**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp. No.** | **LIST OF EXPERIMENTS** | **Date of Experiment** | **Signature** |
| 1. | Write programs using the following system calls of LINUX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir. |  |  |
| 2. | Write programs using the I/O system calls of LINUX operating system (open, read, write, etc). |  |  |
| 3. | Write C programs to simulate LINUX commands like ls, grep, etc. |  |  |
| 4. | Write a program to create processes and threads. |  |  |
| 5. | Write a program solving the Producer-Consumer problem using semaphores. |  |  |
| 6. | Write a program to implement the solution for dining philosopher’s problem. |  |  |
| 7. | Write a program to develop an application using Inter process communication using shared Memory. |  |  |
| 8. | Write a program to implement process scheduling mechanisms using FCFS & SJF. |  |  |
| 9. | Write a program to implement process scheduling mechanisms using Priority & round-robin scheduling. |  |  |
| 10. | Write a program to implement the banker's algorithm. |  |  |
| 11. | Write a program to implement the banker's algorithm. |  |  |
| 12. | Write a program to implement memory allocation using best fit algorithm. |  |  |

## Experiment-01

#### OBJECTIVE : Write programs using the following system calls of LINUX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

1. **Fork ()**

The fork() system call is used to create a new process by duplicating the calling process. The new process is called the child process.

#### Source Code:

#include <stdio.h> #include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed.\n");

} else if (pid == 0) {

printf("This is the child process.\n");

} else {

printf("This is the parent process.\n");

}

return 0;

}

#### Sample Output:

This is the parent process. This is the child process.

#### Exec()

The exec() family of functions replaces the current process image with a new process image.

#### Source code:

#include <stdio.h> #include <unistd.h>

int main() {

printf("Before exec()\n"); execl("/bin/ls", "ls", NULL);

printf("This line will not be executed if exec is successful.\n");

return 0;

}

#### Sample output:

#### Before exec()

#### file1.txt

#### file2.c

#### directory/

#### Getpid ()

The getpid() system call returns the process ID (PID) of the calling process

#### Source code:

#include <stdio.h> #include <unistd.h>

int main() {

pid\_t pid = getpid(); printf("Process ID: %d\n", pid); return 0;

}

#### Sample output:

Process ID: 12345

#### Exit()

The exit() system call terminates the calling process.

#### Source code:

#include <stdio.h> #include <stdlib.h>

int main() {

printf("Program is about to exit.\n"); exit(0);

// This line will not be executed. printf("This won't print.\n");

}

#### Sample output:

Program is about to exit.

#### Wait()

The wait() system call makes the parent process wait until all of its child processes have terminated.

#### Source code:

#include <stdio.h> #include <sys/wait.h> #include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid == 0) {

printf("Child process\n");

} else {

wait(NULL);

printf("Parent process after child terminates\n");

}

return 0;

}

#### Sample output:

Child process

Parent process after child terminates

#### Close()

The close() system call closes a file descriptor.

#### Source code:

#include <stdio.h> #include <unistd.h> #include <fcntl.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening file.\n"); return 1;

}

printf("File opened successfully.\n"); close(fd);

printf("File closed.\n"); return 0;

}

#### Sample output:

File opened successfully. File closed**.**

#### Stat()

The stat() system call retrieves information about the file specified by the pathname.

#### Source code:

#include <stdio.h> #include <sys/stat.h>

int main() {

struct stat fileStat;

if (stat("test.txt", &fileStat) < 0) { printf("Error getting file status.\n"); return 1;

}

printf("File size: %ld bytes\n", fileStat.st\_size); return 0;

}

#### Sample output:

#### File size: 1024 bytes

#### 

#### 12. opendir() and readdir()

The opendir() system call opens a directory stream, and readdir() reads the next directory entry from the directory stream.

#### Source code:

#include <stdio.h> #include <dirent.h>

int main() {

DIR \*d;

struct dirent \*dir; d = opendir("."); if (d) {

while ((dir = readdir(d)) != NULL) { printf("%s\n", dir->d\_name);

}

closedir(d);

}

return 0;

}

**Sample output:**

.

..

file1.txt

file2.c

directory

## Experiment-02

**OBJECTIVE :** Write programs using the I/O system calls of LINUX operating system (open, read, write, etc).

#### open()

The open() system call is used to open a file and returns a file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

printf("File opened successfully with file descriptor: %d\n", fd);

close(fd); return 0;

}

#### Sample output:

File opened successfully with file descriptor: 3

#### read()

The read() system call reads data from a file descriptor into a buffer.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

char buffer[100];

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead == -1) {

printf("Error reading the file.\n"); close(fd);

return 1;

}

buffer[bytesRead] = '\0'; // Null-terminate the string printf("Data read from file:\n%s\n", buffer);

close(fd); return 0;

}

#### Sample output:

#### Data read from file:

#### Hello, world! This is a test file.

#### write()

The write() system call writes data from a buffer to a file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("output.txt", O\_WRONLY | O\_CREAT, 0644); if (fd == -1) {

printf("Error opening/creating the file.\n"); return 1;

}

const char \*data = "Hello, World!\n"; ssize\_t bytesWritten = write(fd, data, 14);

if (bytesWritten == -1) {

printf("Error writing to the file.\n"); close(fd);

return 1;

}

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

return 0;

}

#### Sample output:

Data written to file successfully.

