Savitribai Phule Pune University, Pune
Third Year Information Technology (2019 Course)
314446 : Operating Systems Lab Syllabus

Teaching Scheme: **Credit Scheme:** **Examination Scheme:**
Practical (PR) : 4 hrs/week **02 Credits** **PR: 25 Marks**
TW: 25 Marks

Prerequisites:
1. C Programming
2. Fundamentals of Data Structure

Course Objectives:
1. To introduce and learn Linux commands required for administration.
2. To learn shell programming concepts and applications.
3. To demonstrate the functioning of OS basic building blocks like processes, threads under the LINUX.
4. 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.
5. To demonstrate the functioning of Inter Process Communication under LINUX.
6. To study the functioning of OS concepts in kernel space like embedding the system call in any LINUX
kernel.

Course Outcomes:
On completion of the course, students will be able to–
CO1: Apply the basics of Linux commands.
CO2: Build shell scripts for various applications.
CO3: Implement basic building blocks like processes, threads under the Linux.
CO4: 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.
CO5: Develop system programs for Inter Process Communication in Linux.

Laboratory Assignments

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

Assignment No. 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. 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.
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. Parent process sorts an 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 which display array in reverse order.

Assignment No. 3:

Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

Assignment No. 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.

Assignment No. 5:
Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

Assignment No. 6:
Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

Assignment No. 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.

Assignment No. 8:

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

Study Assignment:
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.

INDEX

SCHEDULE

Sr. No.	Title	No. of Hrs.	Week
1	Assignment No. 1 : A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc..	4	1
2	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	4	2
3	Assignment No. 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. 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.	4	3
4	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. Parent process sorts an 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 which display array in reverse order.	4	4
5	Assignment No. 3: Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.	4	5
6	Assignment No. 4: A. Thread synchronization using counting semaphores. Application to demonstrate: producer- consumer problem with counting semaphores and mutex.	4	6
7	B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority.	4	7
8	Assignment No. 5: Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.	4	8
9	Assignment No. 6: Implement the C program for Page Replacement	4	9

	Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.		
10	Assignment No. 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.	4	10
11	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.	4	11
12	Assignment No. 8: Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.	2	11
13	Study Assignment: 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.	2	12

Assignment No. : 1 (A & B)

Study of Basic Linux Commands &
Shell Programming

ASSIGNMENT NO. 1

A. STUDY OF BASIC LINUX COMMANDS & B. SHELL PROGRAMMING

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

OBJECTIVES : To study

1. Basic Shell commands
2. Shell script

THEORY :

Shell Script: Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands), the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is known as shell script. Shell Script is series of command written in plain text file. This manual is meant as a brief introduction to features found in Bash.

Exit Status:

By default, in Linux if particular command/shell script is executed, it returns two type of values which is used to see whether command or shell script executed is successful or not.

(1) If return value is zero (0), command is successful.

(2) If return value is nonzero, command is not successful or some sort of error executing command/shell script.

This value is known as Exit Status.But how to find out exit status of command or shell script? Simple, to determine this exit Status you can use $? special variable of shell.

For e.g. (This example assumes that unknow1file doest not exist on your hard drive)

$ rm unknow1file

It will show error as follows

rm: cannot remove `unkowm1file': No such file or directory

and after that if you give command

$ echo $?

it will print nonzero value to indicate error.

User defined variables (UDV)

To define UDV use following syntax

Syntax:

variable name=value

'value' is assigned to given 'variable name' and Value must be on right side = sign.

Example:

To define variable called n having value 10

$ n=10

To print or access UDV use following syntax

Syntax:

$variablename

$ n=10

To print contains of variable 'n' type command as follows

$ echo $n

About Quotes

There are three types of quotes

Quotes	Name	Meaning
"	Double Quotes	"Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $).
'	Single quotes	'Single quotes' - Enclosed in single quotes remains unchanged.
`	Back quote	`Back quote` - To execute command

Example:

$ echo "Today is date"

Can't print message with today's date.

$ echo "Today is `date`".

Rules for Naming variable name (Both UDV and System Variable)

(1) Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character. For e.g. Valid shell variable are as follows

HOME

SYSTEM_VERSION

(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g. In following variable declaration there will be no error

$ no=10

But there will be problem for any of the following variable declaration:

$ no =10

$ no= 10

$ no = 10

(3) Variables are case-sensitive, just like filename in Linux.

(4) You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition) For e.g.

$ vech=

$ vech=""

Try to print it's value by issuing following command

$ echo $vech

Nothing will be shown because variable has no value i.e. NULL variable.

(5) Do not use ?,* etc, to name your variable names.

Shell Arithmetic

Use to perform arithmetic operations.

Syntax:

expr op1 math-operator op2

Examples:

$ expr 1 + 3

$ expr 2 - 1

$ expr 10 / 2

$ expr 20 % 3

$ expr 10 * 3

$ echo `expr 6 + 3`

Note:

expr 20 %3 - Remainder read as 20 mod 3 and remainder is 2.

expr 10 * 3 - Multiplication use * and not * since its wild card.

The read Statement

Use to get input (data from user) from keyboard and store (data) to variable.

Syntax:

read variable1, variable2,...variableN

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.

```
$ vi sayH
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Variables in Shell

To process our data/information, data must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).

In Linux (Shell), there are two types of variable:

(1) System variables - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

(2) User defined variables (UDV) - Created and maintained by user. This type of variable defined in lower letters.

You can see system variables by giving command like $ set, some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
HOME=/home/vivek	Our home directory
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name

You can print any of the above variables contains as follows:

$ echo $HOME

test command or [expr]

test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

Syntax:

test expression OR [expression]

Example:

Following script determine whether given argument number is positive.

```
if test $1 -gt 0
then
echo "$1 number is positive"
fi
```

test or [expr] works with

1.Integer (Number without decimal point)

2.File types

3.Character strings

For Mathematics, use following operator in Shell Script

Mathematical	Meaning	Normal Arithmetical/	But in Shell

Operator in Shell Script		Mathematical Statements		
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

NOTE: == is equal, != is not equal.

For string Comparisons use

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Shell also test for file and directory types

	Meaning

-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

Logical Operators:

Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

if condition

if condition which is used for decision making in shell script, If given condition is true then command1 is executed.

Syntax:

