Vidyavardhini's
College of Engineering & Technology
Vasai Road (W)

Department of Artificial Intelligence & Data Science

Laboratory Manual
Student Copy

Semester	IV	Class	S.E
Course Code	CSL403		
Course Name	Operating System Lab		

Vidyavardhini's College of Engineering & Technology

Vision

To be a premier institution of technical education; always aiming at becoming a valuable resource for industry and society.

Mission

- To provide technologically inspiring environment for learning.
- To promote creativity, innovation and professional activities.
- To inculcate ethical and moral values.
- To cater personal, professional and societal needs through quality education.

Department Vision:

To foster proficient artificial intelligence and data science professionals, making remarkable contributions to industry and society.

Department Mission:

- To encourage innovation and creativity with rational thinking for solving the challenges in emerging areas.
- To inculcate standard industrial practices and security norms while dealing with Data.
- To develop sustainable Artificial Intelligence systems for the benefit of various sectors.

Program Specific Outcomes (PSOs):

PSO1: Analyze the current trends in the field of Artificial Intelligence & Data Science and convey their finding by presenting / publishing at a national / international forums.

PSO2: Design and develop Artificial Intelligence & Data Science based solutions and applications for the problems in the different domains catering to industry and society.

Program Outcomes (POs):

Engineering Graduates will be able to:

- **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 teamwork:** 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.

Course Objectives

CSL403: Operating System Lab

1	To gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, memory management, file systems and deadlock handling using C language in Linux environment
2	To familiarize students with the architecture of Linux OS.
3	To provide necessary skills for developing and debugging programs in Linux environment.
4	To learn programmatically to implement simple operation system mechanisms

Course Outcomes

CO	At the end of course students will be able to:	**Action verbs**	**Bloom's Level**
CSL801.1	Demonstrate basic Operating system Commands, Shell scripts, System Calls and API wrt Linux	Apply	Apply (level 3)
CSL801.2	Implement various process scheduling algorithms and evaluate their performance	Apply	Apply (level 3)
CSL801.3	Implement and analyze concepts of synchronization and deadlocks.	Apply	Apply (level 3)
CSL801.4	Implement various Memory Management techniques and evaluate their performance.	Apply	Apply (level 3)
CSL801.5	Implement and analyze concepts of virtual memory	Apply	Apply (level 3)
CSL801.6	Demonstrate and analyze concepts of file management and I/O management techniques.	Apply	Apply (level 3)

Mapping of Experiments with Course Outcomes

List of Experiments	Course Outcomes					
	CSL801.1	CSL801.2	CSL801.3	CSL801.4	CSL801.5	CSL801.6
Implement internal commands of Linux	3	-	-	-	-	-
Write Shell Scripts incorporating Linux Commands	3	-	-	-	-	-
Explore User management commands in Linux.	3		-	-	-	-
Create a child process using fork system call. Obtain process ID of parent and child using getppid and getpid system calls	3	-		-	-	-
Write a program to implement a preemptive process scheduling algorithm	-	3	-	-	-	-
Implement solution of producer consumer problem through semaphore	-	-	3	-	-	-
Implement Banker's algorithm for deadlock avoidance	-	-	3	-	-	-
Build a program to implement page replacement policies	-	-	-	3	-	-

Develop a program to implement dynamic partitioning placement algorithms	-	-	-	-	3	-
Implement Disc Scheduling algorithms	-	-	-	-	-	3

List of Experiments

Sr. No.	Name of Experiment	DOP	DOC	Marks	Sign
Basic Experiments					
1	Implement basic linux commands				
2	Write Shell Scripts incorporating Linux Commands				
3	Explore User management commands in Linux				
4	Create a child process using fork system call. Obtain process ID of parent and child using getppid and getpid system calls				
5	Implement a preemptive process scheduling algorithm				
6	Implement solution of producer consumer problem through semaphore				
7	Implement Banker's algorithm for deadlock avoidance				
8	Implement dynamic partitioning placement algorithms				
9	Implement page replacement policies				
10	Implement Disc Scheduling algorithms				
Project / Assignment					
11	Assignment 1: Operating System Overview				
12	Assignment 2: Process Scheduling				
13	Assignment 3: Deadlocks				

14	Assignment 4:Memory Allocation Strategies				
15	Assignment 5: File Management				
16	Assignment 6: Disk Scheduling Algorithm				
Formative Assessment					
11	Th - Quiz 1: Operating System Overview				
12	Th - Quiz 2: Process Scheduling				
13	Th - Quiz 3: Synchronization &Deadlocks				
14	Th - Quiz 4: Memory Management				
15	Th - Quiz 5: File Management				
16	Th - Quiz 6: I/O Management				
17	Pr - Quiz 1: Linux commands and Shell Scripting				
18	Pr - Quiz 2: Process Scheduling				
19	Pr - Quiz 3: Synchronization & Deadlock				
20	Pr - Quiz 4: Memory management techniques				
21	Pr - Quiz 5: Virtual memory concepts				
22	Pr - Quiz 6: I/O management				

D.O.P: Date of performance

D.O.C : Date of correction

Experiment No. 1
Explore the internal commands of Linux.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Explore the internal commands of Linux.

Objective:

Execute various internal commands of linux

Theory:

ps - report a snapshot of the current processes. ps displays information about a selection of the active processes.

cal — displays a calendar and the date of Easter

date - print or set the system date and time ,Display the current time in the given FORMAT, or set the system date.

rm - remove files or directories

mkdir - make directories , Create the DIRECTORY(ies), if they do not already exist.

rmdir - remove empty directories

cat - concatenate files and print on the standard output

wc - print newline, word, and byte counts for each file, Print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified.

ls - list directory contents

ls [OPTION]... [FILE]...

List information about the FILEs (the current directory by default). Sort entries alphabetically.

