1.

int main() {

// Create a new process

if (fork() == 0) {

// Child process

printf("Child Process - PID: %d, PPID: %d\n", getpid(), getppid());

} else {

// Parent process

printf("Parent Process - PID: %d, PPID: %d\n", getpid(), getppid());

}

return 0;

}

2.

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

int src, dest;

char buffer[1024];

ssize\_t bytes;

// Open the source file

src = open("source.txt", O\_RDONLY);

// Open the destination file

dest = open("destination.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

// Copy the content

while ((bytes = read(src, buffer, sizeof(buffer))) > 0) {

write(dest, buffer, bytes);

}

// Close the files

close(src);

close(dest);

return 0;

}

3.

#include <stdio.h>

int main() {

int n, i;

int burst\_time[10], waiting\_time[10], turnaround\_time[10];

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

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

scanf("%d", &n);

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

printf("Enter burst time for Process %d: ", i + 1);

scanf("%d", &burst\_time[i]);

}

waiting\_time[0] = 0; // First process waiting time is 0

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

waiting\_time[i] = waiting\_time[i - 1] + burst\_time[i - 1];

}

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

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

}

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\n", i + 1, burst\_time[i], waiting\_time[i], turnaround\_time[i]);

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

total\_turnaround\_time += turnaround\_time[i];

}

printf("\nAverage Waiting Time = %.2f\n", (float)total\_waiting\_time / n);

printf("Average Turnaround Time = %.2f\n", (float)total\_turnaround\_time / n);

return 0;

}

4.

#include <stdio.h>

void sort\_by\_burst\_time(int n, int burst\_time[], int p[]) {

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

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

if (burst\_time[i] > burst\_time[j]) {

// Swap burst time

int temp = burst\_time[i];

burst\_time[i] = burst\_time[j];

burst\_time[j] = temp;

// Swap process number

temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}

int main() {

int n;

int burst\_time[10], p[10], waiting\_time[10], turnaround\_time[10];

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

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

scanf("%d", &n);

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

printf("Enter burst time for Process %d: ", i + 1);

scanf("%d", &burst\_time[i]);

p[i] = i + 1; // Process number

}

// Sort processes by burst time

sort\_by\_burst\_time(n, burst\_time, p);

waiting\_time[0] = 0; // Waiting time for first process is 0

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

waiting\_time[i] = waiting\_time[i - 1] + burst\_time[i - 1];

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

}

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

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

total\_turnaround\_time += turnaround\_time[i];

}

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\n", p[i], burst\_time[i], waiting\_time[i], turnaround\_time[i]);

}

printf("\nAverage Waiting Time = %.2f\n", (float)total\_waiting\_time / n);

printf("Average Turnaround Time = %.2f\n", (float)total\_turnaround\_time / n);

return 0;

}

5.

#include <stdio.h>

struct Process {

int id;

int burst\_time;

int priority;

int waiting\_time;

int turnaround\_time;

};

void sort\_by\_priority(struct Process processes[], int n) {

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

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

if (processes[i].priority > processes[j].priority) {

struct Process temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

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

processes[i].id = i + 1;

printf("Enter burst time and priority for Process %d: ", i + 1);

scanf("%d %d", &processes[i].burst\_time, &processes[i].priority);

}

// Sort processes by priority

sort\_by\_priority(processes, n);

// Calculate waiting and turnaround times

processes[0].waiting\_time = 0; // First process waiting time is 0

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

processes[i].waiting\_time = processes[i - 1].waiting\_time + processes[i - 1].burst\_time;

}

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

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

}

// Print results

printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst\_time, processes[i].priority, processes[i].waiting\_time, processes[i].turnaround\_time);

}

return 0;

}

6. #include <stdio.h>

struct Process {

int id, burst\_time, remaining\_time, priority;

};

int main() {

int n, completed = 0, time = 0, highest\_priority;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process p[n];

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

p[i].id = i + 1;

printf("Enter burst time and priority for P%d: ", i + 1);

scanf("%d %d", &p[i].burst\_time, &p[i].priority);

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

}

while (completed != n) {

highest\_priority = -1;

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

if (p[i].remaining\_time > 0 && (highest\_priority == -1 || p[i].priority < p[highest\_priority].priority))

highest\_priority = i;

}

p[highest\_priority].remaining\_time--;

if (p[highest\_priority].remaining\_time == 0) completed++;

time++;

}

printf("All processes completed in %d units of time.\n", time);

return 0;

}

7. #include <stdio.h>

struct Process {

int id, burst\_time, waiting\_time, turnaround\_time;

};

void sort\_by\_burst\_time(struct Process p[], int n) {

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

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

if (p[i].burst\_time > p[j].burst\_time) {

struct Process temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}

int main() {

int n;

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

scanf("%d", &n);

struct Process p[n];

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

p[i].id = i + 1;

printf("Enter burst time for Process %d: ", i + 1);

scanf("%d", &p[i].burst\_time);

}

sort\_by\_burst\_time(p, n);

p[0].waiting\_time = 0;

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

p[i].waiting\_time = p[i - 1].waiting\_time + p[i - 1].burst\_time;

}

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

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

}

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst\_time, p[i].waiting\_time, p[i].turnaround\_time);

