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.

#include<stdio.h>

#include<unistd.h>

int main()

{

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

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

return 0;

}

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

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

FILE \*fp1 = fopen("source.txt", "r");

FILE \*fp2 = fopen("destination.txt", "w");

char buffer[1024];

fread(buffer, 1, 1024, fp1);

fwrite(buffer, 1, 1024, fp2);

fclose(fp1);

fclose(fp2);

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

#include <stdio.h>

int main() {

int n = 3, burst\_time[] = {4, 3, 5}, start\_time = 0, total\_tat = 0, total\_wt = 0;

printf("\nPID\tBT\tST\tCT\tTAT\tWT\n");

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

int completion\_time = start\_time + burst\_time[i];

int turnaround\_time = completion\_time;

int waiting\_time = turnaround\_time - burst\_time[i];

printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, burst\_time[i], start\_time,

completion\_time, turnaround\_time, waiting\_time);

total\_tat += turnaround\_time;

total\_wt += waiting\_time;

start\_time = completion\_time;

}

printf("\nAverage TAT: %.2f\nAverage WT: %.2f\n", (float)total\_tat / n, (float)total\_wt / n);

}

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

#include <stdio.h>

#include <limits.h>

#define N 4

int main() {

int bt[] = {6, 8, 7, 3};

int ct[N], tat[N], wt[N], is\_completed[N] = {0};

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

while (completed < N) {

int shortest = -1, min\_bt = INT\_MAX;

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

if (!is\_completed[i] && bt[i] < min\_bt) {

shortest = i;

min\_bt = bt[i];

}

}

if (shortest == -1) {

break;

}

time += bt[shortest];

ct[shortest] = time;

tat[shortest] = ct[shortest];

wt[shortest] = tat[shortest] - bt[shortest];

is\_completed[shortest] = 1;

total\_tat += tat[shortest];

total\_wt += wt[shortest];

completed++;

}

printf("\nPID\tBT\tCT\tTAT\tWT\n");

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

printf("%d\t%d\t%d\t%d\t%d\n", i + 1, bt[i], ct[i], tat[i], wt[i]);

}

printf("\nAverage TAT: %.2f\n", (float)total\_tat / N);

printf("Average WT: %.2f\n", (float)total\_wt / N);

return 0;

}

1. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

#include <stdio.h>

#include <limits.h>

#define N 4

int main() {

int bt[] = {6, 8, 7, 3}, pri[] = {2, 1, 3, 4}, ct[N], tat[N], wt[N], done[N] = {0};

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

while (completed < N) {

int highest = -1, min\_pri = INT\_MAX;

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

if (!done[i] && pri[i] < min\_pri) {

highest = i;

min\_pri = pri[i];

}

}

if (highest == -1) break;

time += bt[highest];

ct[highest] = time;

tat[highest] = ct[highest];

wt[highest] = tat[highest] - bt[highest];

total\_tat += tat[highest];

total\_wt += wt[highest];

done[highest] = 1;

completed++;

}

printf("\nPID\tBT\tPriority\tCT\tTAT\tWT\n");

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

printf("%d\t%d\t%d\t\t%d\t%d\t%d\n", i + 1, bt[i], pri[i], ct[i], tat[i], wt[i]);

}

printf("\nAverage TAT: %.2f\nAverage WT: %.2f\n", (float)total\_tat / N, (float)total\_wt / N);

return 0;

}

1. Construct a C program to implement preemptive priority scheduling algorithm

#include <stdio.h>

struct Process { int id, burstTime, arrivalTime, priority, remainingTime, waitingTime, turnaroundTime; };

int findNext(struct Process p[], int n, int t) {

int idx = -1, pr = 1e9;

for (int i = 0; i < n; i++) if (p[i].remainingTime > 0 && p[i].arrivalTime <= t && p[i].priority < pr) idx = i, pr = p[i].priority;

return idx;

}

int main() {

struct Process p[] = {{1, 5, 0, 2, 5, 0, 0}, {2, 3, 1, 1, 3, 0, 0}, {3, 8, 2, 3, 8, 0, 0}};

int n = 3, t = 0, c = 0; float wt = 0, tat = 0;

while (c < n) {

int idx = findNext(p, n, t);

if (idx == -1) { t++; continue; }

p[idx].remainingTime--; t++;

if (p[idx].remainingTime == 0) {

c++; p[idx].turnaroundTime = t - p[idx].arrivalTime;

p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;

wt += p[idx].waitingTime; tat += p[idx].turnaroundTime;