-l: use a long listing format

chmod - change file mode bits

chmod changes the file mode bits of each given file according to mode, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.

chown - change file owner and group

chown changes the user and/or group ownership of each given file. If only an owner (a user name or numeric user ID) is given, that user is made the owner of each given file, and the files' group is not changed. If the owner is followed by a colon and a group name (or numeric group ID), with no spaces between them, the group ownership of the files is changed as well.

pwd - print name of current/working directory.

Print the full filename of the current working directory.

umask - set file mode creation mask , umask() sets the calling process's file mode creation mask (umask) to mask & 0777 (i.e., only the file permission bits of mask are used), and returns the previous value of the mask.

Output:

Conclusion:

What Do you mean by System calls?

System calls are fundamental interfaces between a user application and the operating system. When a program running in user mode requires access to system resources or services that only the operating system can provide, it must make a system call. This allows the program to transition from user mode to kernel mode, where the operating system resides, and request the necessary action.

System calls provide a standardized way for applications to interact with the underlying hardware and operating system functionalities. Examples of operations that typically require system calls include reading from or writing to files, creating new processes, allocating memory, managing hardware devices, and performing network communication.

Each operating system has its own set of system calls, and they are usually exposed to user programs through a set of functions provided by the operating system's application programming interface (API). In summary, system calls are crucial for enabling user applications to utilize the full capabilities of the underlying operating system and hardware.

Experiment No.2
Linux shell script 2.1 Write shell scripts to do the following: a. Display OS version, release number, kernel version b. Display top 10 processes in descending order c. Display processes with highest memory usage. d. Display current logged in user and log name. Display current shell, home directory, operating system type, current path setting, current working directory
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Linux shell script 2.1 Write shell scripts to do the following:

Objective:

a. Display OS version, release number, kernel version
b. Display top 10 processes in descending order

c. Display processes with highest memory usage.

d. Display current logged in user and log name.

e. Display current shell, home directory, operating system type, current path setting, current working directory

Theory:

Shell is a user program, or its environment is provided for user interaction. It is a command prompt within Linux where you can type commands. It is a program that takes your commands from the keyboard and gives them to the OS to perform. Shell is not part of system KERNAL but it uses system KERNAL to execute programs, create files,etc. A Shell Script is a text file that contains a sequence of commands for a UNIX based OS. It is called a Shell Script because it combines into a "Script" in a single file a sequence of commands, that would otherwise have to be presented to the system from a keyboard one at a time. A Shell Script is usually created for command sequences for which a user has a repeated need. You initiate the sequence of commands in Shell Script by simply entering the name of the Shell Script on a command line.

Types of Shell Script :-

1. sh - Simple Shell
2. bash - Bourne Again Shell
3. ksh - Korne Shell
4. csh - C Shell
5. ssh - Secure Shell

To use a particular Shell type the Shell name at the command prompt. Eg:- $csh - It will switch the current Shell to C Shell. To view the current Shell that is being used, type echo $ SHELL at the command prompt.

Program:

Output:

Conclusion:

What is a Shell ?

A shell is a command-line interface (CLI) program that serves as the primary interface between a user and an operating system. It allows users to interact with the system by typing commands in a text-based environment rather than relying solely on graphical user interfaces (GUIs). Shells interpret user input, execute commands, and manage the execution of programs. They provide functionalities such as command execution, input/output redirection, piping (connecting the output of one command to the input of another), scripting (automating sequences of commands), and environment variable manipulation. Shells also typically provide features like command history, tab completion, and customizable prompt settings to enhance user productivity. Common examples of shells include Bash (Bourne Again SHell), Zsh (Z shell), and PowerShell, each with its own syntax, features, and extensions. Overall, shells play a crucial role in enabling users to interact with the operating system efficiently and perform various tasks ranging from simple file manipulation to complex system administration tasks.

Name different types of shell

There are several types of shells available in Unix-like operating systems, each with its own features and capabilities.

1. **Bourne Shell (sh): Developed by Stephen Bourne at AT&T Bell Labs, the Bourne Shell was one of the earliest Unix shells. It provides basic command-line interface functionality and scripting capabilities. While it lacks some of the advanced features of later shells, it remains a fundamental component of Unix systems.**

2. **Bourne-Again Shell (bash): Created as an enhanced version of the Bourne Shell, bash has become one of the most widely used shells in Unix-like operating systems. It offers features such as command-line editing, history, aliases, and programmable completion, making it powerful for both interactive use and scripting.**

3. **C Shell (csh): Developed by Bill Joy at the University of California, Berkeley, the C Shell provides a syntax similar to the C programming language. It offers interactive features like command-line history and aliases. While popular among some users, its scripting capabilities are considered less robust compared to bash.**

4. **Korn Shell (ksh): Developed by David Korn at AT&T Bell Labs, the Korn Shell combines features from the Bourne Shell and the C Shell, offering a rich set of programming features for scripting. It provides advanced scripting constructs, command-line editing, and job control, making it popular among power users and system administrators.**

5. **Z Shell (zsh): Zsh is an extended version of the Bourne Shell with additional features such as advanced tab completion, spelling correction, and customizable prompts. It aims to provide an interactive experience with enhanced usability and productivity for users.**

6. **Fish Shell (fish): Fish, short for "friendly interactive shell," is designed to be user-friendly and intuitive for interactive use. It offers features like syntax highlighting, autosuggestions, and powerful scripting capabilities while focusing on simplicity and ease of use.**

Experiment No. 3
Explore Linux Commands
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Explore user management commands of linux.

Objective:

Explore basic commands of linux

Theory:

A user is an entity, in a Linux operating system, that can manipulate files and perform several other operations. Each user is assigned an ID that is unique for each user in the operating system. In this post, we will learn about users and commands which are used to get information about the users. After installation of the operating system, the ID 0 is assigned to the root user and the IDs 1 to 999 (both inclusive) are assigned to the system users and hence the ids for local user begins from 1000 onwards.