#### lseek()

The lseek() system call repositions the file offset of the open file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

off\_t offset = lseek(fd, 5, SEEK\_SET); if (offset == -1) {

printf("Error seeking in the file.\n"); close(fd);

return 1;

}

char buffer[100];

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead == -1) {

printf("Error reading the file.\n"); close(fd);

return 1;

}

buffer[bytesRead] = '\0';

printf("Data read from file after seeking:\n%s\n", buffer);

close(fd); return 0;

}

#### Sample output:

Data read from file after seeking:

<Contents starting from the 6th byte of the file "test.txt">

#### close()

The close() system call closes a file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

printf("File opened successfully with file descriptor: %d\n", fd);

close(fd);

printf("File closed successfully.\n");

return 0;

}

#### Sample output:

File opened successfully with file descriptor:

<file\_descriptor>

File closed successfully.

# **Experiment -3**

**OBJECTIVE :** Write C programs to simulate LINUX commands like ls, grep, etc

**LS**

The ls command is used to list files and directories within a directory.

**Source Code:**

#include <stdio.h>

#include <stdlib.h> // For exit()

#include <dirent.h> // For directory operations

int main() {

char dirname[100]; // Increased the size to handle longer directory names

DIR \*p;

struct dirent \*d;

// Prompt the user to enter the directory name printf("Enter directory name: ");

scanf("%s", dirname);

// Attempt to open the directory p = opendir(dirname);

if (p == NULL) {

perror("Cannot find directory");

exit(EXIT\_FAILURE); // Use EXIT\_FAILURE instead of -1 for portability

}

// Read and print the directory contents while ((d = readdir(p)) != NULL) {

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

}

// Close the directory closedir(p);

return 0; // Return 0 to indicate successful execution

}

**Sample Output:**

Enter directory name: myfolder

.

..

file1.txt

file2.c

subdir

**Grep:**

The grep command is used to search for specific patterns within files.

**Source code:**

#include <stdio.h> #include <string.h>

int main() {

char fn[100], pat[100], temp[1000]; FILE \*fp;

// Prompt the user for the file name printf("Enter file name: "); scanf("%s", fn);

// Prompt the user for the pattern to be searched printf("Enter pattern to be searched: "); scanf("%s", pat);

// Open the file in read mode fp = fopen(fn, "r");

// Check if the file was successfully opened if (fp == NULL) {

printf("Could not open file %s\n", fn); return 1;

}

// Read the file line by line and search for the pattern while (fgets(temp, sizeof(temp), fp) != NULL) {

if (strstr(temp, pat)) { printf("%s", temp);

}

}

// Close the file fclose(fp); return 0;

}

**Sample output :**

Enter file name: example.txt

Enter pattern to be searched: pattern

We are searching for a pattern.

This line does not contain the pattern.

Pattern matching is fun.

**Cp:**

The cp command is used to copy files and directories.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

int main(int argc, char \*\*argv) { char buffer[1024];

int files[2]; ssize\_t count;

if (argc < 3) {

fprintf(stderr, "Usage: %s <source file> <destination file>\n", argv[0]);

return EXIT\_FAILURE;

}

files[0] = open(argv[1], O\_RDONLY); if (files[0] == -1) {

perror("Error opening source file"); return EXIT\_FAILURE;

}

files[1] = open(argv[2], O\_WRONLY | O\_CREAT | O\_TRUNC,

S\_IRUSR | S\_IWUSR);

if (files[1] == -1) {

perror("Error opening destination file"); close(files[0]);

return EXIT\_FAILURE;

}

while ((count = read(files[0], buffer, sizeof(buffer))) >

0) {

if (write(files[1], buffer, count) == -1) { perror("Error writing to destination file"); close(files[0]);

close(files[1]); return EXIT\_FAILURE;

}

}

if (count == -1) {

perror("Error reading from source file");

}

close(files[0]);

close(files[1]);

return (count == -1) ? EXIT\_FAILURE : EXIT\_SUCCESS;

}

**Sample output:**

./filecopy source.txt destination.txt

**EXPERIMENT-4**

**OBJECTIVE :** Write a program to create processes and threads.

1.Process

**Source code:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

} else if (pid == 0) {

printf("This is the child process with PID: %d\n", getpid());

} else {

printf("This is the parent process with PID: %d\n", getpid());

printf("The child process ID is: %d\n", pid);

}

return 0;

}

**Sample output:**

This is the parent process with PID: 1234

The child process ID is: 1235

This is the child process with PID: 1235

2.Thread

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void\* threadFunction(void\* arg) {

printf("This is the thread, Thread ID: %lu\n", pthread\_self());

return NULL;

}

int main() {

pthread\_t thread;

int result;

result = pthread\_create(&thread, NULL, threadFunction, NULL);

if (result != 0) {

printf("Thread creation failed!\n");

return 1;

}

pthread\_join(thread, NULL);

printf("Thread has finished execution.\n");

return 0;

}

**Sample Output:**

This is the thread, Thread ID: 140364648662784

Thread has finished execution.

