ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
Kanyakumari Main Road, near Anjugramam, Palkulam, Anjugramam, Tamil Nadu 629401

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ADD ON COURSE
LINUX PROGRAMMING

COURSE MATERIAL

UNIT-I

Introduction to Linux:

Linux is a Unix-like computer operating system assembled under the model of free and open source software development and distribution. The defining component of Linux is the Linux kernel, an operating system kernel first released 5 October 1991 by Linus Torvalds.

Linux was originally developed as a free operating system for Intel x86-based personal computers. It has since been ported to more computer hardware platforms than any other operating system. It is a leading operating system on servers and other big iron systems such as mainframe computers and supercomputers more than 90% of today's 500 fastest supercomputers run some variant of Linux, including the 10 fastest. Linux also runs on embedded systems (devices where the operating system is typically built into the firmware and highly tailored to the system) such as mobile phones, tablet computers, network routers, televisions and video game consoles; the Android system in wide use on mobile devices is built on the Linux kernel.

Basic Features

Following are some of the important features of Linux Operating System.

- **Portable** - Portability means software’s can works on different types of hardware’s in same way. Linux kernel and application programs support their installation on any kind of hardware platform.
- **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.

Linux Advantages

1.**Low cost:** You don't need to spend time and money to obtain licenses since Linux and much of its software come with the GNU General Public License. You can start to work immediately without worrying that your software may stop working anytime because the free trial version expires. Additionally, there are large repositories from which you can freely download high quality software for almost any task you can think of.

2.**Stability:** Linux doesn't need to be rebooted periodically to maintain performance levels. It doesn't freeze up or slow down over time due to memory leaks and such. Continuous up- times of hundreds of days (up to a year or more) are not uncommon.

3.**Performance:** Linux provides persistent high performance on workstations and on networks. It can handle unusually large numbers of users simultaneously, and can make old computers sufficiently responsive to be useful again.

4.**Network friendliness:** Linux was developed by a group of programmers over the Internet and has therefore strong support for network functionality; client and server systems can be easily set up on any computer running Linux. It can perform tasks such as network backups faster and more reliably than alternative systems.

5.**Flexibility:** Linux can be used for high performance server applications, desktop applications, and embedded systems. You can save disk space by only installing the components needed for a particular use. You can restrict the use of specific computers by installing for example only selected office applications instead of the whole suite.

6.**Compatibility:** It runs all common UNIX software packages and can process all common file formats.

7.**Choice:** The large number of Linux distributions gives you a choice. Each distribution is developed and supported by a different organization. You can pick the one you like best; the

core functionalities are the same; most software runs on most distributions.

8.**Fast and easy installation:** Most Linux distributions come with user-friendly installation and setup programs. Popular Linux distributions come with tools that make installation of additional software very user friendly as well.

9. **Full use of hard disk:** Linux continues work well even when the hard disk is almost full.

10. **Multi-tasking:** Linux is designed to do many things at the same time; e.g., a large printing job in the background won't slow down your other work.

11.**Security:** Linux is one of the most secure operating systems. "Walls" and flexible file access permission systems prevent access by unwanted visitors or viruses. Linux users have to option to select and safely download software, free of charge, from online repositories containing thousands of high quality packages. No purchase transactions requiring credit card numbers or other sensitive personal information are necessary.

12.**Open Source:** If you develop software that requires knowledge or modification of the operating system code, LINUX's source code is at your fingertips. Most Linux applications are Open Source as well.

Difference between UNIX and LINUX

Features	LINUX	UNIX
Cost	Linux can be freely distributed, downloaded freely, distributed through magazines, Books etc. There are priced versions for Linux also, but they are normally cheaper than Windows.	Different flavors of Unix have different cost structures according to vendors
Development and Distribution	Linux is developed by Open Source development i.e. through sharing and collaboration of code and features through forums etc and it is distributed	Unix systems are divided into various other flavors, mostly developed by AT&T as well as various commercial vendors and non-profit organizations.

	by various vendors.	
Manufacturer	Linux kernel is developed by the community. Linus Torvalds oversees things.	Three biggest distributions are Solaris (Oracle), AIX (IBM) & HP-UX Hewlett Packard. And Apple Makes OSX, an unix based os..
User	Everyone. From home users to developers and computer enthusiasts alike.	Unix operating systems were developed mainly for mainframes, servers and workstations except OSX, Which is designed for everyone. The Unix environment and the client-server program model were essential elements in the development of the Internet
Usage	Linux can be installed on a wide variety of computer hardware, ranging from mobile phones, tablet computers and video game consoles, to mainframes and supercomputers.	The UNIX operating system is used in internet servers, workstations & PCs. Backbone of the majority of finance infrastructure and many 24x365 high availability solutions.
File system support	Ext2, Ext3, Ext4, Jfs, ReiserFS, Xfs, Btrfs, FAT, FAT32, NTFS	jfs, gpfs, hfs, hfs+, ufs, xfs, zfs format
Text mode interface	BASH (Bourne Again SHell) is the Linux default shell. It can support multiple command interpreters.	Originally the Bourne Shell. Now it's compatible with many others including BASH, Korn & C.
What is it?	Linux is an example of Open Source software development and Free Operating System (OS).	Unix is an operating system that is very popular in universities, companies, big enterprises etc.

GUI	Linux typically provides two GUIs, KDE and Gnome. But there are millions of alternatives such as LXDE, Xfce, Unity, Mate, twm, ect.	Initially Unix was a command based OS, but later a GUI was created called Common Desktop Environment. Most distributions now ship with Gnome.
Price	Free but support is available for a price.	Some free for development use (Solaris) but support is available for a price.
Security	Linux has had about 60-100 viruses listed till date. None of them actively spreads nowadays.	A rough estimate of UNIX viruses is between 85 -120 viruses reported till date.
Threat detection and solution	In case of Linux, threat detection and solution is very fast, as Linux is mainly community driven and whenever any Linux user posts any kind of threat, several developers start working on it from different parts of the world	Because of the proprietary nature of the original Unix, users have to wait for a while, to get the proper bug fixing patch. But these are not as common.
Processors	Dozens of different kinds.	x86/x64, Sparc, Power, Itanium, PA-RISC, PowerPC and many others.
Examples	Ubuntu, Fedora, Red Hat, Debian, Archlinux, Android etc.	OS X, Solaris, All Linux
Architectures	Originally developed for Intel's x86 hardware, ports available for over two dozen CPU types including ARM	is available on PA-RISC and Itanium machines. Solaris also available for x86/x64 based systems.OSX is PowerPC(10.0-

		10.5)/x86(10.4)/x64(10.5-10.8)
Inception	Inspired by MINIX (a Unix-like system) and eventually after adding many features of GUI, Drivers etc, Linus Torvalds developed the framework of the OS that became LINUX in 1992. The LINUX kernel was released on 17th September, 1991	In 1969, it was developed by a group of AT&T employees at Bell Labs and Dennis Ritchie. It was written in “C” language and was designed to be a portable, multi-tasking and multi-user system in a time-sharing configuration

Linux Distribution (Operating System) Names

A few popular names:

1.Redhat Enterprise Linux

2.Fedora Linux

3.Debian Linux

4.Suse Enterprise Linux

5.Ubuntu Linux

Common things between Linux & UNIX

Both share many common applications such as:

1.GUI, file, and windows managers (KDE, Gnome)

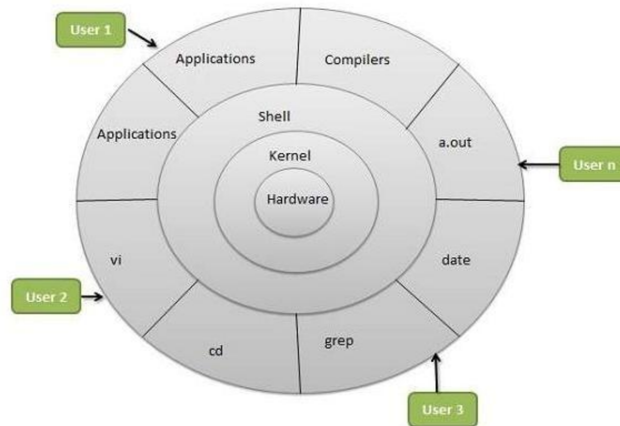
2.Shells (ksh, csh, bash)

3.Various office applications such as OpenOffice.org

4.Development tools (perl, php, python, GNU c/c++ compilers)

5.Posix interface

Layered Architecture:

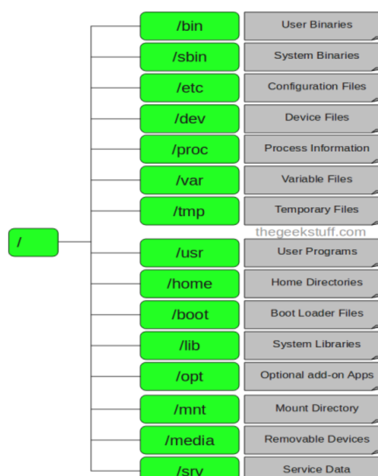


Linux System Architecture is consists of following layers

- **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. Takes commands from user and executes kernel's functions.
- **Utilities** - Utility programs giving user most of the functionalities of an operating systems.

LINUX File system

Linux file structure files are grouped according to purpose. Ex: commands, data files, documentation. Parts of a Unix directory tree are listed below. All directories are grouped under the root entry "/". That part of the directory tree is left out of the below diagram.



1. / – Root
   - Every single file and directory starts from the root directory.
   - Only root user has write privilege under this directory.
   - Please note that /root is root user's home directory, which is not same as /.
2. /bin – User Binaries
   - Contains binary executables.
   - Common linux commands you need to use in single-user modes are located under this directory.
   - Commands used by all the users of the system are located here.
   - For example: ps, ls, ping, grep, cp.
3. /sbin – System Binaries
   - Just like /bin, /sbin also contains binary executables.
   - But, the linux commands located under this directory are used typically by system aministrator, for system maintenance purpose.
   - For example: iptables, reboot, fdisk, ifconfig, swapon
4. /etc – Configuration Files
   - Contains configuration files required by all programs.
   - This also contains startup and shutdown shell scripts used to start/stop individual programs.
   - For example: /etc/resolv.conf, /etc/logrotate.conf
5. /dev – Device Files
   - Contains device files.
   - These include terminal devices, usb, or any device attached to the system.
   - For example: /dev/tty1, /dev/usbmon0
6. /proc – Process Information
   - Contains information about system process.
   - This is a pseudo filesystem contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid.
   - This is a virtual filesystem with text information about system resources. For example: /proc/uptime
7. /var – Variable Files
   - var stands for variable files.
   - Content of the files that are expected to grow can be found under this directory.

- This includes — system log files (/var/log); packages and database files (/var/lib); emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed across reboots (/var/tmp);

8. /tmp – Temporary Files
   - Directory that contains temporary files created by system and users.
   - Files under this directory are deleted when system is rebooted.
9. /usr – User Programs
   - Contains binaries, libraries, documentation, and source-code for second level programs.
   - /usr/bin contains binary files for user programs. If you can't find a user binary under /bin, look under /usr/bin. For example: at, awk, cc, less, scp
   - /usr/sbin contains binary files for system administrators. If you can't find a system binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel
   - /usr/lib contains libraries for /usr/bin and /usr/sbin
   - /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2
10. /home – Home Directories
    - Home directories for all users to store their personal files.
    - For example: /home/john, /home/nikita
11. /boot – Boot Loader Files
    - Contains boot loader related files.
    - Kernel initrd, vmlinux, grub files are located under /boot
    - For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic
12. /lib – System Libraries
    - Contains library files that supports the binaries located under /bin and /sbin
    - Library filenames are either ld* or lib*.so.*
    - For example: ld-2.11.1.so, libncurses.so.5.7
13. /opt – Optional add-on Applications
    - opt stands for optional.
    - Contains add-on applications from individual vendors.
    - add-on applications should be installed under either /opt/ or /opt/ sub-directory.
14. /mnt – Mount Directory
    - Temporary mount directory where sysadmins can mount filesystems.
15. /media – Removable Media Devices
    - Temporary mount directory for removable devices.

- For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer

16. /srv – Service Data

- srv stands for service.
- Contains server specific services related data.
- For example, /srv/cvs contains CVS related data.

Linux Utilities:

File Handling utilities:

Cat Command:

cat linux command concatenates files and print it on the standard output.

SYNTAX:

The Syntax is

cat [OPTIONS] [FILE]...

OPTIONS:

-A	Show all.
-b	Omits line numbers for blank space in the output.
-e	A $ character will be printed at the end of each line prior to a new line.
-E	Displays a $ (dollar sign) at the end of each line.
-n	Line numbers for all the output lines.
-s	If the output has multiple empty lines it replaces it with one empty line.
-T	Displays the tab characters in the output.
-v	Non-printing characters (with the exception of tabs, new-lines and form-feeds) are printed visibly.

Example:

To Create a new file:

cat > file1.txt

This command creates a new file file1.txt. After typing into the file press control+d (^d) simultaneously to end the file.

1. To Append data into the file: cat >> file1.txt

   To append data into the same file use append operator >> to write into the file, else the file will be overwritten (i.e., all of its contents will be erased).

2. To display a

   file: cat

   file1.txt

   This command displays the data in the file.

3. To concatenate several files and

   display: cat file1.txt file2.txt

   The above cat command will concatenate the two files (file1.txt and file2.txt) and it will display the output in the screen. Sometimes the output may not fit the monitor screen. In such situation you can print those files in a nlew file or display the file using less command.

   cat file1.txt file2.txt | less

4. To concatenate several files and to transfer the output to another file.

   cat file1.txt file2.txt > file3.txt

   In the above example the output is redirected to new file file3.txt. The cat command will create new file file3.txt and store the concatenated output into file3.txt.

rm COMMAND:

rm linux command is used to remove/delete the file from the directory.

SYNTAX:

The Syntax is

rm [options..] [file | directory]

OPTIONS:

-f	Remove all files in a directory without prompting the user.
-i	Interactive. With this option, rm prompts for confirmation before removing any files.
-r (or) -R	Recursively remove directories and subdirectories in the argument list. The directory will be emptied of files and removed. The user is normally prompted for removal of any write-protected files which the directory contains.

EXAMPLE:

1. To Remove / Delete a file:

   rm file1.txt

   Here rm command will remove/delete the file file1.txt.

2. To delete a directory tree:

   rm -ir tmp

   This rm command recursively removes the contents of all subdirectories of the tmp directory, prompting you regarding the removal of each file, and then removes the tmp directory itself.

3. To remove more files at once

   rm file1.txt file2.txt

   rm command removes file1.txt and file2.txt files at the same time.

cd COMMAND:

cd command is used to change the directory.

SYNTAX:

The Syntax is

cd [directory | ~ | ./ | ../ | -]

OPTIONS:

-L	Use the physical directory structure.
-P	Forces symbolic links.

EXAMPLE:

1. cd linux-command

   This command will take you to the sub-directory(linux-command) from its parent directory.

2. cd **..**

   This will change to the parent-directory from the current working directory/sub-directory.

3. cd ~

   This command will move to the user's home directory which is "/home/username".

cp COMMAND:

cp command copy files from one location to another. If the destination is an existing file, then the file is overwritten; if the destination is an existing directory, the file is copied into the directory (the directory is not overwritten).

SYNTAX:

The Syntax is

cp [OPTIONS]... SOURCE DEST

cp [OPTIONS]... SOURCE... DIRECTORY

cp [OPTIONS]... --target-directory=DIRECTORY SOURCE...

OPTIONS:

-a	same as -dpR.
--backup[=CONTROL]	make a backup of each existing destination file
-b	like --backup but does not accept an argument.
-f	if an existing destination file cannot be opened, remove it and try again.
-p	same as --preserve=mode,ownership,timestamps.
-preserve[=ATTR_LIST]	preserve the specified attributes (default: mode,ownership,timestamps) and security contexts, if possible additional attributes: links, all.
--no-preserve=ATTR_LIST	don't preserve the specified attribute.
--parents	append source path to DIRECTORY.

EXAMPLE:

Copy two files: cp file1 file2

The above cp command copies the content of file1.php to file2.php.

1. To backup the copied file: cp -b file1.php file2.php

   Backup of file1.php will be created with '~' symbol as file2.php~.
2. Copy folder and subfolders: cp -R scripts scripts1

   The above cp command copy the folder and subfolders from scripts to scripts1.

ls COMMAND:

ls command lists the files and directories under current working directory.

SYNTAX:

The
Syntax is

ls [OPTIONS]... [FILE]

OPTIONS:

-l	Lists all the files, directories and their mode, Number of links, owner of the file, file size, Modified date and time and filename.
-t	Lists in order of last modification time.
-a	Lists all entries including hidden files.
-d	Lists directory files instead of contents.
-p	Puts slash at the end of each directories.
-u	List in order of last access time.
-i	Display inode information.
-ltr	List files order by date.
-lSr	List files order by file size.

EXAMPLE:

Display root directory contents:

ls /

lists the contents of root directory.

1. Display hidden files and directories:

   ls -a

   lists all entries including hidden files and directories.

2. Display inode information:

   ls -i

   7373073 book.gif

   7373074 clock.gif

   7373082 globe.gif

   7373078 pencil.gif

   7373080 child.gif

23

7373081 email.gif

7373076 indigo.gif

The above command displays filename with inode value.

ln COMMAND:

ln command is used to create link to a file (or) directory. It helps to provide soft link for desired files. Inode will be different for source and destination.