}

return 0;

}

8. #include <stdio.h>

struct Process {

int id, burst\_time, remaining\_time, waiting\_time, turnaround\_time;

};

int main() {

int n, time\_quantum, time = 0;

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

scanf("%d", &n);

struct Process p[n];

printf("Enter the time quantum: ");

scanf("%d", &time\_quantum);

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

p[i].id = i + 1;

printf("Enter burst time for Process %d: ", i + 1);

scanf("%d", &p[i].burst\_time);

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

p[i].waiting\_time = 0;

}

while (1) {

int done = 1;

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

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

done = 0;

if (p[i].remaining\_time > time\_quantum) {

time += time\_quantum;

p[i].remaining\_time -= time\_quantum;

} else {

time += p[i].remaining\_time;

p[i].waiting\_time = time - p[i].burst\_time;

p[i].remaining\_time = 0;

}

}

}

if (done)

break;

}

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

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

}

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst\_time, p[i].waiting\_time, p[i].turnaround\_time);

}

return 0;

}

9.

writer

#include <stdio.h>

#include <stdlib.h>

#include <sys/shm.h>

#include <sys/ipc.h>

#include <string.h>

int main() {

key\_t key = ftok("shmfile", 65);

int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);

char \*str = (char\*) shmat(shmid, (void\*)0, 0);

strcpy(str, "Hello from writer process!");

printf("Data written in memory: %s\n", str);

shmdt(str);

return 0;

}

Reader

#include <stdio.h>

#include <stdlib.h>

#include <sys/shm.h>

#include <sys/ipc.h>

int main() {

key\_t key = ftok("shmfile", 65);

int shmid = shmget(key, 1024, 0666);

char \*str = (char\*) shmat(shmid, (void\*)0, 0);

printf("Data read from memory: %s\n", str);

shmdt(str);

shmctl(shmid, IPC\_RMID, NULL);

return 0;

}

10.

Sender

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct message\_buffer {

long message\_type;

char message\_text[100];

};

int main() {

key\_t key;

int msgid;

struct message\_buffer message;

key = ftok("progfile", 65);

msgid = msgget(key, 0666 | IPC\_CREAT);

message.message\_type = 1;

printf("Write Data: ");

fgets(message.message\_text, sizeof(message.message\_text), stdin);

msgsnd(msgid, &message, sizeof(message), 0);

printf("Data sent: %s\n", message.message\_text);

return 0;

}

Receiver

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct message\_buffer {

long message\_type;

char message\_text[100];

};

int main() {

key\_t key;

int msgid;

struct message\_buffer message;

key = ftok("progfile", 65);

msgid = msgget(key, 0666 | IPC\_CREAT);

msgrcv(msgid, &message, sizeof(message), 1, 0);

printf("Data received: %s\n", message.message\_text);

msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

11.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \*print\_message(void \*arg) {

char \*message = (char \*)arg;

printf("%s\n", message);

return NULL;

}

int main() {

pthread\_t thread1, thread2;

const char \*message1 = "Thread 1: Hello, world!";

const char \*message2 = "Thread 2: Multithreading in C!";

// Create two threads

pthread\_create(&thread1, NULL, print\_message, (void \*)message1);

pthread\_create(&thread2, NULL, print\_message, (void \*)message2);

// Wait for threads to finish

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

return 0;

}

12.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define N 5 // Number of philosophers

sem\_t chopsticks[N]; // Semaphores representing chopsticks

void \*philosopher(void \*num) {

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

while (1) {

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

sleep(rand() % 3 + 1); // Random thinking time

// Pick up chopsticks

sem\_wait(&chopsticks[id]);

sem\_wait(&chopsticks[(id + 1) % N]);

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

sleep(rand() % 3 + 1); // Random eating time

// Put down chopsticks

sem\_post(&chopsticks[id]);

sem\_post(&chopsticks[(id + 1) % N]);

}

return NULL;

}

int main() {

pthread\_t philosophers[N];

int ids[N];

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

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

ids[i] = i;

}

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

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

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

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

}

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

sem\_destroy(&chopsticks[i]);

}

return 0;

}

13.

#include <stdio.h>

#include <stdbool.h>

#define MAX 10

void firstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

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

if (blockSize[j] >= processSize[i]) {

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

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

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1)

printf("%d\n", allocation[i] + 1);

else

printf("Not Allocated\n");

}

}

void bestFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

int bestIdx = -1;

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

if (blockSize[j] >= processSize[i]) {

if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {

bestIdx = j;

}

}

}

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

printf("Best Fit Allocation:\n");

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1)

printf("%d\n", allocation[i] + 1);

else

printf("Not Allocated\n");

}

}

void worstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

int worstIdx = -1;

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

if (blockSize[j] >= processSize[i]) {

if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {

worstIdx = j;

}

}

}

if (worstIdx != -1) {

allocation[i] = worstIdx;

blockSize[worstIdx] -= processSize[i];

}

}

printf("Worst Fit Allocation:\n");

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1)

printf("%d\n", allocation[i] + 1);

else

printf("Not Allocated\n");

}

}

int main() {

int blockSize[MAX] = {100, 500, 200, 300, 600};

int processSize[MAX] = {212, 417, 112, 426};

int m = 5, n = 4;

firstFit(blockSize, m, processSize, n);

bestFit(blockSize, m, processSize, n);

worstFit(blockSize, m, processSize, n);

return 0;

}

14.

