DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING

Department of Computer Science & Technology

LAB MANUAL

Integrated Course

OPERATING SYSTEM

Course Code: 22CT3505

Dayananda Sagar University

School of Engineering

Devarakaggalahalli, Harohalli Kanakapura Road, Dt, Ramanagara, Karnataka 562112

Department of Computer Science & Technology

ODD Sem (Aug – Dec 2024)
Academic Year: 2024 – 2025

Operating System Laboratory

Course Code: 22CT3505

Dayananda Sagar University

School of Engineering

DEPARTMENT OF COMPUTER SCIENCE & TECHNOLOGY

Vision

To be recognized globally as a premier department in the field of computer science and Technology that fosters excellence in academics, research, innovation and entrepreneurship with focus on sustainable ecosystem.

Mission

M1: To impart high-quality education rooted in sustainable learning practices, instilling ethical values, fostering leadership qualities, nurturing research skills, and promoting lifelong learning.

M2: To promote innovation and entrepreneurship through cutting-edge research and startups in technology, offering our students a dual-career path that encompasses both technology and entrepreneurship.

M3: To equip our graduates with multi-disciplinary practices and mindset needed to contribute to society.

Programme Educational Outcomes (PEO's)

After few years of graduation, the Computer Science & Technology graduates will:

PEO1: Possess intellectual and critical thinking with modern technology skills, enabling them to address dynamic challenges and develop innovative solutions for real-world problems.

PEO2: Acquire strong ethical reasoning skills and a deep understanding of their social, civic, and professional responsibilities, preparing them for successful careers in various sectors, including industry, start-ups, government, education, and research institutions.

PEO3: Persist in their quest for knowledge, advance their careers and apply interpersonal, leadership skills to become successful leaders and entrepreneurs.

PROGRAMME SPECIFIC OUTCOMES (PSOs):

PSO1: Conceptualize domain-specific areas such as gaming, data analytics, quantum computing, IoT applying tools and disruptive technologies to analyze and build efficient solutions to problems.

PSO2: Apply modern tools and techniques to manage product life cycle with entrepreneurship skills leveraging best engineering practices.

Program Outcome (PO's)

PO1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10. 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.

PO11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

INSTRUCTIONS for LABORATORY EXERCISES

1. The programs with comments are listed for your reference. Write the programs in observation book.
2. Create your own subdirectory in the computer. Edit (type) the programs with program number and place them in your subdirectory.
3. Execute the programs as per the steps discussed earlier and note the results in your observation book.

How to write, compile and run the program on Linux terminal

Steps / Procedure:

Open Terminal, type gedit filename.c

gcc filename.c - To Compile the file

./a.out - To execute the file

DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING

Marks Distribution

SL No.	Particulars	Max Marks (60M)	Marks Obtained
1.	MSE 1	15	
2.	MSE 2	15	
3.	Conduct the regular practical lab and recording of the experiment With Via-Voce	10	
4.	Lab Internal Practical Test	10	
5.	Case Study Report	05	
6.	OBE (Open Book Test)	05	

Sign of the student _____

Sign of the Staff Incharge _____

Sign of the Chairperson _____

DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING

LAB PROGRAMS

Date: __/__/20__

1.Write a C program to create a new process that exec a new program using system calls fork(), execlp() & wait()

SYSTEM CALLS USED:

1. fork ( )

This call is used to create new processes. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero,whereas the value of process id for the parent is an integer value greater than zero.
Syntax: fork ( )

2. execlp( )

Used after the fork () system call by one of the two processes to replace the process memory space with a new program. It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution. The child process overlays its address space with the UNIX command /bin/ls using the execlp system call.
Syntax: execlp( )

3. wait ( )

The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children have terminated.
Syntax: wait (NULL)

4. exit( )

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).
Syntax: exit(0)

ALGORITHM:

Step 1: Declare the variable pid.
Step 2: Get the pid value using system call fork ().
Step 3: If pid value is less than zero then print as “Fork failed”.
Step 4: Else if pid value is equal to zero include the new process in the system’s file using execlp system call.
Step 5: Else if pid is greater than zero then it is the parent process and it waits till the child completes using the system call wait ()
Step 6: Then print “Child complete”.

