**Experiment 1**

**Aim:** Process creation and termination for operating system (fork, wait, signal, exit, etc.) in C

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/wait.h>

void child\_process(int pid) {

printf("Child process with PID %d is created\n", pid);

sleep(5);

printf("Child process with PID %d is terminating\n", pid);

exit(0);

}

void parent\_process(int pid) {

int status;

printf("Parent process with PID %d is created\n", pid);

printf("Waiting for child process to terminate...\n");

wait(&status);

if (WIFEXITED(status)) {

printf("Child process terminated successfully with status code %d\n", WEXITSTATUS(status));

}

printf("Parent process with PID %d is terminating\n", pid);

}

int main() {

pid\_t pid = fork();

if (pid == 0) {

child\_process(getpid());

} else if (pid > 0) {

parent\_process(getpid());

} else {

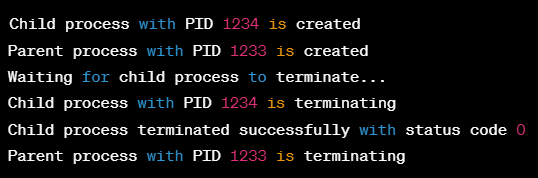
printf("Error: Failed to create child process\n");

}

return 0;

}

**Output:**

****

**Experiment 2**

**Aim:** Threads in C

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 2

void \*thread\_function(void \*arg) {

int thread\_id = \*((int \*) arg);

printf("Thread %d started\n", thread\_id);

for (int i = 1; i <= 5; i++) {

printf("Thread %d: iteration %d\n", thread\_id, i);

sleep(1);

}

printf("Thread %d finished\n", thread\_id);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int thread\_ids[NUM\_THREADS];

int result;

for (int i = 0; i < NUM\_THREADS; i++) {

thread\_ids[i] = i + 1;

result = pthread\_create(&threads[i], NULL, thread\_function, &thread\_ids[i]);

if (result != 0) {

printf("Error: Failed to create thread %d\n", i+1);

exit(-1);

}

}

for (int i = 0; i < NUM\_THREADS; i++) {

result = pthread\_join(threads[i], NULL);

if (result != 0) {

printf("Error: Failed to join thread %d\n", i+1);

exit(-1);

}

}

printf("All threads finished\n");

pthread\_exit(NULL);

}

**Output:**

****

**Experiment 3**

**Aim:** CPU scheduling algorithms: FCFS, SJF, Round Robin, Pre-emptive Priority Scheduling.

**Code (First Come First Serve):**

#include <stdio.h>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

};

void calculate\_waiting\_time\_fcfs(struct Process \*processes, int num\_processes) {

int current\_time = 0;

for (int i = 0; i < num\_processes; i++) {

if (processes[i].arrival\_time > current\_time) {

current\_time = processes[i].arrival\_time;

}

processes[i].waiting\_time = current\_time - processes[i].arrival\_time;

current\_time += processes[i].burst\_time;

}

}

void calculate\_turnaround\_time(struct Process \*processes, int num\_processes) {

for (int i = 0; i < num\_processes; i++) {

processes[i].turnaround\_time = processes[i].burst\_time + processes[i].waiting\_time;

}

}

float calculate\_average\_waiting\_time(struct Process \*processes, int num\_processes) {

int total\_waiting\_time = 0;

for (int i = 0; i < num\_processes; i++) {

total\_waiting\_time += processes[i].waiting\_time;

}

return (float) total\_waiting\_time / num\_processes;

}

int main() {

struct Process processes[] = {

{1, 0, 10},

{2, 5, 4},

{3, 8, 7},

{4, 12, 1},

{5, 14, 5}

};

int num\_processes = sizeof(processes) / sizeof(processes[0]);

calculate\_waiting\_time\_fcfs(processes, num\_processes);

calculate\_turnaround\_time(processes, num\_processes);

float avg\_waiting\_time = calculate\_average\_waiting\_time(processes, num\_processes);

printf("Average waiting time for FCFS: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output (First Come First Serve):**

****

**Code (Shortest Job First):**

#include <stdio.h>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

};

void calculate\_waiting\_time\_sjf(struct Process \*processes, int num\_processes) {

int remaining\_time[num\_processes];

for (int i = 0; i < num\_processes; i++) {

remaining\_time[i] = processes[i].burst\_time;

}

int complete = 0, current\_time = 0, shortest = 0, min\_burst\_time = 99999;

while (complete != num\_processes) {

for (int i = 0; i < num\_processes; i++) {

if (processes[i].arrival\_time <= current\_time && remaining\_time[i] < min\_burst\_time && remaining\_time[i] > 0) {

min\_burst\_time = remaining\_time[i];

shortest = i;

}

}

remaining\_time[shortest]--;

if (remaining\_time[shortest] == 0) {

complete++;

int finish\_time = current\_time + 1;

processes[shortest].waiting\_time = finish\_time - processes[shortest].burst\_time - processes[shortest].arrival\_time;

if (processes[shortest].waiting\_time < 0) {

processes[shortest].waiting\_time = 0;

}

}

current\_time++;

min\_burst\_time = 99999;

}

}

void calculate\_turnaround\_time(struct Process \*processes, int num\_processes) {

for (int i = 0; i < num\_processes; i++) {

processes[i].turnaround\_time = processes[i].burst\_time + processes[i].waiting\_time;

}

}

float calculate\_average\_waiting\_time(struct Process \*processes, int num\_processes) {

int total\_waiting\_time = 0;

for (int i = 0; i < num\_processes; i++) {

total\_waiting\_time += processes[i].waiting\_time;

