Academic Year: 2021-22 **Semester:** I

Class: TE IT **Subject:** Operating System Laboratory

Index

Sr. No.	Title of Assignment	Dates	Page No.
1	**A.** Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc. **B.** Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit		
2	Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states. **A.** Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. **B.** Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. The child process uses EXECVE system call to load new program which display array in reverse order.		
3	Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.		
4	**A.** Thread synchronization using counting semaphores. Application to demonstrate: producer- consumer problem with counting semaphores and mutex. **B.** Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader-Writer problem with reader priority.		
5	Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.		
6	Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.		
7	Inter process communication in Linux using following. **A.** FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output. **B.** Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.		
8	Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.		
9	Implement a new system call, add this new system call in the Linux kernel and demonstrate the use of same.		

Academic Year: 2021-22 **Semester:** I

Class: TE IT **Subject:** Operating System Laboratory

Assignment No. 1

A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

B. Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit.

Title: Shell programming

Aim: **A)** Study of Basic Linux Commands and **B)** Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit.

Objectives: To study basic Linux commands and write a program to implement address book for student.

Course Outcome (CO1): To understand the basics of Linux commands and program the shell of Linux.

Software Used: Ubuntu Operating System

Theory:

What is Unix?

The Unix operating system is a set of programs that act as a link between the computer and the user. The computer programs that allocate the system resources and coordinate all the details of the computer's internal is called the **operating system** or the **kernel**. Users communicate with the kernel through a program known as the **shell**. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

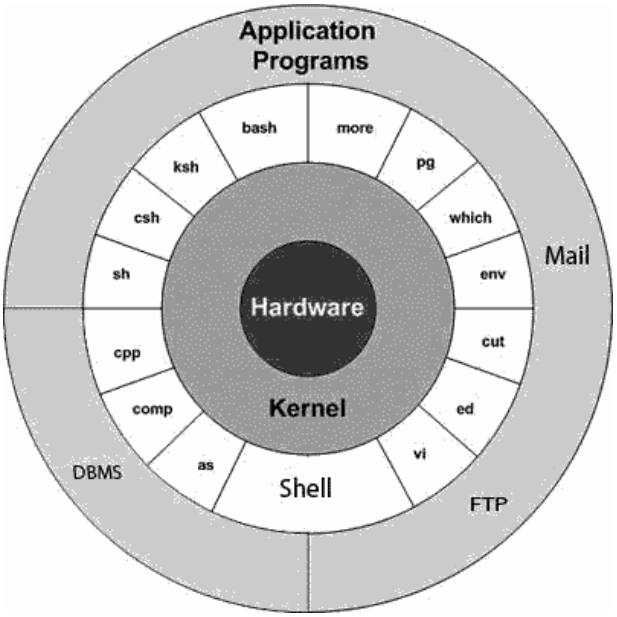
- Unix was originally developed in 1969 by a group of AT&T employees Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna at Bell Labs.
- There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are a few examples. Linux is also a flavor of Unix which is freely available.
- Several people can use a Unix computer at the same time; hence Unix is called a multiuser system.
- A user can also run multiple programs at the same time; hence Unix is a multitasking environment.

Unix Architecture

Here is a basic block diagram of a Unix system −

Academic Year: 2021-22 **Semester:** I

Class: TE IT **Subject:** Operating System Laboratory



The main concept that unites all the versions of Unix is the following four basics −

- **Kernel** − The kernel is the heart of the operating system. It interacts with the hardware and most of the tasks like memory management, task scheduling and file management.
- **Shell** − The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are the most famous shells which are available with most of the Unix variants.
- **Commands and Utilities** − There are various commands and utilities which you can make use of in your day to day activities. **cp**, **mv**, **cat**and **grep**, etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various options.
- **Files and Directories** − All the data of Unix is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the **filesystem**.

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time –

$date
Thu Jun 25 08:30:19 MST 2009

Shell Types

In Unix, there are two major types of shells −

- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell(sh)

Listing Files

To list the files and directories stored in the current directory, use the following command −

$ls

- Creating Files

You can use the **vi** editor to create ordinary files on any Unix system. You simply need to give the following command −

$ vi filename

Display Content of a File

You can use the **cat** command to see the content of a file. Following is a simple example to see the content of the above created file −

$ cat filename
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
$

Counting Words in a File

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is a simple example to see the information about the file created above −

$ wc filename
2 19 103 filename

Copying Files

To make a copy of a file use the **cp** command. The basic syntax of the command is −

$ cp source_file destination_file

Renaming Files

To change the name of a file, use the **mv** command. Following is the basic syntax −

$ mv old_file new_file

Deleting Files

To delete an existing file, use the **rm** command. Following is the basic syntax −

$rm filename

Unix / Linux - Shell Loop Types

A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

- The while loop , The for loop ,The until loop, The select loop

You will use different loops based on the situation. For example, the **while**loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

Unix / Linux - Shell Basic Operators

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators −

- Arithmetic Operators, Relational Operators, Boolean Operators ,String Operators Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.
Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]**is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.
Assume variable **a** holds 10 and variable **b** holds 20 then −
Show Examples

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.

-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Conclusion: We have studied how to implement address book using Shell programming.

Program:

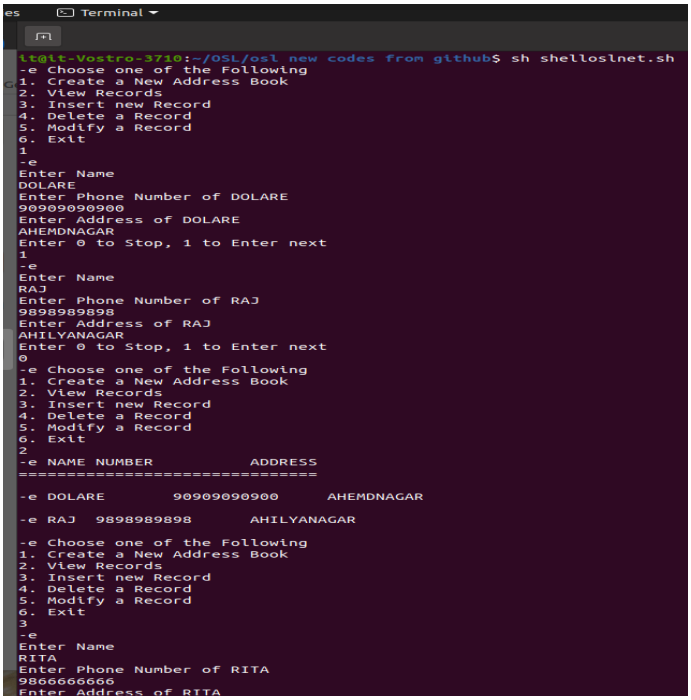
Program code:

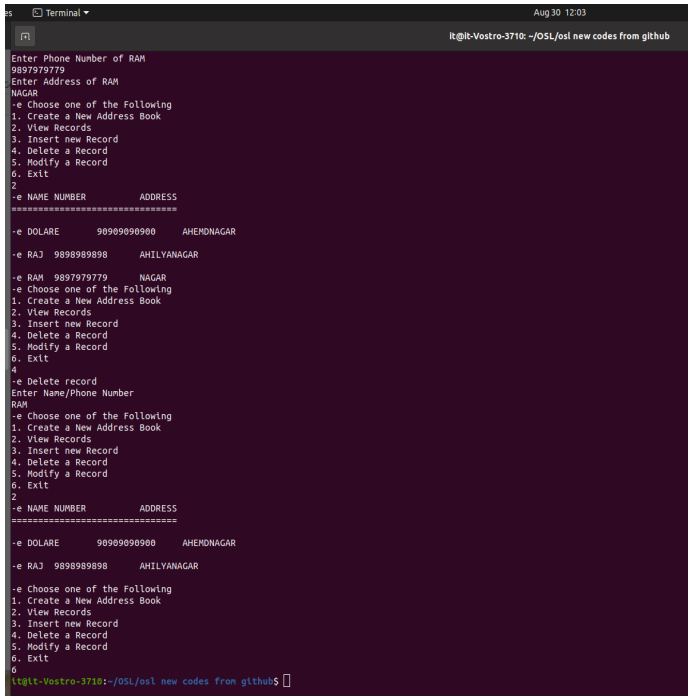
/*shelloslnet.sh */

```
fileName="MyAddressBook"
opt=1
while [ "$opt" -lt 6 ]
do

echo -e "Choose one of the Following\n1. Create a New Address Book\n2. View Records\n3. Insert
new Record\n4. Delete a Record\n5. Modify a Record\n6. Exit"
# echo -e, enables special features of echo to use \n \t \b etc.
read opt
case $opt in
1)
if [ -e $fileName ] ; then# -e to check if file exists, if exits remove the file
        rm $fileName
        fi
        cont=1
    echo -e "NAME\tNUMBER\t\tADDRESS\==========\n" | cat >> $fileName
        while [ "$cont" -gt 0 ]
        do
                echo -e "\nEnter Name"
                read name
                echo "Enter Phone Number of $name"
                read number
                echo "Enter Address of $name"
                read address
                echo -e "$name\t$number\t$address\n" | cat >> $fileName
                echo "Enter 0 to Stop, 1 to Enter next"
                read cont
        done
        ;;

2)
        cat $fileName
```

```
        ;;
3)
        echo -e "\nEnter Name"
        read name
        echo "Enter Phone Number of $name"
        read number
        echo "Enter Address of $name"
        read address
        echo -e "$name\t$number\t\t$address" | cat >> $fileName
        ;;
4)
        echo -e "Delete record\nEnter Name/Phone Number"
        read pattern
        temp="temp"
grep -v $pattern $fileName | cat >> $temp # grep -v selects non-matching lines
        rm $fileName
        cat $temp | cat >> $fileName
        rm $temp
        ;;
5)
        echo -e "Modify record\nEnter Name/Phone Number"
        read pattern
        temp="temp"
        grep -v $pattern $fileName | cat >> $temp
        rm $fileName
        cat $temp | cat >> $fileName
        rm $temp
        echo "Enter Name"
        read name
        echo "Enter Phone Number of $name"
        read number
        echo "Enter Address of $name"
        read address
        echo -e "$name\t$number\t$address" | cat >> $fileName
        ;;
esac
done
```

Output :





Assignment No. 2
Part -A

Academic Year: 2020-21 **Semester:** I
Class : TE IT **Subject:** System Laboratory II (314447)

Title: Process control system calls

Aim: Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

Objectives: To study and implement fork() system call using sorting algorithm.Also study zombie and orphan states.

Course Outcome (CO2): To develop various system programs for the functioning of operating system.

Theory:

fork( ):

It is a system call that creates a new process under the UNIX operating system. It takes no arguments. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- 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.

Therefore, after the system call to fork(), a simple test can tell which process is the child. Note that Unix will make an exact copy of the parent's address space and give it to the child.
Therefore, the parent and child processes have separate address spaces.

Let us take an example:

```
int main()
{
printf("Before Forking");
fork();
printf("After Forking");
return 0;
}
```

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.
- Both processes will start their execution at the next statement following the fork() call.

If we run this program, we might see the following on the screen:

- Before Forking

- After Forking
- After Forking

Here printf() statement after fork() system call executed by parent as well as child process. Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space.

Other address spaces created by fork() calls will not be affected even though they have identical variable names.

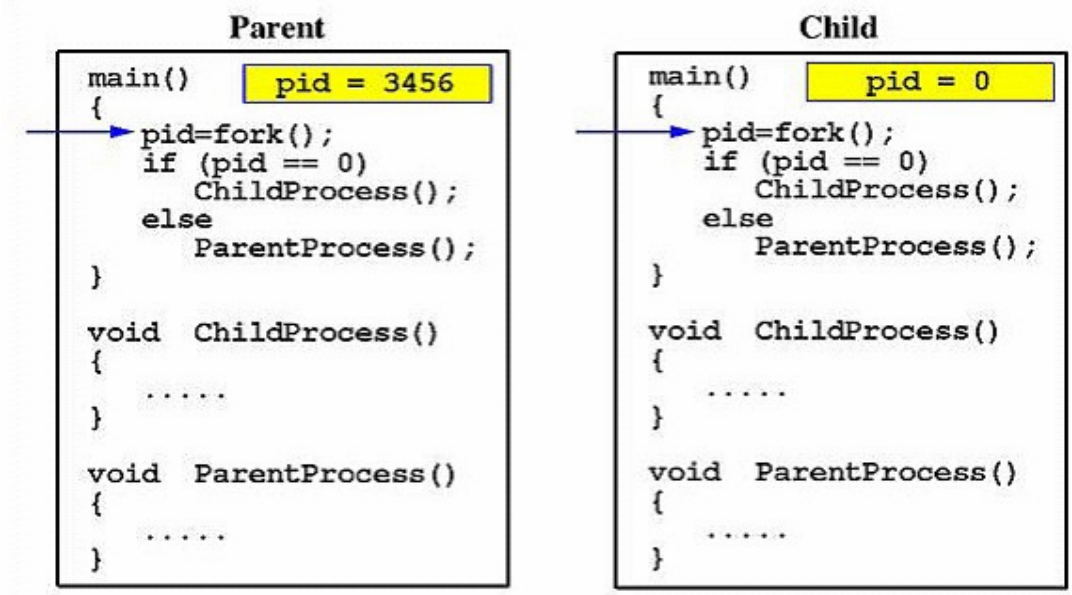
Consider one simpler example, which distinguishes the parent from the child.

```
#include <stdio.h>
#include <sys/types.h>
void ChildProcess(); /* child process prototype */
void ParentProcess(); /* parent process prototype */
int main()
{
pid_t     pid;
pid = fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
return 0;
}
void ChildProcess()
{
}
void ParentProcess()
{
}
```