}

}

printf("\nProcess\tAT\tBT\tP\tWT\tTAT\n");

for (int i = 0; i < n; i++) printf("P%d\t\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrivalTime, p[i].burstTime, p[i].priority, p[i].waitingTime, p[i].turnaroundTime);

printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", wt / n, tat / n);

return 0;

}

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

#include <stdio.h>

struct Process { int id, burstTime, arrivalTime, waitingTime, turnaroundTime; };

int findShortest(struct Process p[], int n, int t) {

int idx = -1, bt = 1e9;

for (int i = 0; i < n; i++) if (p[i].burstTime > 0 && p[i].arrivalTime <= t && p[i].burstTime < bt) idx = i, bt = p[i].burstTime;

return idx;

}

int main() {

struct Process p[] = {{1, 6, 0, 0, 0}, {2, 8, 1, 0, 0}, {3, 7, 2, 0, 0}, {4, 3, 3, 0, 0}};

int n = 4, t = 0, c = 0; float wt = 0, tat = 0;

while (c < n) {

int idx = findShortest(p, n, t);

if (idx == -1) { t++; continue; }

t += p[idx].burstTime; p[idx].turnaroundTime = t - p[idx].arrivalTime;

p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;

wt += p[idx].waitingTime; tat += p[idx].turnaroundTime; p[idx].burstTime = 0; c++;

}

printf("\nProcess\tAT\tBT\tWT\tTAT\n");

for (int i = 0; i < n; i++) printf("P%d\t\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrivalTime, p[i].turnaroundTime - p[i].waitingTime, p[i].waitingTime, p[i].turnaroundTime);

printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", wt / n, tat / n);

return 0;

}

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

#include <stdio.h>

struct Process { int id, burstTime, remainingTime, waitingTime, turnaroundTime; };

int main() {

struct Process p[] = {{1, 6, 6, 0, 0}, {2, 8, 8, 0, 0}, {3, 7, 7, 0, 0}, {4, 3, 3, 0, 0}};

int n = 4, t = 0, tq = 4, c = 0; float wt = 0, tat = 0;

while (c < n) {

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

if (p[i].remainingTime > 0) {

int exec = p[i].remainingTime > tq ? tq : p[i].remainingTime;

t += exec; p[i].remainingTime -= exec;

if (p[i].remainingTime == 0) {

c++; p[i].turnaroundTime = t;

p[i].waitingTime = p[i].turnaroundTime - p[i].burstTime;

wt += p[i].waitingTime; tat += p[i].turnaroundTime;

}

}

}

}

printf("\nProcess\tBT\tWT\tTAT\n");

for (int i = 0; i < n; i++) printf("P%d\t\t%d\t%d\t%d\n", p[i].id, p[i].burstTime, p[i].waitingTime, p[i].turnaroundTime);

printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", wt / n, tat / n);

return 0;

}

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

#include <stdio.h>

int main() {

int sharedMemory[] = {100, 200, 300};

int pid = 1234;

printf("Process %d writes to shared memory:\n", pid);

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

printf("sharedMemory[%d] = %d\n", i, sharedMemory[i]);

}

pid = 5678;

sharedMemory[0] = 400;

sharedMemory[1] = 500;

sharedMemory[2] = 600;

printf("\nProcess %d reads from shared memory:\n", pid);

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

printf("sharedMemory[%d] = %d\n", i, sharedMemory[i]);

}

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_MESSAGE\_SIZE 100

struct Message {

char text[MAX\_MESSAGE\_SIZE];

};

void sendMessage(struct Message\* message) {

printf("Sending message: %s\n", message->text);

}

void receiveMessage(struct Message\* message) {

printf("Received message: %s\n", message->text);

}

int main() {

struct Message message;

strcpy(message.text, "Hello from Process 1!");

sendMessage(&message);

receiveMessage(&message);

return 0;

}

1. Illustrate the concept of multithreading using a C program.

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

void\* thread1\_function(void\* arg) {

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

printf("Thread 1 is running\n");

sleep(1);

}

return NULL;

}

void\* thread2\_function(void\* arg) {

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

printf("Thread 2 is running\n");

sleep(1);

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL, thread1\_function, NULL);

pthread\_create(&thread2, NULL, thread2\_function, NULL);

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

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

return 0;

}

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

#include <stdio.h>

#define NUM\_PHILOSOPHERS 5

void eat(int philosopher) {

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

}

void think(int philosopher) {

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

}

int main() {

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

think(i);

eat(i);

think(i);

}

return 0;