}

return (float) total\_waiting\_time / num\_processes;

}

int main() {

struct Process processes[] = {

{1, 0, 10},

{2, 5, 4},

{3, 8, 7},

{4, 12, 1},

{5, 14, 5}

};

int num\_processes = sizeof(processes) / sizeof(processes[0]);

calculate\_waiting\_time\_sjf(processes, num\_processes);

calculate\_turnaround\_time(processes, num\_processes);

float avg\_waiting\_time = calculate\_average\_waiting\_time(processes, num\_processes);

printf("Average waiting time for SJF: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output (Shortest Job First):**

****

**Code (Round Robin):**

#include <stdio.h>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

};

void calculate\_waiting\_time\_rr(struct Process \*processes, int num\_processes, int quantum) {

int remaining\_time[num\_processes];

for (int i = 0; i < num\_processes; i++) {

remaining\_time[i] = processes[i].burst\_time;

}

int current\_time = 0;

while (1) {

int complete = 1;

for (int i = 0; i < num\_processes; i++) {

if (remaining\_time[i] > 0) {

complete = 0;

if (remaining\_time[i] > quantum) {

current\_time += quantum;

remaining\_time[i] -= quantum;

} else {

current\_time += remaining\_time[i];

processes[i].waiting\_time = current\_time - processes[i].burst\_time - processes[i].arrival\_time;

if (processes[i].waiting\_time < 0) {

processes[i].waiting\_time = 0;

}

remaining\_time[i] = 0;

}

}

}

if (complete == 1) {

break;

}

}

}

void calculate\_turnaround\_time(struct Process \*processes, int num\_processes) {

for (int i = 0; i < num\_processes; i++) {

processes[i].turnaround\_time = processes[i].burst\_time + processes[i].waiting\_time;

}

}

float calculate\_average\_waiting\_time(struct Process \*processes, int num\_processes) {

int total\_waiting\_time = 0;

for (int i = 0; i < num\_processes; i++) {

total\_waiting\_time += processes[i].waiting\_time;

}

return (float) total\_waiting\_time / num\_processes;

}

int main() {

struct Process processes[] = {

{1, 0, 10},

{2, 5, 4},

{3, 8, 7},

{4, 12, 1},

{5, 14, 5}

};

int num\_processes = sizeof(processes) / sizeof(processes[0]);

int quantum = 2;

calculate\_waiting\_time\_rr(processes, num\_processes, quantum);

calculate\_turnaround\_time(processes, num\_processes);

float avg\_waiting\_time = calculate\_average\_waiting\_time(processes, num\_processes);

printf("Average waiting time for Round Robin: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output (Round Robin):**

****

**Code (Pre-emptive Priority):**

#include <stdio.h>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int priority;

int waiting\_time;

int turnaround\_time;

int remaining\_time;

};

void calculate\_waiting\_time\_pp(struct Process \*processes, int num\_processes) {

int remaining\_processes = num\_processes;

int current\_time = 0;

while (remaining\_processes > 0) {

int highest\_priority\_process\_index = -1;

int highest\_priority = -1;

for (int i = 0; i < num\_processes; i++) {

if (processes[i].arrival\_time <= current\_time && processes[i].remaining\_time > 0) {

if (processes[i].priority > highest\_priority) {

highest\_priority = processes[i].priority;

highest\_priority\_process\_index = i;

}

}

}

if (highest\_priority\_process\_index == -1) {

current\_time++;

} else {

processes[highest\_priority\_process\_index].remaining\_time--;

if (processes[highest\_priority\_process\_index].remaining\_time == 0) {

remaining\_processes--;

int process\_index = highest\_priority\_process\_index;

processes[process\_index].turnaround\_time = current\_time - processes[process\_index].arrival\_time + 1;

processes[process\_index].waiting\_time = processes[process\_index].turnaround\_time - processes[process\_index].burst\_time;

}

current\_time++;

}

}

}

float calculate\_average\_waiting\_time(struct Process \*processes, int num\_processes) {

int total\_waiting\_time = 0;

for (int i = 0; i < num\_processes; i++) {

total\_waiting\_time += processes[i].waiting\_time;

}

return (float) total\_waiting\_time / num\_processes;

}

int main() {

struct Process processes[] = {

{1, 0, 10, 3},

{2, 5, 4, 1},

{3, 8, 7, 2},

{4, 12, 1, 4},

{5, 14, 5, 5}

};

int num\_processes = sizeof(processes) / sizeof(processes[0]);

for (int i = 0; i < num\_processes; i++) {

processes[i].remaining\_time = processes[i].burst\_time;

}

calculate\_waiting\_time\_pp(processes, num\_processes);

float avg\_waiting\_time = calculate\_average\_waiting\_time(processes, num\_processes);

printf("Average waiting time for Preemptive Priority: %.2f\n", avg\_waiting\_time);

return 0;

}  
**Output (Pre-emptive Priority):**

****

**Experiment 4**

**Aim:** Inter process communication in C

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#define SHM\_SIZE 1024

int main() {

char \*message = "Hello, world!";

int shmid;

key\_t key = 12345;

char \*shm, \*s;

// create shared memory segment

if ((shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666)) < 0) {

perror("shmget");

exit(1);

}

// attach shared memory segment

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1) {

perror("shmat");

exit(1);

}

// copy message to shared memory segment

strcpy(shm, message);

// read from shared memory segment

for (s = shm; \*s != '\0'; s++) {

putchar(\*s);

