SRINIVAS INSTITUTE OF TECHNOLOGY

(Accredited by NAAC)
Valachil, Mangaluru-574143

Department of Artificial intelligence and Machine Learning

LAB MANUAL

OPERATING SYSTEM LABORATORY
BCS303

USN	
NAME	

Vision:

To be a centre of excellence in Artificial Intelligence and Machine Learning with quality education and research, responsive to the needs of industry and society.

Mission:

To achieve academic excellence through innovative teaching-learning practice.
To inculcate the spirit of innovation, creativity and research.
To enhance employability through skill development and industry-institute interaction.
To develop professionals with ethical values and social responsibilities.

Program Educational Objectives (PEO)

Graduates will be

PEO1: Competent professionals in the field of Artificial Intelligence and Machine Learning to pursue careers in diverse

PEO2: Proficient in designing innovative solutions to real life problems that are technically sound, economically viable and socially acceptable.

PEO3: Capable of working in teams and adapting to new technologies as per the society needs with ethical values.

Programme Specific Outcomes (PSO)

PSO1: Hardware and hardware-software co-design: Ability to identify, design, simulate, analyse and develop AI and ML models, using modern engineering tools and programming languages.

PSO2: Domain specific skills: Ability to work in the field of AI /ML Engineer, Data Scientist, Business Analyst, Product Analyst, Research Scientist and Robotics Professional.

Programme Outcome (PO)

Communication

Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

Sl.NO	Experiments
1	Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)
2	Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCF b) SJF c) Round Robin d) Priority.
3	Develop a C program to simulate producer-consumer problem using semaphores.
4	Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.
5	Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.
6	Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.
7	Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU
8	Simulate following File Organization Techniques a) Single level directory b) Two level directory
9	Develop a C program to simulate the Linked file allocation strategies.
10	Develop a C program to simulate SCAN disk scheduling algorithm.

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

CO 1. Explain the structure and functionality of operating system

CO 2. Apply appropriate CPU scheduling algorithms for the given problem.

CO 3. Analyse the various techniques for process synchronization and deadlock handling.

CO 4. Apply the various techniques for memory management

CO 5. Explain file and secondary storage management strategies.

CO 6. Describe the need for information protection mechanisms

Case study on unix based system

Unix is a powerful, multi-user, and multitasking operating system that originated in the late 1960s and has since influenced the development of many other operating systems. Unix-based systems share certain design philosophies, file structures, and command-line interfaces inspired by Unix.

Here are some key characteristics and aspects of Unix-based systems:

1. Multiuser and Multitasking:Unix is designed to support multiple users concurrently. Each user has their own account and workspace.It is a multitasking system, allowing multiple processes to run simultaneously.
2. Hierarchical File System:The file system in Unix is organized hierarchically, starting from the root directory ("/").Files and directories are organized in a tree-like structure.
3. Shell and Command-Line Interface (CLI):Unix-based systems typically include a command-line interface (CLI) where users interact with the system using commands.The shell is a command interpreter that allows users to run programs, manage files, and perform various system tasks.
4. Portability:Unix was designed to be highly portable, allowing it to run on various hardware architectures.This portability has led to the development of Unix-based systems on different platforms, including workstations, servers, and embedded systems.
5. Security:Unix emphasizes security through user account management, file permissions, and access control lists.The principle of least privilege is applied, granting users the minimum level of access required to perform their tasks.
6. Networking:Unix-based systems have robust networking capabilities, supporting protocols such as TCP/IP.This makes Unix well-suited for networked environments, and it has been widely used in server applications.
7. Programming Environment:Unix provides a rich programming

environment with a variety of programming tools and languages.It supports a wide range of development tools and compilers.

8. Open Standards:Unix systems adhere to open standards, enabling interoperability and compatibility between different Unix-based platforms. The POSIX (Portable Operating System Interface) standard helps ensure compatibility across Unix-like systems.
9. Variety of Implementations:Various Unix-based operating systems exist, including commercial Unix variants like AIX, HP-UX, and Solaris, as well as open-source implementations like Linux and BSD (Berkeley Software Distribution).
10. Philosophy of Simplicity:Unix follows the philosophy of simplicity and modularity. Each program should do one thing well, and programs should work together seamlessly.

Examples of Unix-based systems include:

1. Linux: A popular open-source Unix-like operating system kernel. Many distributions (distros) build on the Linux kernel, such as Ubuntu, Fedora, and CentOS.

BSD (Berkeley Software Distribution): Variants of Unix developed at the University of California, Berkeley, including FreeBSD, OpenBSD, and

1. NetBSD.
2. macOS: The operating system for Apple's Macintosh computers, built on a Unix-based foundation called Darwin.

Algorithm:

1. **Include Necessary Headers:**

   Include the required headers for standard input/output (stdio.h), standard library functions (stdlib.h), process-related functions (unistd.h), and process control functions (sys/wait.h).

2. **Define the main Function:**

   Declare the main function.

3. **Fork a Child Process:**

   Use the fork() system call to create a child process.

   Check the return value of fork() to determine whether the process is the parent or the child.

4. **Child Process Block (pid == 0):**

   Inside the child process block, print a message indicating that it is the child process.

