Slip 1

Slip1 Q.1) Write a C Menu driven Program to implement following functionality

a) Accept Available

b) Display Allocation, Max

c) Display the contents of need matrix

d) Display Available

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	2	3	2	9	7	5	3	3	2
P1	4	0	0	5	2	2			
P2	5	0	4	1	0	4			
P3	4	3	3	4	4	4			
P4	2	2	4	6	5	5			

```
#include<stdio.h>
// Function prototypes
void acceptAvailable(int available[], int n);
void displayAllocationMax(int allocation[][3], int max[][3], int n);
void displayNeed(int allocation[][3], int max[][3], int need[][3], int n);
void displayAvailable(int available[], int n);

int main()
{
   int allocation[5][3] = {{2, 3, 2}, {4, 0, 0}, {5, 0, 4}, {4, 3, 3}, {2, 2, 4}};
   int max[5][3] = {{9, 7, 5}, {5, 2, 2}, {1, 0, 4}, {4, 4, 4}, {6, 5, 5}};
   int available[3];
   int need[5][3];
   int choice, n = 5;

   // Calculate the need matrix
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < 3; j++) {
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }
   Do
 {
      printf("\n\n***** Menu *****\n");
      printf("1. Accept Available\n");
      printf("2. Display Allocation, Max\n");
      printf("3. Display the contents of need matrix\n");
      printf("4. Display Available\n");
      printf("5. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            acceptAvailable(available, 3);
```

```
            break;
        case 2:
            displayAllocationMax(allocation, max, n);
            break;
        case 3:
            displayNeed(allocation, max, need, n);
            break;
        case 4:
            displayAvailable(available, 3);
            break;
        case 5:
            printf("Exiting...");
            break;
        default:
            printf("Invalid choice! Please enter a number between 1 and 5.");
    }
  } while (choice != 5);

  return 0;
}

void acceptAvailable(int available[], int n) {
  printf("Enter the available resources for A, B, and C respectively: ");
  for (int i = 0; i < n; i++) {
    scanf("%d", &available[i]);
  }
}

void displayAllocationMax(int allocation[][3], int max[][3], int n) {
  printf("Process\tAllocation\tMax\n");
  for (int i = 0; i < n; i++) {
    printf("P%d\t", i);
    for (int j = 0; j < 3; j++) {
      printf("%d ", allocation[i][j]);
    }
    printf("\t\t");
    for (int j = 0; j < 3; j++) {
      printf("%d ", max[i][j]);
    }
    printf("\n");
  }
}

void displayNeed(int allocation[][3], int max[][3], int need[][3], int n) {
  printf("Process\tNeed\n");
  for (int i = 0; i < n; i++) {
    printf("P%d\t", i);
    for (int j = 0; j < 3; j++) {
      need[i][j] = max[i][j] - allocation[i][j];
      printf("%d ", need[i][j]);
    }
    printf("\n");
```

```
    }
}

void displayAvailable(int available[], int n) {
    printf("Available Resources: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", available[i]);
    }
}
```

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

[15]

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

// Function to calculate total head movements
int calculateHeadMovements(int requestQueue[], int n, int headPosition)
{
    int totalHeadMovements = 0;
    int currentHeadPosition = headPosition;

    // Loop through the request queue and calculate head movements
    for (int i = 0; i < n; i++) {
        totalHeadMovements += abs(requestQueue[i] - currentHeadPosition);
        currentHeadPosition = requestQueue[i];
    }

    return totalHeadMovements;
}

int main() {
    int n, headPosition;

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

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

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

```
    // Displaying the request order
    printf("Request Order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", requestQueue[i]);
    }
    printf("\n");

    printf("Start Head Position: %d\n", headPosition);

    // Calculating total head movements
    int totalHeadMovements = calculateHeadMovements(requestQueue, n, headPosition);
    printf("Total number of head movements: %d\n", totalHeadMovements);

    return 0;
}
```

Slip2

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_BLOCKS 100

// Node structure for the linked list
struct Node {
    int block;
    struct Node* next;
};

int disk[MAX_BLOCKS] = {0}; // 0 represents free block, 1 represents allocated block
struct Node* freeList = NULL; // Linked list to store the free blocks

void initializeFreeList(int n) {
    for (int i = n - 1; i >= 0; i--) {
        if (disk[i] == 0) {
            struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
            newNode->block = i;
            newNode->next = freeList;
```

```
            freeList = newNode;
        }
    }
}

void showBitVector(int n) {
    printf("\nBit Vector (Disk Allocation):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

void createNewFile(int n) {
    int size;
    printf("\nEnter the size of the file: ");
    scanf("%d", &size);

    if (freeList == NULL) {
        printf("\nNo free blocks available to create the file.\n");
        return;
    }

    struct Node* current = freeList;
    // Allocate the blocks for the file
    for (int i = 0; i < size; i++) {
        disk[current->block] = 1;
        struct Node* temp = current;
        current = current->next;
        free(temp);
    }
    freeList = current;

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

void showDirectory(int n) {
    printf("\nDirectory Listing:\n");
    for (int i = 0; i < n; i++) {
        if (disk[i] == 1) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

int main() {
    int n;

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

```
    srand(time(NULL)); // Seed for random block allocation

    // Randomly mark some blocks as allocated
    for (int i = 0; i < n; i++) {
        if (rand() % 2 == 1) {
            disk[i] = 1;
        }
    }

    initializeFreeList(n);

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

        switch (choice) {
            case 1:
                showBitVector(n);
                break;

            case 2:
                createNewFile(n);
                break;

            case 3:
                showDirectory(n);
                break;

            case 4:
                printf("\nExiting the program.\n");
                break;

            default:
                printf("\nInvalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

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

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

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```

Slip 3

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

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

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

[15]

```
#include <stdio.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

int available[MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
int finish[MAX_PROCESSES] = {0};

int isSafeState(int processes[], int n) {
    int work[MAX_RESOURCES];
    int i, j, finished = 0, safeSequence[MAX_PROCESSES], count = 0;

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

    while (finished < n) {
        int found = 0;
        for (i = 0; i < n; i++) {
            if (!finish[i]) {
                int safe = 1;
                for (j = 0; j < MAX_RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        safe = 0;
                        break;
                    }
                }
                if (safe) {
```

```
                for (j = 0; j < MAX_RESOURCES; j++) {
                    work[j] += allocation[i][j];
                }
                finish[i] = 1;
                safeSequence[count++] = i;
                found = 1;
                finished++;
            }
        }
    }
    if (!found) {
        return 0; // System is not in a safe state
    }
  }

  printf("Safe sequence: ");
  for (i = 0; i < n; i++) {
    printf("P%d ", safeSequence[i]);
  }
  printf("\n");

  return 1; // System is in a safe state
}

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

int main() {
  // Initialize the allocation, max, and available arrays
  int i, j;
  int processes[MAX_PROCESSES] = {0, 1, 2, 3, 4};

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

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

```
        {0, 6, 5, 6}
    };

    int available[MAX_RESOURCES] = {1, 5, 2, 0};

    calculateNeedMatrix();

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

    if (isSafeState(processes, MAX_PROCESSES)) {
        printf("System is in safe state.\n");
    } else {
        printf("System is not in safe state.\n");
    }

    return 0;
}
```

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

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

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }
```

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```

Slip 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

[15]

```
#include <stdio.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

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

```
int available[MAX_RESOURCES] = {7, 2, 6};

// Function prototypes
void acceptAvailable();
void acceptAllocationMax();
void calculateNeed();
void displayAllocationMax();
void displayNeed();
void displayAvailable();

int main() {
   int choice;

   do {
      printf("\n***** Menu *****\n");
      printf("a) Accept Available\n");
      printf("b) Display Allocation, Max\n");
      printf("c) Find Need and display It\n");
      printf("d) Display Available\n");
      printf("e) Exit\n");
      printf("Enter your choice: ");
      scanf(" %c", &choice);

      switch (choice) {
         case 'a':
            acceptAvailable();
            break;
         case 'b':
            acceptAllocationMax();
            break;
         case 'c':
            calculateNeed();
            displayNeed();
            break;
         case 'd':
            displayAvailable();
            break;
         case 'e':
            printf("Exiting...\n");
            break;
         default:
            printf("Invalid choice! Please enter a valid option.\n");
      }
   } while (choice != 'e');

   return 0;
}

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

```
}

// Function to accept Allocation and Max matrices from the user
void acceptAllocationMax() {
    printf("Enter Allocation Matrix (P0 to P4, A to C):\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    printf("Enter Max Matrix (P0 to P4, A to C):\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d: ", i);
        for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &max[i][j]);
        }
    }
}

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

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

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

// Function to display the Need matrix
```

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

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

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>

// Function to perform SCAN disk scheduling
void scanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
    int totalHeadMovements = 0;
    int i, j, temp, currentHead, index;

    // Sort the disk queue in ascending order
    for (i = 0; i < diskSize - 1; i++) {
        for (j = 0; j < diskSize - i - 1; j++) {
            if (diskQueue[j] > diskQueue[j + 1]) {
                // Swap elements if they are in the wrong order
                temp = diskQueue[j];
                diskQueue[j] = diskQueue[j + 1];
                diskQueue[j + 1] = temp;
            }
        }
    }

    // Find index of current head in the sorted disk queue
    for (i = 0; i < diskSize; i++) {
        if (diskQueue[i] >= startHead) {
            index = i;
            break;
```

```
        }
    }

    printf("Order of disk request serving:\n");

    // SCAN algorithm
    if (direction == 'L') {
        // Move left
        for (i = index; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the beginning
        printf("0 ");
        totalHeadMovements += startHead;

        // Move right
        for (i = index + 1; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    } else if (direction == 'R') {
        // Move right
        for (i = index; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the end
        printf("199 ");
        totalHeadMovements += 199 - startHead;

        // Move left
        for (i = index - 1; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

int main() {
    int diskSize, startHead, i;
    char direction;

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

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

    int diskQueue[diskSize];

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);

    scanDisk(diskQueue, diskSize, startHead, direction);

    return 0;
}
```

Slip 5

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

```
#include <stdio.h>
#include <stdlib.h>
// Function to display a matrix
void display_matrix(int **matrix, int m, int n, char *title) {
    printf("%s\n", title);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

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

// Function to check if request can be granted
int check_request(int process, int *request, int **need, int *available, int n) {
```

```
    for (int i = 0; i < n; ++i) {
        if (request[i] > need[process][i] || 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 *available = (int *)malloc(n * sizeof(int));
    printf("Enter the number of instances for each resource type:\n");
    for (int i = 0; i < n; ++i) {
        printf("Resource %d: ", i + 1);
        scanf("%d", &available[i]);
    }

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

    int **maximum = (int **)malloc(m * sizeof(int *));
    printf("Enter the maximum requirement matrix:\n");
    for (int i = 0; i < m; ++i) {
        maximum[i] = (int *)malloc(n * sizeof(int));
        for (int j = 0; j < n; ++j) {
            scanf("%d", &maximum[i][j]);
        }
    }

    int **need = (int **)malloc(m * sizeof(int *));
    for (int i = 0; i < m; ++i) {
        need[i] = (int *)malloc(n * sizeof(int));
    }

    calculate_need_matrix(allocation, maximum, need, m, n);
    display_matrix(need, m, n, "Need matrix:");

    int process;
    printf("Enter the process number making the request: ");
```

```
    scanf("%d", &process);
    process--; // Adjusting to 0-based indexing

    int *request = (int *)malloc(n * sizeof(int));
    printf("Enter the request for each resource type: ");
    for (int i = 0; i < n; ++i) {
        scanf("%d", &request[i]);
    }

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

    // Freeing dynamically allocated memory
    free(available);
    for (int i = 0; i < m; ++i) {
        free(allocation[i]);
        free(maximum[i]);
        free(need[i]);
    }
    free(allocation);
    free(maximum);
    free(need);
    free(request);

    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>

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);
```

```
    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

    // Find local max
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        if (local_numbers[i] > local_max) {
            local_max = local_numbers[i];
        }
    }

    // Find the global max using MPI_Reduce
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local max on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_max);
        }

        printf("\nGlobal max: %d\n", global_max);
    }

    MPI_Finalize();

    return 0;
}
```

Slip 6

Q.1 Write a program to simulate Linked file allocation method. Assume disk with n number of blocks. Give value of n as input. Randomly mark some block as allocated and accordingly maintain the list of free blocks Write menu driver program with menu options as mentioned below and implement each option.

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

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

#define MAX_BLOCKS 100
```

```
// Node structure for the linked list
struct Node {
    int block;
    struct Node* next;
};

int disk[MAX_BLOCKS] = {0}; // 0 represents free block, 1 represents allocated block
struct Node* freeList = NULL; // Linked list to store the free blocks

void initializeFreeList(int n) {
    for (int i = n - 1; i >= 0; i--) {
        if (disk[i] == 0) {
            struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
            newNode->block = i;
            newNode->next = freeList;
            freeList = newNode;
        }
    }
}

void showBitVector(int n) {
    printf("\nBit Vector (Disk Allocation):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

void createNewFile(int n) {
    int size;
    printf("\nEnter the size of the file: ");
    scanf("%d", &size);

    if (freeList == NULL) {
        printf("\nNo free blocks available to create the file.\n");
        return;
    }

    struct Node* current = freeList;
    // Allocate the blocks for the file
    for (int i = 0; i < size; i++) {
        disk[current->block] = 1;
        struct Node* temp = current;
        current = current->next;
        free(temp);
    }
    freeList = current;

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

void showDirectory(int n) {
```

```
    printf("\nDirectory Listing:\n");
    for (int i = 0; i < n; i++) {
        if (disk[i] == 1) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

int main() {
    int n;

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

    srand(time(NULL)); // Seed for random block allocation

    // Randomly mark some blocks as allocated
    for (int i = 0; i < n; i++) {
        if (rand() % 2 == 1) {
            disk[i] = 1;
        }
    }

    initializeFreeList(n);

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

        switch (choice) {
            case 1:
                showBitVector(n);
                break;

            case 2:
                createNewFile(n);
                break;

            case 3:
                showDirectory(n);
                break;

            case 4:
                printf("\nExiting the program.\n");
```

```
            break;

        default:
            printf("\nInvalid choice. Please enter a valid option.\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>

// Function to perform C-SCAN disk scheduling
void cScanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
    int totalHeadMovements = 0;
    int i, j, temp, currentHead, index;

    // Sort the disk queue in ascending order
    for (i = 0; i < diskSize - 1; i++) {
        for (j = 0; j < diskSize - i - 1; j++) {
            if (diskQueue[j] > diskQueue[j + 1]) {
                // Swap elements if they are in the wrong order
                temp = diskQueue[j];
                diskQueue[j] = diskQueue[j + 1];
                diskQueue[j + 1] = temp;
            }
        }
    }

    // Find index of current head in the sorted disk queue
    for (i = 0; i < diskSize; i++) {
        if (diskQueue[i] >= startHead) {
            index = i;
            break;
        }
    }
```

```
    printf("Order of disk request serving:\n");

    // C-SCAN algorithm
    if (direction == 'R') {
        // Move right
        for (i = index; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the end
        printf("199 ");
        totalHeadMovements += 199 - startHead;

        // Move left
        for (i = 0; i < index; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    } else if (direction == 'L') {
        // Move left
        for (i = index; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the beginning
        printf("0 ");
        totalHeadMovements += startHead;

        // Move right
        for (i = diskSize - 1; i > index; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

```
int main() {
    int diskSize, startHead, i;
    char direction;

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

    int diskQueue[diskSize];

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);

    cScanDisk(diskQueue, diskSize, startHead, direction);

    return 0;
}
```

Slip7

Q.1 Consider the following snapshot of the system.

Process	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 [15]

```
#include<stdio.h>

int main() {
    int processes = 5; // Number of processes
    int resources = 4; // Number of resources

    int allocation[5][4] = { {2, 0, 0, 1},   // Allocation Matrix
```

```
                    {3, 1, 2, 1},
                    {2, 1, 0, 3},
                    {1, 3, 1, 2},
                    {1, 4, 3, 2} };

int max[5][4] = { {4, 2, 1, 2},         // Max Matrix
                  {5, 2, 5, 2},
                  {2, 3, 1, 6},
                  {1, 4, 2, 4},
                  {3, 6, 6, 5} };

int available[4] = {3, 3, 2, 1};        // Available Resources

int need[5][4];                         // Need Matrix
int i, j, k;
int finish[5] = {0};                    // To mark whether the process has completed or not

// Calculating the Need matrix
for (i = 0; i < processes; i++) {
   for (j = 0; j < resources; j++) {
      need[i][j] = max[i][j] - allocation[i][j];
   }
}

int work[4];
for (i = 0; i < resources; i++) {
   work[i] = available[i];
}

// Safety Algorithm
int safeSeq[5];
int count = 0;
while (count < processes) {
   int found = 0;
   for (i = 0; i < processes; i++) {
      if (finish[i] == 0) {
         int canExecute = 1;
         for (j = 0; j < resources; j++) {
            if (need[i][j] > work[j]) {
               canExecute = 0;
               break;
            }
         }
         if (canExecute) {
```

```
                for (k = 0; k < resources; k++) {
                    work[k] += allocation[i][k];
                }
                safeSeq[count++] = i;
                finish[i] = 1;
                found = 1;
            }
        }
    }
    if (!found) {
        printf("System is not in a safe state.\n");
        return -1;
    }
}

printf("System is in a safe state.\nSafe sequence is: ");
for (i = 0; i < processes - 1; i++) {
    printf("P%d -> ", safeSeq[i]);
}
printf("P%d\n", safeSeq[processes - 1]);

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>

// Function to perform C-SCAN disk scheduling
void cScanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
    int totalHeadMovements = 0;
    int i, j, temp, currentHead, index;

    // Sort the disk queue in ascending order
    for (i = 0; i < diskSize - 1; i++) {
        for (j = 0; j < diskSize - i - 1; j++) {
            if (diskQueue[j] > diskQueue[j + 1]) {
```

```
            // Swap elements if they are in the wrong order
            temp = diskQueue[j];
            diskQueue[j] = diskQueue[j + 1];
            diskQueue[j + 1] = temp;
        }
    }
}

// Find index of current head in the sorted disk queue
for (i = 0; i < diskSize; i++) {
    if (diskQueue[i] >= startHead) {
        index = i;
        break;
    }
}

printf("Order of disk request serving:\n");

// C-SCAN algorithm
if (direction == 'R') {
    // Move right
    for (i = index; i < diskSize; i++) {
        printf("%d ", diskQueue[i]);
        totalHeadMovements += abs(startHead - diskQueue[i]);
        startHead = diskQueue[i];
    }

    // Move to the end
    printf("199 ");
    totalHeadMovements += 199 - startHead;

    // Move left
    for (i = 0; i < index; i++) {
        printf("%d ", diskQueue[i]);
        totalHeadMovements += abs(startHead - diskQueue[i]);
        startHead = diskQueue[i];
    }
} else if (direction == 'L') {
    // Move left
    for (i = index; i >= 0; i--) {
        printf("%d ", diskQueue[i]);
        totalHeadMovements += abs(startHead - diskQueue[i]);
        startHead = diskQueue[i];
    }
```

```
        // Move to the beginning
        printf("0 ");
        totalHeadMovements += startHead;

        // Move right
        for (i = diskSize - 1; i > index; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

int main() {
    int diskSize, startHead, i;
    char direction;

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

    int diskQueue[diskSize];

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);

    cScanDisk(diskQueue, diskSize, startHead, direction);

    return 0;
}
```

Slip8

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>
#include <time.h>

#define MAX_BLOCKS 100

// Node structure for the linked list
struct Node {
    int block;
    struct Node* next;
};

int disk[MAX_BLOCKS] = {0}; // 0 represents free block, 1 represents allocated block
struct Node* freeList = NULL; // Linked list to store the free blocks

void initializeFreeList(int n) {
    for (int i = n - 1; i >= 0; i--) {
        if (disk[i] == 0) {
            struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
            newNode->block = i;
            newNode->next = freeList;
            freeList = newNode;
        }
    }
}

void showBitVector(int n) {
    printf("\nBit Vector (Disk Allocation):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

void createNewFile(int n) {
    int size;
    printf("\nEnter the size of the file: ");
    scanf("%d", &size);

    if (freeList == NULL) {
```

```
        printf("\nNo free blocks available to create the file.\n");
        return;
    }

    struct Node* current = freeList;
    // Allocate the blocks for the file
    for (int i = 0; i < size; i++) {
        disk[current->block] = 1;
        struct Node* temp = current;
        current = current->next;
        free(temp);
    }
    freeList = current;

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

void showDirectory(int n) {
    printf("\nDirectory Listing:\n");
    for (int i = 0; i < n; i++) {
        if (disk[i] == 1) {
            printf("Block %d: Allocated\n", i);
        } else {
            printf("Block %d: Free\n", i);
        }
    }
}

int main() {
    int n;

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

    srand(time(NULL)); // Seed for random block allocation

    // Randomly mark some blocks as allocated
    for (int i = 0; i < n; i++) {
        if (rand() % 2 == 1) {
            disk[i] = 1;
        }
    }

    initializeFreeList(n);

    int choice;
    do {
        printf("\nMenu:\n");
        printf("1. Show Bit Vector\n");
        printf("2. Create New File\n");
        printf("3. Show Directory\n");
        printf("4. Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                showBitVector(n);
                break;

            case 2:
                createNewFile(n);
                break;

            case 3:
                showDirectory(n);
                break;

            case 4:
                printf("\nExiting the program.\n");
                break;

            default:
                printf("\nInvalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using SSTF algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.
186, 89, 44, 70, 102, 22, 51, 124
Start Head Position: 70

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

// Function to perform SSTF disk scheduling
void sstfDisk(int diskQueue[], int diskSize, int startHead) {
    int totalHeadMovements = 0;
    int i, j, minDistance, minIndex;

    printf("Order of disk request serving:\n");

    for (i = 0; i < diskSize; i++) {
        minDistance = abs(startHead - diskQueue[0]);
        minIndex = 0;

        // Find the request with the shortest seek time
        for (j = 1; j < diskSize; j++) {
```

```
            int distance = abs(startHead - diskQueue[j]);
            if (distance < minDistance) {
                minDistance = distance;
                minIndex = j;
            }
        }

        // Serve the request and update head position
        printf("%d ", diskQueue[minIndex]);
        totalHeadMovements += minDistance;
        startHead = diskQueue[minIndex];

        // Mark the served request as -1 to avoid serving it again
        diskQueue[minIndex] = -1;
    }

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

int main() {
    int diskSize, startHead, i;

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

    int diskQueue[diskSize];

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

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

    sstfDisk(diskQueue, diskSize, startHead);

    return 0;
}
```

Slip 9

Q.1.Consider the following snapshot of system, A, B, C, 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				

Using Resource –Request algorithm to Check whether the current system is in safe state or not . [15]

```
#include <stdio.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

int available[MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
int finish[MAX_PROCESSES] = {0};

int isSafeState(int processes[], int n) {
    int work[MAX_RESOURCES];
    int i, j, finished = 0, safeSequence[MAX_PROCESSES], count = 0;

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

    while (finished < n) {
        int found = 0;
        for (i = 0; i < n; i++) {
            if (!finish[i]) {
                int safe = 1;
                for (j = 0; j < MAX_RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        safe = 0;
                        break;
                    }
                }
                if (safe) {
                    for (j = 0; j < MAX_RESOURCES; j++) {
                        work[j] += allocation[i][j];
                    }
                    finish[i] = 1;
                    safeSequence[count++] = i;
```

```
                found = 1;
                finished++;
            }
        }
    }
    if (!found) {
        return 0; // System is not in a safe state
    }
}

printf("Safe sequence: ");
for (i = 0; i < n; i++) {
    printf("P%d ", safeSequence[i]);
}
printf("\n");

return 1; // System is in a safe state
}

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

int main() {
    // Initialize the allocation, max, and available arrays
    int i, j;
    int processes[MAX_PROCESSES] = {0, 1, 2, 3, 4};

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

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

    int available[MAX_RESOURCES] = {1, 5, 2, 0};
```

```
    calculateNeedMatrix();

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

    if (isSafeState(processes, MAX_PROCESSES)) {
        printf("System is in safe state.\n");
    } else {
        printf("System is not in safe state.\n");
    }

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using LOOK algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.
176, 79, 34, 60, 92, 11, 41, 114
Starting Head Position: 65
Direction: Left

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

void look_algorithm(int requests[], int n, int head_position) {
    int total_head_movements = 0;
    int current_position = head_position;

    printf("Order of service: ");
    while (1) {
        int next_request = -1;
        int min_diff = __INT_MAX__;

        // Find the next request to serve
        for (int i = 0; i < n; i++) {
            if (requests[i] != -1) {
                int diff = abs(requests[i] - current_position);
                if (diff < min_diff) {
                    min_diff = diff;
                    next_request = requests[i];
```

```
            }
        }
    }

    // If no request found, break the loop
    if (next_request == -1) break;

    // Serve the request
    printf("%d ", next_request);
    total_head_movements += abs(next_request - current_position);
    current_position = next_request;
    requests[next_request] = -1; // Mark request as served
  }

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

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

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

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

  printf("Order of service: ");
  look_algorithm(requests, n, head_position);

  return 0;
}
```

Slip 10

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

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

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }

    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>

// Function to perform SCAN disk scheduling
void scanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
    int totalHeadMovements = 0;
    int i, j, temp, currentHead, index;

    // Sort the disk queue in ascending order
    for (i = 0; i < diskSize - 1; i++) {
        for (j = 0; j < diskSize - i - 1; j++) {
            if (diskQueue[j] > diskQueue[j + 1]) {
                // Swap elements if they are in the wrong order
                temp = diskQueue[j];
                diskQueue[j] = diskQueue[j + 1];
                diskQueue[j + 1] = temp;
            }
        }
    }

    // Find index of current head in the sorted disk queue
    for (i = 0; i < diskSize; i++) {
        if (diskQueue[i] >= startHead) {
            index = i;
            break;
        }
    }

    printf("Order of disk request serving:\n");

    // SCAN algorithm
    if (direction == 'L') {
        // Move left
        for (i = index; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the beginning
        printf("0 ");
        totalHeadMovements += startHead;
```

```
        // Move right
        for (i = index + 1; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    } else if (direction == 'R') {
        // Move right
        for (i = index; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the end
        printf("199 ");
        totalHeadMovements += 199 - startHead;

        // Move left
        for (i = index - 1; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

int main() {
    int diskSize, startHead, i;
    char direction;

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

    int diskQueue[diskSize];

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);

    scanDisk(diskQueue, diskSize, startHead, direction);
```

```
    return 0;
}
```

Slip 11

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

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Implement the following Menu.

a) Accept Available
b) Display Allocation, Max
c) Display the contents of need matrix
d) Display Available [15]

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

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

// Function to check if the current process can be executed
bool can_execute(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (need[process_id][i] > available[i]) {
            return false;
        }
    }
    return true;
}

// Function to execute the process and update available resources
void execute_process(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        available[i] += allocation[process_id][i];
        allocation[process_id][i] = 0;
        need[process_id][i] = 0;
    }
}

// Function to accept available resources
void accept_available() {
    printf("Enter available resources:\n");
    for (int i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &available[i]);
    }
}
```

```
// Function to display allocation and max matrices
void display_allocation_max() {
    printf("Allocation Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", allocation[i][j]);
        }
        printf("\n");
    }
    printf("\nMax Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\n");
    }
}

// Function to calculate and display the need matrix
void display_need() {
    printf("Need Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}

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

int main() {
    // Initialize allocation and max matrices based on the snapshot
    int allocation_snapshot[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 3},
        {2, 1, 1},
        {0, 0, 2}
    };
    int max_snapshot[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 0, 0},
        {2, 0, 2},
        {0, 0, 0},
        {1, 0, 0},
        {0, 0, 2}
    };
```

```
    // Copy snapshot to allocation and max matrices
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            allocation[i][j] = allocation_snapshot[i][j];
            max[i][j] = max_snapshot[i][j];
        }
    }

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

        switch (choice) {
            case 'a':
                accept_available();
                break;
            case 'b':
                display_allocation_max();
                break;
            case 'c':
                display_need();
                break;
            case 'd':
                display_available();
                break;
            case 'e':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 'e');

    return 0;
}
```

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

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

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
```

```
    int i;
    int local_max = 0, global_max = 0;

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

    // Find local max
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        if (local_numbers[i] > local_max) {
            local_max = local_numbers[i];
        }
    }

    // Find the global max using MPI_Reduce
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local max on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_max);
        }

        printf("\nGlobal max: %d\n", global_max);
    }

    MPI_Finalize();

    return 0;
}
```

Slip 12

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 ARRAY_SIZE 1000
```

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-LOOK algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.
23, 89, 132, 42, 187, 69, 36, 55
Start Head Position: 40
Direction: Right

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

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

// Function to simulate C-LOOK disk scheduling algorithm
void c_look_algorithm(int requests[], int n, int head_position) {
    int total_head_movements = 0;
    int serving_order[n];
    int serving_index = 0;

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

    // Serve requests to the right of head position
    for (int i = 0; i < n; i++) {
        if (requests[i] >= head_position) {
            total_head_movements += abs(head_position - requests[i]);
            head_position = requests[i];
            serving_order[serving_index++] = head_position;
        }
    }

    // Move head to the beginning and serve requests to the right again
    total_head_movements += abs(head_position - requests[0]);
    head_position = requests[0];
    serving_order[serving_index++] = head_position;

    // Print serving order
    printf("Order of service: ");
    for (int i = 0; i < serving_index; i++) {
        printf("%d ", serving_order[i]);
    }
    printf("\n");

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

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

    int requests[n];
    printf("Enter disk request string (comma-separated numbers): ");
```

```
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }

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

    // Call C-LOOK algorithm function
    c_look_algorithm(requests, n, head_position);

    return 0;
}
```

Slip13

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

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

a) Calculate and display the content of need matrix?
b) Is the system in safe state? If display the safe sequence. [15]

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

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

// Function to check if the current process can be executed
bool can_execute(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (need[process_id][i] > available[i]) {
            return false;
        }
    }
    return true;
}

// Function to execute the process and update available resources
void execute_process(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        available[i] += allocation[process_id][i];
        allocation[process_id][i] = 0;
        need[process_id][i] = 0;
    }
```

```
}

// Function to accept available resources
void accept_available() {
    printf("Enter available resources:\n");
    for (int i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &available[i]);
    }
}

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

// Function to calculate and display the need matrix
void display_need() {
    printf("Need Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}

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

int main() {
    // Initialize allocation and max matrices based on the snapshot
    int allocation_snapshot[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 3},
        {2, 1, 1},
```

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

    // Copy snapshot to allocation and max matrices
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            allocation[i][j] = allocation_snapshot[i][j];
            max[i][j] = max_snapshot[i][j];
        }
    }

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

        switch (choice) {
            case 'a':
                accept_available();
                break;
            case 'b':
                display_allocation_max();
                break;
            case 'c':
                display_need();
                break;
            case 'd':
                display_available();
                break;
            case 'e':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 'e');

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using SCAN algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.
176, 79, 34, 60, 92, 11, 41, 114
Starting Head Position: 65
Direction: Left

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

// Function to perform SCAN disk scheduling
void scanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
   int totalHeadMovements = 0;
   int i, j, temp, currentHead, index;

   // Sort the disk queue in ascending order
   for (i = 0; i < diskSize - 1; i++) {
      for (j = 0; j < diskSize - i - 1; j++) {
         if (diskQueue[j] > diskQueue[j + 1]) {
            // Swap elements if they are in the wrong order
            temp = diskQueue[j];
            diskQueue[j] = diskQueue[j + 1];
            diskQueue[j + 1] = temp;
         }
      }
   }

   // Find index of current head in the sorted disk queue
   for (i = 0; i < diskSize; i++) {
      if (diskQueue[i] >= startHead) {
         index = i;
         break;
      }
   }

   printf("Order of disk request serving:\n");

   // SCAN algorithm
   if (direction == 'L') {
      // Move left
      for (i = index; i >= 0; i--) {
         printf("%d ", diskQueue[i]);
         totalHeadMovements += abs(startHead - diskQueue[i]);
         startHead = diskQueue[i];
      }

      // Move to the beginning
      printf("0 ");
      totalHeadMovements += startHead;

      // Move right
      for (i = index + 1; i < diskSize; i++) {
         printf("%d ", diskQueue[i]);
```

```
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    } else if (direction == 'R') {
        // Move right
        for (i = index; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the end
        printf("199 ");
        totalHeadMovements += 199 - startHead;

        // Move left
        for (i = index - 1; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

int main() {
    int diskSize, startHead, i;
    char direction;

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

    int diskQueue[diskSize];

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);

    scanDisk(diskQueue, diskSize, startHead, direction);

    return 0;
}
```

Slip14

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>
#include <time.h>

#define MAX_BLOCKS 100

int disk[MAX_BLOCKS] = {0}; // 0 represents free block, 1 represents allocated block

void showBitVector(int n) {
    printf("\nBit Vector (Disk Allocation):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

void showDirectory(int n) {
    printf("\nDirectory Listing:\n");
    int start = -1;
    for (int i = 0; i < n; i++) {
        if (disk[i] == 1) {
            if (start == -1) {
                start = i;
            }
        } else {
            if (start != -1) {
                printf("File starting from block %d to %d\n", start, i - 1);
                start = -1;
            }
        }
    }
    if (start != -1) {
        printf("File starting from block %d to %d\n", start, n - 1);
    }
}
```

```
void deleteFile(int n) {
    int start;
    printf("\nEnter the starting block of the file to delete: ");
    scanf("%d", &start);

    if (start < 0 || start >= n || disk[start] == 0) {
        printf("\nInvalid starting block or the block is not allocated.\n");
        return;
    }

    // Deallocate the blocks for the file
    while (start < n && disk[start] == 1) {
        disk[start] = 0;
        start++;
    }

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

int main() {
    int n;

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

    srand(time(NULL)); // Seed for random block allocation

    // Randomly mark some blocks as allocated
    for (int i = 0; i < n; i++) {
        if (rand() % 2 == 1) {
            disk[i] = 1;
        }
    }

    int choice;
    do {
        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", &choice);

        switch (choice) {
            case 1:
                showBitVector(n);
                break;

            case 2:
                showDirectory(n);
                break;
```

```
            case 3:
                deleteFile(n);
                break;

            case 4:
                printf("\nExiting the program.\n");
                break;

            default:
                printf("\nInvalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using SSTF algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.
55, 58, 39, 18, 90, 160, 150, 38, 184
Start Head Position: 50

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

// Function to perform SSTF disk scheduling
void sstfDisk(int diskQueue[], int diskSize, int startHead) {
    int totalHeadMovements = 0;
    int i, j, minDistance, minIndex;

    printf("Order of disk request serving:\n");

    for (i = 0; i < diskSize; i++) {
        minDistance = abs(startHead - diskQueue[0]);
        minIndex = 0;

        // Find the request with the shortest seek time
        for (j = 1; j < diskSize; j++) {
            int distance = abs(startHead - diskQueue[j]);
            if (distance < minDistance) {
                minDistance = distance;
                minIndex = j;
            }
        }

        // Serve the request and update head position
        printf("%d ", diskQueue[minIndex]);
        totalHeadMovements += minDistance;
        startHead = diskQueue[minIndex];
```

```
        // Mark the served request as -1 to avoid serving it again
        diskQueue[minIndex] = -1;
    }

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

int main() {
    int diskSize, startHead, i;

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

    int diskQueue[diskSize];

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

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

    sstfDisk(diskQueue, diskSize, startHead);

    return 0;
}
```

Slip15

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 <stdbool.h>
#include <time.h>

#define MAX_BLOCKS 100

// Node structure for file blocks
typedef struct Node {
    int block_number;
    struct Node *next;
} Node;
```

```
// Function to initialize a linked list
Node* initialize_linked_list(int n) {
    Node *head = NULL;
    Node *prev = NULL;
    for (int i = 0; i < n; i++) {
        Node *new_node = (Node *)malloc(sizeof(Node));
        new_node->block_number = i;
        new_node->next = NULL;
        if (head == NULL) {
            head = new_node;
        } else {
            prev->next = new_node;
        }
        prev = new_node;
    }
    return head;
}

// Function to randomly mark some blocks as allocated
void mark_allocated(Node *head, int num_allocated) {
    srand(time(NULL));
    Node *current = head;
    while (num_allocated > 0 && current != NULL) {
        if (rand() % 2 == 0) { // Randomly mark block as allocated
            current->block_number = -1;
            num_allocated--;
        }
        current = current->next;
    }
}

// Function to display the bit vector
void show_bit_vector(Node *head) {
    printf("Bit Vector: ");
    Node *current = head;
    while (current != NULL) {
        if (current->block_number == -1) {
            printf("1 ");
        } else {
            printf("0 ");
        }
        current = current->next;
    }
    printf("\n");
}

// Function to create a new file
void create_new_file(Node **head, int file_size) {
    int count = 0;
    Node *current = *head;
    while (current != NULL && count < file_size) {
        if (current->block_number != -1) {
            printf("Block %d allocated to file.\n", current->block_number);
            current->block_number = -1;
            count++;
        }
```

```
        current = current->next;
    }
    if (count < file_size) {
        printf("Not enough free space available.\n");
    }
}

// Function to show directory (list of free blocks)
void show_directory(Node *head) {
    printf("Directory (Free Blocks): ");
    Node *current = head;
    while (current != NULL) {
        if (current->block_number != -1) {
            printf("%d ", current->block_number);
        }
        current = current->next;
    }
    printf("\n");
}

// Function to free memory allocated to linked list
void free_memory(Node *head) {
    Node *current = head;
    while (current != NULL) {
        Node *temp = current;
        current = current->next;
        free(temp);
    }
}

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

    Node *head = initialize_linked_list(n);

    // Mark some blocks as allocated
    int num_allocated = rand() % n;
    mark_allocated(head, num_allocated);

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

        switch (choice) {
            case 'a':
                show_bit_vector(head);
                break;
            case 'b': {
```

```
            int file_size;
            printf("Enter file size (number of blocks): ");
            scanf("%d", &file_size);
            create_new_file(&head, file_size);
            break;
        }
        case 'c':
            show_directory(head);
            break;
        case 'd':
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  } while (choice != 'd');

  // Free memory allocated to linked list
  free_memory(head);

  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>

// Function to perform C-SCAN disk scheduling
void cScanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
  int totalHeadMovements = 0;
  int i, j, temp, currentHead, index;

  // Sort the disk queue in ascending order
  for (i = 0; i < diskSize - 1; i++) {
    for (j = 0; j < diskSize - i - 1; j++) {
      if (diskQueue[j] > diskQueue[j + 1]) {
        // Swap elements if they are in the wrong order
        temp = diskQueue[j];
        diskQueue[j] = diskQueue[j + 1];
        diskQueue[j + 1] = temp;
      }
    }
  }

  // Find index of current head in the sorted disk queue
  for (i = 0; i < diskSize; i++) {
    if (diskQueue[i] >= startHead) {
      index = i;
      break;
```

```
        }
    }

    printf("Order of disk request serving:\n");

    // C-SCAN algorithm
    if (direction == 'R') {
        // Move right
        for (i = index; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the end
        printf("199 ");
        totalHeadMovements += 199 - startHead;

        // Move left
        for (i = 0; i < index; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    } else if (direction == 'L') {
        // Move left
        for (i = index; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the beginning
        printf("0 ");
        totalHeadMovements += startHead;

        // Move right
        for (i = diskSize - 1; i > index; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

int main() {
    int diskSize, startHead, i;
    char direction;

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

    int diskQueue[diskSize];
```

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);

    cScanDisk(diskQueue, diskSize, startHead, direction);

    return 0;
}
```

Slip 16

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

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

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

#define MAX_BLOCKS 100

int disk[MAX_BLOCKS] = {0}; // 0 represents free block, 1 represents allocated block

void showBitVector(int n) {
    printf("\nBit Vector (Disk Allocation):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

void showDirectory(int n) {
    printf("\nDirectory Listing:\n");
    int start = -1;
    for (int i = 0; i < n; i++) {
        if (disk[i] == 1) {
            if (start == -1) {
                start = i;
            }
        } else {
```

```
            if (start != -1) {
                printf("File starting from block %d to %d\n", start, i - 1);
                start = -1;
            }
        }
    }
    if (start != -1) {
        printf("File starting from block %d to %d\n", start, n - 1);
    }
}

void deleteFile(int n) {
    int start;
    printf("\nEnter the starting block of the file to delete: ");
    scanf("%d", &start);

    if (start < 0 || start >= n || disk[start] == 0) {
        printf("\nInvalid starting block or the block is not allocated.\n");
        return;
    }

    // Deallocate the blocks for the file
    while (start < n && disk[start] == 1) {
        disk[start] = 0;
        start++;
    }

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

int main() {
    int n;

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

    srand(time(NULL)); // Seed for random block allocation

    // Randomly mark some blocks as allocated
    for (int i = 0; i < n; i++) {
        if (rand() % 2 == 1) {
            disk[i] = 1;
        }
    }

    int choice;
    do {
        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", &choice);

        switch (choice) {
            case 1:
                showBitVector(n);
                break;

            case 2:
                showDirectory(n);
                break;

            case 3:
                deleteFile(n);
                break;

            case 4:
                printf("\nExiting the program.\n");
                break;

            default:
                printf("\nInvalid choice. Please enter a valid option.\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 <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
```

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

    // Find local max
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        if (local_numbers[i] > local_max) {
            local_max = local_numbers[i];
        }
    }

    // Find the global max using MPI_Reduce
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local max on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_max);
        }

        printf("\nGlobal max: %d\n", global_max);
    }

    MPI_Finalize();

    return 0;
}
```

Slip17

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
int disk[MAX_BLOCKS] = {0}; // 0 represents free block, 1 represents allocated block
int indexBlock[MAX_BLOCKS] = {0}; // Index block to store addresses of allocated blocks

void showBitVector(int n) {
```

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

void showDirectory(int n) {
    printf("\nDirectory Listing:\n");
    for (int i = 0; i < n; i++) {
        if (indexBlock[i] != -1) {
            printf("File starting from block %d\n", i);
        }
    }
}

void deleteFile(int n) {
    int start;
    printf("\nEnter the starting block of the file to delete: ");
    scanf("%d", &start);

    if (start < 0 || start >= n || indexBlock[start] == -1) {
        printf("\nInvalid starting block or the block is not allocated.\n");
        return;
    }

    // Deallocate the blocks for the file
    int current = indexBlock[start];
    while (current != -1) {
        int next = disk[current];
        disk[current] = 0;
        current = next;
    }

    indexBlock[start] = -1;

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

int main() {
    int n;

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

```
srand(time(NULL)); // Seed for random block allocation

// Randomly mark some blocks as allocated
for (int i = 0; i < n; i++) {
    if (rand() % 2 == 1) {
        disk[i] = 1;
        indexBlock[i] = i; // Initialize index block with the block itself
    } else {
        indexBlock[i] = -1; // Initialize index block with -1 for free blocks
    }
}

int choice;
do {
    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", &choice);

    switch (choice) {
        case 1:
            showBitVector(n);
            break;

        case 2:
            showDirectory(n);
            break;

        case 3:
            deleteFile(n);
            break;

        case 4:
            printf("\nExiting the program.\n");
            break;
        default:
            printf("\nInvalid choice. Please enter a valid option.\n");
    }
} while (choice != 4);
```

```
    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 look_algorithm(int requests[], int n, int head_position) {
    int total_head_movements = 0;
    int current_position = head_position;

    printf("Order of service: ");
    while (1) {
        int next_request = -1;
        int min_diff = __INT_MAX__;

        // Find the next request to serve
        for (int i = 0; i < n; i++) {
            if (requests[i] != -1) {
                int diff = abs(requests[i] - current_position);
                if (diff < min_diff) {
                    min_diff = diff;
                    next_request = requests[i];
                }
            }
        }

        // If no request found, break the loop
        if (next_request == -1) break;

        // Serve the request
        printf("%d ", next_request);
        total_head_movements += abs(next_request - current_position);
        current_position = next_request;
        requests[next_request] = -1; // Mark request as served
    }

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

```
}

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

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

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

    printf("Order of service: ");
    look_algorithm(requests, n, head_position);

    return 0;
}
```

Slip18

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
int disk[MAX_BLOCKS] = {0}; // 0 represents free block, 1 represents allocated block
int indexBlock[MAX_BLOCKS] = {0}; // Index block to store addresses of allocated blocks
```

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

void showDirectory(int n) {
    printf("\nDirectory Listing:\n");
    for (int i = 0; i < n; i++) {
        if (indexBlock[i] != -1) {
            printf("File starting from block %d\n", i);
        }
    }
}

void deleteFile(int n) {
    int start;
    printf("\nEnter the starting block of the file to delete: ");
    scanf("%d", &start);

    if (start < 0 || start >= n || indexBlock[start] == -1) {
        printf("\nInvalid starting block or the block is not allocated.\n");
        return;
    }

    // Deallocate the blocks for the file
    int current = indexBlock[start];
    while (current != -1) {
        int next = disk[current];
        disk[current] = 0;
        current = next;
    }

    indexBlock[start] = -1;

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

int main() {
    int n;

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

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

srand(time(NULL)); // Seed for random block allocation

// Randomly mark some blocks as allocated
for (int i = 0; i < n; i++) {
    if (rand() % 2 == 1) {
        disk[i] = 1;
        indexBlock[i] = i; // Initialize index block with the block itself
    } else {
        indexBlock[i] = -1; // Initialize index block with -1 for free blocks
    }
}

int choice;
do {
    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", &choice);

    switch (choice) {
        case 1:
            showBitVector(n);
            break;

        case 2:
            showDirectory(n);
            break;

        case 3:
            deleteFile(n);
            break;

        case 4:
            printf("\nExiting the program.\n");
            break;
        default:
            printf("\nInvalid choice. Please enter a valid option.\n");
    }
} while (choice != 4);
```

```
    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>

// Function to perform C-SCAN disk scheduling
void cScanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
    int totalHeadMovements = 0;
    int i, j, temp, currentHead, index;

    // Sort the disk queue in ascending order
    for (i = 0; i < diskSize - 1; i++) {
        for (j = 0; j < diskSize - i - 1; j++) {
            if (diskQueue[j] > diskQueue[j + 1]) {
                // Swap elements if they are in the wrong order
                temp = diskQueue[j];
                diskQueue[j] = diskQueue[j + 1];
                diskQueue[j + 1] = temp;
            }
        }
    }

    // Find index of current head in the sorted disk queue
    for (i = 0; i < diskSize; i++) {
        if (diskQueue[i] >= startHead) {
            index = i;
            break;
        }
    }

    printf("Order of disk request serving:\n");

    // C-SCAN algorithm
    if (direction == 'R') {
        // Move right
        for (i = index; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the end
```

```
        printf("199 ");
        totalHeadMovements += 199 - startHead;

        // Move left
        for (i = 0; i < index; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    } else if (direction == 'L') {
        // Move left
        for (i = index; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the beginning
        printf("0 ");
        totalHeadMovements += startHead;

        // Move right
        for (i = diskSize - 1; i > index; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

int main() {
    int diskSize, startHead, i;
    char direction;

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

    int diskQueue[diskSize];

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);
```

```
    cScanDisk(diskQueue, diskSize, startHead, direction);

    return 0;
}
```

Slip19

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	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>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

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

// Function to check if the current process can be executed
bool can_execute(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (need[process_id][i] > available[i]) {
            return false;
        }
    }
    return true;
}

// Function to execute the process and update available resources
void execute_process(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        available[i] += allocation[process_id][i];
        allocation[process_id][i] = 0;
        need[process_id][i] = 0;
    }
}
```

```
// Function to accept available resources
void accept_available() {
    printf("Enter available resources:\n");
    for (int i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &available[i]);
    }
}

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

// Function to calculate and display the need matrix
void display_need() {
    printf("Need Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}

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

int main() {
    // Initialize allocation and max matrices based on the snapshot
    int allocation_snapshot[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 3, 2 ,4},
        {1 ,2 ,0 ,1},
        {0 ,0 ,0 ,0},
        {3 ,3 ,2 ,2},
        {1 ,4 ,3 ,2},
        {2 ,4 ,1 ,4}
```

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

    // Copy snapshot to allocation and max matrices
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            allocation[i][j] = allocation_snapshot[i][j];
            max[i][j] = max_snapshot[i][j];
        }
    }

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

        switch (choice) {
            case 'a':
                accept_available();
                break;
            case 'b':
                display_allocation_max();
                break;
            case 'c':
                display_need();
                break;
            case 'd':
                display_available();
                break;
            case 'e':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 'e');

    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 perform SCAN disk scheduling
void scanDisk(int diskQueue[], int diskSize, int startHead, char direction) {
    int totalHeadMovements = 0;
    int i, j, temp, currentHead, index;

    // Sort the disk queue in ascending order
    for (i = 0; i < diskSize - 1; i++) {
        for (j = 0; j < diskSize - i - 1; j++) {
            if (diskQueue[j] > diskQueue[j + 1]) {
                // Swap elements if they are in the wrong order
                temp = diskQueue[j];
                diskQueue[j] = diskQueue[j + 1];
                diskQueue[j + 1] = temp;
            }
        }
    }

    // Find index of current head in the sorted disk queue
    for (i = 0; i < diskSize; i++) {
        if (diskQueue[i] >= startHead) {
            index = i;
            break;
        }
    }

    printf("Order of disk request serving:\n");

    // SCAN algorithm
    if (direction == 'L') {
        // Move left
        for (i = index; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the beginning
        printf("0 ");
        totalHeadMovements += startHead;

        // Move right
        for (i = index + 1; i < diskSize; i++) {
```

```
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    } else if (direction == 'R') {
        // Move right
        for (i = index; i < diskSize; i++) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }

        // Move to the end
        printf("199 ");
        totalHeadMovements += 199 - startHead;

        // Move left
        for (i = index - 1; i >= 0; i--) {
            printf("%d ", diskQueue[i]);
            totalHeadMovements += abs(startHead - diskQueue[i]);
            startHead = diskQueue[i];
        }
    }

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

int main() {
    int diskSize, startHead, i;
    char direction;

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

    int diskQueue[diskSize];

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

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

    printf("Enter the direction (L for Left, R for Right): ");
    scanf(" %c", &direction);

    scanDisk(diskQueue, diskSize, startHead, direction);

    return 0;
}
```

Slip20

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>

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

// Function to simulate SCAN disk scheduling algorithm
void scan_algorithm(int requests[], int n, int head_position, char direction) {
    int total_head_movements = 0;
    int serving_order[n];
    int serving_index = 0;

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

    // Serve requests based on direction
    if (direction == 'l' || direction == 'L') {
        // Serve requests to the left of head position
        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] <= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
        // Serve requests to the right
        for (int i = 0; i < n; i++) {
```

```
            if (requests[i] >= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
    } else if (direction == 'r' || direction == 'R') {
        // Serve requests to the right of head position
        for (int i = 0; i < n; i++) {
            if (requests[i] >= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
        // Serve requests to the left
        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] < head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
    }

    // Print serving order
    printf("Order of service: ");
    for (int i = 0; i < serving_index; i++) {
        printf("%d ", serving_order[i]);
    }
    printf("\n");

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

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

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

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

    char direction;
```

```
    printf("Enter direction (left/l or right/r): ");
    scanf(" %c", &direction);

    // Call SCAN algorithm function
    scan_algorithm(requests, n, head_position, direction);

    return 0;
}
```

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

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

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

    // Find local max
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        if (local_numbers[i] > local_max) {
            local_max = local_numbers[i];
        }
    }

    // Find the global max using MPI_Reduce
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local max on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_max);
```

```
        }

        printf("\nGlobal max: %d\n", global_max);
    }

    MPI_Finalize();

    return 0;
}
```

Slip 21

Q.1 Write a simulation program for disk scheduling using FCFS algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments.
55, 58, 39, 18, 90, 160, 150, 38, 184
Start Head Position: 50

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

// Function to simulate FCFS disk scheduling algorithm
void fcfs_algorithm(int requests[], int n, int head_position) {
    int total_head_movements = 0;

    printf("Order of service: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", requests[i]);
        total_head_movements += abs(head_position - requests[i]);
        head_position = requests[i];
    }
    printf("\n");

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

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

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

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

```
    // Call FCFS algorithm function
    fcfs_algorithm(requests, n, head_position);

    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>

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }
```

```
    MPI_Finalize();

    return 0;
}
```

Slip22

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

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

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }
```

```
        printf("\nGlobal sum: %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

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

#define MAX_BLOCKS 100

bool allocated[MAX_BLOCKS];

// Function to mark some blocks as allocated randomly
void mark_allocated(int n) {
    srand(time(NULL));
    int num_allocated = rand() % n;
    for (int i = 0; i < num_allocated; i++) {
        int block = rand() % n;
        if (!allocated[block]) {
            allocated[block] = true;
        }
    }
}

// Function to display the bit vector
void show_bit_vector(int n) {
    printf("Bit Vector: ");
    for (int i = 0; i < n; i++) {
        if (allocated[i]) {
            printf("1 ");
        } else {
            printf("0 ");
        }
    }
    printf("\n");
}

// Function to delete an already created file
void delete_file(int n) {
    int start_block;
    printf("Enter the starting block of the file to delete: ");
```

```
    scanf("%d", &start_block);
    if (start_block < 0 || start_block >= n) {
        printf("Invalid starting block.\n");
        return;
    }
    if (!allocated[start_block]) {
        printf("The specified block is not allocated.\n");
        return;
    }

    // Mark blocks as free
    while (start_block < n && allocated[start_block]) {
        allocated[start_block] = false;
        start_block++;
    }
    printf("File deleted successfully.\n");
}

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

    mark_allocated(n);

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

        switch (choice) {
            case 'a':
                show_bit_vector(n);
                break;
            case 'b':
                delete_file(n);
                break;
            case 'c':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 'c');

    return 0;
}
```

Slip23

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>
// Function to display a matrix
void display_matrix(int **matrix, int m, int n, char *title) {
    printf("%s\n", title);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

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

// Function to check if request can be granted
int check_request(int process, int *request, int **need, int *available, int n) {
    for (int i = 0; i < n; ++i) {
        if (request[i] > need[process][i] || 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 *available = (int *)malloc(n * sizeof(int));
printf("Enter the number of instances for each resource type:\n");
for (int i = 0; i < n; ++i) {
    printf("Resource %d: ", i + 1);
    scanf("%d", &available[i]);
}

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

int **maximum = (int **)malloc(m * sizeof(int *));
printf("Enter the maximum requirement matrix:\n");
for (int i = 0; i < m; ++i) {
    maximum[i] = (int *)malloc(n * sizeof(int));
    for (int j = 0; j < n; ++j) {
        scanf("%d", &maximum[i][j]);
    }
}

int **need = (int **)malloc(m * sizeof(int *));
for (int i = 0; i < m; ++i) {
    need[i] = (int *)malloc(n * sizeof(int));
}

calculate_need_matrix(allocation, maximum, need, m, n);
display_matrix(need, m, n, "Need matrix:");

int process;
printf("Enter the process number making the request: ");
scanf("%d", &process);
process--; // Adjusting to 0-based indexing

int *request = (int *)malloc(n * sizeof(int));
printf("Enter the request for each resource type: ");
for (int i = 0; i < n; ++i) {
    scanf("%d", &request[i]);
}

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

// Freeing dynamically allocated memory
free(available);
```

```
    for (int i = 0; i < m; ++i) {
        free(allocation[i]);
        free(maximum[i]);
        free(need[i]);
    }
    free(allocation);
    free(maximum);
    free(need);
    free(request);

    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

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

// Function to perform SSTF disk scheduling
void sstfDisk(int diskQueue[], int diskSize, int startHead) {
    int totalHeadMovements = 0;
    int i, j, minDistance, minIndex;

    printf("Order of disk request serving:\n");

    for (i = 0; i < diskSize; i++) {
        minDistance = abs(startHead - diskQueue[0]);
        minIndex = 0;

        // Find the request with the shortest seek time
        for (j = 1; j < diskSize; j++) {
            int distance = abs(startHead - diskQueue[j]);
            if (distance < minDistance) {
                minDistance = distance;
                minIndex = j;
            }
        }

        // Serve the request and update head position
        printf("%d ", diskQueue[minIndex]);
        totalHeadMovements += minDistance;
        startHead = diskQueue[minIndex];

        // Mark the served request as -1 to avoid serving it again
        diskQueue[minIndex] = -1;
```

```
    }

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

int main() {
    int diskSize, startHead, i;

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

    int diskQueue[diskSize];

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

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

    sstfDisk(diskQueue, diskSize, startHead);

    return 0;
}
```

Slip24

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>

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
```

```
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```

Q.2 Write a C program to simulate Banker's algorithm for the purpose of deadlock avoidance. The following snapshot of system, A, B, C and D are the resource type.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

a) Calculate and display the content of need matrix?
b) Is the system in safe state? If display the safe sequence. [15]

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

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

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

```
// Function to check if the current process can be executed
bool can_execute(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (need[process_id][i] > available[i]) {
            return false;
        }
    }
    return true;
}

// Function to execute the process and update available resources
void execute_process(int process_id) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        available[i] += allocation[process_id][i];
        allocation[process_id][i] = 0;
        need[process_id][i] = 0;
    }
}

// Function to accept available resources
void accept_available() {
    printf("Enter available resources:\n");
    for (int i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &available[i]);
    }
}

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

// Function to calculate and display the need matrix
void display_need() {
    printf("Need Matrix:\n");
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
}
```

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

int main() {
    // Initialize allocation and max matrices based on the snapshot
    int allocation_snapshot[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 3},
        {2, 1, 1},
        {0, 0, 2}
    };
    int max_snapshot[MAX_PROCESSES][MAX_RESOURCES] = {
        {0, 0, 0},
        {2, 0, 2},
        {0, 0, 0},
        {1, 0, 0},
        {0, 0, 2}
    };

    // Copy snapshot to allocation and max matrices
    for (int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            allocation[i][j] = allocation_snapshot[i][j];
            max[i][j] = max_snapshot[i][j];
        }
    }

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

        switch (choice) {
            case 'a':
                accept_available();
                break;
            case 'b':
                display_allocation_max();
                break;
            case 'c':
                display_need();
```

```
            break;
        case 'd':
            display_available();
            break;
        case 'e':
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  } while (choice != 'e');

  return 0;
}
```

Slip25

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>

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

// Function to simulate LOOK disk scheduling algorithm
void look_algorithm(int requests[], int n, int head_position, char direction) {
  int total_head_movements = 0;
  int serving_order[n];
  int serving_index = 0;

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

  // Serve requests based on direction
```

```
    if (direction == 'l' || direction == 'L') {
        // Serve requests to the left of head position
        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] <= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
        // Serve requests to the right
        for (int i = 0; i < n; i++) {
            if (requests[i] >= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
    } else if (direction == 'r' || direction == 'R') {
        // Serve requests to the right of head position
        for (int i = 0; i < n; i++) {
            if (requests[i] >= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
        // Serve requests to the left
        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] < head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
    }

    // Print serving order
    printf("Order of service: ");
    for (int i = 0; i < serving_index; i++) {
        printf("%d ", serving_order[i]);
    }
    printf("\n");

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

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

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

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

    char direction;
    printf("Enter direction (left/l or right/r): ");
    scanf(" %c", &direction);

    // Call LOOK algorithm function
    look_algorithm(requests, n, head_position, 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

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

#define MAX_BLOCKS 100

bool allocated[MAX_BLOCKS];

// Function to mark some blocks as allocated randomly
void mark_allocated(int n) {
    srand(time(NULL));
    int num_allocated = rand() % n;
    for (int i = 0; i < num_allocated; i++) {
        int block = rand() % n;
        if (!allocated[block]) {
            allocated[block] = true;
        }
    }
}
```

```
// Function to display the bit vector
void show_bit_vector(int n) {
    printf("Bit Vector: ");
    for (int i = 0; i < n; i++) {
        if (allocated[i]) {
            printf("1 ");
        } else {
            printf("0 ");
        }
    }
    printf("\n");
}

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

    // Mark blocks as allocated for the new file
    allocated[start_block] = true;
    printf("New file created successfully.\n");
}

// Function to show directory (list of free blocks)
void show_directory(int n) {
    printf("Directory (Free Blocks): ");
    for (int i = 0; i < n; i++) {
        if (!allocated[i]) {
            printf("%d ", i);
        }
    }
    printf("\n");
}

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

    mark_allocated(n);

    char choice;
    do {
```

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

        switch (choice) {
            case 'a':
                show_bit_vector(n);
                break;
            case 'b':
                create_new_file(n);
                break;
            case 'c':
                show_directory(n);
                break;
            case 'd':
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 'd');

    return 0;
}
```

Slip26

Q.1.Consider the following snapshot of system, A, B, C, 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				

Using Resource –Request algorithm to Check whether the current system is in safe state or not . [15]

```
#include <stdio.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 4

int available[MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
int finish[MAX_PROCESSES] = {0};
```

```
int isSafeState(int processes[], int n) {
    int work[MAX_RESOURCES];
    int i, j, finished = 0, safeSequence[MAX_PROCESSES], count = 0;

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

    while (finished < n) {
        int found = 0;
        for (i = 0; i < n; i++) {
            if (!finish[i]) {
                int safe = 1;
                for (j = 0; j < MAX_RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        safe = 0;
                        break;
                    }
                }
                if (safe) {
                    for (j = 0; j < MAX_RESOURCES; j++) {
                        work[j] += allocation[i][j];
                    }
                    finish[i] = 1;
                    safeSequence[count++] = i;
                    found = 1;
                    finished++;
                }
            }
        }
        if (!found) {
            return 0; // System is not in a safe state
        }
    }

    printf("Safe sequence: ");
    for (i = 0; i < n; i++) {
        printf("P%d ", safeSequence[i]);
    }
    printf("\n");

    return 1; // System is in a safe state
}

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

```
int main() {
    // Initialize the allocation, max, and available arrays
    int i, j;
    int processes[MAX_PROCESSES] = {0, 1, 2, 3, 4};

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

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

    int available[MAX_RESOURCES] = {1, 5, 2, 0};

    calculateNeedMatrix();

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

    if (isSafeState(processes, MAX_PROCESSES)) {
        printf("System is in safe state.\n");
    } else {
        printf("System is not in safe state.\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.
56, 59, 40, 19, 91, 161, 151, 39, 185
Start Head Position: 48

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

// Function to simulate FCFS disk scheduling algorithm
void fcfs_algorithm(int requests[], int n, int head_position) {
    int total_head_movements = 0;

    printf("Order of service: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", requests[i]);
        total_head_movements += abs(head_position - requests[i]);
        head_position = requests[i];
    }
    printf("\n");

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

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

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

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

    // Call FCFS algorithm function
    fcfs_algorithm(requests, n, head_position);

    return 0;
}
```

Slip27

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>

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

// Function to simulate LOOK disk scheduling algorithm
void look_algorithm(int requests[], int n, int head_position, char direction) {
    int total_head_movements = 0;
    int serving_order[n];
    int serving_index = 0;

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

    // Serve requests based on direction
    if (direction == 'l' || direction == 'L') {
        // Serve requests to the left of head position
        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] <= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
                serving_order[serving_index++] = head_position;
            }
        }
        // Serve requests to the right
        for (int i = 0; i < n; i++) {
            if (requests[i] >= head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
```

```
            serving_order[serving_index++] = head_position;
        }
    }
} else if (direction == 'r' || direction == 'R') {
    // Serve requests to the right of head position
    for (int i = 0; i < n; i++) {
        if (requests[i] >= head_position) {
            total_head_movements += abs(head_position - requests[i]);
            head_position = requests[i];
            serving_order[serving_index++] = head_position;
        }
    }
    // Serve requests to the left
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] < head_position) {
            total_head_movements += abs(head_position - requests[i]);
            head_position = requests[i];
            serving_order[serving_index++] = head_position;
        }
    }
}

// Print serving order
printf("Order of service: ");
for (int i = 0; i < serving_index; i++) {
    printf("%d ", serving_order[i]);
}
printf("\n");

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

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

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

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

char direction;
printf("Enter direction (left/l or right/r): ");
scanf(" %c", &direction);
```

```
    // Call LOOK algorithm function
    look_algorithm(requests, n, head_position, 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 <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

    // Find local max
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        if (local_numbers[i] > local_max) {
            local_max = local_numbers[i];
        }
    }

    // Find the global max using MPI_Reduce
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local max on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_max);
        }

        printf("\nGlobal max: %d\n", global_max);
```

```
    }

    MPI_Finalize();

    return 0;
}
```

Slip28

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>

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

// Function to simulate C-LOOK disk scheduling algorithm
void clook_algorithm(int requests[], int n, int head_position, char direction) {
    int total_head_movements = 0;
    int serving_order[n];
    int serving_index = 0;

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

    // Serve requests based on direction
    if (direction == 'l' || direction == 'L') {
        // Serve requests to the left of head position
        for (int i = n - 1; i >= 0; i--) {
            if (requests[i] < head_position) {
                total_head_movements += abs(head_position - requests[i]);
                head_position = requests[i];
```

```
            serving_order[serving_index++] = head_position;
        }
    }
    // Serve requests to the right
    for (int i = 0; i < n; i++) {
        if (requests[i] >= head_position && requests[i] < requests[0]) {
            total_head_movements += abs(head_position - requests[i]);
            head_position = requests[i];
            serving_order[serving_index++] = head_position;
        }
    }
} else if (direction == 'r' || direction == 'R') {
    // Serve requests to the right of head position
    for (int i = 0; i < n; i++) {
        if (requests[i] > head_position) {
            total_head_movements += abs(head_position - requests[i]);
            head_position = requests[i];
            serving_order[serving_index++] = head_position;
        }
    }
    // Serve requests to the left
    for (int i = 0; i < n; i++) {
        if (requests[i] < head_position && requests[i] >= requests[0]) {
            total_head_movements += abs(head_position - requests[i]);
            head_position = requests[i];
            serving_order[serving_index++] = head_position;
        }
    }
}

// Print serving order
printf("Order of service: ");
for (int i = 0; i < serving_index; i++) {
    printf("%d ", serving_order[i]);
}
printf("\n");

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

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

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

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

    char direction;
    printf("Enter direction (left/l or right/r): ");
    scanf(" %c", &direction);

    // Call C-LOOK algorithm function
    clook_algorithm(requests, n, head_position, direction);

    return 0;
}
```

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

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

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
```

```
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```

Slip29

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

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

#define ARRAY_SIZE 1000

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

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

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

    // Sum the local sums on each process
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    // Print results on rank 0
    if (rank == 0) {
        printf("Local sum on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_sum);
        }

        printf("\nGlobal sum: %d\n", global_sum);
    }

    MPI_Finalize();

    return 0;
}
```

Q.2 Write a simulation program for disk scheduling using C-LOOK algorithm. Accept total number of disk blocks, disk request string, and current head position from the user. Display the list of request in the order in which it is served. Also display the total number of head moments..
80, 150, 60,135, 40, 35, 170
Starting Head Position: 70
Direction: Right

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

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

// Function to simulate C-LOOK disk scheduling algorithm
void c_look_algorithm(int requests[], int n, int head_position) {
    int total_head_movements = 0;
    int serving_order[n];
    int serving_index = 0;

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

    // Serve requests to the right of head position
    for (int i = 0; i < n; i++) {
        if (requests[i] >= head_position) {
            total_head_movements += abs(head_position - requests[i]);
```

```
            head_position = requests[i];
            serving_order[serving_index++] = head_position;
        }
    }

    // Move head to the beginning and serve requests to the right again
    total_head_movements += abs(head_position - requests[0]);
    head_position = requests[0];
    serving_order[serving_index++] = head_position;

    // Print serving order
    printf("Order of service: ");
    for (int i = 0; i < serving_index; i++) {
        printf("%d ", serving_order[i]);
    }
    printf("\n");

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

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

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

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

    // Call C-LOOK algorithm function
    c_look_algorithm(requests, n, head_position);

    return 0;
}
```

Slip30

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

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
    int rank, size;
```

```
    int i;
    int local_max = 0, global_max = 0;

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

    // Seed for random number generation based on rank
    srand(rank);

    // Generate local random numbers
    int local_numbers[ARRAY_SIZE / size];
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        local_numbers[i] = rand() % 100;
    }

    // Find local max
    for (i = 0; i < ARRAY_SIZE / size; i++) {
        if (local_numbers[i] > local_max) {
            local_max = local_numbers[i];
        }
    }

    // Find the global max using MPI_Reduce
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print results on rank 0
    if (rank == 0) {
        printf("Local max on each process:\n");
        for (i = 0; i < size; i++) {
            printf("Process %d: %d\n", i, local_max);
        }

        printf("\nGlobal max: %d\n", global_max);
    }

    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>
```

```
// Function to calculate total head movements
int calculateHeadMovements(int requestQueue[], int n, int headPosition)
{
    int totalHeadMovements = 0;
    int currentHeadPosition = headPosition;

    // Loop through the request queue and calculate head movements
    for (int i = 0; i < n; i++) {
        totalHeadMovements += abs(requestQueue[i] - currentHeadPosition);
        currentHeadPosition = requestQueue[i];
    }

    return totalHeadMovements;
}

int main() {
    int n, headPosition;

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

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

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

    // Displaying the request order
    printf("Request Order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", requestQueue[i]);
    }
    printf("\n");

    printf("Start Head Position: %d\n", headPosition);

    // Calculating total head movements
    int totalHeadMovements = calculateHeadMovements(requestQueue, n, headPosition);
    printf("Total number of head movements: %d\n", totalHeadMovements);

    return 0;
}
```

The END