}

putchar('\n');

// detach shared memory segment

if (shmdt(shm) == -1) {

perror("shmdt");

exit(1);

}

// delete shared memory segment

if (shmctl(shmid, IPC\_RMID, NULL) == -1) {

perror("shmctl");

exit(1);

}

return 0;

}

**Output:**

****

**Experiment 5**

**Aim:** Critical Section problem in C

**Code:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM\_THREADS 2

int shared\_variable = 0;

sem\_t mutex;

void \*thread\_function(void \*thread\_id) {

int i;

int id = \*(int \*) thread\_id;

for (i = 0; i < 100000; i++) {

sem\_wait(&mutex);

shared\_variable++;

sem\_post(&mutex);

}

printf("Thread %d: Shared variable = %d\n", id, shared\_variable);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int thread\_ids[NUM\_THREADS];

int i;

sem\_init(&mutex, 0, 1);

for (i = 0; i < NUM\_THREADS; i++) {

thread\_ids[i] = i;

pthread\_create(&threads[i], NULL, thread\_function, &thread\_ids[i]);

}

for (i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

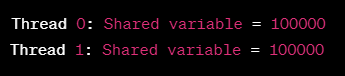
}

sem\_destroy(&mutex);

return 0;

}

**Output:**

****

**Experiment 6**

**Aim:** Producer – Consumer problem using bounded and unbounded buffer in C

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

sem\_t empty;

sem\_t full;

sem\_t mutex;

void \*producer(void \*arg) {

int item;

while (1) {

item = rand() % 10; // Produce an item

sem\_wait(&empty);

sem\_wait(&mutex);

buffer[in] = item;

in = (in + 1) % BUFFER\_SIZE;

printf("Produced item: %d\n", item);

sem\_post(&mutex);

sem\_post(&full);

sleep(rand() % 3); // Sleep for a random amount of time

}

}

void \*consumer(void \*arg) {

int item;

while (1) {

sem\_wait(&full);

sem\_wait(&mutex);

item = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

printf("Consumed item: %d\n", item);

sem\_post(&mutex);

sem\_post(&empty);

sleep(rand() % 3); // Sleep for a random amount of time

}

}

int main() {

pthread\_t prod\_thread, cons\_thread;

// Initialize semaphores

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

sem\_init(&mutex, 0, 1);

// Create threads

pthread\_create(&prod\_thread, NULL, producer, NULL);

pthread\_create(&cons\_thread, NULL, consumer, NULL);

// Wait for threads to finish

pthread\_join(prod\_thread, NULL);

pthread\_join(cons\_thread, NULL);

// Destroy semaphores

sem\_destroy(&empty);

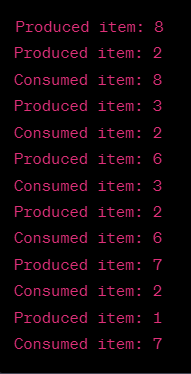
sem\_destroy(&full);

sem\_destroy(&mutex);

return 0;

}

**Output:**

****

**Experiment 7**

**Aim:** Reader Writers problem, Dining Philosophers problem using semaphores in C

**Code (Reader Writers Problem):**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

int readers\_count = 0;

sem\_t rw\_mutex, r\_mutex;

void \*writer(void \*arg) {

while (1) {

sem\_wait(&rw\_mutex);

printf("Writer %d is writing to the shared resource.\n", \*(int \*)arg);

sem\_post(&rw\_mutex);

sleep(rand() % 3);

}

}

void \*reader(void \*arg) {

while (1) {

sem\_wait(&r\_mutex);

readers\_count++;

if (readers\_count == 1) {

sem\_wait(&rw\_mutex);

}

sem\_post(&r\_mutex);

printf("Reader %d is reading the shared resource.\n", \*(int \*)arg);

sem\_wait(&r\_mutex);

readers\_count--;

if (readers\_count == 0) {

sem\_post(&rw\_mutex);

}

sem\_post(&r\_mutex);

sleep(rand() % 3);

}

}

int main() {

pthread\_t writer\_threads[3], reader\_threads[5];

int i, writer\_ids[3], reader\_ids[5];

sem\_init(&rw\_mutex, 0, 1);

sem\_init(&r\_mutex, 0, 1);

for (i = 0; i < 3; i++) {

writer\_ids[i] = i + 1;

pthread\_create(&writer\_threads[i], NULL, writer, (void \*)&writer\_ids[i]);

}

for (i = 0; i < 5; i++) {

reader\_ids[i] = i + 1;

pthread\_create(&reader\_threads[i], NULL, reader, (void \*)&reader\_ids[i]);

}

for (i = 0; i < 3; i++) {

pthread\_join(writer\_threads[i], NULL);

}

for (i = 0; i < 5; i++) {

pthread\_join(reader\_threads[i], NULL);

}

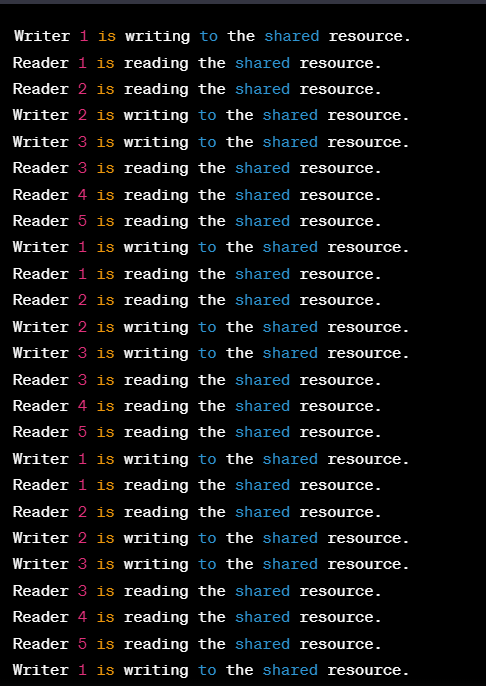
sem\_destroy(&rw\_mutex);

sem\_destroy(&r\_mutex);

return 0;