```
if condition
then
        command1 if condition is true or if exit status
        of condition is 0 (zero)
        …
        …
fi
```

Condition is defined as:

"Condition is nothing but comparison between two values."

For compression you can use test or [expr] statements or even exist status can be also used.

Loops in Shell Scripts

Bash supports:

1) for loop

2) while loop

while :

The syntax of the while is:

while *test-commands*

do

commands

done

Execute *commands* as long as *test-commands* has an exit status of zero.

for :

The syntax of the for is:

for variable in list

do

commands

done

Each white space-separated word in list is assinged to variable in turn and commands executed until list is exhausted.

The case Statement

The case statement is good alternative to Multilevel if-then-else-fi statement. It enable you to match several values against one variable. Its easier to read and write.

Syntax:

case $variable-name in

pattern1) command

...

..

command;;

pattern2) command

...

..

command;;

patternN) command

...

..

command;;

*) command

...

..

command;;

esac

The $variable-name is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found.

Sed Command in Linux/Unix with examples

SED command in UNIX is stands for stream editor and it can perform lots of function on file like, searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening it, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

Syntax:

sed OPTIONS... [SCRIPT] [INPUTFILE...]

Example:

Consider the below text file as an input.

$cat > file.txt

unix is great os. unix is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
unix is easy to learn. unix is a multiuser os. Learn unix. unix is a powerful.

Sample Commands

1. **Replacing or substituting string:** Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word “unix” with “linux” in the file.

   $sed 's/unix/linux/' file.txt

Output:

linux is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os. Learn unix. unix is a powerful.

Here the "s" specifies the substitution operation. The "/" are delimiters. The "unix" is the search pattern and the "linux" is the replacement string.

By default, the sed command replaces the first occurrence of the pattern in each line and it won't replace the second, third…occurrence in the line.

2. **Replacing the nth occurrence of a pattern in a line:** Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word "unix" with "linux" in a line.

   $sed 's/unix/linux/2' file.txt

   Output:
   unix is great os. linux is opensource. unix is free os.
   learn operating system.
   unix linux which one you choose.
   unix is easy to learn. linux is a multiuser os. Learn unix. unix is a powerful.

3. **Replacing all the occurrence of the pattern in a line:** The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.

   $sed 's/unix/linux/g' file.txt

   Output:

   linux is great os. linux is opensource. linux is free os.
   learn operating system.
   linux linux which one you choose.
   linux is easy to learn. linux is a multiuser os. Learn linux. linux is a powerful.

4. **Replacing from nth occurrence to all occurrences in a line:** Use the combination of /1, /2 etc and /g to replace all the patterns from the nth occurrence of a pattern in a line. The following sed command replaces the third, fourth, fifth… "unix" word with "linux" word in a line.

   $sed 's/unix/linux/3g' file.txt

   Output:
   unix is great os. unix is opensource. linux is free os.

learn operating system.
unix linux which one you choose.
unix is easy to learn. unix is a multiuser os. Learn linux. linux is a powerful.

5. **Parenthesize first character of each word:** This sed example prints the first character of every word in parenthesis.
   $ echo "Welcome To The Linux Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g'

   Output:

   (W)elcome (T)o (T)he (L)inux (S)tuff

6. **Replacing string on a specific line number:** You can restrict the sed command to replace the string on a specific line number. An example is

   $sed '3 s/unix/linux/' file.txt

   Output:

   unix is great os. unix is opensource. unix is free os.
   learn operating system.
   linux linux which one you choose.
   unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

   The above sed command replaces the string only on the third line.

7. **Duplicating the replaced line with /p flag:** The /p print flag prints the replaced line twice on the terminal. If a line does not have the search pattern and is not replaced, then the /p prints that line only once.

8. **$sed 's/unix/linux/p' file.txt**

   Output:
   linux is great os. unix is opensource. unix is free os.
   linux is great os. unix is opensource. unix is free os.
   learn operating system.
   linux linux which one you choose.
   linux linux which one you choose.
   linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
   linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

9. **Printing only the replaced lines:** Use the -n option along with the /p print flag to display only the replaced lines. Here the -n option suppresses the duplicate rows generated by the /p flag and prints the replaced lines only one time.
   $sed -n 's/unix/linux/p' file.txt

   Output:
   linux is great os. unix is opensource. unix is free os.

linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

If you use -n alone without /p, then the sed does not print anything.

10. **Replacing string on a range of lines:** You can specify a range of line numbers to the sed command for replacing a string.

    $sed '1,3 s/unix/linux/' file.txt

    Output:
    linux is great os. unix is opensource. unix is free os.
    learn operating system.
    linux linux which one you choose.
    unix is easy to learn. unix is a multiuser os. Learn unix. unix is a powerful.

    Here the sed command replaces the lines with range from 1 to 3. Another example is

    $sed '2,$ s/unix/linux/' file.txt

    Output:

    unix is great os. unix is opensource. unix is free os.
    learn operating system.
    linux linux which one you choose.
    linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful

    Here $ indicates the last line in the file. So the sed command replaces the text from second line to last line in the file.

11. **Deleting lines from a particular file:** SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the file

    Examples:
    1. To Delete a particular line say n in this example
    Syntax:

        $ sed 'nd' filename.txt

    Example:

        $ sed '5d' filename.txt

    2. To Delete a last line

        Syntax:

        $ sed '$d' filename.txt

3. To Delete line from range x to y

Syntax:

$ sed 'x,yd' filename.txt

Example:

$ sed '3,6d' filename.txt

5. To Delete from nth to last line

Syntax:

$ sed 'nth,$d' filename.txt

Example:

$ sed '12,$d' filename.txt

6. To Delete pattern matching line

Syntax:

$ sed '/pattern/d' filename.txt

Example:

$ sed '/abc/d' filename.txt

Conclusion: Thus in shell script we can write series of commands and execute as a single program.

Answer the following questions.

1. What are different types of shell & differentiate them.
2. Explain Exit status of a command.
3. Explain a) User define variables.
4. System variables.
5. What is man command?
6. Explain test command.
7. Explain how shell program get executed.
8. Explain syntax of if-else, for, while, case.
9. Explain use of functions in shell and show it practically.
10. Write a menu driven shell script to execute different commands with options.
11. Execute same assignment in another shell.

Assignment No. : 2

Process Control System Calls

ASSIGNMENT NO. 2

PROCESS CONTROL SYSTEM CALLS

AIM : 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. 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

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. Parent process sorts an 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 which display array in reverse order.

OBJECTIVES : To study

3. Process control
4. Zombie and orphan processes
5. System calls : fork, execv

THEORY :

Process

A process is the basic active entity in most operating-system models. A process is a program in execution in memory or in other words, an instance of a program in memory. Any program executed creates a process. A program can be a command, a shell script, or any binary executable or any application.

Process IDs

Each process in a Linux system is identified by its unique *process ID*, sometimes referred to as *pid*. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

When referring to process IDs in a C or C++ program, always use the pid_t typedef, which is defined in <sys/types.h>. A program can obtain the process ID of the process it's running in with the getpid() system call, and it can obtain the process ID of its parent process with the getppid() system call.

Creating Process

Two common techniques are used for creating a new process.

1. using system() function
2. using fork() system calls

1. system() function

The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell. In fact, system creates a subprocess running the standard Bourne shell (/bin/sh) and hands the command to that shell for execution.

The system function returns the exit status of the shell command. If the shell itself cannot be run, system returns 127; if another error occurs, system returns –1.

2.fork system call

A process can create a new process by calling fork. The calling process becomes the parent, and the created process is called the child. The fork function copies the parent's memory image so that the new process receives a copy of the address space of the parent. Both processes continue at the instruction after the fork statement (executing in their respective memory images).

Synopsis

```
#include <unistd.h>
```

```
pid_t fork(void);
```

The fork function returns 0 to the child and returns the child's process ID to the parent. When fork fails, it returns –1.

Wait Function

When a process creates a child, both parent and child proceed with execution from the point of the fork. The parent can execute wait to block until the child finishes. The wait function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.

Synopsis

```
#include <sys/wait.h>
pid_t wait(int *status);
```

If wait returns because the status of a child is reported, these functions return the process ID of that child. If an error occurs, these functions return –1.

Example:

```
pid_tchildpid;
childpid = wait(NULL);
if (childpid != -1)
printf("Waited for child with pid %ld\n", childpid);
```

Status values

The status argument of wait is a pointer to an integer variable. If it is not NULL, this function stores the return status of the child in this location. The child returns its status by calling exit, _exit or return from main.

A zero return value indicates EXIT_SUCCESS; any other value indicates EXIT_FAILURE.

POSIX specifies six macros for testing the child's return status. Each takes the status value returned by a child to wait as a parameter. Following are the two such macros:

Synopsis

```
#include <sys/wait.h>
WIFEXITED(intstat_val)
WEXITSTATUS(intstat_val)
```

Exec system call

Used for new program execution within the existing process. The fork function creates a copy of the calling process, but many applications require the child process to execute code that is different from that of the parent. The exec family of functions provides a facility for overlaying the process image of the calling process with a new image. The traditional way to use the fork–exec combination is for the child to execute (with an exec function) the new program while the parent continues to execute the original code. 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.

Synopsis

```
#include <unistd.h>
    extern char **environ;
```

Exec family

1. intexecl(const char *path, const char *arg0, ... /*, char *(0) */);
2. intexecle (const char *path, const char *arg0, ... /*, char *(0), char *constenvp[] */);
3. intexeclp (const char *file, const char *arg0, ... /*, char *(0) */);
4. intexecv(const char *path, char *constargv[]);
5. intexecve (const char *path, char *constargv[], char *constenvp[]);
6. intexecvp (const char *file, char *constargv[]);

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. It 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( )

It executes the program pointed to by filename.

intexecve(const char *filename, char *constargv[ ], char *constenvp[ ]);

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(intargc, 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.

All exec functions return –1 if unsuccessful. In case of success these functions never return to the calling function.

Process Termination

Normally, a process terminates in one of two ways. Either the executing program calls the exitfunction, or the program’s main functionreturns. Each process has an exit code: anumber that the process returns to its parent. The exit code is the argument passed to the exit function, or the value returned from main.

Zombie Processes

If a child process terminates while its parent is calling a wait function, the child process vanishes and its termination status is passed to its parent via the wait call. But what happens, when a child process terminates and the parent is not calling wait? Does it simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, it becomes a zombie process.

A *zombie process* is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The wait functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then calls wait. If the child process has not terminated at that point, the parent process will block in the wait call until the child process finishes. If the child process finishes before the parent process calls wait, the child process becomes a zombie. When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.

Orphan Process

Orphan process is the process whose parent process is dead. That is, parent process is terminated, killed or exited but the child process is still alive.

In Linux/Unix like operating systems, as soon as parents of any process are dead, re-parenting occurs, automatically. Re-parenting means orphaned process is immediately adopted by special process. Thing to notice here is that even after re-parenting, the process still remains Orphan as the parent which created the process is dead.

A process can be orphaned either intentionally or unintentionally. Sometimes a parent process exits/terminates or crashes leaving the child process still running, and then they become orphans. Also, a process can be intentionally orphaned just to keep it running. For example when you need to run a job in the background which need any manual intervention and going to take long time, then you detach it from user session and leave it there. Same way, when you need to run a process in the background for infinite time, you need to do the same thing. Processes running in the background like this are known as daemon process.

Finding a Orphan Process

It is very easy to spot a Orphan process. Orphan process is a user process, which is having init process (process id 1) as parent. You can use this command in linux to find the orphan processes.

```
# ps -elf | head -1; ps -elf | awk '{if ($5 == 1 && $3 != "root") {print $0}}' | head
```

This will show you all the orphan processes running in your system. The output from this command confirms that they are orphan processes but does not mean that they are all useless.

Killing a Orphan Process

As orphan processes waste server resources, it is not advised to have lots of orphan processes running into the system. To kill a orphan process is same as killing a normal process.