**EXPERIMENT-5**

**OBJECTIVE :** Write a program solving the Producer-Consumer problem using semaphores.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5 // Buffer size

int buffer[BUFFER\_SIZE];

int in = 0; // Points to the next empty slot

int out = 0; // Points to the next full slot

// Semaphores

sem\_t empty; // Counts empty slots

sem\_t full; // Counts full slots

pthread\_mutex\_t mutex; // Mutex for critical section

// Producer function

void \*producer(void \*arg) {

int item;

while (1) {

item = rand() % 100; // Produce an item

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

pthread\_mutex\_lock(&mutex); // Enter critical section

// Add item to buffer

buffer[in] = item;

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

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

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&full); // Signal that buffer has a new item

sleep(1); // Simulate time taken to produce

}

}

// Consumer function

void \*consumer(void \*arg) {

int item;

while (1) {

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

pthread\_mutex\_lock(&mutex); // Enter critical section

// Remove item from buffer

item = buffer[out];

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

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

pthread\_mutex\_unlock(&mutex); // Exit critical section

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

sleep(1); // Simulate time taken to consume

}

}

int main() {

pthread\_t prod\_thread, cons\_thread;

// Initialize semaphores and mutex

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

sem\_init(&full, 0, 0); // Initially, 0 full slots

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

// Create producer and consumer threads

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

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

// Wait for threads to finish

pthread\_join(prod\_thread, NULL);

pthread\_join(cons\_thread, NULL);

// Cleanup

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

**Sample Output:**

Producer produced: 42

Consumer consumed: 42

Producer produced: 7

Producer produced: 56

Consumer consumed: 7

Producer produced: 93

Consumer consumed: 56

Producer produced: 13

Producer produced: 29

Consumer consumed: 93

Producer produced: 85

Consumer consumed: 13

Producer produced: 74

Consumer consumed: 29

Consumer consumed: 85

Producer produced: 65

Producer produced: 38

Consumer consumed: 74

Producer produced: 99

Consumer consumed: 65

Consumer consumed: 38

Consumer consumed: 99

**EXPERIMENT-6**

**OBJECTIVE :** Write a program to implement the solution for dining philosopher’s problem.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

sem\_t forks[NUM\_PHILOSOPHERS]; // Semaphore array for forks

// Philosopher actions

void think(int philosopher) {

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

sleep(1); // Simulate time spent thinking

}

void eat(int philosopher) {

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

sleep(1); // Simulate time spent eating

}

// Philosopher function

void \*philosopher(void \*arg) {

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

while (1) {

think(philosopher); // Philosopher thinks before eating

// Pick up left fork (fork[philosopher])

sem\_wait(&forks[philosopher]);

// Pick up right fork (fork[(philosopher + 1) % NUM\_PHILOSOPHERS])

sem\_wait(&forks[(philosopher + 1) % NUM\_PHILOSOPHERS]);

// Eating section (both forks are acquired)

eat(philosopher);

// Put down right fork

sem\_post(&forks[(philosopher + 1) % NUM\_PHILOSOPHERS]);

// Put down left fork

sem\_post(&forks[philosopher]);

}

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int philosopher\_ids[NUM\_PHILOSOPHERS];

// Initialize semaphores for each fork

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

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

}

// Create threads for each philosopher

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

philosopher\_ids[i] = i;

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

}

// Wait for all philosopher threads to finish

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

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

}

// Destroy semaphores

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

sem\_destroy(&forks[i]);

}

return 0;

}

**Sample Output:**

Philosopher 0 is thinking.

Philosopher 1 is thinking.

Philosopher 2 is thinking.

Philosopher 3 is thinking.

Philosopher 4 is thinking.

Philosopher 0 is eating.

Philosopher 1 is eating.

Philosopher 2 is eating.

Philosopher 3 is eating.

Philosopher 4 is eating.

Philosopher 0 is thinking.

Philosopher 1 is thinking.

...

**EXPERIMENT-7**

**OBJECTIVE :** Write a program to develop an application using Inter process communication using shared Memory.

**Source code:**

**School of Engineering & Technology**

**Department of Computer Science & Technology**



**Operating System Lab**

**(CSP 244)**

**Lab File**

**(2024-2025)**

**for**

**B. Tech. (CSE)**

**3rd Semester**

**Student details:**

Mohd Zaid

B. Tech. CSE [3rd Semester]

202307325

2023 - 27

**Lab Faculty details:**

Mr. Kanderp Narayan Mishra

Professor,

Department of Computer Science & Engineering

School of Engineering & Technology

Sharda University

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp. No.** | **LIST OF EXPERIMENTS** | **Date of Experiment** | **Signature** |
| 1. | Write programs using the following system calls of LINUX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir. |  |  |
| 2. | Write programs using the I/O system calls of LINUX operating system (open, read, write, etc). |  |  |
| 3. | Write C programs to simulate LINUX commands like ls, grep, etc. |  |  |
| 4. | Write a program to create processes and threads. |  |  |
| 5. | Write a program solving the Producer-Consumer problem using semaphores. |  |  |
| 6. | Write a program to implement the solution for dining philosopher’s problem. |  |  |
| 7. | Write a program to develop an application using Inter process communication using shared Memory. |  |  |
| 8. | Write a program to implement process scheduling mechanisms using FCFS & SJF. |  |  |
| 9. | Write a program to implement process scheduling mechanisms using Priority & round-robin scheduling. |  |  |
| 10. | Write a program to implement the banker's algorithm. |  |  |
| 11. | Write a program to implement the banker's algorithm. |  |  |
| 12. | Write a program to implement memory allocation using best fit algorithm. |  |  |

