**OS Program 1:** Write a program in c to demonstrate process creation and termination

**Theory**: Process creation involves spawning a new process, while process termination involves ending a process's execution. These concepts are essential for multitasking and resource management in operating systems.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h> // for fork() and getpid()

#include <sys/wait.h> // for wait()

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("fork failed");

return 1;

} else if (pid == 0) {

printf("Child process (PID: %d) created.\n", getpid());

exit(0);

} else {

printf("Parent process (PID: %d) created.\n", getpid());

int status;

wait(&status);

printf("Child process terminated with status: %d\n", status);

}

printf("Process terminated.\n");

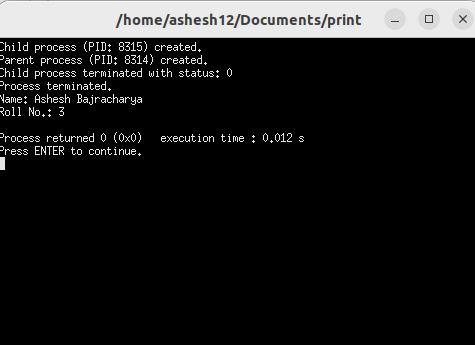
printf("Name: Ashesh Bajracharya\n");

printf("Roll No.: 3\n");

return 0;

}

**Output:**

****

**OS Program 2:** write a program in c to demonstrate thread creation and termination

**Theory**: Thread creation is the process of spawning a new thread within a program's execution space. Threads can be created either by explicitly instantiating a thread object or by invoking a function or method that runs in a separate thread. Thread termination occurs when a thread completes its execution or is explicitly terminated by the program.

**Program**:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_THREADS 2

#define NUM\_ITERATIONS 5

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

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

for (int i = 1; i <= NUM\_ITERATIONS; ++i) {

printf("Thread %d: Iteration %d\n", thread\_num, i);

sleep(1); }

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

pthread\_exit(NULL); // Terminate the thread

}

int main() {

pthread\_t threads[NUM\_THREADS];

int thread\_args[NUM\_THREADS];

int result;

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

thread\_args[i] = i + 1;

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

if (result != 0) {

perror("Thread creation failed");

return 1;}}

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

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

if (result != 0) {

perror("Thread join failed");

return 1;}}

printf("All threads terminated.\n");

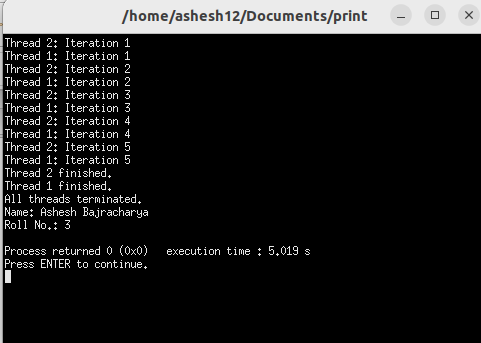
printf("Name: Ashesh Bajracharya\n");

printf("Roll No.: 3\n");

return 0;

}

**Output:**



**OS Program 3:** WAP in C to simulate shared memory concept for IPC.

**Theory:** Shared memory is a mechanism for inter-process communication (IPC) where multiple processes can access the same region of memory. This shared memory segment allows processes to exchange data quickly and efficiently without needing to pass messages between them.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE sizeof(int)

int main() {

int shmid;

key\_t key = 1234; // Unique key for shared memory segment

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

perror("shmget");

exit(1);

}

int \*shmaddr;

if ((shmaddr = shmat(shmid, NULL, 0)) == (int \*) -1) {

perror("shmat");

exit(1);

}

\*shmaddr = 10;

pid\_t pid = fork();

if (pid < 0) {

perror("fork failed");

exit(1);

} else if (pid == 0) {

printf("Child process: Shared value before decrement: %d\n", \*shmaddr);

(\*shmaddr)--;

printf("Child process: Shared value after decrement: %d\n", \*shmaddr);

} else {

printf("Parent process: Shared value before increment: %d\n", \*shmaddr);

(\*shmaddr)++;

printf("Parent process: Shared value after increment: %d\n", \*shmaddr);

wait(NULL);

shmdt(shmaddr);

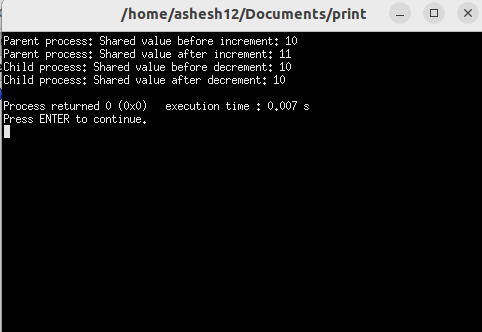
shmctl(shmid, IPC\_RMID, NULL);

}

return 0;

}

**Output:**



**Program 4:** WAP in C to simulate message passing concept for IPC.

**Theory:** Message passing is a mechanism for inter-process communication (IPC) where processes communicate by sending and receiving messages. In this model, processes remain independent and interact by exchanging data through predefined communication channels.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#define BUFFER\_SIZE 100

int main() {

int fd[2]; // File descriptors for the pipe

pid\_t pid;

char message[] = "Hello, this is a message from parent process!";

char buffer[BUFFER\_SIZE];

if (pipe(fd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

pid = fork();

if (pid < 0) {

perror("fork failed");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

close(fd[1]); // Close the write end of the pipe

read(fd[0], buffer, BUFFER\_SIZE); // Read message from the pipe

printf("Child process received message: %s\n", buffer);

close(fd[0]); // Close the read end of the pipe

} else {

close(fd[0]); // Close the read end of the pipe

printf("Parent process sending message: %s\n", message);

write(fd[1], message, strlen(message) + 1); // Write message to the pipe

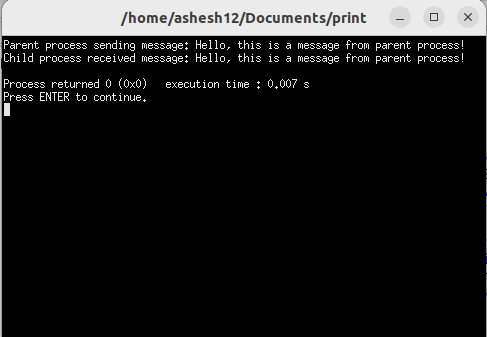
close(fd[1]); // Close the write end of the pipe

}

return 0;

}

**Output:**



**Program 5:** WAP in C to simulate FCFS CPU Scheduling Algorithm

**Theory:** The First-Come, First-Served (FCFS) CPU scheduling algorithm is one of the simplest scheduling algorithms used in operating systems. In FCFS, processes are executed in the order they arrive in the ready queue. When the CPU becomes available, the process at the front of the queue, which has been waiting the longest, is selected for execution.

**Program:**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrival\_time; // Arrival time of the process

int burst\_time; // Burst time of the process

int completion\_time; // Completion time of the process

int turnaround\_time; // Turnaround time of the process

int waiting\_time; // Waiting time of the process

};

void calculateTimes(struct Process processes[], int n) {

int current\_time = 0; // Current time initialized to 0

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

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

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

}

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

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

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

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

}

}

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround\_time, float \*avg\_waiting\_time) {

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

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

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

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

}

\*avg\_turnaround\_time = (float)total\_turnaround\_time / n;

\*avg\_waiting\_time = (float)total\_waiting\_time / n;

}

int main() {

int n; // Number of processes

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

scanf("%d", &n);

struct Process processes[n]; // Array to store processes

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

printf("Enter arrival time of process %d: ", i + 1);

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

printf("Enter burst time of process %d: ", i + 1);

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

processes[i].id = i + 1;

}

calculateTimes(processes, n);

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

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].id, processes[i].arrival\_time,

processes[i].burst\_time, processes[i].completion\_time, processes[i].turnaround\_time,

processes[i].waiting\_time);

}

float avg\_turnaround\_time, avg\_waiting\_time;

calculateAverages(processes, n, &avg\_turnaround\_time, &avg\_waiting\_time);

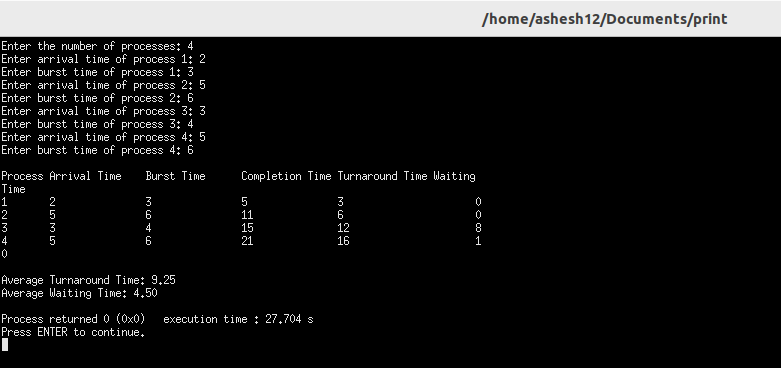
printf("\nAverage Turnaround Time: %.2f", avg\_turnaround\_time);

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output:**



**Program 6:** WAP in C to simulate SJF CPU Scheduling Algorithm

**Theory:** Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time or execution time to execute next. In SJF, the ready queue is ordered based on the length of the CPU burst required by each process. When the CPU becomes available, the process with the shortest burst time is selected for execution.