```
# kill -15 <PID>
```

If that does not work then simply use

```
# kill -9 <PID>
```

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.

vfork

It is alternative of fork. Creates a new process when exec a new program.

Comparison with fork

1. Creates new process without fully copying the address space of the parent.
2. vfork guarantees that the child runs first, until the child calls exec or exit.
3. When child calls either of these two functions (exit, exec), the parent resumes.

Input

1. An integer array with specified size.
2. An integer array with specified size and number to search.

Output

1. Sorted array.
2. Status of number to be searched.

FAQs

- Is Orphan process different from a Zombie process?
- Are Orphan processes harmful for system?

- Is it bad to have Zombie processes on your system?
- How to find an Orphan Process?
- How to find a Zombie Process?
- What is common shared data between parent and child process?
- What are the contents of Process Control Block?

Practice Assignments

Example 1

Printing the Process ID

```
#include <stdio.h>
#include <unistd.h>
        int main()
        {
        printf("The process ID is %d\n", (int) getpid());
        printf("The parent process ID is %d\n", (int) getppid());
        return 0;
        }
```

Example 2

Using the system call

```
#include <stdlib.h>
        int main()
                {
                intreturn_value;
                return_value=system("ls –l /");
                return return_value;
                }
```

Example 3

Using fork to duplicate a program's process

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
        int main()
        {
                pid_t child_pid;
                printf("The main program process ID is %d\n", (int) getpid());
                child_pid=fork();
        if(child_pid!=0)
        {
                printf("This is the parent process ID, with id %d\n", (int)
                getpid());
                printf("The child process ID is %d\n", (int) child_pid);
        }
        else
                printf("This is the child process ID, with id %d\n", (int)
                getpid());
                return 0;
        }
```

Example 4

Determining the exit status of a child

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
        void show_return_status(void)
        {
                pid_tchildpid;
                int status;
                childpid = wait(&status); if (childpid == -1)
```

```
                perror("Failed to wait for child");
                else if (WIFEXITED(status))
                        printf("Child %ld terminated with return status %d\n", (long)childpid,
                        WEXITSTATUS(status));
        }
```

Example 5

A program that creates a child process to run ls -l

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
        int main(void)
        {
                pid_tchildpid;
                childpid = fork();
                if (childpid == -1) {
                        perror("Failed to fork");
                        return 1;
                }
                if (childpid == 0) {
                        /* child code */
                        execl("/bin/ls", "ls", "-l", NULL);
                        perror("Child failed to exec ls"); return 1;
                }
                if (childpid != wait(NULL)) {
                        /* parent code */
                        perror("Parent failed to wait due to signal or error"); return 1;
                }
                return 0;
        }
```

Example 6

Making a zombie process

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
        int main()
        {
                pid_tchild_pid;
                //create a child process
                child_pid=fork();
                if(child_pid>0)  {
                        //This is a parent process. Sleep for a minute
                        sleep(60)
                }
                else
                {
                        //This is a child process. Exit immediately.
                        exit(0);
                }
                return 0;
        }
```

Example 7

Demonstration of fork system call

```
#include<stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
        int main()
        {
                pid_tpid;
                char *msg;
                int n;
                printf("Program starts\n");
```

```
                pid=fork();
                switch(pid)
                {
                        case -1:
                                printf("Fork error\n");
                                exit(-1);
                        case 0:
                                msg="This is the child process";
                                n=5;
                                break;
                        default:
                                msg="This is the parent process";
                                n=3;
                                break;
                }
                while(n>0)
                {
                puts(msg);
                sleep(1);
                n--;
                }
                return 0;
}
```

Example 8

Demo of multiprocess application using fork()system call

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

        #define SIZE 1024
```

```
void do_child_proc(intpfd[2]);
void do_parent_proc(intpfd[2]);
int main()
{
        intpfd[2];
        intret_val,nread;
        pid_tpid;
        ret_val=pipe(pfd);
        if(ret_val==-1)
        {
                perror("pipe error\n");
                exit(ret_val);
        }
        pid=fork();
        switch(pid)
        {
                case -1:
                        printf("Fork error\n");
                        exit(pid);
                case 0:
                        do_child_proc(pfd);
                        exit(0);
                default:
                        do_parent_proc(pfd);
                        exit(pid);
        }
        wait(NULL);
        return 0;
        }
        void do_child_proc(intpfd[2])
        {
                intnread;
                char *buf=NULL;
                printf("5\n");
```

```
                close(pfd[1]);
                while(nread=(read(pfd[0],buf,size))!=0)
                        printf("Child Read=%s\n",buf);
                close(pfd[0]);
                exit(0);
        }
        void do_parent_proc(intpfd[2])
         {
                char ch;
                char *buf=NULL;
                close(pfd[0]);
                while(ch=getchar()!='\n')  {
                        printf("7\n");
                *buf=ch;
                buff+;
        }
        *buf='\0';
        write(pfd[1],buf,strlen(buf)+1);
        close(pfd[1]);
}
```

Assignment No. : 3

CPU Scheduling

ASSIGNMENT NO. 3

CPU SCHEDULING

AIM : Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.

OBJECTIVES : To study

- CPU Scheduling
- Round Robin

THEORY :

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:
Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

Step 5: Set the waiting time of the first process as _0' and its turnaround time as its burst time.

Step 6: Sort the processes names based on their Burst time

Step 7: For each process in the ready queue, calculate a) Waiting time(n)= waiting time (n-1) + Burst time (n-1) b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate c) Average waiting time = Total waiting Time / Number of process d)Average Turnaround time = Total Turnaround Time / Number of process

Step 9: Stop the process

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting

time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1) b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate c) Average waiting time = Total waiting Time / Number of process d) Average Turnaround time = Total Turnaround Time / Number ofprocess

Step 8: Stop the process

Oral Questions

1) Define the following a) Turnaround time b) Waiting time c) Burst time d) Arrival time

2) What is meant by process scheduling?

3) What are the various states of process?

4) What is the difference between preemptive and non-preemptive scheduling

5) What is meant by time slice?

6) What is round robin scheduling?

Assignment No. : 4 (A)

Thread Management using pthread Library

ASSIGNMENTNO: 4 (A)

THREAD SYNCHRONIZATION

AIM : Thread synchronization using counting semaphores and mutual exclusion using mutex. Application to demonstrate: Producer Consumer problem with counting semaphores and mutex.

OBJECTIVES : To study

- Semaphores
- Mutex
- Producer Consumer Problem

THEORY :

Semaphores:

Semaphore is an 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. It is known as a countingsemaphore or a general semaphore. Semaphores are the OS tools for synchronization.

Two types:

1. Binary Semaphore.
2. Counting Semaphore

Binary semaphore

Semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable / available) are called binary semaphores and are used to implement locks.

It is a means of suspending active processes which are later to be reactivated at such time conditions are right for it to continue. A binary semaphore is a pointer which when held by a process grants them exclusive use to their critical section. It is a (sort of) integer variable which can take the values 0 or 1 and be operated upon only by two commands termed in English wait and signal.

Counting semaphore

Semaphores which allow an arbitrary resource count are called counting semaphores.

A counting semaphore comprises:

An integer variable, initialized to a value K (K>=0). During operation it can assume any value <= K, a pointer to a process queue. The queue will hold the PCBs of all those processes, waiting to enter their critical sections. The queue is implemented as a FCFS, so that the waiting processes are served in a FCFS order.

A counting semaphore can be implemented as follows:

- Initialize – initialize to non negative integer
- Decrement (semWait)
    - Process executes this to receive a signal via semaphore.
    - If signal is not transmitted, process is suspended.
    - Decrements semaphore value
    - If value becomes negative , process is blocked
    - Otherwise it continues execution.
- Increment (semSignal)
    - Process executes it to transmit a signal via semaphore.
    - Increments semaphore value
    - If value is less than or equal to zero, process blocked by semWait is unblocked

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Value of semaphore

- Positive

  Indicates number of processes that can issue wait & immediately continue to execute.

- Zero

  By initialization or because number of processes equal to initial semaphore value have issued a wait Next process to issue a wait is blocked.

- Negative

  Indicates number of processes waiting to be unblocked

  Each signal unblocks one waiting process.

The Producer/Consumer Problem

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 into 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:

```
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 */;
}
```

Figure illustrates the structure of buffer b. The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the

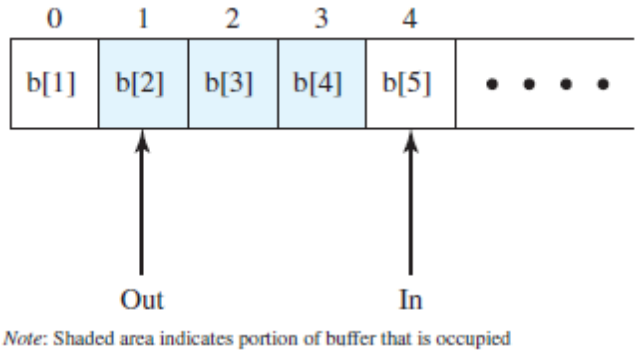


Figure: Infinite buffer for producer/consumer problem

Solution for bounded buffer using counting semaphore

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

POSIX Semaphores

POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem_post(3)); and decrement the semaphore value by one (sem_wait(3)). If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero.

Semaphore functions:

1. **sem_init()**

   It initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

   intsem_init(sem_t *sem, intpshared, unsigned int value);

2. **sem_wait()**

It decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement.

intsem_wait(sem_t *sem);

3. **sem_post()**

It increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

intsem_post(sem_t *sem);

1. **sem_unlink**

It removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

intsem_unlink(const char *name)

All the above functions returns

0 : Success

-1 : Error

Mutex

Mutexes are a method used to be sure two threads, including the parent thread, do not attempt to access shared resource at the same time. A mutex lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.