   Use the exec() system call to replace the current process image with a new program. You can use functions like execlp to execute a specific command.

5. **Parent Process Block (pid > 0):**

   Inside the parent process block, print a message indicating that it is the parent process.

   Use the wait() system call to wait for the child process to finish.

6. **Print Messages for Process Completion:**

   Print messages indicating that the child process is done (inside the child block) and that the parent process is done (inside the parent block).

7. **Compile and Run:**

   Compile the program using a C compiler

Run the compiled program to observe the creation and termination of processe

1.Develop c program to implement the process system calls(fork(),exec(),wait,create process and terminate process)

Program:

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

int main() {
    pid_t pid;

    // Fork a child process
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process (PID %d) is executing.\n", getpid());

        // Execute a different program in the child process using exec
        execlp("ls", "ls", "-l", NULL);

        // If execlp fails
        perror("execlp");
        exit(EXIT_FAILURE);
    } else {
```

```
        // Parent process
        printf("Parent process (PID %d) is waiting for the child to finish.\n",
getpid());

        // Wait for the child process to complete
        wait(NULL);

        printf("Parent process is done.\n");
    }

    return 0;
}
```

Output:

Parent process (PID 23558) is waiting for the child to finish.

Child process (PID 23559) is executing.

execlp: No such file or directory

Parent process is done.

Algorithm:

1. **Include Necessary Headers:**

   Include the required headers for standard input/output (stdio.h), standard library functions (stdlib.h), and any other necessary headers.

2. **Define the Process Structure:**

   Define a structure to represent a process with attributes like process ID, burst time, and priority.

3. **Initialize Variables:**

   Initialize variables and data structures for the number of processes, an array to store processes, and any other necessary variables.

4. **Input Process Data:**

   Input the number of processes and their details such as burst time and priority.

5. **FCFS Algorithm:**

   Sort the processes based on arrival time (FCFS does not use priority).

   Calculate turnaround time and waiting time for each process.

6. **SJF Algorithm:**

   Sort the processes based on burst time (shortest job first).

   Calculate turnaround time and waiting time for each process.

7. **Round Robin Algorithm:**

   Implement the Round Robin scheduling algorithm using a specified time quantum.

   Calculate turnaround time and waiting time for each process.

8. **Priority Algorithm:**

   Sort the processes based on priority.

   Calculate turnaround time and waiting time for each process.

9. **Display Results:**

   Print the results for each scheduling algorithm, displaying process ID, burst time, priority turnaround time, and waiting time.

2.Stimulate the following CPU scheduling algorithm to find out turnaround time and waiting time 1)FCFS 2)SJF 3)Round Robin 4) Priority

Program:

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

struct Process {
    int id;
    int burst_time;
    int priority;
};

void calculateTurnaroundTime(struct Process processes[], int n, int turnaround_time[]) {
    int completion_time = 0;
    for (int i = 0; i < n; i++) {
        completion_time += processes[i].burst_time;
        turnaround_time[i] = completion_time;
    }
}

void calculateWaitingTime(struct Process processes[], int n, int turnaround_time[], int waiting_time[]) {
    for (int i = 0; i < n; i++) {
        waiting_time[i] = turnaround_time[i] - processes[i].burst_time;
    }
}
```

```
void calculateAverageTimes(struct Process processes[], int n) {
    int turnaround_time[n], waiting_time[n];

    // FCFS
    calculateTurnaroundTime(processes, n, turnaround_time);
    calculateWaitingTime(processes, n, turnaround_time, waiting_time);

    // Display results for FCFS
    printf("\nFCFS Scheduling:\n");
    printf("Process\tBurst Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].burst_time, turnaround_time[i], waiting_time[i]);
    }

    // SJF
    // Sort processes based on burst time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                // Swap
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
```

```
// Recalculate times for SJF
    calculateTurnaroundTime(processes, n, turnaround_time);
    calculateWaitingTime(processes, n, turnaround_time, waiting_time);

    // Display results for SJF
    printf("\nSJF Scheduling:\n");
    printf("Process\tBurst Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].burst_time, turnaround_time[i], waiting_time[i]);
    }

    // Round Robin
    int time_quantum = 2;
    int remaining_time[n];
    for (int i = 0; i < n; i++) {
        remaining_time[i] = processes[i].burst_time;
    }

    int t = 0; // Current time
    int done = 0; // Check if all processes are done
    int turn_around_time_rr[n], waiting_time_rr[n];

    // Simulate Round Robin
    while (!done) {
        done = 1; // Assume all processes are done
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0) {
```

```
    done = 0; // There is still a process remaining
            if (remaining_time[i] > time_quantum) {
                t += time_quantum;
                remaining_time[i] -= time_quantum;
            } else {
                t += remaining_time[i];
                remaining_time[i] = 0;
                turn_around_time_rr[i] = t;
                waiting_time_rr[i] = turn_around_time_rr[i] -
processes[i].burst_time;
            }
        }
    }
    }

    // Display results for Round Robin
    printf("\nRound Robin Scheduling (Time Quantum = %d):\n",
time_quantum);
    printf("Process\tBurst Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id,
processes[i].burst_time, turn_around_time_rr[i], waiting_time_rr[i]);
    }

    // Priority
    // Sort processes based on priority
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
```

```
  if (processes[j].priority > processes[j + 1].priority) {
            // Swap
            struct Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
        }
    }
  }

  // Recalculate times for Priority
  calculateTurnaroundTime(processes, n, turnaround_time);
  calculateWaitingTime(processes, n, turnaround_time, waiting_time);

  // Display results for Priority
  printf("\nPriority Scheduling:\n");
  printf("Process\tBurst Time\tPriority\tTurnaround Time\tWaiting Time\n");
  for (int i = 0; i < n; i++) {
     printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, processes[i].priority, turnaround_time[i], waiting_time[i]);
  }
}

int main() {
  int n;
  printf("Enter the number of processes: ");
  scanf("%d", &n);
```

```
    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    calculateAverageTimes(processes, n);

    return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter burst time for process 1: 2
Enter priority for process 1: 3
Enter burst time for process 2: 3
Enter priority for process 2: 4
Enter burst time for process 3: 1
Enter priority for process 3: 1
Enter burst time for process 4: 2
Enter priority for process 4: 5
Enter burst time for process 5: 4
Enter priority for process 5: 2
```