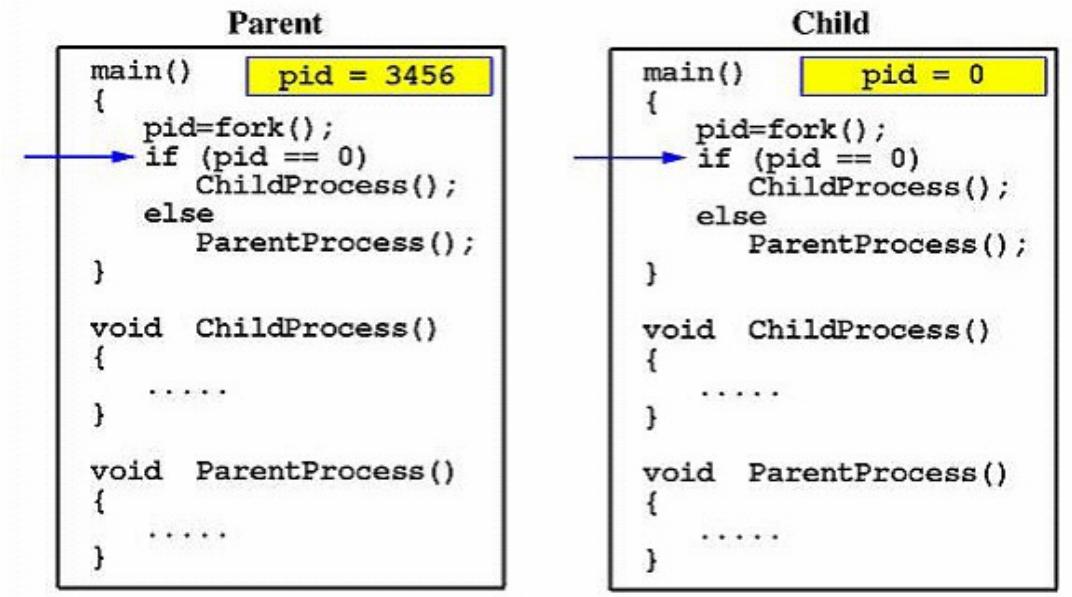
In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable i.

When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent

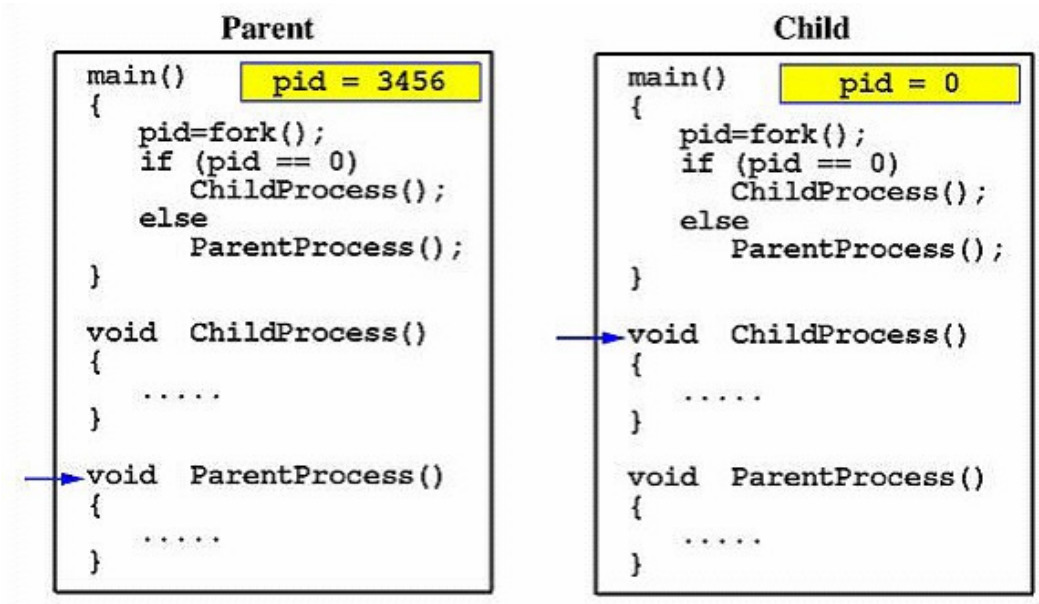
and returns 0 to the child process. The following figure shows that in both address spaces there is a variable pid. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the parent, since pid is non-zero, it calls function parentprocess(). On the other hand, the child has a zero pid and calls childprocess() as shown below:



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process.

ps command:
The ps command shows the processes we're running, the process another user is running, or all the processes on the system.
E.g. **$ ps –ef**

By default, the ps program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. We can use **ps** to see all such processes using the -e option and to get "full" information with -f.

Conclusion: We have studied Process control system calls fork().

Academic Year: 2020-21 **Semester:** I

Class : TE IT **Subject:** System Laboratory II (314447)

Program:

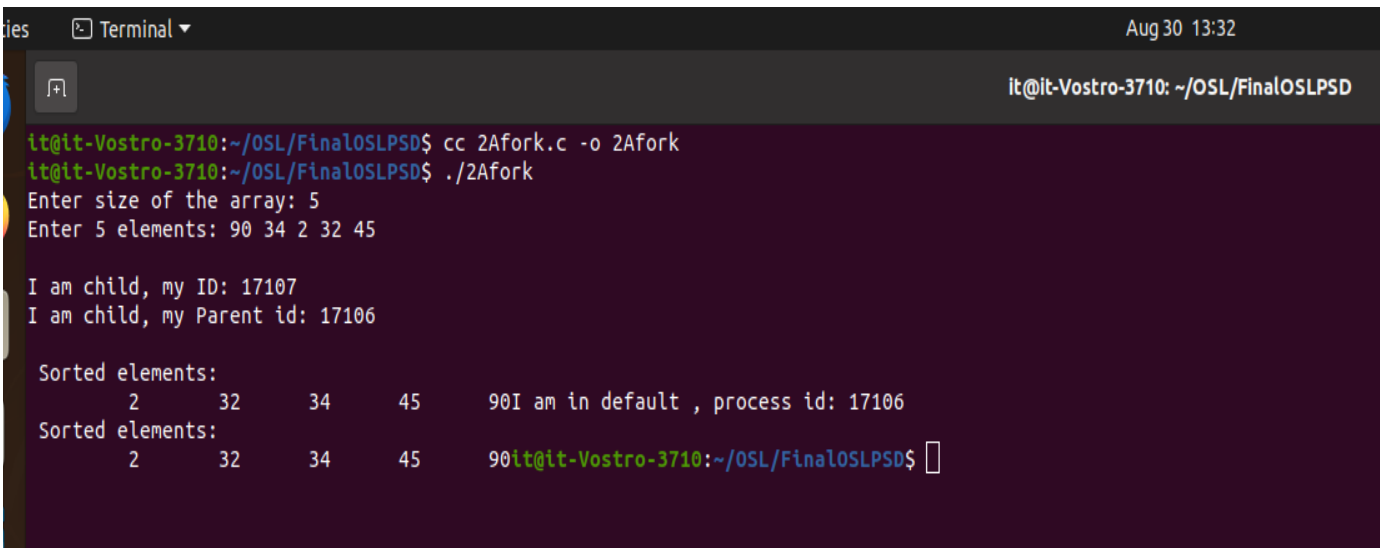
Program code:

```
/*2Afork.c */
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
void quicksort(int a[],int,int);
void merge(int a[], int low, int mid, int high);
void divide(int a[], int low, int high);
int main()
{
int a[20],n,i;
pid_t pid;
printf("Enter size of the array: ");
scanf("%d",&n);
printf("Enter %d elements: ",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
pid = fork();
switch(pid)
{
case 0:
printf ("I am child, my ID: %d", getpid() );
printf ("\nI am child, my Parent id: %d \n",getppid() );
quicksort(a,0,n-1);
break;
case -1:
printf ("The child process has not created");
break;
default:
printf ("\nI am in default , process id: %d ",getpid());
divide( a, 0, n-1);
sleep(3);
break;
} // switch case closed
printf("\n Sorted elements:\n ");
for(i=0;i<n;i++)
printf(" \t %d",a[i]);
return 0;
}
void divide(int a[], int low, int high)
{
if(low<high) // The array has atleast 2 elements
```

```
{
int mid = (low+high)/2;
divide(a, low, mid); // Recursion chain to sort first half of the array
divide(a, mid+1, high); // Recursion chain to sort second half of the array
merge(a, low, mid, high);
}
}
void merge(int a[], int low, int mid, int high)
{
int i, j, k, m = mid-low+1, n = high-mid;
int first_half[m], second_half[n];
for(i=0; i<m; i++) // Extract first half (already sorted)
first_half[i] = a[low+i];
for(i=0; i<n; i++) // Extract second half (already sorted)
second_half[i] = a[mid+i+1];
i = j = 0;
k = low;
while(i<m || j<n) // Merge the two halves
{
if(i >= m)
{
a[k++] = second_half[j++];
continue;
}
if(j >= n)
{
a[k++] = first_half[i++];
continue;
}
if(first_half[i] < second_half[j])
a[k++] = first_half[i++];
else
a[k++] = second_half[j++];
}
}
void quicksort(int a[],int first,int last)
{
int pivot,j,temp,i;
if(first<last){
pivot=first;
i=first;
j=last;
while(i<j){
```

```
while(a[i]<=a[pivot]&&i<last)
i++;
while(a[j]>a[pivot])
j--;
if(i<j){
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
temp=a[pivot];
a[pivot]=a[j];
a[j]=temp;
quicksort(a,first,j-1);
quicksort(a,j+1,last);
}
}
```

Output :



Academic Year: 2020-21 **Semester:** I
Class : TE IT **Subject:** System Laboratory II (314447)

Assignment No.2
Part -B

Title: Process control system calls:

Aim: Implement the C program in which main program accepts an integer array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

Objective: To study and implement execv() system call using sorting and searching algorithm.

Course Outcome (CO2): To develop various system programs for the functioning of operating system.

Theory:
exec() system call:
The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program. The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways. Within the exec family there are functions that vary slightly in their capabilities.
exec family:
1. **execl() and execlp():**

execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL. e.g.
execl("/bin/ls", "ls", "-l", NULL);

execlp(): It does same job except that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. The function execlp() can also take the fully qualified name as it also resolves explicitly. e.g.execlp("ls", "ls", "-l", NULL);

2. execv() and execvp():
execv(): It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string.
e.g.char *argv[] = ("ls", "-l", NULL);
execv("/bin/ls", argv);
execvp(): It does same job expect that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used. e.g.execvp("ls", argv);

3. execve( ):

int execve(const char *filename, char *const argv[ ], char *const envp[ ]);

It executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form:

argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both argv and envp must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

int main(int argc, char *argv[ ] , char *envp[ ])]

execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

The wait() system call:

It blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.The execution of wait() could have two possible situations.

- If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
- If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Zombie Process:

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.

Orphan Process:

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under UNIX, the latter kinds of processes are typically called daemon processes. The UNIX *nohup* command is one means to accomplish this.

Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

Conclusion: We have studied Process control system calls execv().

Program:

(This asssignment needs 2 program for proper demonstartion parent and child.)

/*2Bparent.c*/

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

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
   int n;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   char *args[n + 2];
   args[0] = "./2Bchild"; // Program to execute
   args[n + 1] = NULL; // Null terminate the arguments
   printf("Enter the elements of the array:\n");
   for (int i = 1; i <= n; i++) {
      args[i] = (char *)malloc(10 * sizeof(char));
      scanf("%s", args[i]);
   }
   pid_t pid = fork();
   if (pid < 0) {
      perror("Fork failed");
      exit(EXIT_FAILURE);
   }
   if (pid == 0) { // Child process
      execve(args[0], args, NULL);
      perror("Execve failed");
      exit(EXIT_FAILURE);
   } else { // Parent process
      wait(NULL);
      printf("Parent process: Child process has completed.\n");
      for (int i = 1; i <= n; i++) {
         free(args[i]);
      }   }
   return 0;       }
```

(Save both parent and child separate and run by giving path of child program in parent like shown in above program in bold line.)

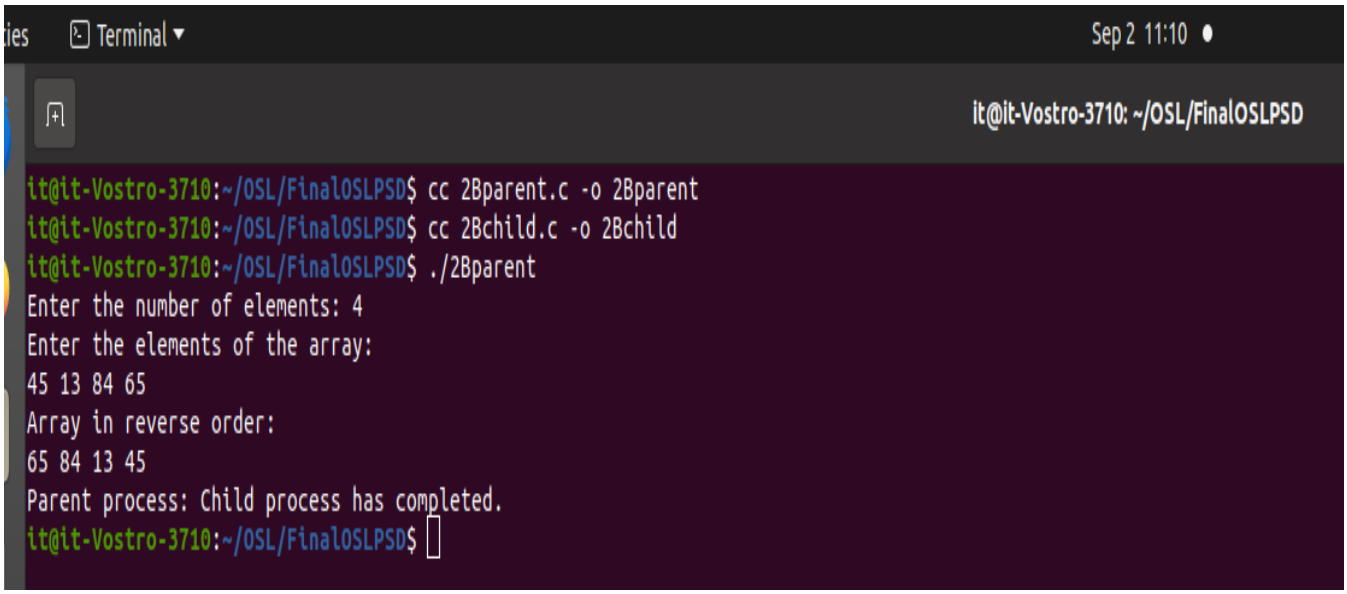
/*2Bchild.c*/