In a single directory, we can create 60,000 users. Now we will discuss the important commands to manage users in Linux

- useradd - create a new user or update default new user information ,useradd is a low level utility for adding users.
- userdel - delete a user account and related files
- groupadd - create a new group , The groupadd command creates a new group account using the values specified on the command line plus the default values from the system. The new group will be entered into the system files as needed.
- groupdel - delete a group , The groupdel command modifies the system account files, deleting all
- entries that refer to GROUP. The named group must exist
- who - show who is logged on , Print information about users who are currently logged in.
- whoami - print effective userid
- passwd - change user password

The passwd command changes passwords for user accounts. A normal user may only change the password for his/her own account, while the superuser may change the password for any account. passwd also changes the account or associated password validity period.

1. to enter in root sudo su then password

2. to add new user type useradd csds11 (username)

3. to check a newly added user you have to type cat etc/passwod

4 set a password to new user : sudo passwd csds11

5. create a new group: groupadd csds12

6. Check group cat /etc/group

7. add new user in newly created group useradd -G csds12 piya1 (group name and new user name)

8. to check : cat /etc/group

9. to enter in new user : su - csds11 (username)

10. to delete user type : userdel csds (username that you have to delete)

11. Again check whether it is deleted or not cat /etc/passwd

10. to delete user type : groupdel csds12 (group that you have to delete)

12. Again check whether it is deleted or not cat /etc/passwd

13. who - show who is logged on Print information about users who are currently logged in.

14. whoami - print effective userid

Output

1) useradd

```
ubuntu@ubuntu-HP-280-Pro-G5-Small-Form-Factor-PC:~$ sudo useradd test1
[sudo] password for ubuntu:
ubuntu@ubuntu-HP-280-Pro-G5-Small-Form-Factor-PC:~$ sudo useradd test1
useradd: user 'test1' already exists
```

2) userdel

```
ubuntu@ubuntu-HP-280-Pro-G5-Small-Form-Factor-PC:~$ sudo userdel test1
ubuntu@ubuntu-HP-280-Pro-G5-Small-Form-Factor-PC:~$ cat /etc/passwd | grep test1
ubuntu@ubuntu-HP-280-Pro-G5-Small-Form-Factor-PC:~$ test1
Command 'test1' not found, did you mean:
  command 'testr' from deb python3-testrepository (0.0.20-6)
  command 'test' from deb coreutils (8.32-4.1ubuntu1)
Try: sudo apt install <deb name>
```

3) groupadd

```
ubuntu@ubuntu-hp1:~$ groupadd traitors
groupadd: Permission denied.
groupadd: cannot lock /etc/group; try again later.
```

4) groupdel

```
ubuntu-hp1:~$ groupdel traitors
l: group 'traitors' does not exist
```

5) who

```
ubuntu@ubuntu-hp1:~$ who -q
ubuntu
```

6)whoami

```
ubuntu@ubuntu-hp1:~$ whoami
ubuntu
```

7) passwd

```
ubuntu@ubuntu-hp1:~$ passwd -e user1
passwd: Permission denied.
```

Conclusion:

Explain Linux API?

The Linux API, or Application Programming Interface, serves as a bridge between user applications and the Linux kernel. It encompasses a vast collection of functions, data structures, and conventions that programmers can utilize to interact with the underlying operating system. At its core, the Linux API provides a set of system calls, which are entry points into the kernel that enable applications to request services such as file I/O, process management, memory allocation, and networking. Additionally, the Linux API includes libraries and utilities that simplify common programming tasks, such as string manipulation, mathematical operations, and input/output handling. These libraries, such as the GNU C Library (glibc), provide a higher-level interface while still ultimately relying on system calls to perform their operations. Furthermore, the Linux API extends beyond system calls and libraries to encompass other interfaces like the /proc and /sys filesystems, which expose information about system configuration, processes, and hardware to user-space programs. Overall, the Linux API serves as a comprehensive toolkit for developers to harness the power and versatility of the Linux operating system in their applications.

Experiment No. 4
Create a child process in Linux using the fork system call.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Create a child process in Linux using the fork system call.

Objective:

Create a child process using fork system call.

From the child process obtain the process ID of both child and parent by using getpid and getppid system calls.

Theory:

A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a **new** process, which becomes the child process of the caller.

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the **process ID** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

If the call to **fork()** is executed successfully, Unix will make two identical copies of address spaces, one for the parent and the other for the child.

getpid, getppid - get process identification

- **getpid()** returns the process ID (PID) of the calling process. This is often used by routines that generate unique temporary filenames.
- **getppid()** returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork().

Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   // fork() Create a child process

   int pid = fork();
```

```
    if (pid > 0) {
        printf("I am Parent process\n");
        printf("ID : %d\n\n", getpid());
    }
    else if (pid == 0) {
        printf("I am Child process\n");
        // getpid() will return process id of child process
        printf("ID: %d\n", getpid());

    }
    else {
        printf("Failed to create child process");
    }

    return 0;
}
```

Output:

```
I am Parent process
ID : 959

I am Child process
ID: 960
```

Conclusion:

What do you mean by system call?

A system call serves as a vital communication link between user-level applications and the kernel of an operating system. It facilitates the transfer of control from user space, where applications reside, to kernel space, where the operating system's core functionalities operate. This transition enables user programs to access privileged operations and resources, such as file I/O, network communication, process management, and hardware control, which are typically restricted to the operating system's domain. System calls follow a predefined interface and protocol, allowing applications to request services from the operating system in a standardized manner. Upon receiving a system call request, the kernel executes the requested operation on behalf of the application, ensuring proper security, resource management, and coordination with other processes. Thus, system calls play a fundamental role in enabling user programs to interact with and harness the full capabilities of the underlying operating system and hardware infrastructure.

Experiment No.5

Process Management: Scheduling a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS) b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement process scheduling algorithms FCFS and SJF

Objective:

a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS)
b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)

Theory:

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

Jobs are executed on a first come, first serve basis. It is a non-preemptive, preemptive scheduling algorithm. Easy to understand and implement. Its implementation is based on the FIFO queue. Poor in performance as average wait time is high.

Shortest Job First (SJF)

This is also known as the shortest job first, or SJF. This is a non-preemptive, preemptive scheduling algorithm. Best approach to minimize waiting time. Easy to implement in Batch systems where required CPU time is known in advance. Impossible to implement in interactive systems where required CPU time is not known. The processor should know in advance how much time the process will take.

Program 1:

```
#include<stdio.h>

 int main()

{
   int n, bt[30],wait_t[30],turn_ar_t[30],av_wt_t=0,avturn_ar_t=0,i,j;
   printf("Please enter the total number of processes(maximum 30):");  // the maximum process that be used to calculate is specified.
   scanf("%d",&n);

   printf("\nEnter The Process Burst Timen");
   for(i=0;i<n;i++)  // burst time for every process will be taken as input
   {
```

```
    printf("P[%d]:",i+1);
    scanf("%d",&bt[i]);
  }

  wait_t[0]=0;

  for(i=1;i<n;i++)
  {
    wait_t[i]=0;
    for(j=0;j<i;j++)
      wait_t[i]+=bt[j];
  }

  printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

  for(i=0;i<n;i++)
  {
    turn_ar_t[i]=bt[i]+wait_t[i];
    av_wt_t+=wait_t[i];
    avturn_ar_t+=turn_ar_t[i];
    printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",i+1,bt[i],wait_t[i],turn_ar_t[i]);
  }

  av_wt_t/=i;
  avturn_ar_t/=i;  // average calculation is done here
  printf("\nAverage Waiting Time:%d",av_wt_t);
  printf("\nAverage Turnaround Time:%d",avturn_ar_t);

  return 0;
}

#include<stdio.h>

 int main()

{
  int n,bt[30],wait_t[30],turn_ar_t[30],av_wt_t=0,avturn_ar_t=0,i,j;
  printf("Please enter the total number of processes(maximum 30):");  // the maximum process that be used to
calculate is specified.
  scanf("%d",&n);

  printf("\nEnter The Process Burst Timen");
  for(i=0;i<n;i++)  // burst time for every process will be taken as input
  {
    printf("P[%d]:",i+1);
    scanf("%d",&bt[i]);
  }

  wait_t[0]=0;

  for(i=1;i<n;i++)
```

```
    {
        wait_t[i]=0;
        for(j=0;j<i;j++)
            wait_t[i]+=bt[j];
    }

    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

    for(i=0;i<n;i++)
    {
        turn_ar_t[i]=bt[i]+wait_t[i];
        av_wt_t+=wait_t[i];
        avturn_ar_t+=turn_ar_t[i];
        printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",i+1,bt[i],wait_t[i],turn_ar_t[i]);
    }

    av_wt_t/=i;
    avturn_ar_t/=i;  // average calculation is done here
    printf("\nAverage Waiting Time:%d",av_wt_t);
    printf("\nAverage Turnaround Time:%d",avturn_ar_t);

    return 0;
}
```

Output:

Output

```
/tmp/pyc6U1ggTT.o
Please enter the total number of processes(maximum 30):3

Enter The Process Burst TimenP[1]:7
P[2]:18
P[3]:45

Process     Burst Time  Waiting Time    Turnaround Time
P[1]        7           0               7
P[2]        18          7               25
P[3]        45          25              70
Average Waiting Time:10
Average Turnaround Time:34
```

Program 2:

```
#include<stdio.h>

int main() {
  int time, burst_time[10], at[10], sum_burst_time = 0, smallest, n, i;
  int sumt = 0, sumw = 0;
  printf("enter the no of processes : ");
  scanf("%d", & n);
  for (i = 0; i < n; i++) {
    printf("the arrival time for process P%d : ", i + 1);
    scanf("%d", & at[i]);
    printf("the burst time for process P%d : ", i + 1);
    scanf("%d", & burst_time[i]);
    sum_burst_time += burst_time[i];
  }
  burst_time[9] = 9999;
  for (time = 0; time < sum_burst_time;) {
    smallest = 9;
    for (i = 0; i < n; i++) {
      if (at[i] <= time && burst_time[i] > 0 && burst_time[i] < burst_time[smallest])
        smallest = i;
    }
    printf("P[%d]\t|\t%d\t|\t%d\n", smallest + 1, time + burst_time[smallest] - at[smallest], time - at[smallest]);
    sumt += time + burst_time[smallest] - at[smallest];
    sumw += time - at[smallest];
    time += burst_time[smallest];
    burst_time[smallest] = 0;
  }
  printf("\n\n average waiting time = %f", sum w * 1.0 / n);
  printf("\n\n average turnaround time = %f", sum t * 1.0 / n);
  return 0;
}
```

Output:

Output

```
/tmp/LwPZwwkQiB.o
enter the no of processes : 2
the arrival time for process P1 : 10
the burst time for process P1 : 5
the arrival time for process P2 : 6
the burst time for process P2 : 3
P[10]   |   9999    |   0


 average waiting time = 0.000000

 average turnaround time = 4999.500000
```

Conclusion:

What is the difference between Preemptive and Non-Preemptive algorithms?

Preemptive and non-preemptive scheduling algorithms are two different approaches used by operating systems to manage the execution of multiple processes or threads. In preemptive scheduling, the operating system can interrupt a currently running process and allocate the CPU to another process according to priority or a predefined scheduling algorithm. This means that higher-priority processes can preempt lower-priority ones, potentially leading to more equitable CPU allocation and better responsiveness. Preemptive scheduling is often associated with real-time operating systems and environments where responsiveness is critical.

On the other hand, non-preemptive scheduling, also known as cooperative scheduling, allows a process to continue running until it voluntarily relinquishes the CPU or blocks for I/O. In this approach, the operating system cannot force a process to give up the CPU, and scheduling decisions are typically made based on process priorities, arrival times, or other criteria when a process voluntarily yields control. Non-preemptive scheduling can be simpler to implement and may result in less overhead compared to preemptive scheduling, but it can also lead to potential issues such as priority inversion and reduced responsiveness if a high-priority process is waiting for a low-priority one to finish.

In summary, the key distinction between preemptive and non-preemptive scheduling algorithms lies in whether the operating system can forcibly interrupt a running process or if processes are allowed to continue executing until they voluntarily yield control. Each approach has its own advantages and disadvantages, and the choice between them depends on factors such as system requirements, performance goals, and the nature of the workload being managed.

Experiment No.6
Process Management: Synchronization a. Write a C program to implement the solution of the Producer consumer problem through Semaphore.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Write a C program to implement solution of Producer consumer problem through Semaphore

Objective:

Solve the producer consumer problem based on semaphore

Theory:

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer. Producer consumer problem is a classical synchronization problem.

A **semaphore** S is an integer variable that can be accessed only through two standard operations : wait() and signal().
The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