#include <stdio.h>

#include <string.h>

#define MAX\_FILES 100

struct Directory {

char name[30];

char files[MAX\_FILES][30];

int file\_count;

};

void add\_file(struct Directory \*dir, char \*filename) {

if (dir->file\_count < MAX\_FILES) {

strcpy(dir->files[dir->file\_count], filename);

dir->file\_count++;

printf("File '%s' added successfully.\n", filename);

} else {

printf("Directory is full. Cannot add more files.\n");

}

}

void search\_file(struct Directory \*dir, char \*filename) {

for (int i = 0; i < dir->file\_count; i++) {

if (strcmp(dir->files[i], filename) == 0) {

printf("File '%s' found in the directory.\n", filename);

return;

}

}

printf("File '%s' not found in the directory.\n", filename);

}

void list\_files(struct Directory \*dir) {

if (dir->file\_count == 0) {

printf("Directory is empty.\n");

return;

}

printf("Files in directory '%s':\n", dir->name);

for (int i = 0; i < dir->file\_count; i++) {

printf("%s\n", dir->files[i]);

}

}

int main() {

struct Directory dir;

strcpy(dir.name, "SingleLevelDirectory");

dir.file\_count = 0;

int choice;

char filename[30];

while (1) {

printf("\n1. Add File\n2. Search File\n3. List Files\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the file name to add: ");

scanf("%s", filename);

add\_file(&dir, filename);

break;

case 2:

printf("Enter the file name to search: ");

scanf("%s", filename);

search\_file(&dir, filename);

break;

case 3:

list\_files(&dir);

break;

case 4:

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

}

15.

#include <stdio.h>

#include <string.h>

#define MAX\_USERS 10

#define MAX\_FILES 10

struct UserDirectory {

char name[30];

char files[MAX\_FILES][30];

int file\_count;

};

struct Directory {

struct UserDirectory users[MAX\_USERS];

int user\_count;

};

void add\_user(struct Directory \*dir, char \*username) {

if (dir->user\_count < MAX\_USERS) {

strcpy(dir->users[dir->user\_count].name, username);

dir->users[dir->user\_count].file\_count = 0;

dir->user\_count++;

printf("User '%s' added successfully.\n", username);

} else {

printf("User directory is full. Cannot add more users.\n");

}

}

void add\_file(struct Directory \*dir, char \*username, char \*filename) {

for (int i = 0; i < dir->user\_count; i++) {

if (strcmp(dir->users[i].name, username) == 0) {

if (dir->users[i].file\_count < MAX\_FILES) {

strcpy(dir->users[i].files[dir->users[i].file\_count], filename);

dir->users[i].file\_count++;

printf("File '%s' added to user '%s' successfully.\n", filename, username);

} else {

printf("User '%s' directory is full. Cannot add more files.\n", username);

}

return;

}

}

printf("User '%s' not found.\n", username);

}

void list\_files(struct Directory \*dir, char \*username) {

for (int i = 0; i < dir->user\_count; i++) {

if (strcmp(dir->users[i].name, username) == 0) {

if (dir->users[i].file\_count == 0) {

printf("User '%s' directory is empty.\n", username);

return;

}

printf("Files in user '%s' directory:\n", username);

for (int j = 0; j < dir->users[i].file\_count; j++) {

printf("%s\n", dir->users[i].files[j]);

}

return;

}

}

printf("User '%s' not found.\n", username);

}

int main() {

struct Directory dir;

dir.user\_count = 0;

int choice;

char username[30];

char filename[30];

while (1) {

printf("\n1. Add User\n2. Add File\n3. List Files\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter username to add: ");

scanf("%s", username);

add\_user(&dir, username);

break;

case 2:

printf("Enter username: ");

scanf("%s", username);

printf("Enter filename to add: ");

scanf("%s", filename);

add\_file(&dir, username, filename);

break;

case 3:

printf("Enter username to list files: ");

scanf("%s", username);

list\_files(&dir, username);

break;

case 4:

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

}

16.

#include <stdio.h>

#include <stdlib.h>

struct Employee {

int id;

char name[30];

float salary;

};

void add\_employee(FILE \*file) {

struct Employee emp;

printf("Enter Employee ID: ");

scanf("%d", &emp.id);

printf("Enter Employee Name: ");

scanf("%s", emp.name);

printf("Enter Employee Salary: ");

scanf("%f", &emp.salary);

fseek(file, 0, SEEK\_END);

fwrite(&emp, sizeof(struct Employee), 1, file);

printf("Employee added successfully.\n");

}

void search\_employee(FILE \*file, int id) {

struct Employee emp;

rewind(file);

while (fread(&emp, sizeof(struct Employee), 1, file)) {

if (emp.id == id) {

printf("Employee Found:\nID: %d\nName: %s\nSalary: %.2f\n", emp.id, emp.name, emp.salary);

return;

}

}

printf("Employee with ID %d not found.\n", id);

