



DHOLE PATIL EDUCATION SOCIETY'S

DHOLE PATIL COLLEGE OF ENGINEERING

Accredited by NAAC with A+ Grade, An ISO 9001:2015 Certified Institute,

1284, Near Eon IT Park Kharadi, Dhole Patil College Road, Wagholi, Pune-412207

Website: www.dpcoepune.edu.in E-mail: dpcoepune@gmail.com, Phone:020-66059900

DEPARTMENT OF INFORMATION TECHNOLOGY

LAB MANUAL

OPERATING SYSTEM

CLASS- TE

(2019 PATTERN)

SUBJECT INCHARGE: PROF.PRIYANKA FULSAUNDAR

Assignment No. : 1

Shell programming

ASSIGNMENTNO. 1

SHELL PROGRAMMING

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

OBJECTIVES : To study

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), then 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 return 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 file1 does not exist on your hard drive)

```
$ rm file1
```

It will show error as follows

```
rm: cannot remove `file1': 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 Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			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 assigned 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.

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

OBJECTIVES : To study

1. Process control
2. Zombie and orphan processes
3. System calls : fork, execve, wait

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 child. If an error occurs, these functions return -1.

Example:

```
pid_t childpid;  
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. `int execl(const char *path, const char *arg0, ... /*, char *(0) */);`
2. `int execl (const char *path, const char *arg0, ... /*, char *(0), char *constenvp[] */);`
3. `int execlp (const char *file, const char *arg0, ... /*, char *(0) */);`
4. `int execv(const char *path, char *constargv[]);`
5. `int execve (const char *path, char *constargv[], char *constenvp[]);`
6. `int execvp (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/lis", "lis", "-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("lis", "lis", "-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[] = {"lis", "-l", NULL};`
`execv("/bin/lis", 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("lis", 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: `#!/path/to/interpreter`. `argv` is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. `envp` is an array of strings, conventionally of the form `key=value`, which are passed as environment to the new program. Both `argv` and `envp` must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

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

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

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 `exit` function, or the program's main function returns. Each process has an exit code: a number 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

List of files in current directory Using the system call

```
#include <stdlib.h>

int main()
{
    Int return_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 main()
{
    pid_t childpid;
    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_t childpid;
    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_t child_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_t pid;
    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])
    {
        int nread;
        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 Algorithms

ASSIGNMENTNO: 3

CPU SCHEDULING

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

Objective: To study : CPU scheduling Algorithm

1. Shortest Job First
2. Round -Robin

Theory:

In multiprogramming CPU scheduling is necessary to have maximum utilization/ throughput of the processor so that multiple processes can handle efficiently without leaving the CPU idle.

It Selects one of the processes in ready queue to be executed, and allocates the CPU to one of them. CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

CPU Scheduling Algorithms: Scheduling of processes/work is done to finish the work on time. Below are different times with respect to a process.

Arrival Time : Time at which the process arrives in the ready queue.

Completion Time : Time at which process completes its execution.

Burst Time : Time required by a process for CPU execution.

Turn Around Time : Time Difference between completion time and arrival time. Turn Around Time = Completion Time - Arrival Time

Waiting Time(W.T) : Time Difference between turn around time and burst time. Waiting Time = Turn Around Time - Burst Time

Why do we need scheduling? A typical process involves both I/O time and CPU time. In a uni-programming system like MSDOS, time spent waiting for I/O is wasted and CPU is free during this time. In multiprogramming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

Objectives of Process Scheduling Algorithm

- Max CPU utilization [Keep CPU as busy as possible]
- Fair allocation of CPU.
- Max throughput [Number of processes that complete their execution per time unit]
- Min turnaround time [Time taken by a process to finish execution]
- Min waiting time [Time a process waits in ready queue]
- Min response time [Time when a process produces first response]

Different Scheduling Algorithms: There are two main strategies used for CPU scheduling

- Non preemptive:- Once the CPU has been allocated to a process, it keeps the CPU until process terminates or by switching to the wait state. Eg. MS Windows 3.x
- Preemptive:- The process can be preempted when a higher priority process is ready for the execution. Eg. Windows 95, Mac OS

First Come First Serve (FCFS): Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non – preemptive scheduling algorithm. Note: First come first serve suffers from convoy effect.

Shortest Job First (SJF): Process which has the shortest burst time is scheduled first. If two processes have the same bust time then FCFS is used to break the tie. It is a non-preemptive scheduling algorithm.

Longest Job First (LJF): It is similar to SJF scheduling algorithm. But, in this scheduling algorithm, we give priority to the process having the longest burst time. This is non – preemptive in nature i.e., when any process starts executing, can't be interrupted before complete execution.

Shortest Remaining Time First (SRTF): It is preemptive mode of SJF algorithm in which jobs are schedule according to shortest remaining time. **Longest Remaining Time First (LRTF):** It is preemptive mode of LJF algorithm in which we give priority to the process having largest burst time remaining.

Round Robin Scheduling: Each process is assigned a fixed time (Time Quantum/Time Slice) in cyclic way. It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Priority Based scheduling (Non - Preemptive): In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is scheduled first. If priorities of two processes match, then schedule according to arrival time. Here starvation of process is possible.

Highest Response Ratio Next (HRRN) In this scheduling, processes with highest response ratio is scheduled. This algorithm avoids starvation.

$$\text{Response Ratio} = (\text{Waiting Time} + \text{Burst time}) / \text{Burst time}$$

Multilevel Queue Scheduling: According to the priority of process, processes are placed in the different queues. Generally high priority process is placed in the top level queue. Only after

completion of processes from top level queue, lower level queued processes are scheduled. It can suffer from starvation.

Multi level Feedback Queue Scheduling: It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue.

CONCLUSION:

Thus, we have implemented preemptive Shortest Path algorithm and Round Robin algorithm with 'C' in Linux.

Questions

- What is the objective/criteria for CPU scheduling
- Define preemptive and non-preemptive CPU scheduling
- Calculate the average waiting time and turnaround time using SJF algorithm for following processes. Consider all the processes are arriving at the same time.

Process ID	Burst time
P1	5
P2	4
P3	1
P4	2

- Calculate the average waiting time and turnaround time using Round -Robin algorithm for following processes. Consider all the processes are arriving at the same time.

Process ID	Burst time
P1	8
P2	2
P3	7
P4	3
P5	2

Assignment No. : 4

Thread Synchronization

ASSIGNMENTNO: 4

THREAD SYNCHRONIZATION

AIM : Thread synchronization using counting semaphores. 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 counting semaphore 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 ($K \geq 0$). During operation it can assume any value $\leq K$, 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

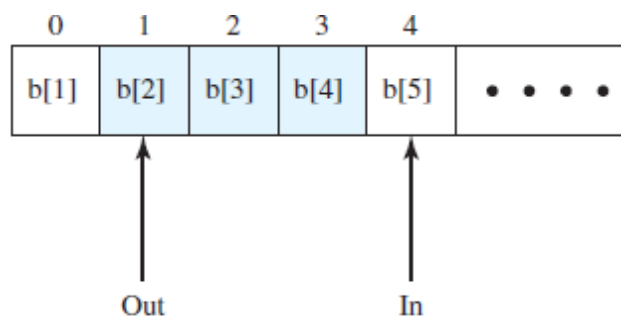
There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer 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



Note: Shaded area indicates portion of buffer that is occupied

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, const 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.

```
int pthread_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.

```
int pthread_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.

```
int pthread_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?

ASSIGNMENTNO: 4B

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 counting semaphore 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 ($K \geq 0$). During operation it can assume any value $\leq K$, 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 is useful to be able to relate the problem at hand to known problems and to be able to 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 shared among 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 that only 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 and writers are processes that are required to exclude all other processes, readers and writers alike.

In the readers/writers problem readers do not also write to the data area, nor do writers read 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 librarians are able to update the catalog. In the general solution, every access to the catalog would be treated as a critical section, and users would be forced to read the catalog one at a time. This would clearly impose intolerable delays. At the same time, it is important to prevent writers from interfering with each other and it 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 reader and a writer; the solution does not change for multiple readers and writers. The writer process is simple. The semaphore `wsem` is used to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader process also makes use of `wsem` to enforce mutual exclusion. However, to allow multiple readers, we require that, when there are no readers reading, the first reader that attempts to read should wait on `wsem`. When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable `readcount` is used to keep track of the number of readers, and the semaphore `x` is used to assure that `readcount` 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, const 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. : 6

Deadlock avoidance using semaphores

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 :

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.
2. A read gets as much data as it requests or as much data as the FIFO has, whichever is less.
3. A write to a FIFO is atomic, as long as the write does not exceed the capacity of the FIFO.
4. 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. : 07B

Implementation of Shared Memory

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

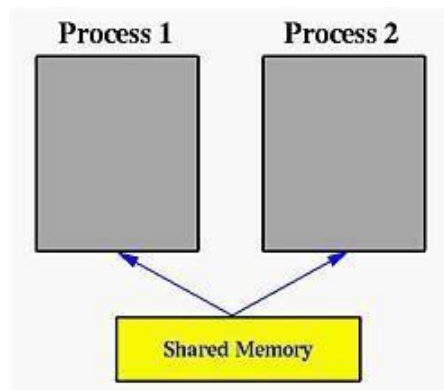
☀ Controlling a Shared Memory Segment

☀ 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 ID */
                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()**.

Study Assignment

Implementation and addition of new system call

Study Assignment

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

```
gedit hello.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.

```
gedit Makefile
```

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

```
gedit Makefile
```

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.

```
asm linkage long sys_hello(void);
```

This defines the prototype of the function of our system call. "asm linkage" 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 an 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).