```
#include <stdio.h>
int main(int argc, char *argv[]) {
   printf("Array in reverse order:\n");
   for (int i = argc - 1; i > 0; i--) {
      printf("%s ", argv[i]);
   }
   printf("\n");
   return 0;       }
```

Academic Year: 2020-21 **Semester:** I

Class : TE IT **Subject:** System Laboratory II (314447)

Output :



Academic Year: 2020-21 **Semester:** I

Class : TE IT **Subject:** System Laboratory II (314447)

Assignment No. 3

Title: CPU Scheduling Algorithms

Aim: Implement the C program for CPU Scheduling Algorithms: Shortest Job First

(Preemptive) and Round Robin with different arrival time.

Objective: To demonstrate the functioning of OS concept in user space like CPU Scheduling.

Course Outcome (CO2): Develop various system programs for the functioning of OS concepts in user space like concurrency, control, CPU Scheduling….

Theory:

Round Robin Scheduling

Introduction: Round Robin (RR) is a preemptive CPU scheduling algorithm that assigns a fixed time slice or quantum to each process in the ready queue. It is designed to provide a fair allocation of CPU time among processes, especially in time-sharing systems.

Working:

- **Time Quantum:** A fixed amount of time is allocated to each process. After this time expires, the process is preempted and placed at the end of the ready queue.
- **Cyclic Order:** Processes are scheduled in a cyclic order, ensuring that each process gets an equal opportunity to execute.

Steps:

1. **Arrival of Processes:** Processes arrive at different times with specific burst times.
2. **Time Quantum Allocation:** Each process gets a fixed time slice (quantum) for execution.
3. **Execution:** The process at the front of the ready queue is executed for the given time quantum.
4. **Preemption:** If the process does not complete within the time quantum, it is preempted and placed at the end of the ready queue.
5. **Completion:** The process continues to cycle through the queue until it completes.

Example: Consider the following processes with a time quantum of 3 units:

Process	Arrival Time	Burst Time
P1	0	5
P2	1	4
P3	2	6
P4	3	2

Execution Order: P1 (0-3), P2 (3-6), P3 (6-9), P4 (9-11), P1 (11-12), P3 (12-15), P2 (15-16), P3 (16-18)

Shortest Job First (Preemptive) Scheduling

Introduction:

Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time for execution. The preemptive version of this algorithm is also known as **Shortest Remaining Time First (SRTF)**. It is designed to minimize the waiting time for processes.

Working:

- **Preemptive Nature:** SJF (Preemptive) continuously monitors the arrival of new processes. If a new process arrives with a burst time shorter than the remaining time of the currently executing process, the CPU will preempt the current process and assign the CPU to the new process.
- **Decision Point:** At every new arrival or completion of a process, the CPU checks if the new process has a burst time shorter than the remaining burst time of the current process. If yes, the current process is preempted.

Steps:

1. **Arrival of Processes:** Processes arrive at different times with specific burst times.
2. **Scheduling Decision:** At each time unit, the scheduler checks if any process has arrived with a shorter burst time than the currently executing process.
3. **Preemption:** If a new process has a shorter burst time, it preempts the currently executing process.
4. **Execution:** The process with the shortest burst time is executed next.
5. **Completion:** Once a process completes, the CPU looks for the next shortest burst time process among the ready queue and the newly arrived processes.

Example:

Consider the following processes:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- **Execution Order:** P1 (0-1), P2 (1-5), P4 (5-10), P1 (10-14), P3 (14-23)

Conclusion: We have implemented CPU Scheduling Algorithms.

Program:

Program code:

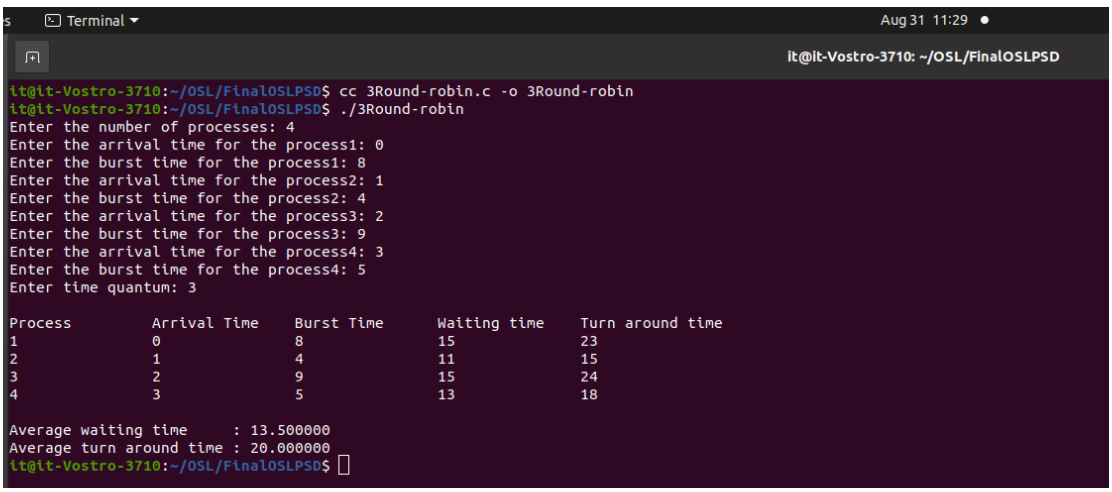
```
/*3Round-robin.c*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define N 100
struct process
{
```

```
int process_id;
int arrival_time;
int burst_time;
int waiting_time;
int turn_around_time;
int remaining_time;
};
int queue[N];
int front = 0, rear = 0;
struct process proc[N];
void push(int process_id)
{
queue[rear] = process_id;
rear = (rear+1)%N;
}

int pop()
{
if(front == rear)
        return -1;

int return_position = queue[front];
front = (front +1)%N;
return return_position;
}

int main()
{
float wait_time_total = 0.0, tat = 0.0;
int n,time_quantum;
printf("Enter the number of processes: ");
scanf("%d", &n);

for(int i=0; i<n; i++)
{
        printf("Enter the arrival time for the process%d: ",i+1);
        scanf("%d", &proc[i].arrival_time);
        printf("Enter the burst time for the process%d: ",i+1);
        scanf("%d", &proc[i].burst_time);
        proc[i].process_id = i+1;
        proc[i].remaining_time = proc[i].burst_time;
}
printf("Enter time quantum: ");
```

```
scanf("%d",&time_quantum);

int time=0;
int processes_left=n;
int position=-1;
int local_time=0;

for(int j=0; j<n; j++)
        if(proc[j].arrival_time == time)
                push(j);

while(processes_left)
{
        if(local_time == 0)
        {
                if(position != -1)
                        push(position);

                position = pop();
        }

        for(int i=0; i<n; i++)
        {
                if(proc[i].arrival_time > time)
                        continue;
                if(i==position)
                        continue;
                if(proc[i].remaining_time == 0)
                        continue;

                proc[i].waiting_time++;
                proc[i].turn_around_time++;
        }
        if(position != -1)
        {
                proc[position].remaining_time--;
                proc[position].turn_around_time++;

                if(proc[position].remaining_time == 0)
                {
                        processes_left--;
                        local_time = -1;
                        position = -1;
                }
```

```
        }
        else
                local_time = -1;

        time++;
        local_time = (local_time +1)%time_quantum;
        for(int j=0; j<n; j++)
                if(proc[j].arrival_time == time)
                        push(j);
    }
    printf("\n");
    printf("Process\t\tArrival Time\tBurst Time\tWaiting time\tTurn around time\n");
    for(int i=0; i<n; i++)
    {
            printf("%d\t\t%d\t\t", proc[i].process_id, proc[i].arrival_time);
    printf("%d\t\t%d\t\t%d\n", proc[i].burst_time, proc[i].waiting_time, proc[i].turn_around_time);

            tat += proc[i].turn_around_time;
            wait_time_total += proc[i].waiting_time;
    }
    tat = tat/(1.0*n);
    wait_time_total = wait_time_total/(1.0*n);

    printf("\nAverage waiting time     : %f",wait_time_total);
    printf("\nAverage turn around time : %f\n", tat);
    }
```

Output :

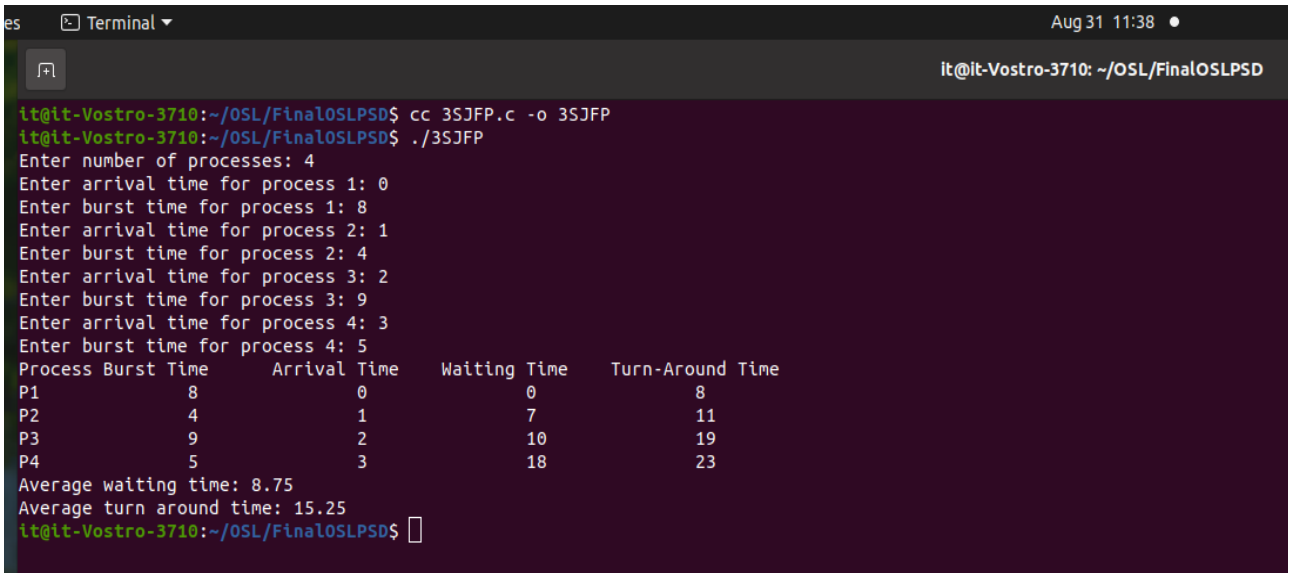


/* 3SJFP.c */

```
#include <stdio.h>
struct Process {
   int id;
   int arrivalTime;
   int burstTime;
   int waitingTime;
   int turnAroundTime;
};
void calculateTimes(struct Process proc[], int n) {
   int totalWaitingTime = 0, totalTurnAroundTime = 0;
   int completionTime[n];
   // Sort the processes by Arrival Time and Burst Time
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (proc[i].arrivalTime > proc[j].arrivalTime ||
          (proc[i].arrivalTime == proc[j].arrivalTime && proc[i].burstTime > proc[j].burstTime)) {
            struct Process temp = proc[i];
            proc[i] = proc[j];
            proc[j] = temp;
         }
      }
   }
   // Initialize the completion time of the first process
   completionTime[0] = proc[0].arrivalTime + proc[0].burstTime;
   proc[0].turnAroundTime = proc[0].burstTime;
   proc[0].waitingTime = 0;
   // Calculate waiting time and turn-around time for each process
   for (int i = 1; i < n; i++) {
      // Calculate completion time for this process
      completionTime[i] = completionTime[i-1] + proc[i].burstTime;
      // Turn Around Time = Completion Time - Arrival Time
      proc[i].turnAroundTime = completionTime[i] - proc[i].arrivalTime;
      // Waiting Time = Turn Around Time - Burst Time
      proc[i].waitingTime = proc[i].turnAroundTime - proc[i].burstTime;
   }
   // Display results
   printf("Process\tBurst Time\tArrival Time\tWaiting Time\tTurn-Around Time\n");
   for (int i = 0; i < n; i++) {
printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].id, proc[i].burstTime, proc[i].arrivalTime,
proc[i].waitingTime, proc[i].turnAroundTime);
      totalWaitingTime += proc[i].waitingTime;
      totalTurnAroundTime += proc[i].turnAroundTime;
   }
```