}

**Output (Readers Writers Problem):**

****

**Code (Dining Philosophers Problem):**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define N 5 // number of philosophers and forks

pthread\_t threads[N];

sem\_t forks[N];

sem\_t mutex;

void \*philosopher(void \*arg) {

int id = \*(int \*)arg;

int left\_fork = id;

int right\_fork = (id + 1) % N;

while (1) {

printf("Philosopher %d is thinking...\n", id);

sem\_wait(&mutex);

sem\_wait(&forks[left\_fork]);

sem\_wait(&forks[right\_fork]);

sem\_post(&mutex);

printf("Philosopher %d is eating using forks %d and %d.\n", id, left\_fork, right\_fork);

sem\_post(&forks[left\_fork]);

sem\_post(&forks[right\_fork]);

printf("Philosopher %d finished eating.\n", id);

sleep(rand() % 3);

}

}

int main() {

int i, philosopher\_ids[N];

sem\_init(&mutex, 0, 1);

for (i = 0; i < N; i++) {

sem\_init(&forks[i], 0, 1);

}

for (i = 0; i < N; i++) {

philosopher\_ids[i] = i;

pthread\_create(&threads[i], NULL, philosopher, (void \*)&philosopher\_ids[i]);

}

for (i = 0; i < N; i++) {

pthread\_join(threads[i], NULL);

}

sem\_destroy(&mutex);

for (i = 0; i < N; i++) {

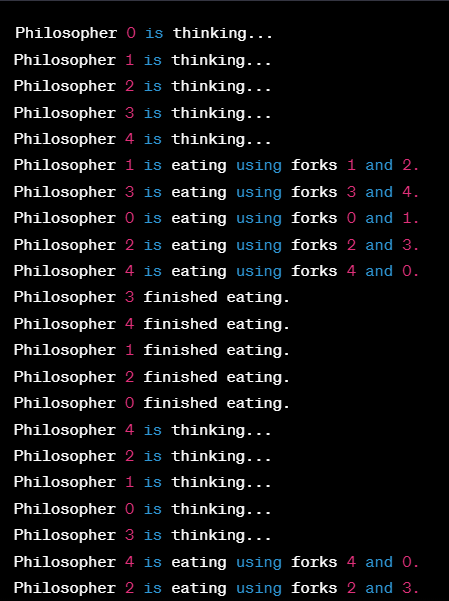
sem\_destroy(&forks[i]);

}

return 0;

}

**Output (Dining Philosophers Problem):**

****

**Experiment 8**

**Aim:** Banker’s algorithm in C

**Code:**

#include <stdio.h>

#include <stdbool.h>

// Define constants

#define NUM\_PROCESSES 5

#define NUM\_RESOURCES 3

// Define available resources

int available[NUM\_RESOURCES] = {10, 5, 7};

// Define maximum demand matrix

int max[NUM\_PROCESSES][NUM\_RESOURCES] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

// Define allocation matrix

int allocation[NUM\_PROCESSES][NUM\_RESOURCES] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

// Define need matrix

int need[NUM\_PROCESSES][NUM\_RESOURCES];

// Define safety algorithm function

bool safety\_algorithm(int work[], bool finish[]) {

int i, j;

// Initialize work and finish arrays

for (i = 0; i < NUM\_RESOURCES; i++) {

work[i] = available[i];

finish[i] = false;

}

// Find an i such that both

// - finish[i] == false

// - need[i] <= work

for (i = 0; i < NUM\_PROCESSES; i++) {

if (!finish[i]) {

bool found = true;

for (j = 0; j < NUM\_RESOURCES; j++) {

if (need[i][j] > work[j]) {

found = false;

break;

}

}

if (found) {

for (j = 0; j < NUM\_RESOURCES; j++) {

work[j] += allocation[i][j];

}

finish[i] = true;

i = -1; // Start again from the beginning

}

}

}

// Return true if all processes are finished

for (i = 0; i < NUM\_PROCESSES; i++) {

if (!finish[i]) {

return false;

}

}

return true;

}

// Define request function

bool request(int pid, int request[]) {

int i, j;

bool finish[NUM\_PROCESSES];

int work[NUM\_RESOURCES];

// Check if request is less than or equal to need

for (i = 0; i < NUM\_RESOURCES; i++) {

if (request[i] > need[pid][i]) {

return false;

}

}

// Check if request is less than or equal to available

for (i = 0; i < NUM\_RESOURCES; i++) {

if (request[i] > available[i]) {

return false;

}

}

// Try to allocate resources and check for safety

for (i = 0; i < NUM\_RESOURCES; i++) {

available[i] -= request[i];

allocation[pid][i] += request[i];

need[pid][i] -= request[i];

}

bool safe = safety\_algorithm(work, finish);

if (!safe) {

// Undo allocation if not safe

for (i = 0; i < NUM\_RESOURCES; i++) {

available[i] += request[i];

allocation[pid][i] -= request[i];

need[pid][i] += request[i];

}

}

return safe;