}

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

#include <stdio.h>

void firstFit(int block[], int process[], int m, int n) {

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

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

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

if (block[j] >= process[i]) {

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

block[j] -= process[i];

break;

}

}

}

}

void bestFit(int block[], int process[], int m, int n) {

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

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

int min = 9999, index = -1;

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

if (block[j] >= process[i] && block[j] < min) {

min = block[j];

index = j;

}

}

if (index != -1) {

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

block[index] -= process[i];

}

}

}

void worstFit(int block[], int process[], int m, int n) {

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

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

int max = -1, index = -1;

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

if (block[j] >= process[i] && block[j] > max) {

max = block[j];

index = j;

}

}

if (index != -1) {

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

block[index] -= process[i];

}

}

}

int main() {

int block[] = {100, 200, 50};

int process[] = {50, 150, 20};

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

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

firstFit(block, process, m, n);

int block1[] = {100, 200, 50};

bestFit(block1, process, m, n);

int block2[] = {100, 200, 50};

worstFit(block2, process, m, n);

return 0;

}

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

#include <dirent.h>

#include <stdio.h>

int main() {

DIR \*dir;

struct dirent \*ent;

dir = opendir(".");

if (dir == NULL) {

perror("opendir");

return 1;

}

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

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

}

closedir(dir);

return 0;

}

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

#include <dirent.h>

#include <stdio.h>

#include <string.h>

int main() {

DIR \*dir, \*subdir;

struct dirent \*ent, \*subent;

dir = opendir(".");

if (dir == NULL) {

perror("opendir");

return 1;

}

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

if (ent->d\_name[0] != '.' && strcmp(ent->d\_name, "..") != 0) {

char path[256];

sprintf(path, "./%s", ent->d\_name);

subdir = opendir(path);

if (subdir != NULL) {

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

while ((subent = readdir(subdir)) != NULL) {

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

}

closedir(subdir);

}

}

}

closedir(dir);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id;

char name[30];

float salary;

} Employee;

int main() {

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

if (!file) file = fopen("employee.dat", "wb+");

if (!file) return 1;

int choice, id;

Employee emp;

do {

printf("1. Add Employee\n2. Display Employee\n3. Exit\nChoice: ");

scanf("%d", &choice);

if (choice == 1) {

printf("Enter ID, Name, Salary: ");

scanf("%d %s %f", &emp.id, emp.name, &emp.salary);

fseek(file, (emp.id - 1) \* sizeof(Employee), SEEK\_SET);

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

} else if (choice == 2) {

printf("Enter ID to display: ");

scanf("%d", &id);

fseek(file, (id - 1) \* sizeof(Employee), SEEK\_SET);

if (fread(&emp, sizeof(Employee), 1, file))

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

else

printf("No record found.\n");

}

} while (choice != 3);

fclose(file);

return 0;

}

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

#include <stdio.h>

int main() {

int processes = 5;

int resources = 3;

int available[resources] = {3, 3, 2};

int max[processes][resources] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int allocation[processes][resources] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int need[processes][resources];

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

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

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

}

}

int safeSequence[processes];

printf("Safe sequence: ");

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

safeSequence[i] = i;

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

}

printf("\n");

return 0;

}

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

#include <semaphore.h>

#include <pthread.h>

#include <stdio.h>

sem\_t empty, full;

int buffer = 0;

void\* producer(void\* arg) {

sem\_post(&full);

buffer++;

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

}

void\* consumer(void\* arg) {

sem\_wait(&full);

buffer--;

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

}

int main() {

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

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

pthread\_t p, c;

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

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

pthread\_join(p, NULL);

pthread\_join(c, NULL);

return 0;

}

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

#include <stdio.h>

#include <pthread.h>

pthread\_mutex\_t mutex;

int shared\_resource = 0;

void\* producer(void\* arg) {

pthread\_mutex\_lock(&mutex);

shared\_resource += 1;

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

pthread\_mutex\_unlock(&mutex);

return NULL;

}

void\* consumer(void\* arg) {

pthread\_mutex\_lock(&mutex);

if (shared\_resource > 0) {

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

shared\_resource -= 1;

} else {

printf("Nothing to consume.\n");

}

pthread\_mutex\_unlock(&mutex);

return NULL;

}

int main() {

pthread\_t t1, t2;

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

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

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

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_mutex\_destroy(&mutex);

return 0;