```
    printf("Average waiting time: %.2f\n", (float)totalWaitingTime / n);
    printf("Average turn around time: %.2f\n", (float)totalTurnAroundTime / n);
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    // Input arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
        proc[i].id = i + 1;
        printf("Enter arrival time for process %d: ", proc[i].id);
        scanf("%d", &proc[i].arrivalTime);
        printf("Enter burst time for process %d: ", proc[i].id);
        scanf("%d", &proc[i].burstTime);
    }
    // Calculate and display the scheduling results
    calculateTimes(proc, n);
    return 0;
}
```

Output :



Assignment No. 4
Part A

Title: Producer-Consumer Problem (Using semaphores)

Aim: Application to demonstrate: producer-consumer problem with counting semaphores and mutex.

Objective: To study the concept of semaphore and mutex by implementing producer-

consumer problem.

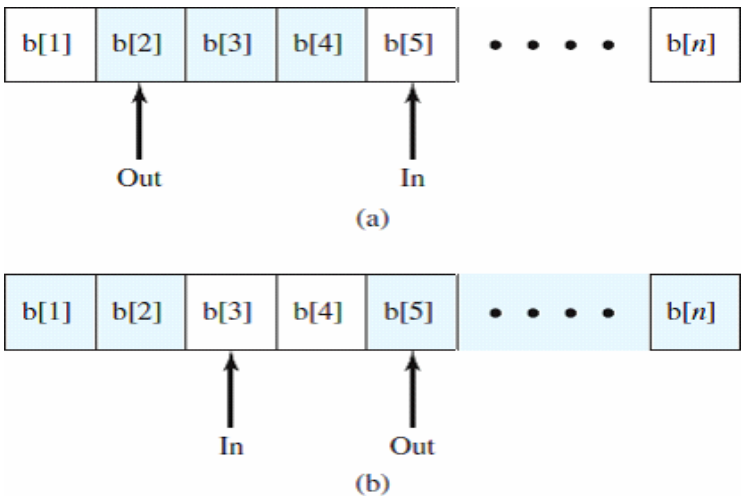
Course Outcome (CO3): To implement basic building blocks like processes, threads under the Linux.

Theory:

The Producer/Consumer Problem

The general statement is this: there are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won't try to add data in to the buffer if it's full and that the consumer won't try to remove data from an empty buffer. To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

Solution to bounded/circular buffer producer-consumer problem



producer:

```
while (true) {
/* produce item v */;
b[in] = v; /*
in++;
}
```

consumer:

```
while (true) {
 while (in <= out)
 do nothing */;
        w = b[out];
        out++;
        /* consume item w */;
        }
```

Solution using Semaphore (Unbounded Buffer).

```
semaphore s = 1, n = 0;
void producer()
{
while (true)
{
produce();
```

```
semWait(s);
append();
semSignal(s);
semSignal(n);
}
}
void consumer()
{
while (true)
{
semWait(n);
semWait(s);
take();
semSignal(s);
consume();
}
}
void main()
{
parbegin (producer, consumer);
}
```

Semaphore

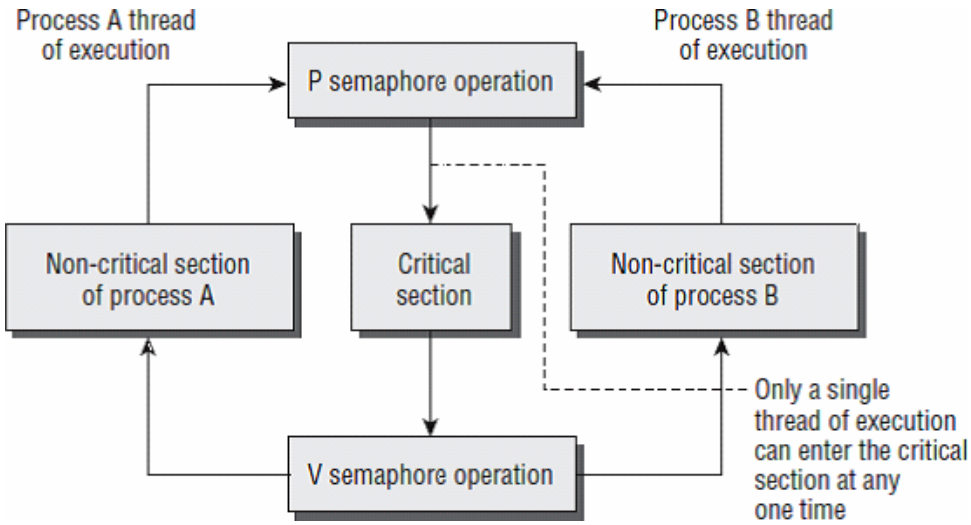
It is a special type of variable containing integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process, also known as a counting semaphore or a general semaphore.

A semaphore may be initialized to a nonnegative integer value. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution. The semSignal operation

increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Generalized use of semaphore for forcing critical section

```
semaphore sv = 1;
loop forever
{
Wait(sv);
critical code
section; signal(sv);
noncritical code section;
}
```



Linux Semaphore Facilities (Binary Semaphore)

A semaphore is created with the sem_init function, which is declared as follows:

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);

This function initializes a semaphore object pointed to by *sem*, sets its sharing option and gives it an initial integer value. The *pshared* parameter controls the type of semaphore. If the value of *pshared* is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn't support this sharing, and passing a nonzero value for *pshared* will cause the call to fail.

The next pair of functions controls the value of the semaphore and is declared as follows:

#include <semaphore.h>
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);

These both take a pointer to the semaphore object initialized by a call to sem_init. The sem_post function atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic "test and set" ability in a single function is what makes semaphores so valuable.

The last semaphore function is sem_destroy. This function tidies up the semaphore when we have finished with it. It is declared as follows:

#include <semaphore.h>
int sem_destroy(sem_t * sem);

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If we attempt to destroy a semaphore for which some thread is waiting, we will get an error. Like most Linux functions, these functions all return 0 on success.

Conclusion: We have studied concept of semaphore and mutex.

Program:

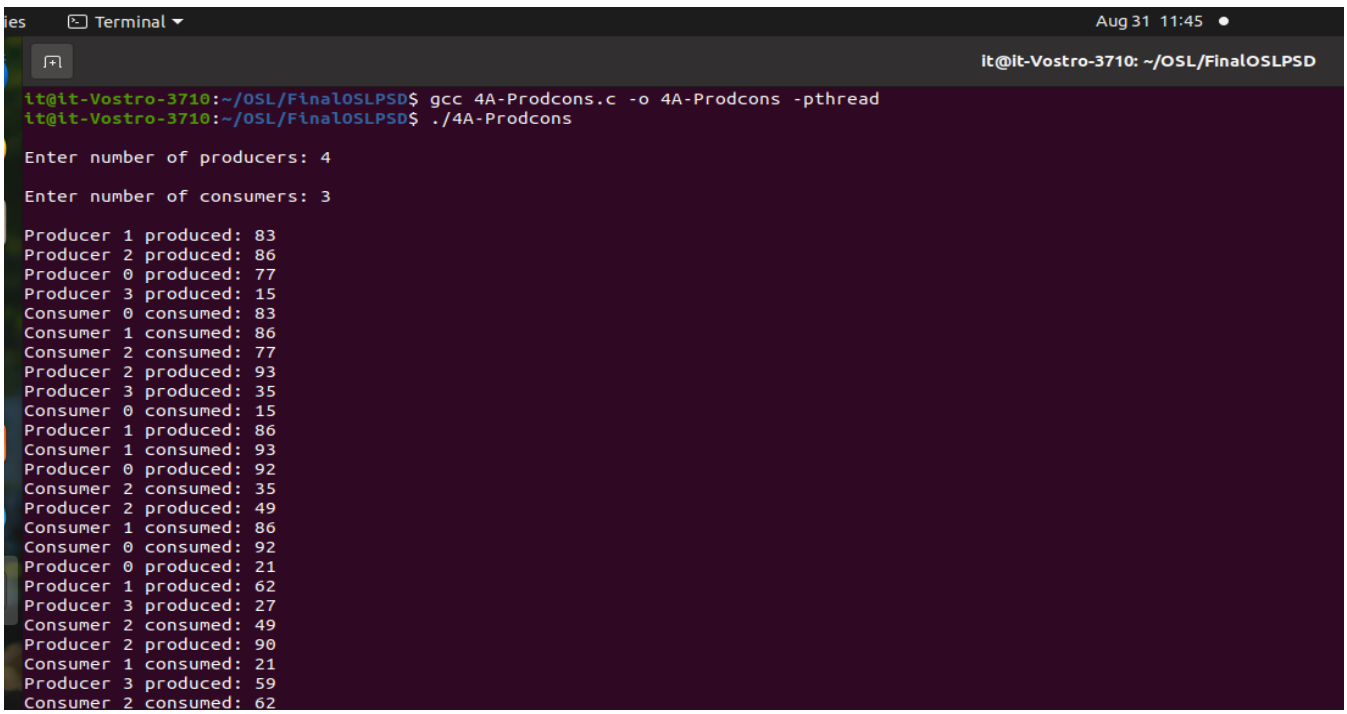
Program Code:

```
/* 4A-Prodcons.c*/
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>
void *producer(void *thread);
void *consumer(void *thread);
int count = 0, in = 0, out = 0, a[5];
sem_t full;
sem_t empty;
pthread_mutex_t mutex;
int main() {
```

```
    int i, p, c;
    pthread_t pid[10], cid[10];
    pthread_mutex_init(&mutex, NULL);
    sem_init (&full, 0, 0);
    sem_init (&empty, 0, 5);
    printf("\nEnter number of producers: ");
    scanf("%d", &p);
    printf("\nEnter number of consumers: ");
    scanf("%d", &c);
    int producer_indices[p];
    int consumer_indices[c];
    for (i = 0; i < p; i++) {
      producer_indices[i] = i;
      pthread_create(&pid[i], NULL, producer, &producer_indices[i]);
    }
    for (i = 0; i < c; i++) {
      consumer_indices[i] = i;
      pthread_create(&cid[i], NULL, consumer, &consumer_indices[i]);
    }
     for (i = 0; i < p; i++) {
      pthread_join(pid[i], NULL);
    }
    for (i = 0; i < c; i++) {
      pthread_join(cid[i], NULL);
    }
    sem_destroy(&full);
    sem_destroy(&empty);
    pthread_mutex_destroy(&mutex);
     return 0;
}
void *producer(void *thread) {
    int t = *(int *)thread;
    while (1) {
      sem_wait(&empty);
      pthread_mutex_lock(&mutex);
      if (count >= 5) {
        printf("\nBuffer is full");
      } else {
        a[in] = rand() % 100;
        printf("\nProducer %d produced: %d", t, a[in]);
        in = (in + 1) % 5;
        count++;
      }
```

```
            pthread_mutex_unlock(&mutex);
            sem_post(&full);
            sleep(1);
        }
        pthread_exit(0);
    }
    void *consumer(void *thread) {
        int t = *(int *)thread;
        while (1) {
            sem_wait(&full);
            pthread_mutex_lock(&mutex);
            if (count <= 0) {
            printf("\nBuffer is empty");
            } else {
                printf("\nConsumer %d consumed: %d", t, a[out]);
                out = (out + 1) % 5;
                count--;
            }
            pthread_mutex_unlock(&mutex);
            sem_post(&empty);
            sleep(1);
        }
        pthread_exit(0);
    }
```

Output: **Execution will be infinite; we have to stop it. Use control +F to stop execution**



Academic Year: 2020-21 **Semester:** I
Class : TE IT **Subject:** System Laboratory II (314447)

Assignment No 4
Part B

Title: Thread Synchronization and mutual exclusion using mutex.

Aim: Application to demonstrate Reader-Writer Problem with reader priority.

Objective: To study the concept of thread synchronization and mutual exclusion by implementing reader-writer problem.

Course Outcome (CO3): To implement basic building blocks like processes, threads under the Linux.

Theory: Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same

time. Precisely in OS we call this situation as the **readers-writers problem**

Problem parameters:

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution when Reader has the Priority over Writer

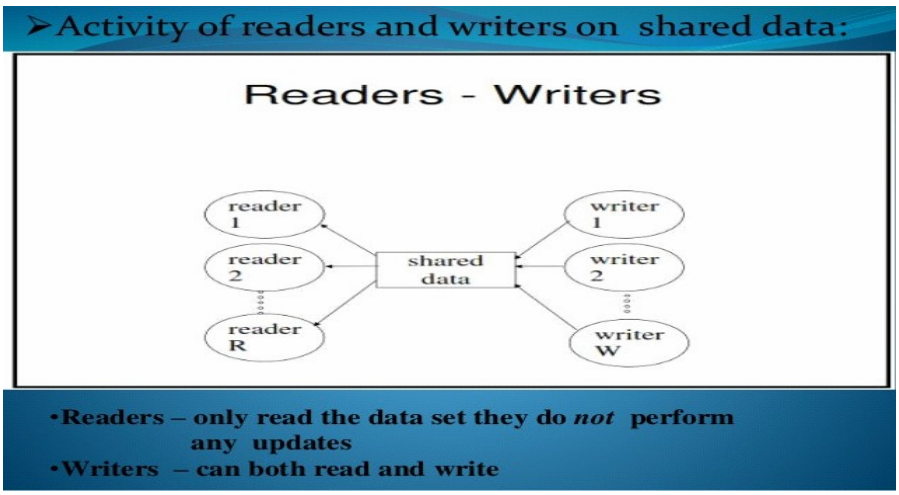
Here priority means, no reader should wait if the share is currently opened for reading.

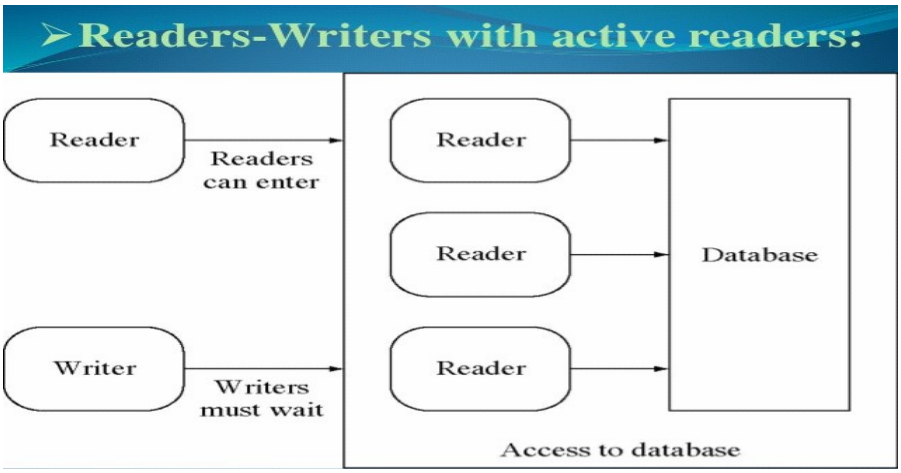
Reader process:

- Reader requests the entry to critical section.
- If allowed:
- it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
- It then, signals mutex as any other reader is allowed to enter while others are already reading.
- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.
- If not allowed, it keeps on waiting.

Writer process:

- Writer requests the entry to critical section.
- If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
- It exits the critical section.





Conclusion: We have studied reader writer problem with readers priority using mutex.

Program Code:

```
/*4B-Readers-writers.c */
#include "stdio.h"
#include "string.h"
#include "pthread.h"
#include "semaphore.h"
#include "stdlib.h"
```

```
#include "unistd.h"
#define BUFFER_SIZE 16
int buffer[BUFFER_SIZE];
sem_t database,mutex;
int counter, readerCount;
pthread_t readerThread[50],writerThread[50];
void init()
{
sem_init(&mutex,0,1);
sem_init(&database,0,1);
counter=0;
readerCount=0;
}

void *writer(void *param)
{
sem_wait(&database);
int item;
item=rand()%5;
buffer[counter]=item;
printf("Data writen by the writer%d is %d\n",  (*(int *)param), buffer[counter]);
counter++;
   sleep(1);
   sem_post(&database);
}
void *reader(void *param)
{
sem_wait(&mutex);
readerCount++;
if(readerCount==1)
{
        sem_wait(&database);
}
sem_post(&mutex);
counter--;
printf("Data read by the reader%d is %d\n",  (*(int *) param), buffer[counter]);
   sleep(1);

   sem_wait(&mutex);
   readerCount--;
   if(readerCount==0)
   {
        sem_post(&database);
```

```
    }
    sem_post(&mutex);
}
int main()
{
init();
int no_of_writers,no_of_readers;
printf("Enter number of readers: ");
scanf("%d",&no_of_readers);
printf("Enter number of writers: ");
scanf("%d",&no_of_writers);
int i;
for(i=0;i<no_of_writers;i++)
{
        pthread_create(&writerThread[i],NULL,writer,&i);
}
for(i=0;i<no_of_readers;i++)
{
        pthread_create(&readerThread[i],NULL,reader, &i);
}
for(i=0;i<no_of_writers;i++)
{
        pthread_join(writerThread[i],NULL);
}
for(i=0;i<no_of_readers;i++)
{
        pthread_join(readerThread[i],NULL);
}}
```

Output :

Academic Year: 2020-21 **Semester:** I
Class : TE IT **Subject:** System Laboratory II (314447)

Assignment No. 5

Title: Deadlock Avoidance Algorithm

Aim: Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

Objective: To demonstrate the functioning of OS concepts in user space like concurrency control (process synchronization, mutual exclusion), CPU Scheduling, Memory Management and Disk Scheduling in LINUX.

Course Outcome (CO3): To develop various system programs for the functioning of OS concepts in user space like concurrency control, CPU Scheduling, Memory Management and Disk Scheduling in Linux.

Theory:

Introduction

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource held by another process. The Banker's Algorithm, developed by Edsger Dijkstra, is a resource allocation and deadlock avoidance algorithm. It is used to ensure that the system remains in a safe state after the allocation of resources.

Definitions

- **Processes and Resources**: In a multiprogramming environment, multiple processes require access to various resources. Resources can be of different types, such as CPU cycles, memory, I/O devices, etc.
- **Safe State**: A state is considered safe if there exists a sequence of processes such that for each process, the resources it still needs can be satisfied by the currently available resources plus the resources held by all the previous processes in the sequence.
- **Unsafe State**: A state is unsafe if it is not safe, meaning that allocating resources could lead to a situation where deadlock might occur.
- **Allocation Matrix (A)**: This matrix represents the number of resources currently allocated to each process.
- **Maximum Matrix (M)**: This matrix represents the maximum number of resources each process may need.
- **Available Vector (V)**: This vector represents the number of available resources of each type in the system.
- **Need Matrix (N)**: This matrix represents the remaining resource needs of each process. It is calculated as:

N[i,j]=M[i,j]−A[i,j]

Algorithm Steps

1. **Initialize**:
   - Read the Allocation matrix, Maximum matrix, and Available vector.
   - Calculate the Need matrix using the formula: Need = Maximum - Allocation.
2. **Check Safety**:
   - Find a process Pi whose needs can be met with the currently available resources (i.e., if Needi≤Available).
   - If such a process is found, assume the resources are allocated to it, then check if the system remains in a safe state after the hypothetical allocation.
3. **Simulate Resource Allocation**:
   - Pretend to allocate the requested resources to the selected process.
   - Update the Available vector to reflect this allocation.
   - Mark the process as complete.
4. **Repeat**:
   - Repeat the process for all other processes.
   - If all processes can be safely completed, the system is in a safe state. Otherwise, it is in an unsafe state, and the resource allocation could lead to a deadlock.

Program:

Program Code:

/*5Banker.c*/

```
#include <stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, r;
void input();
void show();
void cal();
int main() {
   printf("********** Banker's Algorithm ***********\n");
   input();
   show();
   cal();
   getchar(); // Replaces getch() to pause the program
   return 0;
```

```
}
void input() {
    int i, j;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resource instances: ");
    scanf("%d", &r);
    printf("Enter the Max Matrix\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < r; j++) {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter the Allocation Matrix\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < r; j++) {
            scanf("%d", &alloc[i][j]);
        }
    }
    printf("Enter the available resources\n");
    for (j = 0; j < r; j++) {
        scanf("%d", &avail[j]);
    }
}
void show() {
    int i, j;
    printf("Process\t Allocation\t Max\t Available\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t ", i + 1);
        for (j = 0; j < r; j++) {
            printf("%d ", alloc[i][j]);
        }
        printf("\t");
        for (j = 0; j < r; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\t");
        if (i == 0) {
            for (j = 0; j < r; j++) {
                printf("%d ", avail[j]);
            }
        }
        printf("\n");
```

```
    }}
void cal() {
    int finish[100], temp, flag = 1, k, c1 = 0;
    int safe[100];
    int i, j;
    for (i = 0; i < n; i++) {
        finish[i] = 0;
    }
    // Calculate Need matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < r; j++) {
            need[i][j] = max[i][j] - alloc[i][j];
        }   }
    printf("\n");
    while (flag) {
        flag = 0;
        for (i = 0; i < n; i++) {
            int c = 0;
            for (j = 0; j < r; j++) {
                if (finish[i] == 0 && need[i][j] <= avail[j]) {
                    c++;
                }
                if (c == r) {
                    for (k = 0; k < r; k++) {
                        avail[k] += alloc[i][k];
                    }
                    finish[i] = 1;
                    flag = 1;
                    printf("P%d -> ", i);
                }           }       }     }
    for (i = 0; i < n; i++) {
        if (finish[i] == 1) {
            c1++;
        } else {
            printf("P%d -> ", i);
        }   }
    if (c1 == n) {
        printf("\nThe system is in a safe state\n");
    } else {
        printf("\nProcesses are in deadlock\n");
        printf("System is in an unsafe state\n");
    } }
```

Academic Year: 2020-21 **Semester:** I

Class : TE IT **Subject:** System Laboratory II (314447)

Output:

```
Terminal                                                    Aug 31 14:48
                                                  it@it-Vostro-3710: ~/OSL/FinalOSLPSD
it@it-Vostro-3710:~/OSL/FinalOSLPSD$ cc SBanker.c -o SBanker
it@it-Vostro-3710:~/OSL/FinalOSLPSD$ ./SBanker
********** Banker's Algorithm ************
Enter the number of processes: 5
Enter the number of resource instances: 3
Enter the Max Matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available resources
3 3 2
Process  Allocation      Max     Available
P1       0 1 0  7 5 3   3 3 2
P2       2 0 0  3 2 2
P3       3 0 2  9 0 2
P4       2 1 1  2 2 2
P5       0 0 2  4 3 3

P1 -> P3 -> P4 -> P0 -> P2 ->
The system is in a safe state
it@it-Vostro-3710:~/OSL/FinalOSLPSD$
```

OR

```
********** Banker's Algorithm **************
Enter the number of processes: 3
Enter the number of resource instances: 3
Enter the Max Matrix
7 5 3
3 2 2
9 0 2
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 2
Enter the available resources
3 3 2

Process          Allocation       Max              Available
P1               0 1 0            7 5 3            3 3 2
P2               2 0 0            3 2 2
P3               3 0 2            9 0 2

P1 -> P2 -> P3 ->
The system is in a safe state
```

Academic Year: 2020-21 **Semester:** I
Class : TE IT **Subject:** System Laboratory II (314447)

Assignment No. 6

Title : Page Replacement Algorithms

Aim: Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

Objective: To demonstrate the functioning of OS concepts in user space like concurrency control (process synchronization, mutual exclusion), CPU Scheduling, Memory Management and Disk Scheduling in LINUX.

Course Outcome (CO3): To develop various system programs for the functioning of OS concepts in user space like concurrency control, CPU Scheduling, Memory Management and Disk Scheduling in Linux.

Theory: **Page Replacement Algorithms**

In operating systems, page replacement algorithms are crucial for managing the contents of the page table when a page fault occurs. They decide which page to remove from memory when a new page needs to be loaded, optimizing performance based on various criteria. Here, we cover three fundamental algorithms: First-Come-First-Served (FCFS), Least Recently Used (LRU), and Optimal.

1. First-Come-First-Served (FCFS)

Description:

- **FCFS** is the simplest page replacement algorithm. It follows the principle of "first come, first served."
- When a page fault occurs, FCFS replaces the page that has been in memory the longest, regardless of its usage.

Operation:

- Maintain a queue to track the order in which pages were loaded into memory.
- When a new page is needed and there are no free frames, the page at the front of the queue is replaced.

Advantages:

- Simple to implement.
- Fair, as it replaces pages in the order they arrived.

Disadvantages:

- Can lead to poor performance and high page fault rates.

- Does not consider the frequency or recency of page usage.

2. Least Recently Used (LRU)

Description:

- **LRU** is based on the principle that the page which has not been used for the longest period of time is the one to replace.
- This algorithm keeps track of the order in which pages are accessed to decide which page to evict.

Operation:

- Maintain a record of the most recent access times for each page.
- When a page fault occurs, replace the page that has not been used for the longest time.

Advantages:

- Generally provides better performance compared to FCFS.
- More aligned with actual program behavior, assuming that recently used pages are likely to be used again soon.

Disadvantages:

- More complex to implement than FCFS.
- Requires additional hardware or software mechanisms to track page usage.

3. Optimal Page Replacement

Description:

- **Optimal Page Replacement** is a theoretical algorithm that aims to minimize the number of page faults.
- It replaces the page that will not be used for the longest period of time in the future.

Operation:

- Look ahead in the page reference string to determine which page will be used furthest in the future.
- Replace that page when a page fault occurs.

Advantages:

- Provides the best possible page fault rate, serving as a benchmark for other algorithms.

Program Code:

```
/*6FCFS .c*/
#include <stdio.h>
void printFrames(int frames[], int frameSize) {
   for (int i = 0; i < frameSize; i++) {
      if (frames[i] == -1) {
         printf(" - ");
      } else {
         printf(" %d ", frames[i]);
      }
   }
   printf("\n");
}
void fcfs(int refString[], int refSize, int frameSize) {
   int frames[frameSize];
   for (int i = 0; i < frameSize; i++) frames[i] = -1;
   int pageFaults = 0, nextReplace = 0;
   printf("\nFCFS Page Replacement:\n");
   for (int i = 0; i < refSize; i++) {
      int found = 0;
      for (int j = 0; j < frameSize; j++) {
         if (frames[j] == refString[i]) {
            found = 1;
            break;
         }
      }
      if (!found) {
         frames[nextReplace] = refString[i];
         nextReplace = (nextReplace + 1) % frameSize;
         pageFaults++;
      }
      printFrames(frames, frameSize);
   }
   printf("Total Page Faults: %d\n", pageFaults);
}
int main() {
   int refSize, frameSize;
   printf("Enter the number of pages in the reference string: ");
   scanf("%d", &refSize);
   int refString[refSize];
   printf("Enter the reference string:\n");
   for (int i = 0; i < refSize; i++) {
      scanf("%d", &refString[i]);
```

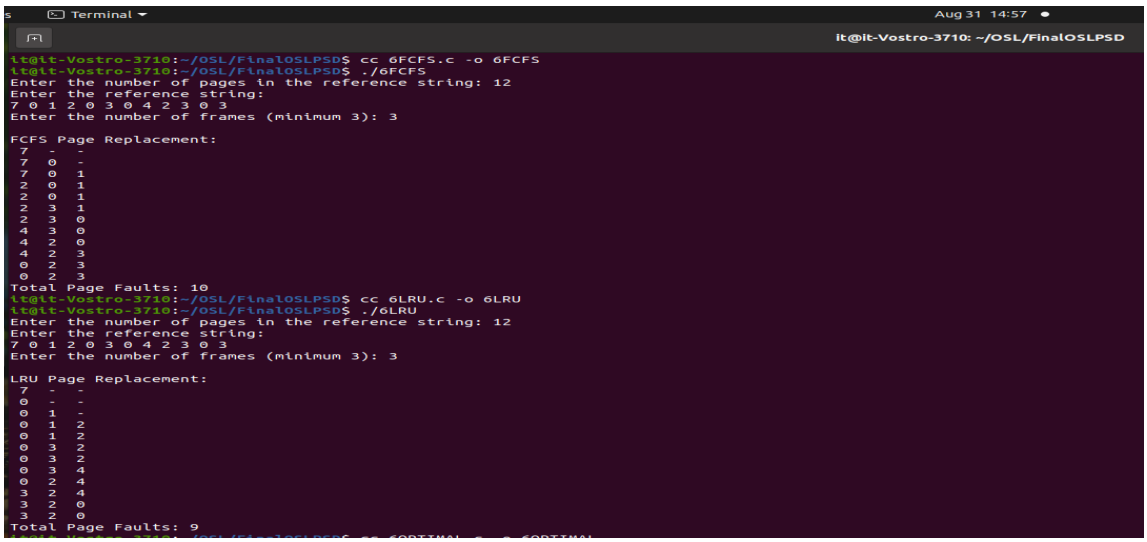
```
        }
        printf("Enter the number of frames (minimum 3): ");
        scanf("%d", &frameSize);
        if (frameSize < 3) {
            printf("Frame size should be at least 3.\n");
            return 1;
        }
        fcfs(refString, refSize, frameSize);
        return 0;
    }
```

/*6LRU .c*/

```
#include <stdio.h>
void printFrames(int frames[], int frameSize) {
    for (int i = 0; i < frameSize; i++) {
        if (frames[i] == -1) {
            printf(" - ");
        } else {
            printf(" %d ", frames[i]);
        }
    }
    printf("\n");
}
void lru(int refString[], int refSize, int frameSize) {
    int frames[frameSize];
    int time[frameSize];
    for (int i = 0; i < frameSize; i++) frames[i] = -1;
    int pageFaults = 0;
    printf("\nLRU Page Replacement:\n");
    for (int i = 0; i < refSize; i++) {
        int found = 0, leastRecentlyUsed = 0;
        for (int j = 0; j < frameSize; j++) {
            if (frames[j] == refString[i]) {
                found = 1;
                time[j] = i;
                break;
            }
            if (time[j] < time[leastRecentlyUsed]) {
                leastRecentlyUsed = j;
            }
        }
        if (!found) {
            frames[leastRecentlyUsed] = refString[i];
```

```
            time[leastRecentlyUsed] = i;
            pageFaults++;
        }
        printFrames(frames, frameSize);
    }
    printf("Total Page Faults: %d\n", pageFaults);
}
int main() {
    int refSize, frameSize;
    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &refSize);
    int refString[refSize];
    printf("Enter the reference string:\n");
    for (int i = 0; i < refSize; i++) {
        scanf("%d", &refString[i]);
    }
    printf("Enter the number of frames (minimum 3): ");
    scanf("%d", &frameSize);
    if (frameSize < 3) {
        printf("Frame size should be at least 3.\n");
        return 1;
    }
    lru(refString, refSize, frameSize);
    return 0;
}
```

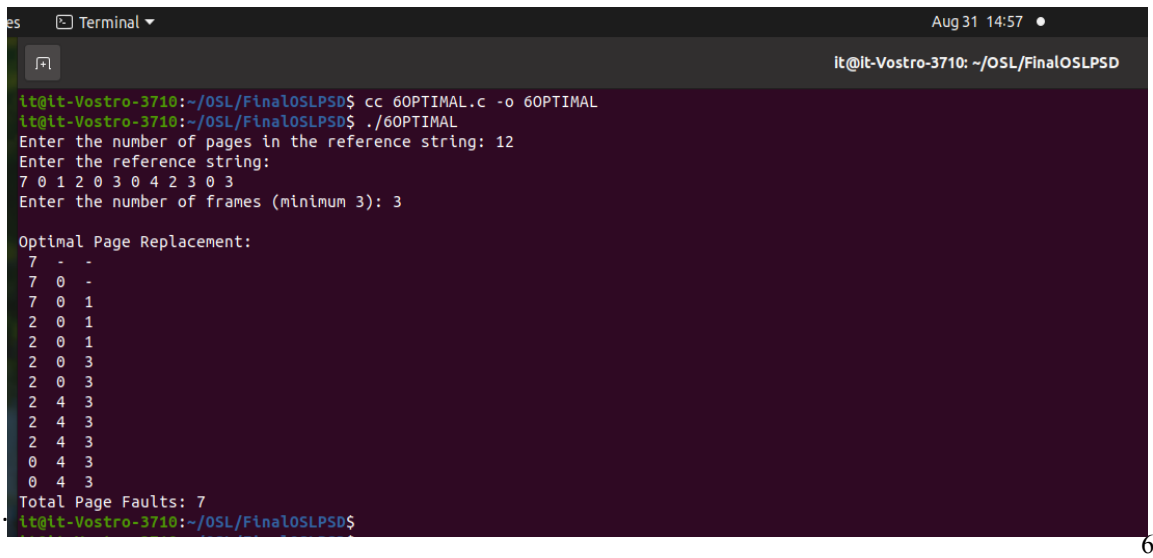
Output :



```
/*6OPTIMAL.c*/
#include <stdio.h>
void printFrames(int frames[], int frameSize) {
   for (int i = 0; i < frameSize; i++) {
      if (frames[i] == -1) {
         printf(" - ");
      } else {
         printf(" %d ", frames[i]);
      }
   }
   printf("\n");
}
int findOptimal(int frames[], int frameSize, int refString[], int refSize, int currentIndex) {
   int farthest = currentIndex;
   int index = -1;
   for (int i = 0; i < frameSize; i++) {
      int j;
      for (j = currentIndex; j < refSize; j++) {
         if (frames[i] == refString[j]) {
            if (j > farthest) {
               farthest = j;
               index = i;
            }
            break;
         }
      }
      if (j == refSize) return i;  // If not found in future, replace this
   }
   return index == -1 ? 0 : index;
}
void optimal(int refString[], int refSize, int frameSize) {
   int frames[frameSize];
   for (int i = 0; i < frameSize; i++) frames[i] = -1;
   int pageFaults = 0;
   printf("\nOptimal Page Replacement:\n");
   for (int i = 0; i < refSize; i++) {
      int found = 0;
      for (int j = 0; j < frameSize; j++) {
         if (frames[j] == refString[i]) {
            found = 1;
            break;
```

```
            }
        }
        if (!found) {
    int replaceIndex = (i < frameSize) ? i : findOptimal(frames, frameSize, refString, refSize, i + 1);
            frames[replaceIndex] = refString[i];
            pageFaults++;
        }
        printFrames(frames, frameSize);
    }
    printf("Total Page Faults: %d\n", pageFaults);
}
int main() {
    int refSize, frameSize;
    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &refSize);
    int refString[refSize];
    printf("Enter the reference string:\n");
    for (int i = 0; i < refSize; i++) {
        scanf("%d", &refString[i]);
    }
    printf("Enter the number of frames (minimum 3): ");
    scanf("%d", &frameSize);
    if (frameSize < 3) {
        printf("Frame size should be at least 3.\n");
        return 1;
    }
    optimal(refString, refSize, frameSize);
    return 0;
}
```

Output :



Academic Year: 2020-21 **Semester:** I
Class : TE IT **Subject:** System Laboratory II (314447)

A

Assignment No 7
Part A

Title Inter process communication in Linux using Pipe.

Aim: FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

Objective: To study the concept of inter-process communication using pipe.

Course Outcome (CO3): To implement basic building blocks like processes, threads under the Linux.

Theory:

The Pipe Call

The lower-level pipe( ) function provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command. It also gives us more control over the reading and writing of data.

The pipe function has the following prototype:

#include <unistd.h>
int pipe(int file_descriptor[2]);

It is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1 and sets errno to indicate the reason for failure. Errors defined in the Linux manual page for pipe (in section 2 of the manual) are

- EMFILE: Too many file descriptors are in use by the process.
- ENFILE: The system file table is full.
- EFAULT: The file descriptor is not valid.

The two file descriptors returned are connected in a special way. Any data written to file_descriptor[1]can be read back from file_descriptor[0]. The data is processed in a first in, first out basis. This means that if we write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2, 3. The illustration is given as below:

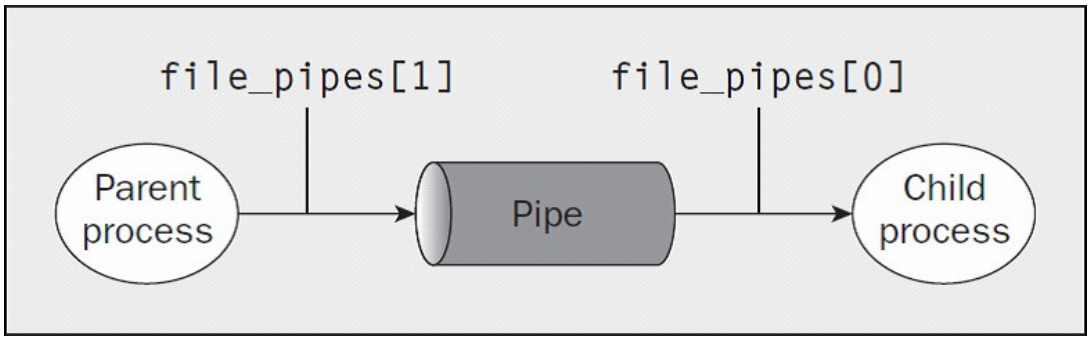
Each running program, called a process, has a number of file descriptors associated with it. These are small integers that we can use to access open files or devices. How many of these are available will vary depending on how the system has been configured. When a program starts, it usually has three of these descriptors already opened.

These are:

- 0: Standard input
- 1: Standard output
- 2: Standard error

We can associate other file descriptors with files and devices by using the open system call.

The file descriptors that are automatically opened, however, already allow you to create some simple programs using write.



The write( ) system call

The write system call arranges for the first *nbytes* bytes from *buf* to be written to the file associated with the file descriptor *fildes*. It returns the number of bytes actually written. This may be less than *nbytes* if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call, and the error will be specified in the *errno* global variable.

#include <unistd.h>

size_t write(int fildes, const void *buf, size_t nbytes); The read( ) system call

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf. It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

#include <unistd.h>

size_t read(int fildes, void *buf, size_t nbytes);

Conclusion: We have studied full duplex communication using PIPE.

Program Code:

/*1st Process: 7ASender.c*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define PIPE1 "/tmp/pipe1"
#define PIPE2 "/tmp/pipe2"
int main() {
   int fd1, fd2;
   char input[1000], output[1000];
   // Create the named pipes
   mkfifo(PIPE1, 0666);
```

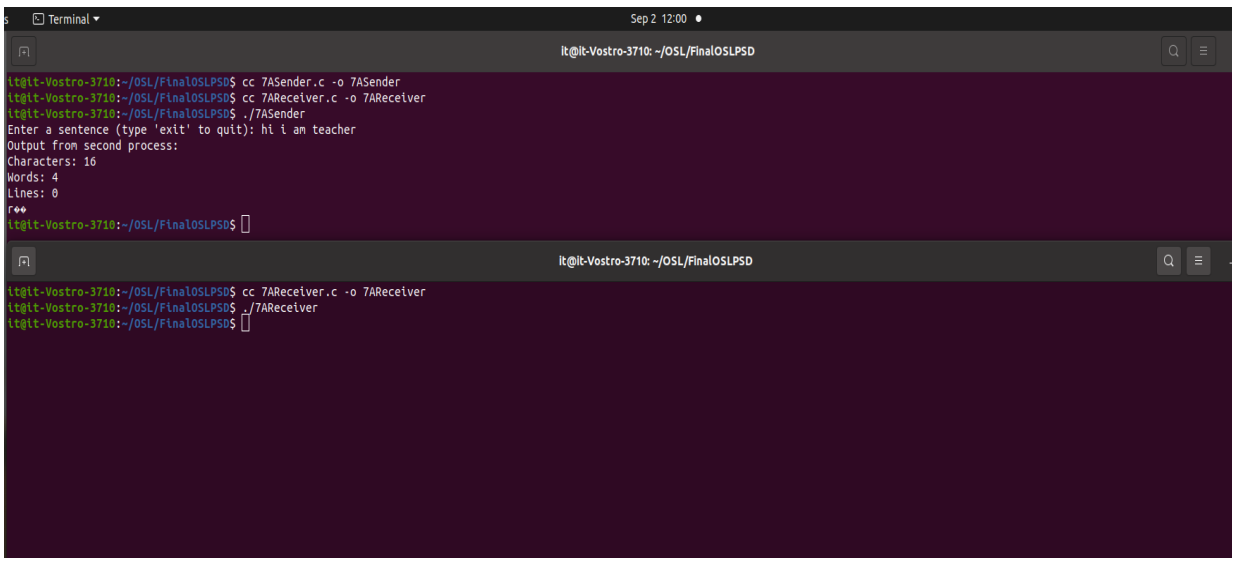
```
        mkfifo(PIPE2, 0666);
        // Get user input
        printf("Enter a sentence (type 'exit' to quit): ");
        fgets(input, sizeof(input), stdin);
        if (strncmp(input, "exit", 4) == 0) {
           return 0;    }
        // Open PIPE1 for writing
        fd1 = open(PIPE1, O_WRONLY);
        write(fd1, input, strlen(input) + 1);
        close(fd1);
        // Open PIPE2 for reading
        fd2 = open(PIPE2, O_RDONLY);
        read(fd2, output, sizeof(output));
        close(fd2);
        // Print the output received from the second process
        printf("Output from second process:\n%s\n", output);
        // Remove the named pipes
        unlink(PIPE1);
        unlink(PIPE2);
        return 0;}
```