```
wait(S){

while(S<=0);  // busy waiting

S--;

}

signal(S){

S++;

}
```

To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Initialization of semaphores –
mutex = 1
Full = 0// Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

Solution for Producer –

```
do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer


signal(mutex);

signal(full);

}while(true)
```

When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of "full" is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Solution for Consumer –

do{

wait(full);

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

Program:

```
#include <stdio.h>

int buffer = 0;

void producer() {
   buffer++;
   printf("Producer produces item %d\n", buffer);
}

void consumer() {
   if (buffer > 0) {
      printf("Consumer consumes item %d\n", buffer);
      buffer--;
   } else {
      printf("Buffer is empty!\n");
   }
}

int main() {
```

```
    int choice;
    while (1) {
        printf("1. Press 1 for Producer\n2. Press 2 for Consumer\n3. Press 3 to Exit\n");
        printf("Enter your choice:");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                producer();
                break;
            case 2:
                consumer();
                break;
            case 3:
                return 0;
            default:
                printf("Invalid choice!\n");
                break;
        }
    }
    return 0;
}
```

Output:

```
Output

/tmp/tl2by3Qxqf.o
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 to Exit
Enter your choice:1
Producer produces item 1
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 to Exit
Enter your choice:2
Consumer consumes item 1
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 to Exit
Enter your choice:3


=== Code Execution Successful ===
```

Conclusion:

What is Semaphore?

A semaphore is a synchronization primitive used in concurrent programming to control access to shared resources. It is essentially a counter that is used to manage access to a finite set of resources, such as a critical section of code, a shared memory buffer, or a shared data structure. Semaphores can be thought of as signaling mechanisms that allow threads or processes to coordinate their activities and avoid race conditions, where multiple entities attempt to access or modify the same resource simultaneously.

There are two types of semaphores: binary semaphores and counting semaphores. Binary semaphores, also known as mutexes (short for mutual exclusion), have two states: locked (1) and unlocked (0). They are typically used to protect critical sections of code, allowing only one thread or process to access the protected resource at a time. Counting semaphores, on the other hand, can have a value greater than one and are used to control access to multiple instances of a resource. Threads or processes can increment or decrement the value of a counting semaphore, depending on whether they are acquiring or releasing the resource.

What are different types of Semaphore?

Semaphores are synchronization primitives used in concurrent programming to control access to shared resources. They come in different types, each with its own characteristics and use cases.

1. **Binary Semaphores**: Also known as mutexes (mutual exclusion semaphores), binary semaphores are the simplest type. They have only two states: locked (1) and unlocked (0). Binary semaphores are typically used to protect critical sections of code or to synchronize access to a single resource between multiple threads or processes. They ensure that only one thread or process can access the resource at a time.

2. **Counting Semaphores**: Unlike binary semaphores, counting semaphores can have a value greater than 1. They allow multiple threads or processes to access a shared resource simultaneously, up to a specified maximum limit. Counting semaphores are often used to control access to a finite pool of identical resources, such as a fixed-size buffer or a pool of worker threads. Each time a resource is acquired, the semaphore's value is decremented, and it's incremented when the resource is released.

3. **Named Semaphores**: Named semaphores are semaphores that are associated with a unique name within the operating system's namespace. They allow unrelated processes to synchronize access to shared resources by using the same named semaphore. Named semaphores are commonly used for inter-process communication (IPC), where multiple processes need to coordinate their activities or share data.

4. **Unnamed Semaphores**: Also referred to as anonymous semaphores, unnamed semaphores are not associated with a specific name. They are typically used for synchronization between threads within the same process. Unnamed semaphores are created dynamically within the program and are often implemented as variables within shared memory or as part of a synchronization primitive provided by threading libraries.

5. **Binary vs. Counting Semaphores**: While binary and counting semaphores serve similar purposes, they differ in their usage scenarios. Binary semaphores are well-suited for protecting critical sections and implementing mutual exclusion, where only one entity should access a resource at a time.

Experiment No.7
Process Management: Deadlock a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Process Management: Deadlock

Objective:

a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm

Theory:

It is a banker algorithm used to avoid deadlock and allocate resources safely to each process in the computer system. The 'S-State' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the Banker's Algorithm in detail. Also, we will solve problems based on the Banker's Algorithm. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an operating system. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as deadlock avoidance algorithm or deadlock detection in the operating system.

Data Structures for the Banker's Algorithm.

Let n = number of processes, and m = number of resources types.

v Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available

v Max: n x m matrix.

If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

v Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

v Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize:
   Work = Available
   Finish [i] = false for i = 0, 1, …, n- 1
2. Find an i such that both:
   (a) Finish [i] = false

   (a) Needi ≤ Work

   If no such i exists, go to step 4
3. Work = Work + Allocationi
   Finish[i] = true
   go to step 2
4. If Finish [i] == true for all i, then the system is in a safe state.

Resource-Request Algorithm for Process Pi

Request i = request vector for process Pi . If Requesti [j] = k then process Pi wants k instances of resource type Rj

1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available 3. Pretend to allocate requested resources to Pi by modifying the state as follows:
Available = Available – Requesti ;
Allocationi = Allocationi + Requesti ;
Needi = Needi – Requesti ;
1. If safe ⇒ the resources are allocated to Pi
2. If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored.

Program:

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
        // P0, P1, P2, P3, P4 are the Process names here

        int n, m, i, j, k;
        n = 5;
        m = 3; //
        int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                                                { 2, 0, 0 }, // P1
                                                { 3, 0, 2 }, // P2
                                                { 2, 1, 1 }, // P3
                                                { 0, 0, 2 } }; // P4

        int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                                        { 3, 2, 2 }, // P1
                                        { 9, 0, 2 }, // P2
                                        { 2, 2, 2 }, // P3
                                        { 4, 3, 3 } }; // P4

        int avail[3] = { 3, 3, 2 }; // Available Resources

        int f[n], ans[n], ind = 0;
        for (k = 0; k < n; k++) {
                f[k] = 0;
        }
        int need[n][m];
```

```
        for (i = 0; i < n; i++) {
                for (j = 0; j < m; j++)
                        need[i][j] = max[i][j] - alloc[i][j];
        }
        int y = 0;
        for (k = 0; k < 5; k++) {
                for (i = 0; i < n; i++) {
                        if (f[i] == 0) {

                                int flag = 0;
                                for (j = 0; j < m; j++) {
                                        if (need[i][j] > avail[j]){
                                                flag = 1;
                                                break;
                                        }
                                }

                                if (flag == 0) {
                                        ans[ind++] = i;
                                        for (y = 0; y < m; y++)
                                                avail[y] += alloc[i][y];
                                        f[i] = 1;
                                }
                        }
                }
        }

        int flag = 1;

        for(int i=0;i<n;i++)
        {
        if(f[i]==0)
        {
                flag=0;
                printf("The following system is not safe");
                break;
        }
        }

        if(flag==1)
        {
        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
                printf(" P%d ->", ans[i]);
        printf(" P%d", ans[n - 1]);
        }

        return (0);
}
```