Program Code:

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

int main(int argc, char *argv[])
{
   int pid;
   pid = fork();

   if (pid < 0)
   {
      printf("Fork failed\n");
      exit(1);
   }
   else if (pid == 0)
   {
      printf("\nNow in Child Process and its output is:\n");
      execlp("ls", "ls", NULL);
      exit(0);
   }
   else
   {
      printf("\nChild Process created successfully\n");
      printf("\nParent Process ID: %d\n", getpid());
      wait(NULL);
      printf("\nReturning to Parent process, now ready to exit\n");
      exit(0);
   }
}
```

Sample Output 1:

Child Process created successfully

Parent Process ID: 3587

Now in Child Process and its output is:
a.c Desktop Downloads OS Public velu.c
a.out Documents Music Pictures Templates Videos

Returning to Parent process, now ready to exit

Sample Output 2:

Child Process created successfully

Parent Process ID: 3717

Now in Child Process and its output is:
a.out Documents Music Pictures Templates Videos
Desktop Downloads OS Public velu.c

Returning to Parent process, now ready to exit

Date: __/__/20__

2. Write a C program to display PID and PPID using system calls getpid () &getppid()

SYSTEM CALLS USED:

1. getpid( )

Each process is identified by its id value. This function is used to get the id value of a particular process.

2. getppid( )

This is used to get particular process parent's id value.

3. perror( )

Indicate the process error.

ALGORITHM:

Step 1: Declare the variables pid , parent pid , child id and grand child id.
Step 2: Get the child id value using system call fork().
Step 3: If child id value is less than zero then print as "error at fork() child".
Step 4: If child id !=0 then using getpid() system call get the process id.
Step 5: Print "I am parent" and print the process id.
Step 6: Get the grandchild id value using system call fork().
Step 7: If the grandchild id value is less than zero then print as "error at fork() grandchild".
Step 8: If the grandchild id !=0 then using getpid system call get the process id.
Step 9: Assign the value of pid to my pid.
Step 10: Print "I am child" and print the value of my pid.
Step 11: Get my parent pid value using system call getppid().
Step 12: Print "My parent's process id" and its value.
Step 13: Else print "I am the grandchild".
Step 14: Get the grandchild's process id using getpid() and print it as "my process id".
Step 15: Get the grandchild's parent process id using getppid() and print it as "my parent's process id

Program Code:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main()
{
        int pid;
        pid=fork();
        if(pid==0)
        {
                printf("Child Process...");
                printf("\n\nChild PID: %d",getpid());
                printf("\nParent PID: %d",getppid());
                printf("\nFinished with child\n");
        }
        else
        {
                wait(NULL);
                printf("\nParent process");
                printf("\nParent PID: %d",getpid());
                printf("\nChild PID: %d\n",pid);
        }
}
```

Sample Output 1:

```
Child Process...
Child PID: 3209
Parent PID: 3208

Finished with child
Parent process
Parent PID: 3208
Child PID: 3209
```

Sample Output 2:

```
Child Process...

Child PID: 4046
Parent PID: 4045
Finished with child

Parent process
Parent PID: 4045
Child PID: 4046
```

Date: __/__/20__

3. Write a C program using I/O system calls open(), read() & write() to copy contents of one file to another file

SYSTEM CALLS USED:

- int open (const char* Path, int flags [, int mode]);
- size_t read (int fd, void* buf, size_t cnt);
- size_t write (int fd, void* buf, size_t cnt);

Algorithm:

1. Open the Source File for Reading
   Use the open() system call to open the source file in read-only mode.
2. Open/Create the Destination File for Writing
   Use open() to open the destination file. If it does not exist, create it with write permissions.
3. Read from the Source File
   Use read() to read data from the source file into a buffer.
4. Write to the Destination File
   Use write() to write data from the buffer to the destination file.
5. Repeat Reading and Writing
   Continue reading from the source file and writing to the destination file until all data has been copied.
6. Close Both Files
   Use close() to close both the source and destination files.