}

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

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t rw\_mutex, mutex;

int read\_count = 0, shared\_data = 0;

void\* reader(void\* arg) {

sem\_wait(&mutex);

if (++read\_count == 1) sem\_wait(&rw\_mutex);

sem\_post(&mutex);

printf("Reader %d reads: %d\n", \*(int\*)arg, shared\_data);

sem\_wait(&mutex);

if (--read\_count == 0) sem\_post(&rw\_mutex);

sem\_post(&mutex);

return NULL;

}

void\* writer(void\* arg) {

sem\_wait(&rw\_mutex);

printf("Writer %d writes: %d\n", \*(int\*)arg, ++shared\_data);

sem\_post(&rw\_mutex);

return NULL;

}

int main() {

pthread\_t r[2], w[2];

int id1 = 1, id2 = 2;

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

pthread\_create(&r[0], NULL, reader, &id1);

pthread\_create(&w[0], NULL, writer, &id1);

pthread\_create(&r[1], NULL, reader, &id2);

pthread\_create(&w[1], NULL, writer, &id2);

pthread\_join(r[0], NULL), pthread\_join(w[0], NULL);

pthread\_join(r[1], NULL), pthread\_join(w[1], NULL);

sem\_destroy(&rw\_mutex), sem\_destroy(&mutex);

return 0;

}

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

#include <stdio.h>

void worstFit(int block[], int process[], int m, int n) {

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

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

int max = -1, index = -1;

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

if (block[j] >= process[i] && block[j] > max) {

max = block[j];

index = j;

}

}

if (index != -1) {

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

block[index] -= process[i];

}

}

}

int main() {

int block[] = {100, 200, 50};

int process[] = {50, 150, 20};

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

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

int block1[] = {100, 200, 50};

int block2[] = {100, 200, 50};

worstFit(block2, process, m, n);

return 0;

}

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

#include <stdio.h>

void bestFit(int block[], int process[], int m, int n) {

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

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

int min = 9999, index = -1;

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

if (block[j] >= process[i] && block[j] < min) {

min = block[j];

index = j;

}

}

if (index != -1) {

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

block[index] -= process[i];

}

}

}

int main() {

int block[] = {100, 200, 50};

int process[] = {50, 150, 20};

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

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

int block1[] = {100, 200, 50};

bestFit(block1, process, m, n);

int block2[] = {100, 200, 50};

return 0;

}

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

#include <stdio.h>

void firstFit(int block[], int process[], int m, int n) {

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

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

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

if (block[j] >= process[i]) {

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

block[j] -= process[i];

break;

}

}

}

}

int main() {

int block[] = {100, 200, 50};

int process[] = {50, 150, 20};

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

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

firstFit(block, process, m, n);

int block1[] = {100, 200, 50};

int block2[] = {100, 200, 50};

return 0;

}

1. Design a C program to demonstrate UNIX system calls for file management.

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

int fd;

char buffer[100];

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

if (fd < 0) {

perror("File creation failed");

return 1;

}

write(fd, "Hello, UNIX!\n", 13);

close(fd);

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

if (fd < 0) {

perror("File open failed");

return 1;

}

int bytes\_read = read(fd, buffer, sizeof(buffer) - 1);

buffer[bytes\_read] = '\0';

printf("File content:\n%s", buffer);

close(fd);

return 0;

}

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

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/stat.h>

#include <dirent.h>

int main() {

int fd = open("example.txt", O\_CREAT | O\_RDWR, 0644);

if (fd < 0) { perror("open failed"); return 1; }

lseek(fd, 0, SEEK\_SET);

struct stat statbuf;

if (stat("example.txt", &statbuf) == 0)

printf("File size: %ld bytes\n", statbuf.st\_size);

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

if (dir) {

struct dirent \*entry;

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

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

closedir(dir);

}

close(fd);

return 0;

}

1. Construct a C program to implement the file management operations.

#include <stdio.h>

int main() {

FILE \*file;

file = fopen("example.txt", "w");

if (file == NULL) {

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

return 1;

}

fprintf(file, "Hello, File Management!\n");

fclose(file);

file = fopen("example.txt", "r");

if (file == NULL) {

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

return 1;

}

char buffer[100];

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

printf("%s", buffer);

}

fclose(file);

return 0;

}

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

#include <stdio.h>

#include <dirent.h>

int main() {

struct dirent \*entry;

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

if (dir == NULL) {

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

return 1;

}

printf("Listing files in current directory:\n");

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

if (entry->d\_name[0] != '.')