## Experiment-01

#### OBJECTIVE : Write programs using the following system calls of LINUX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

1. **Fork ()**

The fork() system call is used to create a new process by duplicating the calling process. The new process is called the child process.

#### Source Code:

#include <stdio.h> #include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed.\n");

} else if (pid == 0) {

printf("This is the child process.\n");

} else {

printf("This is the parent process.\n");

}

return 0;

}

#### Sample Output:

This is the parent process. This is the child process.

#### Exec()

The exec() family of functions replaces the current process image with a new process image.

#### Source code:

#include <stdio.h> #include <unistd.h>

int main() {

printf("Before exec()\n"); execl("/bin/ls", "ls", NULL);

printf("This line will not be executed if exec is successful.\n");

return 0;

}

#### Sample output:

#### Before exec()

#### file1.txt

#### file2.c

#### directory/

#### Getpid ()

The getpid() system call returns the process ID (PID) of the calling process

#### Source code:

#include <stdio.h> #include <unistd.h>

int main() {

pid\_t pid = getpid(); printf("Process ID: %d\n", pid); return 0;

}

#### Sample output:

Process ID: 12345

#### Exit()

The exit() system call terminates the calling process.

#### Source code:

#include <stdio.h> #include <stdlib.h>

int main() {

printf("Program is about to exit.\n"); exit(0);

// This line will not be executed. printf("This won't print.\n");

}

#### Sample output:

Program is about to exit.

#### Wait()

The wait() system call makes the parent process wait until all of its child processes have terminated.

#### Source code:

#include <stdio.h> #include <sys/wait.h> #include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid == 0) {

printf("Child process\n");

} else {

wait(NULL);

printf("Parent process after child terminates\n");

}

return 0;

}

#### Sample output:

Child process

Parent process after child terminates

#### Close()

The close() system call closes a file descriptor.

#### Source code:

#include <stdio.h> #include <unistd.h> #include <fcntl.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening file.\n"); return 1;

}

printf("File opened successfully.\n"); close(fd);

printf("File closed.\n"); return 0;

}

#### Sample output:

File opened successfully. File closed**.**

#### Stat()

The stat() system call retrieves information about the file specified by the pathname.

#### Source code:

#include <stdio.h> #include <sys/stat.h>

int main() {

struct stat fileStat;

if (stat("test.txt", &fileStat) < 0) { printf("Error getting file status.\n"); return 1;

}

printf("File size: %ld bytes\n", fileStat.st\_size); return 0;

}

#### Sample output:

#### File size: 1024 bytes

#### 

#### 12. opendir() and readdir()

The opendir() system call opens a directory stream, and readdir() reads the next directory entry from the directory stream.

#### Source code:

#include <stdio.h> #include <dirent.h>

int main() {

DIR \*d;

struct dirent \*dir; d = opendir("."); if (d) {

while ((dir = readdir(d)) != NULL) { printf("%s\n", dir->d\_name);

}

closedir(d);

}

return 0;

}

**Sample output:**

.

..

file1.txt

file2.c

directory

## Experiment-02

**OBJECTIVE :** Write programs using the I/O system calls of LINUX operating system (open, read, write, etc).

#### open()

The open() system call is used to open a file and returns a file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

printf("File opened successfully with file descriptor: %d\n", fd);

close(fd); return 0;

}

#### Sample output:

File opened successfully with file descriptor: 3

#### read()

The read() system call reads data from a file descriptor into a buffer.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

char buffer[100];

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead == -1) {

printf("Error reading the file.\n"); close(fd);

return 1;

}

buffer[bytesRead] = '\0'; // Null-terminate the string printf("Data read from file:\n%s\n", buffer);

close(fd); return 0;

}

#### Sample output:

#### Data read from file:

#### Hello, world! This is a test file.

#### write()

The write() system call writes data from a buffer to a file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("output.txt", O\_WRONLY | O\_CREAT, 0644); if (fd == -1) {

printf("Error opening/creating the file.\n"); return 1;

}

const char \*data = "Hello, World!\n"; ssize\_t bytesWritten = write(fd, data, 14);

if (bytesWritten == -1) {

printf("Error writing to the file.\n"); close(fd);

return 1;

}

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

return 0;

}

#### Sample output:

Data written to file successfully.