FCFS Scheduling:

Process	Burst Time	Turnaround Time	Waiting Time
1	2	2	0
2	3	5	2
3	1	6	5
4	2	8	6
5	4	12	8

SJF Scheduling:

Process	Burst Time	Turnaround Time	Waiting Time
3	1	1	0
1	2	3	1
4	2	5	3
2	3	8	5
5	4	12	8

Round Robin Scheduling (Time Quantum = 2):

Process	Burst Time	Turnaround Time	Waiting Time
3	1	1	0
1	2	3	1
4	2	5	3
2	3	10	7
5	4	12	8

Priority Scheduling:

Process	Burst Time	Priority	Turnaround Time	Waiting Time
3	1	1	1	0
5	4	2	5	1
1	2	3	7	5
2	3	4	10	7
4	2	5	12	10

Algorithm:

1. **Include Necessary Headers:**

   Include the required headers for standard input/output (stdio.h), standard library functions (stdlib.h), synchronization (semaphore.h), and threading (pthread.h).

2. **Define Constants:**

   Define constants for the buffer size and the number of producers and consumers.

   Define Shared Variables and Semaphores:

   Define a buffer array to represent the shared buffer.

   Define semaphores for mutex (to control access to the buffer) and for tracking the number of empty and full slots in the buffer.

3. **Initialize Semaphores and Variables:**

   Initialize the semaphores using sem_init.

   Initialize any necessary variables (e.g., indices for the buffer).

4. **Define Producer and Consumer Functions:**

   Create functions for the producer and consumer processes.

   In the producer function, generate an item, wait for an empty slot in the buffer (sem_wait), acquire the mutex, add the item to the buffer, release the mutex, and signal that a slot is filled (sem_post).

   In the consumer function, wait for a filled slot in the buffer (sem_wait), acquire the mutex, remove an item from the buffer, release the mutex, and signal that a slot is empty (sem_post).

5. **Create Producer and Consumer Threads:**

   Create threads for the producers and consumers using pthread_create.

6. **Join Threads:**

   Wait for the producer and consumer threads to finish using pthread_join.

7. **Destroy Semaphores:**

   Destroy the semaphores using sem_destroy.

3. Develop c program to Stimulate Producer-Consumer problem using semaphore.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
sem_t mutex, empty, full;

void *producer(void *arg) {
   int item;
   for (int I = 0; I < 10; i++) {
      item = rand() % 100;  // Produce a random item
      sem_wait(&empty);
      sem_wait(&mutex);

      // Critical section: Produce item and add to buffer
      printf("Producer produced item %d\n", item);
      for (int j = 0; j < BUFFER_SIZE; j++) {
         if (buffer[j] == -1) {
            buffer[j] = item;
            break;
         }
```

```
    }

        sem_post(&mutex);
        sem_post(&full);
        sleep(1);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    int item;
    for (int I = 0; I < 10; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        // Critical section: Consume item from buffer
        for (int j = 0; j < BUFFER_SIZE; j++) {
            if (buffer[j] != -1) {
                item = buffer[j];
                buffer[j] = -1;
                break;
            }
        }
        printf("Consumer consumed item %d\n", item);

        sem_post(&mutex);
        sem_post(&empty);
```

```
    sleep(2);
    }
    pthread_exit(NULL);
}

int main() {
    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Initialize buffer
    for (int I = 0; I < BUFFER_SIZE; i++) {
        buffer[i] = -1;
    }

    // Create threads
    pthread_t producer_thread, consumer_thread;
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Join threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Destroy semaphores
    sem_destroy(&mutex);
```

```
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

Output:

```
Producer produced item 41
Consumer consumed item 41
Producer produced item 67
Consumer consumed item 67
Producer produced item 34
Producer produced item 0
Consumer consumed item 34
Producer produced item 69
Producer produced item 24
Consumer consumed item 69
Producer produced item 78
Producer produced item 58
Consumer consumed item 78
Producer produced item 62
Producer produced item 64
Consumer consumed item 62
Consumer consumed item 0
Consumer consumed item 24
Consumer consumed item 58
Consumer consumed item 64
```

Algorithm:

1. **Include Necessary Headers:**

   Include the required headers for standard input/output (stdio.h), standard library functions (stdlib.h), and file-related functions (fcntl.h, sys/stat.h, unistd.h).

