Practical No 4

FCFS Scheduling Algorithm:

```
#include <stdio.h>
typedef struct {
    int id;
    int burst_time;
} Process;

void findWaitingTime(Process processes[], int n, int waiting_time[]) {
    waiting_time[0] = 0;

    for (int i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
    }
}
void findTurnAroundTime(Process processes[], int n, int waiting_time[], int turn_around_time[]) {
    for (int i = 0; i < n; i++) {
        turn_around_time[i] = waiting_time[i] + processes[i].burst_time;
    }
}
void findavgTime(Process processes[], int n) {
    int waiting_time[n], turn_around_time[n];

    findWaitingTime(processes, n, waiting_time);
    findTurnAroundTime(processes, n, waiting_time, turn_around_time);

    printf("Process\tBurst Time\tWaiting Time\tTurn-Around Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, waiting_time[i],
turn_around_time[i]);
    }
```

```c
}

int main() {
   Process processes[] = {{1, 10}, {2, 5}, {3, 8}};
   int n = sizeof(processes) / sizeof(processes[0]);

   findavgTime(processes, n);
   return 0;
}
```

SJF Scheduling Algorithm:

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct {
   int id;
   int burst_time;
} Process;

int compare(const void* a, const void* b) {
   return ((Process*)a)->burst_time - ((Process*)b)->burst_time;
}

void findWaitingTime(Process processes[], int n, int waiting_time[]) {
   waiting_time[0] = 0;

   for (int i = 1; i < n; i++) {
      waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
   }
}
void findTurnAroundTime(Process processes[], int n, int waiting_time[], int turn_around_time[]) {
   for (int i = 0; i < n; i++) {
```

```
        turn_around_time[i] = waiting_time[i] + processes[i].burst_time;
    }
}

void findavgTime(Process processes[], int n) {
    int waiting_time[n], turn_around_time[n];

    findWaitingTime(processes, n, waiting_time);
    findTurnAroundTime(processes, n, waiting_time, turn_around_time);
    printf("Process\tBurst Time\tWaiting Time\tTurn-Around Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, waiting_time[i],
turn_around_time[i]);
    }
}

int main() {
    Process processes[] = {{1, 6}, {2, 8}, {3, 7}, {4, 3}};
    int n = sizeof(processes) / sizeof(processes[0]);
    qsort(processes, n, sizeof(Process), compare);
    findavgTime(processes, n);
    return 0;
}
```

Ronud Robin Scheduling Algorithm:

```
#include <stdio.h>
typedef struct {
    int id;
    int burst_time;
    int remaining_time;
```

```
} Process;

void findavgTime(Process processes[], int n, int quantum) {
    int waiting_time[n], turn_around_time[n];
    int remaining_processes = n;
    int time = 0;

    while (remaining_processes > 0) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                if (processes[i].remaining_time > quantum) {
                    time += quantum;
                    processes[i].remaining_time -= quantum;
                } else {
                    time += processes[i].remaining_time;
                    waiting_time[i] = time - processes[i].burst_time;
                    processes[i].remaining_time = 0;
                    remaining_processes--;
                }
            }
        }
    }

    for (int i = 0; i < n; i++) {
        turn_around_time[i] = processes[i].burst_time + waiting_time[i];
    }

    printf("Process\tBurst Time\tWaiting Time\tTurn-Around Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, waiting_time[i],
turn_around_time[i]);
```