#### lseek()

The lseek() system call repositions the file offset of the open file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

off\_t offset = lseek(fd, 5, SEEK\_SET); if (offset == -1) {

printf("Error seeking in the file.\n"); close(fd);

return 1;

}

char buffer[100];

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead == -1) {

printf("Error reading the file.\n"); close(fd);

return 1;

}

buffer[bytesRead] = '\0';

printf("Data read from file after seeking:\n%s\n", buffer);

close(fd); return 0;

}

#### Sample output:

Data read from file after seeking:

<Contents starting from the 6th byte of the file "test.txt">

#### close()

The close() system call closes a file descriptor.

#### Source code:

#include <stdio.h> #include <fcntl.h> #include <unistd.h>

int main() {

int fd = open("test.txt", O\_RDONLY); if (fd == -1) {

printf("Error opening the file.\n"); return 1;

}

printf("File opened successfully with file descriptor: %d\n", fd);

close(fd);

printf("File closed successfully.\n");

return 0;

}

#### Sample output:

File opened successfully with file descriptor:

<file\_descriptor>

File closed successfully.

# **Experiment -3**

**OBJECTIVE :** Write C programs to simulate LINUX commands like ls, grep, etc

**LS**

The ls command is used to list files and directories within a directory.

**Source Code:**

#include <stdio.h>

#include <stdlib.h> // For exit()

#include <dirent.h> // For directory operations

int main() {

char dirname[100]; // Increased the size to handle longer directory names

DIR \*p;

struct dirent \*d;

// Prompt the user to enter the directory name printf("Enter directory name: ");

scanf("%s", dirname);

// Attempt to open the directory p = opendir(dirname);

if (p == NULL) {

perror("Cannot find directory");

exit(EXIT\_FAILURE); // Use EXIT\_FAILURE instead of -1 for portability

}

// Read and print the directory contents while ((d = readdir(p)) != NULL) {

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

}

// Close the directory closedir(p);

return 0; // Return 0 to indicate successful execution

}

**Sample Output:**

Enter directory name: myfolder

.

..

file1.txt

file2.c

subdir

**Grep:**

The grep command is used to search for specific patterns within files.

**Source code:**

#include <stdio.h> #include <string.h>

int main() {

char fn[100], pat[100], temp[1000]; FILE \*fp;

// Prompt the user for the file name printf("Enter file name: "); scanf("%s", fn);

// Prompt the user for the pattern to be searched printf("Enter pattern to be searched: "); scanf("%s", pat);

// Open the file in read mode fp = fopen(fn, "r");

// Check if the file was successfully opened if (fp == NULL) {

printf("Could not open file %s\n", fn); return 1;

}

// Read the file line by line and search for the pattern while (fgets(temp, sizeof(temp), fp) != NULL) {

if (strstr(temp, pat)) { printf("%s", temp);

}

}

// Close the file fclose(fp); return 0;

}

**Sample output :**

Enter file name: example.txt

Enter pattern to be searched: pattern

We are searching for a pattern.

This line does not contain the pattern.

Pattern matching is fun.

**Cp:**

The cp command is used to copy files and directories.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

int main(int argc, char \*\*argv) { char buffer[1024];

int files[2]; ssize\_t count;

if (argc < 3) {

fprintf(stderr, "Usage: %s <source file> <destination file>\n", argv[0]);

return EXIT\_FAILURE;

}

files[0] = open(argv[1], O\_RDONLY); if (files[0] == -1) {

perror("Error opening source file"); return EXIT\_FAILURE;

}

files[1] = open(argv[2], O\_WRONLY | O\_CREAT | O\_TRUNC,

S\_IRUSR | S\_IWUSR);

if (files[1] == -1) {

perror("Error opening destination file"); close(files[0]);

return EXIT\_FAILURE;

}

while ((count = read(files[0], buffer, sizeof(buffer))) >

0) {

if (write(files[1], buffer, count) == -1) { perror("Error writing to destination file"); close(files[0]);

close(files[1]); return EXIT\_FAILURE;

}

}

if (count == -1) {

perror("Error reading from source file");

}

close(files[0]);

close(files[1]);

return (count == -1) ? EXIT\_FAILURE : EXIT\_SUCCESS;

}

**Sample output:**

./filecopy source.txt destination.txt

**EXPERIMENT-4**

**OBJECTIVE :** Write a program to create processes and threads.

1.Process

**Source code:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

printf("Fork failed!\n");

return 1;

} else if (pid == 0) {

printf("This is the child process with PID: %d\n", getpid());

} else {

printf("This is the parent process with PID: %d\n", getpid());

printf("The child process ID is: %d\n", pid);

}

return 0;

}

**Sample output:**

This is the parent process with PID: 1234

The child process ID is: 1235

This is the child process with PID: 1235

2.Thread

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void\* threadFunction(void\* arg) {

printf("This is the thread, Thread ID: %lu\n", pthread\_self());

return NULL;

}

int main() {

pthread\_t thread;

int result;

result = pthread\_create(&thread, NULL, threadFunction, NULL);

if (result != 0) {

printf("Thread creation failed!\n");

return 1;

}

pthread\_join(thread, NULL);

printf("Thread has finished execution.\n");

return 0;

}

**Sample Output:**

This is the thread, Thread ID: 140364648662784

Thread has finished execution.

