
            return 0;
        }
    }

    printf("System is in safe state.\nSafe sequence: ");
    for (int i = 0; i < n; i++) {
        printf("P%d ", safe_sequence[i]);
    }
    printf("\n");
```

```
    return 1;
}

int main() {
    calculateNeedMatrix();

    printf("Need Matrix:\n");
    printf("   A  B  C\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%2d ", need[i][j]);
        }
        printf("\n");
    }

    printf("\n");

    printf("Enter the available resources for A, B, and C: ");
    scanf("%d %d %d", &available[0], &available[1], &available[2]);

    int processes[MAX_PROCESSES] = {0, 1, 2, 3, 4};
    checkSafeState(processes, MAX_PROCESSES);

    return 0;
}
```

Q.2 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.
176, 79, 34, 60, 92, 11, 41, 114
Starting Head Position: 65
Direction: Left

Ans:

```
#include <stdio.h>
#include <stdlib.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void sort(int *request_string, int n) {
```

```
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (request_string[j] > request_string[j + 1]) {
                swap(&request_string[j], &request_string[j + 1]);
            }
        }
    }
}

void scan(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;

    // Sort the request string
    sort(request_string, n);

    int current_position = start_position;
    int i;

    printf("Order of service: ");

    // Move the head to the leftmost request
    if (direction == 'L') {
        for (i = 0; i < n; i++) {
            if (request_string[i] >= current_position) {
                break;
            }
        }

        for (; i >= 0; i--) {
            total_head_movements += abs(current_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            current_position = request_string[i];
        }

        // Move the head to the rightmost request
        for (i = n - 1; i >= 0; i--) {
            total_head_movements += abs(current_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            current_position = request_string[i];
        }
    }
    // Move the head to the rightmost request
    else if (direction == 'R') {
        for (i = 0; i < n; i++) {
            if (request_string[i] > current_position) {
                break;
            }
        }

        for (; i < n; i++) {
```

```
            total_head_movements += abs(current_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            current_position = request_string[i];
        }

        // Move the head to the leftmost request
        for (i = 0; i < n; i++) {
            total_head_movements += abs(current_position - request_string[i]);
            printf("%d -> ", request_string[i]);
            current_position = request_string[i];
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {176, 79, 34, 60, 92, 11, 41, 114};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 65;
    char direction = 'L'; // Left direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    scan(request_string, n, start_position, direction);

    return 0;
}
```

Slip 14

Q.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
- Show Directory
- Delete File
- Exit

ans:

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <time.h>

#define MAX_BLOCKS 100

int disk[MAX_BLOCKS];
int allocated_blocks = 0;

// Function to initialize the disk
void initializeDisk(int n) {
    for (int i = 0; i < n; i++) {
        disk[i] = 0; // 0 represents free block
    }
}

// Function to display the bit vector
void showBitVector(int n) {
    printf("Bit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

// Function to display the directory
void showDirectory(int n) {
    printf("Directory:\n");
    for (int i = 0; i < n; i++) {
        if (disk[i] == 1) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

// Function to randomly mark some blocks as allocated
void allocateBlocks(int n) {
    int num_allocated = rand() % n + 1; // Random number of allocated blocks (1 to
n)
    for (int i = 0; i < num_allocated; i++) {
        int block = rand() % n; // Random block index
        if (disk[block] == 0) {
            disk[block] = 1; // Mark the block as allocated
            allocated_blocks++;
        }
    }
    printf("Allocated %d blocks randomly.\n", num_allocated);
}
```

```
// Function to delete a file (blocks are marked as free)
void deleteFile(int n) {
    if (allocated_blocks == 0) {
        printf("No files to delete.\n");
        return;
    }

    int blocks_to_delete = rand() % allocated_blocks + 1; // Random number of
blocks to delete (1 to allocated_blocks)
    int deleted_count = 0;
    for (int i = 0; i < n && deleted_count < blocks_to_delete; i++) {
        if (disk[i] == 1) {
            disk[i] = 0; // Mark the block as free
            deleted_count++;
            allocated_blocks--;
        }
    }
    printf("Deleted %d blocks.\n", deleted_count);
}

int main() {
    srand(time(NULL));

    int n;
    printf("Enter the number of blocks on the disk: ");
    scanf("%d", &n);

    initializeDisk(n);

    char choice;
    do {
        printf("\nMenu:\n");
        printf("a) Show Bit Vector\n");
        printf("b) Show Directory\n");
        printf("c) Delete File\n");
        printf("d) Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);

        switch(choice) {
            case 'a':
                showBitVector(n);
                break;
            case 'b':
                showDirectory(n);
                break;
            case 'c':
                deleteFile(n);
                break;
            case 'd':
```

```
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while(choice != 'd');

    return 0;
}
```

Q.2 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.
55, 58, 39, 18, 90, 160, 150, 38, 184
Start Head Position: 50

ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// Function to swap two integers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to sort the request string based on distance from the current position
void sort(int *request_string, int n, int current_position) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (abs(request_string[j] - current_position) > abs(request_string[j + 1] - current_position)) {
                swap(&request_string[j], &request_string[j + 1]);
            }
        }
    }
}

// Function to calculate the total head movements and display the order of service
void sstf(int *request_string, int n, int start_position) {
    int total_head_movements = 0;
    int current_position = start_position;
```

```
    printf("Order of service: ");

    for (int i = 0; i < n; i++) {
        total_head_movements += abs(request_string[i] - current_position);
        printf("%d -> ", request_string[i]);
        current_position = request_string[i];
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {55, 58, 39, 18, 90, 160, 150, 38, 184};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 50;

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);

    sort(request_string, n, start_position);

    printf("\nOrder of service:\n");
    sstf(request_string, n, start_position);

    return 0;
}
```

Slip 15

Q.1 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

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100
```

```
typedef struct Node {
    int block_number;
    struct Node *next;
} Node;

Node *disk[MAX_BLOCKS] = {NULL};
bool allocated_blocks[MAX_BLOCKS] = {false};

// Function to display the bit vector
void showBitVector(int n) {
    printf("Bit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", allocated_blocks[i]);
    }
    printf("\n");
}

// Function to display the directory
void showDirectory(int n) {
    printf("Directory:\n");
    for (int i = 0; i < n; i++) {
        if (allocated_blocks[i]) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

// Function to create a new file
void createNewFile(int n) {
    int starting_block = rand() % n;
    while (allocated_blocks[starting_block]) {
        starting_block = (starting_block + 1) % n; // Find the next free block
    }

    Node *file = (Node *)malloc(sizeof(Node));
    file->block_number = starting_block;
    file->next = NULL;
    disk[starting_block] = file;
    allocated_blocks[starting_block] = true;

    printf("New file created starting from block %d.\n", starting_block);
}

int main() {
    srand(time(NULL));

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

```
    scanf("%d", &n);

    char choice;
    do {
        printf("\nMenu:\n");
        printf("a) Show Bit Vector\n");
        printf("b) Create New File\n");
        printf("c) Show Directory\n");
        printf("d) Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);

        switch (choice) {
            case 'a':
                showBitVector(n);
                break;
            case 'b':
                createNewFile(n);
                break;
            case 'c':
                showDirectory(n);
                break;
            case 'd':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 'd');

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-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..
80, 150, 60,135, 40, 35, 170
Starting Head Position: 70
Direction: Right

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to swap two integers
```

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to sort the request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (request_string[j] > request_string[j + 1]) {
                swap(&request_string[j], &request_string[j + 1]);
            }
        }
    }
}

// Function to implement C-SCAN disk scheduling algorithm
void cscan(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;

    // Sort the request string
    sort(request_string, n);

    int current_position = start_position;

    // Move the head to the rightmost position
    if (direction == 'R') {
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        // Move the head to the beginning (cylinder 0)
        total_head_movements += abs(current_position - 0);
        printf("0 -> ");
        current_position = 0;
        // Move the head to the rightmost request again
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    }
    // Move the head to the leftmost position
    else if (direction == 'L') {
```

```
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        // Move the head to the end (last cylinder)
        total_head_movements += abs(current_position - 199); // Assuming the last
cylinder is 199
        printf("199 -> ");
        current_position = 199;
        // Move the head to the leftmost request again
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {80, 150, 60, 135, 40, 35, 170};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 70;
    char direction = 'R'; // Right direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    cscan(request_string, n, start_position, direction);

    return 0;
}
```

Slip 16

Q.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
- Create New File
- Show Directory
- Exit

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100

bool disk[MAX_BLOCKS];
int allocated_blocks = 0;

// Function to display the bit vector
void showBitVector(int n) {
    printf("Bit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

// Function to display the directory
void showDirectory(int n) {
    printf("Directory:\n");
    for (int i = 0; i < n; i++) {
        if (disk[i]) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

// Function to create a new file
void createNewFile(int n) {
    int start_block = rand() % n;
    int size = rand() % (n - start_block) + 1; // Random size of the file (1 to n -
start_block)

    if (allocated_blocks + size > n) {
        printf("Not enough free space to create the file.\n");
        return;
    }

    for (int i = start_block; i < start_block + size; i++) {
```

```
        if (!disk[i]) {
            disk[i] = true; // Mark the block as allocated
            allocated_blocks++;
        }
    }
    printf("New file created starting from block %d with size %d.\n", start_block,
size);
}

int main() {
    srand(time(NULL));

    int n;
    printf("Enter the number of blocks on the disk: ");
    scanf("%d", &n);

    char choice;
    do {
        printf("\nMenu:\n");
        printf("a) Show Bit Vector\n");
        printf("b) Create New File\n");
        printf("c) Show Directory\n");
        printf("d) Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);

        switch (choice) {
            case 'a':
                showBitVector(n);
                break;
            case 'b':
                createNewFile(n);
                break;
            case 'c':
                showDirectory(n);
                break;
            case 'd':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 'd');

    return 0;
}
```

Q.2 Write an MPI program to find the min number from randomly generated 1000 numbers

(stored in array) on a cluster (Hint: Use MPI_Reduce)

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int min_local, min_global;
    int numbers[ARRAY_SIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed for random number generation
    srand(time(NULL) + rank);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        numbers[i] = rand() % 1000; // Generating numbers between 0 to 999
    }

    // Find local minimum
    min_local = numbers[0];
    for (int i = 1; i < ARRAY_SIZE; i++) {
        if (numbers[i] < min_local) {
            min_local = numbers[i];
        }
    }

    // Reduce the local minimum to find the global minimum
    MPI_Reduce(&min_local, &min_global, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    // Print result
    if (rank == 0) {
        printf("Minimum number: %d\n", min_global);
    }

    MPI_Finalize();
    return 0;
}
```

imp commands for run this code:

1)mpicc mpi_min.c -o mpi_min -Wall

2)mpirun -np <num_processes> ./mpi_min --for bash

Slip 17
Q.1 Write a program to simulate Indexed 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 above and implement each option.

- Show Bit Vector
- Show Directory
- Delete Already File
- Exit

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100

bool disk[MAX_BLOCKS];
bool index_table[MAX_BLOCKS]; // Index table to track allocated blocks
int allocated_blocks = 0;

// Function to display the bit vector
void showBitVector(int n) {
    printf("Bit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

// Function to display the directory
void showDirectory(int n) {
    printf("Directory:\n");
    for (int i = 0; i < n; i++) {
        if (index_table[i]) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

// Function to create a new file
void createNewFile(int n) {
    int index = rand() % n;
```

```
    if (!disk[index]) {
        disk[index] = true; // Mark the block as allocated
        index_table[allocated_blocks] = true; // Update the index table
        allocated_blocks++;
        printf("New file created at block %d.\n", index);
    } else {
        printf("Block %d is already allocated.\n", index);
    }
}

// Function to delete an existing file
void deleteFile(int n) {
    if (allocated_blocks == 0) {
        printf("No files to delete.\n");
        return;
    }

    int index = rand() % allocated_blocks;
    for (int i = 0, j = 0; i < n; i++) {
        if (index_table[i]) {
            if (j == index) {
                disk[i] = false; // Mark the block as free
                index_table[i] = false; // Update the index table
                allocated_blocks--;
                printf("File deleted from block %d.\n", i);
                return;
            }
            j++;
        }
    }
}

int main() {
    srand(time(NULL));

    int n;
    printf("Enter the number of blocks on the disk: ");
    scanf("%d", &n);

    char choice;
    do {
        printf("\nMenu:\n");
        printf("a) Show Bit Vector\n");
        printf("b) Show Directory\n");
        printf("c) Create New File\n");
        printf("d) Delete File\n");
        printf("e) Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);
```

```
        switch (choice) {
            case 'a':
                showBitVector(n);
                break;
            case 'b':
                showDirectory(n);
                break;
            case 'c':
                createNewFile(n);
                break;
            case 'd':
                deleteFile(n);
                break;
            case 'e':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 'e');

    return 0;
}
```

Q.2 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.
23, 89, 132, 42, 187, 69, 36, 55
Start Head Position: 40
Direction: Left

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to swap two integers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to sort the request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n - 1; i++) {
```

```
        for (int j = 0; j < n - i - 1; j++) {
            if (request_string[j] > request_string[j + 1]) {
                swap(&request_string[j], &request_string[j + 1]);
            }
        }
    }
}

// Function to implement LOOK disk scheduling algorithm
void look(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;

    // Sort the request string
    sort(request_string, n);

    int current_position = start_position;
    int prev_position = start_position;

    // Move the head in the specified direction
    if (direction == 'L') { // Left direction
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        // Move the head to the beginning (cylinder 0)
        total_head_movements += abs(current_position - 0);
        printf("0 -> ");
        current_position = 0;
        // Move the head to the rightmost request
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position && request_string[i] <
prev_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    } else if (direction == 'R') { // Right direction
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        // Move the head to the end (last cylinder)
        total_head_movements += abs(current_position - 199); // Assuming the last
```

```
cylinder is 199
        printf("199 -> ");
        current_position = 199;
        // Move the head to the leftmost request
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position && request_string[i] >
prev_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {23, 89, 132, 42, 187, 69, 36, 55};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 40;
    char direction = 'L'; // Left direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    look(request_string, n, start_position, direction);

    return 0;
}
```

Slip 18

Q.1 Write a program to simulate Indexed 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 above and implement each option.