Output:

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

Conclusion:

When can we say that the system is in a safe or unsafe state?

We can determine whether a computer system is in a safe or unsafe state primarily in the context of concurrent execution and resource allocation. In a safe state, the system can ensure that all processes can eventually complete their execution without entering a deadlock, where processes are indefinitely blocked waiting for resources held by others. Additionally, a safe state guarantees that processes can proceed without violating the system's integrity or causing unexpected failures.

Several criteria help evaluate a system's safety:

1. **Deadlock Avoidance**: A system is considered safe if it can avoid deadlock situations entirely or resolve them promptly if they occur. Deadlock occurs when each process in a set is waiting for an event that only another process in the set can cause.

2. **Resource Allocation**: A safe system manages resource allocation effectively, ensuring that processes acquire the resources they need without leading to resource starvation or deadlock. Resource allocation strategies such as bankers' algorithm ensure that processes are allocated resources in a way that avoids deadlock and ensures progress.

3. **Consistency and Integrity**: A safe system maintains data consistency and integrity, preventing processes from interfering with each other's data or corrupting shared resources. Techniques like mutual exclusion, synchronization, and transaction management help uphold data integrity and consistency.

4. **Fault Tolerance**: A safe system can handle failures gracefully, ensuring that a single process failure or system crash does not compromise the overall stability or functionality of the system. Redundancy, error handling mechanisms, and fault-tolerant designs contribute to system safety in the face of failures.

5. **Security**: A safe system protects against unauthorized access, malicious attacks, and data breaches. Security measures such as access control, encryption, authentication, and intrusion detection enhance the safety of the system by safeguarding against threats and vulnerabilities.

Overall, a system is considered safe when it can maintain stability, integrity, availability, and security while efficiently managing resources and accommodating concurrent execution. Unsafe conditions arise when the system fails to meet these criteria, leading to deadlocks, resource contention, data corruption, failures, or security breaches. Therefore, ensuring system safety requires careful design, implementation, and management of the system's resources, processes, and security measures

Experiment No. 8
Memory Management a. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement memory allocation strategy First fit.

Objective:

a. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.

Theory:

The primary role of the memory management system is to satisfy requests for memory allocation. Sometimes this is implicit, as when a new process is created. At other times, processes explicitly request memory. Either way, the system must locate enough unallocated memory and assign it to the process.

Partitioning: The simplest methods of allocating memory are based on dividing memory into areas with fixed partitions.

Selection Policies: If more than one free block can satisfy a request, then which one should we pick? There are several schemes that are frequently studied and are commonly used.

First Fit: In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

- **Advantage:** Fastest algorithm because it searches as little as possible.
- **Disadvantage:** The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished
- Best Fit: The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.
- Worst fit: In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Next Fit: If we want to spread the allocations out more evenly across the memory space, we often use a policy called next fit. This scheme is very similar to the first fit approach, except for the place where the search starts.

Program:

```
// C program for FIFO page replacement algorithm
#include<stdio.h>
int main()
{
   int incomingStream[] = {4, 1, 2, 4, 5};
   int pageFaults = 0;
   int frames = 3;
   int m, n, s, pages;

   pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

   printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");
   int temp[frames];
   for(m = 0; m < frames; m++)
   {
      temp[m] = -1;
   }
```

```
    for(m = 0; m < pages; m++)
    {
        s = 0;

        for(n = 0; n < frames; n++)
        {
            if(incomingStream[m] == temp[n])
            {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;

        if((pageFaults <= frames) && (s == 0))
        {
            temp[m] = incomingStream[m];
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = incomingStream[m];
        }

        printf("\n");
        printf("%d\t\t\t",incomingStream[m]);
        for(n = 0; n < frames; n++)
        {
            if(temp[n] != -1)
                printf(" %d\t\t\t", temp[n]);
            else
                printf(" - \t\t\t");
        }
    }

    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

Output:

```
Output

/tmp/Msa04QZ3Nq.o
Incoming     Frame 1     Frame 2     Frame 3
4            4           -           -
1            4           1           -
2            4           1           2
4            4           1           2
5            5           1           2
Total Page Faults:  4