**EXPERIMENT-5**

**OBJECTIVE :** Write a program solving the Producer-Consumer problem using semaphores.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5 // Buffer size

int buffer[BUFFER\_SIZE];

int in = 0; // Points to the next empty slot

int out = 0; // Points to the next full slot

// Semaphores

sem\_t empty; // Counts empty slots

sem\_t full; // Counts full slots

pthread\_mutex\_t mutex; // Mutex for critical section

// Producer function

void \*producer(void \*arg) {

int item;

while (1) {

item = rand() % 100; // Produce an item

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

pthread\_mutex\_lock(&mutex); // Enter critical section

// Add item to buffer

buffer[in] = item;

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

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

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&full); // Signal that buffer has a new item

sleep(1); // Simulate time taken to produce

}

}

// Consumer function

void \*consumer(void \*arg) {

int item;

while (1) {

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

pthread\_mutex\_lock(&mutex); // Enter critical section

// Remove item from buffer

item = buffer[out];

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

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

pthread\_mutex\_unlock(&mutex); // Exit critical section

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

sleep(1); // Simulate time taken to consume

}

}

int main() {

pthread\_t prod\_thread, cons\_thread;

// Initialize semaphores and mutex

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

sem\_init(&full, 0, 0); // Initially, 0 full slots

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

// Create producer and consumer threads

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

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

// Wait for threads to finish

pthread\_join(prod\_thread, NULL);

pthread\_join(cons\_thread, NULL);

// Cleanup

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

**Sample Output:**

Producer produced: 42

Consumer consumed: 42

Producer produced: 7

Producer produced: 56

Consumer consumed: 7

Producer produced: 93

Consumer consumed: 56

Producer produced: 13

Producer produced: 29

Consumer consumed: 93

Producer produced: 85

Consumer consumed: 13

Producer produced: 74

Consumer consumed: 29

Consumer consumed: 85

Producer produced: 65

Producer produced: 38

Consumer consumed: 74

Producer produced: 99

Consumer consumed: 65

Consumer consumed: 38

Consumer consumed: 99

**EXPERIMENT-6**

**OBJECTIVE :** Write a program to implement the solution for dining philosopher’s problem.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

sem\_t forks[NUM\_PHILOSOPHERS]; // Semaphore array for forks

// Philosopher actions

void think(int philosopher) {

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

sleep(1); // Simulate time spent thinking

}

void eat(int philosopher) {

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

sleep(1); // Simulate time spent eating

}

// Philosopher function

void \*philosopher(void \*arg) {

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

while (1) {

think(philosopher); // Philosopher thinks before eating

// Pick up left fork (fork[philosopher])

sem\_wait(&forks[philosopher]);

// Pick up right fork (fork[(philosopher + 1) % NUM\_PHILOSOPHERS])

sem\_wait(&forks[(philosopher + 1) % NUM\_PHILOSOPHERS]);

// Eating section (both forks are acquired)

eat(philosopher);

// Put down right fork

sem\_post(&forks[(philosopher + 1) % NUM\_PHILOSOPHERS]);

// Put down left fork

sem\_post(&forks[philosopher]);

}

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int philosopher\_ids[NUM\_PHILOSOPHERS];

// Initialize semaphores for each fork

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

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

}

// Create threads for each philosopher

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

philosopher\_ids[i] = i;

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

}

// Wait for all philosopher threads to finish

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

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

}

// Destroy semaphores

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

sem\_destroy(&forks[i]);

}

return 0;

}

**Sample Output:**

Philosopher 0 is thinking.

Philosopher 1 is thinking.

Philosopher 2 is thinking.

Philosopher 3 is thinking.

Philosopher 4 is thinking.

Philosopher 0 is eating.

Philosopher 1 is eating.

Philosopher 2 is eating.

Philosopher 3 is eating.

Philosopher 4 is eating.

Philosopher 0 is thinking.

Philosopher 1 is thinking.

...

**EXPERIMENT-7**

**OBJECTIVE :** Write a program to develop an application using Inter process communication using shared Memory.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <unistd.h>

#include <string.h>

#include <sys/wait.h>

#define SHM\_SIZE 1024 // Shared memory size

struct SharedMemory {

char message[100];

int ready; // Flag to indicate message is ready

};

int main() {

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

int shmid = shmget(key, SHM\_SIZE, 0666 | IPC\_CREAT); // Create shared memory segment

if (shmid == -1) {

perror("Shared memory");

exit(1);

}

struct SharedMemory \*shared\_data = (struct SharedMemory \*) shmat(shmid, (void \*)0, 0); // Attach shared memory

if (shared\_data == (void \*) -1) {

perror("Shared memory attach");

exit(1);

}

shared\_data->ready = 0; // Initialize the ready flag

pid\_t pid = fork(); // Fork a new process

if (pid == -1) {

perror("fork");

exit(1);

}

else if (pid == 0) { // Child Process (Reader)

while (shared\_data->ready == 0) {

// Wait for the writer to set the message

sleep(1);