2. **Define Constants:**

   Define constants for the named pipe file path.

   Create a Named Pipe (FIFO):

   Use mkfifo to create a named pipe (FIFO) with a specified path.

3. **Define Writer Process:**

   Inside the writer process block, open the named pipe for writing using open.

   Write data to the named pipe using write.

   Close the named pipe using close.

4. **Define Reader Process:**

   Inside the reader process block, open the named pipe for reading using open.

   Read data from the named pipe using read.

   Close the named pipe using close.

5. **Compile and Run:**

   Compile the program using a C compiler

   Run the compiled program to observe interprocess communication.

4. Develop c program to which demonstrate the interprocess communication between reader process and writer process.Use mkfifo,open ,read,write,close API in your program

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define FIFO_NAME "myfifo"

void writerProcess() {
   int fd;

   // Create the FIFO (named pipe)
   mkfifo(FIFO_NAME, 0666);

   // Open the FIFO for writing
   fd = open(FIFO_NAME, O_WRONLY);

   // Write data to the FIFO
   char message[] = "Hello, Reader!";
   write(fd, message, sizeof(message));

   // Close the FIFO
   close(fd);
```

```
// Remove the FIFO
    unlink(FIFO_NAME);
}

void readerProcess() {
    int fd;
    char buffer[100];

    // Open the FIFO for reading
    fd = open(FIFO_NAME, O_RDONLY);

    // Read data from the FIFO
    read(fd, buffer, sizeof(buffer));

    // Display the received message
    printf("Reader received: %s\n", buffer);

    // Close the FIFO
    close(fd);
}

int main() {
    // Create two processes - writer and reader
    pid_t pid = fork();

    if (pid < 0) {
```

```
 fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid > 0) {
        // Parent process (writer)
        writerProcess();
    } else {
        // Child process (reader)
        readerProcess();
    }

    return 0;
}
```

Output:

Reader received: Hello, Reader!

Algorithm:

1. **Include Necessary Headers:**

   Include the required headers for standard input/output (stdio.h), standard library functions (stdlib.h), and any other necessary headers.

2. **Define Constants:**

   Define constants for the maximum number of processes and resources.

   Define Global Variables:

   Define global variables for the available, maximum, allocation, and need matrices.

   Define variables for the number of processes and resources.

3. **Initialize Matrices:**

   Input the number of processes and resources.

   Initialize the available, maximum, and allocation matrices.

4. **Implement Banker's Algorithm:**

   Implement the Banker's algorithm logic to check for safety.

   Implement functions to request and release resources.

5. **Display Results:**

   Print the results to display the status of the system, including the safe sequence if applicable.

5. Develop c program to stimulate the bankers algorithm for deadloack avoidance

```
#include <stdio.h>
#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int available[MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];

int nProcesses, nResources;

// Function to check if the system is in a safe state
int isSafe() {
    int work[MAX_RESOURCES];
    int finish[MAX_PROCESSES];

    // Initialize work and finish arrays
    for (int i = 0; i < nResources; i++) {
        work[i] = available[i];
    }

    for (int i = 0; i < nProcesses; i++) {
        finish[i] = 0;
    }
```

```
int count = 0;

// Iterate until all processes are finished or a deadlock is detected
while (count < nProcesses) {
    int found = 0;

    // Find a process that can finish
    for (int i = 0; i < nProcesses; i++) {
        if (finish[i] == 0) {
            int j;
            for (j = 0; j < nResources; j++) {
                if (need[i][j] > work[j]) {
                    break;
                }
            }

            if (j == nResources) {
                // Process i can finish
                for (int k = 0; k < nResources; k++) {
                    work[k] += allocation[i][k];
                }

                finish[i] = 1;
                found = 1;
                count++;
            }
        }
```

```
 }

      // If no process can finish, break the loop (deadlock detected)
      if (!found) {
         break;
      }
   }

   // If all processes finished, the system is in a safe state
   return count == nProcesses;
}

// Function to simulate resource request and allocation
void simulateResourceRequest(int process, int request[]) {
   // Check if the request is within the maximum limits
   for (int i = 0; i < nResources; i++) {
      if (request[i] > need[process][i]) {
         printf("Error: Request exceeds maximum claim.\n");
         return;
      }
   }

   // Check if the request can be satisfied
   for (int i = 0; i < nResources; i++) {
      if (request[i] > available[i]) {
         printf("Process %d must wait. Insufficient resources.\n", process);
```

```
    return;
        }
    }

    // Simulate the allocation
    for (int i = 0; i < nResources; i++) {
        available[i] -= request[i];
        allocation[process][i] += request[i];
        need[process][i] -= request[i];
    }