=== Code Execution Successful ===
```

Conclusion:

Why do we need memory allocation strategies?

Memory allocation strategies are essential for efficient utilization of computer memory in any computing system. These strategies govern how memory is allocated and deallocated to processes or programs running on the system. There are several reasons why these strategies are necessary:

1. **Optimal Resource Utilization**: Memory is a finite resource, and efficient allocation ensures that it is used optimally. By employing appropriate allocation strategies, the system can maximize the amount of memory available for running processes, thereby improving overall system performance.

2. **Prevention of Fragmentation**: Memory fragmentation occurs when memory is allocated and deallocated in a way that leaves small pockets of unused memory scattered throughout the address space. This fragmentation can lead to inefficient memory usage and may eventually result in fragmentation-related issues such as memory exhaustion. Memory allocation strategies help mitigate fragmentation by organizing memory allocation and deallocation in a structured manner.

3. **Fairness and Prioritization**: Different processes or programs running on a system may have varying memory requirements. Memory allocation strategies can prioritize certain processes over others based on factors such as process priority, resource usage, or specific requirements. This ensures fair allocation of memory resources and prevents resource starvation for critical processes.

4. **Performance Optimization**: Efficient memory allocation strategies can significantly impact system performance. For example, strategies that minimize memory overhead or reduce access latency can lead to faster program execution and improved responsiveness of the system as a whole.

5. **Memory Protection and Security**: Memory allocation strategies play a crucial role in enforcing memory protection and security mechanisms. By carefully managing memory allocation and access permissions, these strategies help prevent unauthorized access to sensitive data and mitigate security vulnerabilities such as buffer overflows or memory corruption attacks.

Overall, memory allocation strategies are indispensable for ensuring the smooth operation, performance, and security of computing systems by effectively managing the allocation and utilization of memory resources.

Experiment No.9
Memory Management: Virtual Memory a Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU, Optimal
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Memory Management: Virtual Memory

Objective:

To study and implement page replacement policy FIFO, LRU, OPTIMAL

Theory:

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated.

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference.

First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, the OS maintains a queue that keeps track of all the pages in memory, with the oldest page at the front and the most recent page at the back.
When there is a need for page replacement, the FIFO algorithm, swaps out the page at the front of the queue, that is the page which has been in the memory for the longest time.

Least Recently Used (LRU)

Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.

In LRU, whenever page replacement happens, the page which has not been used for the longest amount of time is replaced.

Optimal Page Replacement

Optimal Page Replacement algorithm is the best page replacement algorithm as it gives the least number of page faults. It is also known as OPT, clairvoyant replacement algorithm, or Belady's optimal page replacement policy.

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future, i.e., the pages in the memory which are going to be referred farthest in the future are replaced.

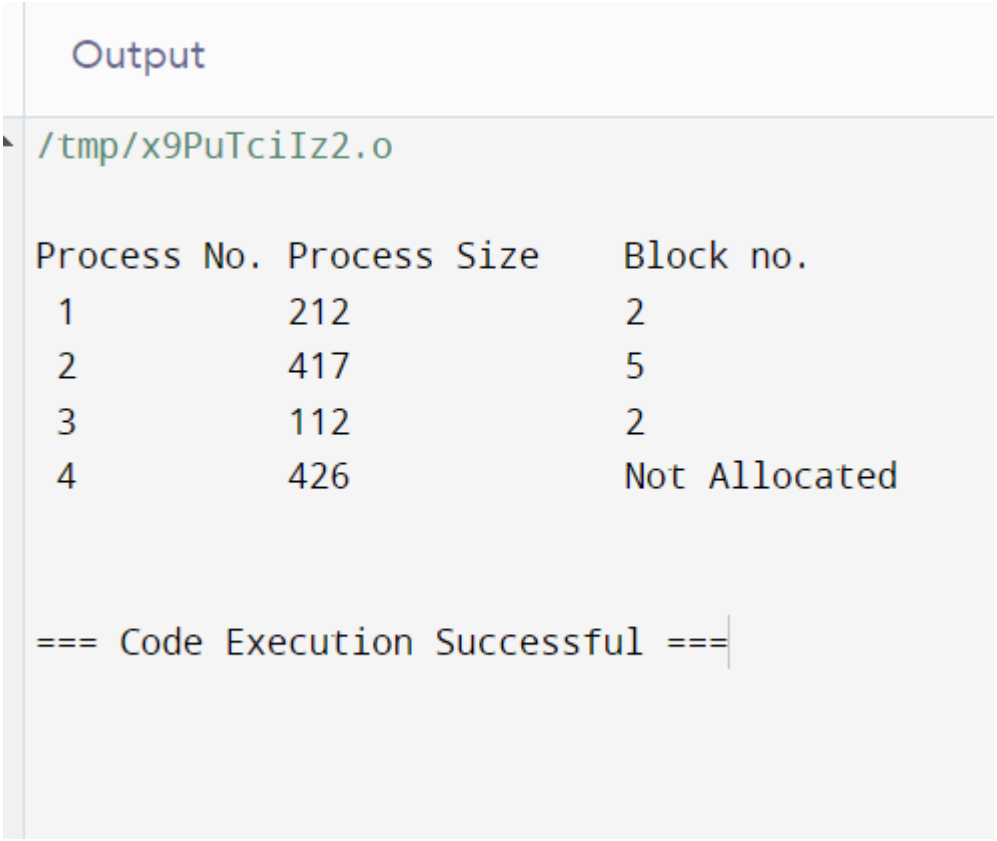
This algorithm was introduced long back and is difficult to implement because it requires future knowledge of the program behavior. However, it is possible to implement optimal page replacement on the second run by using the page reference information collected on the first run.

Program:

```
// First - Fit algorithm
#include<stdio.h>
void firstFit(int blockSize[], int m, int processSize[], int n)
{
	int i, j;
	int allocation[n];
	for(i = 0; i < n; i++)
	{
		allocation[i] = -1;
	}

	for (i = 0; i < n; i++)
	{
		for (j = 0; j < m; j++)
		{
			if (blockSize[j] >= processSize[i])
			{
				allocation[i] = j;
				blockSize[j] -= processSize[i];

				break;
			}
		}
	}

	printf("\nProcess No.\tProcess Size\tBlock no.\n");
	for (int i = 0; i < n; i++)
	{
		printf(" %i\t\t\t", i+1);
		printf("%i\t\t\t", processSize[i]);
		if (allocation[i] != -1)
			printf("%i", allocation[i] + 1);
		else
			printf("Not Allocated");
		printf("\n");
	}
}

int main()
{
	int m;
	int n;
	int blockSize[] = {100, 500, 200, 300, 600};
	int processSize[] = {212, 417, 112, 426};
	m = sizeof(blockSize) / sizeof(blockSize[0]);
	n = sizeof(processSize) / sizeof(processSize[0]);

	firstFit(blockSize, m, processSize, n);
```

```
    return 0 ;
}
```

Output:



Output

```
/tmp/x9PuTciIz2.o

Process No.  Process Size    Block no.
 1           212             2
 2           417             5
 3           112             2
 4           426             Not Allocated


=== Code Execution Successful ===
```

Conclusion:

Why do we need page replacement strategies ?

Page replacement strategies are essential in virtual memory management systems to efficiently utilize limited physical memory resources while providing the illusion of a larger memory space to user applications. The primary reason for employing page replacement strategies stems from the concept of virtual memory, which allows the operating system to use a portion of the disk as an extension of physical memory. However, physical memory is finite, and there are often more processes and data than can fit into it simultaneously.

Page replacement strategies come into play when a new page needs to be loaded into physical memory but there is no free space available. In such cases, the operating system must decide which page currently residing in physical memory to evict or replace with the new page. This decision impacts system performance, as it affects factors such as access time, efficiency of memory usage, and overall system throughput.

Effective page replacement strategies aim to minimize the number of page faults, which occur when a requested page is not found in physical memory and must be retrieved from secondary storage. By selecting the most appropriate page to replace based on certain criteria, such as frequency of use or time since last access, these strategies strive to optimize memory usage and maintain a balance between different processes' memory demands.

Additionally, page replacement strategies play a crucial role in supporting multitasking environments, where multiple processes compete for limited physical memory resources. Without efficient page replacement mechanisms, system performance could degrade significantly due to excessive paging activity, leading to increased response times, thrashing, and overall degradation of system performance.

In summary, page replacement strategies are indispensable components of virtual memory management systems, enabling efficient utilization of physical memory resources, supporting multitasking environments, and maintaining acceptable system performance levels. They are vital for achieving the delicate balance between maximizing memory usage and minimizing the overhead associated with managing memory.

Experiment No.10
File Management & I/O Management Implement disk scheduling algorithms FCFS, SSTF.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement disk scheduling algorithms FCFS.

Objective:

The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

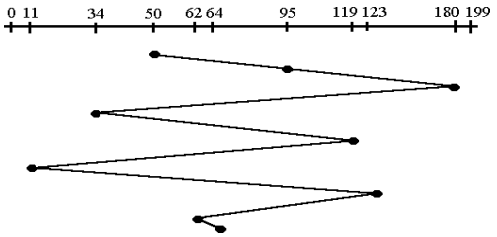
Theory:

TYPES OF DISK SCHEDULING ALGORITHMS

Although there are other algorithms that reduce the seek time of all requests, I will only concentrate on the following disk scheduling algorithms:

1. First Come-First Serve (FCFS)
2. Shortest Seek Time First (SSTF)
3. Elevator (SCAN)
4. Circular SCAN (C-SCAN)
5. C-LOOK

Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.



FCFS

First Come -First Serve (FCFS)

All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.

Program:

```
// FCFS Disk Scheduling algorithm
#include <stdio.h>
#include <math.h>

int size = 8;

void FCFS(int arr[],int head)
{
    int seek_count = 0;
    int cur_track, distance;

    for(int i=0;i<size;i++)
    {
        cur_track = arr[i];

        // calculate absolute distance
        distance = fabs(head - cur_track);
```

```
            seek_count += distance;

            head = cur_track;
        }

        printf("Total number of seek operations: %d\n",seek_count);

        printf("Seek Sequence is\n");

        for (int i = 0; i < size; i++) {
            printf("%d\n",arr[i]);
        }
}

int main()
{

        int arr[8] = { 176, 79, 34, 60, 92, 11, 41, 114 };
        int head = 50;

        FCFS(arr,head);

        return 0;
}
```

Output:

```
Output

/tmp/6aFc6bGU5R.o
Total number of seek operations: 510
Seek Sequence is
176
79
34
60
92
11
41
114


=== Code Execution Successful ===
```

Conclusion:

Why is Disk Scheduling important?

Disk scheduling is a critical component of modern operating systems, particularly in systems where multiple processes are contending for access to the disk. The primary purpose of disk scheduling algorithms is to optimize the order in which disk requests from different processes are serviced, with the goal of reducing disk seek time and maximizing disk throughput.

Disk seek time, the time it takes for the disk's read/write head to move to the appropriate track on the disk, is a significant contributor to overall disk access latency. By arranging disk requests in an efficient order, disk scheduling algorithms aim to minimize seek time and improve system responsiveness and performance.

Moreover, disk scheduling algorithms help in reducing disk fragmentation by organizing disk accesses in a more sequential manner, which can enhance disk I/O efficiency and prolong the lifespan of the storage device.

Different disk scheduling algorithms exist, each with its own set of trade-offs between factors such as fairness, throughput, response time, and starvation prevention. Common disk scheduling algorithms include First-Come-First-Served (FCFS), Shortest Seek Time First (SSTF), SCAN, C-SCAN, LOOK, and C-LOOK.

Overall, disk scheduling plays a vital role in optimizing disk access, improving system performance, and ensuring efficient utilization of disk resources in modern computing environments.