**CSA04 OPERATING SYSTEMS**

**LAB EXPERIMENTS**

**1.Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.**

Program:

#include <stdio.h>

#include <unistd.h> // For fork(), getpid(), getppid()

#include <sys/types.h> // For pid\_t

int main() {

pid\_t pid;

pid = fork();

if (pid < 0) {

perror("Fork failed");

return 1;

} else if (pid == 0) {

// This is the child process

printf("Child Process:\n");

printf(" Process ID (PID): %d\n", getpid());

printf(" Parent Process ID (PPID): %d\n", getppid());

} else {

printf("Parent Process:\n");

printf(" Process ID (PID): %d\n", getpid());

printf(" Parent Process ID (PPID): %d\n", getppid());

printf(" Child Process ID: %d\n", pid);

}

return 0;

}

**2. Identify the system calls to copy the content of one file to another and illustrate the same using a C program.**

Program:

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#define BUFFER\_SIZE 1024

int main() {

int sourceFile, destFile;

char buffer[BUFFER\_SIZE];

ssize\_t bytesRead;

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

if (sourceFile < 0) {

perror("Error opening source file");

return 1;

}

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

if (destFile < 0) {

perror("Error opening/creating destination file");

close(sourceFile);

return 1;

}

while ((bytesRead = read(sourceFile, buffer, BUFFER\_SIZE)) > 0) {

if (write(destFile, buffer, bytesRead) != bytesRead) {

perror("Error writing to destination file");

close(sourceFile);

close(destFile);

return 1;

}

}

if (bytesRead < 0) {

perror("Error reading source file");

}

close(sourceFile);

close(destFile);

printf("File copied successfully!\n");

return 0;

}

**3. Design a CPU scheduling program with C using First Come First Served technique with the following considerations.**

**a. All processes are activated at time 0.**

**b. Assume that no process waits on I/O devices**

program:

#include <stdio.h>

struct Process {

int pid;

int burstTime;

int waitingTime;

int turnAroundTime;

};

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

int totalWaitingTime = 0, totalTurnAroundTime = 0;

processes[0].waitingTime = 0;

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

processes[i].waitingTime = processes[i - 1].waitingTime + processes[i - 1].burstTime;

}

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

processes[i].turnAroundTime = processes[i].waitingTime + processes[i].burstTime;

}

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", processes[i].pid, processes[i].burstTime, processes[i].waitingTime, processes[i].turnAroundTime);

totalWaitingTime += processes[i].waitingTime;

totalTurnAroundTime += processes[i].turnAroundTime;

}

// Print average waiting time and turnaround time

printf("\nAverage Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnAroundTime / n);

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

printf("Enter burst times for each process:\n");

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

processes[i].pid = i + 1; // Assign process IDs

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

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

}

calculateTimes(processes, n);

return 0;

}

**4. Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next**

Program:

#include <stdio.h>

struct Process {

int pid;

int burstTime;

int waitingTime;

int turnAroundTime;

};

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

int completed = 0, currentTime = 0, minIndex, isProcessed[n];

int totalWaitingTime = 0, totalTurnAroundTime = 0;

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

isProcessed[i] = 0;

}

while (completed < n) {

minIndex = -1;

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

if (!isProcessed[i] && (minIndex == -1 || processes[i].burstTime < processes[minIndex].burstTime)) {

minIndex = i;

}

}

processes[minIndex].waitingTime = currentTime;

processes[minIndex].turnAroundTime = currentTime + processes[minIndex].burstTime;

currentTime += processes[minIndex].burstTime;

isProcessed[minIndex] = 1;

completed++;

}

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", processes[i].pid, processes[i].burstTime, processes[i].waitingTime, processes[i].turnAroundTime);

totalWaitingTime += processes[i].waitingTime;

totalTurnAroundTime += processes[i].turnAroundTime;

}

printf("\nAverage Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnAroundTime / n);

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

printf("Enter burst times for each process:\n");

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

processes[i].pid = i + 1; // Assign process IDs

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

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

}

calculateTimes(processes, n);

return 0;

}

**5. Construct a C program to implement preemptive priority scheduling algorithm**

Program:

#include <stdio.h>

typedef struct {

int id;

int burstTime;

int priority;

int waitingTime;

int turnAroundTime;

} Process;

void sortProcessesByPriority(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) {

Process temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

}

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

processes[0].waitingTime = 0;