- Show Bit Vector
- Create New File
- Show Directory
- Delete File
- Exit

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100

bool disk[MAX_BLOCKS];
bool index_table[MAX_BLOCKS]; // Index table to track allocated blocks
int allocated_blocks = 0;

// Function to display the bit vector
void showBitVector(int n) {
    printf("Bit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

// Function to display the directory
void showDirectory(int n) {
    printf("Directory:\n");
    for (int i = 0; i < n; i++) {
        if (index_table[i]) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

// Function to create a new file
void createNewFile(int n) {
    int index = rand() % n;

    if (!disk[index]) {
        disk[index] = true; // Mark the block as allocated
        index_table[allocated_blocks] = true; // Update the index table
        allocated_blocks++;
        printf("New file created at block %d.\n", index);
    } else {
        printf("Block %d is already allocated.\n", index);
    }
}

// Function to delete an existing file
void deleteFile(int n) {
```

```
    if (allocated_blocks == 0) {
        printf("No files to delete.\n");
        return;
    }

    int index = rand() % allocated_blocks;
    for (int i = 0, j = 0; i < n; i++) {
        if (index_table[i]) {
            if (j == index) {
                disk[i] = false; // Mark the block as free
                index_table[i] = false; // Update the index table
                allocated_blocks--;
                printf("File deleted from block %d.\n", i);
                return;
            }
            j++;
        }
    }
}

int main() {
    srand(time(NULL));

    int n;
    printf("Enter the number of blocks on the disk: ");
    scanf("%d", &n);

    char choice;
    do {
        printf("\nMenu:\n");
        printf("a) Show Bit Vector\n");
        printf("b) Create New File\n");
        printf("c) Show Directory\n");
        printf("d) Delete File\n");
        printf("e) Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);

        switch (choice) {
            case 'a':
                showBitVector(n);
                break;
            case 'b':
                createNewFile(n);
                break;
            case 'c':
                showDirectory(n);
                break;
            case 'd':
                deleteFile(n);
```

```
                break;
            case 'e':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 'e');

    return 0;
}
```

Q.2 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: Right

ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to swap two integers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to sort the request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (request_string[j] > request_string[j + 1]) {
                swap(&request_string[j], &request_string[j + 1]);
            }
        }
    }
}

// Function to implement SCAN disk scheduling algorithm
void scan(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;
```

```
    // Sort the request string
    sort(request_string, n);

    int current_position = start_position;

    if (direction == 'L') { // Left direction
        // Move the head to the leftmost request
        total_head_movements += current_position;
        printf("%d -> ", current_position);
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        // Move the head to the rightmost request
        total_head_movements += (199 - current_position); // Assuming the last
cylinder is 199
        printf("199 -> ");
        current_position = 199;
        for (int i = 0; i < n; i++) {
            if (request_string[i] > current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    } else if (direction == 'R') { // Right direction
        // Move the head to the rightmost request
        total_head_movements += (199 - current_position); // Assuming the last
cylinder is 199
        printf("%d -> ", current_position);
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        // Move the head to the leftmost request
        total_head_movements += current_position;
        printf("0 -> ");
        current_position = 0;
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] < current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
```

```
            }
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {33, 99, 142, 52, 197, 79, 46, 65};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 72;
    char direction = 'R'; // Right direction

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Direction: %c\n", direction);

    printf("\nOrder of service:\n");
    scan(request_string, n, start_position, direction);

    return 0;
}
```

Slip 19

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

Proces s	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	3	2	4	6	5	4	4	3	4	4	2
P1	1	2	0	1	4	4	4	4				
P2	0	0	0	0	0	0	1	2				
P3	3	3	2	2	3	9	3	4				
P4	1	4	3	2	2	5	3	3				
P5	2	4	1	4	4	6	3	4				

a) Calculate and display the content of need matrix?

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

Ans:

```
#include <stdio.h>
#include <stdbool.h>
```

```
#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

// Function to calculate the need matrix
void calculateNeed(int need[MAX_PROCESSES][MAX_RESOURCES], int
max[MAX_PROCESSES][MAX_RESOURCES], int allocation[MAX_PROCESSES][MAX_RESOURCES],
int available[MAX_RESOURCES]) {
    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];
        }
    }
}

// Function to check if the system is in a safe state
bool isSafeState(int processes[MAX_PROCESSES], int available[MAX_RESOURCES], int
max[MAX_PROCESSES][MAX_RESOURCES], int allocation[MAX_PROCESSES][MAX_RESOURCES]) {
    int work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES] = {false};

    // Initialize work and finish arrays
    for (int i = 0; i < MAX_RESOURCES; i++) {
        work[i] = available[i];
    }

    // Find an index i such that both:
    // 1. Finish[i] == false
    // 2. Need[i] <= Work
    // If no such i exists, then the system is not in a safe state
    int count = 0;
    while (count < MAX_PROCESSES) {
        bool found = false;
        for (int i = 0; i < MAX_PROCESSES; i++) {
            if (!finish[i]) {
                int j;
                for (j = 0; j < MAX_RESOURCES; j++) {
                    if (max[i][j] - allocation[i][j] > work[j]) {
                        break;
                    }
                }
                if (j == MAX_RESOURCES) {
                    // Resource allocation can be satisfied
                    for (int k = 0; k < MAX_RESOURCES; k++) {
                        work[k] += allocation[i][k];
                    }
                    finish[i] = true;
                    processes[count++] = i;
                    found = true;
                }
            }
```

```
        }
        // If no such process was found, break the loop
        if (!found) {
            break;
        }
    }
    // If all processes are finished, the system is in a safe state
    return count == MAX_PROCESSES;
}

int main() {
    int allocation[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 3, 2, 4},
        {1, 2, 0, 1},
        {0, 0, 0, 0},
        {3, 3, 2, 2},
        {1, 4, 3, 2},
        {2, 4, 1, 4}
    };

    int max[MAX_PROCESSES][MAX_RESOURCES] = {
        {6, 5, 4, 4},
        {4, 4, 4, 4},
        {0, 0, 1, 2},
        {3, 9, 3, 4},
        {2, 5, 3, 3},
        {4, 6, 3, 4}
    };

    int available[MAX_RESOURCES] = {3, 4, 2, 1};
    int need[MAX_PROCESSES][MAX_RESOURCES];

    // Calculate the need matrix
    calculateNeed(need, max, allocation, available);

    // Display the need matrix
    printf("Need Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    // Check if the system is in a safe state
    int safeSequence[MAX_PROCESSES];
    if (isSafeState(safeSequence, available, max, allocation)) {
        printf("\nThe system is in a safe state.\nSafe Sequence: ");
        for (int i = 0; i < MAX_PROCESSES; i++) {
```

```
            printf("P%d ", safeSequence[i]);
        }
        printf("\n");
    } else {
        printf("\nThe system is not in a safe state.\n");
    }

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-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.
23, 89, 132, 42, 187, 69, 36, 55
Start Head Position: 40
Direction: Left

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to swap two integers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to sort the request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (request_string[j] > request_string[j + 1]) {
                swap(&request_string[j], &request_string[j + 1]);
            }
        }
    }
}

// Function to implement C-SCAN disk scheduling algorithm
void cScan(int *request_string, int n, int start_position) {
    int total_head_movements = 0;

    // Sort the request string
    sort(request_string, n);
```

```
    // Move the head to the leftmost request
    total_head_movements += start_position;
    printf("%d -> ", start_position);
    for (int i = 0; i < n; i++) {
        if (request_string[i] >= start_position) {
            total_head_movements += abs(request_string[i] - start_position);
            printf("%d -> ", request_string[i]);
            start_position = request_string[i];
        }
    }
    // Move the head to the beginning of the disk
    total_head_movements += (199 - start_position); // Assuming the last cylinder
is 199
    printf("199 -> 0 -> ");

    // Move the head to the rightmost request
    total_head_movements += 199; // Assuming the last cylinder is 199
    printf("199 -> ");
    for (int i = 0; i < n; i++) {
        if (request_string[i] < start_position) {
            total_head_movements += abs(request_string[i] - start_position);
            printf("%d -> ", request_string[i]);
            start_position = request_string[i];
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {23, 89, 132, 42, 187, 69, 36, 55};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 40;

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);

    printf("\nOrder of service:\n");
    cScan(request_string, n, start_position);

    return 0;
}
```

Slip 20

Q.1 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

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to swap two integers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to sort the request string in ascending order
void sort(int *request_string, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (request_string[j] > request_string[j + 1]) {
                swap(&request_string[j], &request_string[j + 1]);
            }
        }
    }
}

// Function to implement SCAN disk scheduling algorithm
void scan(int *request_string, int n, int start_position, char direction) {
    int total_head_movements = 0;

    // Sort the request string
    sort(request_string, n);

    int current_position = start_position;

    if (direction == 'L') { // Left direction
        // Move the head to the leftmost request
        total_head_movements += current_position;
        printf("%d -> ", current_position);
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] <= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
```

```
            }
        }
        // Move the head to the rightmost request
        total_head_movements += (199 - current_position); // Assuming the last
cylinder is 199
        printf("199 -> ");
        current_position = 199;
        for (int i = 0; i < n; i++) {
            if (request_string[i] > current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    } else if (direction == 'R') { // Right direction
        // Move the head to the rightmost request
        total_head_movements += (199 - current_position); // Assuming the last
cylinder is 199
        printf("%d -> ", current_position);
        for (int i = 0; i < n; i++) {
            if (request_string[i] >= current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
        // Move the head to the leftmost request
        total_head_movements += current_position;
        printf("0 -> ");
        current_position = 0;
        for (int i = n - 1; i >= 0; i--) {
            if (request_string[i] < current_position) {
                total_head_movements += abs(request_string[i] - current_position);
                printf("%d -> ", request_string[i]);
                current_position = request_string[i];
            }
        }
    }

    printf("\nTotal number of head movements: %d\n", total_head_movements);
}

int main() {
    int request_string[] = {33, 99, 142, 52, 197, 79, 46, 65};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 72;
    char direction;

    printf("Disk Request String: ");
    for (int i = 0; i < n; i++) {
```

```
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);
    printf("Enter direction (L/R): ");
    scanf(" %c", &direction);

    printf("\nOrder of service:\n");
    scan(request_string, n, start_position, direction);

    return 0;
}
```

Q.2 Write an MPI program to find the max number from randomly generated 1000 numbers
(stored in array) on a cluster (Hint: Use MPI_Reduce)

ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char **argv) {
    int rank, size;
    int max_local = 0;
    int max_global = 0;
    int numbers[ARRAY_SIZE];

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed random number generator differently on each process
    srand(rank + 1);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        numbers[i] = rand();
    }

    // Find local maximum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        if (numbers[i] > max_local) {
            max_local = numbers[i];
        }
    }
```

```
    // Reduce all local maximums to global maximum
    MPI_Reduce(&max_local, &max_global, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print the result from process 0
    if (rank == 0) {
        printf("The maximum number is: %d\n", max_global);
    }

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

Slip 21

Q.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

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to calculate total head movements
int calculateHeadMovements(int *request_string, int n, int start_position) {
    int total_head_movements = 0;

    // First request served first, so just calculate the absolute difference
    for (int i = 0; i < n; ++i) {
        total_head_movements += abs(request_string[i] - start_position);
        start_position = request_string[i];
    }

    return total_head_movements;
}

int main() {
    int request_string[] = {55, 58, 39, 18, 90, 160, 150, 38, 184};
    int n = sizeof(request_string) / sizeof(request_string[0]);
    int start_position = 50;

    printf("Disk Request String: ");
```

```
    for (int i = 0; i < n; i++) {
        printf("%d ", request_string[i]);
    }
    printf("\nStarting Head Position: %d\n", start_position);

    int total_head_movements = calculateHeadMovements(request_string, n,
start_position);

    printf("Order of service:\n");
    for (int i = 0; i < n; i++) {
        printf("%d -> ", request_string[i]);
    }
    printf("\nTotal number of head movements: %d\n", total_head_movements);

    return 0;
}
```

Q.2 Write an MPI program to calculate sum of all even randomly generated 1000 numbers (stored in array) on a cluster

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char **argv) {
    int rank, size;
    int sum_local = 0;
    int sum_global = 0;
    int numbers[ARRAY_SIZE];

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed random number generator differently on each process
    srand(time(NULL) + rank);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        numbers[i] = rand();
    }

    // Calculate local sum of even numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
```

```
        if (numbers[i] % 2 == 0) {
            sum_local += numbers[i];
        }
    }

    // Reduce all local sums to global sum
    MPI_Reduce(&sum_local, &sum_global, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print the result from process 0
    if (rank == 0) {
        printf("The sum of all even numbers is: %d\n", sum_global);
    }

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

Slip 22

Q.1 Write an MPI program to calculate sum of all odd randomly generated 1000 numbers
(stored in array) on a cluster.

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char **argv) {
    int rank, size;
    int sum_local = 0;
    int sum_global = 0;
    int numbers[ARRAY_SIZE];

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed random number generator differently on each process
    srand(time(NULL) + rank);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
```

```
        numbers[i] = rand();
    }

    // Calculate local sum of odd numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        if (numbers[i] % 2 != 0) {
            sum_local += numbers[i];
        }
    }

    // Reduce all local sums to global sum
    MPI_Reduce(&sum_local, &sum_global, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print the result from process 0
    if (rank == 0) {
        printf("The sum of all odd numbers is: %d\n", sum_global);
    }

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

Q.2 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

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_BLOCKS 1000

// Function to initialize the bit vector
void initializeBitVector(int *bit_vector, int n) {
    for (int i = 0; i < n; i++) {
        bit_vector[i] = 0; // 0 represents free block
    }
}

// Function to display the bit vector
void displayBitVector(int *bit_vector, int n) {
```

```
    printf("Bit Vector: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", bit_vector[i]);
    }
    printf("\n");
}

// Function to create a new file
void createNewFile(int *bit_vector, int n) {
    int file_start, file_size;

    printf("Enter starting block number for the file: ");
    scanf("%d", &file_start);
    printf("Enter size of the file: ");
    scanf("%d", &file_size);

    if (file_start < 0 || file_start >= n || file_start + file_size > n) {
        printf("Invalid file allocation. Please try again.\n");
        return;
    }

    for (int i = file_start; i < file_start + file_size; i++) {
        if (bit_vector[i] == 1) {
            printf("Error: Block %d is already allocated.\n", i);
            return;
        }
    }

    for (int i = file_start; i < file_start + file_size; i++) {
        bit_vector[i] = 1; // 1 represents allocated block
    }

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

// Function to delete an existing file
void deleteFile(int *bit_vector, int n) {
    int file_start, file_size;

    printf("Enter starting block number of the file to delete: ");
    scanf("%d", &file_start);
    printf("Enter size of the file to delete: ");
    scanf("%d", &file_size);

    if (file_start < 0 || file_start >= n || file_start + file_size > n) {
        printf("Invalid file deletion. Please try again.\n");
        return;
    }

    for (int i = file_start; i < file_start + file_size; i++) {
```

```
        if (bit_vector[i] == 0) {
            printf("Error: Block %d is already free.\n", i);
            return;
        }
    }

    for (int i = file_start; i < file_start + file_size; i++) {
        bit_vector[i] = 0; // 0 represents free block
    }

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

int main() {
    int n; // Number of disk blocks
    int bit_vector[MAX_BLOCKS]; // Bit vector to represent disk blocks

    printf("Enter the number of disk blocks: ");
    scanf("%d", &n);

    initializeBitVector(bit_vector, n);

    int choice;
    do {
        printf("\nMenu:\n");
        printf("1. Show Bit Vector\n");
        printf("2. Create New File\n");
        printf("3. Delete Already Created File\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                displayBitVector(bit_vector, n);
                break;
            case 2:
                createNewFile(bit_vector, n);
                break;
            case 3:
                deleteFile(bit_vector, n);
                break;
            case 4:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 4);
```

```
    return 0;
}
```

Slip 23

Q.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

Ans:

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

// Function to calculate the need matrix
void calculateNeedMatrix(int need[MAX_PROCESSES][MAX_RESOURCES], int
allocation[MAX_PROCESSES][MAX_RESOURCES],
                         int max[MAX_PROCESSES][MAX_RESOURCES], int m, int n) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

// Function to check if the request can be granted immediately
int checkRequest(int request[MAX_RESOURCES], int available[MAX_RESOURCES], int
need[MAX_PROCESSES][MAX_RESOURCES],
                 int pid, int m) {
    for (int i = 0; i < m; i++) {
        if (request[i] > need[pid][i] || request[i] > available[i]) {
            return 0;
        }
    }
    return 1;
}

int main() {
    int m, n; // Number of processes and resource types
    int allocation[MAX_PROCESSES][MAX_RESOURCES]; // Allocation matrix
    int max[MAX_PROCESSES][MAX_RESOURCES]; // Maximum requirement matrix
    int available[MAX_RESOURCES]; // Available instances of each resource type
    int need[MAX_PROCESSES][MAX_RESOURCES]; // Need matrix

    // Accept input for number of processes and resource types
    printf("Enter number of processes: ");
```

```
scanf("%d", &m);
printf("Enter number of resource types: ");
scanf("%d", &n);

// Accept allocation matrix
printf("Enter allocation matrix:\n");
for (int i = 0; i < m; i++) {
    printf("Process %d: ", i);
    for (int j = 0; j < n; j++) {
        scanf("%d", &allocation[i][j]);
    }
}

// Accept maximum requirement matrix
printf("Enter maximum requirement matrix:\n");
for (int i = 0; i < m; i++) {
    printf("Process %d: ", i);
    for (int j = 0; j < n; j++) {
        scanf("%d", &max[i][j]);
    }
}

// Accept available instances of each resource type
printf("Enter available instances of each resource type: ");
for (int i = 0; i < n; i++) {
    scanf("%d", &available[i]);
}

// Calculate need matrix
calculateNeedMatrix(need, allocation, max, m, n);

// Display need matrix
printf("Need Matrix:\n");
for (int i = 0; i < m; i++) {
    printf("Process %d: ", i);
    for (int j = 0; j < n; j++) {
        printf("%d ", need[i][j]);
    }
    printf("\n");
}

// Check if a given request can be granted immediately
int pid;
printf("Enter process ID for request: ");
scanf("%d", &pid);

int request[MAX_RESOURCES];
printf("Enter request for process %d: ", pid);
for (int i = 0; i < n; i++) {
    scanf("%d", &request[i]);
```

```
    }

    if (checkRequest(request, available, need, pid, n)) {
        printf("Request can be granted immediately.\n");
    } else {
        printf("Request cannot be granted immediately.\n");
    }

    return 0;
}
```

Q.2 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

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// Function to find the index of the nearest track
int findNearestTrack(int tracks[], int n, int head) {
    int minDist = INT_MAX;
    int index = -1;

    for (int i = 0; i < n; i++) {
        if (abs(tracks[i] - head) < minDist) {
            minDist = abs(tracks[i] - head);
            index = i;
        }
    }

    return index;
}

int main() {
    int n; // Number of disk blocks
    int head; // Current head position
    int totalHeadMovements = 0;

    // Accept total number of disk blocks and head position
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &n);
```

```
    printf("Enter the current head position: ");
    scanf("%d", &head);

    // Accept disk request string
    int *tracks = (int *)malloc(n * sizeof(int));
    printf("Enter the disk request string:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &tracks[i]);
    }

    // Initialize an array to keep track of processed tracks
    int *processed = (int *)calloc(n, sizeof(int));

    printf("Serving requests in SSTF order:\n");

    // SSTF algorithm
    for (int i = 0; i < n; i++) {
        int nearestIndex = findNearestTrack(tracks, n, head);
        if (nearestIndex == -1) {
            break; // No more requests to serve
        }

        printf("%d ", tracks[nearestIndex]);
        processed[nearestIndex] = 1;
        totalHeadMovements += abs(tracks[nearestIndex] - head);
        head = tracks[nearestIndex];
        tracks[nearestIndex] = INT_MAX; // Mark the processed track as visited
    }

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

    // Free dynamically allocated memory
    free(tracks);
    free(processed);

    return 0;
}
```

Slip 24

Q.1 Write an MPI program to calculate sum of all odd randomly generated 1000 numbers
(stored in array) on a cluster.

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
```

```c
#define ARRAY_SIZE 1000

int main(int argc, char **argv) {
    int rank, size;
    int local_sum = 0;
    int total_sum = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Seed for random number generation
    srand(time(NULL) + rank);

    // Generate and sum odd numbers in local process
    for (int i = 0; i < ARRAY_SIZE; i++) {
        int num = rand() % 1000;
        if (num % 2 != 0) {
            local_sum += num;
        }
    }

    // Sum local sums across all processes
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Output total sum from rank 0
    if (rank == 0) {
        printf("Total sum of odd numbers: %d\n", total_sum);
    }

    MPI_Finalize();

    return 0;
}
```

Q.2 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			

a) Calculate and display the content of need matrix?
b) Is the system in safe state? If display the safe sequence.

Ans:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

// Function to calculate the need matrix
void calculateNeedMatrix(int need[MAX_PROCESSES][MAX_RESOURCES], int
allocation[MAX_PROCESSES][MAX_RESOURCES],
                         int max[MAX_PROCESSES][MAX_RESOURCES], int m, int n) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

// Function to check if the system is in a safe state
bool isSafe(int processes[MAX_PROCESSES], int available[MAX_RESOURCES], int
allocation[MAX_PROCESSES][MAX_RESOURCES],
            int need[MAX_PROCESSES][MAX_RESOURCES], int m, int n) {
    bool finish[MAX_PROCESSES] = { false };
    int work[MAX_RESOURCES];

    // Initialize work as available
    for (int i = 0; i < n; i++) {
        work[i] = available[i];
    }

    // Find an index i such that both
    // a) finish[i] == false
    // b) need[i] <= work
    int count = 0;
    while (count < m) {
        bool found = false;
        for (int i = 0; i < m; i++) {
            if (!finish[i]) {
                bool canExecute = true;
                for (int j = 0; j < n; j++) {
                    if (need[i][j] > work[j]) {
                        canExecute = false;
                        break;
                    }
                }
                if (canExecute) {
                    // Execute process i
                    for (int j = 0; j < n; j++) {
                        work[j] += allocation[i][j];
                    }
```

```
                    finish[i] = true;
                    processes[count++] = i;
                    found = true;
                }
            }
        }
        if (!found) {
            // No such process found, system is unsafe
            return false;
        }
    }
    // All processes executed successfully, system is safe
    return true;
}

int main() {
    int allocation[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 3},
        {2, 1, 1},
        {0, 0, 2}
    };
    int max[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 0, 0},
        {2, 0, 2},
        {3, 0, 3},
        {2, 1, 1},
        {0, 0, 2}
    };
    int available[MAX_RESOURCES] = {0, 0, 0};
    int need[MAX_PROCESSES][MAX_RESOURCES];
    int processes[MAX_PROCESSES];

    // Accept available instances of each resource type
    printf("Enter available instances of each resource type: ");
    for (int i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &available[i]);
    }

    // Calculate need matrix
    calculateNeedMatrix(need, allocation, max, MAX_PROCESSES, MAX_RESOURCES);

    // Display need matrix
    printf("Need Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", need[i][j]);
        }
```

```
        printf("\n");
    }

    // Check if the system is in a safe state
    if (isSafe(processes, available, allocation, need, MAX_PROCESSES,
MAX_RESOURCES)) {
        printf("System is in a safe state.\nSafe sequence: ");
        for (int i = 0; i < MAX_PROCESSES; i++) {
            printf("P%d ", processes[i]);
        }
        printf("\n");
    } else {
        printf("System is in an unsafe state.\n");
    }

    return 0;
}
```

Slip 25

Q.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

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort an array in ascending order
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 closest track in the given direction
```

```
int findClosestTrack(int tracks[], int n, int head, int direction) {
    if (direction == 1) { // Move right
        for (int i = 0; i < n; i++) {
            if (tracks[i] >= head) {
                return i;
            }
        }
    } else { // Move left
        for (int i = n - 1; i >= 0; i--) {
            if (tracks[i] <= head) {
                return i;
            }
        }
    }
    return -1;
}

int main() {
    int n; // Number of disk blocks
    int head; // Current head position
    int totalHeadMovements = 0;
    int direction; // Direction of movement: 0 for left, 1 for right

    // Accept total number of disk blocks, disk request string, starting head
position, and direction from the user
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &n);
    int *tracks = (int *)malloc(n * sizeof(int));
    printf("Enter the disk request string:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &tracks[i]);
    }
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    printf("Enter the direction of movement (0 for left, 1 for right): ");
    scanf("%d", &direction);

    // Sort the disk request string
    sort(tracks, n);

    // Find the index of the closest track in the given direction
    int closestIndex = findClosestTrack(tracks, n, head, direction);
    if (closestIndex == -1) {
        printf("No requests to serve.\n");
        return 0;
    }

    printf("Serving requests in LOOK order:\n");

    // Serve requests in LOOK order
```

```
    while (closestIndex >= 0 && closestIndex < n) {
        printf("%d ", tracks[closestIndex]);
        totalHeadMovements += abs(tracks[closestIndex] - head);
        head = tracks[closestIndex];
        closestIndex = findClosestTrack(tracks, n, head, direction);
    }

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

    // Free dynamically allocated memory
    free(tracks);

    return 0;
}
```

Q.2 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

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Structure to represent a block in the disk
typedef struct Block {
    int index;
    bool allocated;
    struct Block* next;
} Block;

// Function to initialize a disk with n blocks
Block* initializeDisk(int n) {
    Block* disk = NULL;
    for (int i = 1; i <= n; i++) {
        Block* newBlock = (Block*)malloc(sizeof(Block));
        newBlock->index = i;
        newBlock->allocated = false;
        newBlock->next = NULL;

        // Add block to the end of the disk
        if (disk == NULL) {
```

```
            disk = newBlock;
        } else {
            Block* temp = disk;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newBlock;
        }
    }
    return disk;
}

// Function to display the bit vector representing allocated and free blocks
void showBitVector(Block* disk) {
    printf("Bit Vector: ");
    while (disk != NULL) {
        if (disk->allocated) {
            printf("1 ");
        } else {
            printf("0 ");
        }
        disk = disk->next;
    }
    printf("\n");
}

// Function to create a new file by allocating blocks
void createNewFile(Block* disk) {
    int index;
    printf("Enter the starting index for the new file: ");
    scanf("%d", &index);

    // Find the block with the given index and allocate it
    Block* temp = disk;
    while (temp != NULL && temp->index != index) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Block with index %d not found.\n", index);
        return;
    }
    if (temp->allocated) {
        printf("Block with index %d is already allocated.\n", index);
        return;
    }
    temp->allocated = true;
    printf("File created successfully.\n");
}

// Function to display the directory showing allocated blocks
```

```
void showDirectory(Block* disk) {
    printf("Directory:\n");
    while (disk != NULL) {
        if (disk->allocated) {
            printf("Block %d: Allocated\n", disk->index);
        }
        disk = disk->next;
    }
}

// Function to deallocate a file by marking blocks as free
void deleteFile(Block* disk) {
    int index;
    printf("Enter the starting index of the file to be deleted: ");
    scanf("%d", &index);

    // Find the block with the given index and deallocate it
    Block* temp = disk;
    while (temp != NULL && temp->index != index) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Block with index %d not found.\n", index);
        return;
    }
    if (!temp->allocated) {
        printf("Block with index %d is already free.\n", index);
        return;
    }
    temp->allocated = false;
    printf("File deleted successfully.\n");
}

// Function to free memory allocated to the disk
void freeDisk(Block* disk) {
    Block* temp;
    while (disk != NULL) {
        temp = disk;
        disk = disk->next;
        free(temp);
    }
}

int main() {
    int n; // Number of disk blocks
    int choice;
    Block* disk = NULL;

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

```
    scanf("%d", &n);

    // Initialize the disk
    disk = initializeDisk(n);

    // Menu-driven program
    do {
        printf("\nMenu:\n");
        printf("1. Show Bit Vector\n");
        printf("2. Create New File\n");
        printf("3. Show Directory\n");
        printf("4. Delete File\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                showBitVector(disk);
                break;
            case 2:
                createNewFile(disk);
                break;
            case 3:
                showDirectory(disk);
                break;
            case 4:
                deleteFile(disk);
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 5);

    // Free memory allocated to the disk
    freeDisk(disk);

    return 0;
}
```

Slip 26

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

Proces
s
Allocation Max Available
A B C D A B C D A B C D
P0 0 0 1 2 0 0 1 2 1 5 2 0
P1 1 0 0 0 1 7 5 0
P2 1 3 5 4 2 3 5 6
P3 0 6 3 2 0 6 5 2
P4 0 0 1 4 0 6 5 6
a) Calculate and display the content of need matrix?
b) Is the system in safe state? If display the safe sequence.

Ans:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

// Function to calculate the need matrix
void calculateNeedMatrix(int need[MAX_PROCESSES][MAX_RESOURCES], int
max[MAX_PROCESSES][MAX_RESOURCES], int allocation[MAX_PROCESSES][MAX_RESOURCES],
int available[MAX_RESOURCES], int num_processes, int num_resources) {
    for (int i = 0; i < num_processes; i++) {
        for (int j = 0; j < num_resources; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

// Function to check if the system is in a safe state
bool isSafeState(int available[MAX_RESOURCES], int
allocation[MAX_PROCESSES][MAX_RESOURCES], int need[MAX_PROCESSES][MAX_RESOURCES],
int num_processes, int num_resources) {
    bool finish[MAX_PROCESSES] = { false };
    int work[MAX_RESOURCES];
    for (int i = 0; i < num_resources; i++) {
        work[i] = available[i];
    }

    int safeSeq[MAX_PROCESSES];
    int count = 0;

    while (count < num_processes) {
        bool found = false;
        for (int p = 0; p < num_processes; p++) {
            if (!finish[p]) {
                int j;
                for (j = 0; j < num_resources; j++) {
                    if (need[p][j] > work[j])
```

```
                    break;
                }
                if (j == num_resources) {
                    for (int k = 0; k < num_resources; k++) {
                        work[k] += allocation[p][k];
                    }
                    safeSeq[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state.\n");
            return false;
        }
    }
    printf("System is in a safe state.\nSafe sequence: ");
    for (int i = 0; i < num_processes; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf("\n");
    return true;
}

int main() {
    int allocation[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 0, 1, 2},
        {1, 0, 0, 0},
        {1, 3, 5, 4},
        {0, 6, 3, 2},
        {0, 0, 1, 4}
    };

    int max[MAX_PROCESSES][MAX_RESOURCES] = {
        {1, 5, 2, 0},
        {1, 7, 5, 0},
        {2, 3, 5, 6},
        {0, 6, 5, 2},
        {0, 6, 5, 6}
    };

    int available[MAX_RESOURCES] = {1, 3, 3, 2};

    int need[MAX_PROCESSES][MAX_RESOURCES];

    // Calculate the need matrix
    calculateNeedMatrix(need, max, allocation, available, MAX_PROCESSES,
MAX_RESOURCES);
```

```
    // Display the need matrix
    printf("Need Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    // Check if the system is in a safe state
    isSafeState(available, allocation, need, MAX_PROCESSES, MAX_RESOURCES);

    return 0;
}
```

Q.2 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.
56, 59, 40, 19, 91, 161, 151, 39, 185
Start Head Position: 48

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to calculate total head movements
int calculateHeadMovements(int diskRequests[], int numRequests, int startHeadPos) {
    int totalHeadMovements = 0;
    int currentHeadPos = startHeadPos;

    for (int i = 0; i < numRequests; i++) {
        totalHeadMovements += abs(diskRequests[i] - currentHeadPos);
        currentHeadPos = diskRequests[i];
    }

    return totalHeadMovements;
}

int main() {
    int numBlocks, startHeadPos;
    printf("Enter the total number of disk blocks: ");
```

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

    int *diskRequests = (int *)malloc(numBlocks * sizeof(int));
    if (diskRequests == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter the disk request string (separated by spaces): ");
    for (int i = 0; i < numBlocks; i++) {
        scanf("%d", &diskRequests[i]);
    }

    printf("Enter the start head position: ");
    scanf("%d", &startHeadPos);

    // Calculate total head movements
    int totalHeadMovements = calculateHeadMovements(diskRequests, numBlocks,
startHeadPos);

    // Display the list of request in the order in which it is served
    printf("Order of disk requests served: ");
    for (int i = 0; i < numBlocks; i++) {
        printf("%d ", diskRequests[i]);
    }
    printf("\n");

    // Display total head movements
    printf("Total number of head movements: %d\n", totalHeadMovements);

    // Free dynamically allocated memory
    free(diskRequests);

    return 0;
}
```

Slip 27

Q.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.

176, 79, 34, 60, 92, 11, 41, 114

Starting Head Position: 65

Direction: Right

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort an array in ascending order
void sortArray(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]) {
                // Swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to calculate total head movements using LOOK algorithm
int calculateHeadMovements(int diskRequests[], int numRequests, int startHeadPos,
int direction) {
    int totalHeadMovements = 0;
    int currentHeadPos = startHeadPos;
    int lowerBound = 0, upperBound = 0;

    // Sort disk requests
    sortArray(diskRequests, numRequests);

    // Find lower and upper bounds based on start head position
    for (int i = 0; i < numRequests; i++) {
        if (diskRequests[i] <= startHeadPos) {
            lowerBound = i;
        } else {
            upperBound = i;
            break;
        }
    }

    // Move towards lower bound
    for (int i = lowerBound; i >= 0; i--) {
        totalHeadMovements += abs(currentHeadPos - diskRequests[i]);
        currentHeadPos = diskRequests[i];
    }

    // Move towards upper bound if direction is right
    if (direction == 1) {
        for (int i = upperBound; i < numRequests; i++) {
            totalHeadMovements += abs(currentHeadPos - diskRequests[i]);
            currentHeadPos = diskRequests[i];
        }
```

```
    }

    return totalHeadMovements;
}

int main() {
    int numBlocks, startHeadPos, direction;
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &numBlocks);

    int *diskRequests = (int *)malloc(numBlocks * sizeof(int));
    if (diskRequests == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter the disk request string (separated by spaces): ");
    for (int i = 0; i < numBlocks; i++) {
        scanf("%d", &diskRequests[i]);
    }

    printf("Enter the start head position: ");
    scanf("%d", &startHeadPos);

    printf("Enter the direction (0 for left, 1 for right): ");
    scanf("%d", &direction);

    // Calculate total head movements
    int totalHeadMovements = calculateHeadMovements(diskRequests, numBlocks,
startHeadPos, direction);

    // Display the list of request in the order in which it is served
    printf("Order of disk requests served: ");
    for (int i = 0; i < numBlocks; i++) {
        printf("%d ", diskRequests[i]);
    }
    printf("\n");

    // Display total head movements
    printf("Total number of head movements: %d\n", totalHeadMovements);

    // Free dynamically allocated memory
    free(diskRequests);

    return 0;
}
```

Q.2 Write an MPI program to find the min number from randomly generated 1000

numbers
(stored in array) on a cluster (Hint: Use MPI_Reduce)

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int numbers[ARRAY_SIZE];
    int local_min = INT_MAX;
    int global_min;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize random number generator with rank-dependent seed
    srand(time(NULL) + rank);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        numbers[i] = rand() % 1000; // Generate random numbers between 0 and 999
    }

    // Find local minimum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        if (numbers[i] < local_min) {
            local_min = numbers[i];
        }
    }

    // Reduce local minimum to global minimum
    MPI_Reduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    // Display result
    if (rank == 0) {
        printf("Minimum number: %d\n", global_min);
    }

    MPI_Finalize();
    return 0;
}
```

Slip 28
Q.1 Write a simulation program for disk scheduling using C-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.
56, 59, 40, 19, 91, 161, 151, 39, 185
Start Head Position: 48
Direction: User Defined

Ans:

```
#include <stdio.h>
#include <stdlib.h>

// Function to sort an array in ascending order
void sortArray(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]) {
                // Swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to calculate total head movements using C-LOOK algorithm
int calculateHeadMovements(int diskRequests[], int numRequests, int startHeadPos,
int direction) {
    int totalHeadMovements = 0;
    int currentHeadPos = startHeadPos;
    int lowerBound = 0, upperBound = 0;

    // Find lower and upper bounds based on start head position
    for (int i = 0; i < numRequests; i++) {
        if (diskRequests[i] <= startHeadPos) {
            lowerBound = i;
        } else {
            upperBound = i;
            break;
        }
    }

    // Move towards upper bound if direction is right
    if (direction == 1) {
        for (int i = lowerBound; i < numRequests; i++) {
```

```
            totalHeadMovements += abs(currentHeadPos - diskRequests[i]);
            currentHeadPos = diskRequests[i];
        }

        // Move to the beginning of the disk
        totalHeadMovements += abs(currentHeadPos - diskRequests[0]);
        currentHeadPos = diskRequests[0];
    }

    // Move towards lower bound
    for (int i = lowerBound; i >= 0; i--) {
        totalHeadMovements += abs(currentHeadPos - diskRequests[i]);
        currentHeadPos = diskRequests[i];
    }

    return totalHeadMovements;
}

int main() {
    int numBlocks, startHeadPos, direction;
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &numBlocks);

    int *diskRequests = (int *)malloc(numBlocks * sizeof(int));
    if (diskRequests == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter the disk request string (separated by spaces): ");
    for (int i = 0; i < numBlocks; i++) {
        scanf("%d", &diskRequests[i]);
    }

    printf("Enter the start head position: ");
    scanf("%d", &startHeadPos);

    printf("Enter the direction (0 for left, 1 for right): ");
    scanf("%d", &direction);

    // Calculate total head movements
    int totalHeadMovements = calculateHeadMovements(diskRequests, numBlocks,
startHeadPos, direction);

    // Display the list of request in the order in which it is served
    printf("Order of disk requests served: ");
    for (int i = 0; i < numBlocks; i++) {
        printf("%d ", diskRequests[i]);
    }
    printf("\n");
```

```
    // Display total head movements
    printf("Total number of head movements: %d\n", totalHeadMovements);

    // Free dynamically allocated memory
    free(diskRequests);

    return 0;
}
```

Q.2 Write an MPI program to calculate sum of randomly generated 1000 numbers (stored in array) on a cluster

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int numbers[ARRAY_SIZE];
    int local_sum = 0;
    int global_sum = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize random number generator with rank-dependent seed
    srand(time(NULL) + rank);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        numbers[i] = rand() % 1000; // Generate random numbers between 0 and 999
    }

    // Calculate local sum
    for (int i = 0; i < ARRAY_SIZE; i++) {
        local_sum += numbers[i];
    }

    // Reduce local sums to global sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Display result
```

```
    if (rank == 0) {
        printf("Sum of numbers: %d\n", global_sum);
    }

    MPI_Finalize();
    return 0;
}
```

Slip 29
Q.1 Write an MPI program to calculate sum of all even randomly generated 1000 numbers
(stored in array) on a cluster.

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int numbers[ARRAY_SIZE];
    int local_sum = 0;
    int global_sum = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize random number generator with rank-dependent seed
    srand(time(NULL) + rank);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        numbers[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 (numbers[i] % 2 == 0) {
            local_sum += numbers[i];
        }
    }

    // Reduce local sums to global sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    // Display result
    if (rank == 0) {
        printf("Sum of even numbers: %d\n", global_sum);
    }

    MPI_Finalize();
    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-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.. [15]

80, 150, 60,135, 40, 35, 170
Starting Head Position: 70
Direction: Right

Ans:

```
#include <stdio.h>
#include <stdlib.h>

void c_look(int *requests, int n, int start, int direction) {
    int head = start;
    int total_head_movement = 0;

    // Sort the requests in ascending order
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (requests[j] > requests[j + 1]) {
                int temp = requests[j];
                requests[j] = requests[j + 1];
                requests[j + 1] = temp;
            }
        }
    }

    // Find the index of the first request greater than or equal to the head position
    int index;
    for (index = 0; index < n; index++) {
        if (requests[index] >= head) {
            break;
        }
    }
```

```
    // Serve requests in the right direction
    printf("Order of requests served: ");
    while (index < n) {
        printf("%d ", requests[index]);
        total_head_movement += abs(head - requests[index]);
        head = requests[index];
        index++;
    }

    // If direction is right and there are still requests pending, move to the
lowest request
    if (direction == 1 && index != n) {
        total_head_movement += abs(head - requests[0]);
        head = requests[0];
        printf("%d ", requests[0]);
    }

    // If direction is left, serve remaining requests in reverse order
    if (direction == -1 && index != 0) {
        index -= 1; // Move back to the last served request
        while (index >= 0) {
            printf("%d ", requests[index]);
            total_head_movement += abs(head - requests[index]);
            head = requests[index];
            index--;
        }
    }

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

int main() {
    int n, start, direction;

    printf("Enter the total number of disk blocks: ");
    scanf("%d", &n);

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

    printf("Enter the disk request string:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }

    printf("Enter the starting head position: ");
    scanf("%d", &start);

    printf("Enter the direction (1 for right, -1 for left): ");
    scanf("%d", &direction);
```

```
    c_look(requests, n, start, direction);

    free(requests);
    return 0;
}
```

Slip 30
Q.1 Write an MPI program to find the min number from randomly generated 1000 numbers
(stored in array) on a cluster (Hint: Use MPI_Reduce)

Ans:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
    int numbers[ARRAY_SIZE];
    int local_min = INT_MAX;
    int global_min;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize random number generator with rank-dependent seed
    srand(time(NULL) + rank);

    // Generate random numbers
    for (int i = 0; i < ARRAY_SIZE; i++) {
        numbers[i] = rand() % 1000; // Generate random numbers between 0 and 999
    }

    // Calculate local min
    for (int i = 0; i < ARRAY_SIZE; i++) {
        if (numbers[i] < local_min) {
            local_min = numbers[i];
        }
    }

    // Reduce local mins to global min
    MPI_Reduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    // Display result
    if (rank == 0) {
```

```
        printf("Minimum number: %d\n", global_min);
    }

    MPI_Finalize();
    return 0;
}
```

Q.2 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.
65, 95, 30, 91, 18, 116, 142, 44, 168
Start Head Position: 52

Ans:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, start_position, total_head_movements = 0;

    // Accepting total number of disk blocks
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &n);

    // Accepting disk request string
    int *requests = (int *)malloc(n * sizeof(int));
    printf("Enter the disk request string:\n");
    for (int i = 0; i < n; ++i) {
        scanf("%d", &requests[i]);
    }

    // Accepting current head position
    printf("Enter the starting head position: ");
    scanf("%d", &start_position);

    // Displaying the order of requests served and calculating total head movements
    printf("Order of requests served: ");
    for (int i = 0; i < n; ++i) {
        printf("%d ", requests[i]);
        total_head_movements += abs(requests[i] - start_position);
        start_position = requests[i];
    }

    // Displaying the total number of head movements
```

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

    // Freeing dynamically allocated memory
    free(requests);

    return 0;
}
```