1. **pthread_mutex_init()**

The function shall initialize the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect shall be the same as passing

the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

intpthread_mutex_init(pthread_mutex_t *restrict mutex, cons pthread_mutexattr_t *restrict attr);

2. **pthread_mutex_lock()**

The mutex object referenced by mutex shall be locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

intpthread_mutex_lock(pthread_mutex_t * mutex);

3. **pthread_mutex_unlock()**

The function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread_mutex_unlock() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

intpthread_mutex_unlock(pthread_mutex_t * mutex);

4. **pthread_mutex_destroy()**

The function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using pthread_mutex_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

intpthread_mutex_destroy(pthread_mutex_t *mutex);

CONCLUSION:

Thus, we have implemented producer-consumer problem using ‘C’ in Linux.

FAQ

1. Explain the concept of semaphore?
2. Explain wait and signal functions associated with semaphores.
3. What is meant by binary and counting semaphores?

Assignment No. : 4 (B)

Thread Synchronization

ASSIGNMENTNO: 4 (B)

THREAD SYNCHRONIZATION

AIM : Thread synchronization and mutual exclusion using mutex. Application to demonstrate:
Reader-Writer problem with reader priority.

OBJECTIVES : To study

- Semaphores
- Mutex
- Producer Consumer Problem

THEORY :

Semaphores:

Semaphore is an 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. It is known as a countingsemaphore or a general semaphore. Semaphores are the OS tools for synchronization.
Two types:

1. Binary Semaphore.
2. Counting Semaphore

Binary semaphore

Semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable / available) are called binary semaphores and are used to implement locks.
It is a means of suspending active processes which are later to be reactivated at such time conditions are right for it to continue. A binary semaphore is a pointer which when held by a process grants them exclusive use to their critical section. It is a (sort of) integer variable which can take the values 0 or 1 and be operated upon only by two commands termed in English wait and signal.

Counting semaphore

Semaphores which allow an arbitrary resource count are called counting semaphores.
A counting semaphore comprises:

An integer variable, initialized to a value K (K>=0). During operation it can assume any value <= K, a pointer to a process queue. The queue will hold the PCBs of all those processes, waiting to enter their critical sections. The queue is implemented as a FCFS, so that the waiting processes are served in a FCFS order.

A counting semaphore can be implemented as follows:

- Initialize – initialize to non negative integer
- Decrement (semWait)
    - Process executes this to receive a signal via semaphore.
    - If signal is not transmitted, process is suspended.
    - Decrements semaphore value
    - If value becomes negative , process is blocked
    - Otherwise it continues execution.
- Increment (semSignal)
    - Process executes it to transmit a signal via semaphore.
    - Increments semaphore value
    - If value is less than or equal to zero, process blocked by semWait is unblocked

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Value of semaphore

- Positive

  Indicates number of processes that can issue wait & immediately continue to execute.

- Zero

  By initialization or because number of processes equal to initial semaphore value have issued a wait Next process to issue a wait is blocked.

- Negative

  Indicates number of processes waiting to be unblocked

  Each signal unblocks one waiting process.

The Reader Writer Problem

In dealing with the design of synchronization and concurrency mechanisms, it isuseful to be able to relate the problem at hand to known problems and to be ableto test any solution in terms of its ability to solve these known problems. The readers/writers problem is defined as follows: There is a data area sharedamong a number of processes. The data area could be a file, a block of main memory,or even a bank of processor registers. There are a number of processes thatonly

read the data area (readers) and a number that only write to the data area(writers). The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.

2. Only one writer at a time may write to the file.

3. If a writer is writing to the file, no reader may read it.

Thus, readers are processes that are not required to exclude one another andwriters are processes that are required to exclude all other processes, readers andwriters alike.

In the readers/writers problem readers do not also write to the data area, nor do writersread the data area while writing.

For example, suppose that the shared area is a library catalog.Ordinary users of the library read the catalog to locate a book. One or more librariansare able to update the catalog. In the general solution, every access to thecatalog would be treated as a critical section, and users would be forced to readthe catalog one at a time. This would clearly impose intolerable delays. At thesame time, it is important to prevent writers from interfering with each other andit is also required to prevent reading while writing is in progress to prevent the access
of inconsistent information.

Readers Have Priority

Figure is a solution using semaphores, showing one instance each of a readerand a writer; the solution does not change for multiple readers and writers.The writer process is simple. The semaphore wsem is used to enforce mutualexclusion. As long as one writer is accessing the shared data area, no other writersand no readers may access it. The reader process also makes use of wsem toenforce mutual exclusion. However, to allow multiple readers, we require that,when there are no readers reading, the first reader that attempts to read shouldwait on wsem. When there is already at least one reader reading, subsequentreaders need not wait before entering. The global variable readcount is used tokeep track of the number of readers, and the semaphore x is used to assure thatreadcount is updated properly.

```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true){
     semWait (x);
     readcount++;
     if(readcount == 1)
         semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
    }
}
void writer()
{
    while (true){
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}
```

POSIX Semaphores

POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem_post(3)); and decrement the semaphore value by one (sem_wait(3)). If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero.

Semaphore functions:

4. **sem_init()**

   It initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

   intsem_init(sem_t *sem, intpshared, unsigned int value);

5. **sem_wait()**

It decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement.

intsem_wait(sem_t *sem);

6. sem_post()

It increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

intsem_post(sem_t *sem);

2. sem_unlink

It removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

intsem_unlink(const char *name)

All the above functions returns

0 : Success

-1 : Error

Mutex

Mutexes are a method used to be sure two threads, including the parent thread, do not attempt to access shared resource at the same time. A mutex lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.

5. pthread_mutex_init()

The function shall initialize the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

intpthread_mutex_init(pthread_mutex_t *restrict mutex, cons pthread_mutexattr_t *restrict attr);

6. pthread_mutex_lock()

The mutex object referenced by mutex shall be locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

intpthread_mutex_lock(pthread_mutex_t * mutex);

7. **pthread_mutex_unlock()**

The function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread_mutex_unlock() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

intpthread_mutex_unlock(pthread_mutex_t * mutex);

8. **pthread_mutex_destroy()**

The function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using pthread_mutex_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

intpthread_mutex_destroy(pthread_mutex_t *mutex);

CONCLUSION:

Thus, we have implemented producer-consumer problem using ‘C’ in Linux.

FAQ

4. Explain the concept of semaphore?
5. Explain wait and signal functions associated with semaphores.
6. What is meant by binary and counting semaphores?

Assignment No. : 5

Deadlock avoidance algorithm

ASSIGNMENT NO: 5

DEADLOCK AVOIDANCE ALGORITHM

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

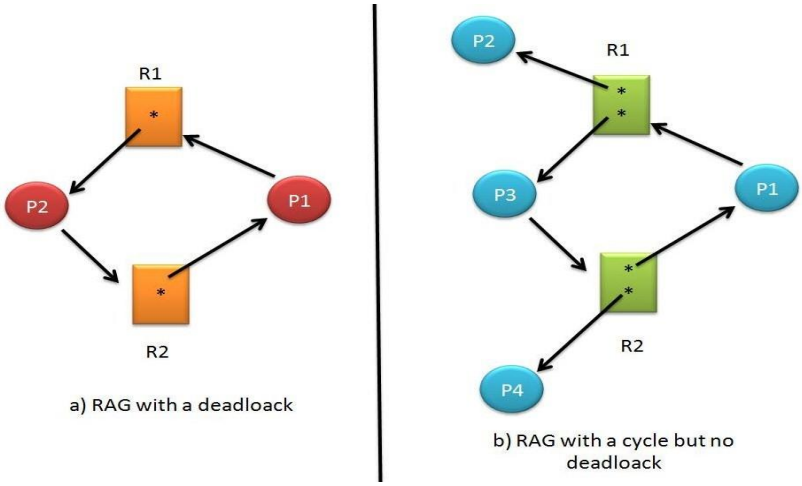
OBJECTIVES : To study

- Deadlock and starvation
- Conditions for Deadlock and its handling
- Banker Algorithm

THEORY :

What is Deadlock?

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no interrupts condition is needed to prevent an otherwise deadlocked process from being awake**.**



Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.

2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.

3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Consider a computer system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

What are the arguments for installing the deadlock-avoidance algorithm?

What are the arguments against installing the deadlock-avoidance algorithm?

An argument for installing deadlock avoidance in the system is that we could ensure deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run. An argument against installing deadlock avoidance software is that deadlocks occur infrequently and they cost little when they do occur. The time used for executing deadlock could be used to run additional jobs.

1. Recall the following example in class:
   - 5 processes are running in the system; *P*0 through *P*4.
   - They are using 3 resource types *A* (with 10 instances), *B* (with 5 instances), and *C* (with 7 instances).
   - At time T0, the snapshot of data structures maintained by the OS are as follows:

Process	Allocation Matrix	Max Matrix	Need Matrix	Available Vector
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

(a) Can request for (3,3,0) by P4 be granted?