    // Check if the system is still in a safe state after the allocation
    if (isSafe()) {
        printf("Request by Process %d granted. New state is safe.\n",
process);
    } else {
        // Rollback the allocation if it makes the system unsafe
        printf("Request by Process %d denied. New state would be unsafe.
Rolling back.\n", process);
        for (int i = 0; i < nResources; i++) {
            available[i] += request[i];
            allocation[process][i] -= request[i];
            need[process][i] += request[i];
        }
    }
}

int main() {
```

```
printf("Enter the number of processes: ");
scanf("%d", &nProcesses);

printf("Enter the number of resources: ");
scanf("%d", &nResources);

// Input maximum resources for each process
printf("Enter the maximum resources for each process:\n");
for (int i = 0; i < nProcesses; i++) {
    printf("Process %d: ", i);
    for (int j = 0; j < nResources; j++) {
        scanf("%d", &maximum[i][j]);
        need[i][j] = maximum[i][j];
    }
}

// Input available resources
printf("Enter the available resources: ");
for (int i = 0; i < nResources; i++) {
    scanf("%d", &available[i]);
}

// Input current allocation for each process
printf("Enter the current allocation for each process:\n");
for (int i = 0; i < nProcesses; i++) {
    printf("Process %d: ", i);
```

```
for (int j = 0; j < nResources; j++) {
            scanf("%d", &allocation[i][j]);
            need[i][j] -= allocation[i][j];
        }
    }

    // Input the process number making a resource request
    int requestingProcess;
    printf("Enter the process number making a resource request: ");
    scanf("%d", &requestingProcess);

    // Input the resource request
    int request[nResources];
    printf("Enter the resource request for Process %d: ",
requestingProcess);
    for (int i = 0; i < nResources; i++) {
        scanf("%d", &request[i]);
    }

    // Simulate the resource request
    simulateResourceRequest(requestingProcess, request);

    return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter the number of resources: 2
Enter the maximum resources for each process:
```

Process 0: 1

Process 1: 2

Process 2: 1

Process 3: 1

Process 4: 3

Enter the available resources: 2

Enter the current allocation for each process:

Process 0: 1

Process 1: 2

Process 2: 0

Process 3: 1

Process 4: 1

Enter the process number making a resource request: 2

Enter the resource request for Process 2: 2

Error: Request exceeds maximum claim.

Worst Fit algorithm:

1. Initialize an empty memory list.

2. For each process request:

a. Search the memory list for the largest available block that can accommodate the process.

b. If found, allocate the process to that block.

c. If not found, request additional memory from the operating system.

3. Repeat step 2 for all process requests.

4. When a process is deallocated, add the freed memory block to the memory list.

Best Fit algorithm:

1. Initialize an empty memory list.

2. For each process request:

a. Search the memory list for the smallest available block that can accommodate the process.

b. If found, allocate the process to that block.

c. If not found, request additional memory from the operating system.

3. Repeat step 2 for all process requests.

4. When a process is deallocated, add the freed memory block to the memory list.

First Fit Algorithm:

1. Initialize an empty memory list.

2. For each process request:

a. Search the memory list for the first available block that can accommodate the process.

b. If found, allocate the process to that block.

c. If not found, request additional memory from the operating system.

3. Repeat step 2 for all process requests.

6. Develop c program to stimulate the following continuous allocation technique .

1)worst fit, 2)best fit 3) first fit

1)worst fit

```
#include <stdio.h>

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

   for (int i = 0; i < n; i++) {
      allocation[i] = -1; // Initialize allocation array to -1 (unallocated)
   }

   for (int i = 0; i < n; i++) {
      int worstFitIndex = -1;
      for (int j = 0; j < m; j++) {
         if (blockSize[j] >= processSize[i]) {
            if (worstFitIndex == -1 || blockSize[j] > blockSize[worstFitIndex])
{
               worstFitIndex = j;
            }
         }
      }

      if (worstFitIndex != -1) {
         allocation[i] = worstFitIndex; // Allocate the block to the process
         blockSize[worstFitIndex] -= processSize[i];
```

4. When a process is deallocated, add the freed memory block to the memory list.

```
 }
   }

   printf("Worst Fit Allocation:\n");
   for (int i = 0; i < n; i++) {
      printf("Process %d -> Block %d\n", i + 1, allocation[i] + 1);
   }
}

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]);

   worstFit(blockSize, m, processSize, n);

   return 0;
}
```

Output:

Worst Fit Allocation:

Process 1 -> Block 5

Process 2 -> Block 2

Process 3 -> Block 5

Process 4 -> Block 0

2)best fit

```
#include <stdio.h>

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

    for (int i = 0; i < n; i++) {
        allocation[i] = -1; // Initialize allocation array to -1 (unallocated)
    }

    for (int i = 0; i < n; i++) {
        int bestFitIndex = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestFitIndex == -1 || blockSize[j] < blockSize[bestFitIndex]) {
                    bestFitIndex = j;
                }
            }
        }

        if (bestFitIndex != -1) {
            allocation[i] = bestFitIndex; // Allocate the block to the process
            blockSize[bestFitIndex] -= processSize[i];
        }
    }

