```
//Dinning Philosopher
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5

sem_t chopstick[N];
sem_t room;

void* philosopher(void* num) {
    int id = *(int*)num;
    while (1) {
        printf("Philosopher %d is thinking...\n", id);
        sleep(rand() % 3 + 1);
        sem_wait(&room);
        sem_wait(&chopstick[id]);
        sem_wait(&chopstick[(id + 1) % N]);
        printf("Philosopher %d is eating...\n", id);
        sleep(rand() % 2 + 1);
        sem_post(&chopstick[id]);
        sem_post(&chopstick[(id + 1) % N]);
        sem_post(&room);
    }
    return NULL;
}

int main() {
    pthread_t tid[N];
    int i;
    int philosopher_ids[N];
    for (i = 0; i < N; i++) {
        sem_init(&chopstick[i], 0, 1);
    }
    sem_init(&room, 0, N - 1);
    for (i = 0; i < N; i++) {
        philosopher_ids[i] = i;
        pthread_create(&tid[i], NULL, philosopher, &philosopher_ids[i]);
    }
    for (i = 0; i < N; i++) {
        pthread_join(tid[i], NULL);
    }
    return 0;
}

//Deadlock Avoidance
#include <stdio.h>
#include <stdbool.h>

#define P 5  // Number of processes
#define R 3  // Number of resources

int main() {
```

```
int alloc[P][R] = {
    {0, 1, 0},
    {2, 0, 0},
    {3, 0, 2},
    {2, 1, 1},
    {0, 0, 2}
};

int max[P][R] = {
    {7, 5, 3},
    {3, 2, 2},
    {9, 0, 2},
    {2, 2, 2},
    {4, 3, 3}
};

int avail[R] = {3, 3, 2};
int need[P][R];

// Calculate need matrix
for (int i = 0; i < P; i++)
    for (int j = 0; j < R; j++)
        need[i][j] = max[i][j] - alloc[i][j];

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

int count = 0;
while (count < P) {
    bool found = false;
    for (int p = 0; p < P; p++) {
        if (!finish[p]) {
            bool canProceed = true;
            for (int j = 0; j < R; j++) {
                if (need[p][j] > work[j]) {
                    canProceed = false;
                    break;
                }
            }

            if (canProceed) {
                for (int k = 0; k < R; k++)
                    work[k] += alloc[p][k];

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

    if (!found) {
```

```
            printf("System is not in a safe state (deadlock may occur)\n");
            return 0;
        }
    }

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

    return 0;
}

//Deadlock Detection
#include <stdio.h>
#include <stdbool.h>

#define P 5  // Processes
#define R 3  // Resources

int main() {
    int alloc[P][R] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 3},
        {2, 1, 1},
        {0, 0, 2}
    };

    int req[P][R] = {
        {0, 0, 0},
        {2, 0, 2},
        {0, 0, 0},
        {1, 0, 0},
        {0, 0, 2}
    };

    int avail[R] = {0, 0, 0};

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

    bool deadlock = false;
    for (int count = 0; count < P; count++) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (!finish[p]) {
                bool canRun = true;
                for (int j = 0; j < R; j++) {
                    if (req[p][j] > work[j]) {
                        canRun = false;
                        break;
                    }
```

```
            }

            if (canRun) {
                for (int k = 0; k < R; k++)
                    work[k] += alloc[p][k];

                finish[p] = true;
                found = true;
            }
        }
    }

    if (!found)
        break;
}

printf("Processes in Deadlock: ");
for (int i = 0; i < P; i++) {
    if (!finish[i]) {
        printf("P%d ", i);
        deadlock = true;
    }
}

if (!deadlock)
    printf("None (No Deadlock Detected)");

printf("\n");

return 0;
}

//FCFS
#include <stdio.h>

typedef struct {
    int pid, at, bt, ct, wt, tat;
} Process;

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

    Process p[n];

    printf("Enter Arrival Time and Burst Time:\n");
    for (i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("P%d (AT BT): ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
    }

    // Sort by Arrival Time
    for (i = 0; i < n - 1; i++)
```

```
        for (int j = i + 1; j < n; j++)
            if (p[j].at < p[i].at) {
                Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }

    int time = 0;
    float total_wt = 0, total_tat = 0;

    printf("\nGantt Chart:\n|");
    for (i = 0; i < n; i++) {
        if (time < p[i].at) time = p[i].at;

        p[i].wt = time - p[i].at;
        p[i].ct = time + p[i].bt;
        p[i].tat = p[i].wt + p[i].bt;
        time += p[i].bt;

        printf(" P%d |", p[i].pid);
    }

    printf("\n0");
    time = 0;
    for (i = 0; i < n; i++) {
        if (time < p[i].at) time = p[i].at;
        time += p[i].bt;
        printf("  %d", time);
    }

    printf("\n\nProcess\tAT\tBT\tWT\tTAT\tCT\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].wt, p[i].tat, p[i].ct);
        total_wt += p[i].wt;
        total_tat += p[i].tat;
    }

    printf("\nAverage Waiting Time = %.2f", total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", total_tat / n);

    return 0;
}

//SJF
#include <stdio.h>
#include <stdbool.h>

typedef struct {
    int pid, at, bt, ct, wt, tat;
    bool completed;
} Process;

int main() {
    int n;
    printf("Enter number of processes: ");
```

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

Process p[n];
printf("Enter Arrival Time and Burst Time:\n");
for (int i = 0; i < n; i++) {
    p[i].pid = i + 1;
    printf("P%d (AT BT): ", i + 1);
    scanf("%d %d", &p[i].at, &p[i].bt);
    p[i].completed = false;
}

int time = 0, completed = 0;
float total_wt = 0, total_tat = 0;

int gantt_order[n];   // store order of execution
int gantt_time[n + 1]; // store start and end times
gantt_time[0] = 0;

printf("\nGantt Chart:\n|");

while (completed < n) {
    int idx = -1;
    int min_bt = 1e9;

    for (int i = 0; i < n; i++) {
        if (!p[i].completed && p[i].at <= time && p[i].bt < min_bt) {
            min_bt = p[i].bt;
            idx = i;
        }
    }

    if (idx != -1) {
        if (time < p[idx].at)
            time = p[idx].at;

        p[idx].wt = time - p[idx].at;
        time += p[idx].bt;
        p[idx].ct = time;
        p[idx].tat = p[idx].ct - p[idx].at;
        p[idx].completed = true;

        total_wt += p[idx].wt;
        total_tat += p[idx].tat;

        gantt_order[completed] = p[idx].pid;
        gantt_time[completed + 1] = time;

        printf(" P%d |", p[idx].pid);
        completed++;
    } else {
        time++; // no process is available, move time forward
    }
}

// Print timeline
```

```
    printf("\n%d", gantt_time[0]);
    for (int i = 1; i <= n; i++) {
        printf("   %d", gantt_time[i]);
    }

    // Results
    printf("\n\nProcess\tAT\tBT\tWT\tTAT\tCT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].wt, p[i].tat, p[i].ct);
    }

    printf("\nAverage Waiting Time = %.2f", total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", total_tat / n);

    return 0;
}

//Round Robin
#include <stdio.h>
#include <stdbool.h>

typedef struct {
    int pid, at, bt, rt, ct, wt, tat;
    bool is_completed;
} Process;

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

    Process p[n];
    printf("Enter Arrival Time and Burst Time:\n");
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("P%d (AT BT): ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
        p[i].rt = p[i].bt;
        p[i].is_completed = false;
    }

    printf("Enter Time Quantum: ");
    scanf("%d", &tq);

    int time = 0, completed = 0;
    int queue[100], front = 0, rear = 0;
    bool in_queue[n];
    for (int i = 0; i < n; i++) in_queue[i] = false;

    printf("\nGantt Chart:\n|");

    while (completed < n) {
        // Add newly arrived processes to queue
        for (int i = 0; i < n; i++) {
```

```
            if (p[i].at <= time && p[i].rt > 0 && !in_queue[i]) {
                queue[rear++] = i;
                in_queue[i] = true;
            }
        }

        if (front == rear) {
            printf(" IDLE |");
            time++;
            continue;
        }

        int idx = queue[front++];
        in_queue[idx] = false;

        int exec_time = (p[idx].rt > tq) ? tq : p[idx].rt;
        printf(" P%d |", p[idx].pid);
        int start_time = time;
        time += exec_time;
        p[idx].rt -= exec_time;

        // Enqueue any newly arrived processes during this slice
        for (int i = 0; i < n; i++) {
            if (p[i].at > start_time && p[i].at <= time && p[i].rt > 0 && !in_queue[i]) {
                queue[rear++] = i;
                in_queue[i] = true;
            }
        }

        if (p[idx].rt == 0 && !p[idx].is_completed) {
            p[idx].ct = time;
            p[idx].tat = p[idx].ct - p[idx].at;
            p[idx].wt = p[idx].tat - p[idx].bt;
            p[idx].is_completed = true;
            completed++;
        } else {
            queue[rear++] = idx;
            in_queue[idx] = true;
        }
    }

    // Output
    printf("\n\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
    float total_tat = 0, total_wt = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);
        total_tat += p[i].tat;
        total_wt += p[i].wt;
    }

    printf("\nAverage Waiting Time = %.2f", total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", total_tat / n);
    return 0;
}
```

```c
//SRTF
#include <stdio.h>
#include <limits.h>

typedef struct {
    int pid, at, bt, rt, ct, wt, tat, start;
    int completed;
} Process;

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

    Process p[n];
    printf("Enter Arrival Time and Burst Time:\n");
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("P%d (AT BT): ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
        p[i].rt = p[i].bt;
        p[i].completed = 0;
        p[i].start = -1;
    }

    int time = 0, completed = 0;
    printf("\nGantt Chart:\n|");

    while (completed < n) {
        int idx = -1, min_rt = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && !p[i].completed && p[i].rt < min_rt && p[i].rt > 0) {
                min_rt = p[i].rt;
                idx = i;
            }
        }

        if (idx != -1) {
            if (p[idx].start == -1)
                p[idx].start = time;

            printf(" P%d |", p[idx].pid);
            p[idx].rt--;
            time++;

            if (p[idx].rt == 0) {
                p[idx].ct = time;
                p[idx].tat = p[idx].ct - p[idx].at;
                p[idx].wt = p[idx].tat - p[idx].bt;
                p[idx].completed = 1;
                completed++;
            }
```

```
        } else {
            time++; // CPU idle
        }
    }

    printf("\n\nProcess\tAT\tBT\tWT\tTAT\tCT\n");
    float total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].wt, p[i].tat, p[i].ct);
        total_wt += p[i].wt;
        total_tat += p[i].tat;
    }

    printf("\nAverage Waiting Time = %.2f", total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", total_tat / n);
    return 0;
}

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

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 3

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    int id = *(int*)arg;
    int item;
    while (1) {
        item = rand() % 100;
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        printf("Producer %d produced: %d\n", id, item);
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
        sleep(rand() % 2 + 1);
    }
    return NULL;
}

void* consumer(void* arg) {
    int item;
```

```
    while (1) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        sleep(rand() % 3 + 1);
    }
    return NULL;
}

int main() {
    pthread_t prod_threads[NUM_PRODUCERS], cons_thread;
    int producer_ids[NUM_PRODUCERS];
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producer_ids[i] = i + 1;
        pthread_create(&prod_threads[i], NULL, producer, &producer_ids[i]);
    }
    pthread_create(&cons_thread, NULL, consumer, NULL);
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(prod_threads[i], NULL);
    }
    pthread_join(cons_thread, NULL);
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}

//Pipe() and Fork()
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2];
    pid_t pid;
    char write_msg[] = "Hello from parent to child via pipe!";
    char read_msg[100];
    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }
    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }
    if (pid > 0) {
```

```
        close(fd[0]);
        write(fd[1], write_msg, strlen(write_msg) + 1);
        printf("Parent: Sent message to child.\n");
        close(fd[1]);
    } else {
        close(fd[1]);
        read(fd[0], read_msg, sizeof(read_msg));
        printf("Child: Received message: '%s'\n", read_msg);
        close(fd[0]);
    }
    return 0;
}

//Shared Memory
// server.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

#define SHM_SIZE 1024

int main() {
    key_t key = ftok("shmfile", 65);
    if (key == -1) {
        perror("ftok");
        exit(1);
    }
    int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }
    char *data = (char *)shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }
    printf("Writing to shared memory...\n");
    strcpy(data, "Hello from Server!");
    printf("Server wrote: %s\n", data);
    sleep(30);
    shmdt(data);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}

//Client
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#include <string.h>

#define SHM_SIZE 1024

int main() {
    key_t key = ftok("shmfile", 65);
    if (key == -1) {
        perror("ftok");
        exit(1);
    }
    int shmid = shmget(key, SHM_SIZE, 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }
    char *data = (char *)shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }
    printf("Client read: %s\n", data);
    shmdt(data);
    return 0;
}

//Message Queue
//Server
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

#define MAX 100

struct msg_buffer {
    long msg_type;
    char msg_text[MAX];
};

int main() {
    key_t key = ftok("msgfile", 65);
    if (key == -1) {
        perror("ftok");
        exit(1);
    }
    int msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }
    struct msg_buffer message;
    printf("Server waiting for message...\n");
    if (msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0) == -1) {
        perror("msgrcv");
```

```
        exit(1);
    }
    printf("Received message: %s\n", message.msg_text);
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}

//Client
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

#define MAX 100

struct msg_buffer {
    long msg_type;
    char msg_text[MAX];
};

int main() {
    key_t key = ftok("msgfile", 65);
    if (key == -1) {
        perror("ftok");
        exit(1);
    }
    int msgid = msgget(key, 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }
    struct msg_buffer message;
    message.msg_type = 1;  // must be > 0
    strcpy(message.msg_text, "Hello from Client!");
    if (msgsnd(msgid, &message, sizeof(message.msg_text), 0) == -1) {
        perror("msgsnd");
        exit(1);
    }
    printf("Message sent: %s\n", message.msg_text);
    return 0;
}

//Reader and Writer
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

#define MAX_WRITES 4
#define MAX_READS 2
#define BUFFER_SIZE 5

char buffer[BUFFER_SIZE][100];
```

```
int write_count = 0, read_count = 0;
int write_index = 0, read_index = 0;

int writer_phase = 1;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_writer = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_reader = PTHREAD_COND_INITIALIZER;

void* writer(void* arg) {
    int id = *(int *)arg;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (!writer_phase || write_count >= MAX_WRITES) {
            pthread_cond_wait(&cond_writer, &mutex);
        }
        snprintf(buffer[write_index], sizeof(buffer[write_index]),
                 "Writer %d wrote message %d", id, write_count + 1);
        printf("Writer %d: %s\n", id, buffer[write_index]);
        write_index = (write_index + 1) % BUFFER_SIZE;
        write_count++;
        if (write_count >= MAX_WRITES) {
            writer_phase = 0;
            pthread_cond_broadcast(&cond_reader);
        }
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}

void* reader(void* arg) {
    int id = *(int *)arg;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (writer_phase || read_count >= write_count) {
            pthread_cond_wait(&cond_reader, &mutex);
        }
        printf("Reader %d: Read => %s\n", id, buffer[read_index]);
        read_index = (read_index + 1) % BUFFER_SIZE;
        read_count++;
        if (read_count >= write_count) {
            // Reset counters
            write_count = 0;
            read_count = 0;
            writer_phase = 1;
            pthread_cond_broadcast(&cond_writer);
        }
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}

int main() {
```

```
    pthread_t writers[4], readers[2];
    int ids[4] = {1, 2, 3, 4};
    for (int i = 0; i < 4; i++) {
        pthread_create(&writers[i], NULL, writer, &ids[i]);
    }
    for (int i = 0; i < 2; i++) {
        pthread_create(&readers[i], NULL, reader, &ids[i]);
    }
    for (int i = 0; i < 4; i++) {
        pthread_join(writers[i], NULL);
    }
    for (int i = 0; i < 2; i++) {
        pthread_join(readers[i], NULL);
    }
    return 0;
}