Request (3, 3, 0) < Available (3, 3, 2), then continue;

Request (3, 3, 0) < Need[4] (4, 3, 1), then continue

Update the data structures as if the request is granted

Process	Allocation Matrix	Max Matrix	Need Matrix	Available Vector
	A B C	**A B C**	**A B C**	**A B C**
P0	**0 1 0**	**7 5 3**	**7 4 3**	**0 0 2**
P1	**2 0 0**	**3 2 2**	**1 2 2**	
P2	**3 0 2**	**9 0 2**	**6 0 0**	
P3	**2 1 1**	**2 2 2**	**0 1 1**	
P4	**3 3 2**	**4 3 3**	**1 0 1**	

Apply the safety algorithm:

Set Work (0, 0, 2) and Finish (F, F, F, F, F)

Search for a process that can terminate,

No process can have its needs (Need Matrix) met by the available resources (Available vector), that is, no Need[i] ≤ Work.

Terminate search.

Since Finish contains processes that have not terminated, the above state is unsafe and the request by P4 for the resources is not granted – P4 must wait.

(b) Can request for (0,2,0) by P0 be granted?

Request (0, 2, 0) < Available (3, 3, 2), then continue;
Request (0, 2, 0) < Need[0] (7, 4, 3), then continue
Update the data structures as if the request is granted

Process	Allocation Matrix	Max Matrix	Need Matrix	Available Vector
	A B C	A B C	A B C	A B C
P0	0 3 0	7 5 3	7 2 3	3 1 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

Apply the safety algorithm:

Set Work (3, 1, 2) and Finish (F, F, F, F, F)

Search for a process that can terminate,

P3 can terminate, Need[3] (0, 1, 1) ≤ Work (3, 1, 2)

Update Work to Work+Allocation[3]: (5, 2, 3), Set Finish[3] to true: Finish (F, F, F, T, F)

Search for a process that can terminate,

P1 can terminate, Need[1] (1, 2, 2) ≤ Work (5, 2, 3)
Update Work to Work+Allocation[1]: (7, 2, 3), Set Finish[1] to true: Finish (F, T, F, T, F)
Search for a process that can terminate,
P0 can terminate, Need[0] (7, 2, 3) ≤ Work (7, 2, 3)
Update Work to Work+Allocation[0]: (7, 5, 3), Set Finish[0] to true: Finish (T, T, F, T, F)
Search for a process that can terminate,
P2 can terminate, Need[2] (6, 0, 0) ≤ Work (7, 5, 3)
Update Work to Work+Allocation[2]: (10, 5, 5), Set Finish[2] to true: Finish (T, T, T, T, F)
Search for a process that can terminate,
P4 can terminate, Need[4] (4, 3, 1) ≤ Work (10, 5, 5)

Terminate Search.
Since Finish contains processes that have not terminated (no safe sequence), the above state is unsafe and the request by P0 for the resources is not granted – P0 must wait.

4. Given the following system state that defines how 4 types of resources are allocated to 5 running processes.

Available = {2, 1, 0, 0 }

$$\textbf{Allocation} = \begin{array}{c} \\ P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} \begin{array}{c} \begin{array}{cccc} r0 & r1 & r2 & r3 \end{array} \\ \begin{Bmatrix} 0 & 0 & 1 & 2 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 4 \\ 2 & 3 & 5 & 4 \\ 0 & 3 & 3 & 2 \end{Bmatrix} \end{array} \quad \textbf{Max} = \begin{Bmatrix} 0 & 0 & 1 & 2 \\ 2 & 7 & 5 & 0 \\ 6 & 6 & 5 & 6 \\ 4 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \end{Bmatrix}$$

a) Complete the Need matrix :

$$\textbf{Need} = \begin{Bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 6 & 6 & 2 & 2 \\ 2 & 0 & 0 & 2 \\ 0 & 3 & 2 & 0 \end{Bmatrix}$$

b) Is the system in a safe state? Why or why not? If in a safe state, give a safe sequence.
Apply the safety algorithm to see if in a safe state:
Initialize: Work = {2, 1, 0, 0 } Finish = {F, F, F, F,F}
Select P0 that does not need any additional resources
Work = {2,1,0,0}+{0,0,1,2}={2, 1, 1, 2 }
Finish = {T,F,F,F,F}
Select P3 that can request up to {2,0,0,2}
Work = {2,1,1,2}+{2,3,5,4} = {4,4,6,6}
Finish= {T,F,F,T,F}
Select P4 that can request up to {0,3,2,0}
Work = {4,4,6,6}+{0,3,3,2} = {4,7,9,8}
Finish= {T,F,F,T,T}
Select P1 that can request up to {0,7,5,0}

Work = {4,7,9,8}+{2,0,0,0} = {6,7,9,8}
Finish= {T,T,F,T,T}
Select P2 that can request up to {6,6,2,2}
Work = {6,7,9,8}+{0,0,3,4} = {6,7,12,12}
Finish= {T,T,T,T,T}
A safe sequence was found : <P0,P3,P4,P1,P2>

c) For each of the following requests:
   a. P0: *Request* = {0, 1, 0, 0}
   b. P1: *Request* = {0, 1, 0, 0}
   c. P2: *Request* = {0, 1, 0, 0}
   d. P3: *Request* = {0, 0, 0, 1}

Determine if the request should be granted. If it can be granted, show that the system is in a safe and give a safe sequence.

P0: *Request* = {0, 1, 0, 0}
Cannot grant request, P0 has been allocated maximum resources, i.e. *Request[i] ≤ Need[0,i] for all i* is not TRUE. An error is returned to P0.

P1: *Request* = {0,1,0,0}
Pretend to grant the request and update the allocation state:

Available = {2, 0, 0, 0 }

$$\textbf{Allocation} = \begin{array}{c} \\ P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} \begin{array}{c} \begin{array}{cccc} r0 & r1 & r2 & r3 \end{array} \\ \begin{Bmatrix} 0 & 0 & 1 & 2 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 3 & 4 \\ 2 & 3 & 5 & 4 \\ 0 & 3 & 3 & 2 \end{Bmatrix} \end{array} \qquad \textbf{Max} = \begin{Bmatrix} 0 & 0 & 1 & 2 \\ 2 & 7 & 5 & 0 \\ 6 & 6 & 5 & 6 \\ 4 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \end{Bmatrix}$$

$$\textbf{Need} = \begin{Bmatrix} 0 & 0 & 0 & 0 \\ 0 & 6 & 5 & 0 \\ 6 & 6 & 2 & 2 \\ 2 & 0 & 0 & 2 \\ 0 & 3 & 2 & 0 \end{Bmatrix}$$

Is projected state a safe state?

Initialize: Work = Available = {2, 0, 0, 0 }
Finish = {F, F, F, F, F}
Select P0 that does not need any additional resources
Work = {2,0,0,0}+{0,0,1,2}={2, 0, 1, 2 }
Finish = {T,F,F,F,F}
Select P3 that can request up to {2,0,0,2}
Work = {2, 0, 1, 2}+{2,3,5,4} = {4,3,6,6}
Finish= {T,F,F,T,F}
Select P4 that can request up to {0,3,2,0}
Work = {4,3,6,6}+{0,3,3,2} = {4,6,9,8}

Finish= {T,F,F,T,T}
Select P1 that can request up to {0,6,5,0}
Work = {4,6,9,8}+{2,1,0,0} = {6,7,9,8}
Finish= {T,T,F,T,T}
Select P2 that can request up to {6,6,2,2}
Work = {6,7,9,8}+{0,0,3,4} = {6,7,12,12}
Finish= {T,T,T,T,T}

A safe sequence was found : <P0,P3,P4,P1,P2>
P2: *Request* = {0, 1, 0, 0}
Pretend to update the request and update the matrices:

Available = {2, 0, 0, 0 }

$$\textbf{Allocation} = \begin{array}{c} \\ P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} \begin{array}{c} \begin{array}{cccc} r0 & r1 & r2 & r3 \end{array} \\ \begin{Bmatrix} 0 & 0 & 1 & 2 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 3 & 4 \\ 2 & 3 & 5 & 4 \\ 0 & 3 & 3 & 2 \end{Bmatrix} \end{array} \qquad \textbf{Max} = \begin{Bmatrix} 0 & 0 & 1 & 2 \\ 2 & 7 & 5 & 0 \\ 6 & 6 & 5 & 6 \\ 4 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \end{Bmatrix}$$

Complete the Need matrix :

$$\textbf{Need} = \begin{Bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 6 & 5 & 2 & 2 \\ 2 & 0 & 0 & 2 \\ 0 & 3 & 2 & 0 \end{Bmatrix}$$

Is projected state a safe state?

Initialize: Work = Available = {2, 0, 0, 0 }
Finish = {F, F, F, F, F}
Select P0 that does not need any additional resources
Work = {2,0,0,0}+{0,0,1,2}={2, 0, 1, 2 }
Finish = {T,F,F,F,F}
Select P3 that can request up to {2,0,0,2}
Work = {2, 0, 1, 2}+{2,3,5,4} = {4,3,6,6}
Finish= {T,F,F,T,F}
Select P4 that can request up to {0,3,2,0}
Work = {4,3,6,6}+{0,3,3,2} = {4,6,9,8}
Finish= {T,F,F,T,T}

Neither P1 nor P2 can be satisfied with the resources available in the Work vector. Thus no safe sequence can be found, and state is not safe – refuse the request.