```
    }
}

int main() {
    Process processes[] = {{1, 10}, {2, 5}, {3, 8}};
    int n = sizeof(processes) / sizeof(processes[0]);
    for (int i = 0; i < n; i++) {
        processes[i].remaining_time = processes[i].burst_time;
    }
    int quantum = 4; // Time quantum
    findavgTime(processes, n, quantum);
    return 0;
}
```

Practical No : 5

Implement multithreading for Matrix Operations using Pthreads without pointer:

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

#define MAX 10  // Maximum size for the matrix

// Struct to hold the arguments for each thread
struct ThreadData {
    int row;      // Row index to compute
    int cols;     // Number of columns in the matrix
};

// Matrices
```

```c
int A[MAX][MAX];
int B[MAX][MAX];
int C[MAX][MAX];
int num_rows;

// Function to add matrices
void* add_matrices(void* arg) {
   struct ThreadData* data = (struct ThreadData*)arg;
   int row = data->row;
   int cols = data->cols;

   for (int j = 0; j < cols; j++) {
      C[row][j] = A[row][j] + B[row][j];
   }
   return NULL;
}

int main() {
   // Initialize matrices A and B
   printf("Enter the number of rows and columns: ");
   scanf("%d", &num_rows);
   int cols = num_rows; // For simplicity, let's keep it square

   printf("Enter elements of Matrix A:\n");
   for (int i = 0; i < num_rows; i++) {
      for (int j = 0; j < cols; j++) {
         scanf("%d", &A[i][j]);
      }
   }

   printf("Enter elements of Matrix B:\n");
```

```
for (int i = 0; i < num_rows; i++) {
    for (int j = 0; j < cols; j++) {
        scanf("%d", &B[i][j]);
    }
}

pthread_t threads[MAX];
struct ThreadData thread_data[MAX];

// Create threads to add matrices
for (int i = 0; i < num_rows; i++) {
    thread_data[i].row = i;
    thread_data[i].cols = cols;
    pthread_create(&threads[i], NULL, add_matrices, &thread_data[i]);
}

// Wait for all threads to finish
for (int i = 0; i < num_rows; i++) {
    pthread_join(threads[i], NULL);
}

// Print the resulting matrix C
printf("Resulting Matrix C (A + B):\n");
for (int i = 0; i < num_rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("%d ", C[i][j]);
    }
    printf("\n");
}

return 0;
```

```
}
```

Practical No : 06

```
//Producer Consumer - p7
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5
#define NUM_ITEMS 10

int buffer[BUFFER_SIZE], in = 0, out = 0;
sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
```

```
    for (int i = 0; i < NUM_ITEMS; i++) {
        int item = rand() % 100;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        printf("Produced: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
        sleep(rand() % 2);
    }
    return NULL;
}

void* consumer(void* arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        sleep(rand() % 2);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;
```

```
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Practical No 7 :

Bankers algorithm

```
#include <stdio.h>

#define P 5

#define R 3

int isSafe(int processes[], int avail[], int max[][R], int alloc[][R]) {

    int need[P][R], finish[P], safeSeq[P], work[R];

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

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

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

        }

    }

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

        work[i] = avail[i];

    }

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

        finish[i] = 0;

    }

    int count = 0;

    while (count < P) {
```

```
        int found = 0;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int canProceed = 1;
                for (int j = 0; j < R; j++) {
                    if (need[p][j] > work[j]) {
                        canProceed = 0;
                        break;
                    }
                }

                if (canProceed) {
                    for (int k = 0; k < R; k++) {
                        work[k] += alloc[p][k];
                    }
                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
            printf("The system is not in a safe state.\n");
            return 0;
        }
    }
    printf("The system is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf("\n");
```