Program Code:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<fcntl.h>
void main()
{
        char buff;
        int fd,fd1;
        fd=open("one.txt",O_RDONLY);
        fd1=open("two.txt",O_WRONLY|O_CREAT);
        while(read(fd,&buff,1))
        write(fd1,&buff,1);
```

```
        printf("The file is successfully copied!!!\n");
        close(fd);
        close(fd1);
}
```

Sample Output:

First create file named as one.txt

cat > one.txt
Hello world
This is the contents of one.txt file.

Now execute the program

gcc pgm3.
./a.out
The file is sucessfully copied.!!!

Give the Read, Write & Execute permission to that file.

Chmod 777 two.txt

Now open the second file and view the contents

cat two.txt
Hello world
This is the contents of one.txt file.

Date: __/__/20__

4. Write a C program to implement multithreaded program using pthreads

Program Code:

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

// Global variable to change in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp) {
    // Store the value argument passed to this thread
    int *myid = (int *)vargp;

    // Change static and global variables
    static int s = 0; // Static variable to observe its changes
    ++s;
    ++g;

    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, s, g);

    // Exit the thread
    pthread_exit(NULL);
}

int main() {
    int i;
    pthread_t tid[3]; // Array to hold thread IDs
    int thread_ids[3]; // Array to hold thread IDs passed to the thread function

    // Create three threads
    for (i = 0; i < 3; i++) {
        thread_ids[i] = i + 1; // Assign thread ID
        pthread_create(&tid[i], NULL, myThreadFun, (void *)&thread_ids[i]);
    }
```

```
    // Wait for all threads to finish
    for (i = 0; i < 3; i++) {
        pthread_join(tid[i], NULL);
    }

    return 0;
}
```

Sample Output 1:

```
Thread ID: 2, Static: 1, Global: 1
Thread ID: 1, Static: 2, Global: 2
Thread ID: 3, Static: 3, Global: 3
```

Sample Output 2:

```
Thread ID: 1, Static: 1, Global: 1
Thread ID: 3, Static: 3, Global: 3
Thread ID: 2, Static: 2, Global: 2
```

Date: __/__/20__

5.Write C program to simulate Round Robin CPU scheduling algorithm

PROBLEM DESCRIPTION:

CPU scheduler will decide which process should be given the CPU for its execution. For this it use different algorithm to choose among the process. One among that algorithm is Round robin algorithm. In this algorithm we are assigning some time slice. The process is allocated according to the time slice, if the process service time is less than the time slice then process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.If the CPU burst of the currently running process is longer than time quantum; the timer will go off and will cause an interrupt to the operating system. A context switch will be executed and the process will be put at the tail of the ready queue.

Algorithm:

- **Start**
- **Input**:
- Number of processes (NOP).
- For each process, input:
  - Arrival time (at[])
  - Burst time (bt[])
- Input time quantum (quant).
- **Initialize variables**:
- Copy bt[] into temp[] for storing remaining burst times.
- Initialize sum = 0, wt = 0, tat = 0, y = NOP, count = 0.
- **While** there are unfinished processes (y != 0):
- **For each process i**:
  1. If temp[i] <= quant and temp[i] > 0:
     - Increment sum by temp[i].
     - Set temp[i] = 0.
     - Set count = 1.
  2. Else if temp[i] > quant:
     - Deduct quant from temp[i].
     - Increment sum by quant.
  3. If temp[i] == 0 and count == 1:
     - Decrement y.
     - Calculate TAT = sum - at[i].
     - Calculate WT = TAT - bt[i].
     - Add to wt and tat.

- Print process number, burst time, turnaround time, and waiting time.
- Set count = 0.

- If i == NOP - 1, set i = 0.
- Else if at[i + 1] <= sum, increment i.
- Else set i = 0.

➢ **Calculate** average waiting time (avg_wt = wt / NOP) and average turnaround time (avg_tat = tat / NOP).
➢ **Output** the average waiting time and turnaround time.
➢ **End**