P3: *Request* = {0, 0, 0, 1}
Cannot grant request, P3 requesting more resources than available, i.e. *Request[i] <= Available[i] for all i* is not true. P3 must wait.

5. Deadlock Detection

Given the following system resource allocation statete:

Available = { 2, 1, 0, 0 }

$$\textbf{Allocation} = \begin{Bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{Bmatrix} \qquad \textbf{Request} = \begin{matrix} P0 \\ P1 \\ P2 \end{matrix} \begin{Bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{Bmatrix}$$

a) Determine if a deadlock exists.

Initialize:
Work = {2,1,0,0}
Finish = {F,F,F}

Select P2 for termination, Work contains enough resources to satisfy P2.
Work = {2,1,0,0} + {0,1,2,0} = {2,2,2,0}
Finish = {F,F,T}
Select P1 for termination, Work contains enough resources to satisfy P1.
Work = {2,2,2,0} + {2,0,0,1} = {4,2,2,1}
Finish = {T,F,T}
Select P0 for termination, Work contains enough resources to satisfy P0.
Work = {4,2,2,1} + {0,0,1,0} = {4,2,3,1}
Finish = {T,T,T}

A safe sequence has been found <P2, P1, P0>, no deadlock exists.

b) Illustrate the system with a resource allocation graph.

CONCLUSION:

Thus, we have implemented dining philosopher's problem using 'C' in Linux.

FAQ:

1. What is dead lock?
2. What are the necessary and sufficient conditions to occur deadlock?
3. What is deadlock avoidance and deadlock prevention techniques?

Assignment No. : 6

Memory Management Page Replacement

ASSIGNMENT NO: 6

PAGE REPLACEMENT ALGORITHMS USING VIRTUAL MEMORY.

AIM : To implement page replacement algorithms using virtual memory.

OBJECTIVES : To study

- Virtual Memory Management
- Page replacement algorithms

In page replacement if there is no free page frame then we find out that is not currently used. Then we free that frame and the new page is loaded in that frame.
The complete process can be summarized into the following steps:

1) Find the location of the derived page on the disk.
2) Find a free frame.
    a) If there is a free frame use it.
    b) If there is no free frame use a page replacement algorithm to select a victim frame.
    c) Write the victim page to the disk, change the page and frame tables accordingly.
3) Read the desired page into the newly free frame, change the page of frame table.
4) Restart the user process.

ALGORITHM:

1) First in first out: FIFO
    a) Check if any frame is not used.
    b) If a frame is freeload the new page into the free space.
    c) Otherwise find the page, which has arrived first, and replace this page with newly arrived page.
2) Least recently used (LRU)
    a) Check if any frame is not used.
    b) If a frame is free, load the new page into the frame (free).
    c) Otherwise find the page that is least recently used, i.e. the page that has not been used for the longest period of time and replace this page with newly arrived page.

3) Optimal Page Replacement:
    a) Check if any frame is not used.
    b) If a frame is free, load the new page into the free frame.
    c) Otherwise replace the place that will not be used for longer period of time.

CONCLUSION:

Thus, we have implemented the page replacement algorithm using virtual memory.

Assignment No. : 7 (a)

Inter process communication using FIFO

ASSIGNMENT NO. 7 (a)

INTER PROCESS COMMUNICATION USING FIFO

AIM : 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.

OBJECTIVES : To study

- FIFO
- FIFO Operations
- Use of FIFO for inter process communication

THEORY :

PIPES:

Pipe provides an inter-process communication channel between related processes. The pipe interface is intended to look like a file interface. Although a pipe may seem like a file, it is not a file. Each write to the pipe fills as many blocks as are needed to satisfy it, provided that it does not exceed the maximum pipe size. Filled blocks are conveyed to the read-end of the pipe, where they are emptied when they are read. These types of pipes are called unnamed pipes because they do not exist anywhere in the file system.

Creating a pipe:

```
#include<unistd.h>
        int pipe(int filedes[2])
```

The system call pipe(fd), given an integer array filedes of size 2, creates a pair of file descriptors, filedes[0]and filedes[1], pointing to the "read-end" and "write-end" of a pipe respectively. If it is successful, it returns a 0, otherwise it returns -1.

Data can be written to the file descriptor fildes[1] and read from the file descriptor fildes[0]. A read on the file descriptor fildes[0] shall access data written to the file descriptor fildes[1] on a

first-in-first-out basis. If the parent wants to receive data from the child, it should close fildes[1], and the child should close fildes[0]. If the parent wants to send data to the child, it should close fildes[0], and the child should close fildes[1]. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.

Writing to pipe

The pipe has a limited size (64K in some systems) -- cannot write to the pipe infinitely.

ssize_t write(intfd, const void *buf, ssize_t count);

Writes upto count bytes to file referenced by file descriptor fd from buffer starting at buf.

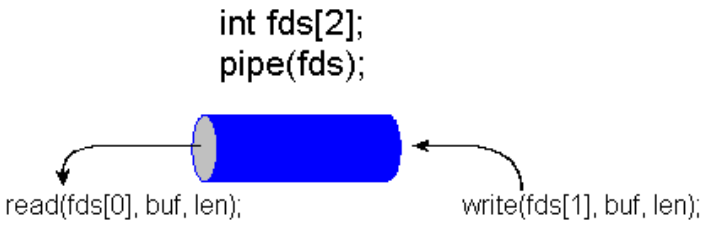
Reading from pipe

ssize_t read(intfd, void *buf, ssize_t count);

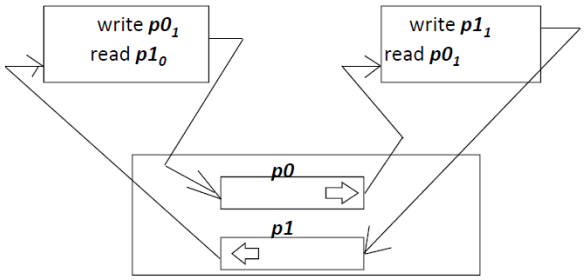
Reads upto count bytes from file descriptor fd into buffer starting at buf.

A UNIX pipe provides a half duplex communication.

The pipe(2) system call returns two file descriptors that form a "pipe", a one-way communication channel with a "read end" and a "write end".



A UNIX pipe provides a Full duplex communication via two pipes.
If we want two way communication generally we create two pipes one for each direction.
Create two separate pipes, say p0 and p1.



Example:

Using a Pipe to Pass Data between a Parent Process and a Child Process. The following example demonstrates the use of a pipe to transfer data between a parent process and a child process. Error handling is excluded, but otherwise this code demonstrates good practice when using pipes: after the fork() the two processes close the unused ends of the pipe before they commence transferring data.

```
#include <stdlib.h>
#include <unistd.h>
...
intfildes[2];
constint BSIZE = 100;
char buf[BSIZE];
ssize_tnbytes;
int status;

status = pipe(fildes);
if (status == -1 ) {
                                        /* an error occurred */
   ...
}
switch (fork())
 {
        case -1:                        /* Handle error */
```

```
                break;
        case 0:                                 /* Child - reads from pipe */
                close(fildes[1]);               /* Write end is unused */
                nbytes = read(fildes[0], buf, BSIZE); // Get data from
pipe

                /* At this point, a further read would see end of file ... */
                close(fildes[0]);               /* Finished with pipe */
                exit(EXIT_SUCCESS);


        default:                                /* Parent - writes to pipe */
                close(fildes[0]);       /* Read end is unused */
                write(fildes[1], "Hello world\n", 12);  // Write data on
pipe
                close(fildes[1]);               /* Child will see EOF */
                exit(EXIT_SUCCESS);
}
```

FIFOs

A FIFO (First In First Out) is a one-way flow of data. FIFOs have a name, so unrelated processes can share the FIFO. FIFO is a named pipe. Any process can open or close the FIFO. FIFOs are also called named pipes.

Properties:

1. After a FIFO is created, it can be opened for read or write.
2. Normally, opening a FIFO for read or write, it blocks until another process opens it for write or read.

c. A read gets as much data as it requests or as much data as the FIFO has, whichever is less.

d. A write to a FIFO is atomic, as long as the write does not exceed the capacity of the

FIFO.

e. FIFO must be opened by two processes; one opens it as reader on one end, the other opens it as sender on the other end. The first/earlier opener has to wait until the second/later opener to come. This is somewhat like a hand-shaking.

Creating a FIFO

A FIFO is created by the mkfifo function. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

```
#include <sys/types.h>
#include <sys/stat.h>
        intmkfifo(const char *pathname, mode_t mode);
```

pathname: a UNIX pathname (path and filename). The name of the FIFO

mode: the file permission bits. It specifies the pipe's owner, group, and world permissions, and a pipe must have a reader and a writer, the permissions must include both read and write permissions.

If the pipe cannot be created (for instance, if a file with that name already exists), mkfifo returns –1.

FIFO can also be created by the mknod system call,

e.g., mknod("fifo1", S_IFIFO|0666, 0) is same as mkfifo("fifo1", 0666).

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions like open, write, read, close or C library I/O functions (fopen, fprintf, fscanf, fclose, and soon) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
Intfd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

Close: To close an open FIFO, use close().