SYNTAX:

The Syntax is

ln [options] existingfile(or directory)name newfile(or directory)name

OPTIONS:

-f	Link files without questioning the user, even if the mode of target forbids writing. This is the default if the standard input is not a terminal.
-n	Does not overwrite existing files.
-s	Used to create soft links.

EXAMPLE:

1. ln -s file1.txt file2.txt

   Creates a symbolic link to 'file1.txt' with the name of 'file2.txt'. Here inode for 'file1.txt' and 'file2.txt' will be different.
2. ln -s nimi nimi1

   Creates a symbolic link to 'nimi' with the name of 'nimi1'.

chown COMMAND:

chown command is used to change the owner / user of the file or directory. This is an admin command, root user only can change the owner of a file or directory.

SYNTAX:

The Syntax is

chown [options] newowner filename/directoryname

OPTIONS:

-R	Change the permission on files that are in the subdirectories of the directory that you are currently in.
-c	Change the permission for each file.
-f	Prevents chown from displaying error messages when it is unable to change the ownership of a file.

EXAMPLE:

1. chown hiox test.txt
   The owner of the 'test.txt' file is root, Change to new user hiox.
2. chown -R hiox test
   The owner of the 'test' directory is root, With -R option the files and subdirectories user also gets changed.
3. chown -c hiox calc.txt
   Here change the owner for the specific 'calc.txt' file only.

Security By File Permissions

Chmod Command:

chmod command allows you to alter / Change access rights to files and directories.

File Permission is given for users, group and others as,

Read	Write	Execute

User	☐
Group	☐
Others	☐
Permission	000

SYNTAX:

The Syntax is

chmod [options] [MODE] FileName

File Permission

#	File Permission
0	none

1	execute only
2	write only
3	write and execute
4	read only
5	read and execute
6	read and write
7	set all permissions

OPTIONS:

-c	Displays names of only those files whose permissions are being changed
-f	Suppress most error messages
-R	Change files and directories recursively
-v	Output version information and exit.

EXAMPLE:

1. To view your files with what permission they are:

   ls -alt

   This command is used to view your files with what permission they are.
2. To make a file readable and writable by the group and others.

   chmod 066 file1.txt
3. To allow everyone to read, write, and execute the file

   chmod 777 file1.txt

mkdir COMMAND:

mkdir command is used to create one or more directories.

SYNTAX:

The Syntax is

mkdir [options] directories

OPTIONS:

-m	Set the access mode for the new directories.
-p	Create intervening parent directories if they don't exist.
-v	Print help message for each directory created.

EXAMPLE:

1. Create directory:
   mkdir test
   The above command is used to create the directory 'test'.
2. Create directory and set permissions:
   mkdir -m 666 test
   The above command is used to create the directory 'test' and set the read and write permission.

rmdir COMMAND:

rmdir command is used to delete/remove a directory and its subdirectories.

SYNTAX:

The Syntax is

rmdir [options..] Directory

OPTIONS:

-p	Allow users to remove the directory dirname and its parent directories which become empty.

EXAMPLE:

1. To delete/remove a directory
   rmdir tmp
   rmdir command will remove/delete the directory tmp if the directory is empty.
2. To delete a directory tree:
   rm -ir tmp
   This command recursively removes the contents of all subdirectories of the tmp directory, prompting you regarding the removal of each file, and then removes the tmp directory itself.

mv COMMAND:

mv command which is short for move. It is used to move/rename file from one directory to another. mv command is different from cp command as it completely removes the file from the source and moves to the directory specified, where cp command just copies the content from one file to another.

SYNTAX:

The Syntax is

mv [-f] [-i] oldname newname

OPTIONS:

-f	This will not prompt before overwriting (equivalent to --reply=yes). mv -f will move the file(s) without prompting even if it is writing over an existing target.
-i	Prompts before overwriting another file.

EXAMPLE:

1. To Rename / Move a file:
   mv file1.txt file2.txt
   This command renames file1.txt as file2.txt
2. To move a directory
   mv hscripts tmp

   In the above line mv command moves all the files, directories and sub-directories from hscripts folder/directory to tmp directory if the tmp directory already exists. If there is no tmp directory it rename's the hscripts directory as tmp directory.

3. To Move multiple files/More files into another directory
   mv file1.txt tmp/file2.txt newdir
   This command moves the files file1.txt from the current directory and file2.txt from the tmp folder/directory to newdir.

diff COMMAND:

diff command is used to find differences between two files.

SYNTAX:

The Syntax is

diff [options..] from-file to-file

OPTIONS:

-a	Treat all files as text and compare them line-by-line.
-b	Ignore changes in amount of white space.
-c	Use the context output format.
-e	Make output that is a valid ed script.
-H	Use heuristics to speed handling of large files that have numerous scattered small changes.
-i	Ignore changes in case; consider upper- and lower-case letters equivalent.

-n	Prints in RCS-format, like -f except that each command specifies the number of lines affected.
-q	Output RCS-format diffs; like -f except that each command specifies the number of lines affected.
-r	When comparing directories, recursively compare any subdirectories found.
-s	Report when two files are the same.
-w	Ignore white space when comparing lines.
-y	Use the side by side output format.

EXAMPLE:

Lets create two files file1.txt and file2.txt and let it have the following data.

Data in file1.txt	Data in file2.txt
HIOX TEST	HIOX TEST
hscripts.com	HSCRIPTS.com
with friend ship	with friend ship
hiox india	

1. Compare files ignoring white space:

   diff -w file1.txt file2.txt

   This command will compare the file file1.txt with file2.txt ignoring white/blank space and it will produce the following output.

   ```
   2c2
   < hscripts.com
   ---
   > HSCRIPTS.com
   4d3
   < Hioxindia.com
   ```

2. Compare the files side by side, ignoring white space:

   diff -by file1.txt file2.txt

   This command will compare the files ignoring white/blank space, It is easier to differentiate the files.

   ```
   HIOX TEST            HIOX TEST
   hscripts.com       |  HSCRIPTS.com
   with friend ship      with  friend  ship
   Hioxindia.com       <
   ```

The third line(with friend ship) in file2.txt has more blank spaces, but still the -b ignores

the blank space and does not show changes in the particular line, -y printout the result side by side.

3. Compare the files ignoring case.

   diff -iy file1.txt file2.txt

   This command will compare the files ignoring case(upper-case and lower-case) and displays the following output.

   HIOX TEST   HIOX TEST

   hscripts.com   HSCRIPTS.com

   with friend ship | with friend ship

chgrp COMMAND:

chgrp command is used to change the group of the file or directory. This is an admin command. Root user only can change the group of the file or directory.

SYNTAX:

The Syntax is

chgrp [options] newgroup filename/directoryname

OPTIONS:

-R	Change the permission on files that are in the subdirectories of the directory that you are currently in.
-c	Change the permission for each file.
-f	Force. Do not report errors.

Hioxindia.com   <

EXAMPLE:

1. chgrp hiox test.txt

   The group of 'test.txt' file is root, Change to newgroup hiox.

2. chgrp -R hiox test

   The group of 'test' directory is root. With -R, the files and its subdirectories also changes to newgroup hiox.

3. chgrp -c hiox calc.txt

   They above command is used to change the group for the specific file('calc.txt') only.

About wc

Short for word count, wc displays a count of lines, words, and characters in a file.

Syntax

wc [-c | -m | -C] [-l] [-w] [file ...]

-c	Count bytes.
-m	Count characters.
-C	Same as -m.
-l	Count lines.
-w	Count words delimited by white space characters or new line characters. Delimiting characters are Extended Unix Code (EUC) characters from any code set defined by iswspace()
File	Name of file to word count.

Examples

wc myfile.txt - Displays information about the file myfile.txt. Below is an example of the output.

5 13 57 myfile.txt

5 = Lines

13 = Words

57 = Characters

About split

Split a file into pieces.

Syntax

split [-linecount | -l linecount] [-a suffixlength] [file [name]]

split -b n [k | m] [-a suffixlength] [file [name]]

-linecount | -l linecount	Number of lines in each piece. Defaults to 1000 lines.
-a suffixlength	Use suffixlength letters to form the suffix portion of the filenames of the split file. If -a is not specified, the default suffix length is 2. If the sum of the name operand and the suffixlength option-argument would create a filename exceeding NAME_MAX bytes, an error will result; split will exit with a diagnostic message and no files will be created.
-b n	Split a file into pieces n bytes in size.

-b n k Split a file into pieces n*1024 bytes in size.

-b n m Split a file into pieces n*1048576 bytes in size.

File The path name of the ordinary file to be split. If no input file is given or file is -, the standard input will be used.

name The prefix to be used for each of the files resulting from the split operation. If no name argument is given, x will be used as the prefix of the output files. The combined length of the basename of prefix and suffixlength cannot exceed NAME_MAX bytes; see OPTIONS.

Examples

split -b 22 newfile.txt new - would split the file "newfile.txt" into three separate files called newaa, newab and newac each file the size of 22.

split -l 300 file.txt new - would split the file "newfile.txt" into files beginning with the name "new" each containing 300 lines of text each

About settime and touch

Change file access and modification time.

Syntax

touch [-a] [-c] [-m] [-r ref_file | -t time] file

settime [-f ref_file] file

-a Change the access time of file. Do not change the modification time unless -m is also specified.

-c Do not create a specified file if it does not exist. Do not write any diagnostic messages concerning this condition.

-m Change the modification time of file. Do not change the access time unless -a is also specified.

-r ref_file Use the corresponding times of the file named by ref_file instead of the current time.

-t time   Use the specified time instead of the current time. time will be a decimal number of the form:

[[CC]YY]MMDDhhmm [.SS]

MM - The month of the year [01-12].
DD - The day of the month [01-31].
hh - The hour of the day [00-23].
mm - The minute of the hour [00-59].
CC - The first two digits of the year.
YY - The second two digits of the year.
SS - The second of the minute [00-61].

-f ref_file   Use the corresponding times of the file named by ref_file instead of the current time.

File   A path name of a file whose times are to be modified.

Examples
settime myfile.txt
Sets the file myfile.txt as the current time / date.
touch newfile.txt

Creates a file known as "newfile.txt", if the file does not already exist. If the file already exists the accessed / modification time is updated for the file newfile.txt

About comm
Select or reject lines common to two files.
Syntax
comm [-1] [-2] [-3] file1 file2

-1   Suppress the output column of lines unique to file1.

-2   Suppress the output column of lines unique to file2.

-3   Suppress the output column of lines duplicated in file1 and file2.

file1   Name of the first file to compare.

file2   Name of the second file to compare.

Examples
comm myfile1.txt myfile2.txt
The above example would compare the two files myfile1.txt and myfile2.txt.

Process utilities:

ps Command:

ps command is used to report the process status. ps is the short name for Process Status.

SYNTAX:

The Syntax is

ps [options]

OPTIONS:

-a	List information about all processes most frequently requested: all those except process group leaders and processes not associated with a terminal..
-A or e	List information for all processes.
-d	List information about all processes except session leaders.
-e	List information about every process now running.
-f	Generates a full listing.
-j	Print session ID and process group ID.
-l	Generate a long listing.

EXAMPLE:

1. ps
   Output:
   ```
   PID TTY          TIME CMD
   2540 pts/1    00:00:00 bash
   2621 pts/1    00:00:00 ps
   ```
   In the above example, typing ps alone would list the current running processes.
2. ps -f
   Output:
   ```
   UID        PID  PPID  C STIME TTY          TIME CMD
   nirmala   2540  2536  0 15:31 pts/1    00:00:00 bash
   nirmala   2639  2540  0 15:51 pts/1    00:00:00       ps      -f
   ```
   Displays full information about currently running processes.

kill COMMAND:

kill command is used to kill the background process.

SYNTAX:

The Syntax is

kill [-s] [-l] %pid

OPTIONS:

-s	Specify the signal to send. The signal may be given as a signal name or number.
-l	Write all values of signal supported by the implementation, if no operand is given.
-pid	Process id or job id.
-9	Force to kill a process.

EXAMPLE:

Step by Step process:

- Open a process music player.

  xmms

  press ctrl+z to stop the process.
- To know group id or job id of the background task.

  jobs -l
- It will list the background jobs with its job id as,
- xmms 3956

  kmail 3467
- To kill a job or process.

  kill 3956

kill command kills or terminates the background process xmms.

About nice

Invokes a command with an altered scheduling priority.

Syntax

nice [-increment | -n increment] command [argument ...]

-increment | -n increment	increment must be in the range 1-19; if not specified, an increment of 10 is assumed. An increment greater than 19 is equivalent to 19. The super-user may run commands with priority higher than normal by using a negative increment such as -10. A negative increment assigned by an unprivileged user is ignored.
command	The name of a command that is to be invoked. If command names any of the special built-in utilities, the results are undefined.
argument	Any string to be supplied as an argument when invoking command.

Examples

nice +13 pico myfile.txt - runs the pico command on myfile.txt with an increment of +13.

About at

Schedules a command to be ran at a particular time, such as a print job late at night.

Syntax

at	executes commands at a specified time.
atq	lists the user's pending jobs, unless the user is the superuser; in that case, everybody's jobs are listed. The format of the output lines (one for each job) is: Job number, date, hour, job class.
atrm	deletes jobs, identified by their job number.
batch	executes commands when system load levels permit; in other words, when the load average drops below 1.5, or the value specified in the invocation of atrun.

at [-c | -k | -s] [-f filename] [-q queuename] [-m] -t time [date] [-l] [-r]

-c	C shell. csh(1) is used to execute the at-job.
-k	Korn shell. ksh(1) is used to execute the at-job.
-s	Bourne shell. sh(1) is used to execute the at-job.
-f filename	Specifies the file that contains the command to run.
-m	Sends mail once the command has been run.

-t time Specifies at what time you want the command to be ran. Format hh:mm. am / pm indication can also follow the time otherwise a 24-hour clock is used. A timezone name of GMT, UCT or ZULU (case insensitive) can follow to specify that the time is in Coordinated Universal Time. Other timezones can be specified using the TZ environment variable. The below quick times can also be entered:
midnight - Indicates the time 12:00 am (00:00).
noon - Indicates the time 12:00 pm.
now - Indicates the current day and time. Invoking at - now will submit submit an at-job for potentially immediate execution.

date Specifies the date you wish it to be ran on. Format month, date, year. The following quick days can also be entered:
today - Indicates the current day.
tomorrow - Indicates the day following the current day.

-l Lists the commands that have been set to run.

-r Cancels the command that you have set in the past.

Examples

at -m 01:35 < atjob = Run the commands listed in the 'atjob' file at 1:35AM, in addition all output that is generated from job mail to the user running the task. When this command has been successfully enter you should receive a prompt similar to the below example.

Commands will be executed using /bin/csh job 1072250520.a at Wed Dec 24 00:22:00 2003

at -l = This command will list each of the scheduled jobs as seen below.

1072250520.a Wed Dec 24 00:22:00 2003

at -r 1072250520.a = Deletes the job just created.

or

atrm 23 = Deletes job 23.

If you wish to create a job that is repeated you could modify the file that executes the commands with another command that recreates the job or better yet use the crontab command.

Note: Performing just the **at** command at the prompt will give you an error "Garbled Time", this is a standard error message if no switch or time setting is given.

Disk utilities:

du (abbreviated from *disk usage*) is a standard Unix program used to estimate file space usage—space used under a particular directory or files on a file system.

du takes a single argument, specifying a pathname for du to work; if it is not specified, the current directory is used. The SUS mandates for du the following options:

-a, display an entry for each file (and not directory) contained in the current directory

-H, calculate disk usage for link references specified on the command line

-k, show sizes as multiples of 1024 bytes, not 512-byte

-L, calculate disk usage for link references anywhere

-s, report only the sum of the usage in the current directory, not for each file

-x, only traverse files and directories on the device on which the pathname argument is specified.

Other Unix and Unix-like operating systems may add extra options. For example, BSD and GNU du specify a -h option, displaying disk usage in a format easier to read by the user, adding units with the appropriate SI prefix'

```
$ du -sk *
 152304 directoryOne
 1856548 directoryTwo
```

Sum of directories in human-readable format (Byte, Kilobyte, Megabyte, Gigabyte, Terabyte and Petabyte):

```
 $ du -sh *
 149M directoryOne
 1.8G directoryTwo
```

disk usage of all subdirectories and files including hidden files within the current directory (sorted by filesize) :

```
 $ du -sk .[!.]* *| sort -n
```

disk usage of all subdirectories and files including hidden files within the current directory (sorted by reverse filesize) :

```
 $ du -sk .[!.]* *| sort –nr
```

The weight of directories:

```
 $ du -d 1 -c -h
```

df command : Report file system disk space usage
Df command examples - to check free disk space
Type df -h or df -k to list free disk space:
$ df -h
OR
$ df –k
Output:

```
Filesystem          Size Used Avail Use% Mounted on
/dev/sdb1           20G 9.2G 9.6G 49% /
varrun              393M   144k   393M   1%  /var/run
varlock             393M   0   393M   0%   /var/lock
procbususb          393M  123k  393M  1%  /proc/bus/usb
udev                393M 123k 393M 1% /dev
devshm              393M   0 393M 0% /dev/shm
lrm                 393M  35M 359M 9% /lib/modules/2.6.20-15-generic/volatile
/dev/sdb5           29G  5.4G  22G 20% /media/docs
/dev/sdb3           30G  5.9G  23G 21% /media/isomp3s
/dev/sda1           8.5G  4.3G  4.3G  51% /media/xp1
/dev/sda2           12G  6.5G  5.2G  56% /media/xp2
/dev/sdc1           40G  3.1G  35G 9% /media/backup
```

du command examples
du shows how much space one ore more files or directories is using.
$ du -sh
103M
-s option summarize the space a directory is using and -h option provides "Human-readable" output.