}

printf("Reader Process: Message from shared memory: %s\n", shared\_data->message);

shared\_data->ready = 0; // Reset the ready flag

// Detach and remove shared memory

shmdt(shared\_data);

}

else { // Parent Process (Writer)

char message[] = "Hello from the writer process!";

strncpy(shared\_data->message, message, sizeof(shared\_data->message) - 1);

shared\_data->message[sizeof(shared\_data->message) - 1] = '\0'; // Null-terminate message

shared\_data->ready = 1; // Set the ready flag

printf("Writer Process: Message written to shared memory: %s\n", shared\_data->message);

// Wait for child to finish reading

wait(NULL);

// Detach and remove shared memory

shmdt(shared\_data);

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

}

return 0;

}

**Sample Output:**

Writer Process: Message written to shared memory: Hello from the writer process!

Reader Process: Message from shared memory: Hello from the writer process!

**EXPERIMENT-8**

**OBJECTIVE :** Write a program to implement process scheduling mechanisms using FCFS & SJF.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

struct Process {

int id;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

};

// Function to calculate Waiting and Turnaround times for FCFS

void calculate\_fcfs(struct Process processes[], int n) {

int current\_time = 0;

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

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

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

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

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

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

}

}

// Function to calculate Waiting and Turnaround times for SJF

void calculate\_sjf(struct Process processes[], int n) {

int completed = 0, current\_time = 0, min\_index;

int \*completed\_processes = calloc(n, sizeof(int));

while (completed < n) {

min\_index = -1;

int min\_burst\_time = \_\_INT\_MAX\_\_;

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

if (!completed\_processes[i] && processes[i].arrival\_time <= current\_time) {

if (processes[i].burst\_time < min\_burst\_time) {

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

min\_index = i;

}

}

}

if (min\_index == -1) {

current\_time++;

} else {

completed\_processes[min\_index] = 1;

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

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

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

completed++;

}

}

free(completed\_processes);

}

// Function to display process scheduling details

void display\_processes(struct Process processes[], int n) {

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

printf("P\_ID\tArrival\tBurst\tWaiting\tTurnaround\n");

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

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

processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

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

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

}

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

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

}

int main() {

int n, choice;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process processes[n];

printf("Enter Arrival Time and Burst Time for each process:\n");

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

processes[i].id = i + 1;

printf("Process %d:\n", processes[i].id);

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

}

printf("Choose Scheduling Algorithm:\n1. FCFS\n2. SJF\n");

scanf("%d", &choice);

if (choice == 1) {

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

calculate\_fcfs(processes, n);

} else if (choice == 2) {

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

calculate\_sjf(processes, n);

} else {

printf("Invalid choice!\n");

return 1;

}

display\_processes(processes, n);

return 0;

}

**Sample Output:**

--- FCFS Scheduling ---

P\_ID Arrival Burst Waiting Turnaround

1 0 4 0 4

2 1 3 3 6

3 2 1 6 7

Average Waiting Time: 3.00

Average Turnaround Time: 5.67

--- SJF Scheduling ---

P\_ID Arrival Burst Waiting Turnaround

1 0 4 2 6

2 1 3 0 3

3 2 1 1 2

Average Waiting Time: 1.00

Average Turnaround Time: 3.67

**EXPERIMENT-9**

**OBJECTIVE :** Write a program to implement process scheduling mechanisms using Priority & round-robin scheduling.

**Source code:**

#include <stdio.h>

#include <stdlib.h>

struct Process {

int id;

int arrival\_time;

int burst\_time;

int priority;

int waiting\_time;

int turnaround\_time;

int remaining\_time; // For Round Robin

};

// Function to calculate Priority scheduling (non-preemptive)

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

int completed = 0, current\_time = 0, min\_index;

while (completed < n) {

min\_index = -1;

int min\_priority = \_\_INT\_MAX\_\_;

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

if (processes[i].arrival\_time <= current\_time && processes[i].waiting\_time == -1) {

if (processes[i].priority < min\_priority) {

min\_priority = processes[i].priority;

min\_index = i;

}

}

}

if (min\_index == -1) {

current\_time++;

} else {

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

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

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

completed++;

}

}

}

// Function to calculate Round Robin scheduling

void calculate\_round\_robin(struct Process processes[], int n, int quantum) {

int completed = 0, current\_time = 0, i = 0;

while (completed < n) {

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

if (processes[i].remaining\_time <= quantum) {

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

processes[i].remaining\_time = 0;

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

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

completed++;

} else {

processes[i].remaining\_time -= quantum;

current\_time += quantum;

}

}

i = (i + 1) % n;

}

}

// Function to display process scheduling details

void display\_processes(struct Process processes[], int n) {

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

printf("P\_ID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n");

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

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

processes[i].burst\_time, processes[i].priority, processes[i].waiting\_time, processes[i].turnaround\_time);

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

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

}

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

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

}

int main() {

int n, choice, quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process processes[n];

printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");

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

processes[i].id = i + 1;

printf("Process %d:\n", processes[i].id);

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

printf("Priority: ");