**Program:**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrival\_time; // Arrival time of the process

int burst\_time; // Burst time of the process

int completion\_time; // Completion time of the process

int turnaround\_time; // Turnaround time of the process

int waiting\_time; // Waiting time of the process

};

void calculateTimes(struct Process processes[], int n) {

int current\_time = 0; // Current time initialized to 0

int remaining\_burst\_time[n]; // Array to store remaining burst time of each process

int completed[n]; // Array to track completion status of each process

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

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

completed[i] = 0; // 0 indicates process is not completed

}

while (1) {

int min\_burst\_time = 9999;

int min\_index = -1;

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

if (processes[i].arrival\_time <= current\_time && completed[i] == 0 &&

remaining\_burst\_time[i] < min\_burst\_time) {

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

min\_index = i;

}}

if (min\_index == -1)

break;

current\_time += remaining\_burst\_time[min\_index];

processes[min\_index].completion\_time = current\_time;

processes[min\_index].turnaround\_time = processes[min\_index].completion\_time - processes[min\_index].arrival\_time;

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

completed[min\_index] = 1;

}}

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround\_time, float \*avg\_waiting\_time) {

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

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

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

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

}

\*avg\_turnaround\_time = (float)total\_turnaround\_time / n;

\*avg\_waiting\_time = (float)total\_waiting\_time / n;

}

int main() {

int n; // Number of processes

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

scanf("%d", &n);

struct Process processes[n]; // Array to store processes

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

printf("Enter arrival time of process %d: ", i + 1);

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

printf("Enter burst time of process %d: ", i + 1);

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

processes[i].id = i + 1;

}

calculateTimes(processes, n);

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

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].id, processes[i].arrival\_time,

processes[i].burst\_time, processes[i].completion\_time, processes[i].turnaround\_time,

processes[i].waiting\_time);

}

average turnaround and waiting time

float avg\_turnaround\_time, avg\_waiting\_time;

calculateAverages(processes, n, &avg\_turnaround\_time, &avg\_waiting\_time);

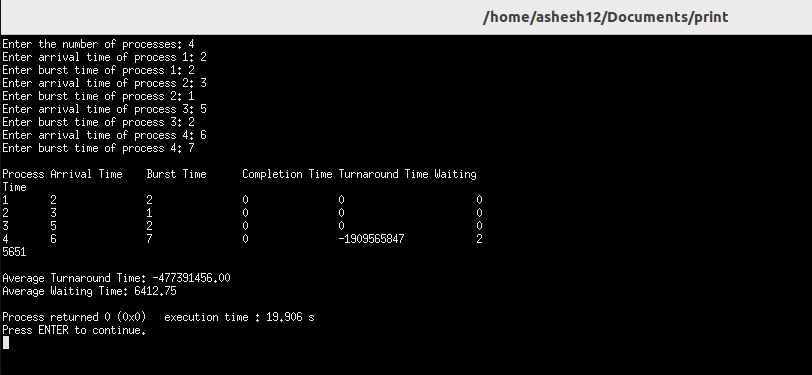
printf("\nAverage Turnaround Time: %.2f", avg\_turnaround\_time);

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output:**



**Program 7:** WAP in C to simulate SRTF CPU Scheduling Algorithm

**Theory:** Shortest Remaining Time First (SRTF), also known as Preemptive Shortest Job First (SJF), is a CPU scheduling algorithm where the scheduler selects the process with the shortest remaining burst time to execute next. Unlike the non-preemptive version of SJF, SRTF allows the scheduler to preempt the currently executing process if a new process with a shorter burst time arrives in the ready queue.

**Program:**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrival\_time; // Arrival time of the process

int burst\_time; // Burst time of the process

int completion\_time; // Completion time of the process

int turnaround\_time; // Turnaround time of the process

int waiting\_time; // Waiting time of the process

};

void calculateTimes(struct Process processes[], int n) {

int current\_time = 0; // Current time initialized to 0

int remaining\_burst\_time[n]; // Array to store remaining burst time of each process

int completed[n]; // Array to track completion status of each process

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

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

completed[i] = 0; // 0 indicates process is not completed

}

while (1) {

int min\_burst\_time = 9999;

int min\_index = -1;

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

if (processes[i].arrival\_time <= current\_time && completed[i] == 0 &&

remaining\_burst\_time[i] < min\_burst\_time) {

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

min\_index = i;

}}

if (min\_index == -1)

break;

remaining\_burst\_time[min\_index]--;

current\_time++;

if (remaining\_burst\_time[min\_index] == 0) {

processes[min\_index].completion\_time = current\_time;

processes[min\_index].turnaround\_time = processes[min\_index].completion\_time - processes[min\_index].arrival\_time;

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

completed[min\_index] = 1;

}}}

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround\_time, float \*avg\_waiting\_time) {

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

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

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

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

}

\*avg\_turnaround\_time = (float)total\_turnaround\_time / n;

\*avg\_waiting\_time = (float)total\_waiting\_time / n;

}

int main() {

int n; // Number of processes

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

scanf("%d", &n);

struct Process processes[n]; // Array to store processes

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

printf("Enter arrival time of process %d: ", i + 1);

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

printf("Enter burst time of process %d: ", i + 1);

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

processes[i].id = i + 1;

}

calculateTimes(processes, n);

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

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].id, processes[i].arrival\_time,

processes[i].burst\_time, processes[i].completion\_time, processes[i].turnaround\_time,

processes[i].waiting\_time);

}

float avg\_turnaround\_time, avg\_waiting\_time;

calculateAverages(processes, n, &avg\_turnaround\_time, &avg\_waiting\_time);

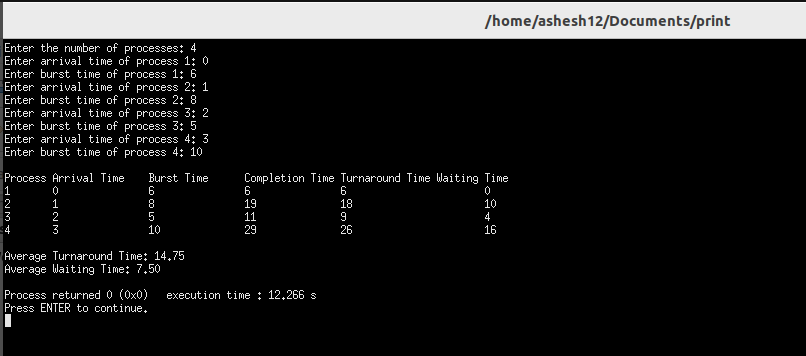
printf("\nAverage Turnaround Time: %.2f", avg\_turnaround\_time);

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output:**



**Program 8:** WAP in C to simulate Round Robin CPU Scheduling Algorithm

**Theory:** Round Robin (RR) is a CPU scheduling algorithm designed for time-sharing systems where each process is allocated a fixed time quantum or time slice. The scheduler cyclically assigns the CPU to processes in the ready queue, allowing each process to execute for a predefined time quantum before preempting it and moving to the next process in the queue.

**Program:**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrival\_time; // Arrival time of the process

int burst\_time; // Burst time of the process

int completion\_time; // Completion time of the process

int turnaround\_time; // Turnaround time of the process

int waiting\_time; // Waiting time of the process

};

void roundRobin(struct Process processes[], int n, int time\_quantum) {

int remaining\_burst\_time[n]; // Array to store remaining burst time of each process

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

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

}

int current\_time = 0; // Current time initialized to 0

int completed\_processes = 0;

while (completed\_processes < n) {

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

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

int execution\_time = (remaining\_burst\_time[i] < time\_quantum) ? remaining\_burst\_time[i] : time\_quantum;

remaining\_burst\_time[i] -= execution\_time;

current\_time += execution\_time;

if (remaining\_burst\_time[i] == 0) {

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

completed\_processes++;

}}}}}

void calculateTimes(struct Process processes[], int n) {

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

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

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

}

}

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround\_time, float \*avg\_waiting\_time) {

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

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

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

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

}

\*avg\_turnaround\_time = (float)total\_turnaround\_time / n;

\*avg\_waiting\_time = (float)total\_waiting\_time / n;

}

int main() {

int n; // Number of processes

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

scanf("%d", &n);

struct Process processes[n]; // Array to store processes

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

printf("Enter arrival time of process %d: ", i + 1);

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

printf("Enter burst time of process %d: ", i + 1);

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

processes[i].id = i + 1;

}

int time\_quantum; // Time quantum for Round Robin algorithm

printf("Enter the time quantum for Round Robin algorithm: ");

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

roundRobin(processes, n, time\_quantum);

calculateTimes(processes, n);

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

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].id, processes[i].arrival\_time,

processes[i].burst\_time, processes[i].completion\_time, processes[i].turnaround\_time,

processes[i].waiting\_time);

}

float avg\_turnaround\_time, avg\_waiting\_time;

calculateAverages(processes, n, &avg\_turnaround\_time, &avg\_waiting\_time);

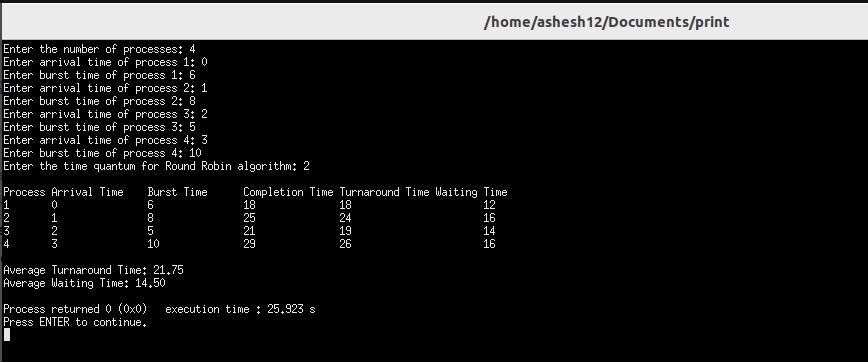
printf("\nAverage Turnaround Time: %.2f", avg\_turnaround\_time);

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output:**



**Program 9:** WAP in C to simulate Non-Preemptive Priority Scheduling Algorithm

**Theory:** Non-Preemptive Priority Scheduling is a CPU scheduling algorithm where each process is assigned a priority value, and the CPU is allocated to the process with the highest priority when it becomes available. Once a process starts executing, it continues until it completes or voluntarily relinquishes the CPU.