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

```
 for (int i = 0; i < n; i++) {
        printf("Process %d -> Block %d\n", i + 1, allocation[i] + 1);
    }
}

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]);

    bestFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

Best Fit Allocation:

Process 1 -> Block 4

Process 2 -> Block 2

Process 3 -> Block 3

Process 4 -> Block 5

3) first fit

```
#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; // Initialize allocation array to -1 (unallocated)
   }

   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         if (blockSize[j] >= processSize[i]) {
            allocation[i] = j; // Allocate the block to the process
            blockSize[j] -= processSize[i];
            break;
         }
      }
   }

   printf("First Fit Allocation:\n");
   for (int i = 0; i < n; i++) {
      printf("Process %d -> Block %d\n", i + 1, allocation[i] + 1);
   }
}

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;
}
```

Output:

First Fit Allocation:

Process 1 -> Block 2

Process 2 -> Block 5

Process 3 -> Block 2

Process 4 -> Block 0

1) FIFO Algorithm:

1. Initialize an empty queue to represent the set of pages currently in memory.

2. For each page reference:

a. Check if the page is in the set of pages currently in memory.

i. If yes, continue to the next page reference.

ii. If no, check if the memory is full.

1. If not full, add the page to the set of pages in memory and enqueue the page.

2. If full, dequeue the oldest page from the queue, remove it from the set of pages in memory, and enqueue the new page.

3. Repeat step 2 for all page references.

2) LRU Algorithm:

1. Initialize an empty set to represent the set of pages currently in memory.

2. Initialize a counter for tracking the usage history of pages.

3. For each page reference:

a. Check if the page is in the set of pages currently in memory.

i. If yes, update the usage counter for the page to the current time.

ii. If no, check if the memory is full.

1. If not full, add the page to the set of pages in memory, set the usage counter for the page to the current time.

2. If full, find the page with the smallest usage counter (least recently used), remove it from the set of pages in memory, and add the new page with the current time.

4. Repeat step 3 for all page references.

7. Develop c program to stimulate the page replacement algorithm

1) FIFO 2) LRU

1) FIFO

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

#define MAX_FRAMES 3

void fifo(int pages[], int n) {
   int frames[MAX_FRAMES];
   int frameCount = 0;
   int pageFaults = 0;

   printf("FIFO Page Replacement Algorithm:\n");

   for (int i = 0; i < n; i++) {
      int currentPage = pages[i];
      int pageHit = 0;

      for (int j = 0; j < frameCount; j++) {
         if (frames[j] == currentPage) {
            pageHit = 1;
            break;
         }
      }

      if (!pageHit) {
```

```
    if (frameCount < MAX_FRAMES) {
            frames[frameCount++] = currentPage;
        } else {
            // Replace the oldest page (first-in)
            frames[0] = currentPage;
        }
        pageFaults++;
    }

    printf("Page %d: [", currentPage);
    for (int j = 0; j < frameCount; j++) {
        printf("%d ", frames[j]);
    }
    printf("] Page Faults: %d\n", pageFaults);
  }

  printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
  int pages[] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5};
  int n = sizeof(pages) / sizeof(pages[0]);

  fifo(pages, n);

  return 0;
}
```

Output:

FIFO Page Replacement Algorithm:

Page 2: [2] Page Faults: 1

Page 3: [2 3] Page Faults: 2

Page 2: [2 3] Page Faults: 2

Page 1: [2 3 1] Page Faults: 3

Page 5: [5 3 1] Page Faults: 4

Page 2: [2 3 1] Page Faults: 5

Page 4: [4 3 1] Page Faults: 6

Page 5: [5 3 1] Page Faults: 7

Page 3: [5 3 1] Page Faults: 7

Page 2: [2 3 1] Page Faults: 8

Page 5: [5 3 1] Page Faults: 9

Total Page Faults: 9

2) LRU

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

#define MAX_FRAMES 3

void lru(int pages[], int n) {
    int frames[MAX_FRAMES];
    int recent[MAX_FRAMES];
    int frameCount = 0;
    int pageFaults = 0;
```

```
printf("LRU Page Replacement Algorithm:\n");

  for (int i = 0; i < n; i++) {
    int currentPage = pages[i];
    int pageHit = 0;

    for (int j = 0; j < frameCount; j++) {
      if (frames[j] == currentPage) {
        pageHit = 1;

        // Update the most recently used page
        for (int k = j; k > 0; k--) {
          recent[k] = recent[k - 1];
        }
        recent[0] = currentPage;

        break;
      }
    }

    if (!pageHit) {
      if (frameCount < MAX_FRAMES) {
        frames[frameCount] = currentPage;
        recent[frameCount++] = currentPage;
      } else {
        // Replace the least recently used page
        frames[0] = currentPage;
```