Networking commands:

These are most useful commands in my list while working on Linux server , this enables you to quickly troubleshoot connection issues e.g. whether other system is connected or not , whether other host is responding or not and while working for FIX connectivity for advanced trading system this tools saves quite a lot of time .

This article is in continuation of my article How to work fast in Unix and Unix Command tutorials and Examples for beginners.

• finding host/domain name and IP address - **hostname**
• test network connection – **ping**
• getting network configuration – **ifconfig**
• Network connections, routing tables, interface statistics – **netstat**
• query DNS lookup name – **nslookup**
• communicate with other hostname – **telnet**
• outing steps that packets take to get to network host – **traceroute**
• view user information – **finger**

• checking status of destination host - **telnet**

Example of Networking commands in Unix

let's see some example of various networking command in Unix and Linux. Some of them are quite basic e.g. ping and telnet and some are more powerful e.g. nslookup and netstat. When you used these commands in combination of find and grep you can get anything you are looking for e.g. hostname, connection end points, connection status etc.

hostname

hostname *with no options displays the machines host name*

hostname –d *displays the domain name the machine belongs to*

hostname –f *displays the fully qualified host and domain name*

hostname –i *displays the IP address for the current machine*

ping

It sends packets of information to the user-defined source. If the packets are received, the destination device sends packets back. Ping can be used for two purposes

1. To ensure that a network connection can be established.
2. Timing information as to the speed of the connection.

If you **do ping www.yahoo.com** it will display its IP address. Use ctrl+C to stop the test.

ifconfig

View network configuration, it displays the current network adapter configuration. It is handy to determine if you are getting transmit (TX) or receive (RX) errors.

netstat

Most useful and very versatile for finding connection to and from the host. You can find out all the multicast groups (network) subscribed by this host by issuing **"netstat -g"**

netstat -nap | grep port *will display process id of application which is using that port*

netstat -a or netstat –all *will display all connections including TCP and UDP*

netstat --tcp or netstat –t *will display only TCP connection*

netstat --udp or netstat –u *will display only UDP connection*

netstat -g *will display all multicast network subscribed by this host.*

slookup

If you know the IP address it will display hostname. To find all the IP addresses for a given domain name, the command nslookup is used. You must have a connection to the internet for this utility to be useful.

E.g. **nslookup blogger.com**

You can also use nslookup to convert hostname to IP Address and from IP Address from hostname.

traceroute

A handy utility to view the number of hops and response time to get to a remote system or web site is traceroute. Again you need an internet connection to make use of this tool.

finger

View user information, displays a user’s login name, real name, terminal name and write status. this is pretty old unix command and rarely used now days.

telnet

Connects destination host via telnet protocol, if telnet connection establish on any port means connectivity between two hosts is working fine.

telnet hostname port *will telnet hostname with the port specified. Normally it is used to see*

whether host is alive and network connection is fine or not.

10 Most important linux networking commands

Linux is most powerful operating system which often needs to use commands to explore it effectively.Some of the commands are restricted to normal user groups as they are powerful and has more functionality involved in it.Here we summarized most interesting and useful networking commands which every linux user are supposed to be familiar with it.

1.Arp manipulates the kernel's ARP cache in various ways. The primary options are clearing an address mapping entry and manually setting up one. For debugging purposes, the arp program also allows a complete dump of the ARP cache.ARP displays the IP address assigned to particular ETH card and mac address

```
[fasil@smashtech ]# arp
Address          HWtype HWaddress       Flags Mask       Iface
59.36.13.1       ether     C                  eth0
```

2.Ifconfig is used to configure the network interfaces. Normally we use this command tocheck the IP address assigned to the system.It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.

```
[fasil@smashtech ~]# /sbin/ifconfig
eth0    UP BROADCAST RUNNING MULTICAST MTU:1500  Metric:1
        RX packets:126341 errors:0 dropped:0 overruns:0 frame:0
        TX packets:44441 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
```

3. Netstat prints information about the networking subsystem. The type of information which is usually printed by netstat are Print network connections, routing tables, interface statistics, masquerade connections, and multicast.

```
[fasil@smashtech ~]# netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0     0              .230.87:https         ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node Path
unix   10   [ ]     DGRAM                    4970 /dev/log
unix   2    [ ]     DGRAM                    6625  @/var/run/hal/hotplug_socket
unix   2    [ ]     DGRAM                    2952 @udevd
unix   2    [ ]     DGRAM                    100564
unix   3    [ ]     STREAM      CONNECTED    62438  /tmp/.X11-unix/X0
unix   3    [ ]     STREAM      CONNECTED    62437
```

```
unix  3   [ ]   STREAM   CONNECTED   10271  @/tmp/fam-root-
unix  3   [ ]   STREAM   CONNECTED   10270
unix  3   [ ]   STREAM   CONNECTED   9276
unix  3   [ ]   STREAM   CONNECTED   9275
```

4.ping command is used to check the connectivity of a system to a network.Whenever there is problem in network connectivity we use ping to ensure the system is connected to network.

```
[root@smashtech ~]# ping google.com
PING google.com (74.125.45.100) 56(84) bytes of data.
64 bytes from yx-in-f100.google.com (74.125.45.100): icmp_seq=0 ttl=241 time=295 ms
64 bytes from yx-in-f100.google.com (74.125.45.100): icmp_seq=1 ttl=241 time=277 ms
64 bytes from yx-in-f100.google.com (74.125.45.100): icmp_seq=2 ttl=241 time=277 ms

--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 6332ms
rtt min/avg/max/mdev = 277.041/283.387/295.903/8.860 ms, pipe 2
```

5. Nslookup is a program to query Internet domain name servers. Nslookup has two modes: interactive and non-interactive. Interactive mode allows the user to query name servers for information about various hosts and domains or to print a list of hosts in a domain. Non-interactive mode is used to print just the name and requested information for a host or domain.

```
[fasil@smashtech ~]# nslookup google.com
Server:      server ip
Address:     gateway ip 3

Non-authoritative answer:
Name:        google.com
Address: 209.85.171.100
Name:      google.com
Address: 74.125.45.100
Name:      google.com
Address: 74.125.67.100
```

6. dig (domain information groper) is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. Most DNS administrators use dig to troubleshoot DNS problems because of its flexibility, ease of use and clarity of output. Other lookup tools tend to have less functionality than dig.

```
[fasil@smashtech ~]# dig google.com
```

```
; <<>> DiG 9.2.4 <<>> google.com
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4716
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 4, ADDITIONAL: 4

;; QUESTION SECTION:
;google.com.                IN      A

;; ANSWER SECTION:
google.com.      122        IN   A        74.125.45.100
google.com.      122        IN   A        74.125.67.100
google.com.      122        IN   A        209.85.171.100

;; AUTHORITY SECTION:
google.com.      326567       IN   NS       ns3.google.com.
google.com.      326567       IN   NS       ns4.google.com.
google.com.      326567       IN   NS       ns1.google.com.
google.com.      326567       IN   NS       ns2.google.com.

;; ADDITIONAL SECTION:
ns1.google.com.    152216      IN   A        216.239.32.10
ns2.google.com.    152216      IN   A        216.239.34.10
ns3.google.com.    152216      IN   A        216.239.36.10
ns4.google.com.    152216      IN   A        216.239.38.10

;; Query time: 92 msec
;; SERVER: 172.29.36.1#53(172.29.36.1)
;; WHEN: Thu Mar 5 14:38:45 2009
;; MSG SIZE rcvd: 212
```

7. Route manipulates the IP routing tables. Its primary use is to set up static routes to specific hosts or networks via an interface after it has been configured with the ifconfig program.When the add or del options are used, route modifies the routing tables. Without these options, route displays the current contents of the routing tables.

```
[fasil@smashtech ~]# route
Kernel IP routing table
Destination     Gateway        Genmask         Flags Metric Ref    Use Iface

 54.192.56.321    *             255.255.255.0   U     0      0       0 eth0
    *            255.255.0.0    U     0      0       0 eth0
default          0.0.0.0        UG    0      0       0 eth0
```

8.Traceroute : Internet is a large and complex aggregation of network hardware, connected together by gateways. Tracking the route one's packets follow (or finding the miscreant gateway that's discarding your packets) can be difficult.
Traceroute utilizes the IP protocol 'time to live' field and attempts to elicit an ICMP TIME_EXCEEDED response from each gateway along the path to some host. The only mandatory parameter is the destination host name or IP number. The default probe datagram length is 40 bytes, but this may be increased by specifying a packet length (in bytes) after the destination host name.

```
[fasil@smashtech ~]# traceroute google.com
traceroute: Warning: google.com has multiple addresses; using 209.85.171.100
traceroute to google.com (209.85.171.100), 30 hops max, 38 byte packets
 1 * * *
```

9.W-displays information about the users currently on the machine, and their processes. The header shows, in this order, the current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes.

```
[fasil@smashtechl ~]# w
 15:18:22 up 4:38, 3 users, load average: 0.89, 0.34, 0.19
USER     TTY      FROM              LOGIN@ IDLE JCPU PCPU WHAT
root     :0       -                10:41  ?xdm?  24:53  1.35s  /usr/bin/gnome-session
root     pts/1    :0.0               10:58    1.00s 0.34s 0.00s w
root     pts/2    :0.0               12:10 23:32 0.03s 0.03s bash
```

Filters:

more COMMAND:

more command is used to display text in the terminal screen. It allows only backward movement.

SYNTAX:

The Syntax is

more [options] filename

OPTIONS:

-c	Clear screen before displaying.
-e	Exit immediately after writing the last line of the last file in the argument list.
-n	Specify how many lines are printed in the screen for a given file.
+n	Starts up the file from the given number.

EXAMPLE:

1. more -c index.php
   Clears the screen before printing the file .
2. more -3 index.php

Prints first three lines of the given file. Press **Enter** to display the file line by line.

head COMMAND:

head command is used to display the first ten lines of a file, and also specifies how many lines to display.

SYNTAX:

The Syntax is

head [options] filename

OPTIONS:

-n	To specify how many lines you want to display.
-n number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines.
-c number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes.

EXAMPLE:

1. head index.php
   This command prints the first 10 lines of 'index.php'.
2. head -5 index.php
   The head command displays the first 5 lines of 'index.php'.
3. head -c 5 index.php
   The above command displays the first 5 characters of 'index.php'.

tail COMMAND:

tail command is used to display the last or bottom part of the file. By default it displays last 10 lines of a file.

SYNTAX:

The Syntax is

tail [options] filename

OPTIONS:

-l	To specify the units of lines.
-b	To specify the units of blocks.
-n	To specify how many lines you want to display.
-c number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes.

-n number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines.

EXAMPLE:

1. tail index.php
   It displays the last 10 lines of 'index.php'.
2. tail -2 index.php
   It displays the last 2 lines of 'index.php'.
3. tail -n 5 index.php
   It displays the last 5 lines of 'index.php'.
4. tail -c 5 index.php
   It displays the last 5 characters of 'index.php'.

cut COMMAND:

cut command is used to cut out selected fields of each line of a file. The cut command uses delimiters to determine where to split fields.

SYNTAX:

The Syntax is

cut [options]

OPTIONS:

-c	Specifies character positions.
-b	Specifies byte positions.
-d flags	Specifies the delimiters and fields.

EXAMPLE:

1. cut -c1-3 text.txt
   Output:
   Thi
   Cut the first three letters from the above line.
2. cut -d, -f1,2 text.txt
   Output:
   This is, an example program
   The above command is used to split the fields using delimiter and cut the first two fields.

paste COMMAND:

paste command is used to paste the content from one file to another file. It is also used to set column format for each line.

SYNTAX:

The Syntax is

paste [options]

OPTIONS:

-s Paste one file at a time instead of in parallel.

-d Reuse characters from LIST instead of TABs .

EXAMPLE:

1. paste test.txt>test1.txt
   Paste the content from 'test.txt' file to 'test1.txt' file.
2. ls | paste - - - -
   List all files and directories in four columns for each line.

sort COMMAND:

sort command is used to sort the lines in a text file.

SYNTAX:

The Syntax is

sort [options] filename

OPTIONS:

-r Sorts in reverse order.

-u If line is duplicated display only once.

-o filename Sends sorted output to a file.

EXAMPLE:

1. sort test.txt
   Sorts the 'test.txt'file and prints result in the screen.
2. sort -r test.txt
   Sorts the 'test.txt' file in reverse order and prints result in the screen.

About uniq
Report or filter out repeated lines in a file.

Syntax
uniq [-c | -d | -u] [-f fields] [-s char] [-n] [+m] [input_file [output_file]]

-c Precede each output line with a count of the number of times the line occurred in the input.

-d Suppress the writing of lines that are not repeated in the input.

-u Suppress the writing of lines that are repeated in the input.

-f fields Ignore the first fields fields on each input line when doing comparisons, where fields is a positive decimal integer. A field is the maximal string matched by the basic regular expression:
[[:blank:]]*[^[:blank:]]*
If fields specifies more fields than appear on an input line, a null string will be used for comparison.

-s char Ignore the first chars characters when doing comparisons, where chars is a positive decimal integer. If specified in conjunction with the -f option, the first chars characters after the first fields fields will be ignored. If chars specifies more characters than remain on an input line, a null string will be used for comparison.

-n Equivalent to -f fields with fields set to n.

+m Equivalent to -s chars with chars set to m.

input_file A path name of the input file. If input_file is not specified, or if the input_file is - ,the standard input will be used.

output_file A path name of the output file. If output_file is not specified, the standard output will be used. The results are unspecified if the file named by output_file is the file named by input_file.

Examples
uniq myfile1.txt > myfile2.txt - Removes duplicate lines in the first file1.txt and outputs the results to the second file.

About tr
Translate characters.
Syntax
tr [-c] [-d] [-s] [string1] [string2]

-c Complement the set of characters specified by string1.

-d Delete all occurrences of input characters that are specified by string1.

-s Replace instances of repeated characters with a single character.

string1 First string or character to be changed.

string2 Second string or character to change the string1.

Examples

echo "12345678 9247" | tr 123456789 computerh - this example takes an echo response of '12345678 9247' and pipes it through the tr replacing the appropriate numbers with the letters. In this example it would return *computer hope*.

tr -cd '\11\12\40-\176' < myfile1 > myfile2 - this example would take the file myfile1 and strip all non printable characters and take that results to myfile2.

Text processing utilities and Backup utilities:

Text processing utilities:

cat : concatenate files and print on the standard output

Usage: cat [OPTION] [FILE]...

eg. cat file1.txt file2.txt

cat n

file1.txt

echo : display a line of text

Usage: echo [OPTION] [string] ...

eg. echo I love India

echo $HOME

wc: print the number of newlines, words, and bytes in files

Usage: wc [OPTION]... [FILE]...

eg. wc file1.txt

wc L

file1.txt

sort :sort lines of text files

Usage: sort [OPTION]... [FILE]...

eg. sort file1.txt

sort r

file1.txt

General Commands:

date COMMAND:

date command prints the date and time.

SYNTAX:

The Syntax is

date [options] [+format] [date]

OPTIONS:

-a	Slowly adjust the time by sss.fff seconds (fff represents fractions of a second). This adjustment can be positive or negative.Only system admin/ super user can adjust the time.
- date - string	Sets the time and date to the value specfied in the datestring. The datestr may contain the month names, timezones, 'am', 'pm', etc.
-u	Display (or set) the date in Greenwich Mean Time (GMT-universal time).

Format:

%a	Abbreviated weekday(Tue).
%A	Full weekday(Tuesday).
%b	Abbreviated month name(Jan).
%B	Full month name(January).
%c	Country-specific date and time format..
%D	Date in the format %m/%d/%y.
%j	Julian day of year (001-366).
%n	Insert a new line.
%p	String to indicate a.m. or p.m.
%T	Time in the format %H:%M:%S.
%t	Tab space.
%V	Week number in year (01-52); start week on Monday.

EXAMPLE:

date command

date

The above command will print Wed Jul 23 10:52:34 IST 2008

1. To use tab space:

   date +"Date is %D %t Time is %T"

   The above command will remove space and print as

   Date is 07/23/08 Time is 10:52:34

2. To know the week number of the year,

   date -V

   The above command will print 30

3. To set the date,

   date -s "10/08/2008 11:37:23"

   The above command will print Wed Oct 08 11:37:23 IST 2008

who COMMAND:

who command can list the names of users currently logged in, their terminal, the time they have been logged in, and the name of the host from which they have logged in.

SYNTAX:

The Syntax is

who [options] [file]

OPTIONS:

am i	Print the username of the invoking user, The 'am' and 'i' must be space separated.
-b	Prints time of last system boot.
-d	print dead processes.
-H	Print column headings above the output.
-i	Include idle time as HOURS:MINUTES. An idle time of . indicates activity within the last minute.
-m	Same as who am i.
-q	Prints only the usernames and the user count/total no of users logged in.
-T,-w	Include user's message status in the output.

EXAMPLE:

1. who –Uh

   Output:

   NAME LINE TIME IDLE PID COMMENT
   hiox ttyp3 Jul 10 11:08 . 4578

   This sample output was produced at 11 a.m. The "." indiacates activity within the last minute.

2. who am i

   who am i command prints the user name.

echo COMMAND:

echo command prints the given input string to standard output.