**Program:**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrival\_time; // Arrival time of the process

int burst\_time; // Burst time of the process

int priority; // Priority of the process

int completion\_time; // Completion time of the process

int turnaround\_time; // Turnaround time of the process

int waiting\_time; // Waiting time of the process

};

void nonPreemptivePriority(struct Process processes[], int n) {

int current\_time = 0; // Current time initialized to 0

int completed\_processes = 0;

int min\_priority\_process\_index;

while (completed\_processes < n) {

min\_priority\_process\_index = -1;

int min\_priority = 9999;

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

if (processes[i].arrival\_time <= current\_time && processes[i].completion\_time == 0 && processes[i].priority < min\_priority) {

min\_priority = processes[i].priority;

min\_priority\_process\_index = i;

}

}

if (min\_priority\_process\_index != -1) {

current\_time += processes[min\_priority\_process\_index].burst\_time;

processes[min\_priority\_process\_index].completion\_time = current\_time;

processes[min\_priority\_process\_index].turnaround\_time = processes[min\_priority\_process\_index].completion\_time - processes[min\_priority\_process\_index].arrival\_time;

processes[min\_priority\_process\_index].waiting\_time = processes[min\_priority\_process\_index].turnaround\_time - processes[min\_priority\_process\_index].burst\_time;

completed\_processes++;

} else {

current\_time++;}}}

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround\_time, float \*avg\_waiting\_time) {

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

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

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

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

\*avg\_turnaround\_time = (float)total\_turnaround\_time / n;

\*avg\_waiting\_time = (float)total\_waiting\_time / n;}

int main() {

int n; // Number of processes

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

scanf("%d", &n);

struct Process processes[n]; // Array to store processes

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

printf("Enter arrival time of process %d: ", i + 1);

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

printf("Enter burst time of process %d: ", i + 1);

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

printf("Enter priority of process %d: ", i + 1);

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

processes[i].id = i + 1;}

nonPreemptivePriority(processes, n);

float avg\_turnaround\_time, avg\_waiting\_time;

calculateAverages(processes, n, &avg\_turnaround\_time, &avg\_waiting\_time);

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

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\t\t\t%d\n", processes[i].id, processes[i].arrival\_time,

processes[i].burst\_time, processes[i].priority, processes[i].completion\_time, processes[i].turnaround\_time,

processes[i].waiting\_time);}

printf("\nAverage Turnaround Time: %.2f", avg\_turnaround\_time);

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output:**

**Program 10:** WAP in C to simulate Preemptive Priority Scheduling Algorithm

**Theory:** Preemptive Priority Scheduling is a CPU scheduling algorithm where each process is assigned a priority value, and the CPU is allocated to the process with the highest priority that is ready to execute. Unlike non-preemptive priority scheduling, preemptive priority scheduling allows the scheduler to preempt the CPU from a lower-priority process to execute a higher-priority process when it becomes available.

**Program:**

#include <stdio.h>

struct Process {

int id; // Process ID

int arrival\_time; // Arrival time of the process

int burst\_time; // Burst time of the process

int priority; // Priority of the process

int remaining\_time; // Remaining burst time of the process

int completion\_time; // Completion time of the process

int turnaround\_time; // Turnaround time of the process

int waiting\_time; // Waiting time of the process

};

void preemptivePriority(struct Process processes[], int n) {

int current\_time = 0; // Current time initialized to 0

int completed\_processes = 0;

int min\_priority\_process\_index;

while (completed\_processes < n) {

min\_priority\_process\_index = -1;

int min\_priority = 9999;

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

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

min\_priority = processes[i].priority;

min\_priority\_process\_index = i;

}

}

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

current\_time++;

continue;

}

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

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

processes[min\_priority\_process\_index].completion\_time = current\_time + 1;

processes[min\_priority\_process\_index].turnaround\_time = processes[min\_priority\_process\_index].completion\_time - processes[min\_priority\_process\_index].arrival\_time;

processes[min\_priority\_process\_index].waiting\_time = processes[min\_priority\_process\_index].turnaround\_time - processes[min\_priority\_process\_index].burst\_time;

completed\_processes++;

}

current\_time++;

}

}

void calculateAverages(struct Process processes[], int n, float \*avg\_turnaround\_time, float \*avg\_waiting\_time) {

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

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

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

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

}

\*avg\_turnaround\_time = (float)total\_turnaround\_time / n;

\*avg\_waiting\_time = (float)total\_waiting\_time / n;

}

int main() {

int n; // Number of processes

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

scanf("%d", &n);

struct Process processes[n]; // Array to store processes

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

printf("Enter arrival time of process %d: ", i + 1);

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

printf("Enter burst time of process %d: ", i + 1);

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

printf("Enter priority of process %d: ", i + 1);

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

processes[i].id = i + 1;

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

}

preemptivePriority(processes, n);

float avg\_turnaround\_time, avg\_waiting\_time;

calculateAverages(processes, n, &avg\_turnaround\_time, &avg\_waiting\_time);

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

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\t\t\t%d\n", processes[i].id, processes[i].arrival\_time,

processes[i].burst\_time, processes[i].priority, processes[i].completion\_time, processes[i].turnaround\_time,

processes[i].waiting\_time);

}

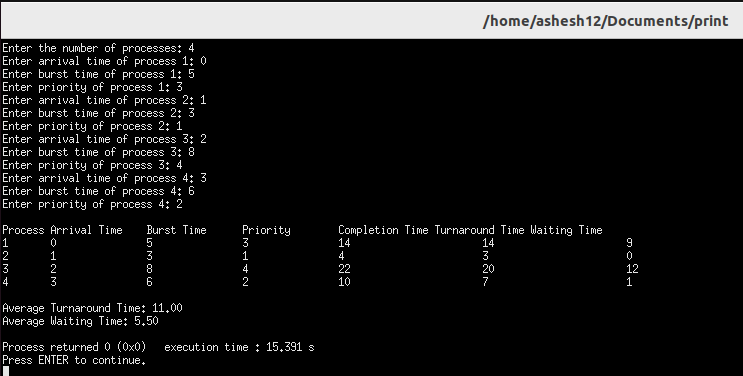
printf("\nAverage Turnaround Time: %.2f", avg\_turnaround\_time);

printf("\nAverage Waiting Time: %.2f\n", avg\_waiting\_time);

return 0;

}

**Output:**



**Program 11:** WAP to implement Bankers Algorithm for multiple type of resources to decide safe/unsafe state.

**Theory:** The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It is designed to ensure that the system never enters a state where it can deadlock, by determining whether allocating a requested resource will result in a safe state. The Banker's Algorithm can handle multiple types of resources and is commonly used in systems with multiple resource types, such as computers with multiple CPU cores or networks with various types of bandwidth.

**Program:**

#include <stdio.h>

#include <stdbool.h>

bool isSafeState(int processes, int resources, int available[], int max[processes][resources], int allocation[processes][resources]) {

int need[processes][resources]; // Need matrix

bool finish[processes]; // Array to track whether a process has finished or not

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

finish[i] = false;

}

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

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

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

}}

int work[resources]; // Available instances of resources

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

work[i] = available[i];

}

int safeSequence[processes]; // Array to store safe sequence

int count = 0; // Count of processes in safe sequence

while (count < processes) {

bool found = false;

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

if (!finish[i]) {

bool canAllocate = true;

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

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

canAllocate = false;

break;}}

if (canAllocate) {

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

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

}

safeSequence[count++] = i;

finish[i] = true;

found = true;}}}

if (!found) {

break;

}

}

if (count == processes) {

printf("Safe Sequence: ");

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

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

}

printf("\n");

return true;

} else {

printf("Unsafe State\n");

return false;

}

}

int main() {

int processes, resources;

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

scanf("%d", &processes);

printf("Enter the number of resource types: ");

scanf("%d", &resources);

int available[resources];

int max[processes][resources];

int allocation[processes][resources];

printf("Enter available instances of each resource type:\n");

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

printf("Resource type %d: ", i + 1);

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

}

printf("Enter maximum instances of each resource type for each process:\n");

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

printf("Process %d: ", i + 1);

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

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

}

}

printf("Enter allocated instances of each resource type for each process:\n");

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

printf("Process %d: ", i + 1);

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

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

}

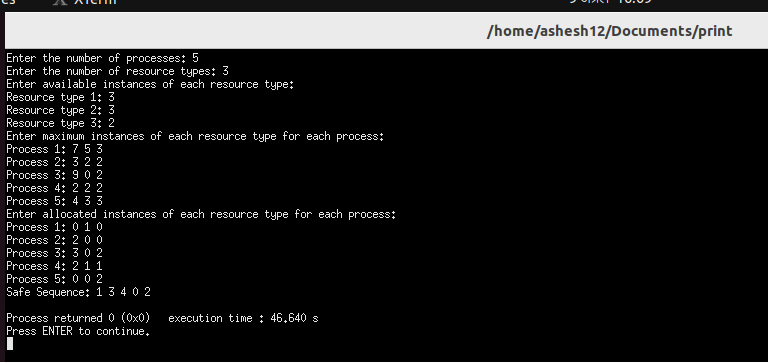
}

isSafeState(processes, resources, available, max, allocation);

return 0;

}

**Output:**



**Program 12:** WAP for deadlock detection in the system having multiple type of resources. The program should list the deadlocked process in case of deadlock detection results

true)

**Theory:** Deadlock detection in a system with multiple types of resources involves identifying whether the system has entered a deadlock state, where processes are unable to proceed because each is waiting for resources held by others.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

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

int max\_claim[MAX\_PROCESSES][MAX\_RESOURCES];

int available[MAX\_RESOURCES];

bool marked[MAX\_PROCESSES];

int num\_processes, num\_resources;

void initialize() {

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

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

printf("Enter the number of resource types: ");

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

printf("Enter the allocated resources for each process:\n");

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

printf("Process %d: ", i + 1);

for (int j = 0; j < num\_resources; j++) {

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

printf("Enter the maximum claim of resources for each process:\n");

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

printf("Process %d: ", i + 1);

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &max\_claim[i][j]);}}

printf("Enter the available instances of each resource type:\n");

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

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

}}

void initializeMarked() {

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

marked[i] = false;