}

void display\_employees(FILE \*file) {

struct Employee emp;

rewind(file);

printf("Employee Details:\n");

while (fread(&emp, sizeof(struct Employee), 1, file)) {

printf("ID: %d, Name: %s, Salary: %.2f\n", emp.id, emp.name, emp.salary);

}

}

int main() {

FILE \*file = fopen("employees.dat", "rb+");

if (file == NULL) {

file = fopen("employees.dat", "wb+");

if (file == NULL) {

perror("Error opening file");

return 1;

}

}

int choice, id;

while (1) {

printf("\n1. Add Employee\n2. Search Employee\n3. Display All Employees\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

add\_employee(file);

break;

case 2:

printf("Enter Employee ID to search: ");

scanf("%d", &id);

search\_employee(file, id);

break;

case 3:

display\_employees(file);

break;

case 4:

fclose(file);

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

}

17.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

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

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int maximum[MAX\_PROCESSES][MAX\_RESOURCES] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int available[MAX\_RESOURCES] = {3, 3, 2};

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

void calculate\_need() {

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

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

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

}

}

}

bool is\_safe() {

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES] = {false};

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

work[i] = available[i];

}

while (true) {

bool found = false;

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

if (!finish[i]) {

bool can\_allocate = true;

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

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

can\_allocate = false;

break;

}

}

if (can\_allocate) {

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

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

}

finish[i] = true;

found = true;

}

}

}

if (!found) {

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

if (!finish[i]) {

return false;

}

}

return true;

}

}

}

int main() {

calculate\_need();

if (is\_safe()) {

printf("The system is in a safe state.\n");

} else {

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

}

return 0;

}

18.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty, full, mutex;

void \*producer(void \*arg) {

int item;

while (1) {

item = rand() % 100; // Produce a random item

sem\_wait(&empty);

sem\_wait(&mutex);

// Add item to buffer

buffer[in] = item;

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

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

sem\_post(&mutex);

sem\_post(&full);

sleep(1);

}

}

void \*consumer(void \*arg) {

int item;

while (1) {

sem\_wait(&full);

sem\_wait(&mutex);

// Remove item from buffer

item = buffer[out];

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

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

sem\_post(&mutex);

sem\_post(&empty);

sleep(1);

}

}

int main() {

pthread\_t prod, cons;

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

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

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

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);

sem\_destroy(&mutex);

return 0;

}

19.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

pthread\_mutex\_t lock;

int counter = 0;

void \*increment\_counter(void \*arg) {

for (int i = 0; i < 1000000; i++) {

pthread\_mutex\_lock(&lock);

counter++;

pthread\_mutex\_unlock(&lock);

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

pthread\_mutex\_init(&lock, NULL);

pthread\_create(&thread1, NULL, increment\_counter, NULL);

pthread\_create(&thread2, NULL, increment\_counter, NULL);

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

printf("Final counter value: %d\n", counter);

pthread\_mutex\_destroy(&lock);

return 0;

}

20.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t mutex, writeblock;

int data = 0, read\_count = 0;

void \*reader(void \*arg) {

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

sem\_wait(&mutex);

read\_count++;

if (read\_count == 1) {

sem\_wait(&writeblock); // First reader locks the writer

}

sem\_post(&mutex);

printf("Reader %d: read data = %d\n", id, data);

sem\_wait(&mutex);

read\_count--;

if (read\_count == 0) {

sem\_post(&writeblock); // Last reader unlocks the writer

}

sem\_post(&mutex);

return NULL;

}

void \*writer(void \*arg) {

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

sem\_wait(&writeblock);

data++;

printf("Writer %d: wrote data = %d\n", id, data);

sem\_post(&writeblock);

return NULL;

}

int main() {

pthread\_t r1, r2, w1, w2;

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

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

int reader\_id1 = 1, reader\_id2 = 2, writer\_id1 = 1, writer\_id2 = 2;

pthread\_create(&r1, NULL, reader, &reader\_id1);

pthread\_create(&r2, NULL, reader, &reader\_id2);

pthread\_create(&w1, NULL, writer, &writer\_id1);

pthread\_create(&w2, NULL, writer, &writer\_id2);

pthread\_join(r1, NULL);

pthread\_join(r2, NULL);

pthread\_join(w1, NULL);

pthread\_join(w2, NULL);

sem\_destroy(&mutex);

sem\_destroy(&writeblock);

return 0;

}

21.

#include <stdio.h>

#define MAX 10

void worstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

int worstIdx = -1;

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

if (blockSize[j] >= processSize[i]) {

if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {

worstIdx = j;

}

}

}

if (worstIdx != -1) {

allocation[i] = worstIdx;

blockSize[worstIdx] -= processSize[i];

}

}

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1)

printf("%d\n", allocation[i] + 1);

else

printf("Not Allocated\n");

}

}

int main() {

int blockSize[MAX] = {100, 500, 200, 300, 600};

int processSize[MAX] = {212, 417, 112, 426};

int m = 5, n = 4;

worstFit(blockSize, m, processSize, n);

return 0;

}

22.