SYNTAX:

The Syntax is

echo [options..] [string]

OPTIONS:

-n	do not output the trailing newline
-e	enable interpretation of the backslash-escaped characters listed below
-E	disable interpretation of those sequences in STRINGs

Without -E, the following sequences are recognized and interpolated:

\NNN	the character whose ASCII code is NNN (octal)
\a	alert (BEL)
\\	backslash
\b	backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

EXAMPLE:

echo command

echo "hscripts Hiox India"

The above command will print as hscripts Hiox India

1. To use backspace:
   echo -e "hscripts \bHiox \bIndia"
   The above command will remove space and print as hscriptsHioxIndia
2. To use tab space in echo command

   echo -e "hscripts\tHiox\tIndia"

   The above command will print as hscripts Hiox India

passwd COMMAND:

passwd command is used to change your password.

SYNTAX:

The Syntax is

passwd [options]

OPTIONS:

-a	Show password attributes for all entries.
-l	Locks password entry for name.
-d	Deletes password for name. The login name will not be prompted for password.
-f	Force the user to change password at the next login by expiring the password for name.

EXAMPLE:

1. passwd
   Entering just passwd would allow you to change the password. After entering passwd you will receive the following three prompts:

   Current Password:

   New Password:

   Confirm New Password:
   Each of these prompts must be entered correctly for the password to be successfully changed.

pwd COMMAND:

pwd - Print Working Directory. pwd command prints the full filename of the current working directory.

SYNTAX:

The Syntax is

pwd [options]

OPTIONS:

-P The pathname printed will not contain symbolic links.

-L The pathname printed may contain symbolic links.

EXAMPLE:

1. Displays the current working directory.

   pwd

   If you are working in home directory then, pwd command displays the current working directory as /home.

cal COMMAND:

cal command is used to display the calendar.

SYNTAX:

The Syntax is

cal [options] [month] [year]

OPTIONS:

-1 Displays single month as output.

-3 Displays prev/current/next month output.

-s Displays sunday as the first day of the week.

-m Displays Monday as the first day of the week.

-j Displays Julian dates (days one-based, numbered from January 1).

-y Displays a calendar for the current year.

EXAMPLE:

1. cal

   Output:

```
  September 2008
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
```

cal command displays the current month calendar.

2. cal -3 5 2008

Output:

```
  April 2008            May 2008            June 2008
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa
      1  2  3  4  5             1 2 3 1 2 3 4 5 6  7
 6 7 8 9 10 11 12 4 5 6 7 8 9 10 8 9 10 11 12 13 14
13 14 15 16 17 18 19  11 12 13 14 15 16 17  15 16 17 18 19 20 21
20 21 22 23 24 25 26  18 19 20 21 22 23 24  22 23 24 25 26 27 28
27 28 29 30          25 26 27 28 29 30 31 29 30
```

Here the cal command displays the calendar of April, May and June month of year 2008.

login Command

Signs into a new system.

Syntax

login [-p] [-d device] [-h hostname | terminal | -r hostname] [name [environ]]

-p	Used to pass environment variables to the login shell.
-d device	login accepts a device option, device. device is taken to be the path name of the TTY port login is to operate on. The use of the device option can be expected to improve login performance, since login will not need to call ttyname. The -d option is available only to users whose UID and effective UID are root. Any other attempt to use -d will cause login to quietly exit.
-h hostname | terminal	Used by in.telnetd to pass information about the remote host and terminal type.
-r hostname	Used by in.rlogind to pass information about the remote host.

Examples

login computerhope.com - Would attempt to login to the computerhope domain.

uname command

Print name of current system.

Syntax

uname [-a] [-i] [-m] [-n] [-p] [-r] [-s] [-v] [-X] [-S systemname]

-a	Print basic information currently available from the system.
-i	Print the name of the hardware implementation (platform).

-m	Print the machine hardware name (class). Use of this option is discouraged; use uname -p instead.
-n	Print the nodename (the nodename is the name by which the system is known to a communications network).
-p	Print the current host's ISA or processor type.
-r	Print the operating system release level.
-s	Print the name of the operating system. This is the default.
-v	Print the operating system version.
-X	Print expanded system information, one information element per line, as expected by SCO Unix. The displayed information includes: • system name, node, release, version, machine, and number of CPUs. • BusType, Serial, and Users (set to "unknown" in Solaris) • OEM# and Origin# (set to 0 and 1, respectively)
-S systemname	The nodename may be changed by specifying a system name argument. The system name argument is restricted to SYS_NMLN characters. SYS_NMLN is an implementation specific value defined in <sys/utsname.h>. Only the super-user is allowed this capability.

Examples

uname -arv

List the basic system information, OS release, and OS version as shown below.

SunOS hope 5.7 Generic_106541-08 sun4m sparc SUNW,SPARCstation-10

uname -p

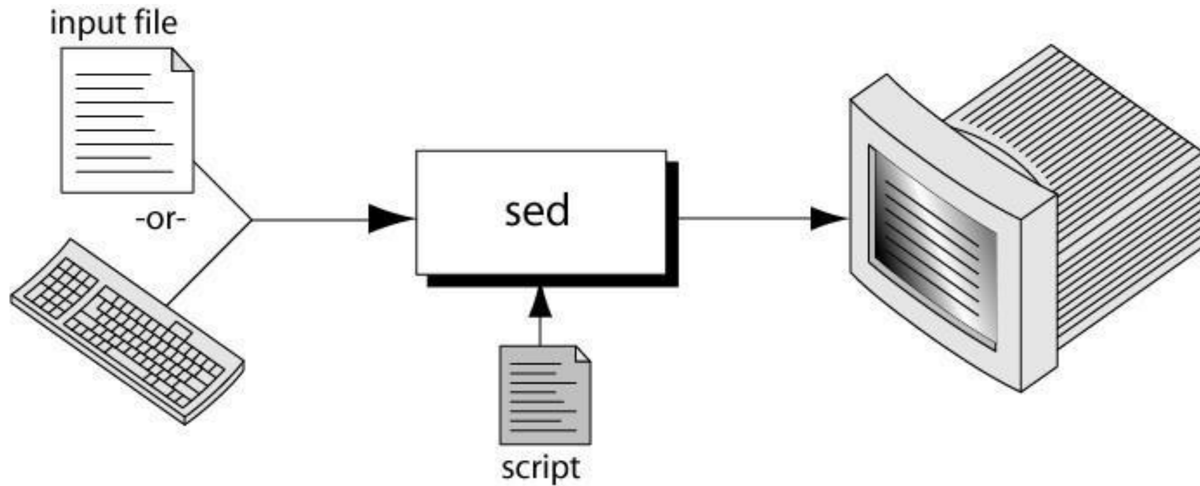
Display the Linux platform.

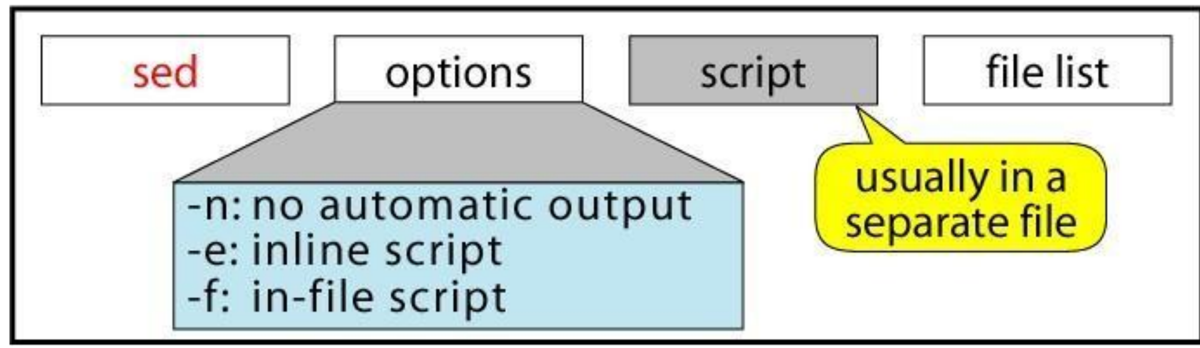
SED:

What is sed?

- A non-interactive stream editor
- Interprets sed instructions and performs actions
- Use sed to:

- Automatically perform edits on file(s)
- Simplify doing the same edits on multiple files
- Write conversion programs





Sed Command Syntax(Sed Scripts):

```
$ sed -e 'address command' input_file
```

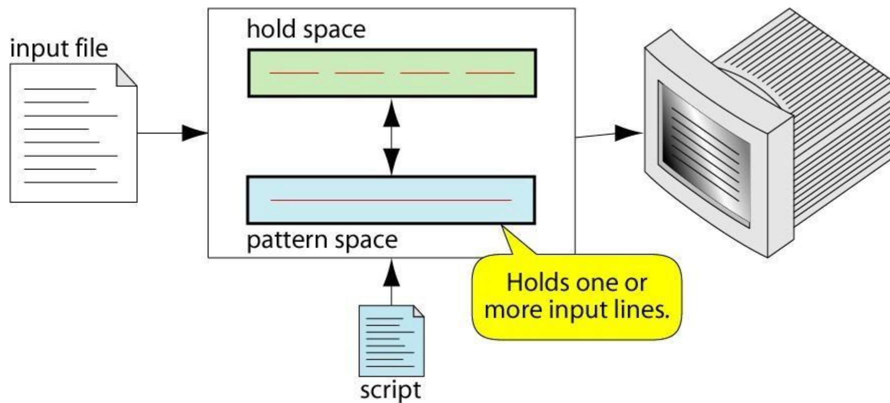
(a) Inline Script

```
$ sed -f script.sed input_file
```

(b) Script File

Sed Operation

How Does sed Work?



- sed reads line of input
- line of input is copied into a temporary buffer called pattern space
- editing commands are applied
- subsequent commands are applied to line in the pattern space, not the original input line
- once finished, line is sent to output (unless –n option was used)
- line is removed from pattern space
- sed reads next line of input, until end of file

Note: input file is unchanged

sed instruction format(Sed Addresses):

- address determines which lines in the input file are to be processed by the command(s)
- if no address is specified, then the command is applied to each input line
- address types:
  - Single-Line address
  - Set-of-Lines address
  - Range address
  - Nested address

Single-Line Address

- Specifies only one line in the input file
  - special: dollar sign ($) denotes last line of input file

Examples:

- show only line 3

sed -n -e '3 p' input-file

- show only last line

sed -n -e '$ p' input-file

- substitute "endif" with "fi" on line 10

sed -e '10 s/endif/fi/' input-file

Set-of-Lines Address

- use regular expression to match lines
  - written between two slashes
  - process only lines that match
  - may match several lines
  - lines may or may not be consecutives

Examples:

sed -e '/key/ s/more/other/' input-file

sed -n -e '/r..t/ p' input-file

Range Address

- Defines a set of consecutive lines

Format:

start-addr,end-addr (inclusive)

Examples:

10,50 line-number,line-number

10,/R.E/ line-number,/RegExp/

/R.E./,10 /RegExp/,line-number

/R.E./,/R.E/ /RegExp/,/RegExp/

Example: Range Address

% sed -n -e '/^BEGIN$/,/^END$/p' input-file

- Print lines between BEGIN and END, inclusive

BEGIN
Line 1 of input

Line 2 of input

Line3 of input

END

Line 4 of input

Line 5 of input

Nested Address

- Nested address contained within another address

Example:

print blank lines between line 20 and 30

```
20,30{

/^$/ p
}
```

Address with !

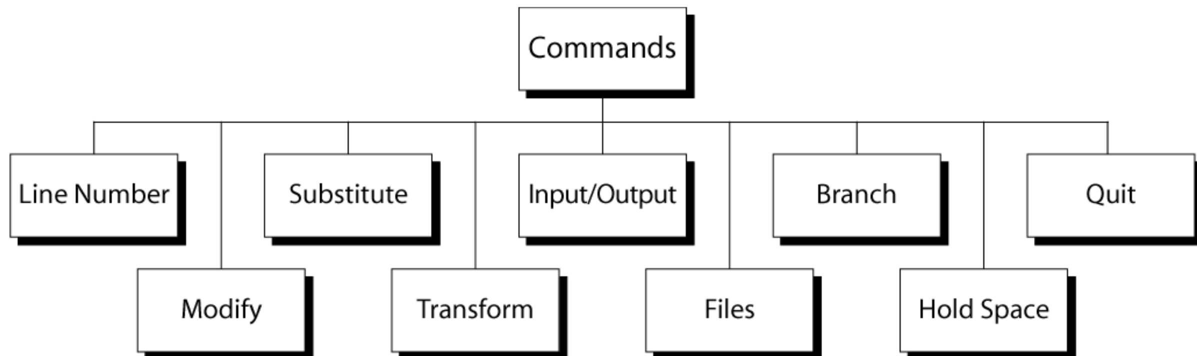
- address with an exclamation point (!):
  instruction will be applied to all lines that do not match the address

Example:

print lines that do not contain “obsolete”

sed -e ‘/obsolete/!p’ input-file

sed commands



Line Number

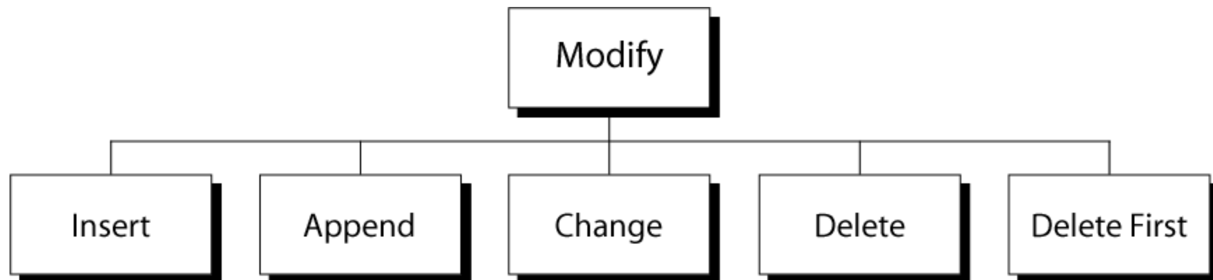
- line number command (=) writes the current line number before each matched/output line

Examples:

sed -e '/Two-thirds-time/=' tuition.data

sed -e '/^[0-9][0-9]/=' inventory

modify commands



Insert Command: i

- adds one or more lines directly to the output before the address:
    - inserted "text" never appears in sed's pattern space
    - cannot be used with a range address; can only be used with the single-line and set-of-lines address types

Syntax:

[address] i

text

Append Command: a

- adds one or more lines directly to the output after the address:
    - Similar to the insert command (i), append cannot be used with a range address.
    - Appended "text" does not appear in sed's pattern space.

Syntax:

[address] a

text

Change Command: c

- replaces an entire matched line with new text
- accepts four address types:
    - single-line, set-of-line, range, and nested addresses.

Syntax:

[address1[,address2]] c

text

Delete Command: d

- deletes the entire pattern space
    - commands following the delete command are ignored since the deleted text is no

longer in the pattern space

Syntax:

[address1[,address2]] d

Substitute Command (s)

Syntax:

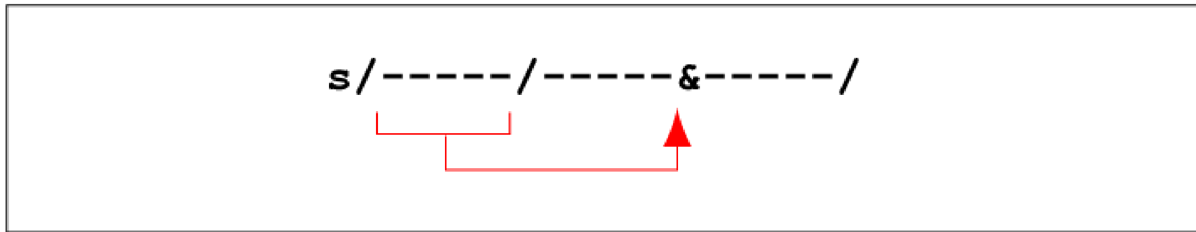
[addr1][,addr2] s/search/replace/[flags]

- replaces text selected by search string with replacement string
- search string can be regular expression
- flags:
  - global (g), i.e. replace all occurrences
  - specific substitution count (integer), default 1

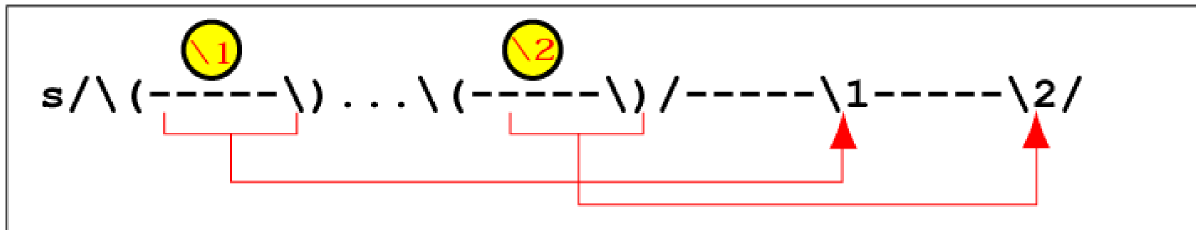
Regular Expressions: use with sed

Metacharacter	Description/Matches…
.	Any one character, except new line
*	Zero or more of preceding character
^	A character at beginning of line
$	A character at end of line
\char	Escape the meaning of *char* following it
[ ]	Any one of the enclosed characters
\(\)	Tags matched characters to be used later
x\{m\}	Repetition of character x, m times
\<	Beginning of word
\>	End of word

Substitution Back References



(a) Whole Pattern Substitution



(b) Numbered Buffer Substitution

Example: Replacement String &

$ cat datafile

Charles Main	**3.0**	**.98**	**3**	**34**
Sharon Gray	**5.3**	**.97**	**5**	**23**
Patricia Hemenway	**4.0**	**.7**	**4**	**17**
TB Savage	**4.4**	**.84**	**5**	**20**
AM Main Jr.	**5.1**	**.94**	**3**	**13**
Margot Weber	**4.5**	**.89**	**5**	**9**
Ann Stephens	**5.7**	**.94**	**5**	**13**