Program Code:

```
#include<stdio.h>
int main()
{
   // Initialize variables
   int i, NOP, sum = 0, count = 0, y, quant, wt = 0, tat = 0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;

   // Input number of processes
   printf("Total number of processes in the system: ");
   scanf("%d", &NOP);
   y = NOP; // Assign the number of processes to variable y

   // Input arrival time and burst time for each process
   for(i = 0; i < NOP; i++)
        {
         printf("\nEnter the Arrival and Burst time of Process[%d]\n", i + 1);
        printf("Arrival time: "); // Accept arrival time
        scanf("%d", &at[i]);
        printf("Burst time: ");   // Accept burst time
        scanf("%d", &bt[i]);
        temp[i] = bt[i];          // Store the burst time in temp array
        }

   // Input the time quantum
   printf("Enter the Time Quantum for the process: ");
   scanf("%d", &quant);

   // Display the table header
```

```
    printf("\nProcess No \tBurst Time \tTurnaround Time \tWaiting Time\n");

    // Round Robin Scheduling algorithm
    for(sum = 0, i = 0; y != 0;)
{
        if(temp[i] <= quant && temp[i] > 0)
          {
                sum += temp[i];  // Increment total time by burst time of the process
                temp[i] = 0;     // Process finished
                count = 1;
          }
        else if(temp[i] > 0)
          {
                temp[i] -= quant; // Decrement burst time by quantum
                sum += quant;     // Increment total time by quantum
          }

        if(temp[i] == 0 && count == 1)
          {
                 y--;  // Decrement number of remaining processes
                 // Calculate waiting time and turnaround time
printf("Process[%d] \t\t %d \t\t %d \t\t\t %d\n", i + 1, bt[i], sum - at[i], sum - at[i] -
bt[i]);
           wt += sum - at[i] - bt[i];  // Total waiting time
           tat += sum - at[i];         // Total turnaround time
           count = 0;
          }

        if(i == NOP - 1)
          {
                i = 0; // Reset process index to 0 (Round Robin cycle)
          }
        else if(at[i + 1] <= sum)
          {
                i++; // Move to next process if it has arrived
          }
        else
          {
                i = 0; // Otherwise, reset to the first process
          }
    }
    // Calculate and display average waiting time and turnaround time
```

```
    avg_wt = (float)wt / NOP;
    avg_tat = (float)tat / NOP;
    printf("\nAverage Turnaround Time: %.2f", avg_tat);
    printf("\nAverage Waiting Time: %.2f\n", avg_wt);

    return 0;
}
```

Sample Output 1:

```
Total number of processes in the system: 3

Enter the Arrival and Burst time of Process[1]
Arrival time: 0
Burst time: 6

Enter the Arrival and Burst time of Process[2]
Arrival time: 1
Burst time: 8

Enter the Arrival and Burst time of Process[3]
Arrival time: 2
Burst time: 7

Enter the Time Quantum for the process: 4


Process No  Burst Time  Turnaround Time  Waiting Time
Process[1]   6           14               8
Process[2]   8           20               12
Process[3]   7           18               11

Average Turnaround Time: 17.33
Average Waiting Time: 10.33
```

Sample Output 2:

Total number of processes in the system: 4

Enter the Arrival and Burst time of Process[1]
Arrival time: 1
Burst time: 6

Enter the Arrival and Burst time of Process[2]
Arrival time: 2
Burst time: 8

Enter the Arrival and Burst time of Process[3]
Arrival time: 5
Burst time: 3

Enter the Arrival and Burst time of Process[4]
Arrival time: 8
Burst time: 5
Enter the Time Quantum for the process: 4

Process No	Burst Time	Turnaround Time	Waiting Time
Process[3]	3	6	3
Process[1]	6	16	10
Process[2]	8	19	11
Process[4]	5	14	9

Average Turnaround Time: 13.75
Average Waiting Time: 8.25

Date: __/__/20__

6. Write a C program to simulate producer-consumer problem using semaphores

PROGRAM DESCRIPTION:

Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores. We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

Algorithm:

- **Initialize**:
- Set mutex = 1, full = 0, empty = 3, and x = 0.
- **Main Loop**:
- **Repeat** until the user selects "Exit":
    1. **Input choice**:
        - If the user selects **Producer**:
            - If mutex == 1 and empty != 0, call producer().
            - Otherwise, print "Buffer is full!!".
        - If the user selects **Consumer**:
            - If mutex == 1 and full != 0, call consumer().
            - Otherwise, print "Buffer is empty!!".
        - If the user selects **Exit**, terminate the program.
- **Producer Process**:
- **Wait(mutex)**: Lock the buffer.
- **Signal(full)**: Increment the number of full slots.
- **Wait(empty)**: Decrement the number of empty slots.
- Produce an item and increment x.
- **Signal(mutex)**: Unlock the buffer.
- **Consumer Process**:
- **Wait(mutex)**: Lock the buffer.
- **Wait(full)**: Decrement the number of full slots.
- **Signal(empty)**: Increment the number of empty slots.
- Consume an item and decrement x.
- **Signal(mutex)**: Unlock the buffer.
- **End**.