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

processes[i].waitingTime = processes[i - 1].waitingTime + processes[i - 1].burstTime;

}

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

processes[i].turnAroundTime = processes[i].waitingTime + processes[i].burstTime;

}

}

int main() {

int n;

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

scanf("%d", &n);

Process processes[n];

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

processes[i].id = i + 1;

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

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

}

sortProcessesByPriority(processes, n);

calculateTimes(processes, n);

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

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

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

processes[i].burstTime, processes[i].priority,

processes[i].waitingTime, processes[i].turnAroundTime);

}

return 0;

}

**6. Construct a C program to implement preemptive priority scheduling algorithm**

Program:

#include <stdio.h>

#include <stdbool.h>

typedef struct {

int id; // Process ID

int burstTime; // Burst Time

int priority; // Priority

int arrivalTime; // Arrival Time

int remainingTime; // Remaining Burst Time

int waitingTime; // Waiting Time

int turnAroundTime; // Turnaround Time

} Process;

int getHighestPriorityProcess(Process processes[], int n, int currentTime) {

int highestPriority = -1;

int selectedProcess = -1;

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

if (processes[i].remainingTime > 0 && processes[i].arrivalTime <= currentTime) {

if (selectedProcess == -1 || processes[i].priority < highestPriority) {

highestPriority = processes[i].priority;

selectedProcess = i;

}

}

}

return selectedProcess;

}

int main() {

int n;

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

scanf("%d", &n);

Process processes[n];

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

processes[i].id = i + 1;

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

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

processes[i].remainingTime = processes[i].burstTime;

processes[i].waitingTime = 0;

processes[i].turnAroundTime = 0;

}

int currentTime = 0;

int completed = 0;

// Preemptive Priority Scheduling Logic

while (completed < n) {

int selectedProcess = getHighestPriorityProcess(processes, n, currentTime);

if (selectedProcess != -1) {

processes[selectedProcess].remainingTime--;

currentTime++;

if (processes[selectedProcess].remainingTime == 0) {

completed++;

processes[selectedProcess].turnAroundTime = currentTime - processes[selectedProcess].arrivalTime;

processes[selectedProcess].waitingTime = processes[selectedProcess].turnAroundTime - processes[selectedProcess].burstTime;

}

} else {

// If no process is ready, increment time

currentTime++;

}

}

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

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

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

processes[i].burstTime, processes[i].priority, processes[i].arrivalTime,

processes[i].waitingTime, processes[i].turnAroundTime);

}

return 0;

}

**7. Construct a C program to implement a non-preemptive SJF algorithm.**

Program:

#include <stdio.h>

typedef struct {

int process\_id;

int burst\_time;

int waiting\_time;

int turn\_around\_time;

} Process;

void findWaitingTime(Process proc[], int n) {

proc[0].waiting\_time = 0;

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

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

}

}

void findTurnAroundTime(Process proc[], int n) {

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

proc[i].turn\_around\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

}

void findAvgTime(Process proc[], int n) {

int total\_waiting\_time = 0, total\_turn\_around\_time = 0;

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

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

total\_turn\_around\_time += proc[i].turn\_around\_time;

}

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

printf("Average Turnaround Time: %.2f\n", (float)total\_turn\_around\_time / n);

}

void sjfScheduling(Process proc[], int n) {

Process temp;

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

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

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

temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

}

}

}

findWaitingTime(proc, n);

findTurnAroundTime(proc, n);

findAvgTime(proc, n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

Process proc[n];

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

proc[i].process\_id = i + 1;

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

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

}

sjfScheduling(proc, n);

return 0;

}

**8. Construct a C program to simulate Round Robin scheduling algorithm with C.**

**Program:**

#include <stdio.h>

void roundRobin(int processes[], int n, int bt[], int quantum) {

int wt[n], tat[n], rem\_bt[n];

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

rem\_bt[i] = bt[i];

}

int t = 0; // current time

while (1) {

int done = 1;

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

if (rem\_bt[i] > 0) {

done = 0;

if (rem\_bt[i] > quantum) {

t += quantum;

rem\_bt[i] -= quantum;

} else {

t += rem\_bt[i];

wt[i] = t - bt[i];

tat[i] = t;

rem\_bt[i] = 0;

}

}

}

if (done == 1) break;

}

printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\n");

int total\_wt = 0, total\_tat = 0;