$ sed -e ‘s/[0-9][0-9]$/&.5/’ datafile

Charles Main	**3.0**	**.98**	**3**	**34.5**
Sharon Gray	**5.3**	**.97**	**5**	**23.5**

Patricia Hemenway 4.0 .7 4 17.5

TB Savage 4.4 .84 5 20.5

AM Main Jr. 5.1 .94 3 13.5

Margot Weber 4.5 .89 5 9

Ann Stephens 5.7 .94 5 13.5

Transform Command (y)

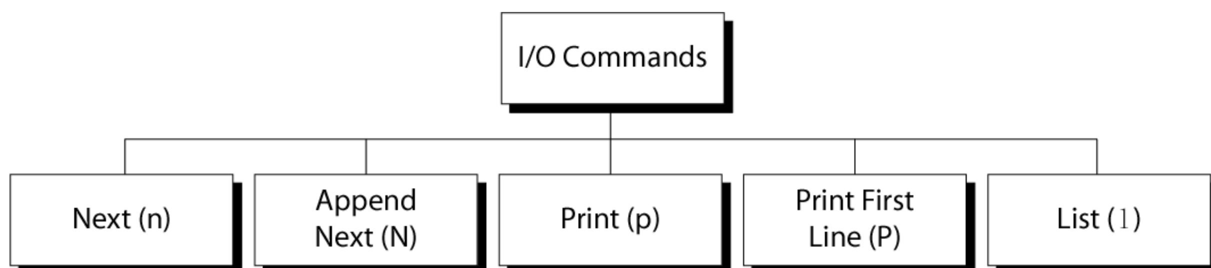
Syntax:

[addr1][,addr2]y/a/b/

- translates one character 'a' to another 'b'
- cannot use regular expression metacharacters
- cannot indicate a range of characters
- similar to “tr” command

Example:

$ sed -e ‘1,10y/abcd/wxyz/’ datafile
sed i/o commands



Input (next) Command: n and N
- Forces sed to read the next input line
    - Copies the contents of the pattern space to output
    - Deletes the current line in the pattern space
    - Refills it with the next input line
    - Continue processing

- N (uppercase) Command
  - adds the next input line to the current contents of the pattern space
  - useful when applying patterns to two or more lines at the same time

Output Command: p and P

- Print Command (p)
  - copies the entire contents of the pattern space to output
  - will print same line twice unless the option "–n" is used
- Print command: P
  - prints only the first line of the pattern space
  - prints the contents of the pattern space up to and including a new line character
  - any text following the first new line is not printed

List Command (l)

- The list command: l
  - shows special characters (e.g. tab, etc)
- The octal dump command (od -c) can be used to produce similar result

Hold Space

- temporary storage area

  used to save the contents of the pattern space

- 4 commands that can be used to move text back and forth between the pattern space and the hold space:

  h, H
  g, G

File commands

- allows to read and write from/to file while processing standard input
- read: r command
- write: w command

Read File command

Syntax: **r filename**

  - queue the contents of filename to be read and inserted into the output stream at

the end of the current cycle, or when the next input line is read

- if filename cannot be read, it is treated as if it were an empty file, without any error indication
- single address only

Write File command

Syntax: **w filename**

- Write the pattern space to filename
- The filename will be created (or truncated) before the first input line is read
- all w commands which refer to the same filename are output through the same FILE stream

Branch Command (b)

- Change the regular flow of the commands in the script file

Syntax: **[addr1][,addr2]b[label]**

- Branch (unconditionally) to 'label' or end of script
- If "label" is supplied, execution resumes at the line following :label; otherwise, control passes to the end of the script
- Branch label

:mylabel

Example: The quit (q) Command

Syntax: **[addr]q**

- Quit (exit sed) when addr is encountered.

Example: Display the first 50 lines and quit

% sed -e '50q' datafile

Same as:

% sed -n -e '1,50p' datafile

% head -50 datafile

Awk

What is awk?

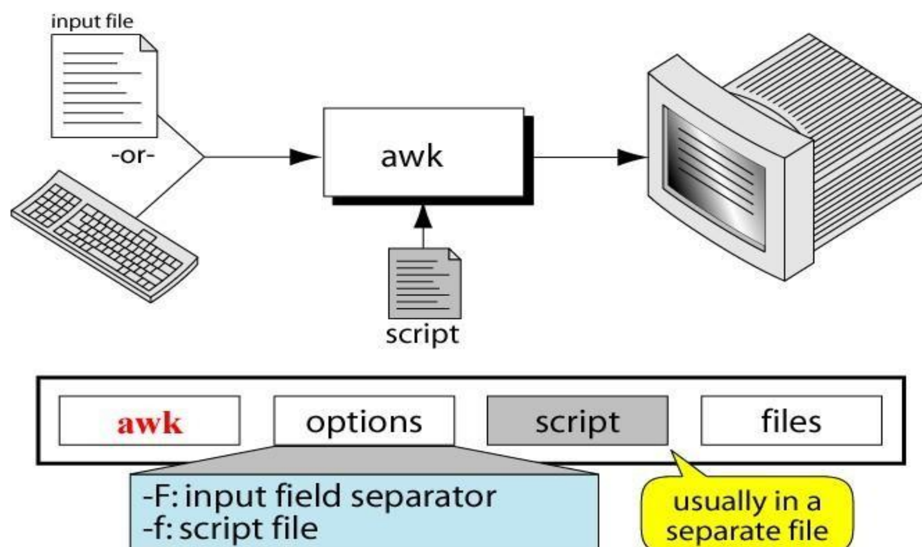
- created by: Aho, Weinberger, and Kernighan
- scripting language used for manipulating data and generating reports
- versions of awk

- awk, nawk, mawk, pgawk, …
- GNU awk: gawk

What can you do with awk?

- awk operation:
  - scans a file line by line
  - splits each input line into fields
  - compares input line/fields to pattern
  - performs action(s) on matched lines
- Useful for:
  - transform data files
  - produce formatted reports
- Programming constructs:
  - format output lines
  - arithmetic and string operations
  - conditionals and loops

The Command: awk



Basic awk Syntax

- **awk [options] ‘script’ file(s)**
- **awk [options] –f scriptfile file(s)**

<u>Options:</u>

-F to change input field separator

-f to name script file

Basic awk Program

- consists of patterns & actions:

  pattern {action}

  - if pattern is missing, action is applied to all lines
  - if action is missing, the matched line is printed
  - must have either pattern or action

<u>Example:</u>

awk '/for/' testfile

- prints all lines containing string “for” in testfile

Basic Terminology: input file

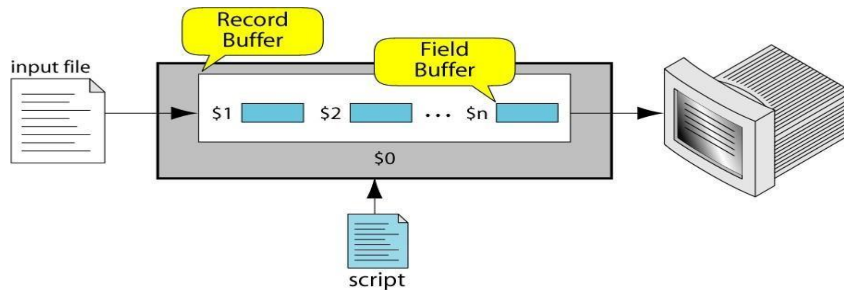
- A <u>field</u> is a unit of data in a line
- Each field is separated from the other fields by the <u>field separator</u>
  - default field separator is whitespace
- A <u>record</u> is the collection of fields in a line
- A data file is made up of records

Example Input File

	Field 1 (First_Name)	Field 2 (Last_Name)	Field 3 (Pay_Rate)	Field 4 (Hours)
	Susan	White	6.00	23
Record 2	Mark	Eagle	6.25	40
	Tuan	Nguyen	7.89	44
Record 4	Dan	Black	7.23	40
	Amanda	Trapp	6.95	40
	Brian	Devaux	7.95	0
	Chris	Walljasper	6.89	32
	Mary	Lamb	8.22	40
	Jackie	Kammaoto	7.59	40
Record 10	Nicky	Barber	6.35	40

A file with 10 records, each with four fields

Buffers



- awk supports two types of buffers:

  record and field

- field buffer:
  - one for each fields in the current record.
  - names: $1, $2, …
- record buffer :
  - $0 holds the entire record

Some System Variables

FS	Field separator (default=whitespace)
RS	Record separator (default=\n)
NF	Number of fields in current record
NR	Number of the current record
OFS	Output field separator (default=space)
ORS	Output record separator (default=\n)
FILENAME	Current filename

Example: Records and Fields

% cat emps

Tom Jones 4424 5/12/66 543354

Mary Adams 5346 11/4/63 28765

Sally Chang 1654 7/22/54 650000

Billy Black 1683 9/23/44 336500

% awk '{print NR, $0}' emps

1 Tom Jones 4424 5/12/66 543354

2 Mary Adams 5346 11/4/63 28765

3 Sally Chang 1654 7/22/54 650000

4 Billy Black 1683 9/23/44 336500

Example: Space as Field Separator

% cat emps

Tom Jones 4424 5/12/66 543354

Mary Adams 5346 11/4/63 28765

Sally Chang 1654 7/22/54 650000

Billy Black 1683 9/23/44 336500

% awk '{print NR, $1, $2, $5}' emps

1 Tom Jones 543354

2 Mary Adams 28765

3 Sally Chang 650000

4 Billy Black 336500

Example: Colon as Field Separator

% cat em2

Tom Jones:4424:5/12/66:543354

Mary Adams:5346:11/4/63:28765

Sally Chang:1654:7/22/54:650000

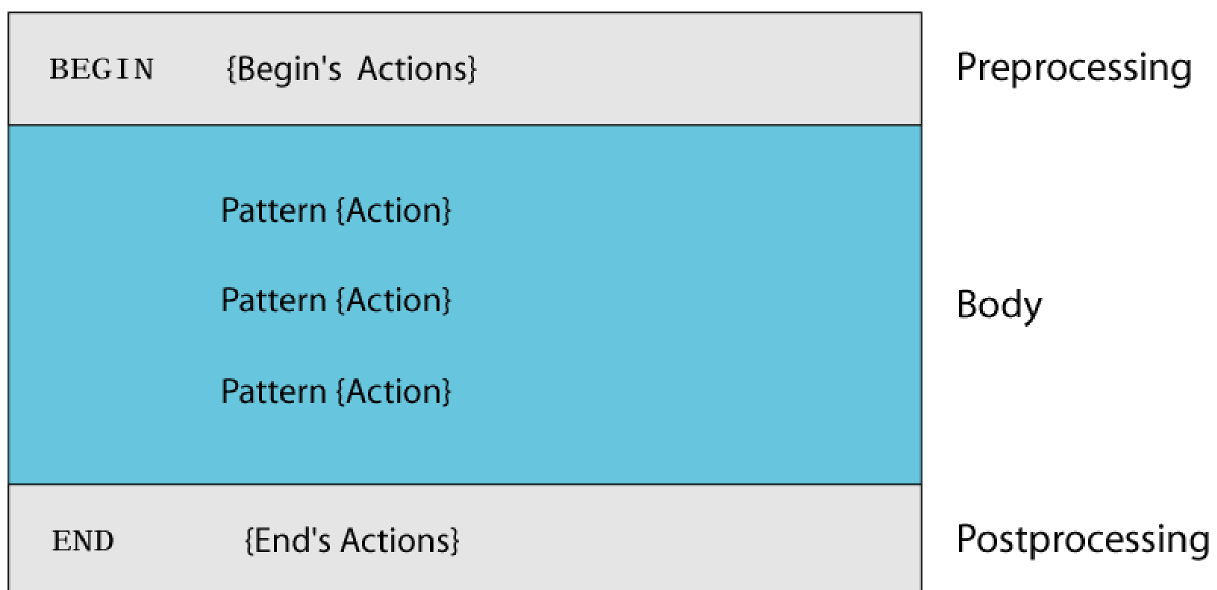
Billy Black:1683:9/23/44:336500

% awk -F: '/Jones/{print $1, $2}' em2

Tom Jones 4424

awk Scripts

- awk scripts are divided into three major parts:



- comment lines start with #
- BEGIN: pre-processing
    - performs processing that must be completed before the file processing starts (i.e., before awk starts reading records from the input file)
    - useful for initialization tasks such as to initialize variables and to create report headings
- BODY: Processing
    - contains main processing logic to be applied to input records
    - like a loop that processes input data one record at a time:
        - if a file contains 100 records, the body will be executed 100 times, one for each record
- END: post-processing
    - contains logic to be executed after all input data have been processed
    - logic such as printing report grand total should be performed in this part of the script

Pattern / Action Syntax

```
pattern {statement}
```

(a) One Statement Action

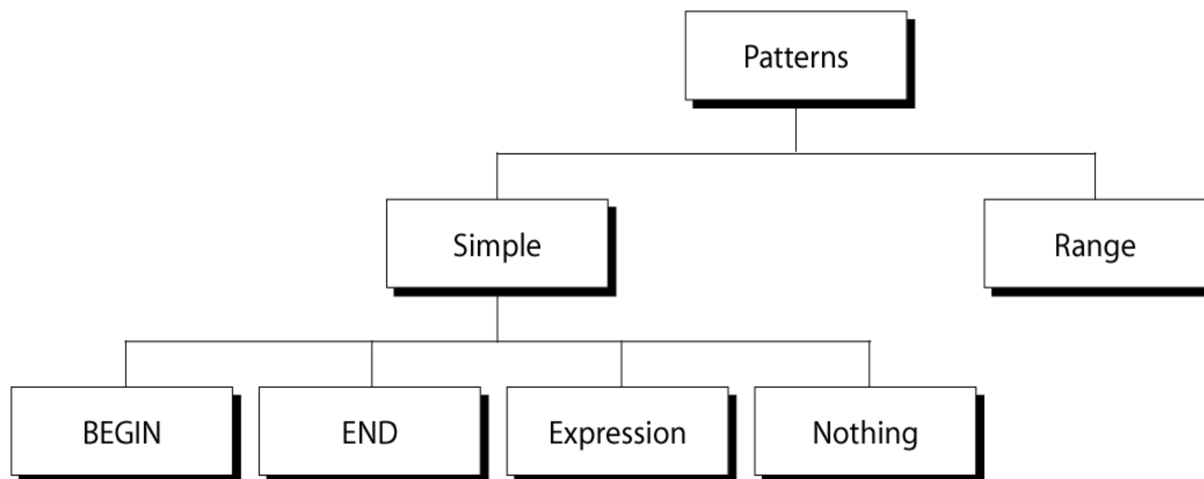
```
pattern {statement1; statement2; statement3}
```

(b) Multiple Statements Separated by Semicolons

```
pattern
{
    statement1
    statement2
    statement3
}
```

(c) Multiple Statements Separated by Newlines

Categories of Patterns



Expression Pattern types

- match
    - entire input record
  - regular expression enclosed by ‘/’s
    - explicit pattern-matching expressions
  - ~ (match), !~ (not match)
  - expression operators
    - arithmetic
    - relational
    - logical

% cat employees2

Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765

Sally Chang:1654:7/22/54:650000

Billy Black:1683:9/23/44:336500

% awk –F: '/00$/' employees2

Sally Chang:1654:7/22/54:650000

Billy Black:1683:9/23/44:336500

Example: explicit match

% cat datafile

northwest NW Charles Main 3.0 .98 3 34

western WE Sharon Gray 5.3 .97 5 23

southwest SW Lewis Dalsass 2.7 .8 2 18

southern SO Suan Chin 5.1 .95 4 15

southeast SE Patricia Hemenway 4.0 .7 4 17

eastern EA TB Savage 4.4 .84 5 20

northeast NE AM Main 5.1 .94 3 13

north NO Margot Weber 4.5 .89 5 9

central CT Ann Stephens 5.7 .94 5 13

% awk '$5 ~ /\.[7-9]+/' datafile

southwest SW Lewis Dalsass 2.7 .8 2 18

central CT Ann Stephens 5.7 .94 5 13

Examples: matching with REs

% awk '$2 !~ /E/{print $1, $2}' datafile

northwest NW

southwest SW

southern SO

north NO

central CT

% awk '/^[ns]/{print $1}' datafile

northwest

southwest

southern

southeast

northeast

north

Arithmetic Operators

Operator	Meaning	Example
+	Add	x + y
-	Subtract	x – y
*	Multiply	x * y
/	Divide	x / y
%	Modulus	x % y
^	Exponential	x ^ y

Example:

% awk '$3 * $4 > 500 {print $0}' file

Relational Operators

Operator	Meaning	Example
<	Less than	x < y
< =	Less than or equal	x < = y
==	Equal to	x == y

!=	Not equal to	x != y
>	Greater than	x > y
> =	Greater than or equal to	x > = y
~	Matched by reg exp	x ~ /y/
!~	Not matched by req exp	x !~ /y/

Logical Operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	NOT	! a

Examples:

% awk '($2 > 5) && ($2 <= 15) {print $0}' file
% awk '$3 == 100 || $4 > 50' file

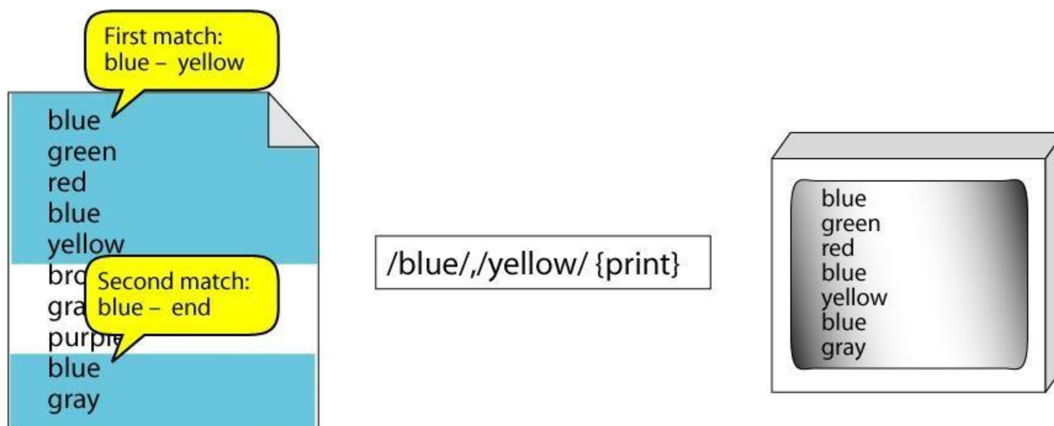
Range Patterns

- Matches ranges of consecutive input lines

Syntax:

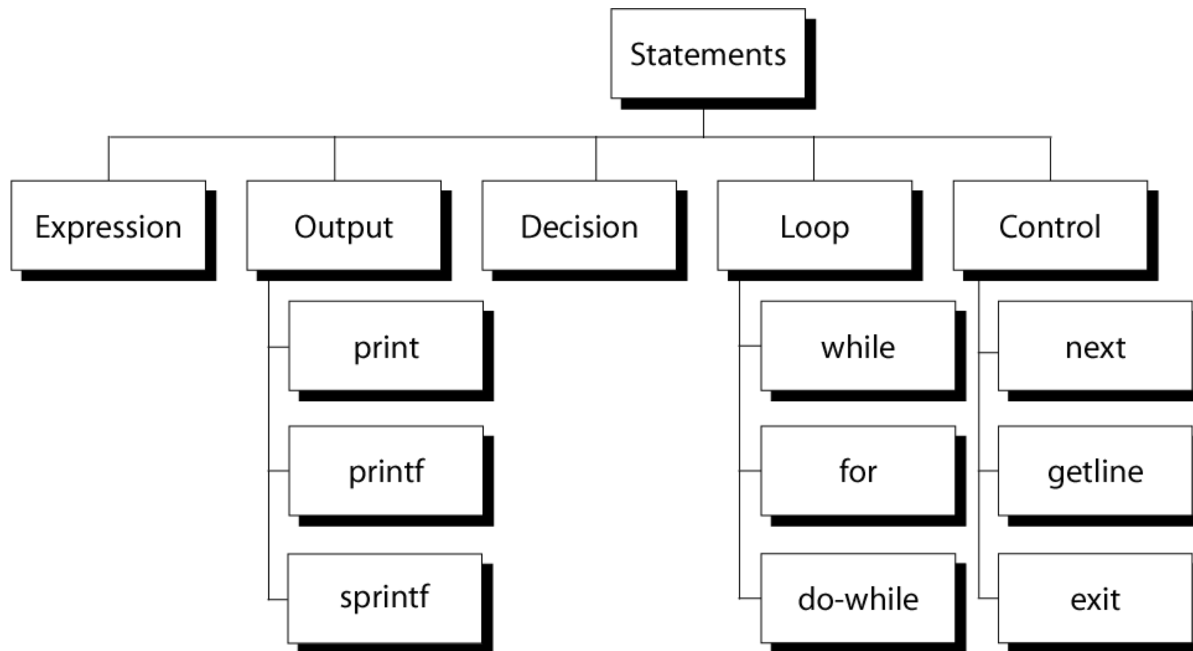
pattern1 , pattern2 {action}

- pattern can be any simple pattern
- **pattern1** turns action on
- **pattern2** turns action off



Range Pattern Example

awk Actions



awk expressions

- Expression is evaluated and returns value
    - consists of any combination of numeric and string constants, variables, operators, functions, and regular expressions
- Can involve variables
    - As part of expression evaluation
    - As target of assignment awk variables
- A user can define any number of variables within an awk script
- The variables can be numbers, strings, or arrays
- Variable names start with a letter, followed by letters, digits, and underscore
- Variables come into existence the first time they are referenced; therefore, they do not need to be declared before use
- All variables are initially created as strings and initialized to a null string "“”"

awk Variables

Format

variable = expression

Examples:

% awk '$1 ~ /Tom/
{wage = $3 * $4; print wage}' filename
% awk '$4 == "CA" {$4 = "California"; print $0}' filename

awk assignment operators

=	assign result of right-hand-side expression to left-hand-side variable
++	Add 1 to variable
--	Subtract 1 from variable
+=	Assign result of addition
-=	Assign result of subtraction
*=	Assign result of multiplication
/=	Assign result of division
%=	Assign result of modulo
^=	Assign result of exponentiation

Awk example:

File: grades

john 85 92 78 94 88

andrea 89 90 75 90 86

jasper 84 88 80 92 84

- awk script: average

```
# average five grades

{ total = $2 + $3 + $4 + $5 + $6

  avg = total / 5

  print $1, avg }
```

- Run as:

awk –f average grades

Output Statements

print

print easy and simple output

printf

print formatted (similar to C printf)

sprintf

format string (similar to C sprintf)

Function: print

- Writes to standard output
- Output is terminated by ORS
    - default ORS is newline
- If called with no parameter, it will print $0
- Printed parameters are separated by OFS,
    - default OFS is blank
- Print control characters are allowed:
    - \n \f \a \t \\ … print example

% awk '{print}' grades

john 85 92 78 94 88

andrea 89 90 75 90 86

% awk '{print $0}' grades

john 85 92 78 94 88

andrea 89 90 75 90 86

% awk '{print($0)}' grades

john 85 92 78 94 88

andrea 89 90 75 90 86

Redirecting print output

- Print output goes to standard output

unless redirected via:

> “file”
>> “file”
| “command”

- will open file or command only once
- subsequent redirections append to already open stream

print Example

% awk '{print $1 , $2 > "file"}' grades

% cat file

john 85

andrea 89

jasper 84

% awk '{print $1,$2 | "sort"}' grades

andrea 89

jasper 84

john 85

% awk '{print $1,$2 | "sort –k 2"}' grades

jasper 84

john 85

andrea 89

% date

Wed Nov 19 14:40:07 CST 2008

% date |

awk '{print "Month: " $2 "\nYear: ", $6}'

Mo0nth: Nov

Year: 2008

printf: Formatting output

<u>Syntax:</u>

printf(format-string, var1, var2, …)

- works like C printf
- each format specifier in "format-string" requires argument of matching type

Format specifiers

%d %i decimal integer

%c single character

%s string of characters

%f floating point number

%o octal number

%x hexadecimal number

%e scientific floating point notation

%% the letter "%"

Format specifier examples

Format specifier modifiers

- between "%" and letter

%10s

%7d

%10.4f

%-20s

- meaning:
    - width of field, field is printed right justified
    - precision: number of digits after decimal point
    - "-" will left justify sprintf: Formatting text

<u>Syntax:</u>

sprintf(format-string, var1, var2, …)

- Works like printf, but does not produce output
- Instead it returns formatted string

<u>Example:</u>

```
{
    text = sprintf("1: %d – 2: %d", $1, $2)

    print text

}
```

awk Array

- awk allows one-dimensional arrays

  to store strings or numbers

- index can be number or string
- array need not be declared
    - its size
    - its elements

- array elements are created when first used
    - initialized to 0 or “”

Arrays in awk

Syntax:

arrayName[index] = value

Examples:

list[1] = "one"

list[2] = "three"

list["other"] = "oh my !"

Illustration: Associative Arrays

- awk arrays can use string as index

Name	Age
"Robert"	46
"George"	22
"Juan"	22
"Nhan"	19
"Jonie"	34
Index	Data

Department	Sales
"19-24"	1,285.72
"81-70"	10,240.32
"41-10"	3,420.42
"17-A1"	46,500.18
"61-61"	1,114.41
Index	Data

Awk builtin split functions

split(string, array, fieldsep)

- divides string into pieces separated by fieldsep, and stores the pieces in array
- if the fieldsep is omitted, the value of FS is used.

Example:

split("auto-da-fe", a, "-")

- sets the contents of the array a as follows:

  a[1] = "auto"

  a[2] = "da"

  a[3] = "fe"

Example: process sales data

- input file:

Sales

1	clothing	3141
1	computers	9161
1	textbooks	21312
2	clothing	3252
2	computers	12321
2	supplies	2242
2	textbooks	15462

  - output:

- summary of category sales Illustration: process each input line

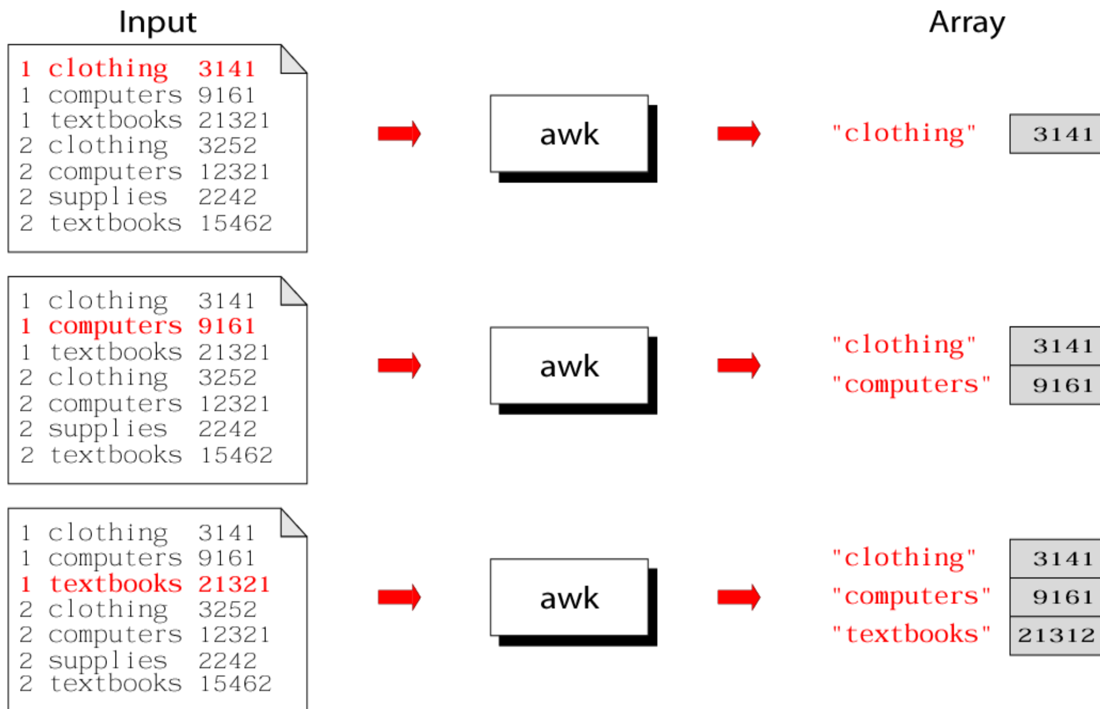
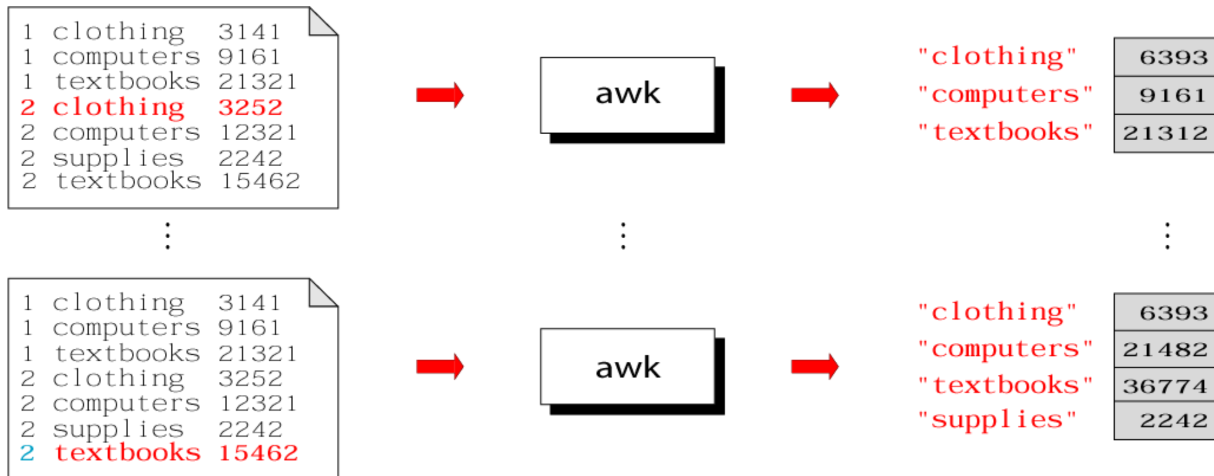
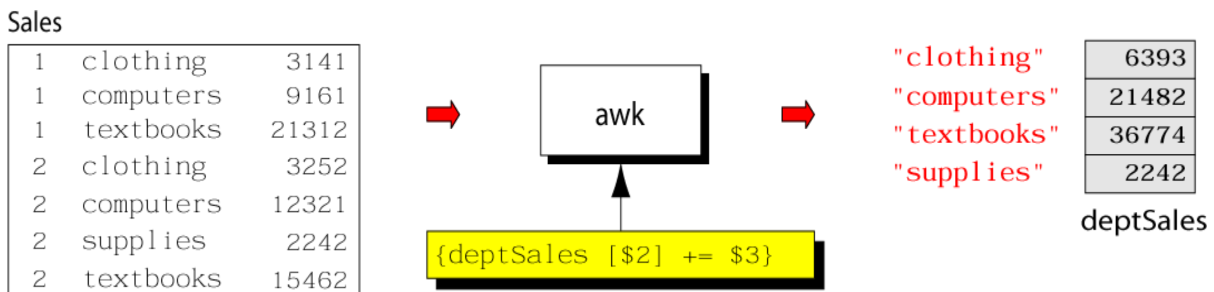


Illustration: process each input line



Summary: awk program



Example: complete program

```
% cat sales.awk
{
      deptSales[$2] += $3
}
END {
   for (x in deptSales)
            print x, deptSales[x]
}
% awk –f sales.awk sales
```

awk builtin functions

tolower(string)

- returns a copy of string, with each upper-case character converted to lower-case. Nonalphabetic characters are left unchanged.

  Example: tolower("MiXeD cAsE 123")

  returns "mixed case 123"

toupper(string)

- returns a copy of string, with each lower-case character converted to upper-case.

awk Example: list of products

```
103:sway bar:49.99

101:propeller:104.99

104:fishing line:0.99

113:premium fish bait:1.00

106:cup holder:2.49

107:cooler:14.89
```

112:boat cover:120.00

109:transom:199.00

110:pulley:9.88

105:mirror:4.99

108:wheel:49.99

111:lock:31.00

102:trailer hitch:97.95

awk Example: output

Marine Parts R Us

Main catalog

Part-id name price

======================================

101 propeller 104.99

102 trailer hitch 97.95

103 sway bar 49.99

104 fishing line 0.99

105 mirror 4.99

106 cup holder 2.49

107 cooler 14.89

108 wheel 49.99

109 transom 199.00

110 pulley 9.88

111 lock 31.00

112 boat cover 120.00

113 premium fish bait 1.00

=====================================

Catalog has 13 parts

awk Example: complete

```
BEGIN {
    FS= ":"
    print "Marine Parts R Us"
    print "Main catalog"
    print "Part-id\tname\t\t\t price"
    print "======================================"
}
{
    printf("%3d\t%-20s\t%6.2f\n", $1, $2, $3)
    count++
}
END {
    print "======================================"
    print "Catalog has " count " parts"
```

}

Applications:

Awk control structures

- Conditional
  - if-else
- Repetition
  - for
    - with counter
    - with array index
  - while
  - do-while
  - also: break, continue

if Statement

Syntax:

```
if (conditional expression)
        statement-1
else
  statement-2
```

Example:

```
if ( NR < 3 )
        print $2
else
        print  $3
```

for Loop

Syntax:

for (initialization; limit-test; update)

statement

Example:

```
for (i = 1; i <= NR; i++)
{
    total += $i
    count++
}
```

for Loop for arrays

Syntax:

for (var in array)

statement

Example:

```
for (x in deptSales)
{
    print x, deptSales[x]
}
```

While Loop

Syntax:

while (logical expression)

statement

Example:

```
i = 1
while (i <= NF)
{
    print i, $i
    i++
}
```

do-while Loop

Syntax:

```
    do
        statement
    while (condition)
```

- statement is executed at least once, even if condition is false at the beginning

Example:

```
 i = 1
 do {
   print $0
   i++
 } while (i <= 10)
```

loop control statements

- **break**

exits loop

- **continue**

skips rest of current iteration, continues with next iteration

Shell Programming

The shell has similarities to the DOS command processor Command.com (actually Dos was design as a poor copy of UNIX shell), it's actually much more powerful, really a programming language in its own right.

