Day 4 lab programs

1.Construct a C program to simulate the First in First Out paging technique of memory management.

Code:

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

#define FRAME_SIZE 3 // Number of frames in memory

void fifo(int pages[], int n) {
    int frame[FRAME_SIZE], pageFaults = 0, index = 0;
    int isPageInFrame;

    // Initialize frames to -1
    for (int i = 0; i < FRAME_SIZE; i++) {
        frame[i] = -1;
    }

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

        // Check if the page is already in one of the frames
        for (int j = 0; j < FRAME_SIZE; j++) {
            if (frame[j] == pages[i]) {
                isPageInFrame = 1;
                break;
            }
        }

        // If the page is not in the frame, we have a page fault
        if (!isPageInFrame) {
            frame[index] = pages[i]; // Replace the oldest page
            index = (index + 1) % FRAME_SIZE; // Move to the next frame
            pageFaults++;
        }
        // Display current frame status
        printf("Current frame: ");
        for (int j = 0; j < FRAME_SIZE; j++) {
            printf("%d ", frame[j]);
        }
        printf("\n");
    }

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

int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4}; // Example page reference string
    int n = sizeof(pages) / sizeof(pages[0]);
    fifo(pages, n);
    return 0;
}
```

Output:

```
Total Page Faults: 7
```

2.Construct a C program to simulate the Least Recently Used paging technique of memory management.

Code:

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

#define FRAME_SIZE 3 // Number of frames in memory

void simulateLRU(int pages[], int n) {
    int frame[FRAME_SIZE], pageFaults = 0, i, j, k, lruIndex;
    int pageFound;
```

```
for (i = 0; i < FRAME_SIZE; i++) {
    frame[i] = -1; // Initialize frames
}

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

    // Check if page is already in frame
    for (j = 0; j < FRAME_SIZE; j++) {
        if (frame[j] == pages[i]) {
            pageFound = 1;
            break;
        }
    }

    // If page is not found, we have a page fault
    if (!pageFound) {
        pageFaults++;
        lruIndex = -1;

        // Find the least recently used page
        for (j = 0; j < FRAME_SIZE; j++) {
            if (frame[j] == -1) {
                lruIndex = j;
                break;
            }
        }

        if (lruIndex == -1) {
            // If all frames are full, replace the least recently used
            lruIndex = 0; // Start with the first frame
            for (j = 1; j < FRAME_SIZE; j++) {
                if (frame[j] < frame[lruIndex]) {
                    lruIndex = j;
                }
            }
        }

        frame[lruIndex] = pages[i]; // Replace the page
    }
```

```
        // Update the usage order
        for (j = 0; j < FRAME_SIZE; j++) {
            if (frame[j] != -1) {
                frame[j] = (frame[j] == pages[i]) ? i : frame[j];
            }
        }
    }

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

int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3};
    int n = sizeof(pages) / sizeof(pages[0]);
    simulateLRU(pages, n);
    return 0;
}
```

Output:

```
Total Page Faults: 5
```

3.Construct a C program to simulate the optimal paging technique of memory management

Code:

```
#include <stdio.h>

#define FRAME_SIZE 3
#define MAX_PAGES 10

void optimalPageReplacement(int pages[], int n) {
    int frame[FRAME_SIZE], pageFaults = 0;
    int i, j, k, pos, max, flag;

    for (i = 0; i < FRAME_SIZE; i++)
```

```
        frame[i] = -1; // Initialize frames

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

        // Check if page is already in frame
        for (j = 0; j < FRAME_SIZE; j++) {
            if (frame[j] == pages[i]) {
                flag = 1;
                break;
            }
        }

        // If page is not found, we need to replace
        if (flag == 0) {
            pageFaults++;
            // Check for empty frame
            for (j = 0; j < FRAME_SIZE; j++) {
                if (frame[j] == -1) {
                    frame[j] = pages[i];
                    flag = 1;
                    break;
                }
            }

            // If no empty frame, find the optimal page to replace
            if (flag == 0) {
                max = -1;
                for (j = 0; j < FRAME_SIZE; j++) {
                    for (k = i + 1; k < n; k++) {
                        if (frame[j] == pages[k]) {
                            if (k > max) {
                                max = k;
                                pos = j;
                            }
                            break;
                        }
                    }
                    if (k == n) {
                        pos = j;
                        break;
                    }
                }
                frame[pos] = pages[i];
            }
        }

        // Display current frame state
        printf("Frame: ");
        for (j = 0; j < FRAME_SIZE; j++)
            printf("%d ", frame[j]);
        printf("\n");
    }
    printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
    int pages[MAX_PAGES] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3};
    int n = sizeof(pages) / sizeof(pages[0]);
    optimalPageReplacement(pages, n);
    return 0;
}
```