```
    return 1;
}

int main() {
    int processes[] = {0, 1, 2, 3, 4};
    int avail[] = {3, 3, 2};
    int max[P][R] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };
    int alloc[P][R] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };
    isSafe(processes, avail, max, alloc);
    return 0;
}
```

Practical No 8 :

//Best , Worst , First FITs

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

#define MAX_BLOCKS 100
#define MAX_PROCESSES 100
```

```
void worstFit(int blockSize[], int numBlocks, int processSize[], int numProcesses) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < numProcesses; i++) allocation[i] = -1;

    for (int i = 0; i < numProcesses; i++) {
        int largestBlockIndex = -1;
        for (int j = 0; j < numBlocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (largestBlockIndex == -1 || blockSize[largestBlockIndex] < blockSize[j])
                    largestBlockIndex = j;
            }
        }
        if (largestBlockIndex != -1) {
            allocation[i] = largestBlockIndex;
            blockSize[largestBlockIndex] -= processSize[i];
        }
    }

    printf("Worst Fit Allocation:\n");
    for (int i = 0; i < numProcesses; i++)
        printf("Process %d allocated to Block %d\n", i, allocation[i]);
}

void bestFit(int blockSize[], int numBlocks, int processSize[], int numProcesses) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < numProcesses; i++) allocation[i] = -1;

    for (int i = 0; i < numProcesses; i++) {
        int bestBlockIndex = -1;
        for (int j = 0; j < numBlocks; j++) {
```

```
            if (blockSize[j] >= processSize[i]) {
                if (bestBlockIndex == -1 || blockSize[bestBlockIndex] > blockSize[j])
                    bestBlockIndex = j;
            }
        }
        if (bestBlockIndex != -1) {
            allocation[i] = bestBlockIndex;
            blockSize[bestBlockIndex] -= processSize[i];
        }
    }

    printf("Best Fit Allocation:\n");
    for (int i = 0; i < numProcesses; i++)
        printf("Process %d allocated to Block %d\n", i, allocation[i]);
}

void firstFit(int blockSize[], int numBlocks, int processSize[], int numProcesses) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < numProcesses; i++) allocation[i] = -1;

    for (int i = 0; i < numProcesses; i++) {
        for (int j = 0; j < numBlocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }

    printf("First Fit Allocation:\n");
```

```
    for (int i = 0; i < numProcesses; i++)
        printf("Process %d allocated to Block %d\n", i, allocation[i]);
}

int main() {
    int blockSize[MAX_BLOCKS], processSize[MAX_PROCESSES];
    int numBlocks, numProcesses;

    printf("Enter the number of memory blocks: ");
    scanf("%d", &numBlocks);
    printf("Enter the sizes of memory blocks:\n");
    for (int i = 0; i < numBlocks; i++) {
        printf("Block %d: ", i);
        scanf("%d", &blockSize[i]);
    }

    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    printf("Enter the sizes of processes:\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("Process %d: ", i);
        scanf("%d", &processSize[i]);
    }

    worstFit(blockSize, numBlocks, processSize, numProcesses);
    printf("\n");
    bestFit(blockSize, numBlocks, processSize, numProcesses);
    printf("\n");
    firstFit(blockSize, numBlocks, processSize, numProcesses);

    return 0;
```

```
}
```

Practical No : 9

//FCFS and SCAN

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_REQUESTS 100

void fcfs(int requests[], int n, int head) {
   int total_movement = 0, current = head;
   printf("FCFS Scheduling Order:\n");
   for (int i = 0; i < n; i++) {
      printf("Move from %d to %d\n", current, requests[i]);
      total_movement += abs(current - requests[i]);
      current = requests[i];
   }
   printf("Total head movement: %d\n", total_movement);
}

void scan(int requests[], int n, int head) {
   int total_movement = 0, current = head;
   int sorted_requests[MAX_REQUESTS];
```

```
for (int i = 0; i < n; i++) sorted_requests[i] = requests[i];

// Bubble sort
for (int i = 0; i < n - 1; i++)
    for (int j = 0; j < n - i - 1; j++)
        if (sorted_requests[j] > sorted_requests[j + 1]) {
            int temp = sorted_requests[j];
            sorted_requests[j] = sorted_requests[j + 1];
            sorted_requests[j + 1] = temp;
        }

printf("SCAN Scheduling Order:\n");
for (int i = 0; i < n; i++) {
    if (sorted_requests[i] >= current) {
        printf("Move from %d to %d\n", current, sorted_requests[i]);
        total_movement += abs(current - sorted_requests[i]);
        current = sorted_requests[i];
    }
}

printf("Move from %d to %d\n", current, sorted_requests[n - 1]);
total_movement += abs(current - sorted_requests[n - 1]);
current = sorted_requests[n - 1];

for (int i = n - 1; i >= 0; i--) {
    if (sorted_requests[i] <= current) {
        printf("Move from %d to %d\n", current, sorted_requests[i]);
        total_movement += abs(current - sorted_requests[i]);
        current = sorted_requests[i];
    }
```

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

int main() {
    int requests[MAX_REQUESTS], n, head;
    printf("Enter the number of requests: ");
    scanf("%d", &n);
    if (n > MAX_REQUESTS) return 1;

    printf("Enter the requests:\n");
    for (int i = 0; i < n; i++) scanf("%d", &requests[i]);
    printf("Enter the initial head position: ");
    scanf("%d", &head);

    fcfs(requests, n, head);
    printf("\n");
    scan(requests, n, head);

    return 0;
}
```