Program Code:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0; //Initialization of semaphores
// mutex = 1
//Full = 0 - Initially, all slots are empty. Thus full slots are 0
//Empty = 3 // All slots are empty initially
void producer();
void consumer();
int wait(int);
int signal(int);
        int main()
        {
                int n;
                printf("\n1.Producer\n2.Consumer\n3.Exit");
        while(1)
        {
                printf("\nEnter your choice:");
                scanf("%d",&n);
                switch(n)
                {
                        case 1: if((mutex==1)&&(empty!=0))
                        producer();
                        else
                        printf("Buffer is full!!");
                        break;
                        case 2: if((mutex==1)&&(full!=0))
                        consumer();
                        else
                        printf("Buffer is empty!!");
                        break;
                        case 3:
                        exit(0);
                        break;
                }
        }
        return 0;
        }

int wait(int s)
```

```
{
        return (--s);
}
int signal(int s)
{
        return(++s);
}
void producer()
{
        mutex=wait(mutex);
        full=signal(full);
        //producer has placed the item and thus the value of "full" is
        //increased by 1
        empty=wait(empty);
        //producer produces an item then the value of "empty" is
        //reduced by 1
        //because one slot will be filled now
        x++;
        printf("\nProducer produces the item %d",x);
        mutex=signal(mutex);
}
void consumer()
{
        mutex=wait(mutex);
        full=wait(full);
        //consumer is removing an item from buffer, therefore
        //value of "full" is reduced by 1
        empty=signal(empty);
        //consumer has consumed the item, thus increasing the value of "empty"
        //by 1
        printf("\nConsumer consumes item %d",x);
        x--;
        mutex=signal(mutex);
}
```

Sample Output 1:

Scenario 1: Normal Operation

1. Producer
2. Consumer
3. Exit

Enter your choice: 1

Producer produces the item 1

1. Producer
2. Consumer
3. Exit

Enter your choice: 1

Producer produces the item 2

1. Producer
2. Consumer
3. Exit

Enter your choice: 2

Consumer consumes item 2

1. Producer
2. Consumer
3. Exit

Enter your choice: 1

Producer produces the item 3

1. Producer
2. Consumer
3. Exit

Enter your choice: 2

Consumer consumes item 3

1. Producer
2. Consumer
3. Exit

Enter your choice: 2

Consumer consumes item 1

1. Producer
2. Consumer
3. Exit
Enter your choice: 3

Sample Output 2:

Scenario 2: Buffer Full and Empty Conditions

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:1
Buffer is full!!
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3

Date: __/__/20__

7. Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

PROGRAM DESCRIPTION:

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Algorithm:

1. Start

2. Input:

- Number of processes P and resources R.
- Matrix Allocation[P][R]: Allocated resources for each process.
- Matrix Max[P][R]: Maximum resources required by each process.
- Vector Available[R]: Available resources in the system.

3.Compute Need matrix:

- For each process i and resource j:
- Need[i][j] = Max[i][j] - Allocation[i][j]

4.Initialize:

- Array Finish[P] to false (i.e., no process has finished yet).
- Count count = 0 (number of processes in the safe sequence).

5.While count < P:

- For each process i:
- ✓ If Finish[i] == false (the process has not finished):
- ✓ Check if Need[i][j] <= Available[j] for all resources j.
- ✓ If true:
- Mark the process i as finished (Finish[i] = true).
- Add the process to the safeSequence[].
- Release its resources: Available[j] += Allocation[i][j].
- Increment count (count++).

6.Check for Deadlock:

If no process can be executed safely in a round, declare "System is not in a safe state".

7.Output:

If all processes can be safely executed, print "System is in a safe state" and the safe sequence.

8.End

Program Code:

```
#include <stdio.h>
#define MAX 10

int main()
{
   int allocation[MAX][MAX], max[MAX][MAX], available[MAX],
need[MAX][MAX];
   int processes, resources;
   int finish[MAX] = {0}, safeSequence[MAX];

   // Input number of processes and resources
   printf("Enter number of processes: ");
   scanf("%d", &processes);

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

   // Input allocation matrix
   printf("Enter allocation matrix:\n");
   for (int i = 0; i < processes; i++)
        {
                for (int j = 0; j < resources; j++)
                {
                        scanf("%d", &allocation[i][j]);
                }
         }

   // Input max matrix
   printf("Enter max matrix:\n");
   for (int i = 0; i < processes; i++)
        {
                for (int j = 0; j < resources; j++)
                {
                        scanf("%d", &max[i][j]);
                }
        }

   // Input available resources
   printf("Enter available resources:\n");
   for (int i = 0; i < resources; i++)
```

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

// Calculate need matrix: Need = Max - Allocation
for (int i = 0; i < processes; i++)
    {
        for (int j = 0; j < resources; j++)
        {
            need[i][j] = max[i][j] - allocation[i][j];
        }
     }

// Banker's Algorithm to check for safe sequence
int count = 0;
for (int k = 0; k < processes; k++)
    {
        for (int i = 0; i < processes; i++)
        {
            if (!finish[i])
            {
                int flag = 1;
                for (int j = 0; j < resources; j++)
                {
                    if (need[i][j] > available[j])
                    {
                    flag = 0;
                     break;
                    }
                }
        if (flag)
            {
                for (int j = 0; j < resources; j++)
            {
                available[j] += allocation[i][j];
            }
            safeSequence[count++] = i;
            finish[i] = 1;
            }
            }
    }
}
```

```
    // Check if system is in a safe state
    if (count == processes) {
        printf("System is in a safe state.\nSafe sequence: ");
        for (int i = 0; i < processes; i++)
            {
              printf("P%d ", safeSequence[i]);
            }
        printf("\n");
    } else
            {
                    printf("System is not in a safe state.\n");
             }
    return 0;
}
```

Sample Output 1:

Safe State

```
Enter number of processes: 4
Enter number of resources: 3

Enter allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1

Enter max matrix:
7 5 3
3 2 2
9 0 2
2 2 2

Enter available resources:
3 3 2

System is in a safe state.
Safe sequence: P1 P3 P4 P0
```

Sample Output 2:

Unsafe State

```
Enter number of processes: 3
Enter number of resources: 2

Enter allocation matrix:
1 0
1 1
0 1

Enter max matrix:
2 1
2 2
1 2

Enter available resources:
1 1

System is not in a safe state.
```

Date: __/__/20__

8. Write a C program to simulate FIFO page replacement algorithm

PROGRAM DESCRIPTION:
In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Algorithm:

1. **Input:**
- Read the number of pages (n).
- Read the reference string (a[]).
- Read the number of frames (no).
2. **Initialization:**
- Set all frames to -1 (empty).
- Set j = 0 to point to the first frame for replacement.
- Initialize count = 0 to keep track of page faults.
3. **For each page in the reference string:**
- Check availability:
    - If the page is already in one of the frames, continue (no page fault).
    - If the page is not in the frames (avail == 0):
        1. Replace the page in frame j.
        2. Increment j in a circular manner (j = (j + 1) % no).
        3. Increment the page fault counter (count++).
        4. Print the current frame content.
4. **End:**
- After all pages are processed, print the total number of page faults.

Program Code:

```
#include<stdio.h>
int main()
{
	int i,j,n,a[50],frame[10],no,k,avail,count=0;
	printf("\n ENTER THE NUMBER OF PAGES:\n");
	scanf("%d",&n);
	printf("\n ENTER THE PAGE NUMBER :\n");
	for(i=1;i<=n;i++)
	scanf("%d",&a[i]);
	printf("\n ENTER THE NUMBER OF FRAMES :");
	scanf("%d",&no);
	for(i=0;i<no;i++)
	frame[i]= -1;
	j=0;
	printf("\tref string\t page frames\n");
	for(i=1;i<=n;i++)
	{
		printf("%d\t\t",a[i]);
		avail=0;
		for(k=0;k<no;k++)
		if(frame[k]==a[i])
		avail=1;
		if (avail==0)
		{
			frame[j]=a[i];
			j=(j+1)%no;
			count++;
			for(k=0;k<no;k++)
			printf("%d\t",frame[k]);
		}
		printf("\n");
	}
	printf("Page Fault is %d",count);
	return 0;
}
```