}}

int findUnmarkedProcess() {

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

if (!marked[i]) {

bool canAllocate = true;

for (int j = 0; j < num\_resources; j++) {

if (max\_claim[i][j] - allocated[i][j] > available[j]) {

canAllocate = false;

break;}}

if (canAllocate) {

return i;}}}

return -1;

}

void markAndReleaseResources(int process) {

marked[process] = true;

for (int j = 0; j < num\_resources; j++) {

available[j] += allocated[process][j];

allocated[process][j] = 0;

}

}

bool detectDeadlock() {

initializeMarked();

int process;

while ((process = findUnmarkedProcess()) != -1) {

markAndReleaseResources(process);

}

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

if (!marked[i]) {

return true;

}

}

return false;

}

int main() {

initialize();

if (detectDeadlock()) {

printf("Deadlock detected.\n");

printf("Deadlocked processes: ");

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

if (!marked[i]) {

printf("%d ", i + 1);

}

}

printf("\n");

} else {

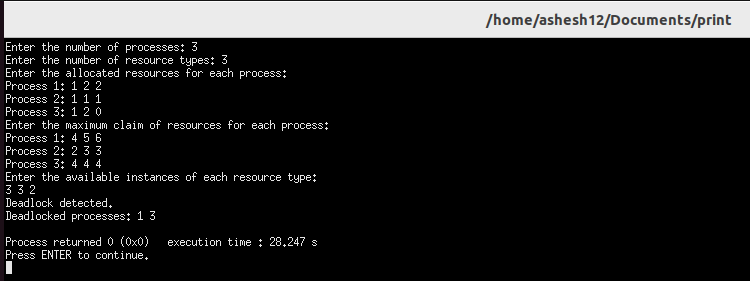
printf("No deadlock detected.\n");

}

return 0;

}

**Output:**



**Program 13:** WAP in C to simulate FIFO Page Replacement Algorithm

**Theory:** The FIFO (First-In, First-Out) Page Replacement Algorithm is one of the simplest and oldest algorithms used in memory management, particularly in virtual memory systems. It operates on the principle that the page that has been in memory the longest is the one selected for replacement.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

bool isPagePresent(int page, int frames[], int num\_frames) {

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

if (frames[i] == page) {

return true;

}}

return false;

}

int findOldestPage(int age[], int num\_frames) {

int oldest\_index = 0;

int oldest\_age = age[0];

for (int i = 1; i < num\_frames; i++) {

if (age[i] < oldest\_age) {

oldest\_age = age[i];

oldest\_index = i;

}}

return oldest\_index;

}

void fifoPageReplacement(int pages[], int num\_pages, int num\_frames) {

int frames[MAX\_FRAMES]; // Array to store frames

int age[MAX\_FRAMES]; // Array to store age of each frame

int page\_faults = 0; // Counter for page faults

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

frames[i] = -1; // -1 indicates an empty frame

age[i] = 0;

}

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

int page = pages[i];

if (!isPagePresent(page, frames, num\_frames)) {

page\_faults++;

int oldest\_index = findOldestPage(age, num\_frames);

frames[oldest\_index] = page;

}

for (int j = 0; j < num\_frames; j++) {

age[j]++;

}}

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

}

int main() {

int pages[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};

int num\_pages = sizeof(pages) / sizeof(pages[0]);

int num\_frames = 3;

printf("Page sequence: ");

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

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

}

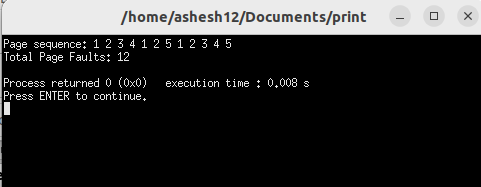
printf("\n");

fifoPageReplacement(pages, num\_pages, num\_frames);

return 0;

}

**Output:**



**Program 14:** WAP in C to simulate Optimal Page Replacement Algorithm

**Theory:** The Optimal Page Replacement Algorithm, also known as the MIN or Belady's optimal algorithm, is a theoretical page replacement algorithm used to study the performance of other page replacement algorithms. It's not practically implementable due to its requirement of future knowledge about page accesses, which is not feasible in real systems.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 100

bool isPagePresent(int page, int frames[], int num\_frames) {

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

if (frames[i] == page) {

return true;

}}

return false;

}

int findFutureUse(int pages[], int num\_pages, int current\_index, int frames[], int num\_frames) {

int future\_index = current\_index;

int farthest = -1;

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

int j;

for (j = current\_index + 1; j < num\_pages; j++) {

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

if (j > farthest) {

farthest = j;

future\_index = j;

}

break;

}}

if (j == num\_pages) {

return i;

}}

return future\_index;

}

void optimalPageReplacement(int pages[], int num\_pages, int num\_frames) {

int frames[MAX\_FRAMES]; // Array to store frames

int page\_faults = 0; // Counter for page faults

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

frames[i] = -1; // -1 indicates an empty frame

}

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

int page = pages[i];

if (!isPagePresent(page, frames, num\_frames)) {

page\_faults++;

int replace\_index = findFutureUse(pages, num\_pages, i, frames, num\_frames);

frames[replace\_index] = page;

}}

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

}

int main() {

int pages[MAX\_PAGES] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int num\_pages = 13;

int num\_frames = 3;

printf("Page sequence: ");

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

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

}

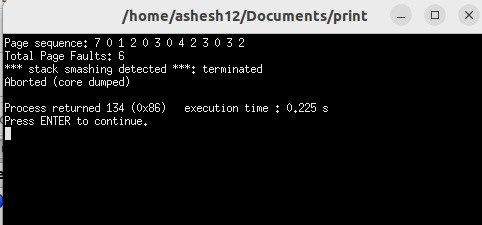
printf("\n");

optimalPageReplacement(pages, num\_pages, num\_frames);

return 0;

}

**Output:**



**Program 15:** WAP in C to simulate LRU Page Replacement Algorithm

**Theory:** The Least Recently Used (LRU) Page Replacement Algorithm is a commonly used algorithm in operating systems for managing memory. It works on the principle that pages that have been least recently accessed are the ones that are least likely to be used in the near future. When a page needs to be replaced, the one that has not been accessed for the longest period of time is selected for replacement.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 100

bool isPagePresent(int page, int frames[], int num\_frames) {

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

if (frames[i] == page) {

return true;

}}

return false;

}

int findLRUIndex(int ages[], int num\_frames) {

int min\_age = ages[0];

int min\_index = 0;

for (int i = 1; i < num\_frames; i++) {

if (ages[i] < min\_age) {

min\_age = ages[i];

min\_index = i;

}}

return min\_index;

}

void lruPageReplacement(int pages[], int num\_pages, int num\_frames) {

int frames[MAX\_FRAMES]; // Array to store frames

int ages[MAX\_FRAMES]; // Array to store age of each frame

int page\_faults = 0; // Counter for page faults

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

frames[i] = -1; // -1 indicates an empty frame

ages[i] = 0;

}

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

int page = pages[i];

if (!isPagePresent(page, frames, num\_frames)) {

page\_faults++;

int lru\_index = findLRUIndex(ages, num\_frames);

frames[lru\_index] = page;

for (int j = 0; j < num\_frames; j++) {

if (j == lru\_index) {

ages[j] = 0;

} else {

ages[j]++;

}}

} else {

for (int j = 0; j < num\_frames; j++) {

if (frames[j] == page) {

ages[j] = 0;

} else {

ages[j]++;

}}}}

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

}

int main() {

int pages[MAX\_PAGES] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int num\_pages = 13;

int num\_frames = 3;

printf("Page sequence: ");

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

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

}

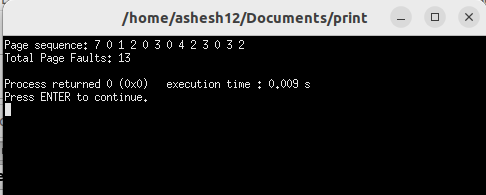
printf("\n");

lruPageReplacement(pages, num\_pages, num\_frames);

return 0;

}

**Output:**



**Program 16:** WAP in C to simulate Second Chance Page Replacement Algorithm

**Theory:** The Second Chance Page Replacement Algorithm is an improvement over the basic FIFO (First-In, First-Out) page replacement algorithm. It aims to reduce the negative impact of FIFO's tendency to replace frequently accessed pages, which may lead to poor performance. Second Chance Algorithm adds a "bit" to each page's entry in the page table to track whether the page has been accessed recently.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 100

typedef struct {

int page;

bool reference\_bit;

} PageEntry;

bool isPagePresent(int page, PageEntry frames[], int num\_frames) {

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

if (frames[i].page == page) {

return true;

}}

return false;

}

void secondChancePageReplacement(int pages[], int num\_pages, int num\_frames) {

PageEntry frames[MAX\_FRAMES]; // Array to store frames

int page\_faults = 0; // Counter for page faults

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

frames[i].page = -1; // -1 indicates an empty frame

frames[i].reference\_bit = false;

}

int hand = 0; // Hand pointer for scanning frames

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

int page = pages[i];

if (!isPagePresent(page, frames, num\_frames)) {

page\_faults++;

while (true) {

if (!frames[hand].reference\_bit) {

frames[hand].page = page;

break;

} else {

frames[hand].reference\_bit = false;

hand = (hand + 1) % num\_frames;}}

hand = (hand + 1) % num\_frames;}

for (int j = 0; j < num\_frames; j++) {

if (frames[j].page == page) {

frames[j].reference\_bit = true;

break;

}}}

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

}

int main() {

int pages[MAX\_PAGES] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int num\_pages = 13;

int num\_frames = 3;

printf("Page sequence: ");

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

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

}

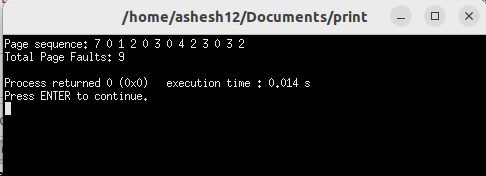
printf("\n");

secondChancePageReplacement(pages, num\_pages, num\_frames);

return 0;