/* 2nd Process: 7AReceiver.c */

```
      #include <stdio.h>
      #include <stdlib.h>
      #include <string.h>
      #include <unistd.h>
      #include <fcntl.h>
      #include <sys/types.h>
      #include <sys/stat.h>
      #define PIPE1 "/tmp/pipe1"
      #define PIPE2 "/tmp/pipe2"
      int countWords(char *str) {
         int count = 0;
         char *token = strtok(str, " \n");
         while (token != NULL) {
            count++;
            token = strtok(NULL, " \n");
         }
         return count;          }
      int countLines(char *str) {
         int count = 0;
         for (int i = 0; str[i] != '\0'; i++) {
            if (str[i] == '\n') {
               count++;
            }    }
         return count;          }
      int main() {
         int fd1, fd2;
         char input[1000], output[1000];
         int charCount, wordCount, lineCount;
```

```
    FILE *file;
    // Open PIPE1 for reading
    fd1 = open(PIPE1, O_RDONLY);
    read(fd1, input, sizeof(input));
    close(fd1);
    // Count characters, words, and lines
    charCount = strlen(input);
    wordCount = countWords(input);
    lineCount = countLines(input);
    // Write the results to a file
    file = fopen("output.txt", "w");
    fprintf(file, "Characters: %d\nWords: %d\nLines: %d\n", charCount, wordCount, lineCount);
    fclose(file);
    // Read the content of the file and send it through PIPE2
    file = fopen("output.txt", "r");
    fread(output, sizeof(char), sizeof(output), file);
    fclose(file);
    // Open PIPE2 for writing
    fd2 = open(PIPE2, O_WRONLY);
    write(fd2, output, strlen(output) + 1);
    close(fd2);
    return 0;
}
```

Output:



Assignment No 7
Part B

Title: Inter-process Communication using Shared Memory using System V.

Aim: Application to demonstrate: Client and Server Programs in which server process creates
a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

Objective: To study the concept of inter-process communication using shared memory. Course Outcome (CO4): To develop various system programs for the functioning of OS concepts in user space like concurrency control and file handling in Linux.

Theory:
IPC: Shared Memory
Shared Memory is an efficient means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.
In the interprocess communication mechanisms like the pipes, fifos and message queues, the work involved in sending data from one process to another is like this. Process P1 makes a system call to send data to Process P2. The message is copied from the address space of the first process to the kernel space during the system call for sending the message. Then, the second process makes a system call to receive the message. The message is copied from the kernel space to the address space of the second process. The shared memory mechanism does away with this copying overhead. The first process simply writes data into the shared memory
segment. As soon as it is written, the data becomes available to the second process. Shared memory is the fastest mechanism for interprocess communication.
Accessing a Shared Memory Segment shmget() is used to obtain access to a shared memory segment. It is prottyped by:
int shmget(key_t key, size_t size, int shmflg);
The key argument is a access value associated with the semaphore ID. The size argument is the size in bytes of the requested shared memory. The shmflg argument specifies the initial access permissions and creation control flags.
When the call succeeds, it returns the shared memory segment ID. This call is also used to get
the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).
The following code illustrates shmget():

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
key_t key; /* key to be passed to shmget() */
```

```
int shmflg; /* shmflg to be passed to shmget() */
int shmid; /* return value from shmget() */
int size; /* size to be passed to shmget() */
...
key = ...
size = ...
shmflg) = ...
if ((shmid = shmget (key, size, shmflg)) == -1) {
perror("shmget: shmget failed"); exit(1); } else {
(void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
exit(0);
}
...
```

Attaching and Detaching a Shared Memory Segment shmat() and shmdt() are used to attach and detach shared memory segments. They are prototypes as follows:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr.

Conclusion: We have studied inter-process Communication using Shared Memory using System **V.**

Program Code:

/*7B-Client.c*/

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_KEY 12345
#define SHM_SIZE 1024
int main() {
   int shmid;
   char *shmaddr;
   // Locate the shared memory segment created by the server
   shmid = shmget(SHM_KEY, SHM_SIZE, 0666);
   if (shmid < 0) {
      perror("shmget");
      exit(1);
   }
   // Attach the shared memory segment to the client's address space
   shmaddr = shmat(shmid, NULL, 0);
   if (shmaddr == (char *) -1) {
```

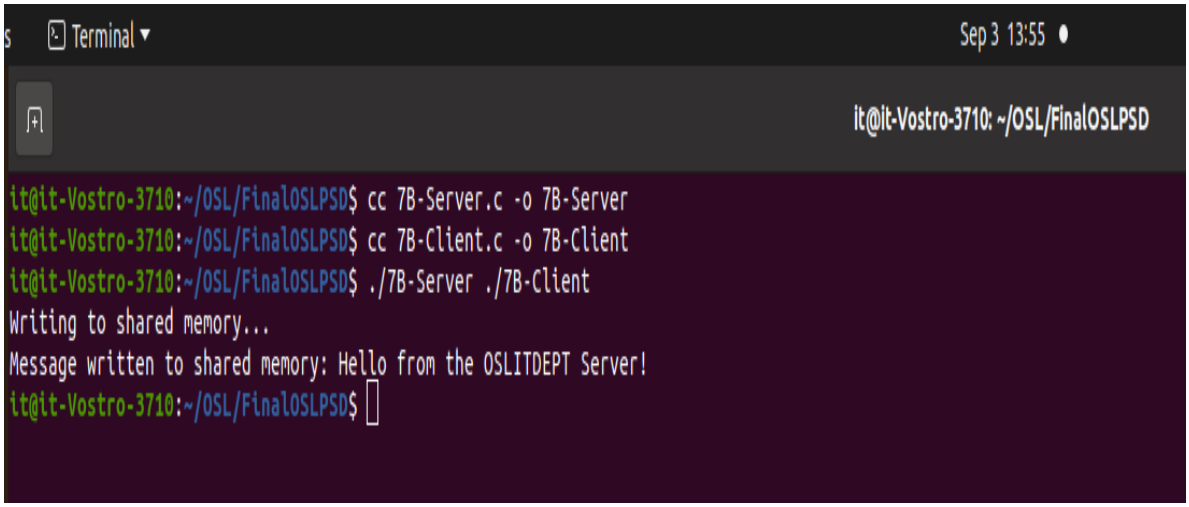
```
        perror("shmat");
        exit(1);
    }
    // Read the message from the shared memory segment
    printf("Reading from shared memory...\n");
    printf("Message from shared memory: %s\n", shmaddr);
    // Detach the shared memory segment
    if (shmdt(shmaddr) == -1) {
        perror("shmdt");
        exit(1);
    }
    return 0;
}
```

/* 7B-Server.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#define SHM_KEY 12345
#define SHM_SIZE 1024
int main() {
    int shmid;
    char *shmaddr;
    // Create a shared memory segment
    shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid < 0) {
        perror("shmget");
        exit(1);
    }
    // Attach the shared memory segment to the server's address space
    shmaddr = shmat(shmid, NULL, 0);
    if (shmaddr == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    // Write a message to the shared memory segment
    printf("Writing to shared memory...\n");
    char *message = "Hello from DVVPCOE,Ahmednagar Server!";
    strncpy(shmaddr, message, SHM_SIZE);
    // Detach the shared memory segment
    if (shmdt(shmaddr) == -1) {
```

```
        perror("shmdt");
        exit(1);
    }
    printf("Message written to shared memory: %s\n", message);
    return 0;   }
```

Output:



Academic Year: 2020-21 **Semester:** I

Class : TE IT **Subject:** System Laboratory II (314447)

Assignment No. 8

Title: Disk Scheduling Algorithms

Aim: Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.

Objective: To demonstrate the functioning of OS concepts in user space like Disk Scheduling In LINUX.

Course Outcome (CO4): Develop various system programs for the functioning of OS concepts in user space like Disk Scheduling in Linux.

Theory:

Disk scheduling algorithms are critical in managing how disk I/O requests are handled in a multi-tasking environment. These algorithms aim to optimize the order in which disk requests are serviced, minimizing the disk arm movement and reducing seek time. In this practical, we implement three popular disk scheduling algorithms: SSTF (Shortest Seek Time First), SCAN (Elevator Algorithm), and C-Look. The primary goal is to simulate these algorithms and compare their performance in terms of head movements and overall efficiency.

1. Shortest Seek Time First (SSTF) Algorithm

The Shortest Seek Time First (SSTF) algorithm services the I/O request that is closest to the current head position, reducing the seek time at each step. The algorithm minimizes the overall seek time by continuously choosing the request that is nearest to the head's current position.

Steps of SSTF:

The current position of the disk head is compared with all the pending requests.
The request with the shortest seek distance is selected next.
After the request is serviced, the disk head moves to the request's position.
Repeat until all requests are serviced.

Advantages: Reduces average seek time compared to FCFS (First-Come, First-Served).

Disadvantages: Starvation: Some requests might experience long delays if closer requests continue to arrive.

When to Use: Useful when minimizing seek time is critical, but starvation risk needs to be managed.

2. SCAN Algorithm (Elevator Algorithm)

The SCAN algorithm, also known as the "Elevator Algorithm," moves the disk head in one direction servicing all requests in its path until it reaches the end of the disk. After reaching one end, the head reverses direction and services requests on its way back. SCAN behaves like an elevator going up and down the disk.

Steps of SCAN:

The head moves in one direction (away from the spindle) servicing all requests in its path.
When the head reaches the end of the disk (or the last request in that direction), it reverses direction.
The head services the remaining requests in the opposite direction.

Advantages:

Avoids starvation, unlike SSTF, since it continuously moves in both directions.
Provides a fair scheduling system for disk requests.

Disadvantages: Can result in longer waiting times for requests far from the initial head position if the head is moving in the opposite direction.
When to Use: Suitable for systems that require fairness in servicing all requests but with acceptable performance.

3. C-Look Algorithm

C-Look is a variant of the SCAN algorithm. However, instead of going to the very end of the disk, the C-Look algorithm only goes as far as the last request in the current direction. After servicing the last request, the disk head jumps back to the first request on the opposite side without servicing any requests on the way back.
Steps of C-Look:
The head moves in one direction servicing all the requests.
When the last request in that direction is serviced, the head jumps back to the first request on the other side without servicing any intermediate requests.
The process is repeated until all requests are serviced.
Advantages: Reduces unnecessary head movement compared to SCAN, which continues to the disk's end even when no requests are pending.
Disadvantages: Requests may have to wait longer if they are at the far end of the list and the head has already moved past them.
When to Use: Ideal for systems where reducing unnecessary head movement is important while still ensuring fairness.

Conclusion: In this practical, we implemented three disk scheduling algorithms: SSTF, SCAN, and C-Look

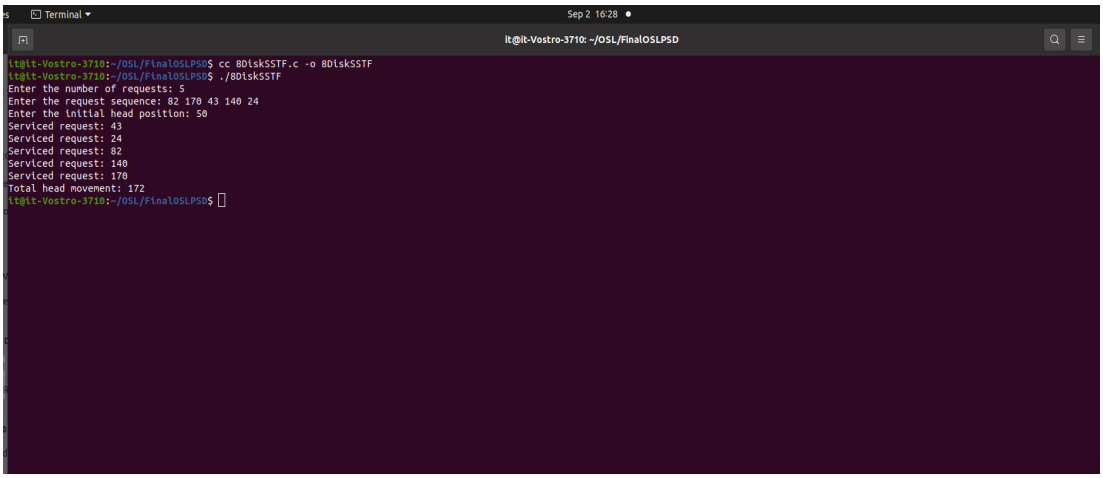
Program Code:

/* 8DiskSSTF.c*/

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int n, i, j, head, total_movement = 0;
   printf("Enter the number of requests: ");
   scanf("%d", &n);
   int requests[n], completed[n];
   printf("Enter the request sequence: ");
   for (i = 0; i < n; i++) {
      scanf("%d", &requests[i]);
      completed[i] = 0; // Mark all requests as uncompleted initially
   }
   printf("Enter the initial head position: ");
   scanf("%d", &head);
   for (i = 0; i < n; i++) {
```

```
        int min = 100000, min_index = -1;
        for (j = 0; j < n; j++) {
            if (!completed[j] && abs(head - requests[j]) < min) {
                min = abs(head - requests[j]);
                min_index = j;
            }
        }
        completed[min_index] = 1; // Mark the request as completed
        total_movement += abs(head - requests[min_index]);
        head = requests[min_index];
        printf("Serviced request: %d\n", head);
    }
    printf("Total head movement: %d\n", total_movement);
    return 0;
}
```