Unlink: To delete a created FIFO, use unlink().

CONCLUSION:
Thus, we studied inter process communication using FIFOs.

Assignment No. : 7 (b)

IPC using shared memory

ASSIGNMENT NO: 7 (b)

IPC USING SHARED MEMORY

AIM : Inter-process Communication using Shared Memory using ystem 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.

OBJECTIVES : To study

Linux kernel architecture

Shared memory

THEORY :

Shared Memory

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.

Objective:

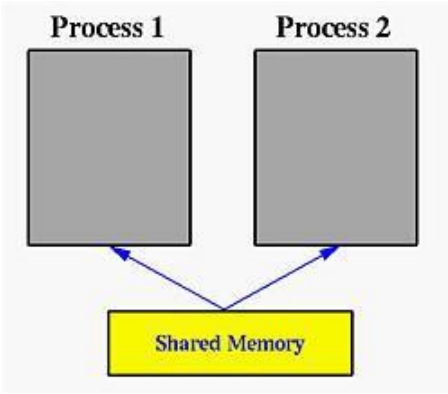
The aim of this laboratory is to show you how the processes can communicate among themselves using the Shared Memory regions. *Shared Memory* is an efficient means of passing data between programs. One program will create a memory portion, which other processes (if permitted) can access. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. In this lab the following issues related to shared memory utilization are discussed:

- Creating a Shared Memory Segment
- Controlling a Shared Memory Segment
- Attaching and Detaching a Shared Memory Segment

Introduction:

What is Shared Memory?

In the discussion of the fork ( ) system call, we mentioned that a parent and its children have separate address spaces. While this would provide a more secured way of executing parent and children processes (because they will not interfere each other), they shared nothing and have no way to communicate with each other. A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address space is "extended" by attaching this shared memory.



Shared memory is a feature supported by UNIX System V, including Linux, SunOS and Solaris. One process must explicitly ask for an area, using a key, to be shared by other processes. This process will be called the server. All other processes, the clients that know the shared area can access it. However, there is no protection to a shared memory and any process that knows it can access it freely. To protect a shared memory from being accessed at the same time by several processes, a synchronization protocol must be setup.

A shared memory segment is identified by a unique integer, the shared memory ID. The shared memory itself is described by a structure of type shmid_ds in header file sys/shm.h. To use this file, files sys/types.h and sys/ipc.h must be included. Therefore, your program should start with the following lines:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

A general scheme of using shared memory is the following:

For a server, it should be started before any client. The server should perform the following tasks:

1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call shmget( ).
2. Attach this shared memory to the server's address space with system call shmat( ).
3. Initialize the shared memory, if necessary.
4. Do something and wait for all client's completion.
5. Detach the shared memory with system call shmdt( ).
6. Remove the shared memory with system call shmctl( ).

For the client part, the procedure is almost the same:

1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
2. Attach this shared memory to the client's address space.
3. Use the memory.
4. Detach all shared memory segments, if necessary.
5. Exit.

Asking for a Shared Memory Segment - shmget( ) :

The system call that requests a shared memory segment is shmget( ). It is defined as follows:

```
shm_id = shmget (
                    key_t k,            /* the key for the segment */
                    int     size,       /* the size of the segment */
                    int     flag
                  );    /* create/use flag */
```

In the above definition, k is of type key_t or IPC_PRIVATE. It is the numeric key to be assigned to the returned shared memory segment. size is the size of the requested shared memory. The purpose of flag is to specify the way that the shared memory will be used.

For our purpose, only the following two values are important:

1. IPC_CREAT | 0666 for a server (i.e., creating and granting read and write access to the server)
2. 0666 for any client (i.e., granting read and write access to the client)

Note that due to UNIX's tradition, IPC_CREAT is correct and IPC_CREATE is not!!!

If shmget( ) can successfully get the requested shared memory, its function value is a non-negative integer, the shared memory ID; otherwise, the function value is negative. The following is a server example of requesting a private shared memory of four integers:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
.....
int  shm_id; /* shared memory ID */
.....
   shm_id = shmget (IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666); if
   (shm_id < 0)
   {
   printf("shmget error\n");
   exit(1);
   }
/* now the shared memory ID is stored in shm_id */
```

If a client wants to use a shared memory created with IPC_PRIVATE, it must be a child process of the server, created after the parent has obtained the shared memory, so that the private key value can be passed to the child when it is created. For a client, changing IPC_CREAT | 0666 to 0666 works fine. A warning to novice C programmers: don't change 0666 to 666. The leading 0 of an integer indicates that the integer is an octal number. Thus, 0666 is 10110110 in binary. If the leading zero is removed, the integer becomes six hundred sixty six with a binary representation 1111011010.

Server and clients can have a parent/client relationship or run as separate and unrelated processes. In the former case, if a shared memory is requested and attached prior to forking the child client process, then the server may want to use IPC_PRIVATE since the child receives an identical copy of the server's address space which includes the attached shared memory. However, if the server and clients are separate processes, using IPC_PRIVATE is unwise since the clients will not be able to request the same shared memory segment with a unique and unknown key.

Keys:

UNIX requires a key of type key_t defined in file sys/types.h for requesting resources such as shared memory segments, message queues and semaphores. A key is simply an integer of type key_t; however, you should not use int or long, since the length of a key is system dependent.

There are three different ways of using keys, namely:

1. a specific integer value (e.g., 123456)
2. a key generated with function ftok( )
3. a uniquely generated key using IPC_PRIVATE (i.e., a private key).

The first way is the easiest one; however, its use may be very risky since a process can access your resource as long as it uses the same key value to request that resource. The following example assigns 1234 to a key:

```
key_t SomeKey;
SomeKey =
1234;
```

The ftok( ) function has the following prototype:

```
key_t  ftok (
                const char *path,      /* a path string */
                int  id                /* an integer value */
                );
```

Function ftok( ) takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type key_t based on the first argument with the value of id in the most significant position. For example, if the generated integer is 35028A5D16 and the value of id is 'a' (ASCII value = 6116), then ftok( ) returns 61028A5D16. That is, 6116 replaces the first byte of 35028A5D16,generating 61028A5D16.

Thus, as long as processes use the same arguments to call ftok( ), the returned key value will always be the same. The most commonly used value for the first argument is ".", the current directory. If all related processes are stored in the same directory, the following call to ftok( ) **will generate the same key value:**

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t SomeKey;
SomeKey = ftok(".", 'x');
```

After obtaining a key value, it can be used in any place where a key is required. Moreover, the place where a key is required accepts a special parameter, IPC_PRIVATE. In this case, the system will generate a unique key and guarantee that no other process will have the same key. If a resource is requested with IPC_PRIVATE in a place where a key is required, that process will receive a unique key for that resource. Since that resource is identified with a unique key unknown to the outsiders, other processes will not be able to share that resource and, as a result, the requesting process is guaranteed that it owns and accesses that resource exclusively.

Attaching a Shared Memory Segment to an Address Space-shmat( ) :

Suppose process 1, a server, uses shmget( ) to request a shared memory segment successfully. That shared memory segment exists somewhere in the memory, but is not yet part of the address space of process 1. Similarly, if process 2 requests the same shared memory segment with the same key value, process 2 will be granted the right to use the shared memory segment; but it is not yet part of the address space of process 2. To make a requested shared memory segment part of the address space of a process, use shmat( ).

After a shared memory ID is returned, the next step is to attach it to the address space of a process. This is done with system call shmat( ). The use of shmat( ) is as follows:

```
shm_ptr = shmat (
                Int     shm_id,         /* shared memory D */
                char    *ptr,           /* a character pointer */
                Int     flag            /* access flag */
                );
```

System call shmat( ) accepts a shared memory ID, shm_id, and attaches the indicated shared memory to the program's address space. The returned value is a pointer of type (void *) to the attached shared memory. Thus, casting is usually necessary. If this call is unsuccessful, the return value is -1. Normally, the second parameter is NULL. If the flag is SHM_RDONLY, this shared memory is attached as a read-only memory; otherwise, it is readable and writable.
Note that the above code assumes the server and client programs are in the current directory. In order for the client to run correctly, the server must be started first and the client can only be started after the server has successfully obtained the shared memory.

Suppose process 1 and process 2 have successfully attached the shared memory segment. This shared memory segment will be part of their address space, although the actual address could be different (i.e., the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2).

Detaching and Removing a Shared Memory Segment - shmdt( ) and shmctl( ):

System call shmdt( ) is used to detach a shared memory. After a shared memory is detached, it cannot be used. However, it is still there and can be re-attached back to a process's address

space, perhaps at a different address. To remove a shared memory, use shmctl( ).

The only argument to shmdt( ) is the shared memory address returned by shmat( ). Thus, the following code detaches the shared memory from a program:

```
shmdt (shm_ptr);
```

where shm_ptr is the pointer to the shared memory. This pointer is returned by shmat( ) when the shared memory is attached. If the detach operation fails, the returned function value is non-zero.

To remove a shared memory segment, use the following code:

```
shmctl (shm_id, IPC_RMID, NULL);
```

where shm_id is the shared memory ID. IPC_RMID indicates this is a remove operation. Note that after the removal of a shared memory segment, if you want to use it again, you should use shmget( ) followed by shmat( ).