}

int main() {

int n, m, i, j, k;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resources: ");

scanf("%d", &m);

// Allocate memory for matrices

int \*available = (int \*)malloc(m \* sizeof(int));

int \*\*maximum = (int \*\*)malloc(n \* sizeof(int \*));

int \*\*allocation = (int \*\*)malloc(n \* sizeof(int \*));

int \*\*need = (int \*\*)malloc(n \* sizeof(int \*));

bool \*finished = (bool \*)malloc(n \* sizeof(bool));

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

maximum[i] = (int \*)malloc(m \* sizeof(int));

allocation[i] = (int \*)malloc(m \* sizeof(int));

need[i] = (int \*)malloc(m \* sizeof(int));

}

// Read in available resources

printf("Enter number of available resources:\n");

for (i = 0; i < m; i++) {

scanf("%d", &available[i]);

}

// Read in maximum resource allocation for each process

printf("Enter maximum resource allocation for each process:\n");

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

printf("Process %d:\n", i);

for (j = 0; j < m; j++) {

scanf("%d", &maximum[i][j]);

}

}

// Read in current resource allocation for each process

printf("Enter current resource allocation for each process:\n");

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

printf("Process %d:\n", i);

for (j = 0; j < m; j++) {

scanf("%d", &allocation[i][j]);

need[i][j] = maximum[i][j] - allocation[i][j];

}

finished[i] = false;

}

// Run banker's algorithm

int \*work = (int \*)malloc(m \* sizeof(int));

for (i = 0; i < m; i++) {

work[i] = available[i];

}

bool found;

int count = 0;

int \*safe\_sequence = (int \*)malloc(n \* sizeof(int));

while (count < n) {

found = false;

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

if (!finished[i]) {

for (j = 0; j < m; j++) {

if (need[i][j] > work[j]) {

break;

}

}

if (j == m) {

for (k = 0; k < m; k++) {

work[k] += allocation[i][k];

}

finished[i] = true;

safe\_sequence[count++] = i;

found = true;

}

}

}

if (!found) {

break;

}

}

if (count == n) {

printf("The system is in a safe state.\nSafe sequence:");

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

printf(" %d", safe\_sequence[i]);

}

printf("\n");

} else {

printf("The system is not in a safe state.\n");

}

// Free memory

free(available);

free(work);

free(safe\_sequence);

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

free(maximum[i]);

free(allocation[i]);

free(need[i]);

}

free(maximum);

free(allocation);

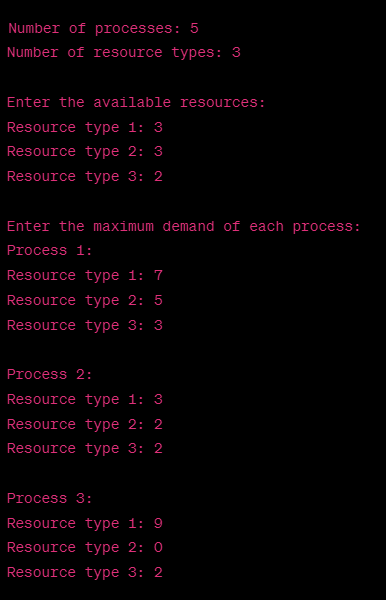
free(need);

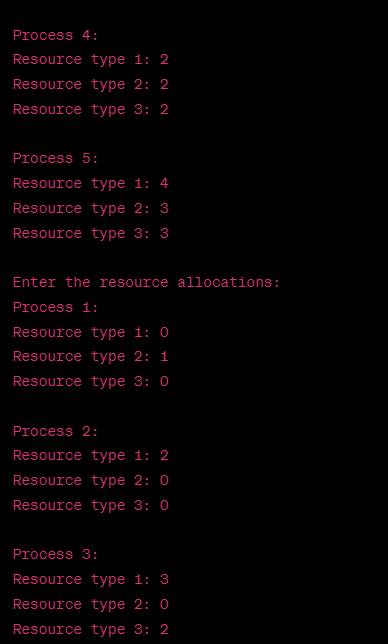
free(finished);

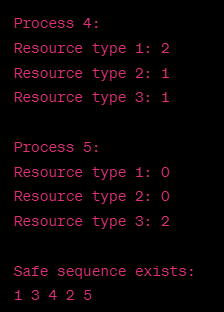
return 0;

}

**Output:**

****

****

****

**Experiment 9**

**Aim:** Page replacement algorithms: LRU, LRU-Approximation, FIFO, Optimal in C

**Code (Least Recently Used Algorithm):**

#include <stdio.h>

#include <stdlib.h>

#define PAGE\_SIZE 100

int main() {

int n, i, j, page\_faults = 0, count = 0, min;

int pages[PAGE\_SIZE], frames[PAGE\_SIZE], counter[PAGE\_SIZE];

printf("Enter number of pages: ");

scanf("%d", &n);

printf("Enter reference string: ");

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

scanf("%d", &pages[i]);

}

printf("Enter number of frames: ");

scanf("%d", &n);

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

frames[i] = -1;

}

for (i = 0; i < PAGE\_SIZE; i++) {

counter[i] = 0;

}

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

int page = pages[i];

int flag = 0;

for (j = 0; j < PAGE\_SIZE; j++) {

if (frames[j] == page) {

counter[j] = ++count;

flag = 1;

break;

}

}

if (flag == 0) {

min = counter[0];

for (j = 1; j < n; j++) {

if (counter[j] < min) {

min = counter[j];

}

}

for (j = 0; j < n; j++) {

if (counter[j] == min) {

frames[j] = page;

counter[j] = ++count;

page\_faults++;

break;