}

**Output:**



**Program 17:** WAP in C to simulate LFU Page Replacement Algorithm

**Theory:** The Least Frequently Used (LFU) Page Replacement Algorithm is a method used in computer operating systems to manage memory and reduce the number of page faults. It works on the principle of replacing the page in memory that has been least frequently accessed.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 100

typedef struct {

int page;

int frequency;

} PageEntry;

bool isPagePresent(int page, PageEntry frames[], int num\_frames) {

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

if (frames[i].page == page) {

return true;

}}

return false;

}

void lfuPageReplacement(int pages[], int num\_pages, int num\_frames) {

PageEntry frames[MAX\_FRAMES]; // Array to store frames

int page\_faults = 0; // Counter for page faults

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

frames[i].page = -1; // -1 indicates an empty frame

frames[i].frequency = 0;

}

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

int page = pages[i];

if (!isPagePresent(page, frames, num\_frames)) {

// If not present, increment page fault counter

page\_faults++;

int min\_frequency\_index = 0;

int min\_frequency = frames[0].frequency;

for (int j = 1; j < num\_frames; j++) {

if (frames[j].frequency < min\_frequency) {

min\_frequency = frames[j].frequency;

min\_frequency\_index = j;

}}

frames[min\_frequency\_index].page = page;

frames[min\_frequency\_index].frequency = 1;

} else {

for (int j = 0; j < num\_frames; j++) {

if (frames[j].page == page) {

frames[j].frequency++;

break;

}}}}

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

}

int main() {

int pages[MAX\_PAGES] = {1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5};

int num\_pages = 15;

int num\_frames = 3;

printf("Page sequence: ");

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

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

}

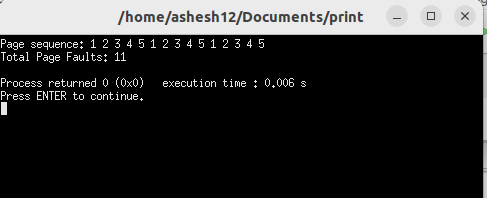
printf("\n");

lfuPageReplacement(pages, num\_pages, num\_frames);

return 0;

}

**Output:**



**Program 18:** WAP to simulate Contiguous File Allocation Technique

**Theory:** Contiguous File Allocation is a technique used in file systems to allocate disk space for files in a contiguous manner. In this method, each file occupies a contiguous block of disk space, meaning that all the blocks needed for a file are located next to each other on the disk.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_BLOCKS 100

// Function to initialize disk blocks

void initializeDisk(bool disk[], int num\_blocks) {

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

disk[i] = false; // false indicates the block is free

}

}

bool allocateBlocks(bool disk[], int num\_blocks, int start\_block, int num\_allocated) {

for (int i = start\_block; i < start\_block + num\_allocated; i++) {

if (disk[i]) {

printf("Error: Block %d is already allocated.\n", i);

return false;

}}

for (int i = start\_block; i < start\_block + num\_allocated; i++) {

disk[i] = true;

}

printf("File allocated from block %d to block %d.\n", start\_block, start\_block + num\_allocated - 1);

return true;

}

void deallocateBlocks(bool disk[], int num\_blocks, int start\_block, int num\_allocated) {

for (int i = start\_block; i < start\_block + num\_allocated; i++) {

disk[i] = false; // Mark blocks as free

}

printf("Blocks deallocated from block %d to block %d.\n", start\_block, start\_block + num\_allocated - 1);

}

void displayDisk(bool disk[], int num\_blocks) {

printf("Disk Status:\n");

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

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

}

printf("\n");

}

int main() {

bool disk[MAX\_BLOCKS]; // Array to represent disk blocks

int num\_blocks; // Total number of disk blocks

int choice, start\_block, num\_allocated;

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

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

initializeDisk(disk, num\_blocks); // Initialize disk blocks

do {

printf("\nContiguous File Allocation Menu:\n");

printf("1. Allocate blocks to file\n");

printf("2. Deallocate blocks from file\n");

printf("3. Display disk status\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter starting block number: ");

scanf("%d", &start\_block);

printf("Enter number of blocks to allocate: ");

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

if (allocateBlocks(disk, num\_blocks, start\_block, num\_allocated)) {

displayDisk(disk, num\_blocks);

}

break;

case 2:

printf("Enter starting block number: ");

scanf("%d", &start\_block);

printf("Enter number of blocks to deallocate: ");

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

deallocateBlocks(disk, num\_blocks, start\_block, num\_allocated);

displayDisk(disk, num\_blocks);

break;

case 3:

displayDisk(disk, num\_blocks);

break;

case 4:

printf("Exiting...\n");

break;

default:

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

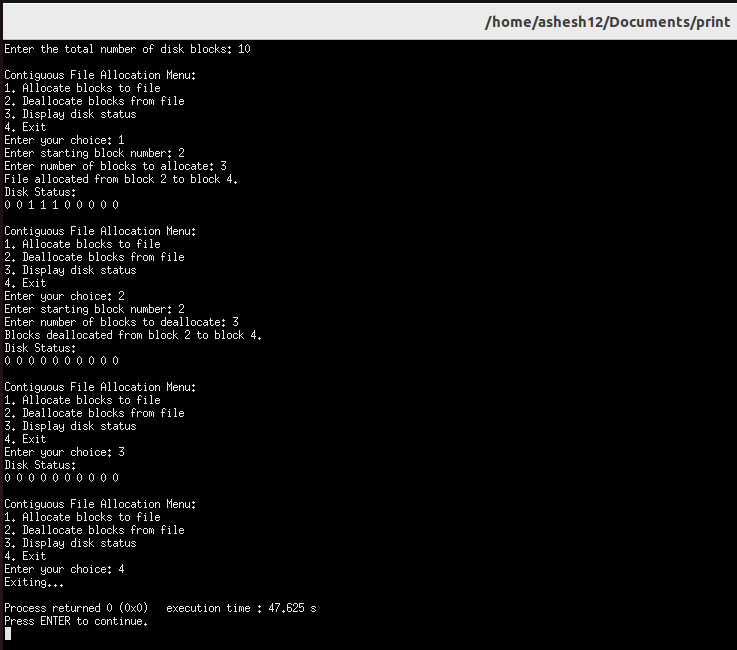
}

} while (choice != 4);

return 0;

}

**Output:**



**Program 19:** WAP to simulate Linked File Allocation Technique

**Theory:** Linked File Allocation is a file allocation technique used in file systems where each file occupies a collection of non-contiguous disk blocks, and these blocks are linked together using pointers. Each block contains a pointer to the next block in the sequence, forming a linked list of blocks for each file.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct Block {

int block\_number; // Block number

struct Block\* next; // Pointer to the next block in the file

} Block;

Block\* createBlock(int block\_number) {

Block\* new\_block = (Block\*)malloc(sizeof(Block));

new\_block->block\_number = block\_number;

new\_block->next = NULL;

return new\_block;

}

void displayFile(Block\* head) {

Block\* current = head;

printf("Blocks in the file: ");

while (current != NULL) {

printf("%d ", current->block\_number);

current = current->next;

}

printf("\n");

}

void allocateBlock(Block\*\* head, int block\_number) {

Block\* new\_block = createBlock(block\_number);

if (\*head == NULL) {

\*head = new\_block;

} else {

Block\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = new\_block;

}

}

void deallocateFile(Block\*\* head) {

Block\* current = \*head;

while (current != NULL) {

Block\* temp = current;

current = current->next;

free(temp);

}

\*head = NULL;

}

int main() {

Block\* file1 = NULL;

Block\* file2 = NULL;

printf("Allocating blocks to file1...\n");

allocateBlock(&file1, 1);

allocateBlock(&file1, 2);

allocateBlock(&file1, 3);

displayFile(file1);

printf("Allocating blocks to file2...\n");

allocateBlock(&file2, 4);

allocateBlock(&file2, 5);

displayFile(file2);

printf("Deallocating blocks from file1...\n");

deallocateFile(&file1);

displayFile(file1);

printf("Deallocating blocks from file2...\n");

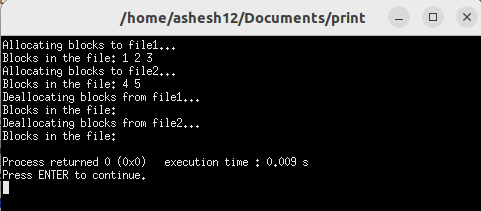
deallocateFile(&file2);

displayFile(file2);

return 0;

}

**Output:**



**Program 20:** WAP to simulate File Allocation using File Allocation Table

**Theory:** File Allocation Table (FAT) is a file system structure used in various file systems, including FAT12, FAT16, FAT32, and exFAT. It's primarily used for organizing and managing file allocation on disk storage.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#define MAX\_FILENAME\_LENGTH 20

#define MAX\_BLOCKS 100

typedef struct {

char filename[MAX\_FILENAME\_LENGTH];

int start\_block;

int num\_blocks;

} FileEntry;

void initializeFAT(FileEntry FAT[], int num\_entries) {

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

strcpy(FAT[i].filename, ""); // Empty filename indicates an unused entry

FAT[i].start\_block = -1; // -1 indicates an unused entry

FAT[i].num\_blocks = 0;

}

}

void displayFAT(FileEntry FAT[], int num\_entries) {

printf("File Allocation Table (FAT):\n");

printf("Filename\tStart Block\tNum Blocks\n");

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

if (strlen(FAT[i].filename) > 0) {

printf("%s\t\t%d\t\t%d\n", FAT[i].filename, FAT[i].start\_block, FAT[i].num\_blocks);

}

}

}

void allocateBlocksToFile(FileEntry FAT[], int num\_entries, char filename[], int start\_block, int num\_blocks) {

int index = -1;

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

if (strlen(FAT[i].filename) == 0) {

index = i;

break;