```
  // Update the most recently used pages
            for (int k = 1; k < MAX_FRAMES; k++) {
               recent[k] = recent[k - 1];
            }
            recent[0] = currentPage;
         }
         pageFaults++;
      }

      printf("Page %d: [", currentPage);
      for (int j = 0; j < frameCount; j++) {
         printf("%d ", frames[j]);
      }
      printf("] Page Faults: %d\n", pageFaults);
   }

   printf("Total Page Faults: %d\n", pageFaults);
}

int main() {
   int pages[] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5};
   int n = sizeof(pages) / sizeof(pages[0]);

   lru(pages, n);

   return 0;
}
```

Output:

LRU Page Replacement Algorithm:

Page 2: [2] Page Faults: 1

Page 3: [2 3] Page Faults: 2

Page 2: [2 3] Page Faults: 2

Page 1: [2 3 1] Page Faults: 3

Page 5: [5 3 1] Page Faults: 4

Page 2: [2 3 1] Page Faults: 5

Page 4: [4 3 1] Page Faults: 6

Page 5: [5 3 1] Page Faults: 7

Page 3: [5 3 1] Page Faults: 7

Page 2: [2 3 1] Page Faults: 8

Page 5: [5 3 1] Page Faults: 9

Total Page Faults: 9

single File Directory Algorithm

Initialization:

Create a structure for a file, which may contain attributes like name, size, and other relevant information.

Create a structure for a single-level directory, which may contain attributes like the directory name and an array to store files.

Algorithm:

Initialize a single-level directory.

Create a file in the single-level directory

Use the createFile function to create files in the single-level directory.

Two-Level Directory Organization Algorithm

Initialization:

Extend the single-level file directory algorithm to include a structure for a two-level directory.

The two-level directory structure should have an array of single-level directories.

Algorithm:

Initialize a two-level directory.

Create a file in a specific subdirectory of the two-level directory.

Use the createFile function to create files in the two-level directory.

8. Using c program to stimulate the following file organization technique

1)single file directory 2) two level directory

Single-Level Directory Structure:

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

struct File {
    char filename[50];
    // Additional file attributes can be added here
};

struct Directory {
    struct File files[100];
    int fileCount;
};

void displayFiles(struct Directory* directory) {
    printf("Files in the directory:\n");
    for (int i = 0; i < directory->fileCount; i++) {
        printf("%s\n", directory->files[i].filename);
    }
}

int main() {
    struct Directory rootDirectory;
    rootDirectory.fileCount = 0;
```

```
    char choice;

    do {
        struct File newFile;
        printf("Enter the filename: ");
        scanf("%s", newFile.filename);

        rootDirectory.files[rootDirectory.fileCount++] = newFile;

        printf("Do you want to add another file? (y/n): ");
        scanf(" %c", &choice);
    } while (choice == 'y' || choice == 'Y');

    displayFiles(&rootDirectory);

    return 0;
}
```

Output:

```
Enter the filename: os
Do you want to add another file? (y/n): y
Enter the filename: bi
Do you want to add another file? (y/n): n
Files in the directory:
os
bi
```

Two-Level Directory Structure:

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

struct File {
   char filename[50];
   // Additional file attributes can be added here
};

struct SubDirectory {
   char dirname[50];
   struct File files[100];
   int fileCount;
};

struct Directory {
   struct SubDirectory subdirectories[100];
   int subdirectoryCount;
};

void displayFiles(struct SubDirectory* subdirectory) {
   printf("Files in the subdirectory %s:\n", subdirectory->dirname);
   for (int i = 0; i < subdirectory->fileCount; i++) {
      printf("%s\n", subdirectory->files[i].filename);
   }
}
```

```
int main() {
   struct Directory rootDirectory;
   rootDirectory.subdirectoryCount = 0;
   char choice;
   do {
      struct SubDirectory newSubdirectory;
      printf("Enter the subdirectory name: ");
      scanf("%s", newSubdirectory.dirname);
      newSubdirectory.fileCount = 0;

      char fileChoice;
      do {
         struct File newFile;
         printf("Enter the filename: ");
         scanf("%s", newFile.filename);

         newSubdirectory.files[newSubdirectory.fileCount++] = newFile;

         printf("Do you want to add another file to this subdirectory? (y/n):
");
         scanf(" %c", &fileChoice);
      } while (fileChoice == 'y' || fileChoice == 'Y');

      rootDirectory.subdirectories[rootDirectory.subdirectoryCount++] =
newSubdirectory;
 printf("Do you want to add another subdirectory? (y/n): ");
      scanf(" %c", &choice);
```

```
 } while (choice == 'y' || choice == 'Y');
    for (int i = 0; i < rootDirectory.subdirectoryCount; i++) {
        displayFiles(&rootDirectory.subdirectories[i]);
    }
    return 0;
}
```

Output:

```
Enter the subdirectory name: text
Enter the filename: os
Do you want to add another file to this subdirectory? (y/n): y
Enter the filename: bi
Do you want to add another file to this subdirectory? (y/n): n
Do you want to add another subdirectory? (y/n): y
Enter the subdirectory name: qn
Enter the filename: IA2seta
Do you want to add another file to this subdirectory? (y/n): y
Enter the filename: IA2setb
Do you want to add another file to this subdirectory? (y/n): n
Do you want to add another subdirectory? (y/n): n
Files in the subdirectory text:
os
bi
Files in the subdirectory qn:


IA2seta
IA2setb
```

Initialization:

Define structures for a file block and a file, which contains pointers to file blocks.

Create functions to allocate and deallocate file blocks.

Create functions to create, read, update, and delete files using the linked file allocation strategy.

Algorithm:

1. **Initialize the File Block Structure:**
2. **Initialize the File Structure:**
3. **Function to Allocate a File Block:**
4. **Function to Deallocate a File Block:**
5. **Function to Create a File:**
6. **Function to Add a Block to a File:**
7. **Function to Delete a File:**

9. Develop a c program to stimulate the linked file allocation stratergy.