//Address translation Under Paging
#include <stdio.h>
#include <stdlib.h>

#define PAGE_SIZE 1024  // 1 KB pages
#define NUM_PAGES 4
#define NUM_FRAMES 8

int main() {
    int page_table[NUM_PAGES] = {3, 1, 4, 7};  // page to frame mapping
    int logical_address;
    printf("Enter a logical address (0 to %d): ", PAGE_SIZE * NUM_PAGES - 1);
    scanf("%d", &logical_address);
    int page_number = logical_address / PAGE_SIZE;
    int offset = logical_address % PAGE_SIZE;
    if (page_number >= NUM_PAGES) {
        printf("Invalid page number!\n");
        return 1;
    }
    int frame_number = page_table[page_number];
    int physical_address = frame_number * PAGE_SIZE + offset;
    printf("Logical Address: %d\n", logical_address);
    printf("Page Number: %d, Offset: %d\n", page_number, offset);
    printf("Frame Number: %d\n", frame_number);
    printf("Physical Address: %d\n", physical_address);
    return 0;
}

//Pagereplacement
//FCFS
#include <stdio.h>

int main() {
    int frames[10], pages[50], n, f, i, j, pos = 0, page_faults = 0;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter the page reference string: ");
    for (i = 0; i < n; i++)
```

```
        scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
    scanf("%d", &f);
    for (i = 0; i < f; i++) frames[i] = -1;
    for (i = 0; i < n; i++) {
        int found = 0;
        for (j = 0; j < f; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            frames[pos] = pages[i];
            pos = (pos + 1) % f;
            page_faults++;
            printf("After inserting %d: ", pages[i]);
            for (j = 0; j < f; j++)
                if (frames[j] != -1)
                    printf("%d ", frames[j]);
            printf("\n");
        }
    }
    printf("Total Page Faults (FCFS): %d\n", page_faults);
    return 0;
}

//LRU
#include <stdio.h>

int main() {
    int frames[10], pages[50], time[10], n, f, i, j, counter = 0, page_faults = 0;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter the page reference string: ");
    for (i = 0; i < n; i++) scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
    scanf("%d", &f);
    for (i = 0; i < f; i++) {
        frames[i] = -1;
        time[i] = 0;
    }
    for (i = 0; i < n; i++) {
        int found = 0;
        for (j = 0; j < f; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                time[j] = ++counter;
                break;
            }
        }
        if (!found) {
            int lru = 0;
            for (j = 1; j < f; j++) {
                if (time[j] < time[lru])
```

```
                lru = j;
            }
            frames[lru] = pages[i];
            time[lru] = ++counter;
            page_faults++;
            printf("After inserting %d: ", pages[i]);
            for (j = 0; j < f; j++)
                if (frames[j] != -1)
                    printf("%d ", frames[j]);
            printf("\n");
        }
    }
    printf("Total Page Faults (LRU): %d\n", page_faults);
    return 0;
}

//Optimal
#include <stdio.h>

int predict(int pages[], int frames[], int n, int index, int f) {
    int farthest = index, res = -1;
    for (int i = 0; i < f; i++) {
        int j;
        for (j = index; j < n; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    res = i;
                }
                break;
            }
        }
        if (j == n) return i;
    }
    return (res == -1) ? 0 : res;
}

int main() {
    int frames[10], pages[50], n, f, i, j, page_faults = 0, filled = 0;
    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &n);
    printf("Enter the page reference string (space-separated): ");
    for (i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }
    printf("Enter number of frames: ");
    scanf("%d", &f);
    for (i = 0; i < f; i++) frames[i] = -1;
    for (i = 0; i < n; i++) {
        int found = 0;
        for (j = 0; j < f; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
```

```
        }
        if (!found) {
            if (filled < f) {
                frames[filled++] = pages[i];
            } else {
                int idx = predict(pages, frames, n, i + 1, f);
                frames[idx] = pages[i];
            }
            page_faults++;
            printf("After inserting %d: ", pages[i]);
            for (j = 0; j < f; j++) {
                if (frames[j] != -1)
                    printf("%d ", frames[j]);
                else
                    printf("- ");
            }
            printf("\n");
        }
    }
    printf("Total Page Faults (Optimal): %d\n", page_faults);
    return 0;
}