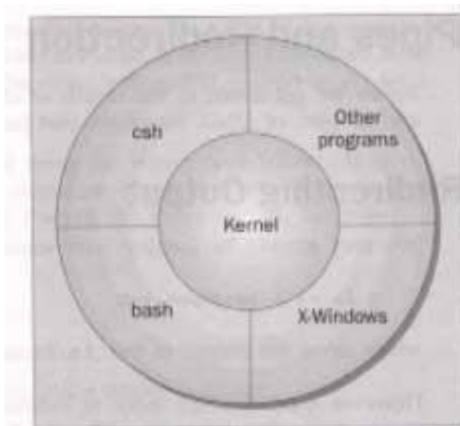
A shell is always available on even the most basic UNIX installation. You have to go through the shell to get other programs to run. You can write programs using the shell. You use the shell to administrate your UNIX system. For example:

```
ls -al | more
```

is a short shell program to get a long listing of the present directory and route the output through the more command.

What is a Shell?

A **shell** is a program that acts as the interface between you and the UNIX system, allowing you to enter commands for the operating system to execute.



Here are some common shells.

Shell Name	A Bit of History
sh (Bourne)	The original shell.
csh, tcsh and zsh	The C shell, created by Bill Joy of Berkeley UNIX fame. Probably the second most popular shell after **bash**.
ksh, pdksh	The Korn shell and its public domain cousin. Written by David Korn.
bash	The Linux staple, from the GNU project. **bash**, or Bourne Again Shell, has the advantage that the source code is available and even if it's not currently running on your UNIX system, it has probably been ported to it.
rc	More C than **csh**. Also from the GNU project.

Introduction- Working with Bourne Shell

- The Bourne shell, or sh, was the default Unix shell of Unix Version 7. It was developed by Stephen Bourne, of AT&T Bell Laboratories.
- A Unix shell, also called "the command line", provides the traditional user interface for the Unix operating system and for Unix-like systems. Users direct the operation of the computer by entering command input as text for a shell to execute.
- There are many different shells in use. They are
  - Bourne shell (sh)
  - C shell (csh)
  - Korn shell (ksh)

Bourne Again shell (bash)

- When we issue a command the shell is the first agency to acquire the information. It accepts and interprets user requests. The shell examines &rebuilds the commands &leaves the execution work to kernel. The kernel handles the h/w on behalf ofthese commands &all processes in the system.
- The shell is generally sleeping. It wakes up when an input is keyed in at the prompt. This input is actually input to the program that represents the shell.
-

Shell responsibilities

1. Program Execution
2. Variable and Filename Substitution
3. I/O Redirection
4. Pipeline Hookup
5. Environment Control
6. Interpreted Programming Language

1.Program Execution:

- The shell is responsible for the execution of all programs that you request from your terminal.
- Each time you type in a line to the shell, the shell analyzes the line and then determines what to do.

- The line that is typed to the shell is known more formally as the command line. The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program.

2. Variable and Filename Substitution:

- Like any other programming language, the shell lets you assign values to variables. Whenever you specify one of these variables on the command line, preceded by a dollar sign, the shell substitutes the value assigned to the variable at that point.

3. I/O Redirection:

- It is the shell's responsibility to take care of input and output redirection on the command line. It scans the command line for the occurrence of the special redirection characters <, >, or >>.

4. Pipeline Hookup:

- Just as the shell scans the command line looking for redirection characters, it also looks for the pipe character |. For each such character that it finds, it connects the standard output from the command preceding the | to the standard input of the one following the |. It then initiates execution of both programs.

5. Environment Control:

- The shell provides certain commands that let you customize your environment. Your environment includes home directory, the characters that the shell displays to prompt you

to type in a command, and a list of the directories to be searched whenever you request that a program be executed.

6. Interpreted Programming Language:

- The shell has its own built-in programming language. This language is interpreted, meaning that the shell analyzes each statement in the language one line at a time and then executes it. This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine-executable form before they are executed.
- Programs developed in interpreted programming languages are typically easier to debug and modify than compiled ones. However, they usually take much longer to execute than their compiled equivalents.

Pipes and Redirection

Pipes connect processes together. The input and output of UNIX programs can be redirected.

Redirecting Output

The > operator is used to redirect output of a program. For example:

```
ls -l > lsoutput.txt
```

redirects the output of the list command from the screen to the file lsoutput.txt.

To 0append to a file, use the >> operator.

```
ps >> lsoutput.txt
```

Redirecting Input

You redirect input by using the < operator. For example:

```
more < killout.txt
```

Pipes

We can connect processes together using the pipe operator (|). For example, the following program means run the ps program, sort its output, and save it in the file pssort.out

```
ps | sort > pssort.out
```

The sort command will sort the list of words in a textfile into alphbetical order according to the ASCII code set character order.

Here Documents

A here document is a special way of passing input to a command from a shell script. The document starts and ends with the same leader after <<. For example:

```
#!/bin/sh

cat < this is a here
document
!FUNKY!
```

How It Works

It executes the here document as if it were input commands.

Running a Shell Script

You can type in a sequence of commands and allow the shell to execute them interactively, or youu can sotre these commands in a file which you can invoke as a program.

Interactive Programs

A quick way of trying out small code fragments is to just type in the shell script on the command line. Here is a shell program to compile only files that contain the string POSIX.

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

The Shell as a Programming Language

Creating a Script

To create a **shell script** first use a text editor to create a file containing the commands. For example, type the following commands and save them as first.sh

```
#!/bin/sh

# first.sh
# This file looks through all the files in the current
# directory for the string POSIX, and then prints those
# files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    more $file
  fi
done

exit 0
```

Note: commands start with a #.

The line

#!/bin/sh

is special and tells the system to use the /bin/sh program to execute this program.

The command

exit 0

Causes the script program to exit and return a value of 0, which means there were not errors.

Making a Script Executable

There are two ways to execute the script. 1) invoke the shell with the name of the script file as a parameter, thus:

/bin/sh first.sh

Or 2) change the mode of the script to executable and then after execute it by just typing its name.

chmod +x first.sh
first.sh

Actually, you may need to type:

./first.sh

to make the file execute unles the path variable has your directory in it.

Shell Syntax

The modern UNIX shell can be used to write quite large, structured programs.

Shell metacharacters

The shell consists of large no. of metacharacters. These characters plays vital role in Unix programming.

<u>Types of metacharacters:</u>

1.File substitution

2.I/O redirection

3.Process execution

4. Quoting metacharacters

5.Positional parameters

6.Special characters

7.Command substitution

Filename substitution:

These metacharacters are used to match the filenames in a directory.

Metacharacter significance

Metacharacter	significance
*	matches any no. of characters
?	matches a single character
[ijk]	matches a single character either i,j,k
[!ijk]	matches a single character that is not an I,j,k

Shell Variables

Variables are generally created when you first use them. By default, all variables are considered and stored as strings. Variable names are case sensitive.

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

- U can define & use variables both in the command line and shell scripts. These variables are called shell variables.
- No type declaration is necessary before u can use a shell variable.
- Variables provide the ability to store and manipulate the information with in the shell program. The variables are completely under the control of user.

- Variables in Unix are of two types.

1) **User-defined variables:**

   Generalized form:

   variable=value.

   Eg: $x=10

   $echo $x

   10

- To remove a variable use unset.
  - $unset x
- All shell variables are initialized to null strings by default. To explicitly set null values use
  - x= or x=‘’ or x=“”
- To assign multiword strings to a variable use
  - $msg=‘u have a mail’

2) **Environment Variables**

   - They are initialized when the shell script starts and normally capitalized to distinguish them from user-defined variables in scripts
   - To display all variables in the local shell and their values, type the **set** command
   - The **unset** command removes the variable from the current shell and sub shell

Environment Variables	Description
$HOME	**Home directory**
$PATH	**List of directories to search for commands**
$PS1	**Command prompt**
$PS2	**Secondary prompt**
$SHELL	**Current login shell**
$0	**Name of the shell script**
$#	**No . of parameters passed**

$$	Process ID of the shell script

Command substitution and Shell commands:

read:

- The read statement is a tool for taking input from the user i.e. making scripts interactive. It is used with one or more variables. Input supplied through the standard input is read into these variables.

$read name

What ever u entered is stored in the variable

name. printf:

Printf is used to print formatted

o/p. printf "format" arg1 arg2 ...

Eg:

$ printf "This is a number: %d\n" 10
This is a number: 10

$

Printf supports conversion specification characters like %d, %s ,%x

,%o…. Exit status of a command:

- Every command returns a value after execution .This value is called the exit status or return value of a command.
- This value is said to be true if the command executes successfully and false if it fails.
- There is special parameter used by the shell it is the $?. It stores the exit status of a command.

exit:

- The exit statement is used to prematurely terminate a program. When this statement is encountered in a script, execution is halted and control is returned to the calling program- in most cases the shell.
- U don't need to place exit at the end of every shell script because the shell knows when script execution is complete.

set:

- Set is used to produce the list of currently defined variables.

$set

- Set is used to assign values to the positional parameters.

$set welcome to Unix

The do-nothing(:)Command

- It is a null command.
- In some older shell scripts, colon was used at the start of a line to introduce a comment, but modern scripts uses # now.
- expr:
- The expr command evaluates its arguments as an expression:

```
$ expr 8 + 6
$ x=`expr 12 / 4`
$ echo $x
3
```

export:

There is a way to make the value of a variable known to a sub shell, and that's by exporting it with the export command. The format of this command is

export variables

where variables is the list of variable names that you want exported. For any sub shells that get executed from that point on, the value of the exported variables will be passed down to the sub shell.

eval:

eval scans the command line twice before executing it. General form for eval

is eval command-line

Eg:

```
$ cat last

eval echo \$$#

$ last one two three four
four
```

${n}

If u supply more than nine arguments to a program, u cannot access the tenth and greater arguments with $10, $11, and so on.

${n} must be used. So to directly access argument 10, you must write

${10}

Shift command:

The shift command allows u to effectively left shift your positional parameters. If u execute the command

Shift

whatever was previously stored inside $2 will be assigned to $1, whatever was previously stored in $3 will be assigned to $2, and so on. The old value of $1 will be irretrievably lost.

The Environment-Environment Variables

It creates the variable salutation, displays its value, and some parameter variables.

- When a shell starts, some variables are initialized from values in the environment. Here is a sample of some of them.

Environment Variable	Description
$HOME	The home directory of the current user.
$PATH	A colon-separated list of directories to search for commands.
$PS1	A command prompt, usually $.
$PS2	A secondary prompt, used when prompting for additional input, usually >.
$IFS	An input field separator. A list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.

Environment Variable	Description
$0	The name of the shell script
$#	The number of parameters passed.
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames, for example /tmp/junk_$$.

Parameter Variables

- If your script is invoked with parameters, some additional variables are created.

Parameter Variable	Description
$1, $2, ...	The parameters given to the script.
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS.
$@	A subtle variation on $*, that doesn't use the IFS environment variable.

Quoting

Normally, parameters are separated by white space, such as a space. Single quot marks can be used to enclose values containing space(s). Type the following into a file called quot.sh

```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

make sure to make it executable by typing the command:

< chmod a+x
quot.sh The results of executing

the file is:

```
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello World
$myvar now equals Hello World
```

How It Works

The variable myvar is created and assigned the string Hi there. The content of the variable is displyed using the echo $. Double quotes don't effect echoing the value. Single quotes and backslash do.

The test, or []Command

Here is how to check for the existance of the file fred.c using the test and using the [] command.

```
if test -f fred.c
then
...
fi
```

We can also write it like this:

```
if [ -f fred.c ]
then
...
fi
```

You can even place the then on the same line as the if, if youu add a semicolon before the word then.

```
if [ -f fred.c ]; then
...
fi
```

Here are the conditon types that can be used with the test command. There are string comparison.

String Comparison	Result
`string`	True if the string is not an empty string.
`string1 = string2`	True if the strings are the same.
`string1 != string2`	True if the strings are not equal.
`-n string`	True if the string is not **null**.
`-z string`	True if the string is **null** (an empty string).

There are arithmetic comparison.

Arithmetic Comparison	Result
expression1 -eq expression2	True if the expressions are equal.
expression1 -ne expression2	True if the expressions are not equal.
expression1 -gt expression2	True if **expression1** is greater than **expression2**.
expression1 -ge expression2	True if **expression1** is greater than or equal to **expression2**.
expression1 -lt expression2	True if **expression1** is less than **expression2**.
expression1 -le expression2	True if **expression1** is less than or equal to **expression2**.
! expression	The ! negates the expression and returns **true** if the expression is **false**, and vice versa.

There are file conditions.

File Conditional	Result
-d file	True if the file is a directory.
-e file	True if the file exists.
-f file	True if the file is a regular file.
-g file	True if **set-group-id** is set on file.
-r file	True if the file is readable.
-s file	True if the file has non-zero size.
-u file	True if **set-user-id** is set on file.
-w file	True if the file is writeable.
-x file	True if the file is executable.

Control Structures

The shell has a set of control structures.

if

The if statement is vary similar other programming languages except it ends with a fi.

```
if condition
then
        statements
else
        statements
fi
```

elif

the elif is better known as "else if". It replaces the else part of an if statement with another if statement. You can try it out by using the following script.

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $ti0meofday = "yes" ]
then
        echo  "Good  morning"
elif [ $timeofday = "no" ]; then
        echo "Good afternoon"
else
echo "Sorry, $timeofday not recognized. Enter yes
or no" exit 1 fi

exit 0
```

How It Works

The above does a second test on the variable timeofday if it isn't equal to yes.

A Problem with Variables

If a variable is set to null, the statement

```
if [ $timeofday = "yes" ]
```

looks like

```
if [ = "yes" ]
```

which is illegal. This problem can be fixed by using double quotes around the variable name.

```
if [ "$timeofday" = "yes" ]
```

.

for

The for construct is used for looping through a range of values, which can be any set of strings. The syntax is:

```
for variable in values
do
        statements
done
```

Try out the following script:

```
#!/bin/sh

for foo in bar fud 43
do
        echo $foo
done
exit 0
```

When executed, the output should be:

```
bar
fud0
43
```

How It Works

The above example creates the variable foo and assigns it a different value each time around the for loop.

How It Works

Here is another script which uses the $(command) syntax to expand a list to chap3.txt, chap4.txt, and chap5.txt and print the files.

```
#!/bin/sh

for file in $(ls chap[345].txt); do
        lpr $file
done0
```

while

While loops will loop as long as some condition exist. OF course something in the body statements of the loop should eventually change the condition and cause the loop to exit. Here is the while loop syntax.

```
while condition do
        statements
done
```

Here is a whil loop that loops 20 times.

```
#!/bin/sh

foo=1

while [ "$foo" -le 20 ]
do
done exit 0
```

How It Works
echo "Here we go again" foo=$(($foo+1))

The above script uses the [ ] command to test foo for <= the value 20. The line

```
foo=$(($fo0o+1))
```

increments the value of foo each time the loop executes..

until

The until statement loops until a condition becomes true! Its syntax is:

```
until condition
do
        statements
done
```

Here is a script using until.

```
#!/bin/sh

until who | grep "$1" > /dev/null
do
        Sl0eep 60
done

# now ring the bell and announce the expected user.

echo -e \\a
echo "**** $1 has just loogged in ****"

exit 0
```

case

The case statement allows the testing of a variable for more then one value. The case statement ends with the word esac. Its syntax is:

```
case variable in
        pattern [ | pattern] ...) statements;;
        pattern [ | pattern] ...) statements;;
        ...
esac
```

Here is a sample script using a case statement:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
        "yes")  echo  "Good  Morning";;
        "no" ) echo "Good Afternoon";;
        0"y"  )  echo  "Good  Morning";;
        "n" ) echo "Good Afternoon";;
        * ) echo "Soory, answer not recognized";;
esac

exit 0
```

The value in the varaible timeofday is compared to various strings. When a match is made, the associated echo command is executed.

Here is a case where multiple strings are tested at a time, to do the some action.

```
case "$timeofday" in
        "yes" | "y" | "yes" | "YES" ) echo "good Morning";;
        "n"* | "N"* ) <echo "Good Afternoon";;
        * ) < echo "Sorry, answer not recognized";;
0esac
```

How It Works

The above has sever strings tested for each possible statement.

Here is a case statement that executes multiple statements for each case.

```
case "$timeofday" in
        "yes" | "y" | "Yes" | "YES" )
                echo "Good Morning"
                echo "Up bright and early this morning"
                ;;
        [nN]*)
                echo "Good Afternoon"
                ;;
        *)
                echo "Sorry, answer not recognized"
                echo  "Please  answer  yes  or  noo"
                exit 1
                ;;
esac
```

How It Works

When a match is found to the variable value of timeofday, all the statements up to the ;; are executed.

Arithmetic in shell

The $((...)) is a better alternative to the expr command, which allows simple arithmetic commands to be processed.

```
x=$(($x+1))
```

Parameter Expansion

Using { } around a variable to protect it against expansion.

```
#!/bin/sh

for i in 1 2
do
        my_secret_process ${i}_tmp
done
```

Here are some of the parameter expansion

Parameter Expansion	Description
${param:-default}	If param is null, set it to the value of default.
${#param}	Gives the length of param.
${param%word}	From the end, removes the smallest part of param that matches word and returns the rest.
${param%%word}	From the end, removes the longest part of param that matches word and returns the rest.
${param#word}	From the beginning, removes the smallest part of param that matches word and returns the rest.
${param##word}	From the beginning, removes the longest part of param that matches word and returns the rest.

How It Works

The try it out exercise uses parameter expansion to demonstrate how parameter expansion works.

Shell Script Examples

Example

```
#!/bin/sh

echo "Is it morning? (Answer yes or no)"

read timeofday

if [ $timeofday = "yes" ]; then
```

```
        echo "Good Morning"
else
        echo "Good afternoon"
fi
exit 0
```

elif - Doing further Checks

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if  [  $timeofday  =  "yes"  ];  then
        echo "Good Morning"
elif [ $timeofday = "no" ]; then
        echo "Good afternoon"
else    echo "Wrong answer! Enter yes or no"
        exit 1
fi      exit 0
```

Interrupt Processing-trap

The trap command is used for secifying the actions to take on receipt of signals. It syntax is:

trap command signal

Here are some of the signals.