#include <stdio.h>

#define MAX 10

void bestFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

int bestIdx = -1;

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

if (blockSize[j] >= processSize[i]) {

if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {

bestIdx = j;

}

}

}

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1)

printf("%d\n", allocation[i] + 1);

else

printf("Not Allocated\n");

}

}

int main() {

int blockSize[MAX] = {100, 500, 200, 300, 600};

int processSize[MAX] = {212, 417, 112, 426};

int m = 5, n = 4;

bestFit(blockSize, m, processSize, n);

return 0;

}

23.

#include <stdio.h>

#define MAX 10

void firstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[n];

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

allocation[i] = -1;

}

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

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

if (blockSize[j] >= processSize[i]) {

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

printf("Process No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1)

printf("%d\n", allocation[i] + 1);

else

printf("Not Allocated\n");

}

}

int main() {

int blockSize[MAX] = {100, 500, 200, 300, 600};

int processSize[MAX] = {212, 417, 112, 426};

int m = 5, n = 4;

firstFit(blockSize, m, processSize, n);

return 0;

}

24.

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

int main() {

int file;

char buffer[100];

ssize\_t bytesRead;

// Create and open a file for writing

file = open("example.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR);

if (file < 0) {

perror("Failed to open file");

return 1;

}

// Write to the file

const char \*text = "Hello, this is a sample text for file management using UNIX system calls.\n";

if (write(file, text, strlen(text)) < 0) {

perror("Failed to write to file");

close(file);

return 1;

}

// Close the file

if (close(file) < 0) {

perror("Failed to close file");

return 1;

}

// Open the file for reading

file = open("example.txt", O\_RDONLY);

if (file < 0) {

perror("Failed to open file for reading");

return 1;

}

// Read from the file

bytesRead = read(file, buffer, sizeof(buffer) - 1);

if (bytesRead < 0) {

perror("Failed to read from file");

close(file);

return 1;

}

buffer[bytesRead] = '\0';

// Print the content read from the file

printf("Content read from file:\n%s", buffer);

// Close the file

if (close(file) < 0) {

perror("Failed to close file");

return 1;

}

return 0;

}

25.

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/stat.h>

#include <dirent.h>

void use\_fcntl(int fd) {

int flags = fcntl(fd, F\_GETFL);

if (flags < 0) {

perror("fcntl F\_GETFL");

return;

}

printf("File status flags: %d\n", flags);

}

void use\_lseek(int fd) {

off\_t offset = lseek(fd, 0, SEEK\_END);

if (offset == (off\_t) -1) {

perror("lseek");

return;

}

printf("File size: %lld bytes\n", (long long) offset);

}

void use\_stat(const char \*filepath) {

struct stat file\_stat;

if (stat(filepath, &file\_stat) < 0) {

perror("stat");

return;

}

printf("File size: %lld bytes\n", (long long) file\_stat.st\_size);

printf("Permissions: %o\n", file\_stat.st\_mode & 0777);

printf("Last modified time: %lld\n", (long long) file\_stat.st\_mtime);

}

void use\_opendir\_readdir(const char \*dirpath) {

DIR \*dir = opendir(dirpath);

if (dir == NULL) {

perror("opendir");

return;

}

struct dirent \*entry;

while ((entry = readdir(dir)) != NULL) {

printf("Name: %s\n", entry->d\_name);

}

closedir(dir);

}

int main() {

const char \*filepath = "example.txt";

const char \*dirpath = ".";

// Open file

int fd = open(filepath, O\_RDONLY | O\_CREAT, S\_IRUSR | S\_IWUSR);

if (fd < 0) {

perror("open");

return 1;

}

use\_fcntl(fd);

use\_lseek(fd);

use\_stat(filepath);

use\_opendir\_readdir(dirpath);

close(fd);

return 0;

}

26.

#include <stdio.h>

#include <stdlib.h>

void create\_file(const char \*filename) {

FILE \*file = fopen(filename, "w");

if (file == NULL) {

perror("Error creating file");

return;

}

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

fclose(file);

}

void write\_to\_file(const char \*filename, const char \*content) {

FILE \*file = fopen(filename, "a");

if (file == NULL) {

perror("Error opening file for writing");

return;

}

fprintf(file, "%s\n", content);

printf("Content written to file '%s' successfully.\n", filename);

fclose(file);

}

void read\_from\_file(const char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file for reading");

return;

}

char buffer[256];

printf("Content of file '%s':\n", filename);

while (fgets(buffer, sizeof(buffer), file) != NULL) {

printf("%s", buffer);

}

fclose(file);

}

void delete\_file(const char \*filename) {

if (remove(filename) == 0) {

printf("File '%s' deleted successfully.\n", filename);

} else {

perror("Error deleting file");

}

}

int main() {

const char \*filename = "example.txt";

const char \*content = "Hello, this is a sample content for file management operations.";

create\_file(filename);

write\_to\_file(filename, content);

read\_from\_file(filename);

delete\_file(filename);

return 0;

}

27.

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

void list\_directory(const char \*path) {

struct dirent \*entry;

DIR \*dp = opendir(path);

if (dp == NULL) {

perror("opendir");

return;

}

while ((entry = readdir(dp))) {

printf("%s\n", entry->d\_name);

}

closedir(dp);

}

int main(int argc, char \*argv[]) {

const char \*path = (argc > 1) ? argv[1] : ".";

printf("Listing directory: %s\n", path);

list\_directory(path);

return 0;

}

28.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void grep\_pattern\_in\_file(const char \*filename, const char \*pattern) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file");

return;

}

char line[256];

int line\_number = 1;

while (fgets(line, sizeof(line), file)) {

if (strstr(line, pattern)) {

printf("Line %d: %s", line\_number, line);

}

line\_number++;

}

fclose(file);

}

int main(int argc, char \*argv[]) {

if (argc != 3) {

fprintf(stderr, "Usage: %s <filename> <pattern>\n", argv[0]);

return 1;

}

const char \*filename = argv[1];

const char \*pattern = argv[2];

grep\_pattern\_in\_file(filename, pattern);

return 0;

}

29. #include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_CHAIRS 3

sem\_t waitingRoom;

sem\_t barberChair;

sem\_t barberSleeping;

sem\_t customerReady;

void \*barber(void \*arg) {

while (1) {

sem\_wait(&customerReady); // Wait for a customer to arrive

sem\_wait(&barberChair); // Acquire access to the barber chair

sem\_post(&waitingRoom); // One waiting room chair becomes free

printf("Barber is cutting hair\n");

sleep(1); // Simulate hair cutting

sem\_post(&barberChair); // Release the barber chair

sem\_post(&barberSleeping); // Barber is ready for the next customer

}

return NULL;

}

void \*customer(void \*arg) {

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

printf("Customer %d arrived\n", id);

if (sem\_trywait(&waitingRoom) == 0) {

sem\_post(&customerReady); // Notify the barber that a customer is ready

sem\_wait(&barberSleeping); // Wait for the barber to be ready

sem\_wait(&barberChair); // Acquire access to the barber chair

printf("Customer %d is getting a haircut\n", id);

sem\_post(&barberChair); // Release the barber chair

} else {

printf("Customer %d leaves, no waiting chairs available\n", id);

}

return NULL;

}

int main() {

pthread\_t barberThread;

pthread\_t customers[10];

int customerIds[10];

sem\_init(&waitingRoom, 0, NUM\_CHAIRS);

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

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

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

pthread\_create(&barberThread, NULL, barber, NULL);

for (int i = 0; i < 10; i++) {

customerIds[i] = i + 1;

pthread\_create(&customers[i], NULL, customer, &customerIds[i]);

sleep(rand() % 3); // Random arrival of customers

}

for (int i = 0; i < 10; i++) {

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

}

pthread\_cancel(barberThread); // End the barber thread for simulation

sem\_destroy(&waitingRoom);

sem\_destroy(&barberChair);

sem\_destroy(&barberSleeping);

sem\_destroy(&customerReady);

return 0;

}

30.

Creat

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

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

printf("Thread created successfully.\n");

return NULL;

}

int main() {

pthread\_t thread;

if (pthread\_create(&thread, NULL, thread\_function, NULL)) {

fprintf(stderr, "Error creating thread\n");

return 1;

}

pthread\_join(thread, NULL);

return 0;

}

Join

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

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

printf("Thread created successfully.\n");

return NULL;

}

int main() {

pthread\_t thread;

if (pthread\_create(&thread, NULL, thread\_function, NULL)) {

fprintf(stderr, "Error creating thread\n");

return 1;

}

pthread\_join(thread, NULL);

return 0;

}

Equal

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

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

pthread\_t \*main\_thread = (pthread\_t \*)arg;

if (pthread\_equal(pthread\_self(), \*main\_thread)) {

printf("This is the main thread.\n");

} else {

printf("This is not the main thread.\n");

}

return NULL;

}

int main() {

pthread\_t main\_thread = pthread\_self();

pthread\_t thread;

if (pthread\_create(&thread, NULL, thread\_function, &main\_thread)) {

fprintf(stderr, "Error creating thread\n");

return 1;

}

pthread\_join(thread, NULL);

return 0;

}

Exit

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

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

printf("Thread is exiting.\n");

pthread\_exit(NULL);

}

int main() {

pthread\_t thread;

if (pthread\_create(&thread, NULL, thread\_function, NULL)) {

fprintf(stderr, "Error creating thread\n");

return 1;

}

pthread\_join(thread, NULL);

return 0;

}

31.

#include <stdio.h>

#define MAX 100

int main() {

int n, frames, pages[MAX], frame[MAX], page\_faults = 0;

int m, k, flag, next = 0;

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

scanf("%d", &n);

printf("Enter the page reference string: ");

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

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

}

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

scanf("%d", &frames);

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

frame[i] = -1; // Initialize frames as empty

}

printf("\nPage reference string:\n");

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

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

}

printf("\n");

printf("\nFrames status after each page reference:\n");

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

flag = 0;

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

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

flag = 1;

break;

}

}

if (flag == 0) {

frame[next] = pages[i];

next = (next + 1) % frames;

page\_faults++;

}

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

if (frame[k] != -1) {

printf("%d ", frame[k]);

} else {

printf("- ");

}

}

printf("\n");

}

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

return 0;

}

32.