}

}

}

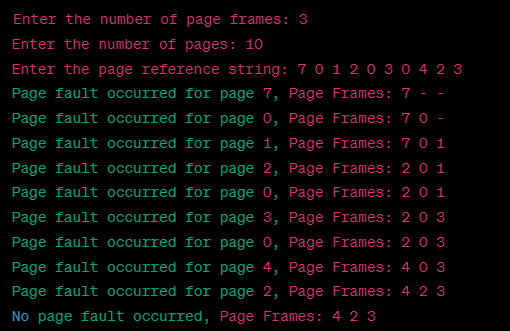
printf("%d\n", frames[i]);}

printf("Total page faults: %d\n", page\_faults);

return 0;

}

**Output (Least Recently Used Algorithm):**

****

**Code (LRU-Approximation Algorithm):**

#include <stdio.h>

#include <stdbool.h>

#define PAGE\_SIZE 100

#define FRAME\_SIZE 10

int main() {

int page[PAGE\_SIZE], frame[FRAME\_SIZE], counter[FRAME\_SIZE];

int page\_faults = 0, index, i, j, k;

for (i = 0; i < FRAME\_SIZE; i++) {

frame[i] = -1;

counter[i] = 0;

}

printf("Enter the reference string:\n");

for (i = 0; i < PAGE\_SIZE; i++) {

scanf("%d", &page[i]);

}

for (i = 0; i < PAGE\_SIZE; i++) {

for (j = 0; j < FRAME\_SIZE; j++) {

if (frame[j] == page[i]) {

counter[j] = counter[j] >> 1 | (1 << 7);

break;

}

}

if (j == FRAME\_SIZE) {

index = 0;

for (k = 1; k < FRAME\_SIZE; k++) {

if (counter[k] < counter[index]) {

index = k;

}

}

frame[index] = page[i];

counter[index] = 1 << 7;

page\_faults++;

}

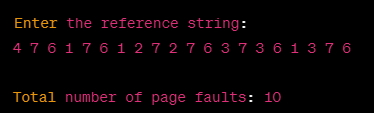
}

printf("\nTotal number of page faults: %d\n", page\_faults);

return 0;

}

**Output (LRU-Approximation Algorithm):**

****

**Code (First in First out):**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 100

int main() {

int n, m, i, j, k, count = 0, fault = 0;

printf("Enter number of pages: ");

scanf("%d", &n);

printf("Enter number of frames: ");

scanf("%d", &m);

int pages[n], frames[m], index[m];

bool hit;

printf("Enter page references:\n");

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

scanf("%d", &pages[i]);

}

for (i = 0; i < m; i++) {

frames[i] = -1;

index[i] = 0;

}

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

hit = false;

for (j = 0; j < m; j++) {

if (frames[j] == pages[i]) {

hit = true;

break;

}

}

if (!hit) {

frames[count] = pages[i];

count = (count + 1) % m;

fault++;

}

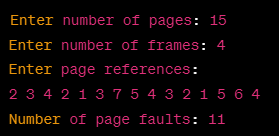
}

printf("Number of page faults: %d\n", fault);

return 0;

}

**Output (First in First out):**

****

**Code (Optimal):**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Optimal Page Replacement Algorithm

void optimal(int pages[], int n, int frames) {

int page\_faults = 0;

int \*frame = (int \*)malloc(frames \* sizeof(int));

bool \*flag = (bool \*)malloc(frames \* sizeof(bool));

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

frame[i] = -1;

flag[i] = false;

}

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

int j;

bool found = false;

for (j = 0; j < frames; j++) {

if (frame[j] == pages[i]) {

found = true;

flag[j] = true;

break;

}

}

if (!found) {

int k;

bool replace = false;

for (j = 0; j < frames; j++) {

if (frame[j] == -1) {

frame[j] = pages[i];

flag[j] = true;

replace = true;

break;

}

}

if (!replace) {

int max\_index = -1, max\_future = -1;

for (j = 0; j < frames; j++) {

if (!flag[j]) {

int furthest = 0;

for (k = i + 1; k < n; k++) {

if (frame[j] == pages[k]) {

furthest = k - i;

break;

}

}

if (furthest > max\_future) {

max\_index = j;

max\_future = furthest;

}

}

}

frame[max\_index] = pages[i];

flag[max\_index] = true;

}

page\_faults++;

}

}

printf("Number of Page Faults: %d\n", page\_faults);

// Free memory

free(frame);

free(flag);

}

// Main function

int main() {

int n, frames, i;

printf("Enter number of pages: ");

scanf("%d", &n);

int \*pages = (int \*)malloc(n \* sizeof(int));

printf("Enter the reference string:\n");

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

scanf("%d", &pages[i]);

}

printf("Enter number of frames: ");

scanf("%d", &frames);

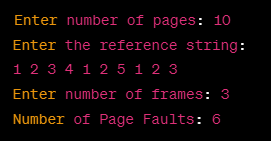
optimal(pages, n, frames);

free(pages);

return 0;

}

**Output (Optimal):**

****

**Experiment 10**

**Aim:** File operation system calls (open, read, close, append etc.) in C

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main()

{

char filename[50], content[1000];

int choice, len;

printf("Enter file name: ");

scanf("%s", filename);

printf("Choose an option:\n");

printf("1. Write to file\n");

printf("2. Read from file\n");

printf("3. Append to file\n");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter content to write to file:\n");

scanf("%s", content);