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

total\_wt += wt[i];

total\_tat += tat[i];

printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);

}

printf("\nAverage Waiting Time: %.2f\n", (float)total\_wt / n);

printf("Average Turnaround Time: %.2f\n", (float)total\_tat / n);

}

int main() {

int n, quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

int processes[n], burst\_time[n];

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

processes[i] = i + 1;

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

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

}

printf("Enter time quantum: ");

scanf("%d", &quantum);

roundRobin(processes, n, burst\_time, quantum);

return 0;

}

**9.Illustrate the concept of inter-process communication using shared memory with a C program**.

Program:

#include <stdio.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

key\_t key = 1234;

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

int \*shm\_ptr = (int \*)shmat(shmid, NULL, 0);

if (fork() == 0) {

\*shm\_ptr = 100; // child process writes to shared memory

printf("Child: Shared memory set to %d\n", \*shm\_ptr);

} else {

wait(NULL); // parent waits for child

printf("Parent: Shared memory value is %d\n", \*shm\_ptr);

}

shmdt(shm\_ptr);

shmctl(shmid, IPC\_RMID, NULL); // remove shared memory

return 0;

}

**10.Illustrate the concept of inter-process communication using message queue with a C program.**

Program:

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

struct msgbuf {

long mtype;

char mtext[100];

};

int main() {

key\_t key = 1234;

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

struct msgbuf message;

if (fork() == 0) {

message.mtype = 1;

strcpy(message.mtext, "Hello from child process");

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

} else {

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

printf("Parent received message: %s\n", message.mtext);

msgctl(msgid, IPC\_RMID, NULL); // remove message queue

}

return 0;

}

**11.Illustrate the concept of multithreading using a C program.**

Program.

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

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

printf("Thread %ld is running\n", (long)arg);

sleep(1);

printf("Thread %ld is finished\n", (long)arg);

return NULL;

}

int main() {

pthread\_t threads[3];

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

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

}

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

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

}

return 0;

}

**12.Design a C program to simulate the concept of Dining-Philosophers problem**

Program

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define NUM\_PHILOSOPHERS 5

sem\_t chopsticks[NUM\_PHILOSOPHERS];

void \*philosopher(void \*num) {

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

sem\_wait(&chopsticks[id]);

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

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

sleep(1);

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

sem\_post(&chopsticks[id]);

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

return NULL;

}

int main() {

pthread\_t phil[NUM\_PHILOSOPHERS];

int ids[NUM\_PHILOSOPHERS];

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

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

ids[i] = i;

}

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

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

}

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

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

}

return 0;

}

**13.Construct a C program for implementation of the various memory allocation strategies.**

Program

**14.Construct a C program to organise the file using a single level directory.**

**Program**

#include <stdio.h>

#include <string.h>

struct File {

char name[50];

};

int main() {

struct File files[10];

int count = 0;

strcpy(files[count++].name, "file1.txt");

strcpy(files[count++].name, "file2.txt");

printf("Files in Single-Level Directory:\n");

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

printf("%s\n", files[i].name);

}

return 0;

}

**15.Design a C program to organise the file using a two level directory structure.**

Program

#include <stdio.h>

#include <string.h>

struct File {

char name[50];

};

struct UserDirectory {

char username[50];

struct File files[10];

int fileCount;

};

int main() {

struct UserDirectory user1 = {"user1", {{"file1.txt"}, {"file2.txt"}}, 2};

printf("User: %s\n", user1.username);

printf("Files:\n");

for (int i = 0; i < user1.fileCount; i++) {

printf("%s\n", user1.files[i].name);

}

return 0;

}

**16.Develop a C program for implementing random access file for processing the employee details.**

Program.

#include <stdio.h>

#include <stdlib.h>

struct Employee {

int id;

char name[50];

float salary;

};

int main() {

FILE \*file = fopen("employee.dat", "wb+");

struct Employee e1 = {1, "Alice", 5000};

fwrite(&e1, sizeof(e1), 1, file);

fseek(file, 0, SEEK\_SET);

struct Employee e2;

fread(&e2, sizeof(e2), 1, file);

printf("ID: %d, Name: %s, Salary: %.2f\n", e2.id, e2.name, e2.salary);

fclose(file);

return 0;

}

**17.Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.**

Program.

#include <stdio.h>

#include <stdbool.h>

#define P 5 // Number of processes

#define R 3 // Number of resource types

int allocate[P][R], max[P][R], need[P][R], available[R];

bool finished[P] = {false};

// Function to check if resources can be allocated to process

bool canAllocate(int process) {

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

if (need[process][i] > available[i])

return false;

}

return true;

}

// Display system state

void displayState() {

printf("\nProcess\tAllocation\tMax\t\tNeed\n");

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

printf("P%d\t", i);

for (int j = 0; j < R; j++) printf("%d ", allocate[i][j]);

printf("\t");

for (int j = 0; j < R; j++) printf("%d ", max[i][j]);

printf("\t");

for (int j = 0; j < R; j++) printf("%d ", need[i][j]);

printf("\n");

}

}

// Banker's Algorithm to check safe sequence

bool isSafeState() {

int work[R];

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

work[i] = available[i];

bool finish[P] = {false};

int safeSeq[P];

int count = 0;

while (count < P) {

bool allocated = false;

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

if (!finish[i] && canAllocate(i)) {

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

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

safeSeq[count++] = i;

finish[i] = true;

allocated = true;

}

}

if (!allocated) {

printf("\nSystem is in an UNSAFE state!\n");

return false;

}

}

printf("\nSystem is in a SAFE state.\nSafe Sequence: ");

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

printf("P%d ", safeSeq[i]);

printf("\n");

return true;

}

// Main function

int main() {

printf("Enter Allocation Matrix (%d processes x %d resources):\n", P, R);

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

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

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

printf("Enter Maximum Matrix (%d processes x %d resources):\n", P, R);

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

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

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

printf("Enter Available Resources (%d resources):\n", R);

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

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

// Calculate Need Matrix

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

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

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

displayState();

isSafeState();

return 0;

}i

**18.Construct a C program to simulate producer-consumer problem using semaphores.**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty, full;

pthread\_mutex\_t mutex;

void \*producer(void \*arg) {

int item = 0;

while (1) {

item++;

sem\_wait(&empty); // Wait if buffer is full

pthread\_mutex\_lock(&mutex); // Lock buffer access

buffer[in] = item;

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

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

pthread\_mutex\_unlock(&mutex); // Unlock buffer

sem\_post(&full); // Signal that buffer has data

sleep(1);

}

}

void \*consumer(void \*arg) {

int item;

while (1) {

sem\_wait(&full); // Wait if buffer is empty

pthread\_mutex\_lock(&mutex); // Lock buffer access

item = buffer[out];

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

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

pthread\_mutex\_unlock(&mutex); // Unlock buffer

sem\_post(&empty); // Signal that buffer has space

sleep(2);

}

}

int main() {

pthread\_t prod, cons;

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

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

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

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

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

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

**19.Design a C program to implement process synchronization using mutex locks.**

#include <pthread.h>

#include <stdio.h>

#include <unistd.h>

pthread\_mutex\_t mutex;

int counter = 0;

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

pthread\_mutex\_lock(&mutex); // Lock before accessing the counter

counter++;

printf("Thread %ld: Counter = %d\n", (long)arg, counter);

sleep(1);

pthread\_mutex\_unlock(&mutex); // Unlock after accessing

return NULL;

}

int main() {

pthread\_t threads[5];

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

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

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

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

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

pthread\_mutex\_destroy(&mutex);

return 0;

}

**20.Construct a C program to simulate Reader-Writer problem using Semaphores.**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

sem\_t resource, readCountAccess;

int readCount = 0;

void \*reader(void \*arg) {

sem\_wait(&readCountAccess);

readCount++;

if (readCount == 1)

sem\_wait(&resource); // First reader locks resource

sem\_post(&readCountAccess);

printf("Reader %ld is reading\n", (long)arg);

sleep(1);

sem\_wait(&readCountAccess);

readCount--;

if (readCount == 0)

sem\_post(&resource); // Last reader unlocks resource

sem\_post(&readCountAccess);

return NULL;

}

void \*writer(void \*arg) {

sem\_wait(&resource); // Lock resource for writer

printf("Writer %ld is writing\n", (long)arg);

sleep(2);

sem\_post(&resource); // Unlock resource

return NULL;

}

int main() {

pthread\_t readers[5], writers[2];

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

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

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

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

for (long i = 0; i < 2; i++)

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

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

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

for (int i = 0; i < 2; i++)

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

sem\_destroy(&resource);

sem\_destroy(&readCountAccess);

return 0;

}

**21.Develop a C program to implement the worst fit algorithm of memory management.**

#include <stdio.h>

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

int allocation[n];

// Initially, no process is allocated

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

allocation[i] = -1;

// Pick each process and find the worst fit block

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[j] > blockSize[worstIdx])

worstIdx = j;

}

}

// If a block was found, allocate it

if (worstIdx != -1) {

allocation[i] = worstIdx;

blockSize[worstIdx] -= processSize[i];

}

}

printf("\nProcess 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[] = {100, 500, 200, 300, 600};

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

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

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

worstFit(blockSize, m, processSize, n);

return 0;

}

**22.Construct a C program to implement the best fit algorithm of memory management.**

#include <stdio.h>

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

int allocation[n];

// Initially, no process is allocated

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

allocation[i] = -1;

// Pick each process and find the best fit block

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[j] < blockSize[bestIdx])

bestIdx = j;

}

}

// If a block was found, allocate it

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

printf("\nProcess 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[] = {100, 500, 200, 300, 600};

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

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

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

bestFit(blockSize, m, processSize, n);

return 0;

}

**23.Construct a C program to implement the first fit algorithm of memory management.**

#include <stdio.h>

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("\nProcess 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[] = {100, 500, 200, 300, 600};

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

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

firstFit(blockSize, m, processSize, n);

return 0;

}

**24.design a C program to demonstrate UNIX system calls for file management.**

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#define BUFFER\_SIZE 100

int main() {

int fd;

char write\_buf[] = "Hello, this is a file system call demo!";

char read\_buf[BUFFER\_SIZE];

// Open a file (create if not exists)

fd = open("demo\_file.txt", O\_CREAT | O\_WRONLY, 0644);

if (fd == -1) {

perror("Error opening file");

exit(1);

}

printf("File opened successfully for writing.\n");

// Write to file

if (write(fd, write\_buf, sizeof(write\_buf)) == -1) {

perror("Error writing to file");

close(fd);

exit(1);

}

printf("Data written to file successfully.\n");

close(fd);

// Open file for reading

fd = open("demo\_file.txt", O\_RDONLY);

if (fd == -1) {

perror("Error opening file for reading");

exit(1);

}

printf("File opened successfully for reading.\n");

// Read from file

if (read(fd, read\_buf, BUFFER\_SIZE) == -1) {

perror("Error reading from file");

close(fd);

exit(1);

}

printf("Data read from file: %s\n", read\_buf);

close(fd);

return 0;

}

**25.Construct a C program to implement the I/O system calls of UNIX (fcntl, seek, stat, opendir, readdir)**

#include <stdio.h>

#include <limits.h>

// Function to display allocation results

void displayAllocation(int allocation[], int processSize[], int 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");

}

}

// First Fit Allocation

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("\nFirst Fit Allocation:\n");

displayAllocation(allocation, processSize, n);

}

// Best Fit Allocation

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[j] < blockSize[bestIdx])

bestIdx = j;

}

}

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

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

displayAllocation(allocation, processSize, n);

}

// Worst Fit Allocation

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[j] > blockSize[worstIdx])

worstIdx = j;

}

}

if (worstIdx != -1) {

allocation[i] = worstIdx;

blockSize[worstIdx] -= processSize[i];

}

}

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

displayAllocation(allocation, processSize, n);

}

// Main function

int main() {

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

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

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

// Backup the original block sizes for reuse

int blockSize1[m], blockSize2[m], blockSize3[m];

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

blockSize1[i] = blockSize[i];

blockSize2[i] = blockSize[i];

blockSize3[i] = blockSize[i];

}

// First Fit

firstFit(blockSize1, m, processSize, n);

// Best Fit

bestFit(blockSize2, m, processSize, n);

// Worst Fit

worstFit(blockSize3, m, processSize, n);

return 0;

}

**26. Construct a C program to implement the file management operations.**

Program.

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*file;

char filename[100], data[100];

// Create a file

printf("Enter the filename to create: ");

scanf("%s", filename);

file = fopen(filename, "w");

if (file == NULL) {

printf("Error creating file!\n");

exit(1);

}

printf("Enter data to write into the file: ");

scanf(" %[^\n]", data);

fprintf(file, "%s", data);

fclose(file);

printf("File created and data written successfully.\n");

// Read the file

file = fopen(filename, "r");

if (file == NULL) {

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

exit(1);

}

printf("File content:\n");

while (fgets(data, sizeof(data), file) != NULL)

printf("%s", data);

fclose(file);

// Delete the file

if (remove(filename) == 0)

printf("\nFile deleted successfully.\n");

else

printf("\nError deleting the file.\n");

return 0;

}

**27.Develop a C program for simulating the function of ls UNIX Command.**

**Program.**

#include <stdio.h>

#include <dirent.h>

#include <stdlib.h>

int main() {

struct dirent \*de;

DIR \*dr = opendir(".");

if (dr == NULL) {

printf("Could not open current directory.\n");

return 1;

}

printf("Files and Directories in current directory:\n");

while ((de = readdir(dr)) != NULL)

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

closedir(dr);

return 0;

}

**28.Write a C program for simulation of GREP UNIX command**

**Program.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char filename[100], search[100], line[256];

FILE \*file;

printf("Enter the filename: ");

scanf("%s", filename);

printf("Enter the word to search: ");

scanf("%s", search);

file = fopen(filename, "r");

if (file == NULL) {

printf("Could not open file.\n");

return 1;

}

printf("Lines containing '%s':\n", search);

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

if (strstr(line, search) != NULL)

printf("%s", line);

}

fclose(file);

return 0;

}

**29.Write a C program to simulate the solution of Classical Process Synchronization Problem**

**Program.**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define SIZE 5

int buffer[SIZE];

int in = 0, out = 0;

sem\_t full, empty;

pthread\_mutex\_t mutex;

void \*producer(void \*arg) {

int item;

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

item = rand() % 100;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

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

in = (in + 1) % SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

}

return NULL;

}

void \*consumer(void \*arg) {

int item;

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

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

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

out = (out + 1) % SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

}

return NULL;

}

int main() {

pthread\_t prod, cons;

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

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

sem\_init(&empty, 0, SIZE);

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

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

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&full);

sem\_destroy(&empty);

return 0;

}

**30.Write C programs to demonstrate the following thread related concepts.**

**(i)create (ii) join (iii) equal (iv) exit**

**Program.**

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

void \*threadFunction(void \*arg) {

printf("Thread ID: %ld is running\n", pthread\_self());

pthread\_exit(NULL);

}

int main() {

pthread\_t t1, t2;

// Create Threads

pthread\_create(&t1, NULL, threadFunction, NULL);

pthread\_create(&t2, NULL, threadFunction, NULL);

// Compare Threads

if (pthread\_equal(t1, t2))

printf("Threads are equal.\n");

else

printf("Threads are not equal.\n");

// Join Threads

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

printf("Both threads have finished execution.\n");

return 0;

}

**31. Construct a C program to simulate the First in First Out paging technique of memory management.**

**Program**

#include <stdio.h>

int main() {

int n, f, pages[50], frames[10], index = 0, page\_faults = 0, flag;

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

scanf("%d", &n);

printf("Enter the reference string: ");

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

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

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

scanf("%d", &f);

for (int i = 0; i < f; i++)

frames[i] = -1; // Initialize all frames to -1

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

flag = 0;

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

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

flag = 1; // Page hit

break;

}

}

if (flag == 0) {

frames[index] = pages[i]; // Replace using FIFO

index = (index + 1) % f;

page\_faults++;

}

printf("\nFrames: ");

for (int j = 0; j < f; j++)

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

}

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

return 0;

}

**32.Construct a C program to simulate the Least Recently Used paging technique of memory management.**

**program**

#include <stdio.h>

#include <limits.h>

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

int min = time[0], pos = 0;

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

if (time[i] < min) {

min = time[i];

pos = i;

}

}

return pos;

}

int main() {

int n, f, pages[50], frames[10], time[10], page\_faults = 0, counter = 0;

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

scanf("%d", &n);

printf("Enter the reference string: ");

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

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

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

scanf("%d", &f);

for (int i = 0; i < f; i++)

frames[i] = -1;

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

int flag = 0;

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

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

flag = 1;

time[j] = counter++;

break;

}

}

if (flag == 0) {

int pos = findLRU(time, f);

frames[pos] = pages[i];

time[pos] = counter++;

page\_faults++;

}

printf("\nFrames: ");

for (int j = 0; j < f; j++)

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

}

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

return 0;

}

**33.Construct a C program to simulate the optimal paging technique of memory management**

**program**

#include <stdio.h>

int predict(int pages[], int n, int frames[], int index, int f) {

int farthest = index, pos = -1;

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

int j;

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

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

if (j > farthest) {

farthest = j;

pos = i;

}

break;

}

}

if (j == n) return i;

}

return (pos == -1) ? 0 : pos;

}

int main() {

int n, f, pages[50], frames[10], page\_faults = 0;

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

scanf("%d", &n);

printf("Enter the reference string: ");

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

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

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

scanf("%d", &f);

for (int i = 0; i < f; i++)

frames[i] = -1;

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

int flag = 0;

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

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

flag = 1;

break;

}

}

if (flag == 0) {

int pos = predict(pages, n, frames, i + 1, f);

frames[pos] = pages[i];

page\_faults++;

}

printf("\nFrames: ");

for (int j = 0; j < f; j++)

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

}

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

return 0;

}

**34.Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.**

**Program**

#include <stdio.h>

int main() {

int start, length;

printf("Enter the starting block and length of the file: ");

scanf("%d %d", &start, &length);

printf("Allocated blocks:\n");

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

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

}

printf("\nFile allocated sequentially.\n");

return 0;

}

**35.Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.**

**Program**

#include <stdio.h>

int main() {

int indexBlock, n, blocks[10];

printf("Enter index block: ");

scanf("%d", &indexBlock);

printf("Enter number of blocks: ");

scanf("%d", &n);

printf("Enter block numbers:\n");

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

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

printf("File allocated using Index Block:\nIndex Block: %d\n", indexBlock);

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

printf("Block %d -> %d\n", i + 1, blocks[i]);

}

return 0;

}

**36. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.**

**Program**

#include <stdio.h>

#include <stdlib.h>

struct Block {

int blockNumber;

struct Block\* next;

};

struct File {

struct Block\* head;

struct Block\* tail;

};

void addBlock(struct File\* file, int blockNumber) {

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

newBlock->blockNumber = blockNumber;

newBlock->next = NULL;

if (file->tail == NULL) {

file->head = file->tail = newBlock;

} else {

file->tail->next = newBlock;

file->tail = newBlock;

}

}

void displayFile(struct File\* file) {

struct Block\* temp = file->head;

printf("File Blocks: ");

while (temp != NULL) {

printf("%d -> ", temp->blockNumber);

temp = temp->next;

}

printf("NULL\n");

}

int main() {

struct File file;

file.head = file.tail = NULL;

addBlock(&file, 1);

addBlock(&file, 3);

addBlock(&file, 5);

addBlock(&file, 7);

displayFile(&file);

return 0;

}

**37.Construct a C program to simulate the First Come First Served disk scheduling algorithm.**

**Program**

#include <stdio.h>

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

int seek\_count = 0, distance, cur\_track;

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

cur\_track = requests[i];

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

printf("Total Seek Count: %d\n", seek\_count);

}

int main() {

int n, head;

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

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests: ");

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

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

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

scanf("%d", &head);

FCFS(requests, n, head);

return 0;

}

**38.Design a C program to simulate SCAN disk scheduling algorithm.**

**Program.**

#include <stdio.h>

#include <stdlib.h>

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

int seek\_count = 0;

int left = 0, right = 0;

// Separate requests in left and right of the head

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

if (requests[i] < head) left++;

else right++;

}

int leftArr[left], rightArr[right];

int l = 0, r = 0;

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

if (requests[i] < head) leftArr[l++] = requests[i];

else rightArr[r++] = requests[i];

}

// Sorting the left and right arrays

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

for (int j = 0; j < left - i - 1; j++)

if (leftArr[j] > leftArr[j + 1])

swap(&leftArr[j], &leftArr[j + 1]);

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

for (int j = 0; j < right - i - 1; j++)

if (rightArr[j] < rightArr[j + 1])

swap(&rightArr[j], &rightArr[j + 1]);

// Move to the left end

if (direction == 0) {

for (int i = left - 1; i >= 0; i--) {

seek\_count += abs(leftArr[i] - head);

head = leftArr[i];

}

// Move to the right end

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

seek\_count += abs(rightArr[i] - head);

head = rightArr[i];

}

} else {

// Move to the right end

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

seek\_count += abs(rightArr[i] - head);

head = rightArr[i];