#include <stdio.h>

#define MAX 100

int findLRU(int time[], int n) {

int i, minimum = time[0], pos = 0;

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

if (time[i] < minimum) {

minimum = time[i];

pos = i;

}

}

return pos;

}

int main() {

int n, frames, pages[MAX], frame[MAX], counter = 0, time[MAX], flag1, flag2, pos, faults = 0;

printf("Enter number of pages: ");

scanf("%d", &n);

printf("Enter page reference string: ");

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

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

}

printf("Enter number of frames: ");

scanf("%d", &frames);

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

frame[i] = -1;

}

printf("\nPage reference string:\n");

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

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

}

printf("\n");

printf("\nFrames status after each page reference:\n");

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

flag1 = flag2 = 0;

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

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

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

if (flag1 == 0) {

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

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

counter++;

faults++;

frame[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

if (flag2 == 0) {

pos = findLRU(time, frames);

counter++;

faults++;

frame[pos] = pages[i];

time[pos] = counter;

}

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

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

printf("%d ", frame[j]);

} else {

printf("- ");

}

}

printf("\n");

}

printf("\nTotal Page Faults: %d\n", faults);

return 0;

}

33.

#include <stdio.h>

#define MAX 100

int findOptimal(int pages[], int n, int frame[], int frames, int index) {

int pos = -1, farthest = index, i, j;

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

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

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

if (j > farthest) {

farthest = j;

pos = i;

}

break;

}

}

if (j == n) {

return i;

}

}

if (pos == -1) {

return 0;

}

return pos;

}

int main() {

int n, frames, pages[MAX], frame[MAX], faults = 0, flag1, flag2;

printf("Enter number of pages: ");

scanf("%d", &n);

printf("Enter page reference string: ");

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

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

}

printf("Enter number of frames: ");

scanf("%d", &frames);

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

frame[i] = -1;

}

printf("\nPage reference string:\n");

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

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

}

printf("\n");

printf("\nFrames status after each page reference:\n");

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

flag1 = flag2 = 0;

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

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

flag1 = flag2 = 1;

break;

}

}

if (flag1 == 0) {

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

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

faults++;

frame[j] = pages[i];

flag2 = 1;

break;

}

}

}

if (flag2 == 0) {

int pos = findOptimal(pages, n, frame, frames, i);

faults++;

frame[pos] = pages[i];

}

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

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

printf("%d ", frame[j]);

} else {

printf("- ");

}

}

printf("\n");

}

printf("\nTotal Page Faults: %d\n", faults);

return 0;

}

34.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_RECORDS 100

#define MAX\_RECORD\_SIZE 256

typedef struct {

int id;

char data[MAX\_RECORD\_SIZE];

} Record;

void write\_record(FILE \*file, Record record) {

fwrite(&record, sizeof(Record), 1, file);

}

void read\_records(FILE \*file) {

Record record;

rewind(file);

while (fread(&record, sizeof(Record), 1, file)) {

printf("Record ID: %d, Data: %s\n", record.id, record.data);

}

}

void append\_record(const char \*filename, Record new\_record) {

FILE \*file = fopen(filename, "ab");

if (file == NULL) {

perror("Error opening file for appending");

return;

}

write\_record(file, new\_record);

fclose(file);

}

int main() {

const char \*filename = "sequential\_file.dat";

FILE \*file = fopen(filename, "wb+");

if (file == NULL) {

perror("Error creating file");

return 1;

}

Record records[MAX\_RECORDS];

int num\_records;

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

scanf("%d", &num\_records);

getchar(); // Consume newline character

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

printf("Enter data for record %d: ", i + 1);

records[i].id = i + 1;

fgets(records[i].data, MAX\_RECORD\_SIZE, stdin);

records[i].data[strcspn(records[i].data, "\n")] = '\0'; // Remove newline character

write\_record(file, records[i]);

}

printf("\nRecords written to the file:\n");

read\_records(file);

fclose(file);

// Append a new record

Record new\_record = {num\_records + 1, "This is a new appended record"};

append\_record(filename, new\_record);

// Re-open the file to read all records including the new appended one

file = fopen(filename, "rb");

if (file == NULL) {

perror("Error opening file for reading");

return 1;

}

printf("\nRecords after appending a new record:\n");

read\_records(file);

fclose(file);

return 0;

}

35.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

typedef struct {

int indexBlock[MAX\_BLOCKS];

} File;

void createFile(File \*file, int numBlocks) {

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

file->indexBlock[i] = rand() % 1000; // Simulate block number

}

}

void displayFile(File \*file, int numBlocks) {

printf("Index Block:\n");

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

printf("Block %d -> %d\n", i, file->indexBlock[i]);

}

}

int main() {

File file;

int numBlocks;

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

scanf("%d", &numBlocks);

if (numBlocks > MAX\_BLOCKS) {

printf("Number of blocks exceeds the maximum limit.\n");

return 1;

}

createFile(&file, numBlocks);

displayFile(&file, numBlocks);

return 0;

}

36.