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

}

closedir(dir);

return 0;

}

1. Write a C program for simulation of GREP UNIX command

#include <stdio.h>

#include <string.h>

int main() {

FILE \*file;

char line[256], pattern[50], filename[50];

printf("Enter file name: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0';

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

fgets(pattern, sizeof(pattern), stdin);

pattern[strcspn(pattern, "\n")] = '\0';

file = fopen(filename, "r");

if (file == NULL) {

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

return 1;

}

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

if (strstr(line, pattern))

printf("%s", line);

}

fclose(file);

return 0;

}

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

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t empty, full;

int buffer = 0;

void\* producer(void\* arg) {

sem\_wait(&empty);

buffer++;

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

sem\_post(&full);

return NULL;

}

void\* consumer(void\* arg) {

sem\_wait(&full);

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

buffer--;

sem\_post(&empty);

return NULL;

}

int main() {

pthread\_t prod, cons;

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

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

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);

return 0;

}

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

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

#include <stdio.h>

#include <pthread.h>

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

printf("Thread created and running\n");

pthread\_exit(NULL);

}

int main() {

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL, thread\_func, NULL);

pthread\_join(thread1, NULL);

printf("Thread joined\n");

int equal = pthread\_equal(thread1, thread1);

if (equal) {

printf("Threads are equal\n");

} else {

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

}

pthread\_exit(NULL);

return 0;

}

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

#include <stdio.h>

#define MAX 3

void fifo(int page\_ref[], int n) {

int frames[MAX], front = 0, page\_faults = 0;

for (int i = 0; i < MAX; i++) frames[i] = -1;

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

int found = 0;

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

if (frames[j] == page\_ref[i]) {

found = 1;

break;

}

}

if (!found) {

frames[front] = page\_ref[i];

front = (front + 1) % MAX;

page\_faults++;

}

printf("Frames: ");

for (int j = 0; j < MAX; j++) printf("%d ", frames[j]);

printf("\n");

}

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

}

int main() {

int page\_ref[] = {1, 2, 3, 2, 1, 4, 1, 3};

int n = sizeof(page\_ref) / sizeof(page\_ref[0]);

fifo(page\_ref, n);

return 0;

}

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

#include <stdio.h>

#define MAX 3

void lru(int page\_ref[], int n) {

int frames[MAX], time[MAX], page\_faults = 0;

for (int i = 0; i < MAX; i++) { frames[i] = -1; time[i] = -1; }

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

int found = 0, lru\_index = 0;

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

if (frames[j] == page\_ref[i]) {

found = 1; time[j] = i; break;

}

}

if (!found) {

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

if (time[j] == -1 || time[j] < time[lru\_index]) lru\_index = j;

}

frames[lru\_index] = page\_ref[i]; time[lru\_index] = i; page\_faults++;

}

printf("Frames: "); for (int j = 0; j < MAX; j++) printf("%d ", frames[j]);

printf("\n");

}

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

}

int main() {

int page\_ref[] = {1, 2, 3, 2, 1, 4, 1, 3};

lru(page\_ref, sizeof(page\_ref) / sizeof(page\_ref[0]));

return 0;

}

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

#include <stdio.h>

#define MAX 3

void optimal(int page\_ref[], int n) {

int frames[MAX] = {-1}, page\_faults = 0;

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

int found = 0, farthest = -1, replace\_index = -1;

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

if (frames[j] == page\_ref[i]) { found = 1; break; }

}

if (!found) {

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

int next\_use = -1;

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

if (frames[j] == page\_ref[k]) { next\_use = k; break; }

}

if (next\_use == -1 || next\_use > farthest) { farthest = next\_use; replace\_index = j; }

}

frames[replace\_index] = page\_ref[i]; page\_faults++;

}

printf("Frames: "); for (int j = 0; j < MAX; j++) printf("%d ", frames[j]);

printf("\n");

}

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

}

int main() {

int page\_ref[] = {1, 2, 3, 2, 1, 4, 1, 3};

optimal(page\_ref, sizeof(page\_ref) / sizeof(page\_ref[0]));

return 0;

}

1. 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.

#include <stdio.h>

#define MAX\_RECORDS 5

void simulate\_file\_allocation(int file\_data[], int n) {

printf("File Records: \n");

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

printf("Record %d: %d\n", i + 1, file\_data[i]);

}

printf("\nAccessing File Records Sequentially:\n");

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

printf("Accessing Record %d: %d\n", i + 1, file\_data[i]);

}

}

int main() {

int file\_data[MAX\_RECORDS] = {100, 200, 300, 400, 500};

simulate\_file\_allocation(file\_data, MAX\_RECORDS);

return 0;

}

1. 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.

#include <stdio.h>

#define MAX\_BLOCKS 5

#define MAX\_ENTRIES 5

void simulate\_file\_allocation(int file\_data[], int index\_block[], int n) {

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

index\_block[i] = file\_data[i];

}

printf("Index Block: ");

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

printf("%d ", index\_block[i]);

}

printf("\n");

printf("Accessing file blocks:\n");

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

printf("Block %d: %d\n", i + 1, index\_block[i]);

}

}

int main() {

int file\_data[MAX\_BLOCKS] = {10, 20, 30, 40, 50};

int index\_block[MAX\_ENTRIES] = {-1};

simulate\_file\_allocation(file\_data, index\_block, MAX\_BLOCKS);

return 0;

}

1. 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.

#include <stdio.h>

#define MAX\_BLOCKS 5

struct Block {

int data;

int next;

};

void simulate\_linked\_allocation(struct Block disk[], int file\_blocks[], int n) {

int head = file\_blocks[0];

printf("File Blocks: ");

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

printf("%d ", disk[file\_blocks[i]].data);

}

printf("\n");

printf("\nAccessing File Blocks:\n");

for (int i = head; i != -1; i = disk[i].next) {

printf("Block %d: %d\n", i + 1, disk[i].data);

}

}

int main() {

struct Block disk[MAX\_BLOCKS] = {{100, 1}, {200, 2}, {300, 3}, {400, -1}, {500, -1}};

int file\_blocks[] = {0, 1, 2, 3};

simulate\_linked\_allocation(disk, file\_blocks, 4);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 5

void fcfs(int requests[], int n, int initial\_position) {

int total\_seek\_time = 0;

int current\_position = initial\_position;

printf("Disk Requests Order: ");

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

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

}

printf("\n");

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

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

current\_position = requests[i];

}

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

}

int main() {

int requests[] = {98, 183, 37, 122, 14};

int initial\_position = 50;

fcfs(requests, 5, initial\_position);

return 0;

}

1. Design a C program to simulate SCAN disk scheduling algorithm.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 5

void scan(int requests[], int n, int initial\_position, int direction) {

int total\_seek\_time = 0, current\_position = initial\_position;

int sorted\_requests[MAX\_REQUESTS];

for (int i = 0; i < n; i++) sorted\_requests[i] = requests[i];

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

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

if (sorted\_requests[i] > sorted\_requests[j]) {

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j];

sorted\_requests[j] = temp;

}

}

}

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

if (sorted\_requests[i] >= current\_position) {

total\_seek\_time += abs(current\_position - sorted\_requests[i]);

current\_position = sorted\_requests[i];

}

}

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

}

int main() {

int requests[] = {98, 183, 37, 122, 14};

int initial\_position = 50;

scan(requests, 5, initial\_position, 1);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 5

void cscan(int requests[], int n, int initial\_position, int disk\_size) {

int total\_seek\_time = 0, current\_position = initial\_position;

int sorted\_requests[MAX\_REQUESTS];

for (int i = 0; i < n; i++) sorted\_requests[i] = requests[i];

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

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

if (sorted\_requests[i] > sorted\_requests[j]) {

int temp = sorted\_requests[i];

sorted\_requests[i] = sorted\_requests[j];

sorted\_requests[j] = temp;

}

}

}

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

if (sorted\_requests[i] >= current\_position) {

total\_seek\_time += abs(current\_position - sorted\_requests[i]);

current\_position = sorted\_requests[i];

}

}

total\_seek\_time += abs(current\_position - (disk\_size - 1));

current\_position = 0;

total\_seek\_time += abs(current\_position - sorted\_requests[0]);

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

}

int main() {

int requests[] = {98, 183, 37, 122, 14};

cscan(requests, 5, 50, 200);

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

int main() {

int permissions = 0644;

printf("File permissions: %o\n", permissions);

printf("Owner permissions: %c%c%c\n",

(permissions & 0400) ? 'r' : '-',

(permissions & 0200) ? 'w' : '-',

(permissions & 0100) ? 'x' : '-');

printf("Group permissions: %c%c%c\n",

(permissions & 0040) ? 'r' : '-',

(permissions & 0020) ? 'w' : '-',

(permissions & 0010) ? 'x' : '-');

printf("Others permissions: %c%c%c\n",

(permissions & 0004) ? 'r' : '-',

(permissions & 0002) ? 'w' : '-',

(permissions & 0001) ? 'x' : '-');

return 0;

}