```
#include <stdio.h>
#include <stdlib.h>
struct Block {
    int blockNumber;
    struct Block* next;
};

// Function to allocate a new block in the linked list
struct Block* allocateBlock(int blockNumber) {
    struct Block* newBlock = (struct Block*)malloc(sizeof(struct Block));
    newBlock->blockNumber = blockNumber;
    newBlock->next = NULL;
    return newBlock;
}

// Function to display the linked file allocation
void displayAllocation(struct Block* head) {
    struct Block* current = head;
    while (current != NULL) {
        printf("%d -> ", current->blockNumber);
        current = current->next;
    }
    printf("NULL\n");
}
```

```
// Function to simulate linked file allocation strategy
void linkedFileAllocation() {
    struct Block* head = NULL;
    struct Block* current = NULL;

    int blockNumber;
    char choice;

    do {
        printf("Enter block number: ");
        scanf("%d", &blockNumber);

        if (head == NULL) {
            head = allocateBlock(blockNumber);
            current = head;
        } else {
            current->next = allocateBlock(blockNumber);
            current = current->next;
        }

        printf("Do you want to add another block? (y/n): ");
        scanf(" %c", &choice);

    } while (choice == 'y' || choice == 'Y');

    printf("Linked File Allocation:\n");
```

```
    displayAllocation(head);
}

int main() {
    linkedFileAllocation();
    return 0;
}
```

Output:

```
Enter block number: 5
Do you want to add another block? (y/n): y
Enter block number: 6
Do you want to add another block? (y/n): y
Enter block number: 3
Do you want to add another block? (y/n): n
Linked File Allocation:
5 -> 6 -> 3 -> NULL
```

Algorithm:

SCAN Disk Scheduling Algorithm

1.Initialization:

Read the initial position of the disk arm.

Determine the direction of movement (towards the end or towards the beginning).

Sort the Request Queue:

Sort the pending requests in the order of their request positions.

2.Scan the Disk:

Move the disk arm in the determined direction.

Service requests in the current direction until the end of the disk is reached.

Change the direction when reaching the end.

3.Continue Scanning:

Continue scanning and servicing requests in the opposite direction.

Change direction when reaching the other end of the disk.

4.Repeat Until All Requests are Serviced:

Repeat steps 3-4 until all requests are serviced.

5.Completion:

Calculate and report the total head movement.

10.Develop a c program to stimulate scan disk scheduling algorithm

Program:

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

// Function to sort the array in ascending order
void sort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

// Function to simulate SCAN disk scheduling algorithm
void scan(int arr[], int n, int head, char direction) {
    int totalSeekOperations = 0;
    int distance, currentTrack;
    int visited[n];

    sort(arr, n);

    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }

    printf("Sequence of tracks visited in SCAN algorithm:\n");

    if (direction == 'l') {
        // Move left
        for (currentTrack = head; currentTrack >= 0; currentTrack--) {
            if (!visited[currentTrack]) {
                printf("%d ", currentTrack);
```

```
            visited[currentTrack] = 1;
                totalSeekOperations += abs(head - currentTrack);
                head = currentTrack;
            }
        }

        // Move right
        for (currentTrack = head + 1; currentTrack < n;
currentTrack++) {
            if (!visited[currentTrack]) {
                printf("%d ", currentTrack);
                visited[currentTrack] = 1;
                totalSeekOperations += abs(head - currentTrack);
                head = currentTrack;
            }
        }
    } else if (direction == 'r') {
        // Move right
        for (currentTrack = head; currentTrack < n; currentTrack++) {
            if (!visited[currentTrack]) {
                printf("%d ", currentTrack);
                visited[currentTrack] = 1;
                totalSeekOperations += abs(head - currentTrack);
                head = currentTrack;
            }
        }

        // Move left
        for (currentTrack = head - 1; currentTrack >= 0; currentTrack--
) {
            if (!visited[currentTrack]) {
                printf("%d ", currentTrack);
                visited[currentTrack] = 1;
                totalSeekOperations += abs(head - currentTrack);
                head = currentTrack;
            }
        }
    }
```

```
    printf("\nTotal seek operations: %d\n", totalSeekOperations);
}

int main() {
    int n, head, direction;
    printf("Enter the number of tracks: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the track positions:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter the initial head position: ");
    scanf("%d", &head);

    printf("Enter the direction (l for left, r for right): ");
    scanf(" %c", &direction);

    scan(arr, n, head, direction);

    return 0;
}
```

Output:

```
Enter the number of tracks: 5
Enter the track positions:
3
6
2
4
1
Enter the initial head position: 1
Enter the direction (l for left, r for right): l
Sequence of tracks visited in SCAN algorithm:
1 0 2 3 4
Total seek operations: 5
```