}}

if (index != -1) {

strcpy(FAT[index].filename, filename);

FAT[index].start\_block = start\_block;

FAT[index].num\_blocks = num\_blocks;

printf("Blocks allocated to file '%s': Start Block: %d, Num Blocks: %d\n", filename, start\_block, num\_blocks);

} else {

printf("Error: FAT is full. Cannot allocate blocks to file '%s'.\n", filename);

}}

void deallocateBlocksFromFile(FileEntry FAT[], int num\_entries, char filename[]) {

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

if (strcmp(FAT[i].filename, filename) == 0) {

printf("Blocks deallocated from file '%s': Start Block: %d, Num Blocks: %d\n", filename, FAT[i].start\_block, FAT[i].num\_blocks);

strcpy(FAT[i].filename, "");

FAT[i].start\_block = -1;

FAT[i].num\_blocks = 0;

return;

}}

printf("Error: File '%s' not found in FAT. Cannot deallocate blocks.\n", filename);

}

int main() {

int num\_entries = 10; // Number of entries in the FAT

FileEntry FAT[num\_entries];

initializeFAT(FAT, num\_entries);

allocateBlocksToFile(FAT, num\_entries, "file1", 0, 3);

allocateBlocksToFile(FAT, num\_entries, "file2", 3, 2);

displayFAT(FAT, num\_entries);

deallocateBlocksFromFile(FAT, num\_entries, "file1");

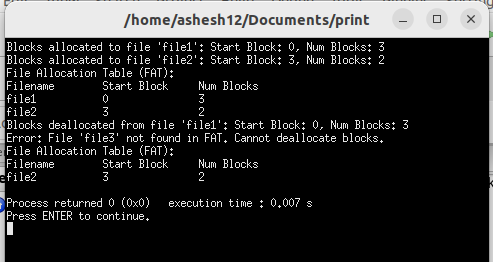
deallocateBlocksFromFile(FAT, num\_entries, "file3");

displayFAT(FAT, num\_entries);

return 0;

}

**Output:**



**Program 21:** WAP to implement File Allocation using Inode

**Theory:** File allocation using an inode (index node) is a method used in Unix-based file systems such as ext2, ext3, and ext4. In this approach, each file is associated with an inode, which contains metadata about the file and pointers to the data blocks that store the file's contents.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define MAX\_FILENAME\_LENGTH 20

#define MAX\_BLOCKS 100

typedef struct {

char filename[MAX\_FILENAME\_LENGTH];

int size;

int blocks[MAX\_BLOCKS];

} Inode;

void initializeInode(Inode\* inode) {

strcpy(inode->filename, "");

inode->size = 0;

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

inode->blocks[i] = -1; // -1 indicates an unused block

}

}

void displayInode(Inode\* inode) {

printf("Filename: %s\n", inode->filename);

printf("Size: %d bytes\n", inode->size);

printf("Blocks allocated: ");

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

if (inode->blocks[i] != -1) {

printf("%d ", inode->blocks[i]);

}}

printf("\n");}

void allocateBlocksToInode(Inode\* inode, char filename[], int size, int block\_numbers[], int num\_blocks) {

strcpy(inode->filename, filename);

inode->size = size;

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

inode->blocks[i] = block\_numbers[i];

}}

void deallocateBlocksFromInode(Inode\* inode) {

initializeInode(inode);}

int main() {

Inode inode;

initializeInode(&inode);

int block\_numbers[] = {0, 1, 2, 3};

allocateBlocksToInode(&inode, "file1", 1024, block\_numbers, 4);

printf("After allocation:\n");

displayInode(&inode);

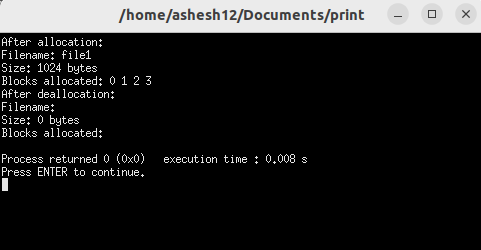
deallocateBlocksFromInode(&inode);

printf("After deallocation:\n");

displayInode(&inode);

return 0;}

**Output:**



**Program 22:** WAP to simulate Free Space Management using Bitmaps

**Theory:** Free space management using bitmaps is a common technique employed in file systems to keep track of available and used disk blocks. In this method, a bitmap is used to represent the allocation status of each disk block, with each bit in the bitmap corresponding to a single disk block.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_BLOCKS 100

typedef struct {

bool bits[MAX\_BLOCKS];

} Bitmap;

void initializeBitmap(Bitmap\* bitmap, int num\_blocks) {

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

bitmap->bits[i] = false; // false indicates a free block

}

}

void displayBitmap(Bitmap\* bitmap, int num\_blocks) {

printf("Bitmap (Free Space Management):\n");

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

printf("%d ", bitmap->bits[i] ? 1 : 0);

}

printf("\n");

}

bool allocateBlock(Bitmap\* bitmap, int block\_number) {

if (bitmap->bits[block\_number]) {

printf("Error: Block %d is already allocated.\n", block\_number);

return false;

} else {

bitmap->bits[block\_number] = true;

printf("Block %d allocated.\n", block\_number);

return true;

}

}

void deallocateBlock(Bitmap\* bitmap, int block\_number) {

if (!bitmap->bits[block\_number]) {

printf("Error: Block %d is already free.\n", block\_number);

} else {

bitmap->bits[block\_number] = false;

printf("Block %d deallocated.\n", block\_number);

}

}

int main() {

int num\_blocks = 10; // Total number of blocks on the disk

Bitmap bitmap;

initializeBitmap(&bitmap, num\_blocks);

printf("Initial Bitmap:\n");

displayBitmap(&bitmap, num\_blocks);

allocateBlock(&bitmap, 2);

allocateBlock(&bitmap, 4);

allocateBlock(&bitmap, 6);

printf("After Allocation:\n");

displayBitmap(&bitmap, num\_blocks);

deallocateBlock(&bitmap, 2);

deallocateBlock(&bitmap, 4);

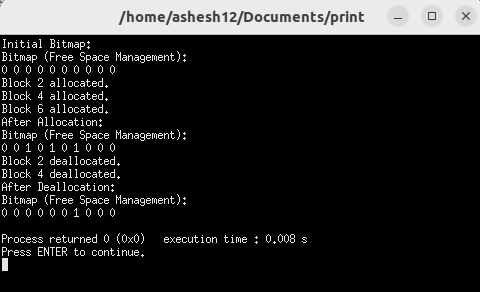
printf("After Deallocation:\n");

displayBitmap(&bitmap, num\_blocks);

return 0;

}

**Output:**



**Program 23:** WAP to simulate Free Space Management using Linked List

**Theory:** Free space management using a linked list is a technique used in file systems to keep track of available disk space. In this method, the free disk blocks are organized into a linked list data structure. Each node in the linked list represents a free disk block, and it contains information about the starting address of the block and its size.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct Node {

int start\_block; // Starting block number of the free space

int num\_blocks; // Number of consecutive free blocks

struct Node\* next; // Pointer to the next node

} Node;

Node\* createNode(int start\_block, int num\_blocks) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->start\_block = start\_block;

new\_node->num\_blocks = num\_blocks;

new\_node->next = NULL;

return new\_node;

}

void insertNode(Node\*\* head, int start\_block, int num\_blocks) {

Node\* new\_node = createNode(start\_block, num\_blocks);

if (\*head == NULL) {

\*head = new\_node;

} else {

Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = new\_node;

}

}

void displayList(Node\* head) {

printf("Free Space Management (Linked List Representation):\n");

printf("Start Block\tNum Blocks\n");

while (head != NULL) {

printf("%d\t\t%d\n", head->start\_block, head->num\_blocks);

head = head->next;

}

}

void allocateBlocks(Node\*\* head, int num\_blocks) {

if (\*head == NULL) {

printf("Error: No free space available.\n");

return;

}

Node\* current = \*head;

while (current != NULL) {

if (current->num\_blocks >= num\_blocks) {

printf("%d blocks allocated starting from block %d.\n", num\_blocks, current->start\_block);

if (current->num\_blocks == num\_blocks) {

if (current == \*head) {

\*head = current->next;

} else {

Node\* temp = \*head;

while (temp->next != current) {

temp = temp->next;

}

temp->next = current->next;}

free(current);

} else {

current->start\_block += num\_blocks;

current->num\_blocks -= num\_blocks;}

return;}

current = current->next;}

printf("Error: Not enough free space available.\n");

}

int main() {

Node\* free\_space = NULL;

insertNode(&free\_space, 0, 3);

insertNode(&free\_space, 4, 2);

insertNode(&free\_space, 7, 4);

displayList(free\_space);

allocateBlocks(&free\_space, 2);

allocateBlocks(&free\_space, 3);

allocateBlocks(&free\_space, 4);

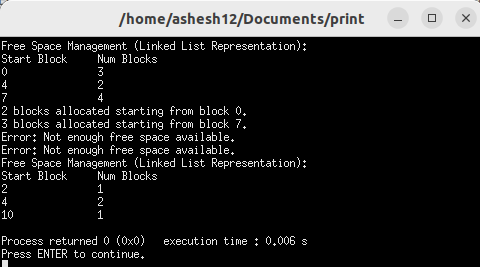
allocateBlocks(&free\_space, 5);

displayList(free\_space);

return 0;

}

**Output:**



**Program 24:** WAP to simulate FCFS Disk Scheduling Algorithm

**Theory:** The FCFS (First-Come, First-Served) Disk Scheduling Algorithm is one of the simplest disk scheduling algorithms used in computer operating systems. In FCFS, the I/O requests are serviced in the order they arrive, without considering the location of the data on the disk or the current position of the disk arm.