Assignment No. 8:
Implement the C program for Disk Scheduling Algorithms:

ASSIGNMENT NO: 8

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

OBJECTIVES : To study

Disk Structure

Disk Scheduling Algorithm

THEORY :

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

Overview

- **Over the past 30 years, the increase in the speed of processors and main memory has far outstripped that of**

disk access, with processor and main memory speeds increasing by about two orders of magnitude compared to one order of magnitude for disk.

- **The result is that disks are currently at least four orders of magnitude slower than main memory.**
- **This gap is expected to continue into the foreseeable future.**
- **Thus, the performance of disk storage subsystems is of vital concern, and much research has gone into schemes for improving that performance.**

Disk Performance Parameters

- **When the disk drive is operating, the disk is rotating at constant speed.**
- **To read or write, the head must be positioned at desired track and at the beginning of the desired sector on the track.**
- **Track selection involves moving the head in a moveable-head system.**
- **On a moveable-head system, the time it takes to position the head at the track is known as seek time.**
- **Once the track is selected, the desired controller waits until the appropriate sector rotates to line up with the head.**
- **The time it takes for the beginning of the sector to reach the head is known as rotational delay, or rotational latency.**
- **The sum of the seek time if any and the rotational delay is the access time, the time it takes to get into position to read or write.**
- **Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer position of the operation.**

Seek Time

- **Seek time is the time required to move the disk arm to the required track.**
- **The seek time consists of two key components: the initial startup time and the time taken to traverse the cylinders that have to be crossed once the access arm is up to speed.**
- **The seek time can be approximate with the following formula:**

$$Ts = m * n + s$$

Where, Ts = estimated seek time
n = number of tracks traversed
m = constant that depends on the disk drive
s = startup time

Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently.

- **For the disk drives, this means having a fast access time and disk bandwidth.**
- **The access time has two major components:**

    1. **the seek time and**
    2. **the rotational latency**

- **The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.**
- **We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.**
- **Whenever a process needs I/O to or from the disk, it issues a system call to the operating system.**
- **The request specifies several pieces of information:**

    1. **Whether this operation is input or output**
    2. **What the disk address for the transfer is**
    3. **What the memory address for the transfer is**
    4. **What the number of bytes to be transferred is**

- **If the desired disk drive and controller are available, the request can be serviced immediately.**
- **If the drive or controller is busy, any new requests for service will need to be placed on the queue of pending requests for that drive.**

First-Come First-Served Scheduling

- **The simplest form of disk scheduling is, of course, FCFS.**
- **This algorithm is intrinsically fair, but it generally does not provide the fastest service.**

Shortest-Seek-Time-First Scheduling

- **It seems reasonable to service all the requests close to the current head positions, before moving the head far away to service other requests.**
- **This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm.**
- **The SSTF algorithm selects the request with the minimum seek time from the current head position.**
- **Since the seek time increases with the number of cylinders traversed by the head SSTF chooses the pending request closest to current head position.**
- **SSTF scheduling is essentially a form of SJF scheduling, and, like SJF scheduling, it may cause starvation of some requests.**
- **That request may arrive at any time.**

SCAN Scheduling

- **In the SCAN algorithm, the disk arm starts at one end of the disk, and moves toward the other end of the disk.**
- **At the other end, the direction of head movement is reversed, and servicing continues.**
- **The head continuously scans back and forth across the disk.**
- **If the request arrives in the queue just in front of the head, it will be serviced almost immediately, a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.**
- **The SCAN algorithm is sometimes called the Elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up, and then reversing to service requests the other way.**

C-SCAN Scheduling

- **Circular SCAN (C-SCAN) is a variant of SCAN that is designed to provide a more uniform wait time.**
- **Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing the request along**

the way.

- **When the head reaches the other end, however it immediately returns to the beginning of the disk, without servicing any request on the return trip.**
- **The C-SCAN scheduling algorithm essentially treats the cylinder as a circular list that wraps around from the final cylinder to the first one.**

LOOK Scheduling

- **Notice that, as we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk.**
- **In practice, neither algorithm is implemented this way.**
- **More commonly, the arm goes only as far as the final request in each direction.**
- **Then, it reverses direction immediately, without first going all the way to the end of the disk.**
- **These versions of SCAN and C-SCAN are called LOOK and C-LOOK, because they look for a request before continuing to move in a given direction.**

Selection of a Disk-Scheduling Algorithm

- **Given so many disk-scheduling algorithms, how do we choose the best one?**
- **SSTF is common and has a natural appeal.**
- **SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to have the starvation problem.**
- **For any particular list of requests, it is possible to define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SSTF or SCAN.**
- **With any scheduling algorithm, however, performance depends heavily on the number and types of requests.**
- **For instance, suppose that the queue usually has just one outstanding request.**
- **Then, all scheduling algorithms are forced to behave the same, because they have only one choice for where to move the disk head.**
- **They all behave like FCFS scheduling.**

- **Note that the requests for disk service can be greatly influenced by the file-allocation method.**
- **A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement.**
- **A linked or indexed file, on the other hand, may include blocks that are widely scattered on the disk, resulting in greater head movement.**
- **The location of directories and index blocks also is important.**
- **Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently.**
- **Suppose a directory entry is on the first cylinder and a file's data are on the final cylinder.**
- **In this case, the disk head has to move the entire width of the disk.**
- **If the directory entry were on the middle cylinder, the head has to move at most one-half the width.**
- **Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.**
- **Because of these complexities, the disk-scheduling algorithm should be written as a separate module of the operating system, so that it can be replaced with a different algorithm if necessary.**
- **Either SSTF or LOOK is a reasonable choice for the default algorithm.**
- **Note that the scheduling algorithms described earlier consider only the seek distances.**
- **For modern disks, the rotational latency can be nearly as large as the average seek time.**
- **But it is difficult for the operating system to schedule for improved rotational latency because modern disks do not disclose the physical location of logical blocks.**
- **Disk manufacturers have been helping with this problem by implementing disk-scheduling algorithms in the controller hardware built into the disk drive.**
- **If the operating system sends a batch of requests to the controller, the controller can queue them and then schedule them to improve both the seek time and the rotational latency.**
- **If I/O performance were the only consideration, the operating system would gladly turn over the responsibility of disk scheduling to the disk**

hardware.

- **In practice, however, the operating system may have other constraints on the service order for requests.**
- **For instance, demand paging may take priority over application I/O, and writes are more urgent than reads if the cache is running out of free pages.**
- **Also, it may be desirable to guarantee the order of a set of disk writes to make the file system robust in the face of system crashes; consider what could happen if the operating system allocated a disk page to a file, and the application wrote data into that page before the operating system had a chance to flush the modified inode and free-space list back to disk.**

To accommodate such requirements, an operating system may choose to do its own disk scheduling, and to "spoon-feed" the requests to the disk controller, one by one.

Assignment No. : 9

Implementation and addition of new system call

ASSIGNMENT NO: 9

STUDY ASSIGNMENT OF IMPLEMENTATION AND ADDITION OF A NEW SYSTEM CALL

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.

OBJECTIVES : To study

Linux kernel architecture

System call

THEORY :

Steps

Following are the steps to add a new system call in Linux.

1. Change to the kernel source directory using,

   cd /usr/src/linux-3.17.7/

2. Define a new system call sys_hello()

   Create a directory hello in the kernel source directory: -

   mkdir hello

   Change into this directory

   cd hello

3. Create a “hello.c” file in this folder and add the definition of the system call to it as given below

   gedithello.c

Add the following code:

```
#include<linux/kernel.h>
        asmlinkage long sys_hello(void)
        {
                printk("Hello world\n");
                return 0;
        }
```

Note that printk prints to the kernel's log file.

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

        geditMakefile

Add the following line to it:-

        obj-y:=hello.o

This is to ensure that the hello.c file is compiled and included in the kernel source code.

5. Add the hello directory to the kernel's Makefile

Change back into the linux-3.17.7 folder and open Makefile

        geditMakefile

Go to 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 present in the hello directory.

6. Add the new system call (sys_hello()) into the system call table (syscall_32.tblfile)

If your system is a 64 bit system, you will need to alter the syscall_64.tblfile.

        cd arch/x86/syscalls

        gedit syscall_32.tbl

Add the following line at 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 user space program.

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

cd include/linux/

geditsyscalls.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 keyword used to indicate that all parameters of the function would be available on the stack.

8. 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

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 popup window with the list of menus and you can select the items for the new configuration. If you run familiar 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-3.17.7/

make

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

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 3.17.7 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.

- System.map-3.16.0
- vmlinuz-3.16.0
- initrd.img-3.16.0
- 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:

```
#include<stdio.h>
#include<linux/kernel.h>
#include<sys/syscall.h>
#include<unistd.h>
        int main()
        {
                slongintmycall =syscall(354);
```

```
        printf("System call sys_hello returned %ld\n", mycall);
        return0;
    }
```

Now compile this program using the following command.

gccuserspace.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.

Say, we wanted to add our own version of the system call getpid(). Let's call our version mygetpid(). The implementation of mygetpid() is:

```
asmlinkage long sys_getpid(void)
    {
    return current->tgid;
    }
```

NOTE: asmlinkage must appear before every system call.

It tells the compiler to only look on the stack for the functions arguments (aka compiler magic).