#include <stdio.h>

#include <stdlib.h>

typedef struct Block {

int blockNumber;

struct Block \*next;

} Block;

typedef struct {

Block \*firstBlock;

Block \*lastBlock;

} File;

void createFile(File \*file, int numBlocks) {

Block \*current, \*newBlock;

file->firstBlock = file->lastBlock = NULL;

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

newBlock = (Block \*)malloc(sizeof(Block));

newBlock->blockNumber = rand() % 1000; // Simulate block number

newBlock->next = NULL;

if (file->firstBlock == NULL) {

file->firstBlock = newBlock;

} else {

file->lastBlock->next = newBlock;

}

file->lastBlock = newBlock;

}

}

void displayFile(File \*file) {

Block \*current = file->firstBlock;

printf("File blocks:\n");

while (current != NULL) {

printf("Block %d -> ", current->blockNumber);

current = current->next;

}

printf("NULL\n");

}

void deleteFile(File \*file) {

Block \*current = file->firstBlock;

Block \*next;

while (current != NULL) {

next = current->next;

free(current);

current = next;

}

file->firstBlock = file->lastBlock = NULL;

}

int main() {

File file;

int numBlocks;

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

scanf("%d", &numBlocks);

createFile(&file, numBlocks);

displayFile(&file);

deleteFile(&file);

return 0;

}

37.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100

void FCFS(int request[], int n, int head) {

int totalMovement = 0;

printf("Disk sequence:\n");

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

printf("Head moves from %d to %d with seek %d\n", head, request[i], abs(request[i] - head));

totalMovement += abs(request[i] - head);

head = request[i];

}

printf("Total head movement = %d\n", totalMovement);

}

int main() {

int request[MAX\_REQUESTS];

int n, head;

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

scanf("%d", &n);

printf("Enter the requests: ");

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

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

}

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

scanf("%d", &head);

FCFS(request, n, head);

return 0;

}

38.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100

void SCAN(int request[], int n, int head, int direction, int diskSize) {

int totalMovement = 0;

int i, j, temp;

// Sort the request array

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

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

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

temp = request[j];

request[j] = request[j + 1];

request[j + 1] = temp;

}

}

}

int seekSequence[MAX\_REQUESTS + 2]; // +2 for 0 and diskSize boundaries

int size = 0;

// Append the disk boundaries

if (direction == 1) {

seekSequence[size++] = 0;

}

seekSequence[size++] = head;

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

if (request[i] >= head) {

seekSequence[size++] = request[i];

}

}

if (direction == 1) {

seekSequence[size++] = diskSize - 1;

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

if (request[i] < head) {

seekSequence[size++] = request[i];

}

}

} else {

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

if (request[i] < head) {

seekSequence[size++] = request[i];

}

}

}

printf("Disk sequence:\n");

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

printf("Head moves from %d to %d with seek %d\n", seekSequence[i], seekSequence[i + 1], abs(seekSequence[i + 1] - seekSequence[i]));

totalMovement += abs(seekSequence[i + 1] - seekSequence[i]);

}

printf("Total head movement = %d\n", totalMovement);

}

int main() {

int request[MAX\_REQUESTS];

int n, head, direction, diskSize;

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

scanf("%d", &n);

printf("Enter the requests: ");

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

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

}

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

scanf("%d", &head);

printf("Enter the disk size: ");

scanf("%d", &diskSize);

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

scanf("%d", &direction);

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

return 0;

}

39.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100

void C\_SCAN(int request[], int n, int head, int diskSize) {

int totalMovement = 0;

int i, j, temp;

// Sort the request array

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

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

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

temp = request[j];

request[j] = request[j + 1];

request[j + 1] = temp;

}

}

}

int seekSequence[MAX\_REQUESTS + 2]; // +2 for 0 and diskSize boundaries

int size = 0;

seekSequence[size++] = head;

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

if (request[i] >= head) {

seekSequence[size++] = request[i];

}

}

seekSequence[size++] = diskSize - 1; // Go to the end of the disk

seekSequence[size++] = 0; // Jump to the start of the disk

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

if (request[i] < head) {

seekSequence[size++] = request[i];

}

}

printf("Disk sequence:\n");

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

printf("Head moves from %d to %d with seek %d\n", seekSequence[i], seekSequence[i + 1], abs(seekSequence[i + 1] - seekSequence[i]));

totalMovement += abs(seekSequence[i + 1] - seekSequence[i]);

}

printf("Total head movement = %d\n", totalMovement);

}

int main() {

int request[MAX\_REQUESTS];

int n, head, diskSize;

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

scanf("%d", &n);

printf("Enter the requests: ");

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

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

}

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

scanf("%d", &head);

printf("Enter the disk size: ");

scanf("%d", &diskSize);

C\_SCAN(request, n, head, diskSize);

return 0;

}

40.

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

void printFilePermissions(struct stat fileStat) {

printf("File Permissions: \t");

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(int argc, char \*argv[]) {

struct stat fileStat;

if (argc != 2) {

printf("Usage: %s <file\_path>\n", argv[0]);

return 1;

}

if (stat(argv[1], &fileStat) < 0) {

perror("stat");

return 1;

}

printf("Information for %s\n", argv[1]);

printf("---------------------------\n");

printf("File Size: \t\t%ld bytes\n", fileStat.st\_size);

printf("Number of Links: \t%ld\n", fileStat.st\_nlink);

printf("File inode: \t\t%ld\n", fileStat.st\_ino);

printFilePermissions(fileStat);

return 0;

}