FILE \*fp1;

fp1 = fopen(filename, "w");

if (fp1 == NULL) {

printf("Error opening file.\n");

exit(1);

}

fprintf(fp1, "%s", content);

printf("Successfully wrote to file.\n");

fclose(fp1);

break;

case 2:

FILE \*fp2;

char ch;

fp2 = fopen(filename, "r");

if (fp2 == NULL) {

printf("Error opening file.\n");

exit(1);

}

printf("File contents:\n");

while ((ch = fgetc(fp2)) != EOF) {

printf("%c", ch);

}

fclose(fp2);

break;

case 3:

printf("Enter content to append to file:\n");

scanf("%s", content);

FILE \*fp3;

fp3 = fopen(filename, "a");

if (fp3 == NULL) {

printf("Error opening file.\n");

exit(1);

}

fprintf(fp3, "%s", content);

printf("Successfully appended to file.\n");

fclose(fp3);

break;

default:

printf("Invalid choice.\n");

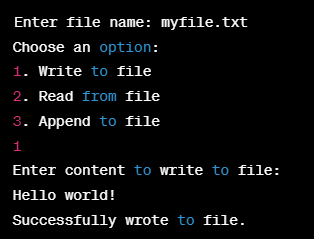
break;

}

return 0;

}

**Output:**

****

**Experiment 11**

**Aim:** Disk scheduling algorithms: FCFS, SSTF, SCAN, CSCAN, LOOK, CLOOK in C

**Code (FCFS):**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int n, i, start, head\_movement = 0;

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

scanf("%d", &n);

int request[n];

printf("Enter the requests: ");

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

scanf("%d", &request[i]);

}

printf("Enter the starting position of the disk head: ");

scanf("%d", &start);

// First come first serve algorithm

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

head\_movement += abs(request[i] - start);

start = request[i];

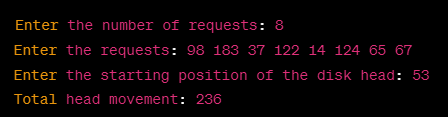
}

printf("Total head movement: %d\n", head\_movement);

return 0;

}

**Output (FCFS):**

****

**Code (SSTF):**

#include<stdio.h>

#include<stdlib.h>

int main() {

int queue[20], q\_size, head, i, j, seek = 0, diff;

float avg;

printf("Enter the size of queue: ");

scanf("%d", &q\_size);

printf("Enter queue elements: ");

for (i = 0; i < q\_size; i++) {

scanf("%d", &queue[i]);

}

printf("Enter initial head position: ");

scanf("%d", &head);

int temp, res[q\_size + 1];

for (i = 0; i <= q\_size; i++) {

res[i] = 0;

}

for (i = 0; i < q\_size; i++) {

int min = 9999, min\_index = -1;

for (j = 0; j < q\_size; j++) {

if (!res[j]) {

diff = abs(head - queue[j]);

if (diff < min) {

min = diff;

min\_index = j;

}

}

}

res[min\_index] = 1;

seek += min;

head = queue[min\_index];

}

printf("Total seek time: %d\n", seek);

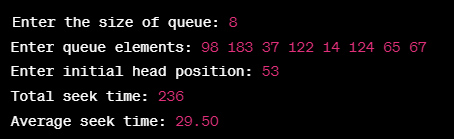
avg = (float)seek / q\_size;

printf("Average seek time: %.2f\n", avg);

return 0;

}

**Output (SSTF):**

****

**Code (SCAN):**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

void SCAN(int arr[], int head, int size, int disk\_size)

{

int i, j, seek\_time = 0, current\_head, direction;

int \*visited = (int\*)malloc(size \* sizeof(int));

int \*requests = (int\*)malloc(size \* sizeof(int));

for(i = 0; i < size; i++) {

requests[i] = arr[i];

visited[i] = 0;

}

current\_head = head;

printf("Starting from %d ", current\_head);

printf("SCAN path: ");

// determine initial direction

if(requests[0] > current\_head) {

direction = 1; // towards right

} else {

direction = -1; // towards left

}

// SCAN algorithm

for(i = 0; i < size; i++) {

for(j = 0; j < size - i - 1; j++) {

if(requests[j] > requests[j + 1]) {

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

for(i = 0; i < size; i++) {

if(requests[i] == current\_head) {

visited[i] = 1;

printf("%d ", requests[i]);

break;

}

}

// scan towards right direction

for(i = i + 1; i < size; i++) {

if(visited[i] == 0) {

seek\_time += abs(current\_head - requests[i]);

current\_head = requests[i];

visited[i] = 1;

printf("%d ", requests[i]);

}

}

// scan towards left direction

if(direction == -1) {

for(i = 0; i < size; i++) {

if(visited[i] == 0) {

seek\_time += abs(current\_head - requests[i]);

current\_head = requests[i];

visited[i] = 1;

printf("%d ", requests[i]);

}

}

}

printf("\nTotal seek time: %d\n", seek\_time);

}

int main()

{

int size, disk\_size, head;

printf("Enter size of request queue: ");

scanf("%d", &size);

int \*arr = (int\*)malloc(size \* sizeof(int));

printf("Enter request queue:\n");

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

scanf("%d", &arr[i]);

}

printf("Enter initial head position: ");

scanf("%d", &head);

printf("Enter size of disk: ");

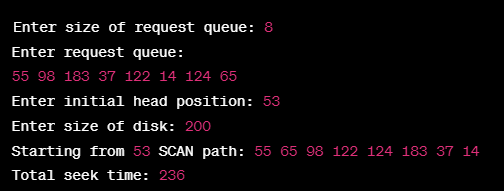
scanf("%d", &disk\_size);