}

// Move to the left end

for (int i = left - 1; i >= 0; i--) {

seek\_count += abs(leftArr[i] - head);

head = leftArr[i];

}

}

printf("Total Seek Count: %d\n", seek\_count);

}

int main() {

int n, head, direction;

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

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests: ");

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

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

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

scanf("%d", &head);

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

scanf("%d", &direction);

SCAN(requests, n, head, direction);

return 0;

}

**39.Develop a C program to simulate C-SCAN disk scheduling algorithm.**

**Program.**

#include <stdio.h>

#include <stdlib.h>

void C\_SCAN(int requests[], int n, int head, int total\_tracks) {

int seek\_count = 0;

int left = 0, right = 0;

// Separate requests in left and right of the head

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

if (requests[i] < head) left++;

else right++;

}

int leftArr[left], rightArr[right];

int l = 0, r = 0;

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

if (requests[i] < head) leftArr[l++] = requests[i];

else rightArr[r++] = requests[i];

}

// Sorting the left and right arrays

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

for (int j = 0; j < left - i - 1; j++)

if (leftArr[j] > leftArr[j + 1])

swap(&leftArr[j], &leftArr[j + 1]);

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

for (int j = 0; j < right - i - 1; j++)

if (rightArr[j] < rightArr[j + 1])

swap(&rightArr[j], &rightArr[j + 1]);

// Move to the right end

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

seek\_count += abs(rightArr[i] - head);

head = rightArr[i];

}

// Return to the beginning

seek\_count += abs(total\_tracks - 1 - head);

head = 0;

// Move from the beginning to the left end

for (int i = left - 1; i >= 0; i--) {

seek\_count += abs(leftArr[i] - head);

head = leftArr[i];

}

printf("Total Seek Count: %d\n", seek\_count);

}

int main() {

int n, head, total\_tracks;

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

scanf("%d", &n);

int requests[n];

printf("Enter the disk requests: ");

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

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

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

scanf("%d", &head);

printf("Enter the total number of tracks: ");

scanf("%d", &total\_tracks);

C\_SCAN(requests, n, head, total\_tracks);

return 0;

}

**40.Illustrate the various File Access Permission and different types of users in Linux.**

**Program**

#include <stdio.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

void print\_permissions(mode\_t mode) {

char perms[10];

perms[0] = (S\_ISDIR(mode)) ? 'd' : '-';

perms[1] = (mode & S\_IRUSR) ? 'r' : '-'; // Owner read

perms[2] = (mode & S\_IWUSR) ? 'w' : '-'; // Owner write

perms[3] = (mode & S\_IXUSR) ? 'x' : '-'; // Owner execute

perms[4] = (mode & S\_IRGRP) ? 'r' : '-'; // Group read

perms[5] = (mode & S\_IWGRP) ? 'w' : '-'; // Group write

perms[6] = (mode & S\_IXGRP) ? 'x' : '-'; // Group execute

perms[7] = (mode & S\_IROTH) ? 'r' : '-'; // Others read

perms[8] = (mode & S\_IWOTH) ? 'w' : '-'; // Others write

perms[9] = (mode & S\_IXOTH) ? 'x' : '-'; // Others execute

perms[10] = '\0'; // Null terminate the string

printf("Permissions: %s\n", perms);

}

int main() {

char filename[100];

struct stat fileStat;

printf("Enter the filename: ");

scanf("%s", filename);

// Get current file stats

if (stat(filename, &fileStat) < 0) {

perror("Error getting file stats");

return 1;

}

// Display current file permissions

printf("Current permissions of %s:\n", filename);

print\_permissions(fileStat.st\_mode);

// Ask the user to change the permissions

printf("Do you want to change the permissions? (y/n): ");

char choice;

scanf(" %c", &choice);

if (choice == 'y' || choice == 'Y') {

int permission;

printf("Enter new permissions (in numeric form e.g. 755): ");

scanf("%d", &permission);

// Change file permissions using chmod system call

if (chmod(filename, permission) < 0) {

perror("Error changing file permissions");

return 1;

}

// Get new file stats and display updated permissions

if (stat(filename, &fileStat) < 0) {

perror("Error getting file stats");

return 1;

}

printf("New permissions of %s:\n", filename);

print\_permissions(fileStat.st\_mode);

}

return 0;

}