**Program:**

#include <stdio.h>

#include <stdlib.h>

int calculateTotalHeadMovement(int initial\_head\_position, int requests[], int num\_requests) {

int total\_head\_movement = 0;

int current\_head\_position = initial\_head\_position;

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

total\_head\_movement += abs(requests[i] - current\_head\_position);

current\_head\_position = requests[i];

}

return total\_head\_movement;

}

int main() {

int initial\_head\_position; // Initial head position of the disk

int num\_requests; // Number of disk I/O requests

int \*requests; // Array to store disk I/O requests

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

scanf("%d", &initial\_head\_position);

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

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

requests = (int \*)malloc(num\_requests \* sizeof(int));

printf("Enter the disk I/O requests:\n");

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

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

}

int total\_head\_movement = calculateTotalHeadMovement(initial\_head\_position, requests, num\_requests);

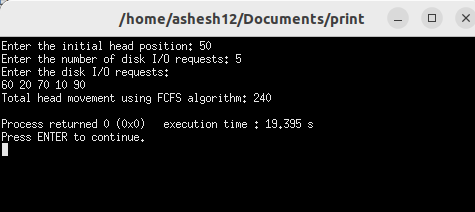
printf("Total head movement using FCFS algorithm: %d\n", total\_head\_movement);

free(requests);

return 0;

}

**Output:**



**Program 25:** WAP to simulate SSTF Disk Scheduling Algorithm

**Theory:** The SSTF (Shortest Seek Time First) Disk Scheduling Algorithm is a disk scheduling technique used to reduce the seek time in computer storage systems. It aims to minimize the distance traveled by the disk arm when seeking data on the disk.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

int findShortestSeekTime(int current\_head\_position, int requests[], bool visited[], int num\_requests) {

int shortest\_seek\_time = INT\_MAX;

int shortest\_seek\_time\_index = -1;

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

if (!visited[i]) {

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

if (seek\_time < shortest\_seek\_time) {

shortest\_seek\_time = seek\_time;

shortest\_seek\_time\_index = i;}}}

return shortest\_seek\_time\_index;

}

int calculateTotalHeadMovement(int initial\_head\_position, int requests[], int num\_requests) {

int total\_head\_movement = 0;

int current\_head\_position = initial\_head\_position;

bool visited[num\_requests];

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

visited[i] = false;

}

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

int shortest\_seek\_time\_index = findShortestSeekTime(current\_head\_position, requests, visited, num\_requests);

visited[shortest\_seek\_time\_index] = true;

total\_head\_movement += abs(requests[shortest\_seek\_time\_index] - current\_head\_position);

current\_head\_position = requests[shortest\_seek\_time\_index];

}

return total\_head\_movement;

}

int main() {

int initial\_head\_position; // Initial head position of the disk

int num\_requests; // Number of disk I/O requests

int \*requests; // Array to store disk I/O requests

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

scanf("%d", &initial\_head\_position);

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

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

requests = (int \*)malloc(num\_requests \* sizeof(int));

printf("Enter the disk I/O requests:\n");

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

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

}

int total\_head\_movement = calculateTotalHeadMovement(initial\_head\_position, requests, num\_requests);

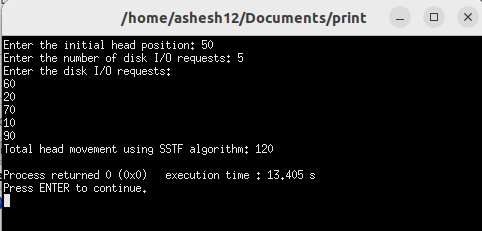
printf("Total head movement using SSTF algorithm: %d\n", total\_head\_movement);

free(requests);

return 0;

}

**Output:**



**Program 26:** WAP to simulate SCAN Disk Scheduling Algorithm

**Theory:** The SCAN (Elevator) Disk Scheduling Algorithm is a disk scheduling technique used to optimize the movement of the disk arm and reduce the seek time in computer storage systems.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void sortRequests(int requests[], int num\_requests) {

for (int i = 0; i < num\_requests - 1; i++) {

for (int j = 0; j < num\_requests - i - 1; j++) {

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

int temp = requests[j];

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

requests[j + 1] = temp;

}}}}

int findClosestRequest(int current\_head\_position, int requests[], bool visited[], int num\_requests, bool moving\_up) {

int closest\_request\_index = -1;

int min\_seek\_distance = \_\_INT\_MAX\_\_;

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

if (!visited[i]) {

int seek\_distance = abs(requests[i] - current\_head\_position);

if (seek\_distance < min\_seek\_distance) {

min\_seek\_distance = seek\_distance;

closest\_request\_index = i;

}}}

if (moving\_up) {

for (int i = closest\_request\_index; i < num\_requests; i++) {

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

closest\_request\_index = i;

break;

}}}

else {

for (int i = closest\_request\_index; i >= 0; i--) {

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

closest\_request\_index = i;

break;

}}}

return closest\_request\_index;

}

int calculateTotalHeadMovement(int initial\_head\_position, int requests[], int num\_requests, bool moving\_up) {

int total\_head\_movement = 0;

int current\_head\_position = initial\_head\_position;

bool visited[num\_requests];

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

visited[i] = false;

}

while (true) {

int closest\_request\_index = findClosestRequest(current\_head\_position, requests, visited, num\_requests, moving\_up);

if (closest\_request\_index == -1) {

break; // No more unvisited requests

}

total\_head\_movement += abs(requests[closest\_request\_index] - current\_head\_position);

current\_head\_position = requests[closest\_request\_index];

visited[closest\_request\_index] = true;

}

return total\_head\_movement;

}

int main() {

int initial\_head\_position; // Initial head position of the disk

int num\_requests; // Number of disk I/O requests

int \*requests; // Array to store disk I/O requests

bool moving\_up; // Flag indicating the direction of head movement

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

scanf("%d", &initial\_head\_position);

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

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

requests = (int \*)malloc(num\_requests \* sizeof(int));

printf("Enter the disk I/O requests:\n");

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

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

}

printf("Enter the direction of head movement (1 for moving up, 0 for moving down): ");

scanf("%d", &moving\_up);

sortRequests(requests, num\_requests);

int total\_head\_movement = calculateTotalHeadMovement(initial\_head\_position, requests, num\_requests, moving\_up);

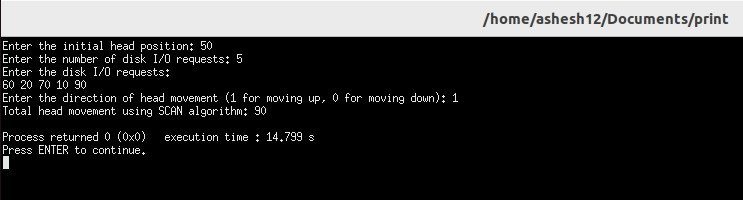
printf("Total head movement using SCAN algorithm: %d\n", total\_head\_movement);

free(requests);

return 0;

}

**Output:**



**Program 27:** WAP to simulate C-SCAN Disk Scheduling Algorithm

**Theory:** The C-SCAN (Circular SCAN) Disk Scheduling Algorithm is an improvement over the SCAN algorithm, which is used to reduce disk arm movement and optimize disk scheduling in computer storage systems.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void sortRequests(int requests[], int num\_requests) {

for (int i = 0; i < num\_requests - 1; i++) {

for (int j = 0; j < num\_requests - i - 1; j++) {

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

int temp = requests[j];

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

requests[j + 1] = temp;

}}}}

int findClosestRequest(int current\_head\_position, int requests[], bool visited[], int num\_requests, bool moving\_up) {

int closest\_request\_index = -1;

int min\_seek\_distance = \_\_INT\_MAX\_\_;

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

if (!visited[i]) {

int seek\_distance = abs(requests[i] - current\_head\_position);

if (seek\_distance < min\_seek\_distance) {

min\_seek\_distance = seek\_distance;

closest\_request\_index = i;

}}}

if (moving\_up) {

for (int i = closest\_request\_index; i < num\_requests; i++) {

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

closest\_request\_index = i;

break;

}}}

else {

for (int i = closest\_request\_index; i >= 0; i--) {

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

closest\_request\_index = i;

break;

}}}

return closest\_request\_index;

}

int calculateTotalHeadMovement(int initial\_head\_position, int requests[], int num\_requests, bool moving\_up) {

int total\_head\_movement = 0;

int current\_head\_position = initial\_head\_position;

bool visited[num\_requests];

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

visited[i] = false;

}

while (true) {

int closest\_request\_index = findClosestRequest(current\_head\_position, requests, visited, num\_requests, moving\_up);

if (closest\_request\_index == -1) {

break; // No more unvisited requests

}

total\_head\_movement += abs(requests[closest\_request\_index] - current\_head\_position);

current\_head\_position = requests[closest\_request\_index];

visited[closest\_request\_index] = true;

if (current\_head\_position == 199 && moving\_up) {

current\_head\_position = 0;}}

return total\_head\_movement;

}

int main() {

int initial\_head\_position; // Initial head position of the disk

int num\_requests; // Number of disk I/O requests

int \*requests; // Array to store disk I/O requests

bool moving\_up; // Flag indicating the direction of head movement

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

scanf("%d", &initial\_head\_position);

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

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

requests = (int \*)malloc(num\_requests \* sizeof(int));

printf("Enter the disk I/O requests:\n");

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

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

}

printf("Enter the direction of head movement (1 for moving up, 0 for moving down): ");

scanf("%d", &moving\_up);

sortRequests(requests, num\_requests);

int total\_head\_movement = calculateTotalHeadMovement(initial\_head\_position, requests, num\_requests, moving\_up);

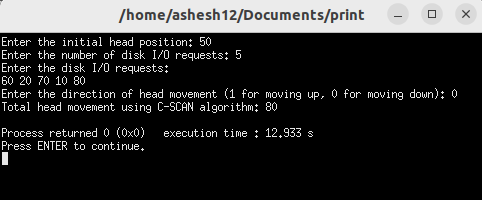
printf("Total head movement using C-SCAN algorithm: %d\n", total\_head\_movement);

free(requests);

return 0;

}

**Output:**



**Program 28:** WAP to simulate LOOK Disk Scheduling Algorithm

**Theory:** The LOOK disk scheduling algorithm is a disk scheduling technique used to optimize the movement of the disk arm and reduce the seek time in computer storage systems. It's a variant of the SCAN algorithm and is particularly suitable for systems with a large number of I/O requests.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void sortRequests(int requests[], int num\_requests) {

// Using bubble sort algorithm

for (int i = 0; i < num\_requests - 1; i++) {

for (int j = 0; j < num\_requests - i - 1; j++) {

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

int temp = requests[j];

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

requests[j + 1] = temp;

}}}}

int findClosestRequest(int current\_head\_position, int requests[], bool visited[], int num\_requests, bool moving\_up) {

int closest\_request\_index = -1;

int min\_seek\_distance = \_\_INT\_MAX\_\_;

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

if (!visited[i]) {

int seek\_distance = abs(requests[i] - current\_head\_position);

if (seek\_distance < min\_seek\_distance) {

min\_seek\_distance = seek\_distance;

closest\_request\_index = i;

}}}

if (moving\_up) {

for (int i = closest\_request\_index; i < num\_requests; i++) {

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

closest\_request\_index = i;

break;

}}}

else {

for (int i = closest\_request\_index; i >= 0; i--) {

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

closest\_request\_index = i;

break;

}}}

return closest\_request\_index;}

int calculateTotalHeadMovement(int initial\_head\_position, int requests[], int num\_requests, bool moving\_up) {

int total\_head\_movement = 0;

int current\_head\_position = initial\_head\_position;

bool visited[num\_requests];

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

visited[i] = false;}

while (true) {

int closest\_request\_index = findClosestRequest(current\_head\_position, requests, visited, num\_requests, moving\_up);

if (closest\_request\_index == -1) {

break; // No more unvisited requests}

total\_head\_movement += abs(requests[closest\_request\_index] - current\_head\_position);

current\_head\_position = requests[closest\_request\_index];

visited[closest\_request\_index] = true;

if (moving\_up && closest\_request\_index == num\_requests - 1) {

moving\_up = false;

} else if (!moving\_up && closest\_request\_index == 0) {

moving\_up = true;}}

return total\_head\_movement;

}

int main() {

int initial\_head\_position; // Initial head position of the disk

int num\_requests; // Number of disk I/O requests

int \*requests; // Array to store disk I/O requests

bool moving\_up; // Flag indicating the direction of head movement

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

scanf("%d", &initial\_head\_position);

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

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

requests = (int \*)malloc(num\_requests \* sizeof(int));

printf("Enter the disk I/O requests:\n");

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

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

printf("Enter the direction of head movement (1 for moving up, 0 for moving down): ");

scanf("%d", &moving\_up);

sortRequests(requests, num\_requests);

int total\_head\_movement = calculateTotalHeadMovement(initial\_head\_position, requests, num\_requests, moving\_up);

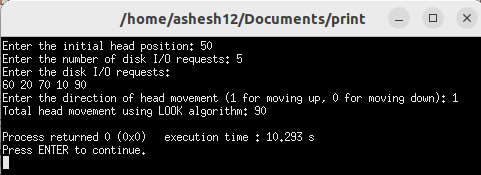
printf("Total head movement using LOOK algorithm: %d\n", total\_head\_movement);

free(requests);

return 0;

}

**Output:**



**Program 29:** WAP to simulate C-LOOK Disk Scheduling Algorithm

**Theory:** The C-LOOK (Circular LOOK) Disk Scheduling Algorithm is a variant of the LOOK disk scheduling algorithm, which is used to optimize disk head movement and reduce seek time in computer storage systems.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void sortRequests(int requests[], int num\_requests) {

// Using bubble sort algorithm

for (int i = 0; i < num\_requests - 1; i++) {

for (int j = 0; j < num\_requests - i - 1; j++) {

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

// Swap requests[j] and requests[j + 1]

int temp = requests[j];

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

requests[j + 1] = temp;

}

}

}

}

int findClosestRequest(int current\_head\_position, int requests[], bool visited[], int num\_requests, bool moving\_up) {

int closest\_request\_index = -1;

int min\_seek\_distance = \_\_INT\_MAX\_\_;

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

if (!visited[i]) {

int seek\_distance = abs(requests[i] - current\_head\_position);

if (seek\_distance < min\_seek\_distance) {

min\_seek\_distance = seek\_distance;

closest\_request\_index = i;

}

}

}

if (moving\_up) {

for (int i = closest\_request\_index; i < num\_requests; i++) {

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

closest\_request\_index = i;

break;

}

}

}

else {

for (int i = closest\_request\_index; i >= 0; i--) {

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

closest\_request\_index = i;

break;

}

}

}

return closest\_request\_index;}

int calculateTotalHeadMovement(int initial\_head\_position, int requests[], int num\_requests, bool moving\_up) {

int total\_head\_movement = 0;

int current\_head\_position = initial\_head\_position;

bool visited[num\_requests];

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

visited[i] = false;}

while (true) {

int closest\_request\_index = findClosestRequest(current\_head\_position, requests, visited, num\_requests, moving\_up);

if (closest\_request\_index == -1) {

break; // No more unvisited requests}

total\_head\_movement += abs(requests[closest\_request\_index] - current\_head\_position);

current\_head\_position = requests[closest\_request\_index];

visited[closest\_request\_index] = true;

if ((moving\_up && closest\_request\_index == num\_requests - 1) || (!moving\_up && closest\_request\_index == 0)) {

current\_head\_position = requests[0];}}

return total\_head\_movement;

}

int main() {

int initial\_head\_position; // Initial head position of the disk

int num\_requests; // Number of disk I/O requests

int \*requests; // Array to store disk I/O requests

bool moving\_up; // Flag indicating the direction of head movement

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

scanf("%d", &initial\_head\_position);

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

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

requests = (int \*)malloc(num\_requests \* sizeof(int));

printf("Enter the disk I/O requests:\n");

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

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

printf("Enter the direction of head movement (1 for moving up, 0 for moving down): ");

scanf("%d", &moving\_up);

sortRequests(requests, num\_requests);

int total\_head\_movement = calculateTotalHeadMovement(initial\_head\_position, requests, num\_requests, moving\_up);

printf("Total head movement using C-LOOK algorithm: %d\n", total\_head\_movement);

free(requests);

return 0;

}

**Output:**

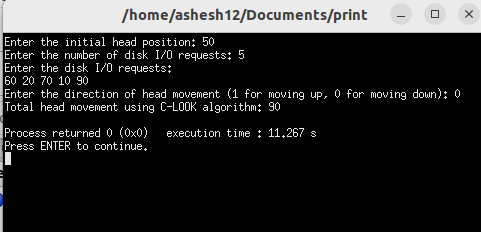


Table of Content

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S.N | Title | Submission  Date | Signature | Remarks |
| 2 | WAP in C to demonstrate the process creation and termination in Linux. | 2024/03/10 |  |  |
| 3 | WAP in C to demonstrate the thread creation and termination in Linux. | 2024/03/10 |  |  |
| 4.a | WAP in C to simulate shared memory concept for IPC. | 2024/03/10 |  |  |
| 4.b | WAP in C to simulate message passing concept for IPC. | 2024/03/10 |  |  |
| 5.a | WAP in C to simulate FCFS CPU Scheduling Algorithm | 2024/03/10 |  |  |
| 5.b | WAP in C to simulate SJF CPU Scheduling Algorithm | 2024/03/10 |  |  |
| 5.c | WAP in C to simulate SRTF CPU Scheduling Algorithm | 2024/03/10 |  |  |
| 5.d | WAP in C to simulate Round Robin CPU Scheduling Algorithm | 2024/03/10 |  |  |
| 5.e | WAP in C to simulate Non-Preemptive Priority Scheduling Algorithm | 2024/03/10 |  |  |
| 5.f | WAP in C to simulate Preemptive Priority Scheduling Algorithm | 2024/03/10 |  |  |
| 6.a | WAP in C to simulate Preemptive Priority Scheduling Algorithm | 2024/03/10 |  |  |
| 6.b | WAP for deadlock detection in the system having multiple type of resources | 2024/03/10 |  |  |
| 7.a | WAP in C to simulate FIFO Page Replacement Algorithm | 2024/03/25 |  |  |
| 7.b | WAP in C to simulate Optimal Page Replacement Algorithm | 2024/03/25 |  |  |
| 7.c | WAP in C to simulate LRU Page Replacement Algorithm | 2024/03/25 |  |  |
| 7.d | WAP in C to simulate Second Chance Page Replacement Algorithm | 2024/03/25 |  |  |
| 7.e | WAP in C to simulate LFU Page Replacement Algorithm | 2024/03/25 |  |  |
| 8.a | WAP to simulate Contiguous File Allocation Technique | 2024/03/25 |  |  |
| 8.b | WAP to simulate Linked File Allocation Technique | 2024/03/25 |  |  |
| 8.c | WAP to simulate File Allocation using File Allocation Table | 2024/03/25 |  |  |
| 8.d | WAP to implement File Allocation using Inode | 2024/03/25 |  |  |
| 9.a | WAP to simulate Free Space Management using Bitmaps | 2024/03/25 |  |  |
| 9.b | WAP to simulate Free Space Management using Linked List | 2024/03/25 |  |  |
| 10.a | WAP to simulate FCFS Disk Scheduling Algorithm | 2024/03/25 |  |  |
| 10.b | WAP to simulate SSTF Disk Scheduling Algorithm | 2024/03/25 |  |  |
| 10.c | WAP to simulate SCAN Disk Scheduling Algorithm | 2024/03/25 |  |  |
| 10.d | WAP to simulate C-SCAN Disk Scheduling Algorithm | 2024/03/25 |  |  |
| 10.e | WAP to simulate LOOK Disk Scheduling Algorithm | 2024/03/25 |  |  |
| 10.f | WAP to simulate C-LOOK Disk Scheduling Algorithm | 2024/03/25 |  |  |