SCAN(arr, head, size, disk\_size);

return 0;

}

**Output (SCAN):**

****

**Code (CSCAN):**

#include<stdio.h>

#include<stdlib.h>

void cscan(int queue[], int n, int head, int maxCylinders) {

int i, seek\_time = 0, distance = 0, current = head;

int visited[n], order[n], index = 0;

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

visited[i] = 0;

}

// sort the queue

for (i = 0; i < n - 1; i++) {

int j, min = i;

for (j = i + 1; j < n; j++) {

if (queue[j] < queue[min]) {

min = j;

}

}

int temp = queue[min];

queue[min] = queue[i];

queue[i] = temp;

}

// move towards the right end

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

if (queue[i] >= current) {

distance = abs(queue[i] - current);

seek\_time += distance;

current = queue[i];

visited[i] = 1;

order[index++] = queue[i];

}

}

// move towards the left end

seek\_time += abs(maxCylinders - current);

current = 0;

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

if (!visited[i]) {

distance = abs(queue[i] - current);

seek\_time += distance;

current = queue[i];

order[index++] = queue[i];

}

}

printf("CSCAN:\n");

printf("Total Seek Time: %d\n", seek\_time);

printf("Order of Access: ");

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

printf("%d ", order[i]);

}

printf("\n");

}

int main() {

int n, head, maxCylinders, i;

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

scanf("%d", &n);

int queue[n];

printf("Enter disk requests: ");

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

scanf("%d", &queue[i]);

}

printf("Enter current head position: ");

scanf("%d", &head);

printf("Enter maximum number of cylinders: ");

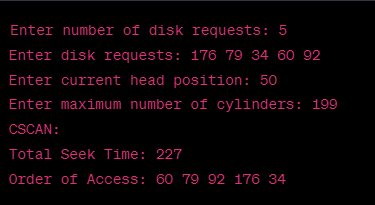
scanf("%d", &maxCylinders);

cscan(queue, n, head, maxCylinders);

return 0;

}

**Output (CSCAN):**

****

**Code (LOOK):**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

int main()

{

int n, head, total\_seek\_time = 0, current\_pos, i, j;

bool direction;

printf("Enter number of requests: ");

scanf("%d", &n);

int requests[n];

bool visited[n];

printf("Enter requests:\n");

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

scanf("%d", &requests[i]);

visited[i] = false;

}

printf("Enter initial head position: ");

scanf("%d", &head);

printf("Enter direction (0 for left, 1 for right): ");

scanf("%d", &direction);

current\_pos = head;

if (direction) {

// sort requests in ascending order

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;

}

}

}

// service requests to the right of current position

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

if (!visited[i] && requests[i] >= current\_pos) {

total\_seek\_time += abs(requests[i] - current\_pos);

current\_pos = requests[i];

visited[i] = true;

}

}

// service remaining requests to the left

for (i = n-1; i >= 0; i--) {

if (!visited[i]) {

total\_seek\_time += abs(requests[i] - current\_pos);

current\_pos = requests[i];

}

}

} else {

// sort requests in descending order

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;

}

}

}

// service requests to the left of current position

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

if (!visited[i] && requests[i] <= current\_pos) {

total\_seek\_time += abs(requests[i] - current\_pos);

current\_pos = requests[i];

visited[i] = true;

}

}

// service remaining requests to the right

for (i = n-1; i >= 0; i--) {

if (!visited[i]) {

total\_seek\_time += abs(requests[i] - current\_pos);

current\_pos = requests[i];

}

}

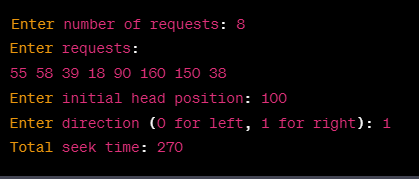
}

printf("Total seek time: %d\n", total\_seek\_time);

return 0;

}

**Output (LOOK):**

****

**Code (CLOOK):**

#include <stdio.h>

#include <stdlib.h>

void clook(int arr[], int head, int n)

{

int i, j, total = 0;

int range = abs(head - arr[0]);

int left = arr[0];

int right = arr[n - 1];

int visited[n];

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

visited[i] = 0;

}

printf("C-LOOK Disk Scheduling Algorithm\n");

printf("Initial Head Position: %d\n", head);

printf("Seek Sequence is: %d ", head);

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

if (arr[i] < head) {

visited[i] = 1;

total += abs(head - arr[i]);

head = arr[i];

printf("%d ", head);

}

}

printf("%d ", left);

total += abs(head - left);

head = left;

for (i = n - 1; i >= 0; i--) {

if (!visited[i]) {

total += abs(head - arr[i]);

head = arr[i];

printf("%d ", head);

}

}

printf("\nTotal Seek Time: %d\n", total);

}

int main()

{

int n, i, head;

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

scanf("%d", &n);

int arr[n];

printf("Enter the requests sequence: ");

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

scanf("%d", &arr[i]);

}

printf("Enter the initial head position: ");

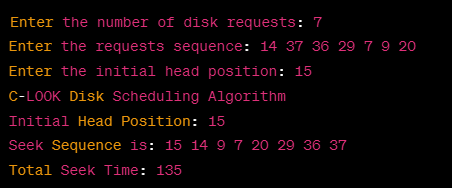
scanf("%d", &head);

clook(arr, head, n);

return 0;

}

**Output (CLOOK):**

****