//Disk Scheduling
//FCFS
#include<stdio.h>
#include<stdlib.h>
int main() {
    int requests[50], n, i, start, total_seek_time = 0;
    printf("Enter number of disk requests: ");
    scanf("%d", &n);
    printf("Enter the disk requests: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial position of the disk head: ");
    scanf("%d", &start);
    for (i = 0; i < n; i++) {
        total_seek_time += abs(requests[i] - start);
        start = requests[i];
    }
    printf("Total Seek Time (FCFS): %d\n", total_seek_time);
    return 0;
}

//SSTF
#include <stdio.h>
#include <stdlib.h>

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

```
int main() {
    int requests[50], n, start, i, j, total_seek_time = 0;
    printf("Enter number of disk requests: ");
    scanf("%d", &n);
    printf("Enter the disk requests: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial position of the disk head: ");
    scanf("%d", &start);
    for (i = 0; i < n-1; i++) {
        for (j = i + 1; j < n; j++) {
            if (requests[i] > requests[j]) {
                swap(&requests[i], &requests[j]);
            }
        }
    }
    int closest;
    while (n > 0) {
        closest = -1;
        int min_diff = 99999;
        for (i = 0; i < n; i++) {
            if (abs(requests[i] - start) < min_diff) {
                closest = i;
                min_diff = abs(requests[i] - start);
            }
        }
        total_seek_time += min_diff;
        start = requests[closest];
        for (i = closest; i < n - 1; i++) {
            requests[i] = requests[i + 1];
        }
        n--;
    }
    printf("Total Seek Time (SSTF): %d\n", total_seek_time);
    return 0;
}

//SCAN
#include <stdio.h>
#include<stdlib.h>
int main() {
    int requests[50], n, start, direction, i, total_seek_time = 0;
    printf("Enter number of disk requests: ");
    scanf("%d", &n);
    printf("Enter the disk requests: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial position of the disk head: ");
    scanf("%d", &start);
    printf("Enter the direction of the disk head (1 for right, 0 for left): ");
    scanf("%d", &direction);
    for (i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
```

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

        for (i = n-2; i >= 0; i--) {
            total_seek_time += abs(requests[i] - start);
            start = requests[i];
        }
    } else {
        for (i = n-1; i >= 0; i--) {
            if (requests[i] <= start) {
                total_seek_time += abs(requests[i] - start);
                start = requests[i];
            }
        }
        total_seek_time += abs(requests[0] - start);
        start = requests[0];

        for (i = 1; i < n; i++) {
            total_seek_time += abs(requests[i] - start);
            start = requests[i];
        }
    }
    printf("Total Seek Time (SCAN): %d\n", total_seek_time);
    return 0;
}

//LOOK
#include <stdio.h>
#include<stdlib.h>

int main() {
    int requests[50], n, start, direction, i, j, total_seek_time = 0;
    printf("Enter number of disk requests: ");
    scanf("%d", &n);
    printf("Enter the disk requests: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial position of the disk head: ");
    scanf("%d", &start);
    printf("Enter the direction of the disk head (1 for right, 0 for left): ");
```

```
    scanf("%d", &direction);
    for (i = 0; i < n-1; i++) {
        for (j = i+1; j < n; j++) {
            if (requests[i] > requests[j]) {
                int temp = requests[i];
                requests[i] = requests[j];
                requests[j] = temp;
            }
        }
    }
    if (direction == 1) {
        for (i = 0; i < n; i++) {
            if (requests[i] >= start) {
                total_seek_time += abs(requests[i] - start);
                start = requests[i];
            }
        }

        for (i = n-1; i >= 0; i--) {
            total_seek_time += abs(requests[i] - start);
            start = requests[i];
        }
    } else {
        for (i = n-1; i >= 0; i--) {
            if (requests[i] <= start) {
                total_seek_time += abs(requests[i] - start);
                start = requests[i];
            }
        }
        for (i = 0; i < n; i++) {
            total_seek_time += abs(requests[i] - start);
            start = requests[i];
        }
    }
    printf("Total Seek Time (LOOK): %d\n", total_seek_time);
    return 0;
}
```