Sample Output 1:

```
ENTER THE NUMBER OF PAGES:
20

 ENTER THE PAGE NUMBER :
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

 ENTER THE NUMBER OF FRAMES :3
        ref string      page frames
7               7       -1      -1
0               7       0       -1
1               7       0       1
2               2       0       1
0
3               2       3       1
0               2       3       0
4               4       3       0
2               4       2       0
3               4       2       3
0               0       2       3
3
2
1               0       1       3
2               0       1       2
0
1
7               7       1       2
0               7       0       2
1               7       0       1
Page Fault is 15
```

Sample Output 2:

```
ENTER THE NUMBER OF PAGES:
8
 ENTER THE PAGE NUMBER :
1 2 3 4 5 6 7 8

 ENTER THE NUMBER OF FRAMES :4
        ref string      page frames
1               1       -1      -1      -1
2               1       2       -1      -1
3               1       2       3       -1
4               1       2       3       4
5               5       2       3       4
6               5       6       3       4
7               5       6       7       4
8               5       6       7       8
Page Fault is 8
```

Date: __/__/20__

9. Write a C program to simulate file organization technique using single level directory.

PROGRAM DESCRIPTION:
It is the simplest of all directory structures, in this the directory system having only one directory, it consisting of all files. Sometimes it is said to be the root directory.

Algorithm:

1. **Input:**
- Read the directory name (mdname).
- Read the number of files (nf).
2. **Loop for file creation:**
- Do while loop:
    1. Prompt the user to enter a new file name.
    2. Check for duplicates:
        - Loop through the existing file names and compare them with the newly entered file name using strcmp().
    3. If the file name is unique:
        - Add the file to the directory (fname[][]).
        - Increment the file counter (j++ and nf++).
    4. If the file name already exists:
        - Print a message indicating that the file name already exists.
    5. Ask the user if they want to add another file. Continue the loop if they choose to do so.
3. **End:**
- After the loop ends, print the directory name and list all the files in the directory.

Program Code:

```
#include<stdio.h>
#include<string.h>
int main()
{
        int nf=0,i=0,j=0,ch;
        char mdname[10],fname[10][10],name[10];
        printf("\nEnter the directory name:");
        scanf("%s",mdname);
        printf("\nEnter the number of files:");
        scanf("%d",&nf);
        do
        {
                printf("\nEnter file name to be created:");
                scanf("%s",name);
                for(i=0;i<nf;i++)
                {
                        if(!strcmp(name,fname[i]))
                        break;
                }
        if(i==nf)
        {
                strcpy(fname[j++],name);
                nf++;
        }
        else
        printf("\nThere is already %s\n",name);
        printf("\nDo you want to enter another file (yes - 1 or no - 0):");
        scanf("%d",&ch);
        }
        while(ch==1);
        printf("\nDirectory name is: %s\n",mdname);
        printf("\nFiles names are:");
        for(i=0;i<j;i++)
        printf("\n%s",fname[i]);
        return 0;
}
```

Sample Output 1:

Basic Scenario with No Duplicate Files

```
Enter the directory name: MyFiles
Enter the number of files: 0
Enter file name to be created: file1
Do you want to enter another file (yes - 1 or no - 0): 1
Enter file name to be created: file2
Do you want to enter another file (yes - 1 or no - 0): 1
Enter file name to be created: file3
Do you want to enter another file (yes - 1 or no - 0): 0

Directory name is: MyFiles
Files names are:
file1
file2
file3
```

Sample Output 2:

Scenario with Duplicate Files

```
Enter the directory name: Documents
Enter the number of files: 0
Enter file name to be created: report
Do you want to enter another file (yes - 1 or no - 0): 1
Enter file name to be created: report
File report already exists.
Do you want to enter another file (yes - 1 or no - 0): 1
Enter file name to be created: summary
Do you want to enter another file (yes - 1 or no - 0): 0

Directory name is: Documents
Files names are:
report
summary
```

Date: __/__/20__

10. Write a C program to simulate Indexed file allocation strategy.

PROGRAM DESCRIPTION:
Indexed allocation supports both sequential and direct access files. The file indexes are not physically stored as a part of the file allocation table. Whenever the file size increases, we can easily add some more blocks to the index. In this strategy, the file allocation table contains a single entry for each file. The entry consisting of one index block, the index blocks having the pointers to the other blocks. No external fragmentation.