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

processes[i].waiting\_time = -1; // Indicates process has not been scheduled yet

processes[i].remaining\_time = processes[i].burst\_time; // Initialize remaining time for RR

}

printf("Choose Scheduling Algorithm:\n1. Priority\n2. Round Robin\n");

scanf("%d", &choice);

if (choice == 1) {

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

calculate\_priority(processes, n);

} else if (choice == 2) {

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

scanf("%d", &quantum);

printf("\n--- Round Robin Scheduling ---\n");

calculate\_round\_robin(processes, n, quantum);

} else {

printf("Invalid choice!\n");

return 1;

}

display\_processes(processes, n);

return 0;

}

**Sample Output:**

--- Priority Scheduling ---

P\_ID Arrival Burst Priority Waiting Turnaround

1 0 10 2 0 10

2 1 4 1 9 13

3 2 5 3 12 17

Average Waiting Time: 7.00

Average Turnaround Time: 13.33

Enter time quantum for Round Robin: 3

--- Round Robin Scheduling ---

P\_ID Arrival Burst Priority Waiting Turnaround

1 0 10 2 9 19

2 1 4 1 4 8

3 2 5 3 10 15

Average Waiting Time: 7.67

Average Turnaround Time: 14.00

**EXPERIMENT-10**

**OBJECTIVE :** Write a program to implement the banker's algorithm.

**Source code:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int n, m;

int available[MAX\_RESOURCES];

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

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

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

void calculate\_need() {

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

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

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

}

}

}

bool is\_safe() {

bool finish[MAX\_PROCESSES] = {0};

int safe\_sequence[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

// Copy available resources to work

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

work[i] = available[i];

}

int count = 0;

while (count < n) {

bool found = false;

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

if (!finish[i]) {

bool can\_allocate = true;

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

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

can\_allocate = false;

break;

}

}

if (can\_allocate) {

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

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

}

safe\_sequence[count++] = i;

finish[i] = true;

found = true;

}

}

}

if (!found) {

printf("System is not in a safe state!\n");

return false;

}

}

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

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

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

}

printf("\n");

return true;

}

void request\_resources(int process\_id, int request[]) {

// Check if request is within need

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

if (request[i] > need[process\_id][i]) {

printf("Error: Process has exceeded its maximum claim.\n");

return;

}

}

// Check if request is within available resources

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

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

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

return;

}

}

// Temporarily allocate resources and check for safety

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

available[i] -= request[i];

allocation[process\_id][i] += request[i];

need[process\_id][i] -= request[i];

}

if (is\_safe()) {

printf("Resources allocated successfully.\n");

} else {

printf("Request cannot be granted as it leads to an unsafe state.\n");

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

available[i] += request[i];

allocation[process\_id][i] -= request[i];

need[process\_id][i] += request[i];

}

}

}

int main() {

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resources: ");

scanf("%d", &m);

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

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

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

}

printf("Enter maximum resources matrix:\n");

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

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

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

}

}

printf("Enter allocation matrix:\n");

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

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

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

}

}

calculate\_need();

if (is\_safe()) {

int process\_id;

int request[MAX\_RESOURCES];

printf("\nEnter process number for resource request: ");

scanf("%d", &process\_id);

printf("Enter resource request for process %d:\n", process\_id);

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

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

}

request\_resources(process\_id, request);

}

return 0;

}

**Sample Output:**

Enter number of processes: 3

Enter number of resources: 3

Enter available resources:

3 3 2

Enter maximum resources matrix:

7 5 3

3 2 2

9 0 2

Enter allocation matrix:

0 1 0

2 0 0

3 0 2

System is in a safe state.

Safe sequence: 1 0 2

Enter process number for resource request: 1

Enter resource request for process 1:

1 0 2

Resources allocated successfully.

System is in a safe state.

Safe sequence: 1 0 2

**EXPERIMENT-11**

**OBJECTIVE :** Write a program to implement memory allocation using best fit algorithm.

**Source code:**

#include <stdio.h>

#define MAX\_BLOCKS 10

#define MAX\_PROCESSES 10

// Function to implement Best Fit allocation

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

int allocation[MAX\_PROCESSES];

// Initialize all allocations to -1 (indicating no allocation)

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

allocation[i] = -1;

}

// Loop through each process and allocate 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 suitable block is found, allocate it to the process

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

// Output the allocation results

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[MAX\_BLOCKS], processSize[MAX\_PROCESSES];

int m, n;

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

scanf("%d", &m);

printf("Enter the size of each block:\n");

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

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

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

}

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

scanf("%d", &n);

printf("Enter the memory requirement of each process:\n");

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

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

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

}

bestFit(blockSize, m, processSize, n);

return 0;

}

**Sample Output:**

Enter the number of memory blocks: 5

Enter the size of each block:

Block 1: 100

Block 2: 500

Block 3: 200

Block 4: 300

Block 5: 600

Enter the number of processes: 4

Enter the memory requirement of each process:

Process 1: 212

Process 2: 417

Process 3: 112

Process 4: 426

Process No. Process Size Block No.

1 212 4

2 417 2

3 112 1

4 426 5