Output:



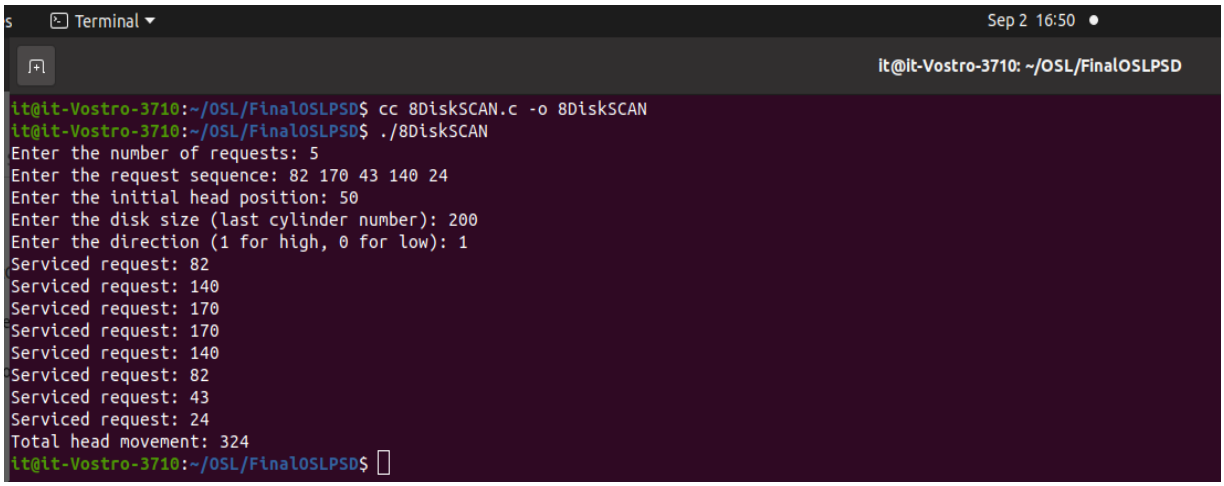
/* 8DiskSCAN.c*/

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, i, head, total_movement = 0, direction;
    printf("Enter the number of requests: ");
    scanf("%d", &n);
    int requests[n];
    printf("Enter the request sequence: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial head position: ");
    scanf("%d", &head);
```

```
printf("Enter the disk size (last cylinder number): ");
int disk_size;
scanf("%d", &disk_size);
printf("Enter the direction (1 for high, 0 for low): ");
scanf("%d", &direction);
// Sort the request array
for (i = 0; i < n - 1; i++) {
   for (int j = i + 1; j < n; j++) {
      if (requests[i] > requests[j]) {
         int temp = requests[i];
         requests[i] = requests[j];
         requests[j] = temp;
      }
   }
}
// SCAN algorithm
if (direction == 1) { // Move towards higher end
   for (i = 0; i < n && requests[i] < head; i++);
   for (; i < n; i++) {
      printf("Serviced request: %d\n", requests[i]);
      total_movement += abs(head - requests[i]);
      head = requests[i];
   }
   if (head < disk_size - 1) {
      total_movement += abs(head - (disk_size - 1));
      head = disk_size - 1;
   }
   for (i--; i >= 0; i--) {
      printf("Serviced request: %d\n", requests[i]);
      total_movement += abs(head - requests[i]);
      head = requests[i];
   }
} else { // Move towards lower end
   for (i = n - 1; i >= 0 && requests[i] > head; i--);
   for (; i >= 0; i--) {
      printf("Serviced request: %d\n", requests[i]);
      total_movement += abs(head - requests[i]);
      head = requests[i];
   }
   if (head > 0) {
      total_movement += head;
      head = 0;
   }
```

```
        for (i++; i < n; i++) {
            printf("Serviced request: %d\n", requests[i]);
            total_movement += abs(head - requests[i]);
            head = requests[i];
        }
    }
    printf("Total head movement: %d\n", total_movement);
    return 0;
}
```

Output:



/* 8DiskCLOOK.c*/

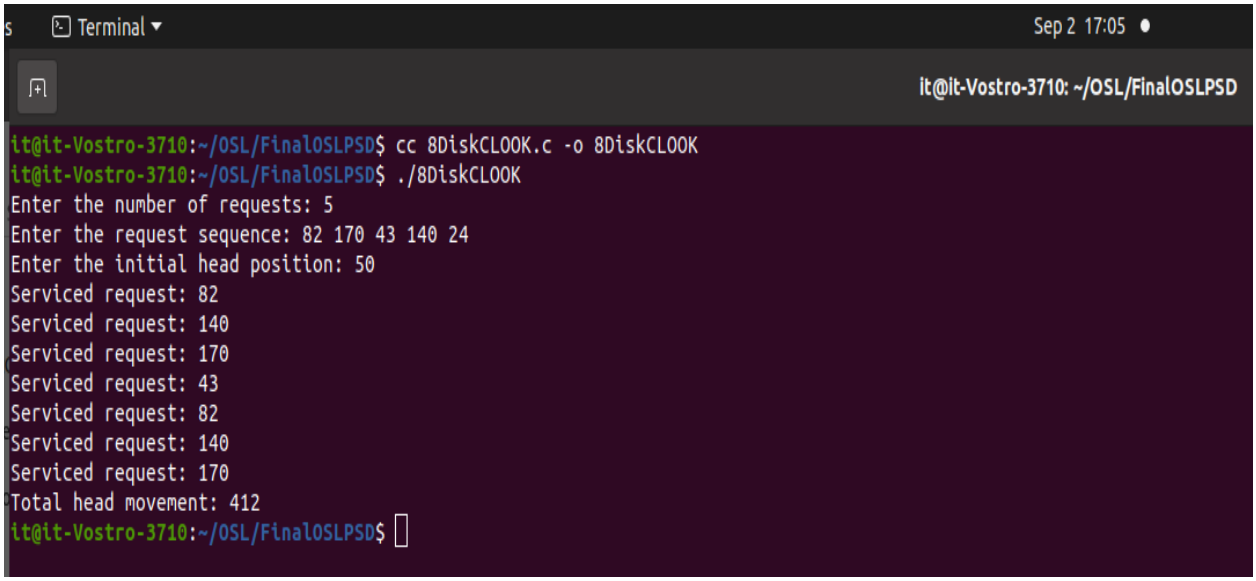
```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, i, head, total_movement = 0;
    printf("Enter the number of requests: ");
    scanf("%d", &n);
```

```
    int requests[n];
    printf("Enter the request sequence: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
    printf("Enter the initial head position: ");
    scanf("%d", &head);
    // Sort the request array
    for (i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (requests[i] > requests[j]) {
                int temp = requests[i];
                requests[i] = requests[j];
                requests[j] = temp;
            }
        }
    }
    // C-LOOK algorithm
    for (i = 0; i < n && requests[i] < head; i++);
    for (; i < n; i++) {
        printf("Serviced request: %d\n", requests[i]);
        total_movement += abs(head - requests[i]);
        head = requests[i];
    }
    if (i > 0) {
        total_movement += abs(head - requests[0]);
        head = requests[0];
        for (i = 1; i < n; i++) {
            printf("Serviced request: %d\n", requests[i]);
            total_movement += abs(head - requests[i]);
            head = requests[i];
        }
    }
    printf("Total head movement: %d\n", total_movement);
    return 0;
}
```

Academic Year: 2020-21 **Semester:** I

Class : TE IT **Subject:** System Laboratory II (314447)

Output:



Assignment No. 9

Title: Kernel Programming

Aim: Implement a new system call in the kernel space, add this new system call in the Linux kernel by the compilation of this kernel (any kernel source, any architecture and any Linux kernel distribution) and demonstrate the use of this embedded system call using C program in user space.

Objective: To study and implement new system call in the kernal space.

Course Outcome (CO5): To design and implement Linux Kernel Source Code.
Course Outcome (CO6): To develop the system program for the functioning of OS concepts in kernel space like embedding the system call in any Linux kernel.

Theory:

The steps to be followed to add a hello world system call in your kernel are :-

You can gain root access by using “sudo -s” command and typing in the password when prompted; to avoid typing “sudo” separately for all commands.

• Download the kernel source Open the terminal and use the following command to download the kernel course file

https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.16.tar.xz

• Extract the kernel source code Extract the kernel source code from the linux-3.16.tar.xz file in /usr/src/ directory using the following command. Since the downloaded tar file will be in Downloads folder, use cd to change into Downloads folder before executing the below command. sudo tar -xvf linux-3.16.tar.xz -C/usr/src/

sudo – to gain root access.

tar – Tar stores and extracts files from a tape or disk archive

. -x – extract files from an archive

-v – requested using the –verbose option, when extracting archives

-f – –file archive; use archive file or device archive -C, –directory DIR,change to directory DIR(here to change to /usr/src/) Now after extraction change to the kernel source directory using, cd /usr/src/linux-3.16/

• Define a new system call sys_hello()

Create a directory hello in the kernel source directory:- mkdir hello Change into this directory

cd hello

•Create a "hello.c" file in this folder and add the definition of the system call to it as given below (you can use any text editor).

gedit hello.c add the following code:-

```
#include <linux/kernel.h>

asmlinkage long sys_hello(void)
{
printk("Hello world\n");
return 0;
}
```

Create a "Makefile" in the hello folder and add the given line to it.

//Makefile

obj-y := hello.o

This is to ensure that the hello.c file is compiled and included in the kernel source code. • Add the hello directory to the kernel's Makefile change back into the linux-3.16 folder and open Makefile gedit Makefile goto line number 842 which says :-

"core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ " change this to "core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ hello/"

This is to tell the compiler that the source files of our new system call (sys_hello()) are in present in the hello directory.

Add the new system call (sys_hello()) into the system call table (syscall_32.tbl file) If your system is a 64 bit system you will need to alter the syscall_64.tbl file.

cd arch/x86/syscalls gedit syscall_32.tbl

add the following line in the end of the file :- 354 i386 hello sys_hello 354 – It is the number of the system call . It should be one plus the number of the last system call. (it was 354 in my system). This has to be noted down to make the system call in the userspace program.

Add the new system call(sys_hello()) in the system call header file. cd

include/linux/ gedit syscalls.h

add the following line to the end of the file just before the #endif statement at the very bottom.
asmlinkage long sys_hello(void);

This defines the prototype of the function of our system call."asmlinkage" is a key word used to indicate that all parameters of the function would be available on the stack. • Compile this kernel on your system.

To compile Linux Kernel the following are required to be installed.

1. gcc latest version,
2. ncurses development package
3. system packages should be up-to date Hence do the following in the terminal.

1. sudo apt-get install gcc
2. sudo apt-get install libncurses5-dev
3. sudo apt-get update
4. sudo apt-get upgrade =>

To configure your kernel use the following command:- sudo make

menuconfig

Once the above command is used to configure the Linux kernel, you will get a pop up window with the list of menus and you can select the items for the new configuration. If your unfamiliar with the configuration just check for the file systems menu and check whether "ext4" is chosen or not, if not select it and save the configuration.

If you like to have your existing configuration then run the below command. sudo make

oldconfig

Now to compile the kernel ; do make .

cd /usr/src/linux-3.16/ make

This might take several hours depending on your system (using hypervisors can take a longer time).

It took me 2-3 hours to get this compiled.

- To install /update the kernel.

To install this edited kernel run the following command:- sudo make modules_install install The above command will install the Linux Kernel 3.16 into your system. It will create some files under /boot/ directory and it will automatically make a entry in your grub.cfg. To check whether it made correct entry; check the files under /boot/ directory. If you have followed the steps without any error you will find the following files in it in addition to others. 1. System.map-3.16.0 2. vmlinuz-3.16.0 3. initrd.img-3.16.0 4. config-3.16.0 Now to update the kernel in your system reboot the system . You can use the following command. shutdown -r now After rebooting you can verify the kernel version using the following command; uname -r

- To test the system call.

Create a "userspace.c" program in your home folder and type in the following code :

```
//userspace.c
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
int main()
{
long int amma = syscall(354);
printf("System call sys_hello returned %ld\n", amma);
return 0;
}
```

Now compile this program using the following command. gcc userspace.c

If all goes well you will not have any errors else, rectify the errors.

Now run the program using the following command. .

/a.out You will see the following line getting printed in the terminal if all the steps were followed correctly. "System call sys_hello returned 0"

Now to check the message of the kernel you can run the following command.

dmesg This will display "Hello world" at the end of the kernel's message

Conclusion: We have studied how to add a new system call in newly compiled the kernel .

Academic Year: 2020-21 **Semester:** I
Class : TE IT **Subject:** System Laboratory II (314447)

Program:

```
#include <linux/kernel.h>

asmlinkage long sys_hello(void)
{
printk(“Hello world\n”);
return 0;
}

//Makefile

obj-y := hello.o

//userspace.c

#include <stdio.h>
#include <linux/kernel.h>
#include   <sys/syscall.h>
#include <unistd.h>
int main()
{
long int amma = syscall(354);
printf(“System call sys_hello returned %ld\n”, amma);
return 0;
}
```

Output: