UNIT-I

Syllabus

Linux Utilities-File handling utilities, Security by file permissions, Process utilities, Disk utilities, Networking commands, Filters, Text processing utilities and Backup utilities, sed – scripts, operation, addresses, commands, applications, awk – execution, fields and records, scripts, operation, patterns, actions, functions, using system commands in awk.(http://www.grymoire.com/Unix/Awk.html)

Working with the Bourne again shell(bash): Introduction, shell responsibilities, pipes and input Redirection, output redirection, here documents, running a shell script, the shell as a programming language, shell meta characters, file name substitution, shell variables, command substitution, shell commands, the environment, quoting, test command, control structures, arithmetic in shell, shell script examples, interrupt processing, functions, debugging shell scripts.

History of UNIX:

In 1969-1970, Kenneth Thompson, Dennis Ritchie, and others at AT&T Bell Labs began developing a small operating system on a little-used PDP-7. The operating system was soon christened Unix, a pun on an earlier operating system project called MULTICS. In 1972-1973 the system was rewritten in the programming language C, an unusual step that was visionary: due to this decision, Unix was the first widely-used operating system that could switch from and outlive its original hardware. Other innovations were added to Unix as well, in part due to synergies between Bell Labs and the academic community. In 1979, the ``seventh edition" (V7) version of Unix was released, the grandfather of all extant Unix systems.

History of Linux:

In 1991 Linus Torvalds began developing an operating system kernel, which he named ``Linux" [Torvalds 1999]. This kernelcould be combined with the FSF material and other components (in particular some of the BSD components and MIT's X-windows software) to produce a freely-modifiable and very useful operating system. This book will term the kernel itself the ``Linux kernel" and an entire combination as ``Linux". Note that many use theterm ``GNU/Linux" instead for this combination.

Comparing Linux and UNIX

Originally, the term ``Unix" meant a particular product developed by AT&T. Today, the Open Group owns the Unix trademark, and it defines Unix as ``the worldwide Single UNIX Specification".Linux is not derived from Unix source code, but its interfaces are intentionally like Unix. Therefore, Unix lessons learned generally apply to both, including information on security. Most of the information in this book applies to any Unix-like system. Linux-specific information has been intentionally added to enable those using Linux to take advantage of Linux's capabilities.

Unix-like systems share a number of security mechanisms, though there are subtle differences and not all systems have all mechanisms available. All include user and group ids (uids and gids) for each process and a file system with read, write, and execute permissions (for user, group, and other). See Thompson [1974] and Bach [1986] for general information on Unix systems, including their basic security mechanisms.

Features of Linux:

Portable – Portability means software's can works on different types of hardware's in same way. Linux kernel and application programs supports their installation on any kind of hardwareplatform.
Open Source – Linux source code is freely available and it is community based development project. Multiple teams works in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
Multi-User – Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.
Multiprogramming – Linux is a multiprogramming system means multiple applications can run at same time.
Hierarchical File System – Linux provides a standard file structure in which system files/ user files are arranged.
Shell – Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs etc.
Security – Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Help facility– use man utility to take complete description of any utility which means provides help facility to all commands as various manuals.

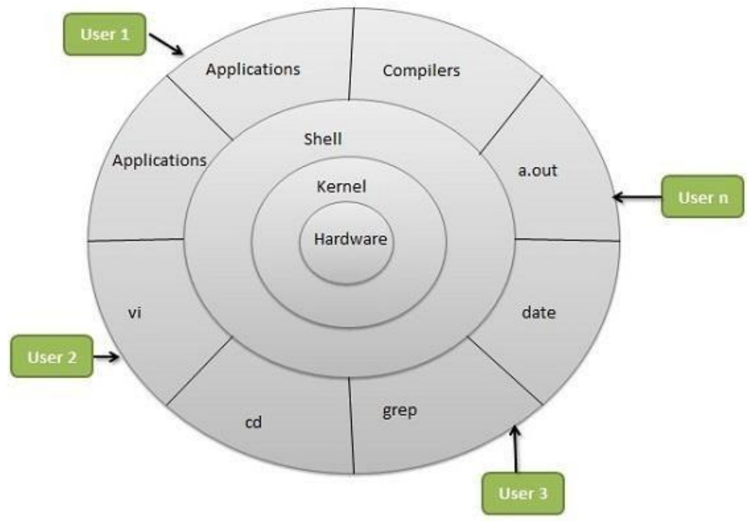
Faculty tolerance – automatically reset the failure of Hardware

Communication- Provides various interprocess communication utilities and system calls for sending and receiving data among the process created by users.

Virtual memory- using paging (not swapping whole processes) to disk: to a separate partition or a file in the file system, or both, with the possibility of adding more swapping areas during runtime

Architecture of UNIX

Architecture. Hardware layer - Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc). Kernel - Core component of **Operating System**, interacts directly with hardware, provides low level services to upper layer components. Shell - An interface to kernel, hiding complexity of kernel's functions from users.



Kernel is also called as the heart of the Operating System and the Every Operation is performed by using the Kernel , When the Kernel Receives the Request from the Shell then this will Process the Request and Display the Results on the Screen. The various Types of Operations those are Performed by the Kernel are as followings:-

1) It Controls the State the Process Means it checks whether the Process is running or Process is Waiting for the Request of the user.

2) Provides the Memory for the Processes those are Running on the System Means Kernel Runs the Allocation and De-allocation Process , First When we Request for the service then the Kernel

will Provides the Memory to the Process and after that he also Release the Memory which is Given to a Process.

3) The Kernel also Maintains a Time table for all the Processes those are Running Means the Kernel also Prepare the Schedule Time means this will Provide the Time to various Process of the CPU and the Kernel also Puts the Waiting and Suspended Jobs into the different Memory Area.

4) When a Kernel determines that the Logical Memory doesn't fit to Store the Programs. Then he uses the Concept of the Physical Memory which Will Stores the Programs into Temporary Manner. Means the Physical Memory of the System can be used as Temporary Memory.

5) Kernel also maintains all the files those are Stored into the Computer System and the Kernel Also Stores all the Files into the System as no one can read or Write the Files without any Permissions. So that the Kernel System also Provides us the Facility to use the Passwords and also all the Files are Stored into the Particular Manner.

SHELL:

The **shell** acts as an interface between the user and the **kernel**. When a user logs in, the login program checks the username and password, and then starts another program called the **shell**. The **shell** is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out.

Types of Different Shells

Name of shell	**Command name**	**Description**
C Shell	csh	Similar to the C programming language in syntax
Bash Shell	bash	Bourne Again Shell combines the advantages of the Korn Shell and the C Shell. The default on most Linux distributions.
tcsh	tcsh	Similar to the C Shell

Shell script is a program written in a shell programming language (e.g., bash, csh, ksh or sh) that allows users to issue a single command to execute any combination of commands, including those with options and/or arguments, together with redirection. Shell scripts are well suited for automating simple tasks and creating custom-made filters.

Shell Responsibilities:

- Command line interpreter
- Program Execution
- Meta characters
- Variable and Filename Substitution

- I/O Redirection
- Pipeline Hookup
- Environment Control
- Interpreted Programming Language

Command or Utility:

A shell is a program that reads commands that are typed on a keyboard and then executes (i.e., runs) them. Shells are the most basic method for a user to interact with the system. Every Unix-like operating system has at least one shell, and most have several. The default shell on most Linux systems is bash.

A command is an instruction given by a user telling a computer to do something, such a run a single program or a group of linked programs. Commands are generally issued by typing them in at the command line (i.e., the all-text display mode) and then pressing the ENTER key, which passes them to the shell.

Types of Utilities:

i) Basic Commads: echo,cal,date,man,pwd,man,who,whoami,bc

ii)File handling Utilities: ls,cat,rm,more,mv,cd,cp,touch,wc

iii) Security related utilities: chmod,chown,

iv) Process Utilities: ps,kill,bg,ipcs

v) Disk Utilities: du,df

vi) Networking commands: finger,arp,telnet,rlogin,ftp,chfn

vii) Filtering commands: head,tail,grep,find,sort,cp,uniq,tr,tee

viii) Text processing commands: cut,paste,join,

ix) Comparison commands: diff,cmp,comp

x) Backup commands: tar,zip,gzip,unzip,gunzip,cpio

xi) Link commands: ln, ln-s,unlink

i) Basic Commands: echo, cal, date, man, pwd, man, who, whoami, bc

NAME
echo - display a line of text
SYNOPSIS
echo [OPTION]... [STRING]...
DESCRIPTION
Echo the STRING(s) to standard output.

Tag	Description
-n	do not output the trailing newline
-e	enable interpretation of backslash escapes
-E	disable interpretation of backslash escapes (default)

```
$ cal
   April 2016
Su Mo Tu We Th Fr Sa
                1  2
3 4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

ii) File handling Utilities: ls,cat,rm,more,mv,cd,cp,touch,wc

Listing Files

To list the files and directories stored in the current directory. Use the following command −

```
$ls
```

Here is the sample output of the above command −

```
$ls


bin        hosts  lib     res.03
ch07       hw1    pub     test_results
ch07.bak   hw2    res.01 users
docs       hw3    res.02  work
```

The command **ls** supports the **-l** option which would help you to get more information about the listed files −

```
$ls -l
total 1962188
drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
drwxr-xr-x 11 amrood amrood      4096 May 29  2007 zlib-1.2.3
```

Here is the information about all the listed columns −

- First Column: represents file type and permission given on the file. Below is the description of all type of files.
- Second Column: represents the number of memory blocks taken by the file or directory.
- Third Column: represents owner of the file. This is the Unix user who created this file.
- Fourth Column: represents group of the owner. Every Unix user would have an associated group.
- Fifth Column: represents file size in bytes.
- Sixth Column: represents date and time when this file was created or modified last time.
- Seventh Column: represents file or directory name.

In the ls -l listing example, every file line began with a d, -, or l. These characters indicate the type of file that's listed.

Prefix	**Description**
-	Regular file, such as an ASCII text file, binary executable, or hard link.

b	Block special file. Block input/output device file such as a physical hard drive.
c	Character special file. Raw input/output device file such as a physical hard drive
d	Directory file that contains a listing of other files and directories.
l	Symbolic link file. Links on any regular file.
p	Named pipe. A mechanism for interprocess communications
s	Socket used for interprocess communication.

Meta Characters

Meta characters have special meaning in Unix. For example ***** and **?** are metacharacters. We use ***** to match 0 or more characters, a question mark **?**matches with single character.

For Example −

```
$ls ch*.doc
```

Displays all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc c
```

Here ***** works as meta character which matches with any character. If you want to display all the files ending with just **.doc** then you can use following command −

```
$ls *.doc
```

Hidden Files

An invisible file is one whose first character is the dot or period character (.). UNIX programs (including the shell) use most of these files to store configuration information.

Some common examples of hidden files include the files −

- **.profile** − the Bourne shell (sh) initialization script
- **.kshrc** − the Korn shell (ksh) initialization script
- **.cshrc** − the C shell (csh) initialization script
- **.rhosts** − the remote shell configuration file

To list invisible files, specify the -a option to ls −

```
$ ls -a


.        .profile      docs     lib     test_results
..       .rhosts       hosts    pub     users
.emacs    bin           hw1      res.01  work
.exrc     ch07          hw2      res.02
.kshrc    ch07.bak      hw3      res.03
$
```

- Single dot **.** − This represents current directory.
- Double dot **..** − This represents parent directory.

Creating Files

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

```
$ vi filename
```

Above command would open a file with the given filename. You would need to press key **i** to come into edit mode. Once you are in edit mode you can start writing your content in the file as below −

```
This is unix file....I created it for the first time.....

I'm going to save this content in this file.
```

Once you are done, do the following steps −

- Press key **esc** to come out of edit mode.
- Press two keys **Shift + ZZ** together to come out of the file completely.

Now you would have a file created with **filename** in the current directory.

```
$ vi filename

$
```

Editing Files

You can edit an existing file using **vi** editor. We would cover this in detail in a separate tutorial. But in short, you can open existing file as follows −

```
$ vi filename
```

Once file is opened, you can come in edit mode by pressing key **i** and then you can edit file as you like. If you want to move here and there inside a file then first you need to come out of edit mode by pressing key **esc** and then you can use following keys to move inside a file −

- **l** key to move to the right side.
- **h** key to move to the left side.
- **k** key to move up side in the file.
- **j** key to move down side in the file.

So using above keys you can position your cursor where ever you want to edit. Once you are positioned then you can use **i** key to come in edit mode. Edit the file, once you are done press **esc** and finally two keys **Shift + ZZ** together to come out of the file completely.

Display Content of a File

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

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

You can display line numbers by using **-b** option along with **cat** command as follows −

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

Counting Words in a File

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

```
$ wc filename
2  19 103 filename
$
```

Here is the detail of all the four columns −

- First Column: represents total number of lines in the file.
- Second Column: represents total number of words in the file.

- Third Column: represents total number of bytes in the file. This is actual size of the file.
- Fourth Column: represents file name.

You can give multiple files at a time to get the information about those file. Here is simple syntax −

```
$ wc filename1 filename2 filename3
```

Copying Files:

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

```
$ cp source_file destination_file
```

Following is the example to create a copy of existing file **filename**.

```
$ cp filename copyfile

$
```

Now you would find one more file **copyfile** in your current directory. This file would be exactly same as original file **filename**.

Renaming Files

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

```
$ mv old_file new_file
```

Following is the example which would rename existing file **filename** to**newfile**:

```
$ mv filename newfile

$
```

The **mv** command would move existing file completely into new file. So in this case you would fine only **newfile** in your current directory.

Deleting Files

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

```
$ rm filename
```

Caution: It may be dangerous to delete a file because it may contain useful information. So be careful while using this command. It is recommended to use**-i** option along with **rm** command.

Following is the example which would completely remove existing file**filename**:

```
$ rm filename

$
```

You can remove multiple files at a tile as follows −

```
$ rm filename1 filename2 filename3

$
```

Standard Unix Streams

Under normal circumstances every Unix program has three streams (files) opened for it when it starts up −

- **stdin** − This is referred to as standard input and associated file descriptor is 0. This is also represented as STDIN. Unix program would read default input from STDIN.
- **stdout** − This is referred to as standard output and associated file descriptor is 1. This is also represented as STDOUT. Unix program would write default output at STDOUT
- **stderr** − This is referred to as standard error and associated file descriptor is 2. This is also represented as STDERR. Unix program would write all the error message at STDERR.

iii) Security related utilities: chmod,chown,

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes −

- **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

While using **ls -l** command it displays various information related to file permission as follows −

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile
drwxr-xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

Here first column represents different access mode ie. permission associated with a file or directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

- The first three characters (2-4) represent the permissions for the file's owner. For example -rwxr-x**r--** represents that owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example -rwxr-x**r--**represents that group has read (r) and execute (x) permission but no write permission.

- The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-x**r--** represents that other world has read (r) only permission.

File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and**execute** permissions, which are described below −

1. Read

Grants the capability to read ie. view the contents of the file.

2. Write

Grants the capability to modify, or remove the content of the file.

3. Execute

User with execute permissions can run a file as a program.

Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

1. Read

Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.

2. Write

Access means that the user can add or delete files to the contents of the directory.

3. Execute

Executing a directory doesn't really make a lot of sense so think of this as a traverse permission.

A user must have execute access to the **bin** directory in order to execute ls or cd command.

Changing Permissions

To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode.

Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Chmod operator	**Description**
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood  users 1024  Nov 2 00:10  testfile
```

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes −

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx 1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx 1 amrood   users 1024  Nov 2 00:10  testfile
```

```
$chmod g=rx testfile
$ls -l testfile
-rw-r-xrwx 1 amrood  users 1024  Nov 2 00:10  testfile
```

Here's how you could combine these commands on a single line:

```
$chmod o+wx,u-x,g=rx testfile
$ls -l testfile
-rw-r-xrwx 1 amrood   users 1024  Nov 2 00:10  testfile
```

Using chmod with Absolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	**Octal Permission Representation**	**Ref**
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x

6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

hanging Owners and Groups

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.

Two commands are available to change the owner and the group of files −

- **chown** − The chown command stands for "change owner" and is used to change the owner of a file.
- **chgrp** − The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership

The chown command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of user can be either the name of a user on the system or the user id (uid) of a user on the system.

Following example −

```
$ chown amrood testfile
$
```

Changes the owner of the given file to the user **amrood**.

iv) Process Utilities:

e operating system tracks processes through a five digit ID number known as the **pid** or process ID . Each process in the system has a unique pid.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any one time, no two processes with the same pid exist in the system because it isthe pid that UNIX uses to track each process.

Starting a Process

When you start a process (run a command), there are two ways you can run it −

- Foreground Processes
- Background Processes

Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the ls command. If I want to list all the files in my current directory, I can use the following command −

```
$ls ch*.doc
```

This would display all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc  ch02-1.doc
```

The process runs in the foreground, the output is directed to my screen, and if the ls command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in foreground and taking much time, we cannot run any other commands (start any other processes) because prompt would not be available until program finishes its processing and comes out.

Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

```
$ls ch*.doc &
```

This would also display all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc  ch02-1.doc
```

Here if the **ls** command wants any input (which it does not), it goes into a stop state until I move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and process ID. You need to know the job number to manipulate it between background and foreground.

If you press the Enter key now, you see the following −

```
[1]   +   Done                ls ch*.doc &
$
```

The first line tells you that the **ls** command background process finishes successfully. The second is a prompt for another command.

Listing Running Processes

It is easy to see your own processes by running the **ps** (process status) command as follows −

```
$ps
PID       TTY      TIME        CMD
18358     ttyp3    00:00:00    sh
18361     ttyp3    00:01:31    abiword
```

```
18789    ttyp3    00:00:00    ps
```

One of the most commonly used flags for ps is the **-f** (f for full) option, which provides more information as shown in the following example −

```
$ps -f
UID      PID  PPID C STIME    TTY  TIME CMD
amrood  6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood  6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fields displayed by ps -f command −

Column	**Description**
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.

CMD	The command that started this process.

There are other options which can be used along with **ps** command −

Option	**Description**
-a	Shows information about all users
-x	Shows information about processes without terminals.
-u	Shows additional information like -f option.
-e	Display extended information.

Stopping Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when process is running in foreground mode.

If a process is running in background mode then first you would need to get its Job IDusing **ps** command and after that you can use **kill** command to kill the process as follows −

```
$ps -f
UID      PID  PPID C STIME    TTY  TIME CMD
amrood  6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood  6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here **kill** command would terminate first_one process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows −

```
$kill -9 6738
Terminated
```

Parent and Child Processes

Each unix process has two ID numbers assigned to it: Process ID (pid) and Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check ps -f example where this command listed both process ID and parent process ID.

Zombie and Orphan Processes

Normally, when a child process is killed, the parent process is told via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of allprocesses," **init** process, becomes the new PPID (parent process ID). Sometime these processes are called orphan process.

When a process is killed, a ps listing may still show the process with a Z state. This is a zombie, or defunct, process. The process is dead and not being used. These processes are different from orphan processes.They are the processes that has completed execution but still has an entry in the process table.

Daemon Processes

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon process has no controlling terminal. It cannot open /dev/tty. If you do a "ps -ef" and look at the tty field, all daemons will have a ? for the tty.

More clearly, a daemon is just a process that runs in the background, usually waiting for something to happen that it is capable of working with, like a printer daemon is waiting for print commands.

If you have a program which needs to do long processing then its worth to make it a daemon and run it in background.

The top Command

The **top** command is a very useful tool for quickly showing processes sorted by various criteria.

It is an interactive diagnostic tool that updates frequently and shows information about physical and virtual memory, CPU usage, load averages, and your busy processes.

Here is simple syntax to run top command and to see the statistics of CPU utilization by different processes −

```
$top
```

Job ID Versus Process ID

Background and suspended processes are usually manipulated via job number (job ID). This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in series or at the same time, in parallel, so using the job ID is easier than tracking the individual processes.

V. Disk Utilities:

The df Command

The first way to manage your partition space is with the df (disk free) command. The command df -k (disk free) displays the disk space usage in kilobytes, as shown below −

```
$df -k
Filesystem      1K-blocks      Used   Available Use% Mounted on
/dev/vzfs        10485760   7836644     2649116  75% /
/devices                0         0           0   0% /devices
$
```

Some of the directories, such as /devices, shows 0 in the kbytes, used, and avail columns as well as 0% for capacity. These are special (or virtual) file systems, and although they reside on the disk under /, by themselves they do not take up disk space.

The df -k output is generally the same on all Unix systems. Here's what it usually includes −

Column	**Description**
Filesystem	The physical file system name.
kbytes	Total kilobytes of space available on the storage medium.
used	Total kilobytes of space used (by files).
avail	Total kilobytes available for use.
capacity	Percentage of total space used by files.
Mounted on	What the file system is mounted on.

You can use the -h (human readable) option to display the output in a format that shows the size in easier-to-understand notation.

The du Command

The du (disk usage) command enables you to specify directories to show disk space usage on a particular directory.

This command is helpful if you want to determine how much space a particular directory is taking. Following command would display number of blocks consumed by each directory. A single block may take either 512 Bytes or 1 Kilo Byte depending on your system.

```
$du /etc
10     /etc/cron.d
126    /etc/default
6      /etc/dfs
...
$
```

The -h option makes the output easier to comprehend −

```
$du -h /etc
5k    /etc/cron.d
63k /etc/default
3k    /etc/dfs
...
$
```

Vi. Networking Commands

The ping Utility

The ping command sends an echo request to a host available on the network. Using this command you can check if your remote host is responding well or not.

The ping command is useful for the following −

- Tracking and isolating hardware and software problems.
- Determining the status of the network and various foreign hosts.
- Testing, measuring, and managing networks.

Syntax

Following is the simple syntax to use **ping** command −

```
$ping hostname or ip-address
```

Above command would start printing a response after every second. To come out of the command you can terminate it by pressing CNTRL + C keys.

Example

Following is the example to check the availability of a host available on the network −

```
$ping google.com
PING google.com (74.125.67.100) 56(84) bytes of data.
64 bytes from 74.125.67.100: icmp_seq=1 ttl=54 time=39.4 ms
64 bytes from 74.125.67.100: icmp_seq=2 ttl=54 time=39.9 ms
64 bytes from 74.125.67.100: icmp_seq=3 ttl=54 time=39.3 ms
64 bytes from 74.125.67.100: icmp_seq=4 ttl=54 time=39.1 ms
64 bytes from 74.125.67.100: icmp_seq=5 ttl=54 time=38.8 ms
--- google.com ping statistics ---
22 packets transmitted, 22 received, 0% packet loss, time 21017ms
rtt min/avg/max/mdev = 38.867/39.334/39.900/0.396 ms
$
```

If a host does not exist then it would behave something like this −

```
$ping giiiiiigle.com
ping: unknown host giiiiigle.com
$
```

The ftp Utility

Here ftp stands for **F**ile **T**ransfer **P**rotocol. This utility helps you to upload and download your file from one computer to another computer.

The ftp utility has its own set of UNIX like commands which allow you to perform tasks such as −

- Connect and login to a remote host.
- Navigate directories.
- List directory contents
- Put and get files
- Transfer files as ascii, ebcdic or binary

Syntax

Following is the simple syntax to use **ping** command −

```
$ftp hostname or ip-address
```

Above command would prompt you for login ID and password. Once you are authenticated, you would have access on the home directory of the login account and you would be able to perform various commands.

Few of the useful commands are listed below −

Command	**Description**
put filename	Upload filename from local machine to remote machine.
get filename	Download filename from remote machine to local machine.
mput file list	Upload more than one files from local machine to remote machine.
mget file list	Download more than one files from remote machine to local machine.
prompt off	Turns prompt off, by default you would be prompted to upload or download

	movies using mput or mget commands.
prompt on	Turns prompt on.
dir	List all the files available in the current directory of remote machine.
cd dirname	Change directory to dirname on remote machine.
lcd dirname	Change directory to dirname on local machine.
quit	Logout from the current login.

It should be noted that all the files would be downloaded or uploaded to or from current directories. If you want to upload your files in a particular directory then first you change to that directory and then upload required files.

Example

Following is the example to show few commands −

```
$ftp amrood.com
Connected to amrood.com.
220 amrood.com FTP server (Ver 4.9 Thu Sep 2 20:35:07 CDT 2009)
Name (amrood.com:amrood): amrood
331 Password required for amrood.
Password:
230 User amrood logged in.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
total 1464
```

```
drwxr-sr-x   3 amrood   group       1024 Mar 11 20:04 Mail
drwxr-sr-x   2 amrood   group       1536 Mar  3 18:07 Misc
drwxr-sr-x   5 amrood   group        512 Dec 7 10:59 OldStuff
drwxr-sr-x   2 amrood   group       1024 Mar 11 15:24 bin
drwxr-sr-x   5 amrood   group       3072 Mar 13 16:10 mpl
-rw-r--r--   1 amrood   group     209671 Mar 15 10:57 myfile.out
drwxr-sr-x   3 amrood   group        512 Jan  5 13:32 public
drwxr-sr-x   3 amrood   group        512 Feb 10 10:17 pvm3
226 Transfer complete.
ftp> cd mpl
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
total 7320
-rw-r--r--   1 amrood   group       1630 Aug  8 1994  dboard.f
-rw-r-----   1 amrood   group       4340 Jul 17 1994  vttest.c
226 Transfer complete.
ftp> get wave_shift
200 PORT command successful.
150 Opening data connection for wave_shift (525574 bytes).
226 Transfer complete.
528454 bytes received in 1.296 seconds (398.1 Kbytes/s)
ftp> quit
221 Goodbye.
$
```

The telnet Utility

Many times you would be in need to connect to a remote Unix machine and work on that machine remotely. Telnet is a utility that allows a computer user at one site to make a connection, login and then conduct work on a computer at another site.

Once you are login using telnet, you can perform all the activities on your remotely connect machine. Here is example telnet session −

```
C:>telnet amrood.com
Trying...
Connected to amrood.com.
Escape character is '^]'.

login: amrood
amrood's Password:
*****************************************************
*                                                   *
*                                                   *
*    WELCOME TO AMROOD.COM                          *
*                                                   *
*                                                   *
*****************************************************

Last unsuccessful login: Fri Mar  3 12:01:09 IST 2009
Last login: Wed Mar 8 18:33:27 IST 2009 on pts/10

   {  do your work }

$ logout
Connection closed.
```

```
C:>
```

The finger Utility

The finger command displays information about users on a given host. The host can be either local or remote.

Finger may be disabled on other systems for security reasons.

Following are the simple syntax to use finger command −

Check all the logged in users on local machine as follows −

```
$ finger
Login     Name       Tty      Idle  Login Time   Office
amrood              pts/0         Jun 25 08:03 (62.61.164.115)
```

Get information about a specific user available on local machine −

```
$ finger amrood
Login: amrood                          Name: (null)
Directory: /home/amrood                   Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
No Plan.
```

Check all the logged in users on remote machine as follows −

```
$ finger @avtar.com
Login     Name       Tty      Idle  Login Time   Office
amrood              pts/0         Jun 25 08:03 (62.61.164.115)
```

Get information about a specific user available on remote machine −

```
$ finger amrood@avtar.com
Login: amrood                          Name: (null)
```

```
Directory: /home/amrood              Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
No Plan.
```

VII) Filtering commands:

The grep Command

The grep program searches a file or files for lines that have a certain pattern. The syntax is −

```
$grep pattern file(s)
```

The name "grep" derives from the ed (a UNIX line editor) command g/re/p which means "globally search for a regular expression and print all lines containing it."

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work −

```
$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

There are various options which you can use along with grep command −

Option	**Description**

-v	Print all lines that do not match pattern.
-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (letter "l")
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

The sort Command

The **sort** command arranges lines of text alphabetically or numerically. The example below sorts the lines in the food file −

```
$sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java
Mandalay
Sushi and Sashimi
Sweet Tooth
Tio Pepe's Peppers
$
```

The **sort** command arranges lines of text alphabetically by default. There are many options that control the sorting −

Option	Description
-n	Sort numerically (example: 10 will sort after 2), ignore blanks and tabs.
-r	Reverse the order of sort.
-f	Sort upper- and lowercase together.
+x	Ignore first x fields when sorting.

x) Backup Utilities:

Backup restore and disk copy with tar :

– Backing up all files in a directory including subdirectories to a tape device (/dev/rmt/0),

```
tar cvf /dev/rmt/0 *
```

Viewing a tar backup on a tape

```
tar tvf /dev/rmt/0
```

Extracting tar backup from the tape

```
tar xvf /dev/rmt/0
```

(Restoration will go to present directory or original backup path depending on
relative or absolute path names used for backup)

Backup restore and disk copy with tar :

Back up all the files in current directory to tape .

```
find . -depth -print | cpio -ovcB > /dev/rmt/0
```

cpio expects a list of files and find command provides the list , cpio has
to put these file on some destination and a > sign redirect these files to tape . This can be a file as well .

Viewing cpio files on a tape

```
cpio -ivtB < /dev/rmt/0
```

Restoring a cpio backup

```
cpio -ivcB < /dev/rmt/0
```

Compress/uncompress files :

You may have to compress the files before or after the backup and it can be done with following commands .

Compressing a file

```
compress -v file_name
gzip filename
```

To uncompress a file

```
uncompress file_name.Z
or
gunzip filename
```

Stream Editor (Sed)

sed stands for **s**tream **ed**itor is a stream oriented editor which was created exclusively for executing scripts. Thus all the input you feed into it passes through and goes to STDOUT and it does not change the input file.

Invoking sed

Before we start, let us take make sure you have a local copy of /etc/passwd text file to work with **sed**.

As mentioned previously, sed can be invoked by sending data through a pipe to it as follows −

```
$ cat /etc/passwd | sed
Usage: sed [OPTION]... {script-other-script} [input-file]...


  -n, --quiet, --silent
               suppress automatic printing of pattern space
  -e script, --expression=script
..............................
```

The cat command dumps the contents of /etc/passwd to sed through the pipe into sed's pattern space. The pattern space is the internal work buffer that sed uses to do its work.

The sed General Syntax

Following is the general syntax for sed

```
/pattern/action
```

Here, **pattern** is a regular expression, and **action** is one of the commands given in the following table. If **pattern** is omitted, **action** is performed for every line as we have seen above.

The slash characters (/) that surround the pattern are required because they are used as delimiters.

Range	**Description**
p	Prints the line
d	Deletes the line
s/pattern1/pattern2/	Substitutes the first occurrence of pattern1 with pattern2.

Deleting All Lines with sed

Invoke sed again, but this time tell sed to use the editing command delete line, denoted by the single letter d −

```
$ cat /etc/passwd | sed 'd'
$
```

Instead of invoking sed by sending a file to it through a pipe, you can instruct sed to read the data from a file, as in the following example.

The following command does exactly the same thing as the previous Try It Out, without the cat command −

```
$ sed -e 'd' /etc/passwd
$
```

The sed Addresses

Sed also understands something called addresses. Addresses are either particular locations in a file or a range where a particular editing command should be applied. When sed encounters no addresses, it performs its operations on every line in the file.

The following command adds a basic address to the sed command you've been using −

```
$ cat /etc/passwd | sed '1d' |more
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
$
```

Notice that the number 1 is added before the delete edit command. This tells sed to perform the editing command on the first line of the file. In this example, sed will delete the first line of /etc/password and print the rest of the file.

The sed Address Ranges

So what if you want to remove more than one line from a file? You can specify an address range with sed as follows −

```
$ cat /etc/passwd | sed '1, 5d' |more
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
$
```

Above command would be applied on all the lines starting from 1 through 5. So it deleted first five lines.

Try out the following address ranges −

Range	Description
'4,10d'	Lines starting from 4th till 10th are deleted
'10,4d'	Only 10th line is deleted, because sed does not work in reverse direction.
'4,+5d'	This will match line 4 in the file, delete that line, continue to delete the next five lines, and then cease its deletion and print the rest
'2,5!d'	This will deleted everything except starting from 2nd till 5th line.
'1~3d'	This deletes the first line, steps over the next three lines, and then deletes the fourth line. Sed continues applying this pattern until the end of the file.
'2~2d'	This tells sed to delete the second line, step over the next line, delete the next line, and repeat until the end of the file is reached.
'4,10p'	Lines starting from 4th till 10th are printed
'4,d'	This would generate syntax error.
',10d'	This would also generate syntax error.

Note: While using **p** action, you should use **-n** option to avoid repetition of line printing. Check the difference in betweek following two commands −

```
$ cat /etc/passwd | sed -n '1,3p'
```

Check the above command without **-n** as follows −

```
$ cat /etc/passwd | sed '1,3p'
```

NAME

awk - Finds and Replaces text, database sort/validate/index

SYNOPSIS

awk 'Program' input-file1 input-file2 ... awk -f PROGRAM-FILE input-file1 input-file2 ...

DESCRIPTION

awk command searches files for text containing a pattern. When a line or text matches, awk performs a specific action on that line/text. The Program statement tells awk what operation to do; Program statement consists of a series of "rules" where each rule specifies one pattern to search for, and one action to perform when a particular pattern is found. A regular expression enclosed in slashes (/) is an awk pattern to match every input record whose text belongs to that set.

OPTIONS

Tag	**Description**
-F FS --field-separator FS	Use FS for the input field separator (the value of the 'FS' predefined variable).
-f PROGRAM-FILE --file PROGRAM-FILE	Read the awk program source from the file PROGRAM-FILE, instead of from the first command line argument.
-mf NNN -mr NNN	The 'f' flag sets the maximum number of fields, and the 'r' flag sets the maximum record size. These options are ignored by 'gawk', since 'gawk' has no predefined limits; they are only for compatibility with the Bell Labs research version of Unix awk.

-v VAR=VAL --assign VAR=VAL	Assign the variable VAR the value VAL before program execution begins.
-W traditional -W compat --traditional --compat	Use compatibility mode, in which 'gawk' extensions are turned off.
-W lint --lint	Give warnings about dubious or non-portable awk constructs.
-W lint-old --lint-old	Warn about constructs that are not available in the original Version 7 Unix version of awk.
-W posix --posix	Use POSIX compatibility mode, in which 'gawk' extensions are turned off and additional restrictions apply.
-W re-interval --re-interval	Allow interval expressions, in regexps.
-W source=PROGRAM-TEXT --source PROGRAM-TEXT	Use PROGRAM-TEXT as awk program source code. This option allows mixing command line source code with source code from files, and is particularly useful for mixing command line programs with library functions.
--	Signal the end of options. This is useful to allow further arguments to the awk program itself to start with a '-'. This is mainly for consistency with POSIX argument parsing conventions.
'Program'	A series of patterns and actions
Input-File	If no Input-File is specified then awk applies the Program to "standard input", (piped output of some other command or the terminal. Typed input will continue until end-of-file (typing 'Control-d')

EXAMPLES

- To return the second item($2) from each line of the output from an ls - l listing.

```
• $ ls -l | awk '{print $2}'
• 13
• 3
• 17
• 7
```

- To print the Row Number (NR), then a dash and space ("- ") and then the first item ($1) from each line in sample.txt.

  First create a sample.txt file

```
Sample Line 1
Sample Line 2
Sample Line 3
$ awk '{print NR "- " $1 }' sample.txt
1 - Sample
2 - Sample
3 - Sample
```

- To print the first item ($1) and then the second last item $(NF-1) from each line in sample.txt.

```
• $ awk '{print $1, $(NF-1) }' sample.txt
• Sample Line
• Sample Line
• Sample Line
```

- To print non-empty line from a file.

```
• $ awk 'NF > 0' sample.txt
```

- To print the length of the longest input line.

```
• $ awk '{ if (length($0) > max) max = length($0) } END { print max }' sample.txt
• 13
```

- To print seven random numbers from zero to 100, inclusive.

```
$  awk 'BEGIN { for (i = 1; i <= 7; i++) print int(101 * rand()) }'
24
29
85
15
59
19
81
```

- To count the lines in a file

```
$ awk 'END { print NR }' sample.txt
3
```

basic syntax of AWK:

```
awk 'BEGIN {start_action} {action} END {stop_action}' filename
```

Here the actions in the begin block are performed before processing the file and the actions in the end block are performed after processing the file. The rest of the actions are performed while processing the file.

Examples:

Create a file input_file with the following data. This file can be easily created using the output of ls -l.

```
-rw-r--r-- 1 center center  0 Dec  8 21:39 p1

-rw-r--r-- 1 center center 17 Dec  8 21:15 t1
```

```
-rw-r--r-- 1 center center 26 Dec  8 21:38 t2

-rw-r--r-- 1 center center 25 Dec  8 21:38 t3

-rw-r--r-- 1 center center 43 Dec  8 21:39 t4

-rw-r--r-- 1 center center 48 Dec  8 21:39 t5
```

From the data, you can observe that this file has rows and columns. The rows are separated by a new line character and the columns are separated by a space characters. We will use this file as the input for the examples discussed here.

1. awk '{print $1}' input_file

Here $1 has a meaning. $1, $2, $3... represents the first, second, third columns... in a row respectively. This awk command will print the first column in each row as shown below.

```
-rw-r--r--

-rw-r--r--

-rw-r--r--

-rw-r--r--

-rw-r--r--

-rw-r--r--
```

To print the 4th and 6th columns in a file use awk '{print $4,$5}' input_file

Here the Begin and End blocks are not used in awk. So, the print command will be executed for each row it reads from the file. In the next example we will see how to use the Begin and End blocks.

2. awk 'BEGIN {sum=0} {sum=sum+$5} END {print sum}' input_file

This will prints the sum of the value in the 5th column. In the Begin block the variable sum is assigned with value 0. In the next block the value of 5th column is added to the sum variable. This addition of the 5th column to the sum variable repeats for every row it processed. When all the rows are processed the sum variable will hold the sum of the values in the 5th column. This value is printed in the End block.

3. In this example we will see how to execute the awk script written in a file. Create a file sum_column and paste the below script in that file

```
#!/usr/bin/awk -f

BEGIN {sum=0}

{sum=sum+$5}

END {print sum}
```

Now execute the the script using awk command as

awk -f sum_column input_file.

This will run the script in sum_column file and displays the sum of the 5th column in the input_file.

4. awk '{ if($9 == "t4") print $0;}' input_file

This awk command checks for the string "t4" in the 9th column and if it finds a match then it will print the entire line. The output of this awk command is

```
-rw-r--r-- 1 pcenter pcenter 43 Dec 8 21:39 t4
```

5. awk 'BEGIN { for(i=1;i<=5;i++) print "square of", i, "is",i*i; }'

This will print the squares of first numbers from 1 to 5. The output of the command is

```
square of 1 is 1

square of 2 is 4

square of 3 is 9

square of 4 is 16

square of 5 is 25
```

Notice that the syntax of “if” and “for” are similar to the C language.

Awk Built in Variables:

You have already seen $0, $1, $2... which prints the entire line, first column, second column... respectively. Now we will see other built in variables with examples.

FS - Input field separator variable:

So far, we have seen the fields separted by a space character. By default Awk assumes that fields in a file are separted by space characters. If the fields in the file are separted by any other character, we can use the FS variable to tell about the delimiter.

6. awk 'BEGIN {FS=":"} {print $2}' input_file
OR
awk -F: '{print $2}' input_file

This will print the result as

```
39 p1

15 t1

38 t2

38 t3

39 t4
```

```
39 t5
```

OFS - Output field separator variable:

By default whenever we printed the fields using the print statement the fields are displayed with space character as delimiter. For example

7. awk '{print $4,$5}' input_file

The output of this command will be

```
center 0

center 17

center 26

center 25

center 43

center 48
```

We can change this default behavior using the OFS variable as

awk 'BEGIN {OFS=":"} {print $4,$5}' input_file

```
center:0

center:17

center:26
```

```
center:25

center:43

center:48
```

Note: print $4,$5 and print $4$5 will not work the same way. The first one displays the output with space as delimiter. The second one displays the output without any delimiter.

NF - Number of fileds variable:

The NF can be used to know the number of fields in line

8. awk '{print NF}' input_file
This will display the number of columns in each row.

NR - number of records variable:
The NR can be used to know the line number or count of lines in a file.

9. awk '{print NR}' input_file
This will display the line numbers from 1.

10. awk 'END {print NR}' input_file
This will display the total number of lines in the file.

String functions in Awk:
Some of the string functions in awk are:

index(string,search)
length(string)
split(string,array,separator)
substr(string,position)
substr(string,position,max)
tolower(string)
toupper(string)

Advanced Examples:

1. Filtering lines using Awk split function

The awk split function splits a string into an array using the delimiter.

The syntax of split function is
split(string, array, delimiter)

Now we will see how to filter the lines using the split function with an example.

The input "file.txt" contains the data in the following format

```
1 U,N,UNIX,000

2 N,P,SHELL,111

3 I,M,UNIX,222

4 X,Y,BASH,333

5 P,R,SCRIPT,444
```

Required output: Now we have to print only the lines in which whose 2nd field has the string "UNIX" as the 3rd field(The 2nd filed in the line is separated by comma delimiter).
The ouptut is:

```
1 U,N,UNIX,000

3 I,M,UNIX,222
```

The awk command for getting the output is:

```
awk '{

        split($2,arr,",");

        if(arr[3] == "UNIX")
        print $0

} ' file.txt
```

The Substitution Command

The substitution command, denoted by **s**, will substitute any string that you specify with any other string that you specify.

To substitute one string with another, you need to have some way of telling sed where your first string ends and the substitution string begins. This is traditionally done by bookending the two strings with the forward slash (/) character.

The following command substitutes the first occurrence on a line of the string**root** with the string **amrood**.

```
$ cat /etc/passwd | sed 's/root/amrood/'
amrood:x:0:0:root user:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
..........................
```

It is very important to note that sed substitutes only the first occurrence on a line. If the string root occurs more than once on a line only the first match will be replaced.

To tell sed to do a global substitution, add the letter **g** to the end of the command as follows −

```
$ cat /etc/passwd | sed 's/root/amrood/g'
amrood:x:0:0:amrood user:/amrood:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
...........................
```

Substitution Flags

There are a number of other useful flags that can be passed in addition to the g flag, and you can specify more than one at a time.

Flag	**Description**
g	Replace all matches, not just the first match.
NUMBER	Replace only NUMBERth match.
p	If substitution was made, print pattern space.
w FILENAME	If substitution was made, write result to FILENAME.
I or i	Match in a case-insensitive manner.
M or m	In addition to the normal behavior of the special regular expression characters ^ and $, this flag causes ^ to match the empty string after a newline and $ to match the empty string before a newline.

Using an Alternative String Separator

You may find yourself having to do a substitution on a string that includes the forward slash character. In this case, you can specify a different separator by providing the designatedcharacter after the s.

```
$ cat /etc/passwd | sed 's:/root:/amrood:g'
amrood:x:0:0:amrood user:/amrood:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

In the above example we have used **:** as delimeter instead of slash / because we were trying to search /root instead of simple root.

Replacing with Empty Space

Use an empty substitution string to delete the root string from the /etc/passwd file entirely −

```
$ cat /etc/passwd | sed 's/root//g'
:x:0:0::/:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

Address Substitution

If you want to substitute the string sh with the string quiet only on line 10, you can specify it as follows −

```
$ cat /etc/passwd | sed '10s/sh/quiet/g'
root:x:0:0:root user:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/quiet
```

Similarly, to do an address range substitution, you could do something like the following −

```
$ cat /etc/passwd | sed '1,5s/sh/quiet/g'
root:x:0:0:root user:/root:/bin/quiet
daemon:x:1:1:daemon:/usr/sbin:/bin/quiet
bin:x:2:2:bin:/bin:/bin/quiet
```

```
sys:x:3:3:sys:/dev:/bin/quiet
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

As you can see from the output, the first five lines had the string sh changed to quiet, but the rest of the lines were left untouched.

The Matching Command

You would use **p** option along with **-n** option to print all the matching lines as follows −

```
$ cat testing | sed -n '/root/p'
root:x:0:0:root user:/root:/bin/sh
[root@ip-72-167-112-17 amrood]# vi testing
root:x:0:0:root user:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Using Regular Expression

While matching pattern, you can use regular expression which provides more flexibility.

Check following example which matches all the lines starting with daemon and then deleting them −

```
$ cat testing | sed '/^daemon/d'
root:x:0:0:root user:/root:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Following is the example which would delete all the lines ending with **sh** −

```
$ cat testing | sed '/sh$/d'
sync:x:4:65534:sync:/bin:/bin/sync
```

The following table lists four special characters that are very useful in regular expressions.

Character	**Description**
^	Matches the beginning of lines.
$	Matches the end of lines.
.	Matches any single character.
*	Matches zero or more occurrences of the previous character

[chars]	Matches any one of the characters given in chars, where chars is a sequence of characters. You can use the - character to indicate a range of characters.

Matching Characters

Look at a few more expressions to demonstrate the use of the metacharacters. For example, the following pattern −

Expression	**Description**
/a.c/	Matches lines that contain strings such as a+c, a-c, abc, match, and a3c, whereas the pattern
/a*c/	Matches the same strings along with strings such as ace, yacc, and arctic.
/[tT]he/	Matches the string The and the:
/^$/	Matches Blank lines
/^.*$/	Matches an entire line whatever it is.
/ */	Matches one or more spaces
/^$/	Matches Blank lines

Following table shows some frequently used sets of characters −

Set	**Description**

[a-z]	Matches a single lowercase letter
[A-Z]	Matches a single uppercase letter
[a-zA-Z]	Matches a single letter
[0-9]	Matches a single number
[a-zA-Z0-9]	Matches a single letter or number

Character Class Keywords

Some special keywords are commonly available to regexps, especially GNU utilities thatemploy regexps. These are very useful for sed regular expressions as they simplify things and enhance readability.

For example, the characters a through z as well as the characters A through Z constitute one such class of characters that has the keyword [[:alpha:]]

Using the alphabet character class keyword, this command prints only those lines in the /etc/syslog.conf file that start with a letter of the alphabet −

```
$ cat /etc/syslog.conf | sed -n '/^[[:alpha:]]/p'
authpriv.*                    /var/log/secure
mail.*                       -/var/log/maillog
cron.*                       /var/log/cron
uucp,news.crit                    /var/log/spooler
local7.*                     /var/log/boot.log
```

The following table is a complete list of the available character class keywords in GNU sed.

Character Class	**Description**
[[:alnum:]]	Alphanumeric [a-z A-Z 0-9]
[[:alpha:]]	Alphabetic [a-z A-Z]
[[:blank:]]	Blank characters (spaces or tabs)
[[:cntrl:]]	Control characters
[[:digit:]]	Numbers [0-9]
[[:graph:]]	Any visible characters (excludes whitespace)
[[:lower:]]	Lowercase letters [a-z]
[[:print:]]	Printable characters (noncontrol characters)
[[:punct:]]	Punctuation characters
[[:space:]]	Whitespace
[[:upper:]]	Uppercase letters [A-Z]
[[:xdigit:]]	Hex digits [0-9 a-f A-F]

Aampersand Referencing

The sed metacharacter & represents the contents of the pattern that was matched. For instance, say you have a file called phone.txt full of phone numbers, such as the following −

```
5555551212
5555551213
5555551214
6665551215
6665551216
7775551217
```

You want to make the area code (the first three digits) surrounded by parentheses for easier reading. To do this, you can use the ampersand replacement character, like so −

```
$ sed -e 's/^[[:digit:]][[:digit:]][[:digit:]]/(&)/g' phone.txt
(555)5551212
(555)5551213
(555)5551214
(666)5551215
(666)5551216
(777)5551217
```

Here in pattern part you are matching first 3 digits and then using & you are replacing those 3 digits with surrounding parentheses.

Using Multiple sed Commands

You can use multiple sed commands in a single sed command as follows −

```
$ sed -e 'command1' -e 'command2' ... -e 'commandN' files
```

Here command1 through commandN are sed commands of the type discussed previously. These commands are applied to each of the lines in the list of files given by files.

Using the same mechanism, we can write above phone number example as follows −

```
$ sed -e 's/^[[:digit:]]\{3\}/(&)/g'  \
                    -e 's/)[[:digit:]]\{3\}/&-/g' phone.txt
(555)555-1212
(555)555-1213
(555)555-1214
(666)555-1215
(666)555-1216
(777)555-1217
```

Note − In the above example, instead of repeating the character class keyword [[:digit:]] three times, you replaced it with \{3\}, which means to match the preceding regular expression three times. Here I used \ to give line break you should remove this before running this command.

Back References

The ampersand metacharacter is useful, but even more useful is the ability to define specific regions in a regular expressions so you can reference them in your replacement strings. By defining specific parts of a regular expression, you can then refer back to those parts with a special reference character.

To do back references, you have to first define a region and then refer back to that region. To define a region you insert backslashed parentheses around each region of interest. The first region that you surround with backslashes is then referenced by \1, the second region by \2, and so on.

Assuming phone.txt has the following text −

```
(555)555-1212
(555)555-1213
(555)555-1214
(666)555-1215
(666)555-1216
```

```
(777)555-1217
```

Now try the following command −

```
$ cat phone.txt | sed 's/\(.*)\)\(.*-\)\(.*$\)/Area \
                    code: \1 Second: \2 Third: \3/'
Area code: (555) Second: 555- Third: 1212
Area code: (555) Second: 555- Third: 1213
Area code: (555) Second: 555- Third: 1214
Area code: (666) Second: 555- Third: 1215
Area code: (666) Second: 555- Third: 1216
Area code: (777) Second: 555- Third: 1217
```

Note − In the above example each regular expression inside the parenthesis would be back referenced by \1, \2 and so on. Here I used \ to give line break you should remove this before running this command.

Shell Prompt

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

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

Following is a simple example of **date** command which displays current date and time:

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

You can customize your command prompt using environment variable PS1 explained in Environment tutorial.

Shell Types

In UNIX there are two major types of shells:

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

There are again various subcategories for Bourne Shell which are listed as follows −

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

The different C-type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".

The Bourne shell is usually installed as /bin/sh on most versions of UNIX. For this reason, it is the shell of choice for writing scripts to use on several different versions of UNIX.

In this tutorial, we are going to cover most of the Shell concepts based on Borne Shell.

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound sign, #, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell scripts and functions are both interpreted. This means they are not compiled.

We are going to write a many scripts in the next several tutorials. This would be a simple text file in which we would put our all the commands and several other required constructs that tell the shell environment what to do and when to do it.

Example Script

Assume we create a test.sh script. Note all the scripts would have **.sh**extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example −

```
#!/bin/sh
```

This tells the system that the commands that follow are to be executed by the Bourne shell. It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.

To create a script containing these commands, you put the shebang line first and then add the commands −

```
#!/bin/bash
pwd
ls
```

Shell Comments

You can put your comments in your script as follows −

```
#!/bin/bash


# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pwd
ls
```

Now you save the above content and make this script executable as follows −

```
$chmod +x test.sh
```

Now you have your shell script ready to be executed as follows −

```
$./test.sh
```

This would produce following result −

```
/home/amrood
index.htm unix-basic_utilities.htm unix-directories.htm
test.sh    unix-communication.htm    unix-environment.htm
```

Note: To execute your any program available in current directory you would execute using **./program_name**

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, however, it is still just a list of commands executed sequentially.

Following script use the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh


# Author : Zara Ali

# Copyright (c) Tutorialspoint.com

# Script follows here:


echo "What is your name?"

read PERSON

echo "Hello, $PERSON"
```

Here is sample run of the script −

```
$./test.sh

What is your name?

Zara Ali

Hello, Zara Ali

$
```

The following table shows a number of special variables that you can use in your shell scripts −

Variable	**Description**
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.

$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Command-Line Arguments

The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to command line −

```
#!/bin/sh


echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script −

```
$./test.sh Zara Ali
```

```
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

Special Parameters $* and $@

There are special parameters that allow accessing all of the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".

Both the parameter specifies all command-line arguments but the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters −

```
#!/bin/sh

for TOKEN in $*
do
  echo $TOKEN
done
```

There is one sample run for the above script −

```
$./test.sh Zara Ali 10 Years Old
Zara
Ali
10
Years
```

```
Old
```

Note: Here **do**...**done** is a kind of loop which we would cover in subsequent tutorial.

Exit Status

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```

There are various operators supported by each shell. Our tutorial is based on default shell (Bourne) so we are going to cover all the important Bourne Shell operators in the tutorial.

There are following operators which we are going to discuss −

- Arithmetic Operators.

- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.

Here is simple example to add two numbers −

```
#!/bin/sh


val=`expr 2 + 2`
echo "Total value : $val"
```

This would produce following result −

```
Total value : 4
```

There are following points to note down −

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- Complete expression should be enclosed between ``, called inverted commas.

Arithmetic Operators

There are following arithmetic operators supported by Bourne Shell.

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

Show Examples

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

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators:

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

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

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

Show Examples

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

-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.

Boolean Operators

There are following boolean operators supported by Bourne Shell.

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

Show Examples

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

String Operators

There are following string operators supported by Bourne Shell.

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

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-n $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

File Test Operators

There are following operators to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on −

Show Examples

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-c $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.

-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

- The **if...else** statements
- The **case...esac** statement

The if...else statements:

If else statements are useful decision making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if..else statement −

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

Most of the if statements check relations using relational operators discussed in previouschapter.

The case...esac Statement

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of case...esac statement which is detailed here −

- case...esac statement

Unix Shell's case...esac is very similar to switch...case statement we have in other programming languages like C or C++ and PERL etc.

Loops are a powerful programming tool that enable you to execute a set of commands repeatedly. In this tutorial, you would examine the following types of loops available to shell programmers −

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

You would use different loops based on dfferent situation. For example while loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes true.

Once you have good programming practice you would start using appropriate loop based on situation. Here while and for loops are available in most of the other programming languages like C, C++ and PERL etc.

Nesting Loops

All the loops support nesting concept which means you can put one loop inside another similar or different loops. This nesting can go upto unlimited number of times based on your requirement.

Here is an example of nesting **while** loop and similar way other loops can be nested based on programming requirement −

Nesting while Loops

It is possible to use a while loop as part of the body of another while loop.

Syntax

```
while command1 ; # this is loop1, the outer loop
do
   Statement(s) to be executed if command1 is true

   while command2 ; # this is loop2, the inner loop
   do
      Statement(s) to be executed if command2 is true
   done

   Statement(s) to be executed if command1 is true
done
```

Example

Here is a simple example of loop nesting, let's add another countdown loop inside the loop that you used to count to nine −

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ] # this is loop2
   do
```

```
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```

This will produce following result. It is important to note how **echo -n** works here. Here **-n** option let echo to avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

What is Substitution?

The shell performs substitution when it encounters an expression that contains one or more special characters.

Example

Following is the example, while printing value of the variable its substitued by its value. Same time "\n" is substituted by a new line −

```
#!/bin/sh

a=10
```

```
echo -e "Value of a is $a \n"
```

This would produce following result. Here **-e** option enables interpretation of backslash escapes.

```
Value of a is 10
```

Command Substitution

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Syntax

The command substitution is performed when a command is given as:

```
`command`
```

When performing command substitution make sure that you are using the backquote, not the single quote character.

Example

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrate command substitution −

```
#!/bin/sh

DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

This will produce following result −

```
Date is Thu Jul 2 03:59:57 MST 2009

Logged in user are 1

Uptime is Thu Jul  2 03:59:57 MST 2009

03:59:57 up 20 days, 14:03, 1 user,  load avg: 0.13, 0.07, 0.15
```

Variable Substitution

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Here is the following table for all the possible substitutions −

Form	**Description**
${var}	Substitue the value of var.
${var:-word}	If var is null or unset, word is substituted for **var**. The value of var does not change.
${var:=word}	If var is null or unset, var is set to the value of **word**.
${var:?message}	If var is null or unset, message is printed to standard error. This checks that variables are set correctly.
${var:+word}	If var is set, word is substituted for var. The value of vardoes not change.

Example

Following is the example to show various states of the above substitution −

```
#!/bin/sh


echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"


echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"


unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"


var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"


echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

This would produce following result −

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set


3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

The Metacharacters

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

For example **?** matches with a single charater while listing files in a directory and an ***** would match more than one characters. Here is a list of most of the shell special characters (also called metacharacters) −

```
* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab
```

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

Example

Following is the example which show how to print a ***** or a **?** −

```
#!/bin/sh


echo Hello; Word
```

This would produce following result −

```
Hello
./test.sh: line 2: Word: command not found


shell returned 127
```

Now let us try using a quoted character −

```
#!/bin/sh


echo Hello\; Word
```

This would produce following result −

```
Hello; Word
```

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell −

```
#!/bin/sh


echo "I have \$1200"
```

This would produce following result −

```
I have $1200
```

There are following four forms of quotings −

Quoting	**Description**
Single quote	All special characters between these quotes lose their special meaning.
Double quote	Most special characters between these quotes lose their special meaning with these exceptions: • $ • ` • \$ • \' • \" • \\
Backslash	Any character immediately following the backslash loses its special meaning.
Back Quote	Anything in between back quotes would be treated as a command and would be executed.

The Single Quotes

Consider an echo command that contains many special shell characters −

```
echo <-$1500.**>; (update?) [y|n]
```

Putting a backslash in front of each special character is tedious and makes the line difficult to read −

```
echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
```

There is an easy way to quote a large group of characters. Put a single quote (') at the beginning and at the end of the string −

```
echo '<-$1500.**>; (update?) [y|n]'
```

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

```
echo 'It\'s Shell Programming'
```

The Double Quotes

Try to execute the following shell script. This shell script makes use of single quote −

```


VAR=ZARA

echo '$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]'
```

This would produce following result −

```
$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]
```

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows −

```
VAR=ZARA
echo "$VAR owes <-\$1500.**>; [ as of (`date +%m/%d`) ]"
```

Now this would produce following result −

```
ZARA owes <-$1500.**>; [ as of (07/02) ]
```

Double quotes take away the special meaning of all characters except the following −

- $ for parameter substitution.
- Backquotes for command substitution.
- \$ to enable literal dollar signs.
- \` to enable literal backquotes.
- \" to enable embedded double quotes.
- \\ to enable embedded backslashes.
- All other \ characters are literal (not special).

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

```
echo 'It\'s Shell Programming'
```

The Back Quotes

Putting any Shell command in between back quotes would execute the command

Syntax:

Here is the simple syntax to put any Shell **command** in between back quotes −

Example

```
var=`command`
```

Example

Following would execute **date** command and produced result would be stored in DATA variable.

```
DATE=`date`


echo "Current Date: $DATE"
```

This would produce following result −

```
Current Date: Thu Jul 2 05:28:45 MST 2009
```

Output Redirection

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection:

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal −

Check following **who** command which would redirect complete output of the command in users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check users file then it would have complete content −

```
$ cat users
```

```
oko        tty01   Sep 12 07:30
ai         tty15   Sep 12 13:32
ruth       tty21   Sep 12 10:10
pat        tty24   Sep 12 13:07
steve      tty25   Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example −

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows −

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less- than character < is used to redirect the input of a command.

The commands that normally take their input from standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file users generated above, you can execute the command as follows −

```
$ wc -l users
```

```
2 users
$
```

Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the wc command from the file users −

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

Here Document

A here document is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a here document is −

```
command << delimiter
document
delimiter
```

Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.

The delimiter tells the shell that the here document has completed. Without it, the shell continues to read input forever. The delimiter must be a single word that does not contain spaces or tabs.

Following is the input to the command **wc -l** to count total number of line −

```
$wc -l << EOF
        This is a simple lookup program
        for good (and bad) restaurants
        in Cape Town.
EOF
3
$
```

You can use here document to print multiple lines using your script as follows −

```
#!/bin/sh

cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

This would produce following result −

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

The following script runs a session with the vi text editor and save the input in the file test.txt.

```
#!/bin/sh

filename=test.txt
vi $filename <<EndOfCommands
i
```

```
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

If you run this script with vim acting as vi, then you will likely see output like the following −

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

After running the script, you should see the following added to the file test.txt −

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

Discard the output

Sometimes you will need to execute a command, but you don't want the output displayed to the screen. In such cases you can discard the output by redirecting it to the file /dev/null −

```
$ command > /dev/null
```

Here command is the name of the command you want to execute. The file /dev/null is a special file that automatically discards all its input.

To discard both output of a command and its error output, use standard redirection to redirect STDERR to STDOUT −

```
$ command > /dev/null 2>&1
```

Here 2 represents STDERR and 1 represents STDOUT. You can display a message on to STDERR by redirecting STDOUT into STDERR as follows −

```
$ echo message 1>&2
```

Redirection Commands

Following is the complete list of commands which you can use for redirection −

Command	**Description**
pgm > file	Output of pgm is redirected to file
pgm < file	Program pgm reads its input from file.
pgm >> file	Output of pgm is appended to file.
n > file	Output from stream with descriptor n redirected to file.
n >> file	Output from stream with descriptor n appended to file.
n >& m	Merge output from stream n with stream m.
n <& m	Merge input from stream n with stream m.
<< tag	Standard input comes from here through next tag at start of line.
|	Takes output from one program, or process, and sends it to another.

Note that file descriptor 0 is normally standard input (STDIN), 1 is standard output (STDOUT), and 2 is standard error output (STDERR).

UNIT II

Syllabus

Files and Directories - File Concept, File types, File System Structure, file metadata –Inodes, kernel support for files, system calls for file I/O operations -open, creat,read,write,close,lseek,dup2, file status information –stat family, file and record locking – fcntl function, file permissions –chmod, fchmod, file ownership –chown, lchown, fchown, links –soft links and hard links –symlink, link, unlink.

Directories –Creating, removing and changing Directories –mkdir, rmdir,chdir,obtaining current working directory –getcwd, Directory contents, Scanning Directories- opendir, readdir, closedir, rewinddir functions.

File Concept

At the very top of the **file** system is single directory called "root" which is represented by a / (slash). All other files are "descendents" of root. The number of levels is largely arbitrary, although most **UNIX** systems share some organizational similarities.

File Types

The UNIX filesystem contains several different types of files:

- Ordinary Files
  - Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
  - Always located within/under a directory file
  - Do not contain other files
- Directories
  - Branching points in the hierarchical tree
  - Used to organize groups of files
  - May contain ordinary files, special files or other directories
  - Never contain "real" information which you would work with (such as text). Basically, just used for organizing files.
  - All files are descendants of the root directory, (named /) located at the top of the tree.
- Special Files
  - Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Ouput (I/O) operations

- Unix considers any device attached to the system to be a file - including your terminal:
  - By default, a command treats your terminal as the standard input file (stdin) from which to read its input
  - Your terminal is also treated as the standard output file (stdout) to which a command's output is sent
  - Stdin and stdout will be discussed in more detail later
- Two types of I/O: character and block
- Usually only found under directories named /dev

- Pipes
  - UNIX allows you to link commands together using a pipe. The pipe acts a temporary file which only exists to hold data from one command until it is read by another
  - For example, to pipe the output from one command into another command:
  -
  - **who | wc -l**

Hierarchical File Structure

A file system is a logical collection of files on a partition or disk. A partition is a container for information and can span an entire hard drive if desired.
Your hard drive can have various partitions which usually contains only one file system, such as one file system housing the / file system or another containing the /home file system.
One file system per partition allows for the logical maintenance and management of differing file systems.
Everything in Unix is considered to be a file, including physical devices such as DVD-ROMs, USB devices, floppy drives, and so forth.

All of the files in the UNIX file system are organized into a multi-leveled hierarchy called a directory tree.
A family tree is an example of a hierarchical structure that represents how the UNIX file system is organized. The UNIX file system might also be envisioned as an inverted tree orthe root system of plant.
At the very top of the file system is single directory called "root" which is represented by a / (slash). All other files are "descendents" of root.
The number of levels is largely arbitrary, although most UNIX systems share some organizational similarities. The "standard" UNIX file system is discussed later.
Example:

```
          / (root)
          |
   ----------------
```

```
      |          |          |
     /bin      /usr      /tmp
                 |
                 |
    ------------------------------------------------------------------
        |       |       |                                  |
       /public  /misc    /staff                           /students
        |               |                                  |
    ---------        ---------------       -----------------------------
    |        |       |     |     |     |      |      |      |      |
   /software  /doc      /john /mary /bill /carl  /tom  /dick  /mary  /lisa
```

File metadata

An **inode** is an entry in **inode** table, containing information (the metadata) about a regular file and directory. An **inode** is a data structure on a traditional Unix-style file system such as ext3 or ext4. **Inode** number also called as index number , it consists following attributes

Commands to access Inode numbers

Following are some commands to access the Inode numbers for files :

1) ls -i Command

the flag -i is used to print the Inode number for each file.

```
$ ls -i

1448240 a 1441807 Desktop 1447344 mydata 1441813 Pictures 1442737 testfile 1448145 worm

1448240 a1 1441811 Documents 1442707 my_ls 1442445 practice 1442739 test.py

1447139 alpha 1441808 Downloads 1447278 my_ls_alpha.c 1441810 Public 1447099
```

Kernel Supports for files:

To understand the file system, you must first think about how the kernel organizing and maintains information. The kernel does a lot of book keeping; it needs to know which process are running, what their memory layout is, what open filres they have, and etc. To support this, the kernel maintains three important tables for managing open files for a process: the process table, the file table, and the v-node/i-node info.

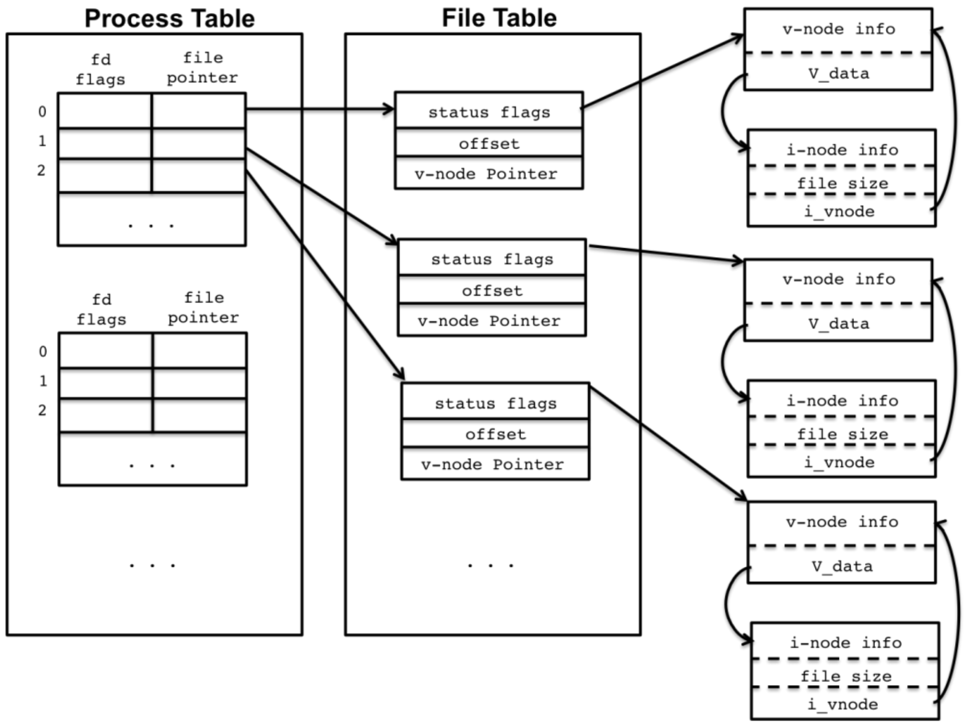


Fig. Kernel File System Data Structures

Introduction to System Calls

System calls are commands that are executed by the operating system. "System calls are the only way to access kernel facilities such as file system, multitasking mechanisms and the interprocess communication primitives."(Rochkind's book, *Advanced Unix Programming*)

Here is a short list of System Calls that we may be using in your labs:

- **For file I/0**
  - creat(name, permissions)
  - open(name, mode)
  - close(fd)
  - read(fd, buffer, num)
  - write(fd, buffer, num)
  - stat(name, buffer)
  - fstat(fd, buffer)
- **For process control**
  - fork()
  - wait(status)
  - execl(), execlp(), execv(), execvp()

  - exit()
  - signal(sig, handler)
  - kill(sig, pid)
- **For interprocess communication**
  - pipe(fildes)

System Calls versus Library Routines

The tricky thing about system calls is that they look very much like a library routine (or a regular function) that you have already been using (for instance, printf). The only way to tell which is a library routine and which is a system call is to remember which is which.

Another way to obtain information about the system call is to refer to Section 2 of the man pages. For instance, to find more about the "read" system call you could type:

```
% man -S 2 read
```

By contrast, on our current version of Linux, if you try:

```
% man read
```

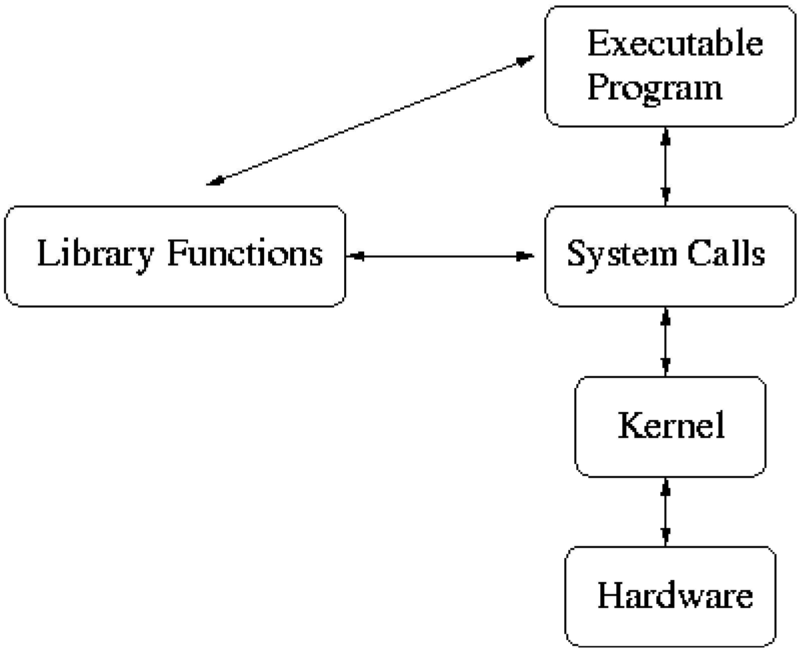
Section 1 of the library will be displayed (indicated by the 1 in parenthesis).

Section 3 contains library routines. By issuing a:

```
% man -S 3 fread
```

you will learn more about the "fread" library routine.

Some library functions have embedded system calls. For instance, the library routines scanf and printf make use of the system calls read and write. The relationship of library functions and system calls is shown in the below diagram (taken from John Shapley Gray's *Interprocess Communications in UNIX*)



"The arrows in the diagram indicate possible paths of communication. As shown, executable programs may make use of system calls directly to request the kernel to perform a specific function. Or, the executable program may invoke a library function which in turn may perform system calls." (page 4 and 5, *Interprocess Communications in UNIX*).

Using System Calls for File I/O

The main systems calls that will be needed for this lab are

- open()
- close()
- read()
- write()
- stat()
- fstat()

Each of these will be covered in their own subsection

open():

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int flags [, mode_t mode ]);
```

path is a string that contains the name and location of the file to be opened.

The mode of the file to be opened is determined by the oflag variable. It must have one of the following values:

O_RDONLY	Read only mode
O_WRONLY	Write only mode
O_RDWR	Read/Write mode

and may have one or more options bitwise OR-ed on. A few examples of options include:

O_APPEND	If the file exists, append to it.
O_CREAT	If the file does not exist create it.
O_EXCL	(Only with O_CREAT.) If the file already exists, open fails and returns an error.
O_TRUNC	If the file exists and is being opened for writing, truncate it to length 0.

Upon success, open returns a file descriptor. If the operation fails, a -1 is returned. Use the 'man -S 2 open' command for more information on this system call.

The convenience system call creat(const char *path, mode_t mode) is equivalent to open(path, O_WRONLY|O_CREAT|O_TRUNC, mode). Use the 'man -S 2 creat' command for more information on this command.

When you use the O_CREAT flag you are required to provide the mode argument to set access permissions for the new file. The permissions you supply will be AND-ed against the complement of the user's umask. For example:

```
outFile=open("myfile",O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
```

In the above example, the permissions set on "myfile" will be read and write by the user (S_IRUSR | S_IWUSR). For more details on these modes or permissions see man -S 2 open.

close():

```
  #include < unistd.h>
  int close(int fildes);
```

close() closes the file indicated by the file descriptor fildes.

The operating system will free any resources allocated to the file during its operation. Use the 'man -S 2 close' command for more information on this system call.

read():

```
  #include < unistd.h>
  ssize_t read(int fildes, void *buf, size_t nbyte);
```

read() attempts to read nbyte bytes from the file associated with fildes into the buffer pointed to by buf.

If nbyte is zero, read returns zero and has no other results. On success a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned. Use the 'man -S 2 read' command for more information on this system call.

write():

```
#include < unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

write() attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with fildes.

If nbyte is zero and the file is a regular file, write returns zero and has no other results. On success, write returns the number of bytes actually written. Otherwise, it returns -1. Use the command 'man -S 2 write' for more information on this system call.

The following is a sample program reads a file and displays its contents on the screen.

```
//Modified from page 7 of Interprocess Communication in Unix by John
//Shapely Gray
//Usage: a.out filename
//Displays the contents of filename

#include <cstdio>
#include <unistd.h>
#include <cstdlib>
#include <sys/types.h> //needed for open
#include <sys/stat.h> //needed for open
#include <fcntl.h>     //needed for open

using namespace std;

int main (int argc, char *argv[])
{
        int inFile;
        int  n_char=0;
           char buffer[10];

        inFile=open(argv[1],O_RDONLY);
        if (inFile==-1)
        {
                exit(1);
        }
        //Use the read system call to obtain 10 characters from inFile
        while( (n_char=read(inFile, buffer, 10))!=0)
        {
                //Display the characters read
                n_char=write(1,buffer,n_char);
        }

        return 0;
}
```

You will notice that the first argument to a read/write system call is an integer value indicating the file descriptor. When a program executes, the operating system will automatically open three file descriptors:

- 0 stdin --standard input (defaults to the keyboard)
- 1 stdout --standard output (defaults to the terminal)
- 2 stderr --standard error (defaults to the console device)

A write to file descriptor 1 (as in the above code) will be writing to your terminal.

stat():

```
#include < sys/types.h>
#include < sys/stat.h>
int stat(const char *path, struct stat *buf)
```

stat() obtains information about the named file, and puts them in the stat structure.

This command comes in handy if you want to emulate the ls command. All the information that you need to know about a file is given in the stat structure.

The stat structure is defined as:

```
mode_t st_mode;          /* protection */

ino_t st_ino;            /* this file's number */

dev_t st_dev;            /* device file resides on */

dev_t st_rdev;           /* device identifier (special files only) */

nlink_t st_nlink;        /* number of hard links to the file*/
```

```
uid_t st_uid;           /* user ID of owner */

gid_t st_gid;           /* group ID of owner */

off_t st_size;          /* file size in bytes */

time_t st_atime;        /* time of last access */

time_t st_mtime;        /* time of last data modification */

time_t st_ctime;        /* time of last file status change*/

blksize_t st_blksize;   /* preferred I/O block size */

blkcnt_t st_blocks;     /* number 512 byte blocks allocated */
```

Use the 'man -S 2 stat' command for more information on this system call.

fstat():

```
#include < sys/types.h>
#include < sys/stat.h>
int fstat(int fildes, struct stat *buf)
```

fstat() is like stat(), but works on a file that is specified by the fildes file descriptor. Use the 'man -S 2 fstat' command for more information on the fstat system call.

Example

```
struct                          stat                          statBuf;

int err, FD;
FD = open("openclose.in", O_WRONLY | O_CREAT, S_IREAD | S_IWRITE);

if(FD == -1) /* open failed? */
        exit(-1);

err = fstat(FD, &statBuf);
if(err == -1) /* fstat failed? */
        exit(-1);

printf("The number of blocks = %d\n", statBuf.st_blocks);
```

perror()

In most cases, if a system call or library function fails, it returns a value of -1 and assigns a value to an external variable called errno. This value indicates where the actual problem occurred.

It is a good programming habit to examine the return value from a system call or library function to determine if something went wrong. If there was a failure, the program should do something such as display a short error message and exit (terminate) the program. The library function perror can be used to produce an error message.

The following is an example of using perror to provide some error checking:

```
//Modified from page 7 of Interprocess Communication in Unix by John
//Shapely Gray

//checking errno and using perror


#include <cstdio>
#include <unistd.h>
#include <cstdlib>
#include <errno.h>      //must use for perror


using namespace std;

//extern int errno;  //in linux, don't seem to need this


int main (int argc, char *argv[])
{
        int  n_char=0;
           char buffer[10];

        //Initially n_char is set to 0 -- errno is 0 by default
        printf("n_char = %d \t errno = %d\n", n_char, errno);

        //Display a prompt to stdout
        n_char=write(1, "Enter a word  ", 14);

        //Use the read system call to obtain 10 characters from stdin
        n_char=read(0, buffer, 10);
        printf("\nn_char = %d \t errno = %d\n", n_char, errno);

        //If read has failed
        if (n_char==-1)
        {
                perror(argv[0]);
                exit (1);
        }

        //Display the characters read
        n_char=write(1,buffer,n_char);

        return 0;
}
```

This is what the program will do if you run it and type in the word "hello":

```
% a.out
n_char = 0      errno = 0
Enter a word  hello

n_char = 6      errno = 0
hello
```

If you want it to display a error message, change the file number for the read system call to 3 (a file number that we haven't opened)

```
% a.out
n_char = 0 errno = 0
Enter a word
n_char = -1 errno = 9
a.out: Bad file descriptor
```

File permissions –chmod(),fchmod()

Name

chmod - change mode of a file

Synopsis

#include <sys/stat.h>
int chmod(const char **path*, mode_t *mode*);
int fchmod(int *fd*, mode_t *mode*);

Description

These system calls change the permissions of a file. They differ only in how the file is specified:

chmod() changes the permissions of the file specified whose pathname is given in *path*, which is dereferenced if it is a symbolic link.
fchmod() changes the permissions of the file referred to by the open file descriptor *fd*.
The new file permissions are specified in *mode*, which is a bit mask created by ORing together zero or more of the following:
S_ISUID (04000)
set-user-ID (set process effective user ID on **execve**(2))
S_ISGID (02000)
set-group-ID (set process effective group ID on **execve**(2); mandatory locking, as describedin **fcntl**(2); take a new file's group from parent directory, as described in **chown**(2) and **mkdir**(2))
S_ISVTX (01000)sticky bit (restricted deletion flag, as described in **unlink**(2))
S_IRUSR (00400)read by owner
S_IWUSR (00200)write by owner
S_IXUSR (00100)execute/search by owner ("search" applies for directories, and means that entries within the directory can be accessed)
S_IRGRP (00040)read by group
S_IWGRP (00020)write by group
S_IXGRP (00010)execute/search by group
S_IROTH (00004)read by others
S_IWOTH (00002)write by others
S_IXOTH (00001)execute/search by others
Return Value
On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

File ownership –chown, lchown, fchown

Name

chown, fchown, lchown - change ownership of a file

Synopsis

#include <unistd.h>

int chown(const char ****path*, uid_t** *owner***, gid_t***group***);**
int fchown(int *fd***, uid_t** *owner***, gid_t** *group***);**
int lchown(const char ****path*, uid_t** *owner***, gid_t***group***);**

Description

These system calls change the owner and group of a file. They differ only in how the file is specified:

chown() changes the ownership of the file specified by *path*, which is dereferenced if it is a symbolic link.

fchown() changes the ownership of the file referred to by the open file descriptor *fd*.

lchown() is like **chown**(), but does not dereference symbolic links.

Only a privileged process (Linux: one with the **CAP_CHOWN** capability) may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member. A privileged process (Linux: with**CAP_CHOWN**) may change the group arbitrarily.

If the *owner* or *group* is specified as -1, then that ID is not changed.

When the owner or group of an executable file are changed by an unprivileged userthe **S_ISUID** and **S_ISGID** mode bits are cleared. POSIX does not specify whether this also should happen when root does the **chown**(); the Linux behavior depends on the kernel version. Incase of a non-group-executable file (i.e., one for which the **S_IXGRP** bit is not set) the**S_ISGID** bit indicates mandatory locking, and is not cleared by a **chown**().

Return Value

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

Links –soft links and hard links –symlink, link, unlink

Name
link - make a new name for a file
Synopsis
#include <unistd.h>
int link(const char ***oldpath, const char ***newpath**);**
Description
link() creates a new link (also known as a hard link) to an existing file.
If newpath exists it will not be overwritten.
This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".
Return Value
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.
Name
symlink - make a symbolic link to a file
Synopsis
#include <unistd.h>
int symlink(const char *path1**, const char ***path2**);**

Description
The symlink() function shall create a symbolic link called path2 that contains the string pointedto by path1(path2 is the name of the symbolic link created, path1is the string contained in the symbolic link).
The string pointed to by path1 shall be treated only as a character string and shall not bevalidated as a pathname.
If the symlink() function fails for any reason other than [EIO], any file named by path2 shall be unaffected.
Return Value
Upon successful completion, symlink() shall return 0; otherwise, it shall return -1 andset errno to indicate the error.
Name
unlink - delete a name and possibly the file it refers to
Synopsis
#include <unistd.h>
int unlink(const char ***pathname);**
Description
unlink() deletes a name from the file system. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.
If the name was the last link to a file but any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed.
If the name referred to a symbolic link the link is removed.
If the name referred to a socket, fifo or device the name for it is removed but processes which have the object open may continue to use it.
Return Value
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Directories –Creating, removing and changing Directories –mkdir, rmdir,chdir,obtaining current working directory –getcwd

Name
mkdir - create a directory
Synopsis
#include <sys/stat.h>#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
Description
mkdir() attempts to create a directory named*pathname*.
The argument *mode* specifies the permissions to use. It is modified by the process's *umask* in the usual way: the permissions of the created directory are (*mode* & ~*umask* & 0777). Other mode bits of the created directory depend on the operating system. For Linux, see below.
The newly created directory will be owned by the effective user ID of the process. If the directory containing the file has the set-group-ID bit set, or if the file system is mounted with

BSD group semantics (*mount -o bsdgroups* or, synonymously *mount -o grpid*), the new directory will inherit the group ownership from its parent; otherwise it will be owned by the effective group ID of the process.
If the parent directory has the set-group-ID bit set then so will the newly created directory.
Return Value
mkdir() returns zero on success, or -1 if an error occurred (in which case, *errno* is set appropriately).
Name
rmdir - delete a directory
Synopsis
#include <unistd.h>
int rmdir(const char ***pathname*);**
Description
rmdir() deletes a directory, which must be empty.
Return Value
On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.
Name
chdir, fchdir - change working directory
Synopsis
#include <unistd.h>
int chdir(const char ***path*);**
int fchdir(int *fd***);**
Description
chdir() changes the current working directory of the calling process to the directory specified in *path*.
fchdir() is identical to **chdir**(); the only difference is that the directory is given as an open file descriptor.
Return Value
On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.
Name
getcwd, getwd, get_current_dir_name - get current working directory
Synopsis
#include <unistd.h>

char *getcwd(char ***buf*, size_t** *size***);**

char *getwd(char ***buf*);**

char *get_current_dir_name(void);
Description
These functions return a null-terminated string containing an absolute pathname that is the current working directory of the calling process. The pathname is returned as the function result and via the argument*buf*, if present.
The **getcwd**() function copies an absolute pathname of the current working directory to the array pointed to by*buf*, which is of length *size*.

If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds *size* bytes, NULL is returned, and *errno* is set to **ERANGE**; an application should check for this error, and allocate a larger buffer if necessary.
As an extension to the POSIX.1-2001 standard, Linux (libc4, libc5, glibc) **getcwd**() allocates the buffer dynamically using **malloc**(3) if *buf* is NULL. In this case, the allocated buffer has the length *size* unless *size* is zero, when *buf* is allocated as big as necessary. The caller should **free**(3)the returned buffer.
get_current_dir_name() will **malloc**(3) an array big enough to hold the absolute pathname of the current working directory. If the environment variable **PWD** is set, and its value is correct, then that value will be returned. The caller should **free**(3) the returned buffer.
getwd() does not **malloc**(3) any memory. The *buf* argument should be a pointer to an array at least **PATH_MAX** bytes long. If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds**PATH_MAX** bytes, NULL is returned, and *errno* is set to **ENAMETOOLONG**. (Note that on some systems, **PATH_MAX** may not be a compile-time constant; furthermore, its value may depend on the file system, see **pathconf**(3).) For portability and security reasons, use of **getwd**() is deprecated.
Return Value
On success, these functions return a pointer to a string containing the pathname of the current working directory. In the case**getcwd**() and **getwd**() this is the same value as *buf*.
On failure, these functions return NULL, and *errno* is set to indicate the error. The contents ofthe array pointed to by *buf* are undefined on error.

Directory contents, Scanning Directories- opendir, readdir, closedir, rewinddir functions.

Name
opendir, fdopendir - open a directory
Synopsis
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char ***name*);**
DIR *fdopendir(int *fd***);**
Description
The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.
The **fdopendir**() function is like **opendir**(), but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to**fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.
Return Value
The **opendir**() and **fdopendir**() functions return a pointer to the directory stream. On error, NULL is returned, and *errno* is set appropriately.

Name

readdir, readdir_r - read a directory

Synopsis

#include <dirent.h>

struct dirent *readdir(DIR ***dirp*);**

int readdir_r(DIR ***dirp*, struct dirent ***entry***, struct dirent ****result***);**

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

readdir_r():

_POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _BSD_SOURCE || _SVID_SOURCE || _POSIX_SOURCE

Description

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

On Linux, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

The only fields in the *dirent* structure that are mandated by POSIX.1 are: *d_name*[], of unspecified size, with at most**NAME_MAX** characters preceding the terminating null byte; and (as an XSI extension) *d_ino*. The other fields are unstandardized, and not present on all systems; see NOTES below for some further details.

The data returned by **readdir**() may be overwritten by subsequent calls to **readdir**() for the same directory stream.

The **readdir_r**() function is a reentrant version of **readdir**(). It reads the next directory entry from the directory stream *dirp*, and returns it in the caller-allocated buffer pointed to by *entry*. (See NOTES for information on allocating this buffer.) A pointer to the returned item is placedin **result*; if the end of the directory stream was encountered, then NULL is instead returnedin **result*.

Return Value

On success, **readdir**() returns a pointer to a *dirent*structure. (This structure may be statically allocated; do not attempt to **free**(3) it.) If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno*is set appropriately.

The **readdir_r**() function returns 0 on success. On error, it returns a positive error number (listed under ERRORS). If the end of the directory stream is reached, **readdir_r**() returns 0, and returns NULL in**result*.

Name
closedir - close a directory
Synopsis
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
Description
The **closedir**() function closes the directory stream associated with *dirp*. A successful call to **closedir**() also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.
Return Value
The **closedir**() function returns 0 on success. On error, -1 is returned, and *errno* is set appropriately.

Name
rewinddir - reset directory stream
Synopsis
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dirp);
Description
The **rewinddir**() function resets the position of the directory stream *dirp* to the beginning of the directory.
Return Value
The **rewinddir**() function returns no value.

UNIT III

Syllabus

Process – Process concept, Layout of a C program in main memory, Process environment – environment list, environment variables, getenv, setenv, kernel support for process, process identification, process control –process creation , replacing a process image, waiting for a process, process termination, zombie process, orphan process, system calls interface for process management –fork, vfork, exit, wait, waitpid, exec family, process Groups, Sessions and Controlling Terminal, Differences between threads and processes. Signals: -Introduction to signals, signal generation and handling, kernel support for signals, signal function, unreliable signals, reliable signals, kill, raise, alarm, pause, abort, sleep functions.

Process Concept

- A process is an instance of a program in execution.
- Batch systems work in terms of "jobs". Many modern process concepts are still expressed in terms of jobs, (e.g. job scheduling), and the two terms are often used interchangeably.

The Process

- Process memory is divided into four sections as shown in Figure 3.1 below:
    - The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
    - The data section stores global and static variables, allocated and initialized prior to executing main.
    - The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
    - The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope. Note that the stack is also used for functionreturn values, and the exact mechanisms of stack management may be language specific.
    - Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.

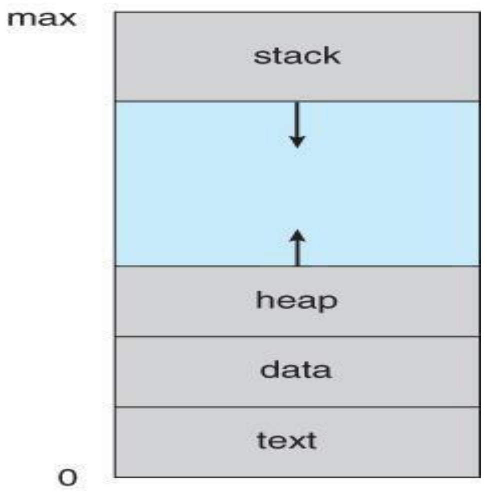


Figure - A process in memory

Process State

- Processes may be in one of 5 states, as shown in Figure below.
    - **New -** The process is in the stage of being created.
    - **Ready -** The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
    - **Running -** The CPU is working on this process's instructions.
    - **Waiting -** The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
    - **Terminated -** The process has completed.
- The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.
- Some systems may have other states besides the ones listed here.

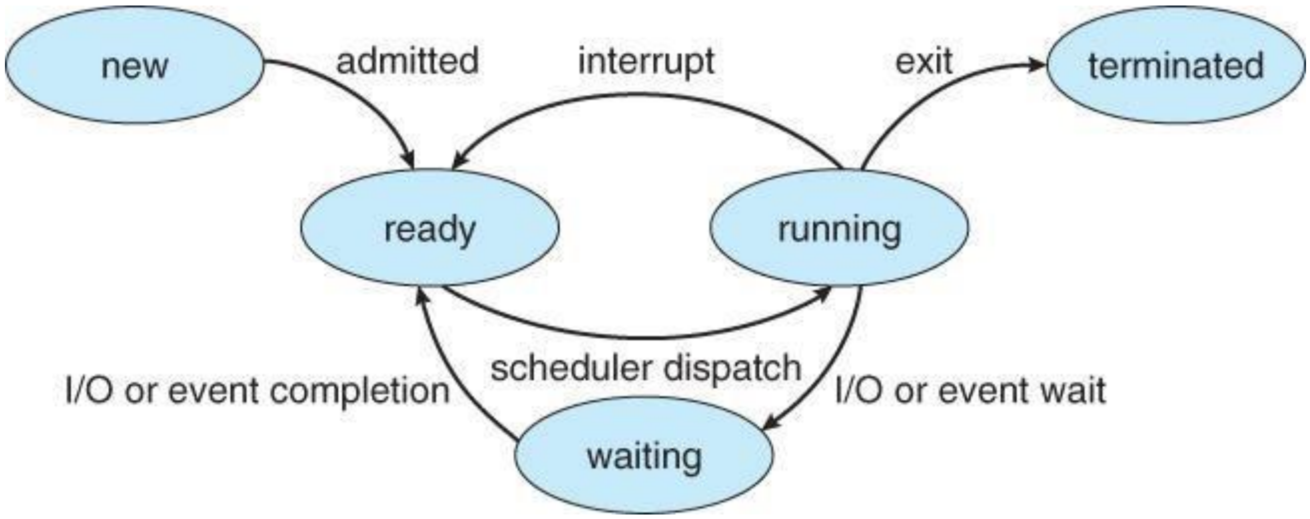


Figure - Diagram of process state

Process Control Block

For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information, as illustrated in Figure 3.1. (Specific details may vary from system to system.)

- **Process State -** Running, waiting, etc., as discussed above.
- **Process ID**, and parent process ID.
- **CPU registers and Program Counter -** These need to be saved and restored when swapping processes in and out of the CPU.
- **CPU-Scheduling information -** Such as priority information and pointers to scheduling queues.
- **Memory-Management information -** E.g. page tables or segment tables.
- **Accounting information -** user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information -** Devices allocated, open file tables, etc.

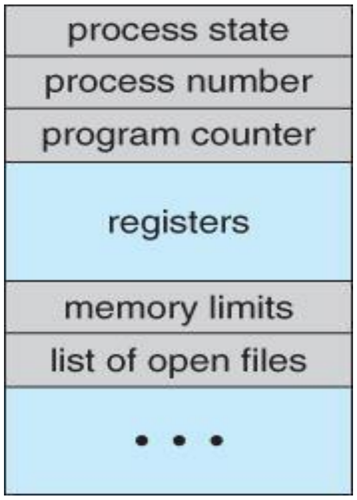
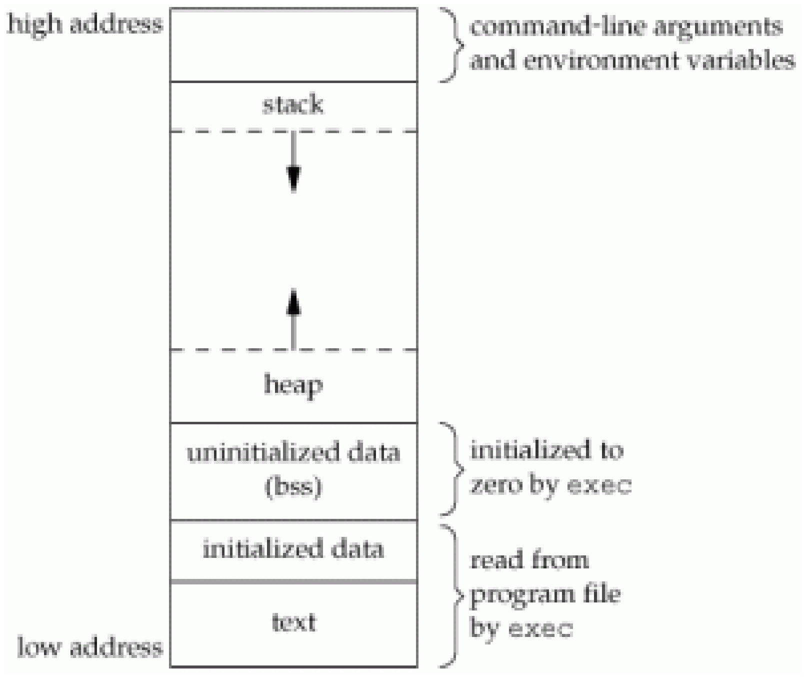


Figure - Process control block (PCB)

Memory Layout of C Programs

A typical memory representation of C program consists of following sections.
1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

A typical memory layout of a running process

1. Text Segment:
A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.
Note that, data segment is not read-only, since the values of the variables can be altered at run time.
This segment can be further classified into initialized read-only area and initialized read-write area.
For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = “hello world” makes the string literal “hello world” to be stored in initialized read- only area and the character pointer variable string in initialized read-write area.
Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment:

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing
uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
For instance a variable declared static int i; would be contained in the BSS segment.For instance a global variable declared int j; would be contained in the BSS segment.

4. Stack:

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)
The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.
Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

5. Heap:

Heap is the segment where dynamic memory allocation usually takes place.
The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.
Examples.
The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. (for more details please refer man page of size(1))

1. Check the following simple C program

```
#include<stdio.h>
Int main(void)
{
Return 0;
}
```

OUTPUT:

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text     data     bss      dec      hex  filename
960       248       8      1216      4c0    memory-layout
```

Process environment –environment list, environment variables, getenv, setenv:

- A set of name-value pairs associated with a process
- Keys and values are strings
- Passed to children processes
- Cannot be passed back up
- Common examples: – PATH: Where to search for programs

Environment Variables

Variable	**Description**
PATH	Indicates search path for commands. It is a colon-separated list of directories in which the shell looks for commands.
PWD	Indicates the current working directory as set by the cd command.
RANDOM	Generates a random integer between 0 and 32,767 each time it is referenced.
$HOME	Absolute pathname of your home directory
$PATH	A list of directories to search for
$MAIL	Absolute pathname to mailbox
$USER	Your user id
$SHELL	Absolute pathname of login shell
$TERM	Type of your terminal
$PS1	Prompt

NAME

getenv, secure_getenv - get an environment variable

SYNOPSIS

#include <stdlib.h>

char *getenv(const char ***_name_);**

char *secure_getenv(const char ***name*);**

DESCRIPTION

The **getenv**() function searches the environment list to find the environment variable *name,* and returns a pointer to the corresponding *value* string.

The GNU-specific **secure_getenv**() function is just like **getenv**() except that it returns NULL in cases where "secure execution" is required. Secure execution is required if one of the following conditions was true when the program run by the calling process was loaded:

RETURN VALUE

The **getenv**() function returns a pointer to the value in the environment, or NULL if there is no match.

NAME

setenv - change or add an environment variable

SYNOPSIS

#include <stdlib.h>

int setenv(const char ***name*, const char ***value***, int** *overwrite***);**

int unsetenv(const char ***name*);**

DESCRIPTION

The **setenv**() function adds the variable *name* to the environment with the value *value,* if *name* does not already exist. If *name* does exist in the environment, then its value is changed to *value* if *overwrite* is nonzero; if *overwrite* is zero, then the value of *name* is not changed (and **setenv**() returns a success status). This function makes copies of the strings pointed to by *name* and *value* (by contrast with putenv(3)).The **unsetenv**() function deletes the variable *name* from the environment. If *name* does not exist in the environment, then the function succeeds, and the environment is unchanged.

RETURN VALUE

The **setenv**() function returns zero on success, or -1 on error, with *errno* set to indicate the cause of the error.The **unsetenv**() function returns zero on success, or -1 on error, with *errno* set to indicate the cause of the error.

Kernel support for process

Each of these processes could be responsible for a logical subset of the services offered by the kernel. Thus one process could be responsible for scheduling, another for managing disk I/O, a third for processing information received from other computers and so on.

There are some advantages of supporting kernel processes:

The kernel is well-structured since kernel services are encapsulated within the processes that provide them. This encapsulation is similar to the one provided by modules and monitors. (Recall the Lauer and Needham paper on the duality of operating systems).

Kernel code does not have to run as part of user processes. In a system that does not support kernel processes, all kernel code runs as part of user processes. For example in Xinu, when a service call is made, the kernel code to service the call runs as part of the process that made the call. In a system that supports kernel processes, the service call would result in a message to a kernel process, which would then execute the appropriate code. In some ways the latter approach is more `elegant'.

In a multiprocessor system kernel processes could execute concurrently.

The idea of kernel processes may seem contradictory to our discussion of the minimum functionality provided by a kernel. If a kernel is responsible for supporting processes and interprocess communication, how can it use these facilities itself? Here are two solutions:

Kernel processes and communication among them may be supported by the programming language. Charlotte, a distributed operating system developed at Madison, uses this approach.

A portion of the kernel, which we shall call the **nugget**, could support kernel processes and communication among these processes, and higher layers could be implemented as communicating kernel processes. This approach is implemented in Sun Unix.

Should kernel processes be any different from ordinary user processes? It is important to execute kernel code efficiently, therefore these processes should be lightweight processes.

Process identification:

A PID (i.e., **process identification** number) is an **identification** number that is automatically assigned to each **process** when it is created on a **Unix**-like operating system. A **process** is an executing (i.e., running) instance of a program.

Process control –process creation

Process creation in UNIX

The seven-state logical process model we considered in a previous lecture can accommodate the UNIX process model with some modifications, actually becoming a ten state model.

First, as we previously observed, UNIX executes most kernel services within a process's context, by implementing a mechanism which separates between the two possible modes of execution of a process. Hence our previously unique ``Running" state must actually be split in a ``User Running" state and a ``Kernel Running" state. Moreover a process preemption mechanism is usually implemented in the UNIX scheduler to enforce priority. This allows a process returning from a system call (hence after having run in kernel mode) to be immediately blocked and put in the ready processes queue instead of returning to user mode running, leaving the CPU to another process. So it's worth considering a ``Preempted" state as a special case of ``Blocked". Moreover,among exited processes there's a distinction between those which have a parent process thatwaits for their completion (possibly to clean after them), and those which upon termination have an active parent that might decide to wait for them sometime in the future (and then be immediately notified of its children's termination)�. These last processes are called ``Zombie",while the others are ``Exited". The difference is that the system needs to maintain an entry in the process table for a zombie, since its parent might reference it in the future, while the entry for an exited (and waited for) process can be discarded without further fiddling. So the much talked about ``Zombie" processes of UNIX are nothing but entries in a system table, the system having already disposed of all the rest of their image. This process model is depicted in fig.

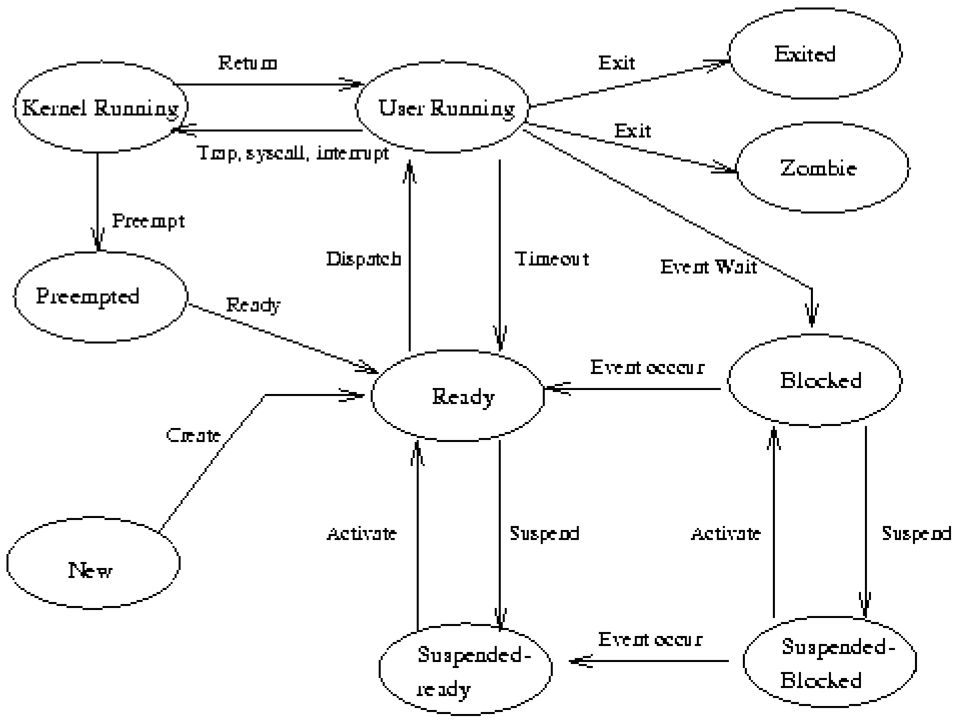


Figure : UNIX process state model

All processes in UNIX are created using the fork() system call. System calls in UNIX can be best thought of as C functions provided with the standard C library. Even if their particular implementation is depends on the particular UNIX flavor (and on hardware, for many of them), a C API is always provided, and is consistent among the different unices, at least in the fundamental traits.

UNIX implements through the fork() and exec() system calls an elegant two-step mechanism for process creation and execution. fork() is used to create the image of a process using the one of an existing one, and exec is used to execute a program by overwriting that image with the program's one. This separation allows to perform some interesting housekeeping actions in between, as we'll see in the following lectures.

A call to fork() of the form:

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

pid_t childpid;
...
childpid = fork(); /* child's pid in the parent, 0 in the child */
...
```

creates (if it succeeds) a new process, which a child of the caller's, and is an exact copy of of the (parent) caller itself. By exact copy we mean that it's image is a physical bitwise copy of the parent's (in principle, they *do not* share the image in memory: though there can be exceptions to this rule, we can always thing of the two images as being stored in two separate and protected address spaces in memory, hence a manipulation of the parent's variables won't affect the child's copies, and vice versa). The only visible differences are in the PCB, and the most relevant (for now) of them are the following:

- The two processes obviously have two different process id.s. (pid). In a C program process id.s are conveniently represented by variables of pid_t type, the type being defined in the sys/types.h header.
- In UNIX the PCB of a process contains the id of the process's parent, hence the child's PCB will contain as parent id (ppid) the pid of the process that called fork(), while the caller will have as ppid the pid of the process that spawned it.
- The child process has its own copy of the parent's file descriptors. These descriptors reference the same under-lying objects, so that files are shared between the child and the parent. This makes sense, since other processes might access those files as well, and having them already open in the child is a time-saver.

The fork() call returns in both the parent and the child, and both resume their execution from the statement immediately following the call. One usually wants that parent and child behave differently, and the way to distinguish between them in the program's source code is to test the value returned by fork(). This value is 0 in the child, and the child's pid in the parent. Since fork() returns -1 in case the child spawning fails, a catch-all C code fragment to separate behaviours may look like the following:

```
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
...
pid_t childpid;
...
childpid=fork();
switch(childpid)
{
  case -1:
    fprintf(stderr,"ERROR: %s\n", sys_errlist[errno]);
    exit(1);
    break;
  case 0:
    /* Child's code goes here */
    break;
  default:
    /* Parent's code goes here */
    break;
}
```

The array of strings char *sys_errlist[] and the global integer variable int errno are defined in the errno.h header. The former contains a list of system error messages, and the latter is set to index the appropriate message whenever an error occurs. For each system call several possible error conditions are defined. Each of them is associated to an integer constant - defined viaa #define directive in one system header file - whose value is exactly the one that errno takes when an error occurs.

The vfork() system call
A BSD variant of *fork*(), now supported by SVR4.
Similar to *fork*(); however, is used to *exec* a new program only.
Child running in the parent address space until it calls *exec*()/*exit*().
Not fully copying the address space of the parent into the child.
vfork() guarantees that the child runs first until it calls *exec*()/*exit*().
Deadlock is possible if the child needs information from the parent.

waiting for a process

wait()

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

pid_t wait(int *status);
```

A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions.

The wait() system call allows the parent process to suspend its activities until one of these actions has occurred.

The wait() system call accepts a single argument, which is a pointer to an integer and returns a value defined as type pid_t.

If the calling process does not have any child associated with it, wait will return immediately with a value of -1.

If any child processes are still active, the calling process will suspend its activity until a child process terminates.
(see the programs in ***Interprocess Communication in Unix*** pg 73 and 74)

Example of wait()

```
int status;
pid_t fork_return;
```

```
fork_return = fork();

if (fork_return == 0) /* child process */
{
  printf("\n I'm the child!");
  exit(0);
}
else /* parent process */
{
  wait(&status);
  printf("\n I'm the parent!");
  if (WIFEXITED(status))
      printf("\n Child returned: %d\n", WEXITSTATUS(status));
}
```

A few notes on this program:

- wait(&status) causes the parent to sleep until the child process is finished execution
- details of how the child stopped are returned via the status variable to the parent. Several macros are available to interpret the information. Two useful ones are:
    - WIFEXITED evaluates as true, or 0, if the process ended normally with an exit or return call.
    - WEXITSTATUS if a process ended normally you can get the value that was returned with this macro.

  Consult a man file for more.

exec*()

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const  char  *path,  const  char  *arg  , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

"The exec family of functions replaces the current process image with a new process image." (man pages)

Commonly a process generates a child process because it would like to transform the child process by changing the program code the child process is executing.

The text, data and stack segment of the process are replaced and only the **u** (user) area of the process remains the same.

If successful, the **exec** system calls do not return to the invoking program as the calling image is lost.

It is possible for a user at the command line to issue an exec system call, but it takes over the current shell and terminates the shell.

```
% exec  command  [arguments]
```

The versions of exec are:

- execl
- execv
- execle
- execve
- execlp
- execvp

The naming convention: exec*

- 'l' indicates a list arrangement (a series of null terminated arguments)
- 'v' indicate the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their own environment variable list
- 'p' indicates the current PATH string should be used when the system searches for executable files.

  NOTE:

- In the four system calls where the PATH string is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

exec system call functionality

Library Call Name	Argument List	Pass Current Environment Variables	Search PATH automatic?
execl	list	yes	no

execv	array	yes	no
execle	list	no	no
execve	array	no	no
execlp	list	yes	yes
execvp	array	yes	yes

execlp

- this system call is used when the number of arguments to be passed to the program to be executed is known in advance

execvp

- this system call is used when the numbers of arguments for the program to be executed is dynamic

```
/* using execvp to execute the contents of argv */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  execvp(argv[1], &argv[1]);
  perror("exec failure");
  exit(1);
}
```

Things to remember about exec*:

- this system call simply replaces the current process with a new program -- the pid does not change
- the exec() is issued by the calling process and what is exec'ed is referred to as the new program -- not the new process since no new process is created
- it is important to realize that control is not passed back to the calling process unless an error occurred with the exec() call
- in the case of an error, the exec() returns a value back to the calling process
- if no error occurs, the calling process is lost

A few more Examples of valid exec commands:

```
execl("/bin/date","",NULL); // since the second argument is the program name,
                          // it may be null

execl("/bin/date","date",NULL);

execlp("date","date", NULL); //uses the PATH to find date, try: %echo $PATH
```

getpid()

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

    pid_t getpid(void);
    pid_t getppid(void);
```

getpid() returns the process id of the current process. The process ID is a unique positive integer identification number given to the process when it begins executing.

getppid() returns the process id of the parent of the current process. The parent process forked the current child process.

getpgrp()

```
    #include <unistd.h>

    pid_t getpgrp(void);
```

Every process belongs to a process group that is identified by an integer process group ID value.When a process generates a child process, the operating system will automatically create a process group.The initial parent process is known as the process leader.getpgrp() will obtain the process group id.

Process termination

Normal termination
  Return from main().

Calling *exit()*
Calling *_exit()*.
Abnormal termination

Calling *abort()*.

Terminated by a signal.

The exit() system call
 Performs a standard I/O cleanup
 Executes all registered *exit handlers*.
 Flushes all C/C++ output buffers.
 Closes all open streams.
Terminates the calling process.
The _exit() system call
Terminates the calling process without performing some cleanup.

Zombie process:

On Unix and Unix-like computer operating systems, **a zombie process or defunct process is a process that has completed execution but still has an entry in the process table**. This entry is still needed to allow the parent process to read its child's exit status. The term zombie process derives from the common definition of zombie — an undead person. In the term's metaphor, the child process has "died" but has not yet been "reaped". Also, unlike normal processes, the kill command has no effect on a zombie process.
When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, whereupon the zombie is removed. The wait call may be executed in sequential code, but it is commonly executed in a handler for theSIGCHLD signal, which the parent receives whenever a child has died.
After the zombie is removed, its process identifier (PID) and entry in the process table can then be reused. However, if a parent fails to call wait, the zombie will be left in the process table. In some situations this may be desirable, for example if the parent creates another child process it ensures that it will not be allocated the same PID. On modern UNIX-like systems (that comply with SUSv3 specification in this respect), the following special case applies: if the parentexplicitly ignores SIGCHLD by setting its handler toSIG_IGN (rather than simply ignoring the signal by default) or has the SA_NOCLDWAIT flag set, all child exit status information will be discarded and no zombie processes will be left
A **zombie process is not the same as an orphan process**. An orphan process is a process that is still executing, but whose parent has died. They do not become zombie processes; instead, they are adopted by init (process ID 1), which waits on its children.

Orphan Process

An orphan process is a computer **process whose parent process has finished or terminated**, though it remains running itself.
In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called re-parenting and occurs automatically. Even though technically the process has the init process as its parent, it is still called an orphan processsince the process that originally created it no longer exists.

A process can be *orphaned unintentionally*, such as when the parent process terminates orcrashes. The process group mechanism in most Unix-like operation systems can be used to help protect against accidental orphaning, where in coordination with the user’s shell will try to terminate all the child processes with the SIGHUPprocess signal, rather than letting them continue to run as orphans.
A process may also be *intentionally orphaned* so that it becomes detached from the user’ssession and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under Unix, the latter kinds of processes are typically called daemon processes. The Unix nohup command is one means to accomplish this.

Differences between threads and processes:

1. Threads are easier to create than processes since they don't require a separate address space.

2. Multithreading requires careful programming since threads share data strucures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.

3. Threads are considered lightweight because they use far less resources than processes.

4. Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other. This is really another way of stating #2 above.

5. A process can consist of multiple threads.

Signals: -Introduction to signals:

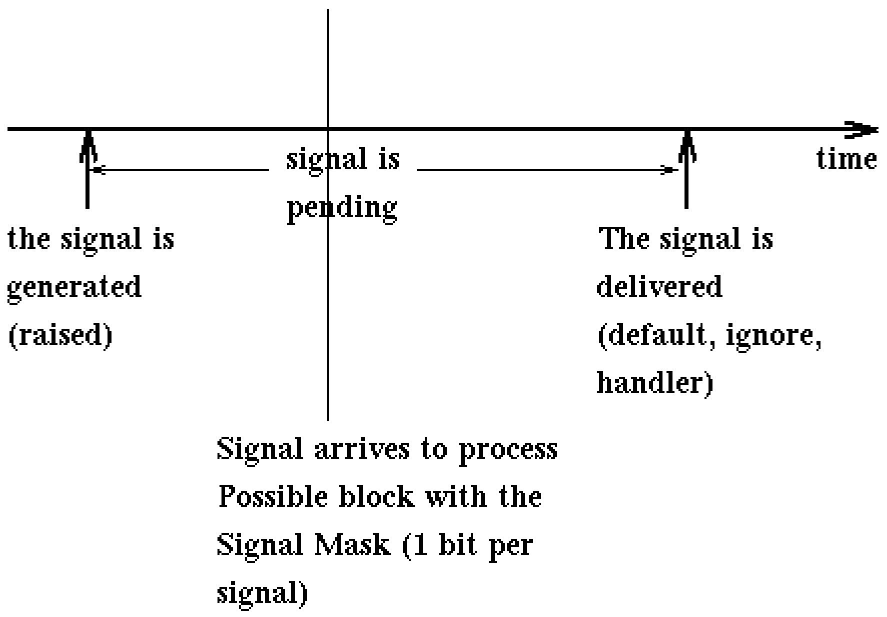
Signals are a limited form of inter-process communication used in **Unix**, **Unix**-like, and other POSIX-compliant operating systems. A **signal** is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred. Signals represent a very limited form of interprocess communication. They are easy to use (hard to use well) but they communicate very little information. In addition the sender (if it is a process) and the receiver must belong to the same user id, or the sender must be the superuser. Signals are sent explicitly to a process from another process using the **kill** function. Signals can indicate some significant terminal action, such as Hang Up. Alternatively, signals are sent to a process from the hardware (to indicate things like illegal operator, or illegal address) through mediation of the OS. Beware that signals could also be caused by an internet connection, for example a TCP/IP OOB (out of band) message could cause the SIGURG signal. There are only a few possible signals (usually 32 - wait: I just checked on my GNU/Linux 2.6.17-1.2142_FC4 and I have now 64 possible signals!)). A process can specify with a **mask** (1 bit per signal) what it wants to be done with a signal directed to it, whether to block it or to deliver it. Blocked signals remain pending, i.e. we may ask to have them delivered later. A process specifies with the **signal** function (obsolete) or with the **sigaction** function what it wants it to be done when signals are delivered to it. When a signal is delivered to a process (i.e. it is not blocked), an action takes place. For each signal there is a default action specific to that signal. Or for a signal an action (a handler) can be specified by the program.

There are three actions that can take place when a signal is delivered to a process:

- it can be ignored; or
- the process can be terminated (with or without core dumping); or
- a handler function can be called. This function receives as its only argument the number identifying the signal it is handling.

The association of a signal to an action remains in place until it is explicitly modified with a signal or sigaction call. [This is the behavior at least in modern Unix systems. In older Unix aftereach signal handling occurrence the handler reverted to the default handler. This created a race condition if we were intending to re-establish a non default handler. For this reason old Unix wassaid to be with**unrelieable signals**.]
In writing a handler function one has to be conscious that it is executed as an asynchronous action during the execution of a process (well, there are also synchronous signals like SIGBUS due to the use of an illegal address). The handler code may execute a routine that was interrupted by the signal, or access variables that were being used in the process. Thus one has to be careful about the code that is executed in the handler and in the variables it accesses. [You may want to examine in C or C++ the use of the attribute **volatile**.] During execution of the handler associated to a signal, that specific signal is automatically blocked thus preventing a race condition. But beware that if the same handler is specified for two different signals A and B, then during the execution of the handler for A, a B signal will not be blocked and the handler will be reentered. After the handler function is completed, execution resumes at the statement being executed when the signal was received. If the signal occurred while the process was executing a system call, things become more comples. The system call may be terminated without completing and return the value EINTR. In some systems it is possible to specify that interrupted system calls be automatically restarted by using the flag SA_RESTART. In other system SA_RESTART is the default.

The following diagram describes how a signal is raised, possibly blocked before delivery, and then handled.


signal is pending
time
the signal is generated (raised)
The signal is delivered (default, ignore, handler)
Signal arrives to process Possible block with the Signal Mask (1 bit per signal)

We may worry that signals may be lost while pending, but that is not the case. If multiple copies of a signal are delivered to a process while that signal is blocked, normally only a single copy of that signal will be delivered to the process when the signal becomes unblocked.

Here are some of the possible signals, with the number associated to them, and their default handling.

SIGNAL	ID	DEFAULT	DESCRIPTION
SIGHUP	1	Termin.	Hang up on controlling terminal
SIGINT	2	Termin.	Interrupt. Generated when we enter CNRTL-C and it is delivered to all processes/threads associated to the current terminal. If generated with kill, it is delivered to only one process/thread.
SIGQUIT	3	Core	Generated when at terminal we enter CNRTL-\
SIGILL	4	Core	Generated when we executed an illegal instruction
SIGTRAP	5	Core	Trace trap (not reset when caught)
SIGABRT	6	Core	Generated by the abort function
SIGFPE	8	Core	Floating Point error
SIGKILL	9	Termin.	Termination (can't catch, block, ignore)
SIGBUS	10	Core	Generated in case of hardware fault
SIGSEGV	11	Core	Generated in case of illegal address
SIGSYS	12	Core	Generated when we use a bad argument in a system service call

SIGPIPE	13	Termin.	Generated when writing to a pipe or a socket while no process is reading at other end
SIGALRM	14	Termin.	Generated by clock when alarm expires
SIGTERM	15	Termin.	Software termination signal
SIGURG	16	Ignore	Urgent condition on IO channel
SIGCHLD	20	Ignore	A child process has terminated or stopped
SIGTTIN	21	Stop	Generated when a backgorund process reads from terminal
SIGTTOUT	22	Stop	Generated when a background process writes to terminal
SIGXCPU	24	Discard	CPU time has expired
SIGUSR1	30	Termin.	User defiled signal 1
SIGUSR2	31	Termin.	User defined signal 2

One can see the effect of these signal by executing in the shell the kill command. For example

```
% kill -TERM pidid
```

You may see what are the defined signals with the shell command

```
% kill -l
```

It is easy to see that some signals occur synchronously (i.e. they are directly related to the instruction being executed) with the executing program (for example SIGSEGV), others asynchronously (for example SIGINT), and others are explicitly directed from one process to another (for example SIGKILL).

Handling and Generating Signals

Now that we have a decent understanding of signals and how they communicate information to a process, let's move on to investigate how we can write program that take some action based on a signal. This is described as **signal handling**, a program that handles a signal, either by ignoring it or taking some action when the signal is delivered. We will also explore how signals can be sent from one program to another, again, we'll use a kill for that.

Hello world of Signal Handling

The primary system call for signal handling is signal(), which given a signal and function, will execute the function whenever the signal is delivered. This function is called the **signal handler** because it *handles* the signal.The signal() function has a strange declaration:

```
int signal(int signum, void (*handler)(int))
```

That is, signal takes two arguments: the first argument is the signal number, such as SIGSTOP or SIGINT, and the second is a reference to a handler function whose first argument is an int and returns void. It's probably best to explore signal() through an example, and hello world program is where we always start.

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

#include <signal.h> /*for signal() and raise()*/
```

```
void hello(int signum){
  printf("Hello World!\n");
}

int main(){

  //execute hello() when receiving signal SIGUSR1
  signal(SIGUSR1, hello);

  //send SIGUSR1 to the calling process
  raise(SIGUSR1);
}
```

The above program first establishes a signal handler for the user signal SIGUSR1. The signal handling function hello() does as expected: prints "Hello World!" to stdout. The program then sends itselfthe SIGUSR1 signal, which is accomplished via raise(), and the result of executing the program is the beautiful phrase:

```
#> ./hello_signal
Hello World!
```

Asynchronous Execution

Some key points to take away from the hello program is that the second argument to signal() is a function pointer, a reference to a function to call. This tells the operating system that whenever this signal is sent to this process, run this function as the signal handler.
Also, the execution of the signal handler is asynchronous, which means the current state of the program will be paused while the signal handler executes, and then execution will resume from the pause point, much like context switching.
Let's look at another example hello world program:

```
void hello(int signum){
  printf("Hello World!\n");
}

int main(){

  //Handle SIGINT with hello
  signal(SIGINT, hello);

  //loop forever!
  while(1);

}
```

The above program will set a signal handler for SIGINT the signal that is generated when you type Ctrl-C. The question is, when we execute this program, what will happen when we type Ctrl-C?
To start, let's consider the execution of the program. It will register the signal handler and then will enter the infinite loop. When we hit Ctrl-C, we can all agree that the signal handler hello() should execute and "Hello World!" prints to the screen, but the program was in an infinite loop. In order to print "Hello World!" it must have been the case that it broke the loop to execute the signal handler, right? So it should exit the loop as well as the program. Let's see:

```
#> ./hello_loop
^CHello World!
```

```
^CHello World!
^CHello World!
^CHello World!
^CHello World!
^CHello World!
^CHello World!
^\Quit: 3
```

As the output indicates, every time we issued Ctrl-C "Hello World!" prints, but the program returns to the infiinite loop. It is only after issue a SIGQUITsignal with Ctrl-\ did the program actually exit.
While the interoperation that the loop would exit is reasonable, it doesn't consider the primary reason for signal handling, that is, asynchronous event handling. That means the signal handler acts out of the standard flow of the control of the program; in fact, the whole program is saved within a context, and a new context is created just for the signal handler to execute in. If you think about it some more, you realize that this is pretty cool, and also a totally new way to view programming.

Inter Process Communication

Signals are also a key means for inter-process communication. One process can send a signal to another indicating that an action should be taken. To send a signal to a particular process, we use the kill() system call. The function declaration is below.

```
int kill(pid_t pid, int signum);
```

Much like the command line version, kill() takes a process identifier and a signal, in this case the signal value as an int, but the value is #defined so you can use the name. Let's see it in use.

```
void hello(){
  printf("Hello World!\n");
}

int main(){

  pid_t cpid;
  pid_t ppid;

  //set handler for SIGUSR1 to hello()
  signal(SIGUSR1, hello);

  if ( (cpid = fork()) == 0){
    /*CHILD*/

    //get parent's pid
    ppid = getppid();

    //send SIGUSR1 signal to parrent
    kill(ppid, SIGUSR1);
    exit(0);
  }else{
    /*PARENT*/
    //just wait for child to terminate
    wait(NULL);
  }

}
```

In this program, first a signal handler is established for SIGUSR1, the hello() function. After the fork, the parent calls wait() and the child will communicate to the parent by "killing" it with the SIGUSR1 signal. The result is that the handler is invoked in the parent and "Hello World!" is printed to stdout from the parent.
While this is a small example, signals are integral to inter process communication. In previous lessons, we've discussed how to communicate data between process with pipe(), signals is the way process communicate state changes and other asynchronous events. Perhaps most relevant is state change in child processes. The SIGCHLD signal is the signal that gets delivered to the parent when a child terminates. So far, we've been handling this signal implicitly through wait(), but you can choose instead to handle SIGCHLD and take different actions when a child terminates. We'll look at that in more detail in a future lesson.

Ignoring Signals

While we've so far looked at changing the actions for a set of signals using a handler. So far, our handlers have been doing things — mostly, printing "Hello World!" — but we might just want our handler to do nothing, essentially, ignoring the signal. That is easy enough to write in code, for example, here is a program that will ignore SIGINT by handling the signal and do nothing:

```
#include <signal.h>
#include <sys/signal.h>

void nothing(int signum){ /*DO NOTHING*/ }

int main(){

  signal(SIGINT, nothing);

  while(1);
}
```

And if we run this program, we see that, yes, it Ctrl-c is ineffective and we have to use Ctrl-\ to quit the program:

```
>./ignore_sigint
^C^C^C^C^C^C^C^C^C^C^\Quit: 3
```

But, it would seem like a pain to always have to write the silly little ignore function that does nothing, and so, when there is a need, there's a way. Thesignal.h header defines a set of actions that can be used in place of the handler:

- SIG_IGN : Ignore the signal
- SIG_DFL : Replace the current signal handler with the default handler

With these keywords, we can rewrite the program simply as:

```
int main(){

  // using SIG_IGN
  signal(SIGINT, SIG_IGN);

  while(1);
}
```

kill and raise Functions

The kill function sends a signal to a process or a group of processes. The raise function allows a process to send a signal to itself.

raise was originally defined by ISO C. POSIX.1 includes it to align itself with the ISO C standard, but POSIX.1 extends the specification of raise to deal with threads (we discuss how threads interact with signals in Section 12.8). Since ISO C does not deal with multiple processes, it could not define a function, such as kill, that requires a process ID argument.

```
#include <signal.h>
 int kill(pid_t pid, int signo);
int raise(int signo);
```

Both return: 0 if OK, 1 on error

The call

raise(signo);
is equivalent to the call

kill(getpid(), signo);

There are four different conditions for the pid argument to kill.

pid > 0	The signal is sent to the process whose process ID is pid.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. Note that the term all processes excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and init (pid 1).
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal. Again, the set of all processes excludes certain system processes, as described earlier.
pid == 1	The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

Alarm and pause Functions

The alarm function allows us to set a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>  unsigned int alarm(unsigned int seconds);
```

Returns: 0 or number of seconds until previously set alarm

The seconds value is the number of clock seconds in the future when the signal should be generated. Be aware that when that time occurs, the signal is generated by the kernel, but there could be additional time before the process gets control to handle the signal, because of processor scheduling delays.

Earlier UNIX System implementations warned that the signal could also be sent up to 1 second early. POSIX.1 does not allow this.

There is only one of these alarm clocks per process. If, when we call alarm, a previouslyregistered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function. That previously registered alarm clock is replaced by the new value.

If a previously registered alarm clock for the process has not yet expired and if the seconds value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

Although the default action for SIGALRM is to terminate the process, most processes that use an alarm clock catch this signal. If the process then wants to terminate, it can perform whatever cleanup is required before terminating. If we intend to catch SIGALRM, we need to be careful to install its signal handler before calling alarm. If we call alarm first and are sent SIGALRM before we can install the signal handler, our process will terminate.

The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>
 int pause(void);
```

Returns: 1 with errno set to EINTR

The only time pause returns is if a signal handler is executed and that handler returns. In that case,pause returns 1 with errno set to EINTR.

Example

Using alarm and pause, we can put a process to sleep for a specified amount of time. The sleep1function in Figure 10.7 appears to do this (but it has problems, as we shall see shortly).

This function looks like the sleep function, which we describe in Section 10.19, but this simple implementation has three problems.

1. If the caller already has an alarm set, that alarm is erased by the first call to alarm. Wecan correct this by looking at the return value from the first call to alarm. If the number of seconds until some previously set alarm is less than the argument, then we should wait only until the previously set alarm expires. If the previously set alarm will go off after ours, then before returning we should reset this alarm to occur at its designated time in the future.
2. We have modified the disposition for SIGALRM. If we're writing a function for others to call, we should save the disposition when we're called and restore it when we're done. We can correct this by saving the return value from signal and resetting the disposition before we return.
3. There is a race condition between the first call to alarm and the call to pause. On a busy system, it's possible for the alarm to go off and the signal handler to be called before we call pause. If that happens, the caller is suspended forever in the call to pause (assuming that some other signal isn't caught).

Earlier implementations of sleep looked like our program, with problems 1 and 2 corrected as described. There are two ways to correct problem 3. The first uses setjmp, which we show in the next example.

```
include <signal.h>
#include <unistd.h>
static void sig_alrm(int signo) { /* nothing to do, just return to wake up the pause */ }
 unsigned int sleep1(unsigned int nsecs)
{ if (signal(SIGALRM, sig_alrm) == SIG_ERR)
return(nsecs);
alarm(nsecs); /* start the timer */ pause();
 /* next caught signal wakes us up */ return(alarm(0)); /* turn off timer, return unslept time */
}
```

Abort & sleep functions.

NAME

abort - cause abnormal process termination

SYNOPSIS

#include <stdlib.h>
void abort(void);

DESCRIPTION

The **abort**() first unblocks the **SIGABRT** signal, and then raises that signal for the calling process (as though raise(3) was called). This results in the abnormal termination of the process unless the **SIGABRT** signal is caught and the signal handler does not return . If the **abort**() function causes process termination, all open streams are closed and flushed.If the **SIGABRT** signal is ignored, or caught by a handler that returns, the **abort**() function will still terminate the process. It does this by restoring the default disposition for **SIGABRT** and then raising the signal for a second time.

RETURN VALUE

The **abort**() function never returns.

Name

sleep - sleep for the specified number of seconds

Synopsis

#include <unistd.h>
unsigned int sleep(unsigned int seconds);

Description

sleep() makes the calling thread sleep until *seconds* seconds have elapsed or a signal arrives which is not ignored.

Return Value

Zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler.

UNIT IV

Syllabus

Intercrosses Communication : Introduction to IPC, IPC between processes on a single computer system, IPC between processes on different systems pipes-creation, IPC between related processes using named pipes, FIFOs-creation, IPC between unrelated processes using FIFOs (Named pipes),differences between unnamed and named pipes, popen and pclose library functions. Message Queues- Kernel support for messages, APIs for messages, client/server example. Semaphores – Kernel support for semaphores, APIs for semaphores, file locking with semaphores.

Inter Process communication (IPC): Inter process communication (IPC) is a mechanism whereby two or more processes communicate with each other to perform tasks. These processes may interact in a client/server manner or in a peer to peer fashion.
Examples: Database servers, E-Mails, IPC is supported is supported by all UNIX systems. However, different UNIX systems implement different methods for IPC. i.e BSD UNIX-Sockets, Unix System V.5 for Pipes, FIFOs, messages, semaphores and shared memory.

In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow processes it manages toshare data. Typically, applications can use IPC categorized as clients and servers, where the client requests data and the server responds to client requests. Many applications are both clients and servers, as commonly seen in distributed computing. Methods for achieving IPC are divided into categories which vary based on software requirements, such as performance and modularity requirements, and system circumstances, such as network bandwidth and latency.
Examples: These processes are not bound to one computer, and can run on various computers connected by network. Inter-process communication techniques can be divided into various types. These are:

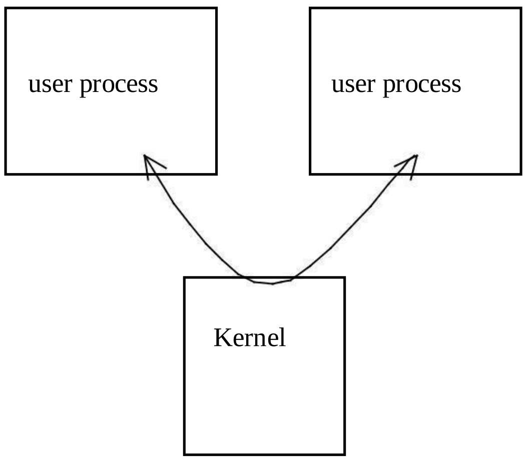
1. Pipes
2. FIFO
3. Shared memory
4. Mapped memory
5. Message queues
6. Sockets

Pipes: The most basic versions of the UNIX operating system gave birth to pipes. These were used to facilitate one-directional communication between single-system processes. We can create a pipe by using the pipe system call, thus creating a pair of file descriptors
FIFO: A FIFO or ‘first in, first out’ is a one-way flow of data. FIFOs are similar to pipes, the only difference being that FIFOs are identified in the file system with a name. In simple terms, FIFOs are ‘named pipes’.
Shared memory: Shared memory is an efficient means of passing data between programs. An area is created in memory by a process, which is accessible by another process. Therefore,processes communicate by reading and writing to that memory space.
Mapped memory: This method can be used to share memory or files between different processors in a Windows environment. A 32-bit API can be used with Windows. This mechanism speeds up file access, and also facilitates inter-process communication.

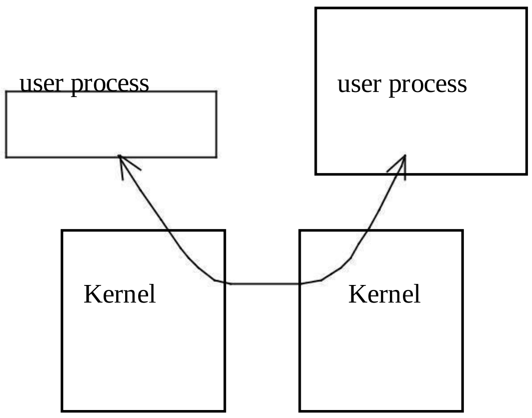
Message queues: By using this method, a developer can pass messages between messages via a single queue or a number of message queues. A system kernel manages this mechanism. An application program interface (API) coordinates the messages.
Sockets: We use this mechanism to communicate over a network, between a client and a server. This method facilitates a standard connection that is independent of the type of computer and the type of operating system used.

IPC between processes on a single computer system:

Communication is everywhere from intraprocess to Interprocess. Interprocess communication has 2 forms:



IPC on one host



IPC on different hosts:

Network Programming

IPC between processes on different systems pipes-creation:

Pipes:

A pipe is a technique for passing information from one program process to another.
A simple, unnamed pipe provides a one-way flow of data(Half Duplex)
Pipe sets up communication channel between two (related) processes.
An unnamed pipe is created by calling *pipe(),* which returns an array of 2 file descriptors (int).
An unnamed pipe does not associate with any physical file.
It can only be shared by related processes (descendants of a process that creates the unnamed pipe).

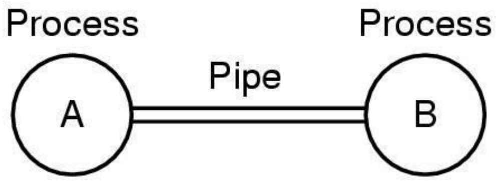


Fig: Two processes connected by a pipe

Features of Pipes:

On many systems, pipes are limited to 10 logical blocks, each block has 512 bytes.

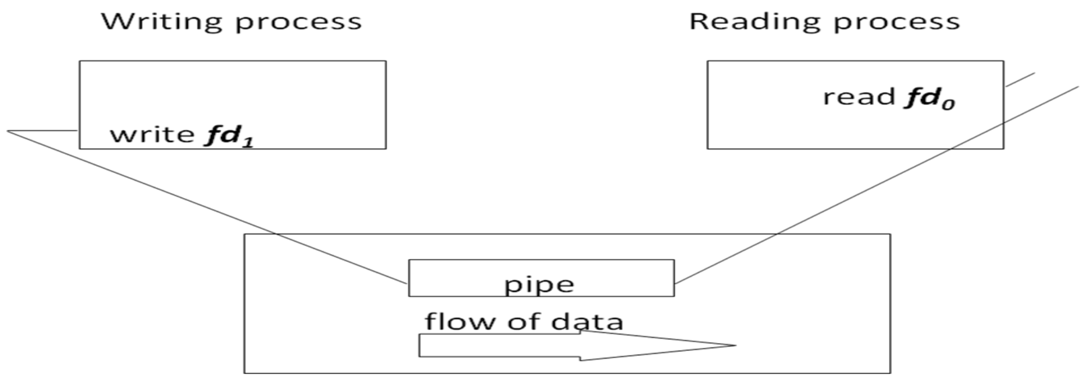
As a general rule, one process will write to the pipe (as if it were a file), while another process will read from the pipe.

Data is written to one end of the pipe and read from the other end.

A pipe exists until both file descriptors are closed in all processes

Piping between Two Processes

The pipe is represented in an array of 2 file descriptors (int)



A pipe provides a one-way flow of data. A pipe can be created by using pipe()

#include<pipe.h>

int pipe (int * *filedes*);

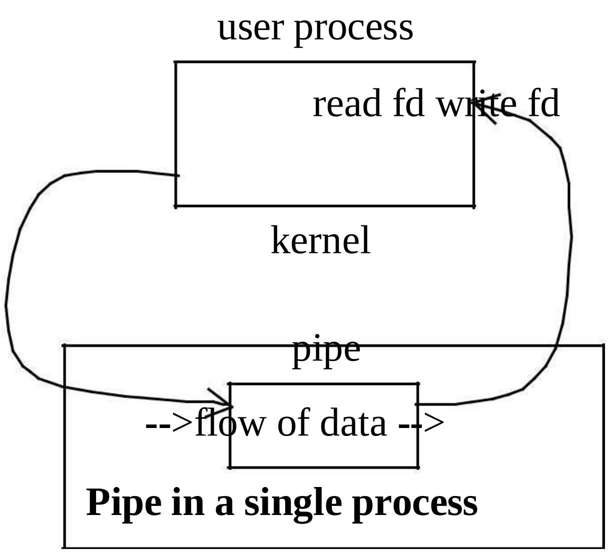
int pipefd[2]; /* pipefd[0] is opened for reading; pipefd[1] is opened for writing */

Example to show how to create and use a pipe:

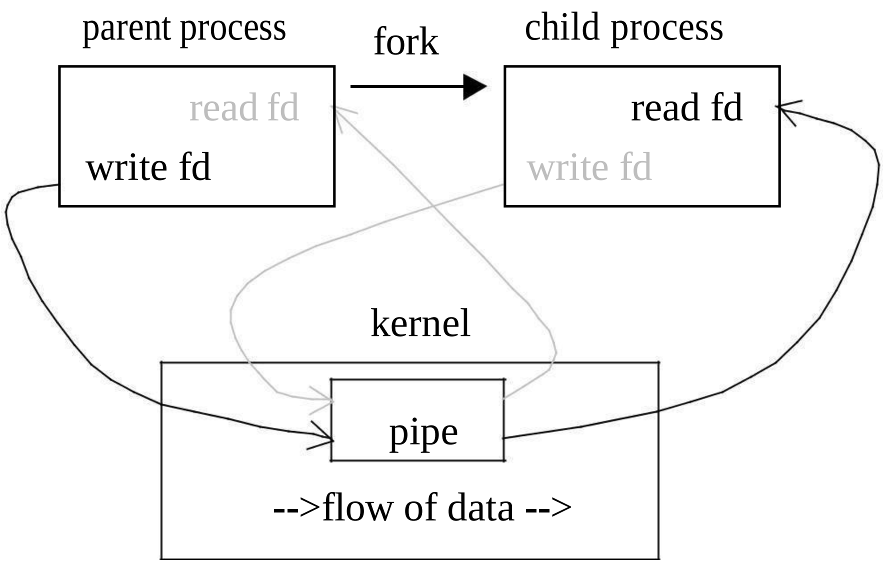
```
#include<stdio.h>
#include<pipe.h>
#include<error.h>
main()
{  int  pipefd[2],
        n;     char
        buff[100
        ];
         if (pipe(pipefd) < 0 ) err_sys("pipe error");
         printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
         if (write(pipefd[1], "hello world\n", 12) != 12) err_sys("write error");
         if ( (n=read(pipefd[0], buff, sizeof(buff))) < =0) err_sys("read
         error"); write(1, buff, n); /*fd=1=stdout*/
}
```

```
result:
  hello world
  read fd=3, write df =4
```

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

int main(void)
{
    int pfds[2];

    pipe(pfds);

    if (!fork()) {
        close(1);       /* close normal stdout */
        dup(pfds[1]);  /* make stdout same as pfds[1] */
        close(pfds[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0);       /* close normal stdin */
        dup(pfds[0]);  /* make stdin same as pfds[0] */
        close(pfds[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }

    return 0;
}
```

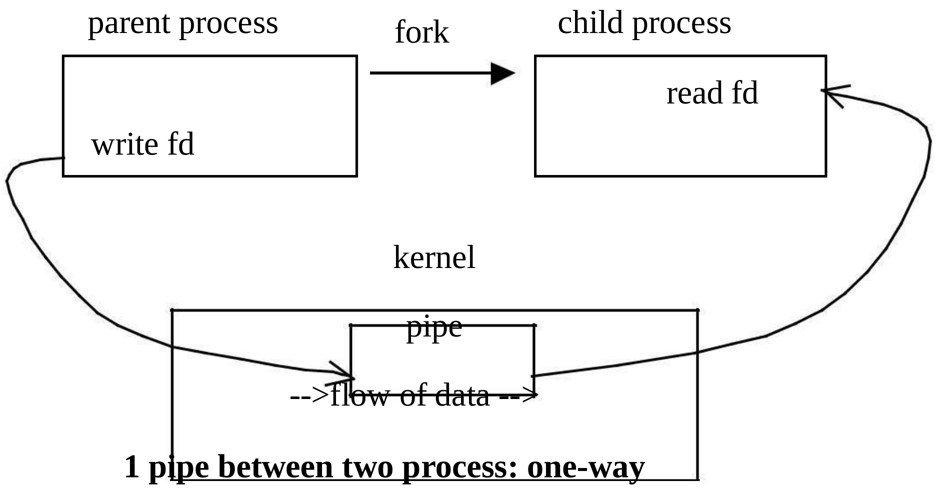


Pipe in a single process



Pipe in a single process, immediately after fork

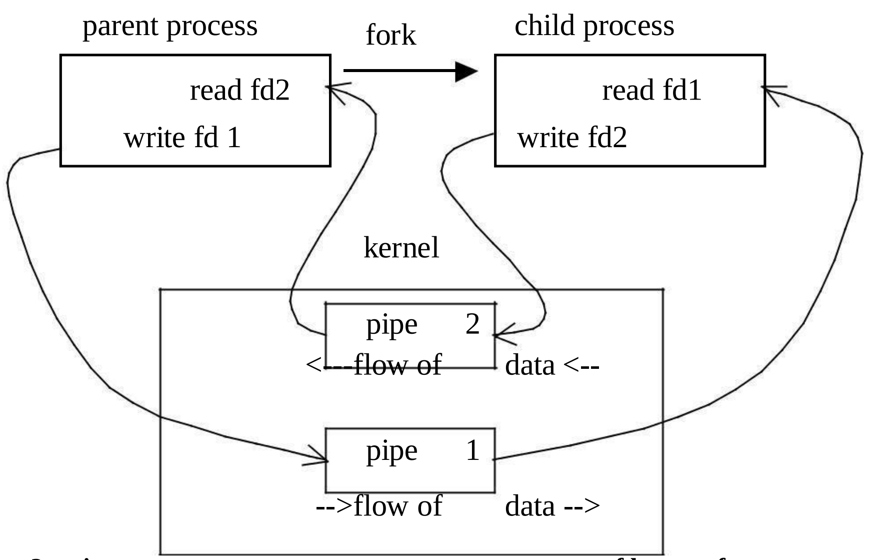
Pipes between two processes: unidirectional



1 pipe between two process: one-way

Steps : 1) opening a pipe
2) forking off another process
3) closing the oppropriate pipes on each end

Pipes between two processes: bidirectional



2 pipes to provide a bidirectional flow of data

Steps: 1) create pipe1 + pipe2 : int pipe1[2], pipe2[2] ----- must be the first step

2) forking off a child process, executing another program as a server

3) parent closes read end of pipe 1 + write end of pipe 2,

4) child closes write end of pipe 1 + read end of pipe 2

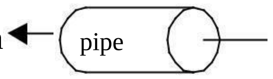
Properties of Pipe:

1) Pipes do not have a name. For this reason, the processes must share a parent process. This is the main drawback to pipes. However, pipes are treated as file descriptors, so the pipes remain open even after fork and exec.
2) Pipes do not distinguish between messages; they just read a fixed number of bytes. Newline (\n) can be used to separate messages. A structure with a length field can be used for message containing binary data.
3) Pipes can also be used to get the output of a command or to provide input to a command

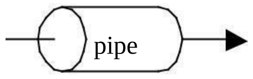
popen and pclose library functions:

```
FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

When type is “r”:

Calling process reads from 

When type is “w”:



Calling process writes to

For example:

```
#include <stdio.h> #define MAXLINE 1024 main()
{ int n;
    char line[MAXLINE]; FILE *fp;
    fp=popen(“cat .cshrc”, “r”);
    \*read the lines in .cshrc from fp*\
    while ((fgets(line, MAXLINE, fp)) != NULL) { n=strlen(line);
      if (write(1, line, n)!=n) printf(“print data error”); pclose(fp);
}
```

IPC between related processes using named pipes: FIFOs-creation:

FIFOs(Named Pipes)

A FIFO ("First In, First Out", pronounced "Fy-Foh") is sometimes known as a *named pipe*. That is, it's like apipe, except that it has a name! In this case, the name is that of a file that multiple processes can **open()**and read and write to.

This latter aspect of FIFOs is designed to let them get around one of the shortcomings of normal pipes: you can't grab one end of a normal pipe that was created by an unrelated process. See, if I run two individual copies of a program, they can both call **pipe()** all they want and still not be able to speak to one another. (This is because you must **pipe()**, then **fork()** to get a child process that can communicate to the parent via the pipe.) With FIFOs, though, each unrelated process can simply **open()** the pipe and transfer data through it.

A New FIFO is Born

Since the FIFO is actually a file on disk, you have to do some fancy-schmancy stuff to create it. It's not that hard. You just have to call **mknod()** with the proper arguments. Here isa **mknod()** call that creates a FIFO:

```
mknod("myfifo", S_IFIFO | 0644 , 0);
```

In the above example, the FIFO file will be called "*myfifo*". The second argument is the creation mode, which is used to tell **mknod()** to make a FIFO (the S_IFIFO part of the OR) and sets access permissions to that file (octal 644, or rw-r--r--) which can also be set by ORing together macros from *sys/stat.h*. This permission is just like the one you'd set using the **chmod** command. Finally, a device number is passed. This is ignored when creating a FIFO, so you can put anything you want in there.

(An aside: a FIFO can also be created from the command line using the Unix **mknod** command.)

Producers and Consumers

Once the FIFO has been created, a process can start up and open it for reading or writing using the standard **open()** system call.

Since the process is easier to understand once you get some code in your belly, I'll present here two programs which will send data through a FIFO. One is *speak.c* which sends data through the FIFO, and the other is called *tick.c*, as it sucks data out of the FIFO.

Here is *speak.c*:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
   char s[300];
   int num, fd;
```

```
    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("got a reader--type some stuff\n");

    while (gets(s), !feof(stdin)) {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("speak: wrote %d bytes\n", num);
    }

    return 0;
}
```

What **speak** does is create the FIFO, then try to **open()** it. Now, what will happen is that the **open()** call will *block* until some other process opens the other end of the pipe for reading. (There is a way around this—see O_NDELAY, below.) That process is *tick.c*, shown here:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mknod(FIFO_NAME, S_IFIFO | 0666, 0);
    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
        else {
            s[num] = '\0';
            printf("tick: read %d bytes: \"%s\"\n", num, s);
        }
    } while (num > 0);

    return 0;
}
```

Like *speak.c,* **tick** will block on the **open()** if there is no one writing to the FIFO. As soon as someone opens the FIFO for writing,**tick** will spring to life.

Try it! Start **speak** and it will block until you start **tick** in another window. (Conversely, if you start **tick**, it will block until you start**speak** in another window.) Type away in the **speak** window and **tick** will suck it all up.

Now, break out of **speak**. Notice what happens: the **read()** in **tick** returns 0, signifying EOF. In this way, the reader can tell when all writers have closed their connection to the FIFO. "What?" you ask "There can be multiple writers to the same pipe?" Sure! That can be very useful, you know. Perhaps I'll show you later in the document how this can be exploited.

But for now, lets finish this topic by seeing what happens when you break outof **tick** while **speak** is running. "Broken Pipe"! What does this mean? Well, what has happenedis that when all readers for a FIFO close and the writer is still open, the writer will receiver the signal SIGPIPE the next time it tries to **write()**. The default signal handler for this signal prints "Broken Pipe" and exits. Of course, you can handle this more gracefully by catching SIGPIPE through the **signal()** call.

Finally, what happens if you have multiple readers? Well, strange things happen. Sometimes one of the readers get everything. Sometimes it alternates between readers. Why do you want to have multiple readers, anyway?

O_NDELAY! I'm UNSTOPPABLE!

Earlier, I mentioned that you could get around the blocking **open()** call if there was no corresponding reader or writer. The way to do this is to call **open()** with the O_NDELAY flag set in the mode argument:

```
fd = open(FIFO_NAME, O_RDONLY | O_NDELAY);
```

This will cause **open()** to return -1 if there are no processes that have the file open for reading.

Likewise, you can open the reader process using the O_NDELAY flag, but this has a different effect: all attempts to **read()** from the pipe will simply return 0 bytes read if there is no data in the pipe. (That is, the **read()** will no longer block until there is some data in the pipe.) Note that you can no longer tell if **read()** is returning 0 because there is no data in the pipe, or because the writer has exited. This is the price of power, but my suggestion is to try to stick with blocking whenever possible.

Differences between unnamed and named pipes:

Pipes and FIFOs

A *pipe* is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

A *FIFO special file* is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.

A pipe or FIFO has to be open at both ends simultaneously. If you read from a pipe or FIFO file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file. Writing to a pipe or FIFO that doesn't have a reading process is treated as an error condition; it generates a SIGPIPE signal, and fails with error code EPIPE if the signal is handled or blocked.

Neither pipes nor FIFO special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

System V IPC Mechanisms

Linux supports three types of interprocess communication mechanisms that first appeared in Unix TM System V (1983). These are message queues, semaphores and shared memory. These System V IPC mechanisms all share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions, much like accessesto files are checked. The access rights to the System V IPC object is set by the creator of the object via system calls. The object's reference identifier is used by each mechanism as an index into a table of resources. It i s not a straight forward index but requires some manipulation to generate the index. All Linux data structures representing System V IPC objects in the system include an ipc_perm structure which contains the owner and creator process's user and group identifiers. The access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier. Two sets of keys are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object. System V IPC objects can never be referenced with a key, only by their reference identifier.

Message Queues

IPC:Message Queues:<sys/msg.h>

The basic idea of a ***message queue*** is a simple one.

Two (or more) processes can exchange information via access to a common system message queue. The ***sending*** process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure). Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place (subject to other permissions -- see below).

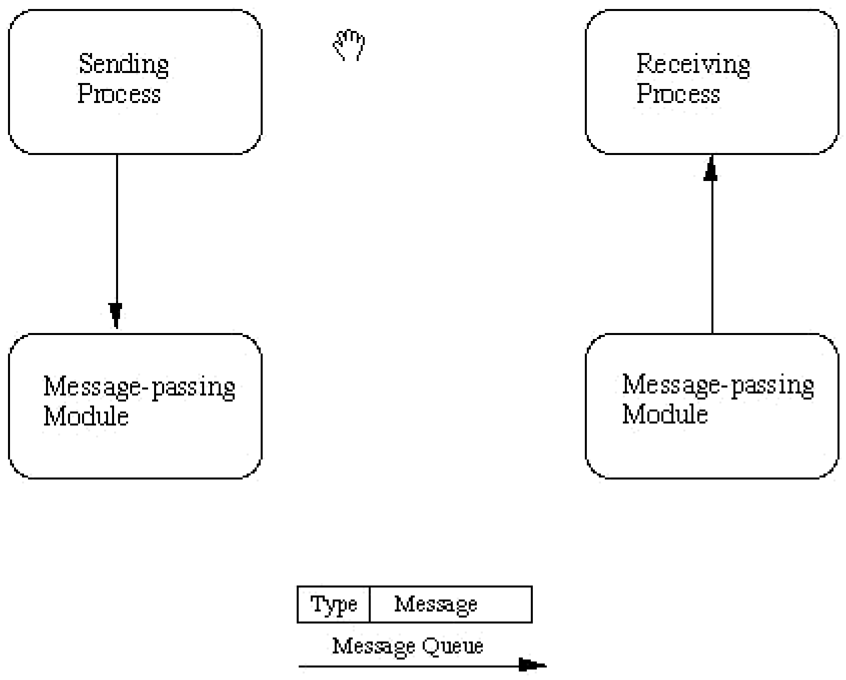


Fig. Basic Message Passing IPC messaging lets processes send and receivemessages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can beassigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the msgget function see below) Operations to send and receive messages are performed bythe msgsnd() and msgrcv() functions, respectively.

When a message is sent, its text is copied to the message queue. The msgsnd() and msgrcv() functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous messagepassing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Kernel support for messages:

Message queues allow one or more processes to write messages, which will be read by one or more reading processes. Linux maintains a list of message queues, the msgque vector; each element of which points to a msqid_ds data structure that fully describes the message queue. When message queues are created a new msqid_ds data structure is allocated from system memory and inserted into the vector.

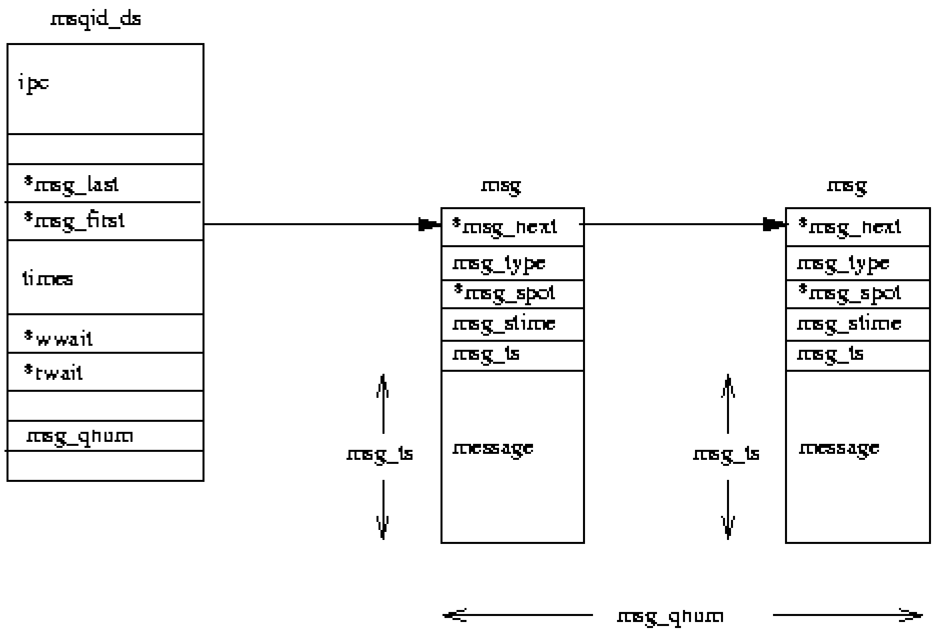


Figure: System V IPC Message Queues

Each msqid_ds data structure contains an ipc_perm data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The msqid_ds also contains two wait queues; one for the writers to the queue and one for the readers of the message queue.

Each time a process attempts to write a message to the write queue its effective user and group identifiers are compared with the mode in this queue's ipc_perm data structure. If the process can write to the queue then the message may be copied from the process's address space into a msg data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be woken up when one or more messages have been read from this message queue.

Reading from the queue is a similar process. Again, the processes access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be woken up and run again.

APIs for messages:

Initializing the Message Queue

The msgget() function initializes a new message queue:

```
int msgget(key_t key, int msgflg)
```

It can also return the message queue ID (msqid) of the queue corresponding to the key argument. The value passed as the msgflgargument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the msgget() function.

```
#include <sys/ipc.h>;
#include <sys/msg.h>;

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
...
```

IPC Functions, Key Arguments, and Creation Flags: <sys/ipc.h>

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a key_t key argument. (key_t is essentially an int type defined in <sys/types.h>

The key is an arbitrary value or one that can be derived from a common seed at run time. One way is with ftok() , which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as IPC_PRIVATE, the call initializes a new instance of an IPC facility that is private to the creating process. When the IPC_CREAT flag is suppliedin the flags argument appropriate to the call, the function tries to create the facility if it does not exist already. When called with both the IPC_CREAT and IPC_EXCL flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with IPC_EXCL in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If IPC_CREAT is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp",
key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

Controlling message queues

The msgctl() function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using msgctl() Also, any process with permission to do so can use msgctl() for control operations.

The msgctl() function is prototypes as follows:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf )
```

The msqid argument must be the ID of an existing message queue. The cmd argument is one of:

IPC_STAT

-- Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.

IPC_SET

-- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID

-- Remove the message queue specified by the msqid argument.

The following code illustrates the msgctl() function with all its various flags:

```
#include<sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
if (msgctl(msqid, IPC_SET, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

Sending and Receiving Messages

The msgsnd() and msgrcv() functions send and receive messages, respectively:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,
        int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
        int msgflg);
```

The msqid argument **must** be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
 struct mymsg {
      long      mtype;    /* message type */
      char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The msgsz argument specifies the length of the message in bytes.

The structure member msgtype is the received message's type as specified by the sending process.

The argument msgflg specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to `msg_qbytes`.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (`msgflg & IPC_NOWAIT`) is non-zero, the message will not be sent and the calling process will return immediately.
- If (`msgflg & IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
    - The condition responsible for the suspension no longer exists, in which case the message is sent.
    - The message queue identifier `msqid` is removed from the system; when this occurs, `errno` is set equal to `EIDRM` and -1 is returned.
    - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`:

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set equal to the process ID of the calling process.
- `msg_stime` is set equal to the current time.

The following code illustrates `msgsnd()` and `msgrcv()`:

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

```
...

int msgflg; /* message flags for the operation */
struct msgbuf *msgp; /* pointer to the message buffer */
int msgsz; /* message size */
long msgtyp; /* desired message type */
int msqid /* message queue ID to be used */

...

msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
- sizeof msgp->mtext + maxmsgsz));

if (msgp == NULL) {
(void) fprintf(stderr, "msgop: %s %d byte messages.\n",
"could not allocate message buffer for", maxmsgsz);
exit(1);

...

msgsz = ...
msgflg = ...

if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
perror("msgop: msgsnd failed");
...
msgsz = ...
msgtyp = first_on_queue;
msgflg = ...
if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
perror("msgop: msgrcv failed");
```

Client/server example:

```
#include<string.h>#include<time.h>
#include<sys/ipc.h>#include<sys/msg.h>
#include<sys/wait.h>#include<sys/errno.h>
extern int errno;       // error NO.
#define MSGPERM 0600    // msg queue permission
#define MSGTXTLEN 128  // msg text length
int msgqid, rc;
int done;
struct msg_buf {
  long mtype;
  char mtext[MSGTXTLEN];
} msg;
int main(int argc,char **argv)
{
  // create a message queue. If here you get a invalid msgid and use it in msgsnd() or msgrcg(), an
Invalid Argument error will be returned.
  msgqid = msgget(IPC_PRIVATE, MSGPERM|IPC_CREAT|IPC_EXCL);
  if (msgqid < 0) {    perror(strerror(errno));
    printf("failed to create message queue with msgqid = %d\n", msgqid);
    return 1;  }  printf("message queue %d created\n",msgqid);
  // message to send

  msg.mtype = 1; // set the type of message
  sprintf (msg.mtext, "%s\n", "a text msg..."); /* setting the right time format by means of ctime()
*/
  // send the message to queue
  rc = msgsnd(msgqid, &msg, sizeof(msg.mtext), 0); // the last param can be: 0, IPC_NOWAIT,
MSG_NOERROR, or IPC_NOWAIT|MSG_NOERROR.
  if (rc < 0) {    perror( strerror(errno) );
    printf("msgsnd failed, rc = %d\n", rc);
    return 1;
  }  // read the message from queue
  rc = msgrcv(msgqid, &msg, sizeof(msg.mtext), 0, 0);
  if (rc < 0) {    perror( strerror(errno) );
    printf("msgrcv failed, rc=%d\n", rc);
    return 1;  }
  printf("received msg: %s\n", msg.mtext);
  // remove the queue
  rc=msgctl(msgqid,IPC_RMID,NULL);
  if (rc < 0) {    perror( strerror(errno) );
    printf("msgctl (return queue) failed, rc=%d\n", rc);
    return 1;  }
  printf("message queue %d is gone\n",msgqid);
  return 0;
}
```

Semaphores:

In its simplest form a semaphore is a location in memory whose value can be tested and set by more than one process. The test and set operation is, so far as each process is concerned, uninterruptible or atomic; once started nothing can stop it. The result of the test and set operation is the addition of the current value of the semaphore and the set value, which can be positive or negative. Depending on the result of the test and set operation one process may have to sleep until the semphore's value is changed by another process. Semaphores can be used to implement *critical regions*, areas of critical code that only one process at a time should be executing.

Say you had many cooperating processes reading records from and writing records to a single data file. You would want that file access to be strictly coordinated. You could use a semaphore with an initial value of 1 and, around the file operating code, put two semaphore operations, the first to test and decrement the semaphore's value and the second to test and increment it. The first process to access the file would try to decrement the semaphore's value and it would succeed, the semaphore's value now being 0. This process can now go ahead and use the data file but if another process wishing to use it now tries to decrement the semaphore's value it would fail asthe result would be -1. That process will be suspended until the first process has finished with thedata file. When the first process has finished with the data file it will increment the semaphore's value, making it 1 again. Now the waiting process can be woken and this time its attempt to increment the semaphore will succeed.

Kernel support for semaphores:

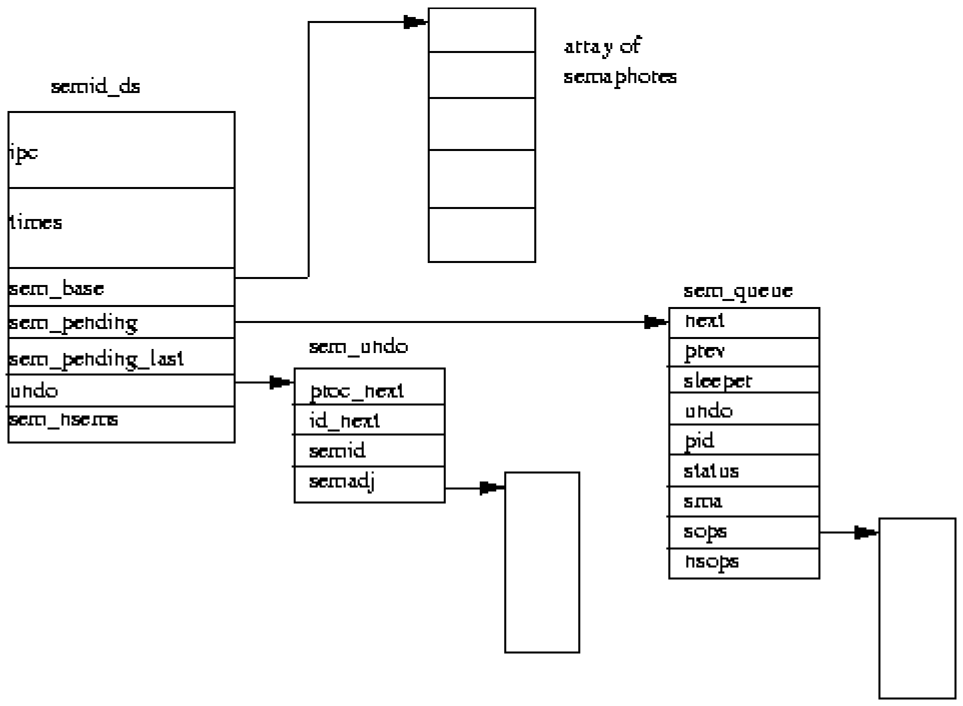


Figure: System V IPC Semaphores

System V IPC semaphore objects each describe a semaphore array and Linux uses the semid_ds data structure to represent this. All of the semid_ds data structures in the system are pointed at by the semary, a vector of pointers. There are sem_nsems in each semaphore array, each one described by a semdata structure pointed at by sem_base. All of the processes that are allowed to manipulate the semaphore array of a System V IPC semaphore object may make system calls that perform operations on them. The system call can specify many operations and each operation is described by three inputs; the semaphore index, the operation value and a set of flags. The semaphore index is an index into the semaphore array and the operation value is a numerical value that will be added to the current value of the semaphore. First Linux tests whether or not all of the operations would succeed. An operation will succeed if the operation value added to the semaphore's current value would be greater than zero or if both the operation value and the semaphore's current value are zero. If any of the semaphore operations would fail Linux may suspend the process but only if the operation flags have not requested that the system call is non-blocking. If the process is to be suspended then Linux must save the state of the semaphore operations to be performed and put the current process onto a wait queue. It does this by building a sem_queue data structure on the stack and filling it out. The new sem_queue data structure is put at the end of this semaphore object's wait queue (using the sem_pending and sem_pending_last pointers). The current process is put on the wait queue in the sem_queue data structure (sleeper) and the scheduler called to choose another process to run.
If all of the semaphore operations would have succeeded and the current process does not need to be suspended, Linux goes ahead and applies the operations to the appropriate members of the semaphore array. Now Linux must check that any waiting, suspended, processes may now apply

their semaphore operations. It looks at each member of the operations pending queue (sem_pending) in turn, testing to see if the semphore operations will succeed this time. If they will then it removes the sem_queue data structure from the operations pending list and applies the semaphore operations to the semaphore array. It wakes up the sleeping process making it available to be restarted the next time the scheduler runs. Linux keeps looking through the pending list from the start until there is a pass where no semaphore operations can be applied andso no more processes can be woken.

There is a problem with semaphores, *deadlocks*. These occur when one process has altered the semaphores value as it enters a critical region but then fails to leave the critical region because it crashed or was killed. Linux protects against this by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphores will be put back to the state that they were in before the a process's set of semaphore operations were applied. These adjustments are kept in sem_undo data structures queued both onthe semid_ds data structure and on the task_struct data structure for the processes using these semaphore arrays. Each individual semaphore operation may request that an adjustment be maintained. Linux will maintain at most one sem_undo data structure per process for each semaphore array. If the requesting process does not have one, then one is created when it is needed. The new sem_undo data structure is queued both onto this process's task_struct data structure and onto the semaphore array's semid_ds data structure. As operations are applied to the semphoresin the semaphore array the negation of the operation value is added to this semphore's entry inthe adjustment array of this process's sem_undo data structure. So, if the operation value is 2, then -2 is added to the adjustment entry for this semaphore.

When processes are deleted, as they exit Linux works through their set of sem_undo data structures applying the adjustments to the semaphore arrays. If a semaphore set is deleted,the sem_undo data structures are left queued on the process's task_struct but the semaphore arrayidentifier is made invalid. In this case the semaphore clean up code simply discards the sem_undo data structure.

APIs for semaphores:

The semget function creates a semaphore set or accesses an existing semaphore set.

```
#include <sys/sem.h>
int semget(key-t key, int nsems, int oflag);
```

Returns: nonnegative identifier if OK, -1 on error

Arguments:

Key –returns from ftok()

Nsem-number of semaphore sets are created

Oflag – Open flag for accessing is IPC_CREAT|0666

Program:

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

```
Int main()
{
Int semid;
Semid=semget(15,2,IPC_CREAT|0644);
If(semid==-1)
{
Printf("semget error");
exit(1);
}
else
{
printf("semaphore set created :");
Printf(semid=%d",semid);
exit(1);
}
Return 0;
}
```

Semop()

Once a semaphore set is opened with semget, operations are performed on one or more of the semaphores in the set using the semop function.

```
#include <sys/sem.h>
int semop (int semid, struct sembuf *opsptr, size-t nops) ;
Returns: 0 if OK, -1 on error
```

Arugments:

Semid: returns from semget()

opsptr points to an array of the following structures:

```
struct sembuf {
short sem-num; /* semaphore number: 0, 1, ..., nsems-1 */
short sem-op; /* semaphore operation: <0, 0, >O */
short sem-flg; /* operation flags: 0, IPC-NOWAIT, SEM-UNDO */
};
```

Nops: specifies how many entries are in the array pointed to by opPtr

Program

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
Strct semid_ds sbuf[2]={{0,-1,SEM_UNDO||IPC_NOWAIT},1,0,0}};
Int main()
{
```

```
Int perms=S_IRWSU|S_IRWXG|S_IRWXO;
Int fd=semget(100,2,IPC_CREAT|S_IRWXG|S_IRWXO);
If(fd==-1)
Perrror("semget");
Exit(1);
If(semop(fd,sbuf,2==-1)
Perror("semop");
Return 0;
}
```

Semctl()

The semctl function performs various control operations on a semaphore.

#include <sys/sem.h>
int semctl (int semid, int semnum, int cmd, union semun arg) ;
Returns: nonnegative value if OK (see text), -1 on error
Arguments:
Semid –returns from semget()
Semnum –is a semaphore index where the next argument cmd specifies an operation to be performed.
The possible values of **cmd** are IPC_STAT,IPC_SET,IPC_RMID,GETALL,SETALL,GETVAL,SETVAL,GETPID,GETNCNT,GETZCNT.
Union semun arg-is a union typed object that may be used to specify r retrieve the control data of one or more semaphores in the set.

```
Union semun
{
Int val; //a semaphore value
Struct semid_ds *buf; //control data of a semaphore set
Ushort *array; //an array of semaphore values
};
/* ** semrm.c -- removes a semaphore */
 #include <stdio.h>
 #include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
 int main(void)
 {
 key_t key;
 int semid; if ((key = ftok("semdemo.c", 'J')) == -1)
 {
 perror("ftok");
 exit(1);
} /* grab the semaphore set created by seminit.c: */
 if ((semid = semget(key, 1, 0)) == -1)
{ perror("semget");
exit(1);
 } /* remove it: */
if (semctl(semid, 0, IPC_RMID) == -1)
{
 perror("semctl");
 exit(1); }
 return 0;
 }
```

File locking with semaphores:

There are two of them. The first, *semdemo.c*, creates the semaphore if necessary, and performs some pretend file locking on it in a demo very much like that in the File Locking document. The second program, *semrm.c* is used to destroy the semaphore (again, **ipcrm** could be used to accomplish this.)

The idea is to run run *semdemo.c* in a few windows and see how all the processes interact. When you're done, use *semrm.c* to remove the semaphore. You could also try removing the semaphore while running *semdemo.c* just to see what kinds of errors are generated.

Here's *semdemo.c*, including a function named **initsem()** that gets around the semaphore race conditions, Stevens-style:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
#define MAX_RETRIES 10

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};

/*
** initsem() -- more-than-inspired by W. Richard Stevens' UNIX Network
** Programming 2nd edition, volume 2, lockvsem.c, page 295.
*/
int initsem(key_t key, int nsems)  /* key from ftok() */
{
    int i;
    union semun arg;
    struct semid_ds buf;
    struct sembuf sb;
    int semid;

    semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | 0666);

    if (semid >= 0) { /* we got it first */
        sb.sem_op = 1; sb.sem_flg = 0;
        arg.val = 1;

        printf("press return\n"); getchar();

        for(sb.sem_num = 0; sb.sem_num < nsems; sb.sem_num++) {
            /* do a semop() to "free" the semaphores. */
            /* this sets the sem_otime field, as needed below. */
            if (semop(semid, &sb, 1) == -1) {
                int e = errno;
                semctl(semid, 0, IPC_RMID); /* clean up */
                errno = e;
                return -1; /* error, check errno */
            }
        }

    } else if (errno == EEXIST) { /* someone else got it first */
        int ready = 0;

        semid = semget(key, nsems, 0); /* get the id */
        if (semid < 0) return semid; /* error, check errno */

        /* wait for other process to initialize the semaphore: */
        arg.buf = &buf;
        for(i = 0; i < MAX_RETRIES && !ready; i++) {
            semctl(semid, nsems-1, IPC_STAT, arg);
            if (arg.buf->sem_otime != 0) {
                ready = 1;
            } else {
                sleep(1);
            }
        }
        if (!ready) {
            errno = ETIME;
            return -1;
        }
    } else {
        return semid; /* error, check errno */
    }

    return semid;
}

int main(void)
{
    key_t key;
    int semid;
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;  /* set to allocate resource */
    sb.sem_flg = SEM_UNDO;

    if ((key = ftok("semdemo.c", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* grab the semaphore set created by seminit.c: */
    if ((semid = initsem(key, 1)) == -1) {
        perror("initsem");
        exit(1);
    }

    printf("Press return to lock: ");
    getchar();
    printf("Trying to lock...\n");

    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(1);
    }
    printf("Locked.\n");
    printf("Press return to unlock: ");
    getchar();

    sb.sem_op = 1; /* free resource */
    if (semop(semid, &sb, 1) == -1) {
        perror("semop");
        exit(1);
    }

    printf("Unlocked\n");

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(void)
{
    key_t key;
    int semid;
    union semun arg;

    if ((key = ftok("semdemo.c", 'J')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* grab the semaphore set created by seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    /* remove it: */
    if (semctl(semid, 0, IPC_RMID, arg) == -1) {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```

UNIT -V

Syllabus

Shared Memory- Kernel support for shared memory, Unix system V APIs for shared memory, shared memory example. Sockets – Introduction to Berkeley sockets, IPC over a network, Client-Server model, Socket address structures (Unix domain and Internet domain), Socketsystem calls for connection oriented protocol and connectionless protocol, example –client/server programs –Single Server –Client connection, Multiple simultaneous clients, Socket options –setsockopt and fcntl system calls, Comparison of IPC mechanisms

Shared Memory

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The pages of the virtual memory is referenced by page table entries in each of the sharing processes' page tables. It does not have to be at the same address in all of the processes' virtual memory. As with all System V IPC objects, access to shared memory areas is controlled via keys and access rights checking. Once the memory is being shared, there are no checks on how the processes are using it. They must rely on other mechanisms, for example System V semaphores, to synchronize access to the memory.

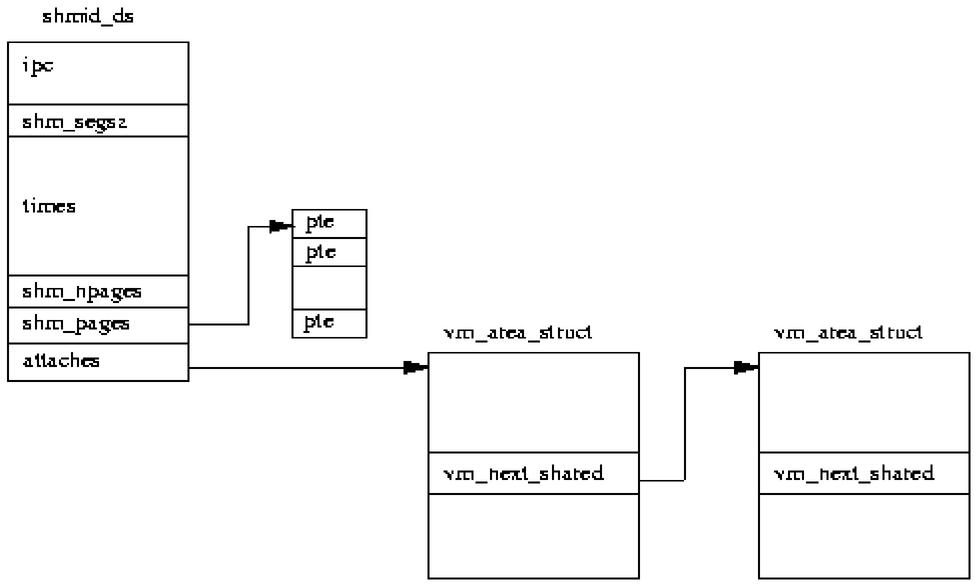


Figure: System V IPC Shared Memory

Each newly created shared memory area is represented by a shmid_ds data structure. These are kept in the shm_segs vector.

The shmid_ds data structure decribes how big the area of shared memory is, how many processes are using it and information about how that shared memory is mapped into their address spaces. It is the creator of the shared memory that controls the access permissions to that memory and whether its key is public or private. If it has enough access rights it may also lock the shared memory into physical memory.

Each process that wishes to share the memory must attach to that virtual memory via a system call. This creates a new vm_area_struct data structure describing the shared memory for this process. The process can choose where in its virtual address space the shared memory goes or it can let Linux choose a free area large enough. The new vm_area_struct structure is put into the list of vm_area_struct pointed at by the shmid_ds. The vm_next_shared and vm_prev_shared pointers are used to link them together. The virtual memory is not actually created during the attach; it happens when the first process attempts to access it.
The first time that a process accesses one of the pages of the shared virtual memory, a page fault will occur. When Linux fixes up that page fault it finds the vm_area_struct data structure describing it. This contains pointers to handler routines for this type of shared virtual memory. The shared memory page fault handling code looks in the list of page table entries for this shmid_ds to see if one exists for this page of the shared virtual memory. If it does not exist,it will allocate a physical page and create a page table entry for it. As well as going into the current process's page tables, this entry is saved in the shmid_ds. This means that when the next process that attempts to access this memory gets a page fault, the shared memory fault handling code will use this newly created physical page for that process too. So, the first process that accesses a page of the shared memory causes it to be created and thereafter access by the other processes cause that page to be added into their virtual address spaces.

When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory the detach only affects the current process.Its vm_area_struct is removed from the shmid_ds data structure and deallocated. The current process's page tables are updated to invalidate the area of virtual memory that it used to share. When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed, as is the shmid_ds data structure for this shared memory.
Further complications arise when shared virtual memory is not locked into physical memory. In this case the pages of the shared memory may be swapped out to the system's swap disk during periods of high memory usage.

Shared Memory Data Structure

```
/* One shmid data structure for each shared memory segment in the system. */
struct shmid_ds {
  struct ipc_perm shm_perm;  /* operation perms */
  int shm_segsz;             /* size of segment (bytes) */
  time_t shm_atime;          /* last attach time */
  time_t shm_dtime;          /* last detach time */
  time_t shm_ctime;          /* last change time */
  unsigned short shm_cpid;   /* pid of creator */
  unsigned short shm_lpid;   /* pid of last operator */
  short shm_nattch;          /* no. of current attaches */
                  /* the following are private */
  unsigned short shm_npages;          /* size of segment (pages) */
  unsigned long *shm_pages; /* array of ptrs to frames -> SHMMAX */
  struct vm_area_struct *attaches; /* descriptors for attaches */
};
```

Creating Shared Memory

int shmget(key_t key, size_t size, int shmflg);
key is either a number or the constant IPC_PRIVATE (man ftok)
a shmid is returned
key_t ftok(const char * path, int id) will return a key value for IPC usage
size is the size of the shared memory data
shmflg is a rights mask (0666) OR'd with one of the following:

IPC_CREAT	will create or attach
IPC_EXCL	creates new or it will error if it exists

Attaching to Shared Memory

- After obtaining a shmid from shmget(), you need to *attach* or map the shared memory segment to your data reference:

void * shmat(int shmid, void * shmaddr, int shmflg)

- shmid is the id returned from shmget()
- shmaddr is the shared memory segment address. Set this to NULL and let the system handle it.
- shmflg is one of the following (usually 0):
  - SHM_RDONLY sets the segment readonly
  - SHM_RND sets page boundary access
  - SHM_SHARE_MMUset first available aligned address

Shared Memory Control

```
struct shmid_ds {
int shm_segsz;                  /* size of segment in bytes */
__time_t shm_atime;             /* time of last shmat command */
__time_t shm_dtime;             /* time of last shmdt command */
...
unsigned short int __shm_npages;        /* size of segment in pages */
msgqnum_t shm_nattach;                  /* number of current attaches */
...                       /* pids of creator and last shmop */
};
```

int shmctl(int shmid, int cmd, struct shmid_ds * buf);
cmd can be one of:

IPC_RMID destroy the memory specified by shmid
IPC_SET set the uid, gid, and mode of the shared mem

IPC_STAT	get the current shmid_ds struct for the queue
SHM_LOCK	Lock the shared memory in memory.
SHM_UNLOCK	Unlock the shared memory in memory.

Shmat() &shmdt()

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include<unistd.h>
#include<stdio.h>
int main()
{
Int perms=S_IRWXU|S_IRWXG|S_IRWXO;
int fd=shmget(100,1024,IPC_CREAT|0644);
If(fd==-1)
Perror(“shmget”);
Exit(1);
Char* addr=(char*)shmat(fd,0,0);
If(addr==(charr*)-1
Perror(“shmat);
Exit(1);
Strcpy(addr,”Hello”);
If(shmdt(addr)==-1)
Perror(“shmdt”);
Return 0;
}
```

What is socket?

A socket acts as an end point in connection between client and a server present in a network.
Sockets can run on either TCP or UDP protocols
Socket is an interface between application and network.
-The application creates a socket
-The socket *type* dictates the style of communication
End point determined by two things:

– Host address: IP address is *Network Layer*

  - Port number: is *Transport Layer*

- Two end-points determine a connection: socket pair

  - ex: 206.62.226.35,p21 + 198.69.10.2,p1500

ex: 206.62.226.35,p21 + 198.69.10.2,p1499

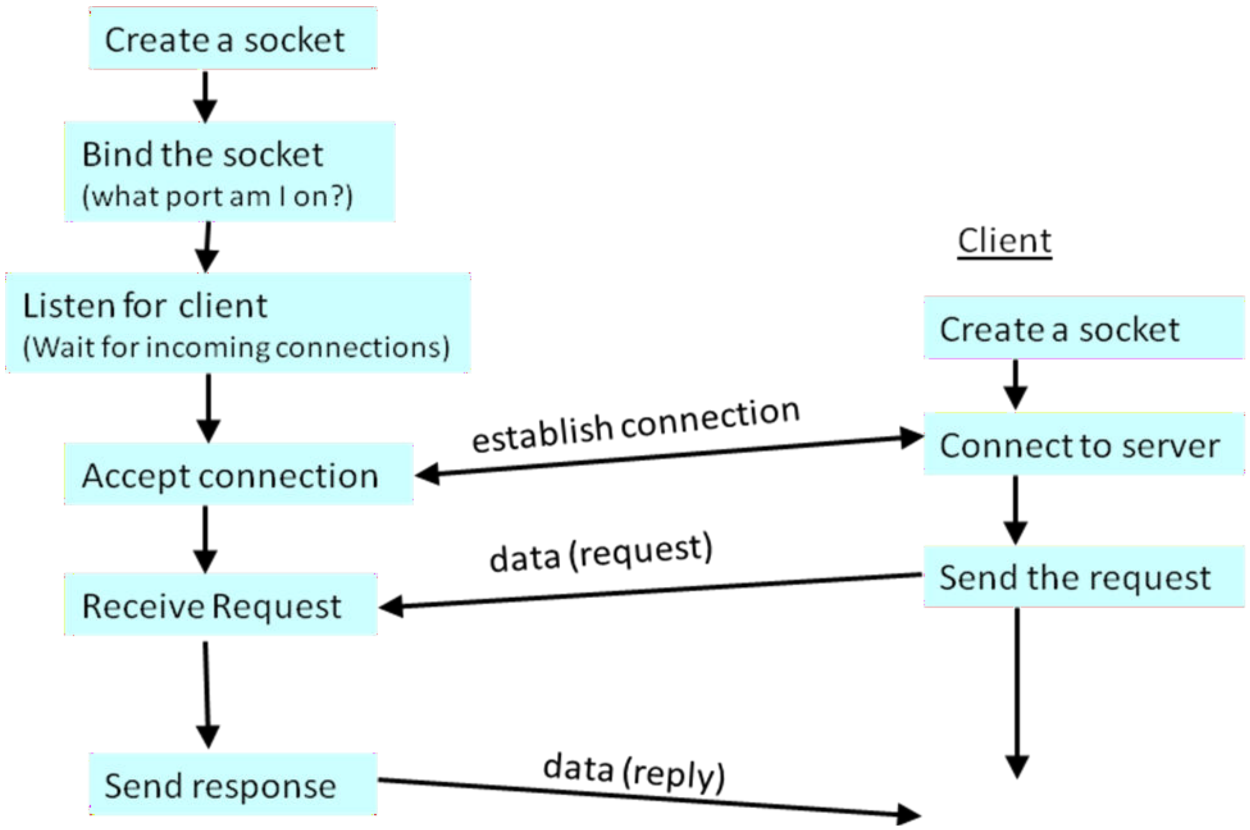
Datagram Socket (UDP)

  - Collection of messages
  - Best effort
  - Connectionless: sender or receiver address must be passed along with each message sent from one process to another
  - Unreliable
  - High speed

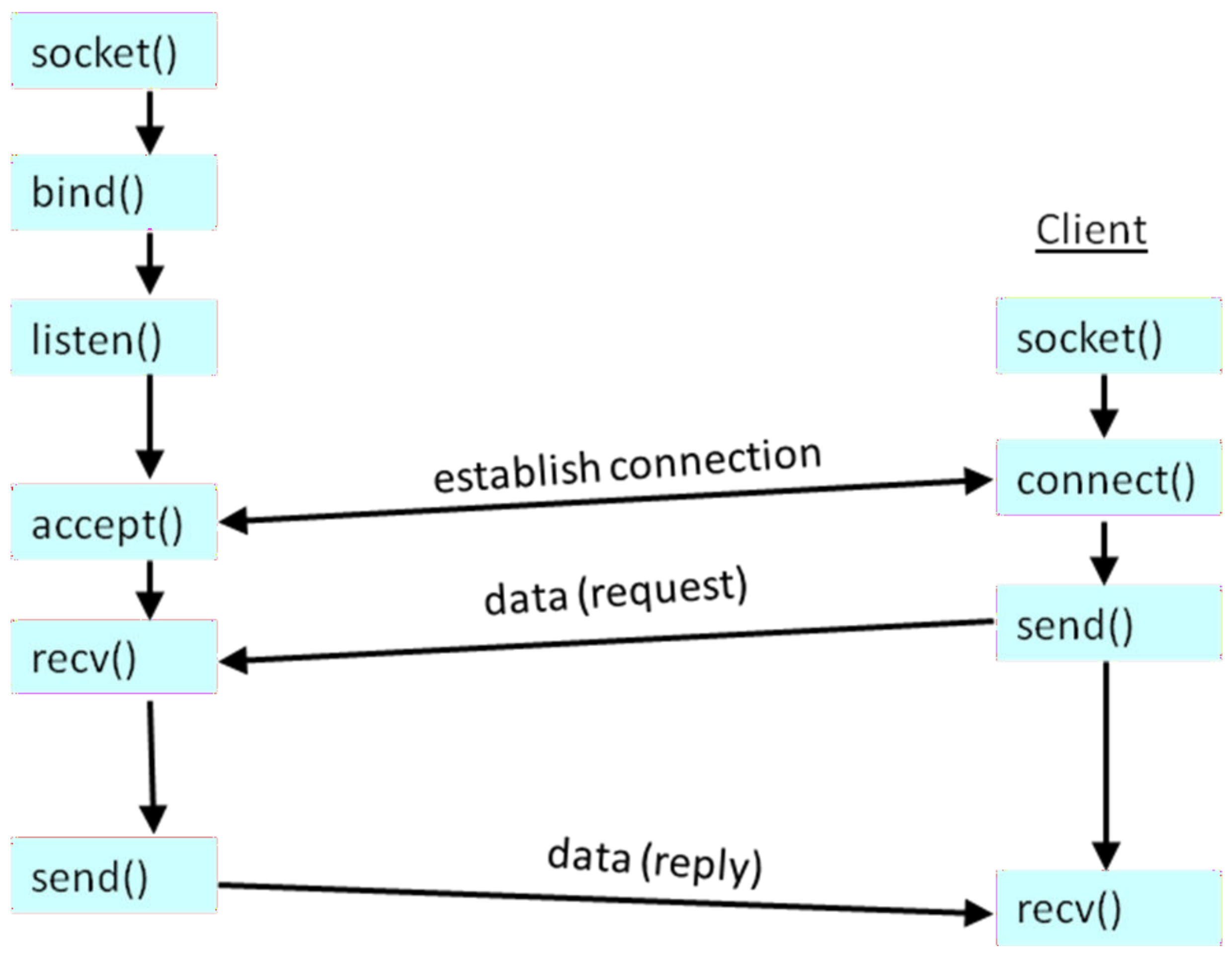
Stream Socket (TCP)

  - Stream of bytes
  - Reliable
  - Connection-oriented: sender and receiver socket addresses are pre established before messages are passed between them.
  - Low efficiency

Client-Server Communication Stream Sockets (TCP): Connection-oriented



Connection-oriented Example (Stream Sockets -TCP)



Socket Address Structure

1. A socket address structure is a special structure that stores the connection details of a socket.
2. It mainly consists of fields like IP address,port number and protocol family.
3. Different protocol suites use different socket address structures.
4. The different socket address structures are
5. IPv4 socket address structure: struct sockaddr_in
6. IPv46 socket address structure: struct sockaddr_in6
7. Generic socket address structure: struct sockaddr

socket(): creating a socket

socket - create an endpoint for communication

The *socket()* function creates an unbound socket in a communications domain, and returns a file descriptor that can be used in later function calls that operate on sockets.
Syntax:
#include <sys/socket.h>
int socket(int *domain*, int *type*, int *protocol*);
The *domain* argument specifies the address family used in the communications domain. The address families supported by the system are implementation-dependent.
The *<sys/socket.h>* header defines at least the following values for the *domain* argument:

AF_UNIX	File system pathnames.
AF_INET	internet address.

The *type* argument specifies the socket type, which determines the semantics ofcommunication over the socket. The socket types supported by the system are implementation-dependent.
Possible socket types include:
SOCK_STREAM :Establishes a virtual circuit for communication. Messages are sent in a sequenced,reliable,
SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.
SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode transmission path for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.
*Protocol :*Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes *socket()* to use an unspecified default protocol appropriate for the requested socket type.
If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address family. The protocols supported by the system are implementation-dependent.
RETURN VALUE

Upon successful completion, *socket()* returns a nonnegative integer, the socket file descriptor. Otherwise a value of -1 is returned and *errno* is set to indicate the error.

bind()- binds a name to a socket

The *bind()* function assigns an *address* to an unnamed socket. Sockets created with *socket()* function are initially unnamed; they are identified only by their address family.
Syntax:
#include<sys/types.h>
#include <sys/socket.h>
int bind(int *socketid*, const struct sockaddr **address*, socklen_t *address_len*);
The function takes the following arguments:
*socketid :*Specifies the socket descriptor of the socket to be bound.
address: Points to a sockaddr structure containing the address to be assigned to the socket. The length and format of the address depend on the address family of the socket.
address_len Specifies the length of the sockaddr structure pointed to by the *address* argument.

Upon successful completion, *bind()* returns 0. Otherwise, -1 is returned and *errno* is set to indicate the error

listen() listen for socket connections and limit the queue of incoming connections .

Syntax:
#include<sys/types.h>
#include <sys/socket.h>
int listen(int *socketid*, int *backlog*);

The *listen()* function marks a connection-mode socket, specified by the *socket* argument, as accepting connections, and limits the number of outstanding connections in the socket's listen queue to the value specified by the *backlog* argument.
The socketid argument is a socket descriptor , as returned by a socket sunction call.
The backlog argument specifies the maximum number of connection requests may be queued for the socket.
In most UNIX systems , the maximum allowed value for yhe backlog argument is 5
Upon successful completions, *listen()* returns 0. Otherwise, -1 is returned and *errno* is set to indicate the error.

accept() - accept a new connection on a socket

A server accepts a connection request from a client socket.
This is called in a server process to establish a connection based socket connection with a client socket(which calls *connect* to request connection establishment
Syntax
#include<sys/types.h>
#include <sys/socket.h>
int accept (int *socketid*, struct sockaddr **address*, socklen_t **address_len*);

- The *accept()* function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.
- The socketid argument is a socket descriptor , as returned by a socket function call.
- The address argument is a pointer to the address of a socketaddr typed object that holds the name of a client socket where the server socket is connected.
- The address_len argument is initially set to the maximum size of the object pointed to by the address argument.

Connect()-connect a socket

- The *connect()* function requests a connection to be made on a socket.
- A client socket sends a connection request to a server socket.
- Syntax:

#include<sys/types.h>
#include <sys/socket.h>
int connect(int *socketid*, const struct sockaddr **address*, socklen_t *address_len*);

- *socketid* Specifies the file descriptor associated with the socket.
- *address* Points to a sockaddr structure that holds the name of the server socket to be connected.

- *address_len* Specifies the length of the sockaddr structure pointed to by the *address* argument.

send(), sendto() - send a message on a socket

The *send()* function initiates transmission of a message from the specified socket to its peer. The *send()* function sends a message only when the socket is connected.

Syntax:

```
#include<sys/types.h>
#include <sys/socket.h>
int  send(int socketid, const void *buffer, size_t length, int flags);
```

This function sends a message, contained in buffer of size length bytes, to a socket that is connected to the socket, as designated by socketid

sendto() - send a message on a socket

The *sendto()* function sends a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message will be sent to the address specified by *dest_addr*. If the socket is connection-mode, *dest_addr* is ignored.

Syntax

```
#include<sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int socketid, const void *buffer, size_t length, int flags, const struct sockaddr
*dest_addr, socklen_t dest_len);
```

This function is same as the send API , except that the calling process also specifies the address of the reciepent socket name via dest_addr and dest_len.

recv() - receive a message from a connected socket

Syntax:

```
#include<sys/types.h>
#include <sys/socket.h>
ssize_t recv(int socketid, void *buffer, size_t length, int flags);
```

The *recv()* function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

The recv() function takes the following arguments:

socketid Specifies the socket file descriptor.

buffer Points to a buffer where the message should be stored.

length Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

flags Specifies the type of message reception.

Values of flags argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK Peeks at an incoming message. The data is treated as unread and the next *recv()* or similar function will still return this data. MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket

recvfrom() - receive a message from a socket

Syntax
#include<sys/types.h>
#include <sys/socket.h>
ssize_t recvfrom(int *socketid*, void **buffer*, size_t *length*, int *flags*, struct sockaddr **address*, socklen_t **address_len*);

The *recvfrom()* function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.
This function is same as the recv API , except that the calling process also specifies the address of the sender socket name via address and address_len.
The function takes the following arguments:
Socketid Specifies the socket file descriptor.
Buffer Points to the buffer where the message should be stored.
length Specifies the length in bytes of the buffer pointed to by the *buffer* argument.
flags Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:
MSG_PEEK: Peeks at an incoming message. The data is treated as unread and the next *recvfrom()* or similar function will still return this data.
MSG_OOB :Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
MSG_WAITALL: Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socke

Shutdown ()

- This function closes the connection between a server and client socket.

Syntax:
#include<sys/types.h>
#include <sys/socket.h>
int shutdown(int sid,int mode);
The sid argument is a socket descriptor, as returned from a socket function call. This is the socket where the shutdown should occur.
The mode argument specifies the type of shutdown desired . Its possible values and meanings are:

Mode	Meaning
0	Closes the socket for reading. All further reading will return zero bytes(EOF)
1	Closes the socket for writing. Further attempts to send data to the socket will retun a -1 failure code.
2	Closes the socket for reading and writing. Further attempts to send data the data to the socket will return a -1 failure code, and any attempt to read data from the socket will receive a zero value(EOF)

Client/server programs: Connection oriented sockets:

In the following example of the server program, the number of incoming connections that the server allows depends on the first parameter that is passed to the server. The default is for the server to allow only one connection.

```
/**** iserver.c ****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_PORT 12345

/* Run with a number of incoming connection as argument */
int main(int argc, char *argv[])
{
int i, len, num, rc;
int listen_sd, accept_sd;
/* Buffer for data */
char buffer[100];
struct sockaddr_in addr;

/* If an argument was specified, use it to */
/* control the number of incoming connections */
if(argc >= 2)
num = atoi(argv[1]);
/* Prompt some message */
else
{
printf("Usage: %s <The_number_of_client_connection else 1 will be used>\n", argv[0]);
num = 1;
}

/* Create an AF_INET stream socket to receive */
/* incoming connections on */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
if(listen_sd < 0)
{
perror("Iserver - socket() error");
exit(-1);
}
else
printf("Iserver - socket() is OK\n");

printf("Binding the socket...\n");
/* Bind the socket */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(SERVER_PORT);
rc = bind(listen_sd, (struct sockaddr *)&addr, sizeof(addr));
if(rc < 0)
{
perror("Iserver - bind() error");
close(listen_sd);
exit(-1);
}
else
printf("Iserver - bind() is OK\n");

/* Set the listen backlog */
rc = listen(listen_sd, 5);
if(rc < 0)
{
perror("Iserver - listen() error");
close(listen_sd);
exit(-1);
}
else
printf("Iserver - listen() is OK\n");

/* Inform the user that the server is ready */
printf("The Iserver is ready!\n");
/* Go through the loop once for each connection */
for(i=0; i < num; i++)
{
/* Wait for an incoming connection */
printf("Iteration: #%d\n", i+1);
printf(" waiting on accept()\n");
accept_sd = accept(listen_sd, NULL, NULL);
if(accept_sd < 0)
{
perror("Iserver - accept() error");
close(listen_sd);
exit(-1);
}
else
printf("accept() is OK and completed successfully!\n");

/* Receive a message from the client */
printf("I am waiting client(s) to send message(s) to me...\n");
rc = recv(accept_sd, buffer, sizeof(buffer), 0);
if(rc <= 0)
{
perror("Iserver - recv() error");
close(listen_sd);
close(accept_sd);
exit(-1);
}
else
printf("The message from client: \"%s\"\n", buffer);
/* Echo the data back to the client */
printf("Echoing it back to client...\n");
len = rc;
rc = send(accept_sd, buffer, len, 0);
if(rc <= 0)
{
perror("Iserver - send() error");
close(listen_sd);
close(accept_sd);
exit(-1);
}
else
printf("Iserver - send() is OK.\n");
/* Close the incoming connection */
close(accept_sd);
}
/* Close the listen socket */
close(listen_sd);
return 0;
}
```

- Compile and link.

```
[bodo@bakawali testsocket]$ gcc -g iserver.c -o iserver
```

- Run the server program.

```
[bodo@bakawali testsocket]$ ./iserver
Usage: ./iserver <The_number_of_client_connection else 1 will be used>
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
```

- The server is waiting the connections from clients. The following program example is a client program.

Example: Connection-oriented common client

- This example provides the code for the client job. The client job does a socket(), connect(), send(), recv(), and close().

- The client job is not aware that the data buffer it sent and received is going to a worker job rather than to the server.
- This client job program can also be used to work with other previous connection-oriented server program examples.

```
/****** comclient.c ******/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
/* Our server port as in the previous program */
#define SERVER_PORT 12345

main (int argc, char *argv[])
{
int len, rc;
int sockfd;
char send_buf[100];
char recv_buf[100];
struct sockaddr_in addr;

if(argc !=2)
{
printf("Usage: %s <Server_name or Server_IP_address>\n", argv[0]);
exit (-1);
}
/* Create an AF_INET stream socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd < 0)
{
perror("client - socket() error");
exit(-1);
}
else
printf("client - socket() is OK.\n");
/* Initialize the socket address structure */
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(SERVER_PORT);
/* Connect to the server */
rc = connect(sockfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
if(rc < 0)
{
perror("client - connect() error");
close(sockfd);
exit(-1);
}
else
{
printf("client - connect() is OK.\n");
printf("connect() completed successfully.\n");
printf("Connection with %s using port %d established!\n", argv[1], SERVER_PORT);
}

/* Enter data buffer that is to be sent */
printf("Enter message to be sent to server:\n");
gets(send_buf);
/* Send data buffer to the worker job */
len = send(sockfd, send_buf, strlen(send_buf) + 1, 0);
if(len != strlen(send_buf) + 1)
{
perror("client - send() error");
close(sockfd);
exit(-1);
}
else
printf("client - send() is OK.\n");
printf("%d bytes sent.\n", len);
/* Receive data buffer from the worker job */
len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
if(len != strlen(send_buf) + 1)
{
perror("client - recv() error");
close(sockfd);
exit(-1);
}
else
{
printf("client - recv() is OK.\n");
printf("The sent message: \"%s\" successfully received by server and echoed back to client!\n",
recv_buf);
printf("%d bytes received.\n", len);
}
/* Close the socket */
close(sockfd);
return 0;
}
```

- Compile and link

```
[bodo@bakawali testsocket]$ gcc -g comclient.c -o comclient
```

```
/tmp/ccG1hQSw.o(.text+0x171): In function `main':
/home/bodo/testsocket/comclient.c:53: warning: the `gets' function is dangerous and should not be used.
```

- You may want to change the gets() to the secure version, gets_s(). Run the program and make sure you run the server program as in the previous program example.

```
[bodo@bakawali testsocket]$ ./comclient
Usage: ./comclient <Server_name or Server_IP_address>
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
This is a test message from a stupid client lol!
client - send() is OK.
49 bytes sent.
client - recv() is OK.
The sent message: "This is a test message from a stupid client lol!" successfully received by server and echoed back to client!
49 bytes received.
[bodo@bakawali testsocket]$
```

- And the message at the server console.

```
[bodo@bakawali testsocket]$ ./iserver
Usage: ./iserver <The_number_of_client_connection else 1 will be used>
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "This is a test message from a stupid client lol!"
Echoing it back to client...
Iserver - send() is OK.
[bodo@bakawali testsocket]$
```

- Let try more than 1 connection. Firstly, run the server.

```
[bodo@bakawali testsocket]$ ./iserver 2
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
```

```
Iteration: #1
 waiting on accept()
```

- Then run the client twice.

```
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
Test message #1
client - send() is OK.
16 bytes sent.
client - recv() is OK.
The sent message: "Test message #1" successfully received by server and echoed back to client!
16 bytes received.
[bodo@bakawali testsocket]$ ./comclient bakawali
client - socket() is OK.
client - connect() is OK.
connect() completed successfully.
Connection with bakawali using port 12345 established!
Enter message to be sent to server:
Test message #2
client - send() is OK.
16 bytes sent.
client - recv() is OK.
The sent message: "Test message #2" successfully received by server and echoed back to client!
16 bytes received.
[bodo@bakawali testsocket]$
```

- The message on the server console.

```
[bodo@bakawali testsocket]$ ./iserver 2
Iserver - socket() is OK
Binding the socket...
Iserver - bind() is OK
Iserver - listen() is OK
The Iserver is ready!
Iteration: #1
 waiting on accept()
accept() is OK and completed successfully!
I am waiting client(s) to send message(s) to me...
The message from client: "Test message #1"
Echoing it back to client...
Iserver - send() is OK.
Iteration: #2
 waiting on accept()
accept() is OK and completed successfully!
```

```
I am waiting client(s) to send message(s) to me...
The message from client: "Test message #2"
Echoing it back to client...
Iserver - send() is OK.
[bodo@bakawali testsocket]$
```

Client/server programs: Connectionless sockets:

```
#include <stdio.h>
 #include <errno.h>
 #include <netinet/in.h>
 #define DATA_BUFFER 5000
 int main () {
     struct sockaddr_in saddr, new_addr;
     int fd, ret_val;
     char buf[DATA_BUFFER];
     socklen_t addrlen;
     /* Step1: open a UDP socket */
     fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
     if (fd == -1) {
         fprintf(stderr, "socket failed [%s]\n", strerror(errno));
         return -1;
     }
     printf("Created a socket with fd: %d\n", fd);
     /* Initialize the socket address structure */
     saddr.sin_family = AF_INET;
     saddr.sin_port = htons(7000);
     saddr.sin_addr.s_addr = INADDR_ANY;
     /* Step2: bind the socket */
     ret_val = bind(fd, (struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
     if (ret_val != 0) {
         fprintf(stderr, "bind failed [%s]\n", strerror(errno));
         close(fd);
         return -1;
     }
     /* Step3: Start receiving data. */
     printf("Let us wait for a remote client to send some data\n");
     ret_val = recvfrom(fd, buf, DATA_BUFFER, 0,
                 (struct sockaddr *)&new_addr, &addrlen);
     if (ret_val != -1) {
         printf("Received data (len %d bytes): %s\n", ret_val, buf);
```

```
    } else {
        printf("recvfrom() failed [%s]\n", strerror(errno));
    }
    /* Last step: close the socket */
    close(fd);
    return 0;
}
```

```
#include <stdio.h>
 #include <errno.h>
 #include <string.h>
 #include <netinet/in.h>
 #include <netdb.h>
 #define DATA_BUFFER "Mona Lisa was painted by Leonardo da Vinci"
 int main () {
     struct sockaddr_in saddr;
     int fd, ret_val;
     struct hostent *host; /* need netdb.h for this   */
    /* Step1: open a UDP socket */
     fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
     if (fd == -1) {
         fprintf(stderr, "socket failed [%s]\n", strerror(errno));
         return -1;
     }
     printf("Created a socket with fd: %d\n", fd);
     /* Next, initialize the server address */
     saddr.sin_family = AF_INET;
     saddr.sin_port = htons(7000);
     host = gethostbyname("127.0.0.1");
     saddr.sin_addr = *((struct in_addr *)host->h_addr);
     /* Step2: send some data */
     ret_val = sendto(fd,DATA_BUFFER, strlen(DATA_BUFFER) + 1, 0,
         (struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
     if (ret_val != -1) {
         printf("Successfully sent data (len %d bytes): %s\n", ret_val, DATA_BUFFER);
     } else {
         printf("sendto() failed [%s]\n", strerror(errno));
     }
     /* Last step: close the socket */
    close(fd);
    return 0;
 }
```

```
gcc udp-server.c -o udp_server

 $

 $ ./udp_server

 Created a socket with fd: 3

 Let us wait for a remote client to send some data

 Received data (len 43 bytes): Mona Lisa was painted by Leonardo da Vinci

$ gcc udp-client.c -o udp_client

 $.

 $ ./udp_client

 Created a socket with fd: 3

 Successfully sent data (len 43 bytes): Mona Lisa was painted by Leonardo da
Vinci
```