Output:

```
            Frame: 2 4 3
Total Page Faults: 6
```

4.Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

Code:

```
#include <stdio.h>

#define MAX_RECORDS 100
```

```
void readRecords(char records[MAX_RECORDS][100], int count) {
    for (int i = 0; i < count; i++) {
        printf("Record %d: %s\n", i + 1, records[i]);
    }
}

int main() {
    char records[MAX_RECORDS][100];
    int count;

    printf("Enter the number of records (max %d): ", MAX_RECORDS);
    scanf("%d", &count);
    getchar(); // To consume the newline character

    for (int i = 0; i < count; i++) {
        printf("Enter record %d: ", i + 1);
        fgets(records[i], sizeof(records[i]), stdin);
    }

    printf("\nReading all records:\n");
    readRecords(records, count);

    return 0;
}
```

Output:

```
Record 5: 2
```

5. Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.

Code:

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

typedef struct Block {
    int data;              // Data stored in the block
    struct Block* next;    // Pointer to the next block
} Block;
```

```
typedef struct File {
    Block* first;          // Pointer to the first block
    Block* last;           // Pointer to the last block
} File;

// Function to create a new block
Block* createBlock(int data) {
    Block* newBlock = (Block*)malloc(sizeof(Block));
    newBlock->data = data;
    newBlock->next = NULL;
    return newBlock;
}

// Function to add a block to the file
void addBlock(File* file, int data) {
    Block* newBlock = createBlock(data);
    if (file->first == NULL) {
        file->first = newBlock;
        file->last = newBlock;
    } else {
        file->last->next = newBlock;
        file->last = newBlock;
    }
}

// Function to display the file blocks
void displayFile(File* file) {
    Block* current = file->first;
    while (current != NULL) {
        printf("Block Data: %d\n", current->data);
        current = current->next;
    }
}

int main() {
    File myFile = {NULL, NULL};
    addBlock(&myFile, 10);
    addBlock(&myFile, 20);
    addBlock(&myFile, 30);

    printf("File Blocks:\n");
    displayFile(&myFile);

    return 0;
}
```

Output:

```
Block Data: 20
Block Data: 30
```

6.With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.

Code:

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

typedef struct Block {
    int data;               // Simulated data in the block
    struct Block* next;     // Pointer to the next block
} Block;

typedef struct File {
    Block* first;           // Pointer to the first block
    Block* last;            // Pointer to the last block
} File;

// Function to create a new block
Block* createBlock(int data) {
    Block* newBlock = (Block*)malloc(sizeof(Block));
    newBlock->data = data;
    newBlock->next = NULL;
    return newBlock;
}
```

```
// Function to add a block to the file
void addBlock(File* file, int data) {
    Block* newBlock = createBlock(data);
    if (file->last) {
        file->last->next = newBlock; // Link the new block
    } else {
        file->first = newBlock;       // First block
    }
    file->last = newBlo
ck;           // Update last block
}

// Function to display the file blocks
void displayFile(File* file) {
    Block* current = file->first;
    while (current) {
        printf("Block Data: %d\n", current->data);
        current = current->next;
    }
}

int main() {
    File myFile = {NULL, NULL}; // Initialize file
    addBlock(&myFile, 10);
    addBlock(&myFile, 20);
    addBlock(&myFile, 30);

    printf("File Blocks:\n");
    displayFile(&myFile);

    return 0;
}
```

Output:

```
Block Data: 20
Block Data: 30
```

7.Construct a C program to simulate the First Come First Served disk scheduling Algorithm.

Code:

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

// Function to calculate the total head movement
void FCFS(int requestQueue[], int n, int head) {
   int totalHeadMovement = 0;
   printf("\nSequence of disk accesses: %d", head);

   for (int i = 0; i < n; i++) {
      printf(" -> %d", requestQueue[i]);
      totalHeadMovement += abs(requestQueue[i] - head);
      head = requestQueue[i];
   }

   printf("\nTotal Head Movement: %d cylinders\n", totalHeadMovement);
}

int main() {
   int n, head;

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

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

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

   FCFS(requestQueue, n, head);

   return 0;
}
```

Output:

```
Enter the number of disk requests: 5
Enter the request sequence: 98 183 37 122 14
Enter the initial head position: 53

Sequence of disk accesses: 53 -> 98 -> 183 -> 37 -> 122 -> 14
Total Head Movement: 355 cylinders
```

8.Design a C program to simulate SCAN disk scheduling algorithm.

Code:

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

// Function to perform SCAN disk scheduling
void SCAN(int requestQueue[], int n, int head, int diskSize, int direction) {
    int totalHeadMovement = 0;
    int sortedQueue[n + 2]; // Include boundaries (0 and diskSize-1)
    int index = 0;

    // Copy requests into sortedQueue and add boundaries
    for (int i = 0; i < n; i++) {
        sortedQueue[i] = requestQueue[i];
    }
    sortedQueue[n] = 0; // Lower boundary
    sortedQueue[n + 1] = diskSize - 1; // Upper boundary

    // Sort the queue
    for (int i = 0; i < n + 2; i++) {
        for (int j = i + 1; j < n + 2; j++) {
            if (sortedQueue[i] > sortedQueue[j]) {
                int temp = sortedQueue[i];
                sortedQueue[i] = sortedQueue[j];
                sortedQueue[j] = temp;
            }
        }
    }

    // Find the position of the initial head
    for (int i = 0; i < n + 2; i++) {
        if (sortedQueue[i] >= head) {
            index = i;
            break;
        }
    }

    printf("\nSequence of disk accesses: %d", head);

    // Move in the selected direction
    if (direction == 1) { // Moving towards larger values
        for (int i = index; i < n + 2; i++) {
            printf(" -> %d", sortedQueue[i]);
            totalHeadMovement += abs(sortedQueue[i] - head);
            head = sortedQueue[i];
        }
        // Move back to the beginning
        for (int i = index - 1; i >= 0; i--) {
            printf(" -> %d", sortedQueue[i]);
            totalHeadMovement += abs(sortedQueue[i] - head);
            head = sortedQueue[i];
        }
    } else { // Moving towards smaller values
        for (int i = index - 1; i >= 0; i--) {
            printf(" -> %d", sortedQueue[i]);
            totalHeadMovement += abs(sortedQueue[i] - head);
            head = sortedQueue[i];
        }
        // Move towards larger values
        for (int i = index; i < n + 2; i++) {
            printf(" -> %d", sortedQueue[i]);
            totalHeadMovement += abs(sortedQueue[i] - head);
            head = sortedQueue[i];
        }
    }

    printf("\nTotal Head Movement: %d cylinders\n", totalHeadMovement);
}

int main() {
    int n, head, diskSize, direction;
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

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

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

    printf("Enter the disk size: ");
    scanf("%d", &diskSize);

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

    SCAN(requestQueue, n, head, diskSize, direction);

    return 0;
}
```

Output:

```
Enter the number of disk requests: 5
Enter the request sequence: 98 183 37 122 14
Enter the initial head position: 53
Enter the total disk size: 200
Enter the direction (1 for right, 0 for left): 1

Disk Access Sequence: 53 -> 98 -> 122 -> 183 -> 199 -> 37 -> 14
Total Head Movement: 339 cylinders
```

9.Develop a C program to simulate C-SCAN disk scheduling algorithm.

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

```
void c_scan_disk_scheduling(int request[], int n, int head, int disk_size, int direction) {
    int total_movement = 0;
    int left[100], right[100], left_count = 0, right_count = 0;

    // Separating requests based on their position relative to the head
    for (int i = 0; i < n; i++) {
        if (request[i] < head)
            left[left_count++] = request[i];
        else
            right[right_count++] = request[i];
    }

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

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

    // Start movement
    printf("\nDisk Access Sequence: %d", head);

    if (direction == 1) { // Moving right
        for (int i = 0; i < right_count; i++) {
            total_movement += abs(head - right[i]);
            head = right[i];
            printf(" -> %d", head);
        }
        // Move to the end of the disk
        if (head != disk_size - 1) {
            total_movement += abs(head - (disk_size - 1));
            head = disk_size - 1;
            printf(" -> %d", head);
        }
        // Jump to the start of the disk
        total_movement += abs(head - 0);
        head = 0;
        printf(" -> %d", head);
        for (int i = 0; i < left_count; i++) {
            total_movement += abs(head - left[i]);
            head = left[i];
            printf(" -> %d", head);
        }
    } else { // Moving left
        for (int i = left_count - 1; i >= 0; i--) {
            total_movement += abs(head - left[i]);
            head = left[i];
            printf(" -> %d", head);
        }
        // Move to the start of the disk
        if (head != 0) {
            total_movement += abs(head - 0);
            head = 0;
            printf(" -> %d", head);
        }
        // Jump to the end of the disk
        total_movement += abs(head - (disk_size - 1));
        head = disk_size - 1;
        printf(" -> %d", head);
        for (int i = right_count - 1; i >= 0; i--) {
            total_movement += abs(head - right[i]);
            head = right[i];
            printf(" -> %d", head);
        }
    }

    printf("\nTotal Head Movement: %d cylinders\n", total_movement);
}
```

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

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

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

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

    printf("Enter the total disk size: ");
    scanf("%d", &disk_size);

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

    c_scan_disk_scheduling(request, n, head, disk_size, direction);

    return 0;
}
```

Output:

```
Enter the number of disk requests: 5
Enter the request sequence: 98 183 37 122 14
Enter the initial head position: 53
Enter the total disk size: 200
Enter the direction (1 for right, 0 for left): 1

Disk Access Sequence: 53 -> 98 -> 122 -> 183 -> 199 -> 0 -> 14 -> 37
Total Head Movement: 382 cylinders
```

10.Illustrate the various File Access Permission and different types of users in Linux.

Code:

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

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

void display_permissions(const char *filename) {
    struct stat fileStat;

    if (stat(filename, &fileStat) < 0) {
        perror("Failed to get file status");
        return;
    }

    printf("\nPermissions for %s: ", filename);
    printf((S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf((fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf((fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf((fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf((fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf((fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf((fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf((fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf((fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf((fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n");
}

int main() {
    const char *filename = "testfile.txt";

    // Create a file
    int fd = open(filename, O_CREAT | O_WRONLY, 0644);
    if (fd < 0) {
        perror("File creation failed");
        return 1;
    }
```

```c
    close(fd);

    printf("File %s created successfully.\n", filename);
    display_permissions(filename);

    // Change permissions to 755 (rwxr-xr-x)
    if (chmod(filename, 0755) < 0) {
        perror("chmod failed");
        return 1;
    }
    printf("\nPermissions changed to 755 (rwxr-xr-x).\n");
    display_permissions(filename);

    // Change ownership (Requires root privileges)
    // if (chown(filename, 1000, 1000) < 0) {
    //     perror("chown failed");
    // } else {
    //     printf("\nOwnership changed successfully.\n");
    // }

    return 0;
}
```

Output:

```
File testfile.txt created successfully.

Permissions for testfile.txt: -rw-r--r--

Permissions changed to 755 (rwxr-xr-x).

Permissions for testfile.txt: -rwxr-xr-x
```