Slip no 1

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.

```
#include <stdio.h>
#define PROCESS_COUNT 5
#define RESOURCE_COUNT 3
int available[RESOURCE_COUNT];
int allocation[PROCESS_COUNT][RESOURCE_COUNT] = {
   {2, 3, 2},
   {4, 0, 0},
   {5, 0, 4},
   {4, 3, 3},
   {2, 2, 4}
};
int max[PROCESS_COUNT][RESOURCE_COUNT] = {
   {9, 7, 5},
   {5, 2, 2},
   {1, 0, 4},
   {4, 4, 4},
   {6, 5, 5}
```

```
};
int need[PROCESS_COUNT][RESOURCE_COUNT];
void calculateNeedMatrix() {
    for (int i = 0; i < PROCESS_COUNT; i++) {
        for (int j = 0; j < RESOURCE_COUNT; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
void displayAllocationAndMax() {
    printf("Process\tAllocation\tMax\n");
    printf("\tA B C\t\tA B C\n");
    for (int i = 0; i < PROCESS_COUNT; i++) {
        printf("P%d\t", i);
        for (int j = 0; j < RESOURCE_COUNT; j++) {
            printf("%d ", allocation[i][j]);
        }
        printf("\t\t");
        for (int j = 0; j < RESOURCE_COUNT; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\n");
    }
}
void displayNeedMatrix() {
    printf("Process\tNeed\n");
    printf("\tA B C\n");
    for (int i = 0; i < PROCESS_COUNT; i++) {
```

```
        printf("P%d\t", i);
        for (int j = 0; j < RESOURCE_COUNT; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}
void displayAvailable() {
    printf("Available Resources: ");
    for (int i = 0; i < RESOURCE_COUNT; i++) {
        printf("%d ", available[i]);
    }
    printf("\n");
}
void acceptAvailable() {
    printf("Enter available resources (A B C): ");
    for (int i = 0; i < RESOURCE_COUNT; i++) {
        scanf("%d", &available[i]);
    }
}
int main() {
    int choice;
    calculateNeedMatrix();
    while (1) {
        printf("\nMenu:\n");
        printf("1. Accept Available\n");
        printf("2. Display Allocation and Max\n");
        printf("3. Display Need Matrix\n");
```

```
        printf("4. Display Available\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                acceptAvailable();
                break;
            case 2:
                displayAllocationAndMax();
                break;
            case 3:
                displayNeedMatrix();
                break;
            case 4:
                displayAvailable();
                break;
            case 5:
                return 0;
            default:
                printf("Invalid choice, try again.\n");
        }
    }
    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

```
#include <stdio.h>
#include <stdlib.h>
void fcfs(int requests[], int n, int head) {
    int totalMovement = 0;
    printf("Request served in order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", requests[i]);
        totalMovement += abs(requests[i] - head);
        head = requests[i];
    }
    printf("\nTotal head movements: %d\n", totalMovement);
}
int main() {
    int n, head;
    printf("Enter total number of requests: ");
    scanf("%d", &n);
    int requests[n];
    printf("Enter the disk request string: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the starting head position: ");
    scanf("%d", &head);
```

```
    fcfs(requests, n, head);
    return 0;
}
```

Slip no 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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_FILES 10
#define MAX_BLOCKS 100
int bitVector[MAX_BLOCKS];
int directory[MAX_FILES][MAX_BLOCKS];
int fileStart[MAX_FILES];
int fileCount = 0;
void initializeDisk(int n) {
    srand(time(0));
```

```
    for (int i = 0; i < n; i++) {
        bitVector[i] = rand() % 2;
    }
}
void showBitVector(int n) {
    printf("Bit Vector: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", bitVector[i]);
    }
    printf("\n");
}
void createNewFile(int n) {
    if (fileCount >= MAX_FILES) {
        printf("Directory full! Cannot create more files.\n");
        return;
    }
    int fileSize, block, currentBlock = -1, previousBlock = -1;
    printf("Enter file size: ");
    scanf("%d", &fileSize);
    if (fileSize > n) {
        printf("File size too large.\n");
        return;
    }
    fileStart[fileCount] = -1;
    for (int i = 0; i < fileSize; i++) {
        do {
            block = rand() % n;
        } while (bitVector[block] != 0);
```

```
            bitVector[block] = 1;
            if (previousBlock == -1) {
                fileStart[fileCount] = block;
            } else {
                directory[fileCount][previousBlock] = block;
            }
            previousBlock = block;
            directory[fileCount][block] = -1;
        }
        printf("File created with starting block %d\n", fileStart[fileCount]);
        fileCount++;
    }
    void showDirectory() {
        printf("Directory:\n");
        for (int i = 0; i < fileCount; i++) {
            printf("File %d: ", i + 1);
            int block = fileStart[i];
            while (block != -1) {
                printf("%d -> ", block);
                block = directory[i][block];
            }
            printf("NULL\n");
        }
    }
    int main() {
        int n, choice;
        printf("Enter total number of blocks: ");
        scanf("%d", &n);
```

```
    initializeDisk(n);
    while (1) {
        printf("\nMenu:\n");
        printf("1. Show Bit Vector\n");
        printf("2. Create New File\n");
        printf("3. Show Directory\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                showBitVector(n);
                break;
            case 2:
                createNewFile(n);
                break;
            case 3:
                showDirectory();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

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

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define ARRAY_SIZE 1000
int main(int argc, char *argv[]) {
   int rank, size, i, sum = 0, total_sum = 0;
   int numbers[ARRAY_SIZE], local_sum = 0;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0) {
      srand(time(NULL));
      for (i = 0; i < ARRAY_SIZE; i++) {
         numbers[i] = rand() % 100;
      }
   }
   int chunk_size = ARRAY_SIZE / size;
   int local_numbers[chunk_size];
   MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);
   for (i = 0; i < chunk_size; i++) {
```

```
        local_sum += local_numbers[i];
    }
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Total sum of 1000 numbers: %d\n", total_sum);
    }
    MPI_Finalize();
    return 0;
}
```

Slip no 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.

```
#include <stdio.h>
#define PROCESS_COUNT 5
#define RESOURCE_COUNT 4
int allocation[PROCESS_COUNT][RESOURCE_COUNT] = {
    {0, 0, 1, 2},
    {1, 0, 0, 0},
    {1, 3, 5, 4},
    {0, 6, 3, 2},
    {0, 0, 1, 4}
};
int max[PROCESS_COUNT][RESOURCE_COUNT] = {
    {0, 0, 1, 2},
    {1, 7, 5, 0},
    {2, 3, 5, 6},
    {0, 6, 5, 2},
    {0, 6, 5, 6}
};
int available[RESOURCE_COUNT] = {1, 5, 2, 0};
int need[PROCESS_COUNT][RESOURCE_COUNT];
void calculateNeedMatrix() {
    for (int i = 0; i < PROCESS_COUNT; i++) {
        for (int j = 0; j < RESOURCE_COUNT; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
```

```
int isSafeState() {
    int work[RESOURCE_COUNT];
    int finish[PROCESS_COUNT] = {0};
    int safeSequence[PROCESS_COUNT];
    int count = 0;
    for (int i = 0; i < RESOURCE_COUNT; i++) {
        work[i] = available[i];
    }
    while (count < PROCESS_COUNT) {
        int found = 0;
        for (int i = 0; i < PROCESS_COUNT; i++) {
            if (finish[i] == 0) {
                int j;
                for (j = 0; j < RESOURCE_COUNT; j++) {
                    if (need[i][j] > work[j])
                        break;
                }
                if (j == RESOURCE_COUNT) {
                    for (int k = 0; k < RESOURCE_COUNT; k++) {
                        work[k] += allocation[i][k];
                    }
                    safeSequence[count++] = i;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
```

```
            printf("System is not in a safe state.\n");
            return 0;
        }
    }
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < PROCESS_COUNT; i++) {
        printf("P%d ", safeSequence[i]);
    }
    printf("\n");
    return 1;
}
int main() {
    calculateNeedMatrix();
    printf("Need Matrix:\n");
    for (int i = 0; i < PROCESS_COUNT; i++) {
        for (int j = 0; j < RESOURCE_COUNT; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
    isSafeState();
    return 0;
}
```

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

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define ARRAY_SIZE 1000
int main(int argc, char *argv[]) {
   int rank, size, i;
   int numbers[ARRAY_SIZE], local_sum = 0, total_sum = 0;
   double average;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0) {
      srand(time(NULL));
      for (i = 0; i < ARRAY_SIZE; i++) {
         numbers[i] = rand() % 100;
      }
   }
   int chunk_size = ARRAY_SIZE / size;
   int local_numbers[chunk_size];
   MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);
   for (i = 0; i < chunk_size; i++) {
      local_sum += local_numbers[i];
   }
   MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if (rank == 0) {
```

```
        average = total_sum / (double)ARRAY_SIZE;
        printf("Total sum of 1000 numbers: %d\n", total_sum);
        printf("Average of 1000 numbers: %f\n", average);
    }
    MPI_Finalize();
    return 0;
}
```

Slip no 4

Q.1 Implement the Menu driven Banker's algorithm for accepting Allocation, Max fromuser.
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

```
#include <stdio.h>
#define PROCESS_COUNT 5
```

```
#define RESOURCE_COUNT 3
int allocation[PROCESS_COUNT][RESOURCE_COUNT] = {
    {0, 1, 0},
    {4, 0, 0},
    {5, 0, 4},
    {4, 3, 3},
    {2, 2, 4}
};
int max[PROCESS_COUNT][RESOURCE_COUNT] = {
    {0, 0, 0},
    {5, 2, 2},
    {1, 0, 4},
    {4, 4, 4},
    {6, 5, 5}
};
int available[RESOURCE_COUNT] = {7, 2, 6};
int need[PROCESS_COUNT][RESOURCE_COUNT];
void calculateNeedMatrix() {
    for (int i = 0; i < PROCESS_COUNT; i++) {
        for (int j = 0; j < RESOURCE_COUNT; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
void displayMatrix(int matrix[PROCESS_COUNT][RESOURCE_COUNT], char *name) {
    printf("%s Matrix:\n", name);
    for (int i = 0; i < PROCESS_COUNT; i++) {
        for (int j = 0; j < RESOURCE_COUNT; j++) {
```

```
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
void displayAvailable() {
    printf("Available Resources:\n");
    for (int i = 0; i < RESOURCE_COUNT; i++) {
        printf("%d ", available[i]);
    }
    printf("\n");
}
int main() {
    int choice;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Accept Available Resources\n");
        printf("2. Display Allocation and Max\n");
        printf("3. Find and Display Need\n");
        printf("4. Display Available Resources\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter available resources for A, B, and C: ");
                for (int i = 0; i < RESOURCE_COUNT; i++) {
                    scanf("%d", &available[i]);
```

```
            }
            break;
        case 2:
            displayMatrix(allocation, "Allocation");
            displayMatrix(max, "Max");
            break;
        case 3:
            calculateNeedMatrix();
            displayMatrix(need, "Need");
            break;
        case 4:
            displayAvailable();
            break;
        case 5:
            return 0;
        default:
            printf("Invalid choice!\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. 86, 147, 91, 170, 95, 130, 102, 70 Starting Head position= 125 Direction: Left

```
#include <stdio.h>
#include <stdlib.h>
void sort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
void scan(int requests[], int n, int head, int direction, int disk_size) {
    int total_movement = 0;
    int current = head;
    // Sort the requests in ascending order
    sort(requests, n);
    int index;
    for (index = 0; index < n; index++) {
        if (requests[index] > head) {
            break;
        }
    }
    printf("Order of servicing requests: ");
    if (direction == 0) { // Move left
        for (int i = index - 1; i >= 0; i--) {
```

```
            printf("%d ", requests[i]);
            total_movement += abs(current - requests[i]);
            current = requests[i];
        }
        total_movement += current;
        current = 0;
        for (int i = index; i < n; i++) {
            printf("%d ", requests[i]);
            total_movement += abs(current - requests[i]);
            current = requests[i];
        }
    } else { // Move right
        for (int i = index; i < n; i++) {
            printf("%d ", requests[i]);
            total_movement += abs(current - requests[i]);
            current = requests[i];
        }
        total_movement += abs(disk_size - 1 - current);
        current = disk_size - 1;
        for (int i = index - 1; i >= 0; i--) {
            printf("%d ", requests[i]);
            total_movement += abs(current - requests[i]);
            current = requests[i];
        }
    }
    printf("\nTotal head movement: %d\n", total_movement);
}
int main() {
```

```
    int n, head, direction, disk_size;
    printf("Enter total number of disk blocks: ");
    scanf("%d", &disk_size);
    printf("Enter number of requests: ");
    scanf("%d", &n);
    int requests[n];
    printf("Enter the request string: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter starting head position: ");
    scanf("%d", &head);
    printf("Enter direction (0 for Left, 1 for Right): ");
    scanf("%d", &direction);
    scan(requests, n, head, direction, disk_size);
    return 0;
}
```

Slip no 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

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

```
int main() {
    int m, n;
    printf("Enter number of processes (m): ");
    scanf("%d", &m);
    printf("Enter number of resource types (n): ");
    scanf("%d", &n);
    int allocation[m][n], max[m][n], need[m][n], available[n];
    int request[n], process;
    // Input Available resources for each resource type
    printf("Enter the number of available instances for each resource type: \n");
    for (int i = 0; i < n; i++) {
        printf("Resource %d: ", i + 1);
        scanf("%d", &available[i]);
    }
    // Input Allocation matrix
    printf("Enter the Allocation matrix:\n");
    for (int i = 0; i < m; i++) {
        printf("Process %d Allocation: ", i);
        for (int j = 0; j < n; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }
    // Input Maximum requirement matrix
    printf("Enter the Maximum requirement matrix:\n");
    for (int i = 0; i < m; i++) {
        printf("Process %d Maximum: ", i);
        for (int j = 0; j < n; j++) {
            scanf("%d", &max[i][j]);
```

```
    }
}
// Calculate the Need matrix (Need = Max - Allocation)
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        need[i][j] = max[i][j] - allocation[i][j];
    }
}
// Display the Need matrix
printf("\nNeed Matrix:\n");
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        printf("%d ", need[i][j]);
    }
    printf("\n");
}
// Accept a request from a process
printf("\nEnter the process number making the request (0 to %d): ", m - 1);
scanf("%d", &process);
printf("Enter the resource request for process %d:\n", process);
for (int i = 0; i < n; i++) {
    printf("Resource %d: ", i + 1);
    scanf("%d", &request[i]);
}
// Check if the request can be granted immediately
int can_be_granted = 1;
// Check if the request is less than the available resources
for (int i = 0; i < n; i++) {
```

```
        if (request[i] > available[i]) {
            can_be_granted = 0;
            break;
        }
    }
    // Check if the request is less than the need for the process
    for (int i = 0; i < n; i++) {
        if (request[i] > need[process][i]) {
            can_be_granted = 0;
            break;
        }
    }
    if (can_be_granted) {
        printf("\nThe request can be granted immediately.\n");
    } else {
        printf("\nThe 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)

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

```
#include <time.h>
#define N 1000
int main(int argc, char *argv[]) {
   int rank, size;
   int arr[N], local_max, global_max;
   int local_start, local_end;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   srand(time(NULL) + rank);
   int local_array_size = N / size;
   int local_array[local_array_size];
   if (rank == 0) {
      for (int i = 0; i < N; i++) {
         arr[i] = rand() % 1000;
      }
   }
   MPI_Scatter(arr, local_array_size, MPI_INT, local_array, local_array_size, MPI_INT, 0,
MPI_COMM_WORLD);
   local_max = local_array[0];
   for (int i = 1; i < local_array_size; i++) {
      if (local_array[i] > local_max) {
         local_max = local_array[i];
      }
   }
   MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
   if (rank == 0) {
```

```
        printf("The maximum value is %d\n", global_max);
    }
    MPI_Finalize();
    return 0;
}
```

Slip no 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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 100
int n;  // Number of blocks
int bitVector[MAX_BLOCKS];  // Bit vector for allocation
int directory[MAX_BLOCKS];  // Directory to store file block allocations
void showBitVector() {
    printf("Bit Vector (0=Free, 1=Allocated):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", bitVector[i]);
    }
    printf("\n");
}
```

```
void createNewFile() {
    int fileSize;
    printf("Enter the size of the new file (number of blocks): ");
    scanf("%d", &fileSize);

    int allocatedBlocks = 0;
    int fileBlocks[fileSize];
    for (int i = 0; i < n && allocatedBlocks < fileSize; i++) {
        if (bitVector[i] == 0) {  // Free block found
            bitVector[i] = 1;
            fileBlocks[allocatedBlocks] = i;
            allocatedBlocks++;
        }
    }
    if (allocatedBlocks == fileSize) {
        printf("File created successfully with blocks: ");
        for (int i = 0; i < fileSize; i++) {
            printf("%d ", fileBlocks[i]);
        }
        printf("\n");
    } else {
        printf("Not enough free blocks to create the file.\n");
    }
}
void showDirectory() {
    printf("Directory (File Allocations):\n");
    for (int i = 0; i < n; i++) {
        if (bitVector[i] == 1) {
```

```
            printf("Block %d allocated\n", i);
        }
    }
}
int main() {
    int choice;
    printf("Enter the total number of blocks: ");
    scanf("%d", &n);
    // Initialize bit vector and directory
    for (int i = 0; i < n; i++) {
        bitVector[i] = 0;  // 0 means free block
        directory[i] = -1;  // No file allocated yet
    }
    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();
                break;
            case 2:
                createNewFile();
                break;
```

```
            case 3:
                showDirectory();
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 4);
    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

```
#include <stdio.h>
#include <stdlib.h>
void sortRequests(int requests[], int n) {
    int temp;
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            if (requests[i] > requests[j]) {
                temp = requests[i];
                requests[i] = requests[j];
```

```
                requests[j] = temp;
            }
        }
    }
}
void cscan(int requests[], int n, int head, int diskSize, char direction) {
    int totalHeadMovements = 0;
    int index = 0;
    int servedRequests[n];

    sortRequests(requests, n);

    if (direction == 'R') {
        for (int i = 0; i < n; i++) {
            if (requests[i] >= head) {
                index = i;
                break;
            }
        }

        for (int i = index; i < n; i++) {
            servedRequests[i - index] = requests[i];
        }

        for (int i = 0; i < index; i++) {
            servedRequests[n - index + i] = requests[i];
        }
```

```
        totalHeadMovements += (diskSize - 1 - head) + (diskSize - 1) - servedRequests[n-1];

        printf("Requests in the order they are served:\n");
        for (int i = 0; i < n; i++) {
            printf("%d ", servedRequests[i]);
        }
    }
    printf("\nTotal head movements: %d\n", totalHeadMovements);
}
int main() {
    int requests[] = {80, 150, 60, 135, 40, 35, 170};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head, diskSize = 200;
    char direction;
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    printf("Enter the direction (R for Right, L for Left): ");
    scanf(" %c", &direction);
    cscan(requests, n, head, diskSize, direction);
    return 0;
}
```

Slip no 7

Q.1 Consider the following snapshot of the system. Proces s 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

```
#include <stdio.h>
#include <stdbool.h>
#define P 5
#define R 4
void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {
   for (int i = 0; i < P; i++) {
      for (int j = 0; j < R; j++) {
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }
}
bool isSafe(int available[], int max[][R], int allocation[][R], int need[][R]) {
   int work[R];
   bool finish[P] = {false};

   for (int i = 0; i < R; i++) {
      work[i] = available[i];
   }

   while (true) {
```

```
    bool found = false;
    for (int i = 0; i < P; i++) {
        if (!finish[i]) {
            bool canAllocate = true;
            for (int j = 0; j < R; j++) {
                if (need[i][j] > work[j]) {
                    canAllocate = false;
                    break;
                }
            }
            if (canAllocate) {
                for (int j = 0; j < R; j++) {
                    work[j] += allocation[i][j];
                }
                finish[i] = true;
                found = true;
                break;
            }
        }
    }
    if (!found) {
        break;
    }
}

for (int i = 0; i < P; i++) {
    if (!finish[i]) {
        return false;
```

```
        }
    }
    return true;
}
int main() {
    int allocation[P][R] = {
        {2, 0, 0, 1},
        {3, 1, 2, 1},
        {2, 1, 0, 3},
        {1, 3, 1, 2},
        {1, 4, 3, 2}
    };
    int max[P][R] = {
        {4, 2, 1, 2},
        {5, 2, 5, 2},
        {2, 3, 1, 6},
        {1, 4, 2, 4},
        {3, 6, 6, 5}
    };
    int available[R] = {3, 3, 2, 1};
    int need[P][R];

    calculateNeed(need, max, allocation);

    if (isSafe(available, max, allocation, need)) {
        printf("The system is in a safe state.\n");
    } else {
        printf("The 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

```
#include <stdio.h>
#include <stdlib.h>
void sortRequests(int requests[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (requests[i] > requests[j]) {
                temp = requests[i];
                requests[i] = requests[j];
                requests[j] = temp;
            }
        }
    }
}
void scan(int requests[], int n, int head, int diskSize, char direction) {
    int totalHeadMovements = 0;
    int left[100], right[100];
    int leftCount = 0, rightCount = 0;
```

```
for (int i = 0; i < n; i++) {
    if (requests[i] < head) {
        left[leftCount++] = requests[i];
    } else {
        right[rightCount++] = requests[i];
    }
}
sortRequests(left, leftCount);
sortRequests(right, rightCount);
if (direction == 'R') {
    totalHeadMovements += (diskSize - 1 - head);
    for (int i = 0; i < rightCount; i++) {
        totalHeadMovements += abs(head - right[i]);
        head = right[i];
    }
    totalHeadMovements += (head - 0);
    head = 0;
    for (int i = leftCount - 1; i >= 0; i--) {
        totalHeadMovements += abs(head - left[i]);
        head = left[i];
    }
}
printf("Requests in the order they are served:\n");
for (int i = 0; i < rightCount; i++) {
    printf("%d ", right[i]);
}
for (int i = leftCount - 1; i >= 0; i--) {
```

```
        printf("%d ", left[i]);
    }
    printf("\nTotal head movements: %d\n", totalHeadMovements);
}
int main() {
    int requests[] = {82, 170, 43, 140, 24, 16, 190};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head, diskSize = 200;
    char direction;
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    printf("Enter the direction (R for Right, L for Left): ");
    scanf(" %c", &direction);
    scan(requests, n, head, diskSize, direction);
    return 0;
}
```

Slip no 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

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

```
#include <stdlib.h>
#define MAX_BLOCKS 100
int n;
int bitVector[MAX_BLOCKS];
int directory[MAX_BLOCKS];
void showBitVector() {
    printf("Bit Vector: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", bitVector[i]);
    }
    printf("\n");
}
void createNewFile(int fileSize) {
    int start = -1;
    for (int i = 0; i <= n - fileSize; i++) {
        int found = 1;
        for (int j = i; j < i + fileSize; j++) {
            if (bitVector[j] == 1) {
                found = 0;
                break;
            }
        }
        if (found) {
            start = i;
            break;
        }
    }
    if (start == -1) {
```

```
        printf("Not enough space to allocate the file.\n");
        return;
    }
    for (int i = start; i < start + fileSize; i++) {
        bitVector[i] = 1;
        directory[i] = 1;
    }
    printf("File allocated starting from block %d.\n", start);
}
void showDirectory() {
    printf("Directory: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", directory[i]);
    }
    printf("\n");
}
int main() {
    int choice, fileSize;

    printf("Enter the number of blocks in the disk: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        bitVector[i] = 0;
        directory[i] = 0;
    }
    while (1) {
        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();
                break;
            case 2:
                printf("Enter the size of the file to create: ");
                scanf("%d", &fileSize);
                createNewFile(fileSize);
                break;
            case 3:
                showDirectory();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice. Please try again.\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. 186, 89, 44, 70, 102, 22, 51, 124 Start Head Position: 70

```
#include <stdio.h>
#include <stdlib.h>
void sortRequests(int requests[], int n, int head) {
   int temp;
   int minDist, minIndex;
   for (int i = 0; i < n - 1; i++) {
      minDist = abs(requests[i] - head);
      minIndex = i;
      for (int j = i + 1; j < n; j++) {
         if (abs(requests[j] - head) < minDist) {
            minDist = abs(requests[j] - head);
            minIndex = j;
         }
      }
      if (minIndex != i) {
         temp = requests[i];
         requests[i] = requests[minIndex];
         requests[minIndex] = temp;
      }
   }
}
void sstf(int requests[], int n, int head) {
   int totalHeadMovements = 0;
```

```
    int served[n];
    for (int i = 0; i < n; i++) {
        served[i] = 0;
    }
    printf("Request order: ");
    for (int i = 0; i < n; i++) {
        int minDist = -1;
        int index = -1;
        for (int j = 0; j < n; j++) {
            if (!served[j]) {
                int dist = abs(requests[j] - head);
                if (minDist == -1 || dist < minDist) {
                    minDist = dist;
                    index = j;
                }
            }
        }
        served[index] = 1;
        totalHeadMovements += minDist;
        head = requests[index];
        printf("%d ", requests[index]);
    }
    printf("\nTotal head movements: %d\n", totalHeadMovements);
}
int main() {
    int requests[] = {186, 89, 44, 70, 102, 22, 51, 124};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head;
```

```
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    sstf(requests, n, head);
    return 0;
}
\
```

Slip no 9

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.

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

Starting Head Position: 65

Direction: Left

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

void look(int request[], int n, int head, char direction[]) {
    int seek_count = 0;
    int distance, cur_track;
    int left[100], right[100];
    int l = 0, r = 0;

    // Split requests into left and right based on the head position
```

```
for (int i = 0; i < n; i++) {
    if (request[i] < head)
        left[l++] = request[i];
    else
        right[r++] = request[i];
}

// Sort both left and right arrays
for (int i = 0; i < l - 1; i++) {
    for (int j = 0; j < l - i - 1; j++) {
        if (left[j] < left[j + 1]) {
            int temp = left[j];
            left[j] = left[j + 1];
            left[j + 1] = temp;
        }
    }
}

for (int i = 0; i < r - 1; i++) {
    for (int j = 0; j < r - i - 1; j++) {
        if (right[j] > right[j + 1]) {
            int temp = right[j];
            right[j] = right[j + 1];
            right[j + 1] = temp;
        }
    }
}
```

```
// Service the requests based on the direction
printf("Sequence of serviced tracks: \n");
if (strcmp(direction, "Left") == 0) {
    // Move left first
    for (int i = 0; i < l; i++) {
        cur_track = left[i];
        printf("%d ", cur_track);
        distance = abs(head - cur_track);
        seek_count += distance;
        head = cur_track;
    }
    // Then move right
    for (int i = 0; i < r; i++) {
        cur_track = right[i];
        printf("%d ", cur_track);
        distance = abs(head - cur_track);
        seek_count += distance;
        head = cur_track;
    }
} else {
    // Move right first
    for (int i = 0; i < r; i++) {
        cur_track = right[i];
        printf("%d ", cur_track);
        distance = abs(head - cur_track);
        seek_count += distance;
        head = cur_track;
    }
```

```
        // Then move left
        for (int i = 0; i < l; i++) {
            cur_track = left[i];
            printf("%d ", cur_track);
            distance = abs(head - cur_track);
            seek_count += distance;
            head = cur_track;
        }
    }

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

int main() {
    int n, head;
    char direction[10];

    printf("Enter number of requests: ");
    scanf("%d", &n);

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

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

```
    printf("Enter direction (Left/Right): ");
    scanf("%s", direction);

    look(request, n, head, direction);

    return 0;
}
```

Slip no 10

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

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define SIZE 1000
int main(int argc, char *argv[]) {
    int rank, size;
    int numbers[SIZE], local_sum = 0, total_sum = 0;
    float average;

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

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Generate random numbers on the root process
    if (rank == 0) {
        for (int i = 0; i < SIZE; i++) {
            numbers[i] = rand() % 1000;  // Random numbers between 0 and 999
        }
    }
    // Distribute the data among all processes
    int chunk_size = SIZE / size;
    int local_numbers[chunk_size];
    MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);
    // Calculate local sum
    for (int i = 0; i < chunk_size; i++) {
        local_sum += local_numbers[i];
    }
    // Reduce the local sums to compute the total sum
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    // Calculate average on the root process
    if (rank == 0) {
        average = (float)total_sum / SIZE;
        printf("Total Sum: %d\n", total_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

```
#include <stdio.h>
#include <stdlib.h>
void sortRequests(int requests[], int n) {
   int temp;
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (requests[i] > requests[j]) {
            temp = requests[i];
            requests[i] = requests[j];
            requests[j] = temp;
         }
      }
   }
}
void cscan(int requests[], int n, int head, int total_blocks, int direction) {
   int total_head_movements = 0;
   int served[n];
   for (int i = 0; i < n; i++) {
      served[i] = 0;
   }
   sortRequests(requests, n);
```

```
int left[n], right[n], left_count = 0, right_count = 0;
for (int i = 0; i < n; i++) {
    if (requests[i] < head) {
        left[left_count++] = requests[i];
    } else {
        right[right_count++] = requests[i];
    }
}
if (direction == 0) {
    for (int i = left_count - 1; i >= 0; i--) {
        total_head_movements += abs(head - left[i]);
        head = left[i];
        printf("%d ", head);
    }
    total_head_movements += abs(head - 0);
    head = 0;
    for (int i = 0; i < right_count; i++) {
        total_head_movements += abs(head - right[i]);
        head = right[i];
        printf("%d ", head);
    }
} else {
    for (int i = 0; i < right_count; i++) {
        total_head_movements += abs(head - right[i]);
        head = right[i];
        printf("%d ", head);
    }
```

```
        total_head_movements += abs(head - (total_blocks - 1));
        head = total_blocks - 1;
        for (int i = left_count - 1; i >= 0; i--) {
            total_head_movements += abs(head - left[i]);
            head = left[i];
            printf("%d ", head);
        }
    }
    printf("\nTotal head movements: %d\n", total_head_movements);
}
int main() {
    int requests[] = {33, 99, 142, 52, 197, 79, 46, 65};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head, direction;
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    printf("Enter the direction (0 for left, 1 for right): ");
    scanf("%d", &direction);
    cscan(requests, n, head, 200, direction);
    return 0;
}
```

Slip no 12

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

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

```
#define SIZE 1000

int main(int argc, char* argv[]) {
    int rank, size;
    int numbers[SIZE];
    int local_sum = 0, total_sum = 0;
    double average = 0.0;
    int chunk_size;

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

    chunk_size = SIZE / size;
    int local_numbers[chunk_size];

    // Master process generates random numbers
    if (rank == 0) {
        srand(42);  // Seed for consistency
        for (int i = 0; i < SIZE; i++) {
            numbers[i] = rand() % 100;
        }
    }

    // Scatter numbers to all processes
    MPI_Scatter(numbers, chunk_size, MPI_INT,
                local_numbers, chunk_size, MPI_INT,
```

```
            0, MPI_COMM_WORLD);

    // Each process computes its local sum
    for (int i = 0; i < chunk_size; i++) {
        local_sum += local_numbers[i];
    }

    // Gather all local sums to compute total sum
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Master process calculates the average
    if (rank == 0) {
        average = (double)total_sum / SIZE;
        printf("Total Sum: %d\n", total_sum);
        printf("Average: %.2f\n", 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

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

// Function to sort the requests array
void sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n, head, total_movement = 0, direction;

    // Take user inputs
    printf("Enter the total number of requests: ");
    scanf("%d", &n);

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

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

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

// Sort the request queue
sort(requests, n);

// Find the index where the head should start in the sorted array
int index = 0;
while (index < n && requests[index] < head) {
    index++;
}

printf("\nSeek Sequence: ");

// Move in the chosen direction (Right)
if (direction == 1) {
    for (int i = index; i < n; i++) {
        printf("%d ", requests[i]);
        total_movement += abs(head - requests[i]);
        head = requests[i];
    }
    // Wrap around to the beginning
    for (int i = 0; i < index; i++) {
        printf("%d ", requests[i]);
```

```
            total_movement += abs(head - requests[i]);
            head = requests[i];
        }
    }
    // Move in the other direction (Left)
    else {
        for (int i = index - 1; i >= 0; i--) {
            printf("%d ", requests[i]);
            total_movement += abs(head - requests[i]);
            head = requests[i];
        }
        // Wrap around to the end
        for (int i = n - 1; i >= index; i--) {
            printf("%d ", requests[i]);
            total_movement += abs(head - requests[i]);
            head = requests[i];
        }
    }

    // Display total head movement
    printf("\nTotal Head Movement: %d\n", total_movement);

    return 0;
}
```

Slip no 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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 100
int disk[MAX_BLOCKS];
int n;
void showBitVector() {
   printf("Bit Vector: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", disk[i]);
   }
   printf("\n");
}
void showDirectory() {
   printf("Directory (Allocated blocks): ");
   for (int i = 0; i < n; i++) {
      if (disk[i] == 1) {
         printf("%d ", i);
      }
```

```
    }
    printf("\n");
}
void deleteFile(int start, int length) {
    int i;
    for (i = start; i < start + length && i < n; i++) {
        if (disk[i] == 1) {
            disk[i] = 0;
        }
    }
    printf("File deleted from blocks %d to %d\n", start, i-1);
}
int main() {
    int option, start, length;

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

    for (int i = 0; i < n; i++) {
        disk[i] = rand() % 2;
    }
    while (1) {
        printf("\nMenu:\n");
        printf("1. Show Bit Vector\n");
        printf("2. Show Directory\n");
        printf("3. Delete File\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
```

```
        scanf("%d", &option);
        switch (option) {
            case 1:
                showBitVector();
                break;
            case 2:
                showDirectory();
                break;
            case 3:
                printf("Enter the starting block and length of the file to delete: ");
                scanf("%d %d", &start, &length);
                deleteFile(start, length);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\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. 55, 58, 39, 18, 90, 160, 150, 38, 184 Start Head Position: 50

```
#include <stdio.h>
#include <stdlib.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int findClosestRequest(int requests[], int n, int head) {
    int min_dist = 99999, index = -1;
    for (int i = 0; i < n; i++) {
        if (requests[i] != -1) {
            int dist = abs(requests[i] - head);
            if (dist < min_dist) {
                min_dist = dist;
                index = i;
            }
        }
    }
    return index;
}
void sstf(int requests[], int n, int head) {
    int total_head_movements = 0;
    int served[n];
    for (int i = 0; i < n; i++) {
        served[i] = 0;
    }
    printf("Disk Requests Order: ");
    for (int i = 0; i < n; i++) {
```

```
        int index = findClosestRequest(requests, n, head);
        total_head_movements += abs(head - requests[index]);
        head = requests[index];
        requests[index] = -1;  // Mark the request as served
        printf("%d ", head);
    }
    printf("\nTotal head movements: %d\n", total_head_movements);
}
int main() {
    int requests[] = {55, 58, 39, 18, 90, 160, 150, 38, 184};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head;
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    sstf(requests, n, head);
    return 0;
}
```

Slip no 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

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

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

#define MAX_FILES 10
#define FILENAME_LEN 20

typedef struct Block {
    int block_num;
    struct Block *next;
} Block;

typedef struct File {
    char name[FILENAME_LEN];
    Block *start;
} File;

int *bitVector;
int total_blocks;
File directory[MAX_FILES];
int file_count = 0;

void initializeDisk(int n) {
    total_blocks = n;
    bitVector = (int *)malloc(n * sizeof(int));
    srand(time(0));
    for (int i = 0; i < n; i++) {
        bitVector[i] = rand() % 2; // Randomly allocate blocks
    }
```

```
}

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

int findFreeBlock() {
    for (int i = 0; i < total_blocks; i++) {
        if (bitVector[i] == 0) return i;
    }
    return -1;
}

void createNewFile() {
    if (file_count >= MAX_FILES) {
        printf("Directory is full!\n");
        return;
    }
    char filename[FILENAME_LEN];
    int size;
    printf("Enter file name: ");
    scanf("%s", filename);
    printf("Enter number of blocks required: ");
    scanf("%d", &size);
```

```
    Block *head = NULL, *temp = NULL;
    for (int i = 0; i < size; i++) {
        int block_num = findFreeBlock();
        if (block_num == -1) {
            printf("Not enough free blocks available!\n");
            return;
        }
        bitVector[block_num] = 1;
        Block *newBlock = (Block *)malloc(sizeof(Block));
        newBlock->block_num = block_num;
        newBlock->next = NULL;
        if (head == NULL) head = newBlock;
        else temp->next = newBlock;
        temp = newBlock;
    }

    File newFile;
    strcpy(newFile.name, filename);
    newFile.start = head;
    directory[file_count++] = newFile;
    printf("File created successfully!\n");
}

void showDirectory() {
    if (file_count == 0) {
        printf("Directory is empty!\n");
        return;
```

```
    }
    printf("Directory:\n");
    for (int i = 0; i < file_count; i++) {
        printf("%s -> ", directory[i].name);
        Block *temp = directory[i].start;
        while (temp) {
            printf("%d ", temp->block_num);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    int n;
    printf("Enter total number of blocks in disk: ");
    scanf("%d", &n);
    initializeDisk(n);

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

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

    free(bitVector);
    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

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

```
#include <stdlib.h>

void sort(int arr[], int n) {
    // Simple bubble sort for sorting requests
    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;
            }
        }
    }
}

int main() {
    int total_blocks, n_requests, head, direction;
    printf("Enter total number of disk blocks: ");
    scanf("%d", &total_blocks);

    printf("Enter number of disk requests: ");
    scanf("%d", &n_requests);

    int requests[n_requests];
    printf("Enter disk request queue: ");
    for (int i = 0; i < n_requests; i++) {
        scanf("%d", &requests[i]);
    }
```

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

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

// Sort the request array
sort(requests, n_requests);

int total_head_movement = 0;
int index = 0;

// Find the index where head is or would be placed
for (int i = 0; i < n_requests; i++) {
    if (requests[i] >= head) {
        index = i;
        break;
    }
}

printf("Request order: ");

if (direction == 1) { // Moving right
    for (int i = index; i < n_requests; i++) {
        printf("%d ", requests[i]);
        total_head_movement += abs(requests[i] - head);
        head = requests[i];
```

```
    }
    // Move to end of disk
    total_head_movement += abs(total_blocks - 1 - head);
    head = total_blocks –
```

Slip no 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

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

#define MAX_BLOCKS 100
#define MAX_FILES 10

typedef struct {
   char name[20];
   int start;
   int length;
```

```
} File;

int disk[MAX_BLOCKS];
File directory[MAX_FILES];
int file_count = 0;
int n;

void initialize_disk() {
    printf("Enter number of blocks on the disk: ");
    scanf("%d", &n);
    if (n > MAX_BLOCKS) n = MAX_BLOCKS;

    for (int i = 0; i < n; i++) {
        disk[i] = rand() % 2; // Randomly allocate blocks (0 = free, 1 = allocated)
    }
}

void show_bit_vector() {
    printf("Disk Bit Vector:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
        if ((i + 1) % 10 == 0) printf("\n");
    }
    printf("\n");
}

void create_new_file() {
    if (file_count >= MAX_FILES) {
```

```
    printf("Directory is full!\n");
    return;
}

char filename[20];
int length;
printf("Enter file name: ");
scanf("%s", filename);
printf("Enter file length (in blocks): ");
scanf("%d", &length);

int start = -1;
int count = 0;

for (int i = 0; i < n; i++) {
    if (disk[i] == 0) {
        if (count == 0) start = i;
        count++;
    } else {
        count = 0;
        start = -1;
    }

    if (count == length) break;
}

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

```
            disk[i] = 1; // Mark blocks as allocated
        }
        strcpy(directory[file_count].name, filename);
        directory[file_count].start = start;
        directory[file_count].length = length;
        file_count++;
        printf("File '%s' created at block %d, length %d blocks.\n", filename, start, length);
    } else {
        printf("Not enough contiguous space for the file.\n");
    }
}

void show_directory() {
    printf("\nDirectory:\n");
    printf("%-20s %-10s %-10s\n", "File Name", "Start", "Length");
    for (int i = 0; i < file_count; i++) {
        printf("%-20s %-10d %-10d\n", directory[i].name, directory[i].start, directory[i].length);
    }
    if (file_count == 0) printf("No files in the directory.\n");
}

int main() {
    initialize_disk();

    int choice;
    do {
        printf("\nMenu:\n");
        printf("1. Show Bit Vector\n");
```

```
        printf("2. Create New File\n");
        printf("3. Show Directory\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                show_bit_vector();
                break;
            case 2:
                create_new_file();
                break;
            case 3:
                show_directory();
                break;
            case 4:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice. Try again.\n");
        }
    } while (choice != 4);

    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)

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

#define SIZE 1000

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

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

    // Seed the random generator differently for each rank
    srand(time(NULL) + rank);

    // Each process generates its portion of numbers
    int chunk_size = SIZE / size;
    int local_numbers[chunk_size];

    // Root process generates the full array
    if (rank == 0) {
```

```
        printf("Generated numbers:\n");
        for (int i = 0; i < SIZE; i++) {
            numbers[i] = rand() % 10000;
            printf("%d ", numbers[i]);
        }
        printf("\n");
    }

    // Scatter the numbers to all processes
    MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process finds its local minimum
    local_min = local_numbers[0];
    for (int i = 1; i < chunk_size; i++) {
        if (local_numbers[i] < local_min) {
            local_min = local_numbers[i];
        }
    }

    // Reduce all local minimums to get the global minimum at rank 0
    MPI_Reduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    // Root process prints the final minimum
    if (rank == 0) {
        printf("The minimum number is: %d\n", global_min);
    }
```

```
    MPI_Finalize();
    return 0;
}
```

Slip no 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

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

#define MAX_BLOCKS 100
#define MAX_FILES 10

int n; // Total number of blocks
int bitVector[MAX_BLOCKS];

struct File {
    char name[20];
    int indexBlock;
    int blocks[10];
    int blockCount;
```

```
} directory[MAX_FILES];

int fileCount = 0;

// Initialize bit vector with random allocations
void initializeDisk() {
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        bitVector[i] = rand() % 2;  // Randomly allocate blocks
    }
}

// Show the bit vector
void showBitVector() {
    printf("\nBit Vector: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", bitVector[i]);
    }
    printf("\n");
}

// Show the directory
void showDirectory() {
    printf("\nDirectory:\n");
    if (fileCount == 0) {
        printf("No files available.\n");
        return;
    }
```

```
    for (int i = 0; i < fileCount; i++) {
        printf("File: %s\nIndex Block: %d\nBlocks: ", directory[i].name, directory[i].indexBlock);
        for (int j = 0; j < directory[i].blockCount; j++) {
            printf("%d ", directory[i].blocks[j]);
        }
        printf("\n");
    }
}

// Delete an existing file
void deleteFile() {
    char fileName[20];
    printf("Enter file name to delete: ");
    scanf("%s", fileName);

    for (int i = 0; i < fileCount; i++) {
        if (strcmp(directory[i].name, fileName) == 0) {
            bitVector[directory[i].indexBlock] = 0; // Free index block
            for (int j = 0; j < directory[i].blockCount; j++) {
                bitVector[directory[i].blocks[j]] = 0; // Free data blocks
            }
            printf("File '%s' deleted successfully.\n", fileName);
            for (int k = i; k < fileCount - 1; k++) {
                directory[k] = directory[k + 1];
            }
            fileCount--;
            return;
        }
```

```
    }
    printf("File not found.\n");
}

// Menu-driven program
int main() {
    printf("Enter number of blocks in the disk: ");
    scanf("%d", &n);

    if (n > MAX_BLOCKS) {
        printf("Number of blocks exceeds maximum limit!\n");
        return 1;
    }

    initializeDisk();

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

        switch (choice) {
            case 1:
```

```
            showBitVector();
            break;
        case 2:
            showDirectory();
            break;
        case 3:
            deleteFile();
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice!\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. 23, 89, 132, 42, 187, 69, 36, 55

Start Head Position: 40

Direction: Left

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

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;
            }
        }
    }
}

int main() {
    int n, head, total_head_movement = 0;
    char direction[10];

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

    int requests[n];

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

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

```
printf("Enter direction (Left/Right): ");
scanf("%s", direction);

sort(requests, n);

int index;
for (int i = 0; i < n; i++) {
    if (requests[i] >= head) {
        index = i;
        break;
    }
}

printf("\nSequence of disk access: \n");

if (direction[0] == 'L' || direction[0] == 'l') {
    for (int i = index - 1; i >= 0; i--) {
        printf("%d ", requests[i]);
        total_head_movement += abs(head - requests[i]);
        head = requests[i];
    }
    for (int i = index; i < n; i++) {
        printf("%d ", requests[i]);
        total_head_movement += abs(head - requests[i]);
        head = requests[i];
    }
} else {
    for (int i = index; i < n; i++) {
```

```
            printf("%d ", requests[i]);
            total_head_movement += abs(head - requests[i]);
            head = requests[i];
        }
        for (int i = index - 1; i >= 0; i--) {
            printf("%d ", requests[i]);
            total_head_movement += abs(head - requests[i]);
            head = requests[i];
        }
    }

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

    return 0;
}
```

Slip no 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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_BLOCKS 100
#define MAX_FILES 10

typedef struct {
    char name[20];
    int indexBlock;
    int blocks[MAX_BLOCKS];
    int blockCount;
} File;

int bitVector[MAX_BLOCKS];
File directory[MAX_FILES];
int fileCount = 0;
int n;

void initializeDisk() {
    printf("Enter number of blocks in the disk: ");
    scanf("%d", &n);
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        bitVector[i] = rand() % 2; // Randomly allocate blocks
    }
}

void showBitVector() {
```

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

int findFreeBlock() {
    for (int i = 0; i < n; i++) {
        if (bitVector[i] == 0) return i;
    }
    return -1;
}

void createFile() {
    if (fileCount >= MAX_FILES) {
        printf("\nDirectory is full!\n");
        return;
    }
    File newFile;
    printf("Enter file name: ");
    scanf("%s", newFile.name);
    int indexBlock = findFreeBlock();
    if (indexBlock == -1) {
        printf("\nNo free blocks available for index block!\n");
        return;
    }
    bitVector[indexBlock] = 1;
```

```
    newFile.indexBlock = indexBlock;

    printf("Enter number of blocks for the file: ");
    scanf("%d", &newFile.blockCount);
    if (newFile.blockCount > n - 1) {
        printf("\nNot enough blocks available!\n");
        return;
    }

    printf("Allocating blocks:\n");
    int allocated = 0;
    for (int i = 0; i < n && allocated < newFile.blockCount; i++) {
        if (bitVector[i] == 0) {
            newFile.blocks[allocated++] = i;
            bitVector[i] = 1;
            printf("Block %d allocated\n", i);
        }
    }

    if (allocated < newFile.blockCount) {
        printf("\nNot enough blocks allocated. File creation failed!\n");
        return;
    }
    directory[fileCount++] = newFile;
    printf("File created successfully!\n");
}

void showDirectory() {
```

```
    if (fileCount == 0) {
        printf("\nDirectory is empty!\n");
        return;
    }
    printf("\nFile Directory:\n");
    for (int i = 0; i < fileCount; i++) {
        printf("File: %s, Index Block: %d, Blocks: ", directory[i].name, directory[i].indexBlock);
        for (int j = 0; j < directory[i].blockCount; j++) {
            printf("%d ", directory[i].blocks[j]);
        }
        printf("\n");
    }
}

void deleteFile() {
    char fileName[20];
    printf("Enter file name to delete: ");
    scanf("%s", fileName);
    for (int i = 0; i < fileCount; i++) {
        if (strcmp(directory[i].name, fileName) == 0) {
            bitVector[directory[i].indexBlock] = 0;
            for (int j = 0; j < directory[i].blockCount; j++) {
                bitVector[directory[i].blocks[j]] = 0;
            }
            printf("File %s deleted successfully!\n", fileName);
            for (int k = i; k < fileCount - 1; k++) {
                directory[k] = directory[k + 1];
            }
```

```
            fileCount--;
            return;
        }
    }
    printf("File not found!\n");
}

int main() {
    int choice;
    initializeDisk();
    while (1) {
        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(); break;
            case 2: createFile(); break;
            case 3: showDirectory(); break;
            case 4: deleteFile(); break;
            case 5: exit(0);
            default: printf("Invalid choice!\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. 33, 99, 142, 52, 197, 79, 46, 65

Start Head Position: 72

Direction: Right

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

void scan_disk_scheduling(int requests[], int n, int head, int total_blocks, char direction[]) {
    int seek_count = 0;
    int distance, cur_track;
    int left[100], right[100];
    int l = 0, r = 0, order[100], index = 0;

    // Split requests into left and right based on the head position
    for (int i = 0; i < n; i++) {
        if (requests[i] < head)
            left[l++] = requests[i];
        else
            right[r++] = requests[i];
    }
```

```
// Sort left and right arrays
for (int i = 0; i < l - 1; i++) {
    for (int j = 0; j < l - i - 1; j++) {
        if (left[j] < left[j + 1]) {
            int temp = left[j];
            left[j] = left[j + 1];
            left[j + 1] = temp;
        }
    }
}

for (int i = 0; i < r - 1; i++) {
    for (int j = 0; j < r - i - 1; j++) {
        if (right[j] > right[j + 1]) {
            int temp = right[j];
            right[j] = right[j + 1];
            right[j + 1] = temp;
        }
    }
}

// Move in the direction specified
if (strcmp(direction, "Right") == 0) {
    for (int i = 0; i < r; i++) {
        cur_track = right[i];
        order[index++] = cur_track;
        seek_count += abs(cur_track - head);
```

```
            head = cur_track;
        }
        if (head != total_blocks - 1) {
            seek_count += abs(total_blocks - 1 - head);
            head = total_blocks - 1;
        }
        for (int i = 0; i < l; i++) {
            cur_track = left[i];
            order[index++] = cur_track;
            seek_count += abs(cur_track - head);
            head = cur_track;
        }
    } else {
        for (int i = 0; i < l; i++) {
            cur_track = left[i];
            order[index++] = cur_track;
            seek_count += abs(cur_track - head);
            head = cur_track;
        }
        if (head != 0) {
            seek_count += abs(head - 0);
            head = 0;
        }
        for (int i = 0; i < r; i++) {
            cur_track = right[i];
            order[index++] = cur_track;
            seek_count += abs(cur_track - head);
            head = cur_track;
```

```
        }
    }

    // Display the order of requests served
    printf("Order of requests served: ");
    for (int i = 0; i < index; i++) {
        printf("%d ", order[i]);
    }
    printf("\n");

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

int main() {
    int n, head, total_blocks;
    char direction[10];

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

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

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

```
    }

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

    printf("Enter the direction (Left/Right): ");
    scanf("%s", direction);

    scan_disk_scheduling(requests, n, head, total_blocks, direction);

    return 0;
}
```

Slip no 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.

```
#include <stdio.h>
#define P 6 // Number of processes
#define R 4 // Number of resources

// Function to calculate Need matrix
void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

// Function to check if the system is in a safe state
int isSafe(int processes[], int available[], int max[P][R], int allocation[P][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);

    int work[R];
    for (int i = 0; i < R; i++)
        work[i] = available[i];

    int finish[P] = {0};
    int safeSequence[P];
    int count = 0;

    while (count < P) {
        int found = 0;
```

```
    for (int p = 0; p < P; p++) {
        if (!finish[p]) {
            int canAllocate = 1;
            for (int j = 0; j < R; j++) {
                if (need[p][j] > work[j]) {
                    canAllocate = 0;
                    break;
                }
            }
            if (canAllocate) {
                for (int k = 0; k < R; k++)
                    work[k] += allocation[p][k];
                safeSequence[count++] = p;
                finish[p] = 1;
                found = 1;
            }
        }
    }
    if (!found) {
        printf("System is not in a safe state.\n");
        return 0;
    }
}
printf("System is in a safe state.\nSafe sequence: ");
for (int i = 0; i < P; i++)
    printf("P%d ", safeSequence[i]);
printf("\n");
return 1;
```

```
}

int main() {
    int processes[P] = {0, 1, 2, 3, 4, 5};
    int allocation[P][R] = {{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[P][R] = {{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[R] = {3, 4, 4, 2};
    int need[P][R];

    // Calculate the Need matrix
    calculateNeed(need, max, allocation);
    printf("Need matrix:\n");
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    // Check if system is in a safe state
    isSafe(processes, available, max, allocation);

    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

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

// Function to sort 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;
            }
        }
    }
}

int main() {
    int n, head, total_blocks, total_head_movement = 0;
    char direction[10];

    // Input the total number of disk blocks and requests
    printf("Enter total number of disk blocks: ");
    scanf("%d", &total_blocks);

    printf("Enter the number of disk requests: ");
```

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

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

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

printf("Enter the direction (Left/Right): ");
scanf("%s", direction);

// Sort the requests
sort(requests, n);

int seek_sequence[n + 2];
int index = 0;

if (strcmp(direction, "Right") == 0) {
   // Process the requests to the right
   for (int i = 0; i < n; i++) {
      if (requests[i] >= head) {
         seek_sequence[index++] = requests[i];
      }
   }
   // Go to the end of the disk
```

```
    seek_sequence[index++] = total_blocks - 1;
    // Wrap around to the start
    seek_sequence[index++] = 0;
    // Process the remaining requests on the left
    for (int i = 0; i < n; i++) {
        if (requests[i] < head) {
            seek_sequence[index++] = requests[i];
        }
    }
} else { // If direction is Left
    // Process the requests to the left
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] <= head) {
            seek_sequence[index++] = requests[i];
        }
    }
    // Go to the start of the disk
    seek_sequence[index++] = 0;
    // Wrap around to the end
    seek_sequence[index++] = total_blocks - 1;
    // Process the remaining requests on the right
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] > head) {
            seek_sequence[index++] = requests[i];
        }
    }
}
```

```
    // Calculate total head movement
    printf("\nSeek Sequence: ");
    printf("%d", head);
    for (int i = 0; i < index; i++) {
        printf(" -> %d", seek_sequence[i]);
        total_head_movement += abs(seek_sequence[i] - head);
        head = seek_sequence[i];
    }

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

    return 0;
}
```

Slip no 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

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

void scanDiskScheduling(int requests[], int n, int head, int totalTracks, char direction) {
    int seekCount = 0;
    int distance, curTrack;
```

```
int left[20], right[20];
int l = 0, r = 0;
int sequence[20], seqIndex = 0;

// Add boundary values based on direction
if (direction == 'l' || direction == 'L') left[l++] = 0;
if (direction == 'r' || direction == 'R') right[r++] = totalTracks - 1;

// Separate requests into left and right of the head
for (int i = 0; i < n; i++) {
    if (requests[i] < head)
        left[l++] = requests[i];
    else
        right[r++] = requests[i];
}

// Sort left and right arrays
for (int i = 0; i < l - 1; i++)
    for (int j = 0; j < l - i - 1; j++)
        if (left[j] > left[j + 1]) {
            int temp = left[j];
            left[j] = left[j + 1];
            left[j + 1] = temp;
        }

for (int i = 0; i < r - 1; i++)
    for (int j = 0; j < r - i - 1; j++)
        if (right[j] > right[j + 1]) {
```

```
            int temp = right[j];
            right[j] = right[j + 1];
            right[j + 1] = temp;
        }

// Serve the requests
printf("\nOrder of servicing requests: ");
if (direction == 'r' || direction == 'R') {
    // Move right first
    for (int i = 0; i < r; i++) {
        curTrack = right[i];
        sequence[seqIndex++] = curTrack;
        distance = abs(curTrack - head);
        seekCount += distance;
        head = curTrack;
    }
    // Then move to the left side
    for (int i = l - 1; i >= 0; i--) {
        curTrack = left[i];
        sequence[seqIndex++] = curTrack;
        distance = abs(curTrack - head);
        seekCount += distance;
        head = curTrack;
    }
} else {
    // Move left first
    for (int i = l - 1; i >= 0; i--) {
        curTrack = left[i];
```

```
            sequence[seqIndex++] = curTrack;
            distance = abs(curTrack - head);
            seekCount += distance;
            head = curTrack;
        }
        // Then move to the right side
        for (int i = 0; i < r; i++) {
            curTrack = right[i];
            sequence[seqIndex++] = curTrack;
            distance = abs(curTrack - head);
            seekCount += distance;
            head = curTrack;
        }
    }

    for (int i = 0; i < seqIndex; i++) {
        printf("%d ", sequence[i]);
    }
    printf("\nTotal number of head movements: %d\n", seekCount);
}

int main() {
    int n, head, totalTracks;
    char direction;

    printf("Enter total number of disk tracks: ");
    scanf("%d", &totalTracks);
```

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

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

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

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

    scanDiskScheduling(requests, n, head, totalTracks, 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

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

```
#define SIZE 1000

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

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

    int chunk_size = SIZE / size;
    int local_numbers[chunk_size];

    // Generate numbers only on root process
    if (rank == 0) {
        srand(time(NULL));
        for (int i = 0; i < SIZE; i++) {
            numbers[i] = rand() % 10000; // Random numbers between 0 and 9999
        }
    }

    // Scatter numbers to all processes
    MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process finds the local max
```

```
    local_max = local_numbers[0];
    for (int i = 1; i < chunk_size; i++) {
        if (local_numbers[i] > local_max) {
            local_max = local_numbers[i];
        }
    }

    // Use MPI_Reduce to find the global maximum
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

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

    MPI_Finalize();
    return 0;
}
```

Slip no 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

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

```
#include <stdlib.h>
void fcfs(int requests[], int n, int head) {
    int total_head_movements = 0;

    printf("Disk Requests Order: ");
    for (int i = 0; i < n; i++) {
        total_head_movements += abs(head - requests[i]);
        head = requests[i];
        printf("%d ", head);
    }
    printf("\nTotal head movements: %d\n", total_head_movements);
}
int main() {
    int requests[] = {55, 58, 39, 18, 90, 160, 150, 38, 184};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head;
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    fcfs(requests, n, head);
    return 0;
}
```

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

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

```
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
   int rank, size, n = 1000;
   int numbers[n], local_sum = 0, global_sum = 0;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0) {
      for (int i = 0; i < n; i++) {
         numbers[i] = rand() % 1000;
      }
   }
   MPI_Bcast(numbers, n, MPI_INT, 0, MPI_COMM_WORLD);
   for (int i = rank; i < n; i += size) {
      if (numbers[i] % 2 == 0) {
         local_sum += numbers[i];
      }
   }
   MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if (rank == 0) {
      printf("Sum of all even numbers: %d\n", global_sum);
   }
   MPI_Finalize();
   return 0;
}
```

Slip no 22

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

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
   int rank, size, n = 1000;
   int numbers[n], local_sum = 0, global_sum = 0;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0) {
      for (int i = 0; i < n; i++) {
         numbers[i] = rand() % 1000;
      }
   }
   MPI_Bcast(numbers, n, MPI_INT, 0, MPI_COMM_WORLD);
   for (int i = rank; i < n; i += size) {
      if (numbers[i] % 2 != 0) {
         local_sum += numbers[i];
```

```
    }
  }
  MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank == 0) {
    printf("Sum of all odd numbers: %d\n", global_sum);
  }
  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

```
import random

class SequentialFileAllocation:
  def __init__(self, n):
    self.n = n  # Total number of blocks
    self.bit_vector = [0] * n  # Initialize all blocks as free
    self.files = {}  # Dictionary to store file allocations

    # Randomly allocate some blocks at the start
    allocated_blocks = random.sample(range(n), n // 4)  # Allocating 25% blocks randomly
    for block in allocated_blocks:
      self.bit_vector[block] = 1
```

```
def show_bit_vector(self):
    """Displays the current state of the bit vector."""
    print("\nBit Vector (0: Free, 1: Allocated):")
    print(" ".join(map(str, self.bit_vector)))

def create_file(self, file_name, size):
    """Allocates contiguous blocks for a new file."""
    free_blocks = []
    for i in range(self.n):
        if self.bit_vector[i] == 0:
            free_blocks.append(i)
            if len(free_blocks) == size:
                # Allocate the file in these blocks
                for block in free_blocks:
                    self.bit_vector[block] = 1
                self.files[file_name] = free_blocks
                print(f"File '{file_name}' allocated in blocks: {free_blocks}")
                return
        else:
            free_blocks = []  # Reset if contiguous space is broken
    print("Error: Not enough contiguous free space available!")

def delete_file(self, file_name):
    """Deletes an existing file and marks its blocks as free."""
    if file_name in self.files:
        for block in self.files[file_name]:
            self.bit_vector[block] = 0  # Mark blocks as free
```

```
            print(f"File '{file_name}' deleted successfully.")
            del self.files[file_name]
        else:
            print("Error: File not found!")

    def menu(self):
        """Displays the menu and handles user input."""
        while True:
            print("\nMenu:")
            print("1. Show Bit Vector")
            print("2. Create File")
            print("3. Delete Already Created File")
            print("4. Exit")
            choice = input("Enter your choice: ")

            if choice == "1":
                self.show_bit_vector()
            elif choice == "2":
                file_name = input("Enter file name: ")
                size = int(input("Enter file size (in blocks): "))
                self.create_file(file_name, size)
            elif choice == "3":
                file_name = input("Enter file name to delete: ")
                self.delete_file(file_name)
            elif choice == "4":
                print("Exiting program.")
                break
            else:
```

```
            print("Invalid choice! Please try again.")

# Input for total number of disk blocks
n = int(input("Enter total number of disk blocks: "))
fs = SequentialFileAllocation(n)
fs.menu()
```

Slip no 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

```
#include <stdio.h>
#include <stdlib.h>
void displayMatrix(int matrix[][10], int m, int n) {
   for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
}
void calculateNeedMatrix(int allocation[][10], int max[][10], int need[][10], 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];
        }
    }
}
int canGrantRequest(int request[], int available[], int need[], int n) {
    for (int i = 0; i < n; i++) {
        if (request[i] > need[i]) {
            return 0;
        }
        if (request[i] > available[i]) {
            return 0;
        }
    }
    return 1;
}
int main() {
    int m, n;
    printf("Enter the number of processes: ");
    scanf("%d", &m);
    printf("Enter the number of resource types: ");
    scanf("%d", &n);
    int allocation[m][n], max[m][n], available[n], need[m][n], request[n];
    printf("Enter the Allocation Matrix:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &allocation[i][j]);
        }
```

```
    }
    printf("Enter the Maximum Matrix:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter the Available resources:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &available[i]);
    }
    calculateNeedMatrix(allocation, max, need, m, n);
    printf("Need Matrix:\n");
    displayMatrix(need, m, n);
    printf("Enter the request from process: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &request[i]);
    }
    if (canGrantRequest(request, available, need[0], n)) {
        printf("The request can be granted immediately.\n");
    } else {
        printf("The 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

```
def sstf_disk_scheduling(requests, head):
    total_head_movements = 0
    sequence = []

    while requests:
        # Find the request with the shortest seek time
        closest_request = min(requests, key=lambda x: abs(x - head))
        sequence.append(closest_request)
        total_head_movements += abs(closest_request - head)
        head = closest_request  # Move head to new position
        requests.remove(closest_request)  # Remove from list

    return sequence, total_head_movements

# Input values
requests = [24, 90, 133, 43, 188, 70, 37, 55]
start_head = 58

# Process SSTF Scheduling
served_sequence, total_movements = sstf_disk_scheduling(requests, start_head)

# Output the results
print("Order of request execution:", served_sequence)
print("Total head movements:", total_movements)
```

Slip no 24

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

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

#define ARRAY_SIZE 1000

// Function to check if a number is odd
int is_odd(int num) {
    return num % 2 != 0;
}

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

```
    MPI_Init(&argc, &argv);  // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get process rank
    MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get total processes

    chunk_size = ARRAY_SIZE / size;  // Divide array among processes
    int* sub_array = (int*)malloc(chunk_size * sizeof(int));

    // Process 0 generates the random numbers
    if (rank == 0) {
        srand(time(NULL));
        for (int i = 0; i < ARRAY_SIZE; i++) {
            numbers[i] = rand() % 1000;  // Generate random numbers (0-999)
        }
    }

    // Distribute chunks of array to all processes
    MPI_Scatter(numbers, chunk_size, MPI_INT, sub_array, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process calculates sum of odd numbers in its chunk
    for (int i = 0; i < chunk_size; i++) {
        if (is_odd(sub_array[i])) {
            local_sum += sub_array[i];
        }
    }

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

```
    // Process 0 prints the final result
    if (rank == 0) {
        printf("Sum of all odd numbers: %d\n", global_sum);
    }

    free(sub_array);  // Free dynamically allocated memory
    MPI_Finalize();  // Finalize MPI
    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

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.

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

#define P 5  // Number of processes
```

```
#define R 3  // Number of resources

// Function to calculate the Need Matrix
void calculateNeed(int need[P][R], int max[P][R], int alloc[P][R]) {
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
}

// Function to check if system is in a safe state
bool isSafe(int processes[], int avail[], int max[P][R], int alloc[P][R]) {
    int need[P][R];
    calculateNeed(need, max, alloc);

    bool finish[P] = {false};  // Mark all processes as unfinished
    int safeSeq[P];           // Store the safe sequence
    int work[R];              // Copy of available resources

    // Initialize work as a copy of available resources
    for (int i = 0; i < R; i++)
        work[i] = avail[i];

    int count = 0;
    while (count < P) {
        bool found = false;
```

```
    for (int p = 0; p < P; p++) {
        if (!finish[p]) { // Process is not yet finished
            bool canAllocate = true;

            for (int j = 0; j < R; j++) {
                if (need[p][j] > work[j]) {
                    canAllocate = false;
                    break;
                }
            }

            if (canAllocate) {
                for (int j = 0; j < R; j++)
                    work[j] += alloc[p][j]; // Release resources

                safeSeq[count++] = p;
                finish[p] = true;
                found = true;
            }
        }
    }

    if (!found) {
        printf("\nSystem is in an unsafe state!\n");
        return false;
    }
}
```

```
    // If system is safe, print the safe sequence
    printf("\nSystem is in a safe state.\nSafe sequence: ");
    for (int i = 0; i < P; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n");

    return true;
}

// Main function
int main() {
    int processes[] = {0, 1, 2, 3, 4};

    int alloc[P][R] = { {0, 1, 0}, {2, 0, 0}, {3, 0, 3}, {2, 1, 1}, {0, 0, 2} };
    int max[P][R] = { {0, 0, 0}, {2, 0, 2}, {0, 0, 0}, {1, 0, 0}, {0, 0, 2} };
    int avail[R] = {0, 0, 0}; // Available resources

    // Display Need Matrix
    int need[P][R];
    calculateNeed(need, max, alloc);

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

```
    // Check for safe state
    isSafe(processes, avail, max, alloc);

    return 0;
}
```

Slip no 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

```
#include <stdio.h>
#include <stdlib.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void sortRequests(int requests[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (requests[j] > requests[j + 1]) {
                swap(&requests[j], &requests[j + 1]);
            }
```

```
        }
    }
}
void look(int requests[], int n, int head, int direction) {
    int total_head_movements = 0;
    int served[n];
    for (int i = 0; i < n; i++) {
        served[i] = 0;
    }
    sortRequests(requests, n);
    int start = (direction == 1) ? 0 : n - 1;
    int end = (direction == 1) ? n - 1 : 0;
    int step = (direction == 1) ? 1 : -1;
    printf("Disk Requests Order: ");
    for (int i = start; (direction == 1 && i <= end) || (direction == 0 && i >= end); i += step) {
        total_head_movements += abs(head - requests[i]);
        head = requests[i];
        printf("%d ", head);
    }
    printf("\nTotal head movements: %d\n", total_head_movements);
}
int main() {
    int requests[] = {86, 147, 91, 170, 95, 130, 102, 70};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head, direction;
    printf("Enter the starting head position: ");
    scanf("%d", &head);
    printf("Enter the direction (1 for right, 0 for left): ");
```

```
    scanf("%d", &direction);
    look(requests, n, head, direction);
    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

```
import random

class Block:
    def __init__(self, index):
        self.index = index
        self.next = None

class LinkedFile:
    def __init__(self, name, start_block):
        self.name = name
        self.start_block = start_block

class FileSystem:
    def __init__(self, n):
```

```
    self.n = n
    self.blocks = [None] * n
    self.free_blocks = set(range(n))
    self.files = []
    self.generate_random_allocations()

def generate_random_allocations(self):
    allocated_blocks = random.sample(range(self.n), self.n // 4)
    for block in allocated_blocks:
        self.blocks[block] = Block(block)
        self.free_blocks.discard(block)

def show_bit_vector(self):
    bit_vector = ['1' if block is not None else '0' for block in self.blocks]
    print("Bit Vector:", ''.join(bit_vector))

def create_file(self, filename):
    if not self.free_blocks:
        print("No free blocks available to create a file.")
        return

    start_block_index = self.free_blocks.pop()
    start_block = Block(start_block_index)
    self.blocks[start_block_index] = start_block

    current_block = start_block
    while random.choice([True, False]) and self.free_blocks:
        next_block_index = self.free_blocks.pop()
```

```
            next_block = Block(next_block_index)
            self.blocks[next_block_index] = next_block
            current_block.next = next_block
            current_block = next_block

        new_file = LinkedFile(filename, start_block)
        self.files.append(new_file)
        print(f"File '{filename}' created starting at block {start_block_index}.")

    def show_directory(self):
        if not self.files:
            print("Directory is empty.")
            return

        print("Directory:")
        for file in self.files:
            print(f"File: {file.name}, Start Block: {file.start_block.index}")

    def menu(self):
        while True:
            print("\nMenu:")
            print("1. Show Bit Vector")
            print("2. Create New File")
            print("3. Show Directory")
            print("4. Exit")

            choice = input("Enter your choice: ")
```

```
        if choice == '1':
            self.show_bit_vector()
        elif choice == '2':
            filename = input("Enter file name: ")
            self.create_file(filename)
        elif choice == '3':
            self.show_directory()
        elif choice == '4':
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    n = int(input("Enter number of blocks: "))
    filesystem = FileSystem(n)
    filesystem.menu()
```

Slip no 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.

```
#include <stdio.h>
#define P 5 // Number of processes
#define R 4 // Number of resources

// Function to calculate the Need matrix
void calculateNeed(int need[P][R], int max[P][R], int alloc[P][R]) {
   for (int i = 0; i < P; i++) {
      for (int j = 0; j < R; j++) {
         need[i][j] = max[i][j] - alloc[i][j];
      }
   }
}

// Function to check if the system is in a safe state
int isSafe(int processes[], int avail[], int max[][R], int alloc[][R]) {
   int need[P][R];
   calculateNeed(need, max, alloc);

   int finish[P] = {0};
   int safeSeq[P];
   int work[R];
   for (int i = 0; i < R; i++) work[i] = avail[i];
```

```
int count = 0;
while (count < P) {
    int found = 0;
    for (int p = 0; p < P; p++) {
        if (!finish[p]) {
            int canAllocate = 1;
            for (int j = 0; j < R; j++) {
                if (need[p][j] > work[j]) {
                    canAllocate = 0;
                    break;
                }
            }
            if (canAllocate) {
                for (int j = 0; j < R; j++) work[j] += alloc[p][j];
                safeSeq[count++] = p;
                finish[p] = 1;
                found = 1;
            }
        }
    }
    if (!found) {
        printf("The system is not in a safe state.\n");
        return 0;
    }
}

printf("The system is in a safe state.\nSafe sequence: ");
for (int i = 0; i < P; i++) printf("P%d ", safeSeq[i]);
```

```
    printf("\n");
    return 1;
}

int main() {
    int processes[P] = {0, 1, 2, 3, 4};
    int allocation[P][R] = {{0, 0, 1, 2}, {1, 0, 0, 0}, {1, 3, 5, 4}, {0, 6, 3, 2}, {0, 0, 1, 4}};
    int max[P][R] = {{0, 0, 1, 2}, {1, 7, 5, 0}, {2, 3, 5, 6}, {0, 6, 5, 2}, {0, 6, 5, 6}};
    int available[R] = {1, 5, 2, 0};

    int need[P][R];
    calculateNeed(need, max, allocation);

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

    isSafe(processes, available, max, allocation);
    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

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

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

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

    printf("Enter number of disk requests: ");
    int requests_count;
    scanf("%d", &requests_count);

    int requests[requests_count];

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

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

    printf("\nOrder of disk requests served: \n");
    printf("%d -> ", head);

    for (int i = 0; i < requests_count; i++) {
```

```
        printf("%d", requests[i]);
        if (i < requests_count - 1) printf(" -> ");

        total_head_movements += abs(head - requests[i]);
        head = requests[i];
    }

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

    return 0;
}
```

Slip no 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

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

void look_disk_scheduling(int requests[], int n, int head, int direction) {
    int total_movement = 0;
    int i, j, temp;

    // Sort the request array
    for (i = 0; i < n - 1; i++) {
```

```
    for (j = 0; j < n - i - 1; j++) {
        if (requests[j] > requests[j + 1]) {
            temp = requests[j];
            requests[j] = requests[j + 1];
            requests[j + 1] = temp;
        }
    }
}

printf("Request order: %d ", head);
int start_index = 0;

// Find the first request larger than or equal to head
for (i = 0; i < n; i++) {
    if (requests[i] >= head) {
        start_index = i;
        break;
    }
}

// If direction is right (1), move to the right first
if (direction == 1) {
    for (i = start_index; i < n; i++) {
        printf("-> %d ", requests[i]);
        total_movement += abs(requests[i] - head);
        head = requests[i];
    }
    // Then move to the left
```

```
        for (i = start_index - 1; i >= 0; i--) {
            printf("-> %d ", requests[i]);
            total_movement += abs(requests[i] - head);
            head = requests[i];
        }
    } else {  // Direction is left
        for (i = start_index - 1; i >= 0; i--) {
            printf("-> %d ", requests[i]);
            total_movement += abs(requests[i] - head);
            head = requests[i];
        }
        // Then move to the right
        for (i = start_index; i < n; i++) {
            printf("-> %d ", requests[i]);
            total_movement += abs(requests[i] - head);
            head = requests[i];
        }
    }

    printf("\nTotal head movement: %d\n", total_movement);
}

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

    printf("Enter number of disk requests: ");
    scanf("%d", &n);
    int requests[n];
```

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

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

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

    look_disk_scheduling(requests, n, head, direction);

    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)

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

#define SIZE 1000

int main(int argc, char** argv) {
```

```
    int rank, size;
    int numbers[SIZE];
    int local_min, global_min;

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

    int chunk_size = SIZE / size;
    int local_numbers[chunk_size];

    // Root process generates random numbers
    if (rank == 0) {
        srand(time(NULL));
        for (int i = 0; i < SIZE; i++) {
            numbers[i] = rand() % 10000;  // Random numbers between 0 and 9999
        }
    }

    // Scatter the array to all processes
    MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Find local minimum in each chunk
    local_min = local_numbers[0];
    for (int i = 1; i < chunk_size; i++) {
        if (local_numbers[i] < local_min) {
            local_min = local_numbers[i];
```

```
    }
  }

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

  // Root process prints the result
  if (rank == 0) {
    printf("The minimum number is: %d\n", global_min);
  }

  MPI_Finalize();
  return 0;
}
```

Slip no 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

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

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;
            }
        }
    }
}

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

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

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

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

    printf("Enter direction (0 for lower to higher, 1 for higher to lower): ");
    scanf("%d", &direction);

    // Sort the request array
```

```
sort(requests, n);

int total_movement = 0;
int index = 0;
while (index < n && requests[index] < head) {
    index++;
}

printf("\nOrder of servicing requests: ");

if (direction == 0) { // Moving towards higher numbers
    for (int i = index; i < n; i++) {
        printf("%d ", requests[i]);
        total_movement += abs(head - requests[i]);
        head = requests[i];
    }
    for (int i = 0; i < index; i++) {
        printf("%d ", requests[i]);
        total_movement += abs(head - requests[i]);
        head = requests[i];
    }
} else { // Moving towards lower numbers
    for (int i = index - 1; i >= 0; i--) {
        printf("%d ", requests[i]);
        total_movement += abs(head - requests[i]);
        head = requests[i];
    }
    for (int i = n - 1; i >= index; i--) {
```

```
            printf("%d ", requests[i]);
            total_movement += abs(head - requests[i]);
            head = requests[i];
        }
    }

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

    return 0;
}
```

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

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

#define ARRAY_SIZE 1000

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

    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    chunk_size = ARRAY_SIZE / size;
    int local_numbers[chunk_size];

    // Generate random numbers on root process
    if (rank == 0) {
        srand(42);  // Fixed seed for reproducibility
        for (int i = 0; i < ARRAY_SIZE; i++) {
            numbers[i] = rand() % 100;
        }
    }

    // Scatter the array to all processes
    MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process computes its local sum
    for (int i = 0; i < chunk_size; i++) {
        local_sum += local_numbers[i];
    }

    // Gather all local sums to the root process
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print the final result from the root process
    if (rank == 0) {
```

```
    printf("Total sum of 1000 random numbers: %d\n", total_sum);
  }

  MPI_Finalize();
  return 0;   }
```

Slip no 29

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

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

#define SIZE 1000

int main(int argc, char* argv[]) {
  int rank, size;
  int numbers[SIZE];
  int local_sum = 0, total_sum = 0;
  int chunk_size;

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

```
    chunk_size = SIZE / size;

    // Master process generates random numbers
    if (rank == 0) {
        srand(time(NULL));
        for (int i = 0; i < SIZE; i++) {
            numbers[i] = rand() % 1000;
        }
    }

    // Scatter the numbers to all processes
    int* local_numbers = (int*)malloc(chunk_size * sizeof(int));
    MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process calculates the sum of even numbers in its chunk
    for (int i = 0; i < chunk_size; i++) {
        if (local_numbers[i] % 2 == 0) {
            local_sum += local_numbers[i];
        }
    }

    // Gather all local sums to the master process
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Master process prints the total sum
    if (rank == 0) {
        printf("Total sum of even numbers: %d\n", total_sum);
```

```
    }

    // Clean up
    free(local_numbers);
    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 Posi

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

void sort_requests(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;
            }
        }
    }
}
```

```
int main() {
    int n, head, total_movement = 0;

    // Get number of disk blocks and requests
    printf("Enter number of requests: ");
    scanf("%d", &n);

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

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

    // Sort the request array
    sort_requests(requests, n);

    // Find the point where head is greater than the request
    int i;
    for (i = 0; i < n; i++) {
        if (requests[i] > head) {
            break;
        }
    }
```

```
    printf("\nOrder of serving requests: ");

    // Serve right side first
    for (int j = i; j < n; j++) {
        printf("%d ", requests[j]);
        total_movement += abs(head - requests[j]);
        head = requests[j];
    }

    // Then wrap around to the beginning
    for (int j = 0; j < i; j++) {
        printf("%d ", requests[j]);
        total_movement += abs(head - requests[j]);
        head = requests[j];
    }

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

    return 0;
}
```

Slip no 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)

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

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

#define ARRAY_SIZE 1000

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

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

    int chunk_size = ARRAY_SIZE / size;
    int local_numbers[chunk_size];

    // Master process generates the random numbers
    if (rank == 0) {
        srand(time(NULL));
        printf("Generated numbers: ");
        for (int i = 0; i < ARRAY_SIZE; i++) {
            numbers[i] = rand() % 1000;
            printf("%d ", numbers[i]);
        }
        printf("\n");
    }
```

```
    // Scatter the array to all processes
    MPI_Scatter(numbers, chunk_size, MPI_INT, local_numbers, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process finds its local minimum
    local_min = local_numbers[0];
    for (int i = 1; i < chunk_size; i++) {
        if (local_numbers[i] < local_min) {
            local_min = local_numbers[i];
        }
    }

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

    // Master process prints the global minimum
    if (rank == 0) {
        printf("Minimum number found: %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

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

int main() {
    int n, head, total_movement = 0;

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

    int requests[n];

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

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

    printf("\nOrder of servicing requests: \n%d", head);

    // Process each request in order
    for (int i = 0; i < n; i++) {
```

```
        printf(" -> %d", requests[i]);
        total_movement += abs(head - requests[i]);
        head = requests[i];
    }

    // Display the total head movement
    printf("\n\nTotal head movement: %d cylinders\n", total_movement);

    return 0;
}
```