Algorithm:

1. **Input**:
- Read the number of files (n).
- For each file:
    1. Read the starting block (sb[i]).
    2. Read the size of the file (s[i]).
    3. Read the number of blocks allocated (m[i]).
    4. Read the block numbers (b[i][j]).
2. **Display file information**:
- Print the file index, starting block, and the number of blocks allocated.
3. **Request file information**:
- Ask the user to input a file number (x).
- Display the index and the list of blocks allocated to the requested file.

Program Code:

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

int main() {
    int n, m[20], i, j, sb[20], s[20], b[20][20], x;

    // Input number of files
    printf("Enter number of files: ");
    scanf("%d", &n);

    // Input details for each file
    for (i = 0; i < n; i++) {
```

```
    printf("\nEnter starting block and size of file %d: ", i + 1);
    scanf("%d%d", &sb[i], &s[i]);

    printf("\nEnter number of blocks occupied by file %d: ", i + 1);
    scanf("%d", &m[i]);

    printf("\nEnter blocks of file %d: ", i + 1);
    for (j = 0; j < m[i]; j++)
        scanf("%d", &b[i][j]);
}

// Print file information
printf("\nFile\tIndex\tLength\n");
for (i = 0; i < n; i++) {
    printf("%d\t%d\t%d\n", i + 1, sb[i], m[i]);
}

// Query for a specific file
printf("\nEnter file number to query (1 to %d): ", n);
scanf("%d", &x);

if (x < 1 || x > n) {
    printf("Invalid file number!\n");
    return 1; // Exit with error code
}

printf("\nFile number is: %d\n", x);
i = x - 1;
printf("Index is: %d\n", sb[i]);
printf("Blocks occupied are: ");
for (j = 0; j < m[i]; j++)
    printf("%3d ", b[i][j]);
printf("\n");

return 0;
}
```

Sample Output 1:

Basic Operation with Two Files

```
Enter number of files: 2
Enter starting block and size of file 1: 10 50
Enter number of blocks occupied by file 1: 3
Enter blocks of file 1: 1 2 3
Enter starting block and size of file 2: 20 100
Enter number of blocks occupied by file 2: 4
Enter blocks of file 2: 4 5 6 7
File    Index Length
1       10    3
2       20    4

Enter file number to query (1 to 2): 1
File number is: 1
Index is: 10
Blocks occupied are:   1   2   3
```

Sample Output 2:

Querying an Invalid File Number

```
Enter the number of files: 3
Enter starting block and size of file 1: 300 70
Enter number of blocks occupied by file 1: 4
Enter blocks of file 1: 1 2 3 4
Enter starting block and size of file 2: 400 30
Enter number of blocks occupied by file 2: 2
Enter blocks of file 2: 5 6
Enter starting block and size of file 3: 500 40
Enter number of blocks occupied by file 3: 3
Enter blocks of file 3: 7 8 9
Enter file number to query (1 to 3): 4

File    Index   Length
1       300     4
2       400     2
3       500     3

Enter file number to query (1 to 3): 4

Invalid file number!
```

OPEN ENDED EXPERIMENTS

1. Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

2. Implement a C program to demonstrate the dining philosopher problem.

3. Implement a C program to demonstrate the SCAN Disk scheduling algorithm.

4. Write a C program to simulate paging technique of memory management.

5. Write a C Program to Implement and test strategies like checkpointing, rollback, or redundancy. Evaluate their effectiveness in maintaining system stability and data integrity in the event of failures.

6. Write a C program to Test the protocol's performance under various network conditions such as high latency, packet loss, or different traffic loads.

7. Implement a C Program to compare different memory allocation strategies such as best-fit, worst-fit, and buddy allocation.

8.Implement a C program to demonstrate Earliest Deadline First (EDF) & Rate Monotonic Scheduling (RMS)

9.Implement a C Program to demonstrate Least Laxity First (LLF) & Lottery Scheduling

Note: All the students should compulsory complete the open-ended experiments.