Signal	Description
HUP (1)	Hang up; usually sent when a terminal goes off line, or a user logs out.
INT (2)	Interrupt; usually sent by pressing *Ctrl-C*.
QUIT (3)	Quit; usually sent by pressing *Ctrl-*.
ABRT (6)	Abort; usually sent on some serious execution error.
ALRM (14)	Alarm; usually used for handling time-outs.
TERM (15)	Terminate; usually sent by the system when it's shutting down.

How It Works

The try it out section has you type in a shell script to test the trap command. It creates a file and

keeps saying that it exists until youu cause a control-C interrupt. It does it all again.

Functions

You can define functions inthe shell. The syntax is:

```
function_name () {
        statements
}0
```

Here is a sample function and its execution.

```
#!/bin/sh

foo() {
        echo "Function foo is executing"
}
```

```
echo "script starting"
foo
echo "script ended"

exit 0
```

How It Works

When the above script runs, it defines the funcion foo, then script echos script starting, then it runs the functions foo which echos Function foo is executing, then it echo script ended.

Here is another sample script with a function in it. Save it as my_name

```
#!/bin/sh

yes_or_no() {
        echo "Parameters are $*"
        while true
        do
                echo -n "Enter yes or no"
                read x
                0case "$x" in
                        y | yes ) return 0;;
                        n | no ) return 1;;
 * ) echo "Answer yes or no"
                esac
        done
}

echo "Original parameters are $*"
if yes_or_no "IS your naem $1"
then
        echo "Hi $1"
else
        echo "Never mind"
fi

0exit 0
```

How It Works

When my_name is execute with the statement:

    my_name Rick and Neil

. gives the output of:

    Original parameters are Rick and Neil
    Parameters are Is your name Rick
    Enter yes or no
    no
    Never mind

Commands

You can execute normal command and built-in commands from a shell script. Built-in commands are defined and only run inside of the script.

break

It is used to escape from an enclosing for, while or until loop before the controlling condition has been met.

The : Command

The colon command is a null command. It can be used for an alias for true..

Continue

The continue command makes the enclosing for, while, or until loop continue at the next iteration.

The Command

The dot command executes the command in the current shell:

    . shell_script

echo

The echo command simply outputs a string to the standard output device followed by a newline character.

Eval

The eval command evaluates arguments and give s the results.

exec

The exec command can replace the current shell with a different program. It can also modify the current file descriptors.

exit n

The exit command causes the script to exit with exit code n. An exit code of 0 means success. Here are some other codes.

Exit Code	Description
126	The file was not executable.
127	A command was not found.
128 and above	A signal occurred.

export

The export command makes the variable named as its parameter available in subshells.

expr

The expr command evaluates its arguments as an expression.

0x = `expr $x + 1`

Here are some of its expression evaluations

Expression Evaluation	Description
expr1 | expr2	expr1 if expr1 is non-zero, otherwise expr2.
expr1 & expr2	Zero if either expression is zero, otherwise expr1.
expr1 = expr2	Equal.
expr1 > expr2	Greater than.
expr1 >= expr2	Greater or equal to.
expr1 < expr2	Less than.
expr1 <= expr2	Less or equal to.
expr1 != expr2	Not equal.
expr1 + expr2	Addition.
expr1 - expr2	Subtraction.
expr1 * expr2	Multiplication.
expr1 / expr2	Integer division.
expr1 % expr2	Integer modulo.

printf

The printf command is only available in more recent shells. It works similar to the echo command. Its general form is:

printf "format string" parameter1 parameter2 ...

Here are some characters and format specifiers.

Escape Sequence	Description
\\	Backslash character
\a	Alert (ring the bell or beep)
\b	Backspace character
\f	Form feed character
\n	Newline character
\r	Carriage return
\t	Tab character
\v	Vertical tab character
\ooo	The single character with octal value ooo

Conversion Specifier	Description
d	Output a decimal number
c	Output a character
s	Output a string
%	Output the % character

return

The return command causes functions to return. It can have a value parameter which it returns.

set

The set command sets the parameter variables for the shell.

shift

The shift command moves all the parameters variables down by one, so $2 becomes $1, $3 becomes $2, and so on.

unset

The unset command removes variables or functions from the environment.

Command Execution

The result of $(command) is simply the output string from the command, which is then available to the script.

Debugging Shell Scripts

When an error occurs in a script, the shell prints out the line number with an error. You can use the set command to set various shell option. Here are some of them.

Command Line Option	set Option	Description
sh -n <script>	set -o noexec set -n	Checks for syntax errors only; doesn't execute commands.
sh -v <script>	set -o verbose set -v	Echoes commands before running them.
sh -x <script>	set -o xtrace set -x	Echoes commands after processing on the command line.
	set -o nounset set -u	Gives an error message when an undefined variable is used.

Unit II

Files and Directories

UNIX File Structure

In UNIX, everything is a file.

Programs can use disk files, serial ports, printers and other devices in the exactly the same way as they would use a file.

Directories, too, are special sorts of files.

File types

Most files on a UNIX system are regular files or directories, but there are additional types of files:

1. **Regular files**: The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary.
2. **Directory file:** A file contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write to a directory file.
3. **Character special file**: A type of file used for certain types of devices on a system.
4. **Block special file**: A type of file typically used for disk devices. All devices on a system are either character special files or block special files.
5. **FIFO**: A type of file used for interprocess communication between processes. It's sometimes called a named pipe.
6. **Socket:** A type of file used for network communication between processes. A socket can also be used for nonnetwork communication between processes on a single host.
7. **Symbolic link:** A type of file that points to another file.

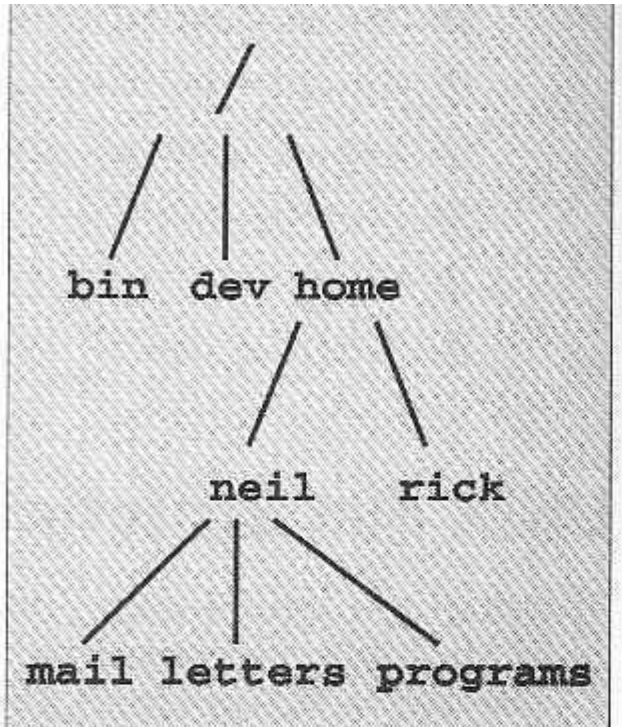
The argument to each of different file types is defined as follows_

Macro	Type of file
S_ISREG()	Regular file
S_ISDIR()	Directory file
S_ISCHR()	Character special file
S_ISBLK()	Block special file
S_ISFIFO()	Pipe or FIFO
S_ISLNK()	Symbolic link
S_ISSOCK()	Socket

File System Structure

Files are arranged in directories, which also contain subdirectories.

A user, **neil**, usually has his files stores in a 'home' directory, perhaps **/home/neil**.



Files and Devices

Even hardware devices are represented (mapped) by files in UNIX. For example, as **root**, you mount a CD-ROM drive as a file,

```
$ mount -t iso9660 /dev/hdc /mnt/cd_rom
$ cd /mnt/cd_rom
```

/dev/console - this device represents the system console.
/dev/tty - This special file is an alias (logical device) for controlling terminal (keyboard and screen, or window) of a process.
/dev/null - This is the null device. All output written to this device is discarded.

File Metadata Inodes

- A structure that is maintained in a separate area of the hard disk.
- File attributes are stored in the inode.
- Every file is associated with a table called the inode.
- The inode is accessed by the inode number.
- Inode contains the following attributes of a file: file type, file permissions , no. of links UID of the owner, GID of the group owner, file size date and time of last modification, last access, change.

File attributes

Attribute	**value meaning**
File type	type of the file
Access permission	file access permission for owner, group and others
Hard link count	no.of hard links of a file.
UID	file owner user ID.
GID	the file group ID.
File size	file size in bytes.
Inode number	system inode number of the file.
File system ID	file system ID where the file is stored.

Kernel Support For Files:

UNIX supports the sharing of open files between different processes. Kernel has three data structures are used and the relationship among them determines the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which is taken as a vector, with one entry per descriptor. Associated with each file descriptor are
    a. The file descriptor flags.
    b. A pointer to a file table entry.
2. The kernel maintains a file table for all open files. Each file table entry contains
    a. The file status flags for the file(read, write, append, sync, nonblocking, etc.),
    b. The current file offset,
    c. A pointer to the v-node table entry for the file.
3. Each open file (or device) has a v-node structure. The v-node contains information about the type of file and pointers to functions that operate on the file. For most files the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available.

The arrangement of these three tables for a single process that has two different files open one file is open on standard input (file descriptor 0) and the other is open standard output (file descriptor 1).

Here, the first process has the file open descriptor 3 and the second process has file open descriptor 4. Each process that opens the file gets its own file table entry, but only a single v-node table entry. One reason each process gets its own file table entry is so that each process has its own current offset for the file.

- After each 'write' is complete, the current file offset in the file table entry is incremented by the number of bytes written. If this causes the current file offset to exceed the current file size, the current file size, in the i-node table the entry is to the current file offset(Ex: file is extended).
- If a file is opened with O_APPEND flag, a corresponding flag is set in the file status flags of the file table entry. Each time a 'write' is performed for a file with this append flag

set, the current file offset in the file table entry is first set to the current file size from the i-node table entry. This forces every 'write' to be appended to the current end of file.

- The 'lseek' function only modifies the current offset in the file table entry. No I/O table place.
- If a file is positioned to its current end of file using lseek, all that happens is the current file offset in the file table entry is set to the current file size from the i-node table entry.

It is possible for more than a descriptor entry to point to the same file table only. The file descriptor flag is linked with a single descriptor in a single process, while file status flags are descriptors in any process that point to given file table entry.

System Calls and Device Drivers

System calls are provided by UNIX to access and control files and devices.

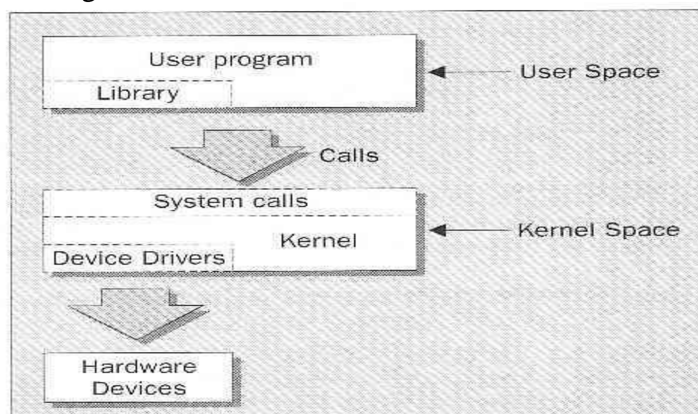
A number of **device drivers** are part of the kernel.

The system calls to access the device drivers include:

`open`	Open a file or device.
`read`	Read from an open file or device.
`write`	Write to a file or device.
`close`	Close the file or device.
`ioctl`	Specific control the device.

Library Functions

To provide a higher level interface to device and disk files, UNIIX provides a number of standard libraries.



Low-level File Access

Each running program, called a **process**, has associated with it a number of file descriptors.

When a program starts, it usually has three of these descriptors already opened. These are:

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

```
#include <unistd.h>

size_t write(int fildes, const void *buf, size_t nbytes);
```

The **write** system call arranges for the first **nbytes** bytes from **buf** to be written to the file associated with the file descriptor **fildes**.

With this knowledge, let's write our first program, **simple_write.c**:

```
#include <unistd.h>

int main()
{
    if ((write(1, "Here is some data\n", 18)) != 18)
        write(2, "A write error has occurred on file descriptor 1\n",46);

    exit(0);
}
```

Here is how to run the program and its output.

```
$    simple_write
Here is some data
$
```

read

```
#include <unistd.h>

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

The **read** system call reads up to **nbytes** of data from the file associated with the file decriptor **fildes** and places them in the data area **buf**.

This program, **simple_read.c**, copies the first 128 bytes of the standard input to the standard output.

```
#include <unistd.h>

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);

    exit(0);
}
```

If you run the program, you should see:

```
$ echo hello there | simple_read
hello there
$ simple_read < draft1.txt
Files
```

open

To create a new file descriptor we need to use the **open** system call.

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

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

open establishes an access path to a file or device.

The name of the file or device to be opened is passed as a parameter, **path**, and the **oflags** parameter is used to specify actions to be taken on opening the file.

The **oflags** are specified as a bitwise OR of a mandatory file access mode and other optional modes. The **open** call must specify one of the following file access modes:

Mode	Description
O_RDONLY	Open for read-only
O_WRONLY	Open for write-only
O_RDWR	Open for reading and writing

The call may also include a combination (bitwise OR) of the following optional modes in the **oflags** parameter:

- O_APPEND Place written data at the end of the file.
- O_TRUNC Set the length of the file to zero, discarding existing contents.
- O_CREAT Creates the file, if necessary, with permissions given in mode.
- O_EXCL Used with O_CREAT, ensures that the caller creates the file. The open is atomic, i.e. it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

Initial Permissions

When we create a file using the **O_CREAT** flag with **open**, we must use the three parameter form. **mode**, the third parameter, is made form a bitwise OR of the flags defined in the header file **sys/stat.h**. These are:

- S_IRUSR Read permission, owner.
- S_IWUSR Write permission, owner.
- S_IXUSR Execute permission, owner.
- S_IRGRP Read permission, group.
- S_IWGRP Write permission, group.
- S_IXGRP Execute permission, group.
- S_IROTH Read permission, others.
- S_IWOTH Write permission, others.
- S_IXOTH Execute permission, others.

For example

```
open ("myfile", O_CREAT, S_IRUSR|S_IXOTH);
```

Has the effect of creating a file called **myfile**, with read permission for the owner and execute permission for others, and only those permissions.

```
$ ls -ls myfile
0 -r-------x   1 neil     software          0 Sep 22 08:11 myfile*
```

umask

The **umask** is a system variable that encodes a mask for file permissions to be used when a file is created.

You can change the variable by executing the **umask** command to supply a new value.

The value is a three-digit octal value. Each digit is the results of ANDing values from 1, 2, or 4.

Digit	Value	Meaning
1	0	No user permissions are to be disallowed.
	4	User read permission is disallowed.
	2	User write permission is disallowed.
	1	User execute permission is disallowed.

Digit	Value	Meaning
2	0	No group permissions are to be disallowed.
	4	Group read permission is disallowed.
	2	Group write permission is disallowed.
	1	Group execute permission is disallowed.
3	0	No other permissions are to be disallowed.
	4	Other read permission is disallowed.
	2	Other write permission is disallowed.
	1	Other execute permission is disallowed.

For example, to block 'group' write and execute, and 'other' write, the **umask** would be:

Digit	Value
1	0
2	2
	1
3	2

Values for each digit are ANDed together; so digit 2 will have 2 & 1, giving 3. The resulting **umask** is **032**.

close

```
#include <unistd.h>

int close(int fildes);
```

We use **close** to terminate the association between a file descriptor, **fildes**, and its file.

ioctl

```
#include <unistd.h>

int ioctl(int fildes, int cmd, ...);
```

ioctl is a bit of a rag-bag of things. It provides an interface for controlling the behavior of devices, their descriptors and configuring underlying services.

ioctl performs the function indicated by **cmd** on the object referenced by the descriptor **fildes**.

Try It Out - A File Copy Program

We now know enough about the **open**, **read** and **write** system calls to write a low-level program, **copy_system.c**, to copy one file to another, character by character.

We'll do this in a number of ways during this chapter to compare the efficiency of each method. For brevity, we'll assume that the input file exists and the output file does not. Of course, in real-life programs, we would check that these assumptions are valid!

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
     char c;
     int in, out;

     in = open("file.in", O_RDONLY);
     out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
     while(read(in,&c,1) == 1)
         write(out,&c,1);

     exit(0);
}
```

Note that the **#include <unistd.h>** *line must come first as it defines flags regarding POSIX compliance that may affect other include files.*

Running the program will give the following:

```
$ time copy_system
4.67user 146.90system 2:32.57elapsed 99%CPU
...
$ ls -ls file.in file.out
1029 -rw-r---r-    1 neil     users     1048576 Sep 17 10:46 file.in
1029 -rw-------    1 neil     users     1048576 Sep 17 10:51 file.out
```

We used the UNIX **time** facility to measure how long the program takes to run. It took 2 and one half minutes to copy the 1Mb file.

We can improve by copying in larger blocks. Here is the improved **copy_block.c** program.

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char block[1024];
    int in, out;
    int nread;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while((nread = read(in,block,sizeof(block))) > 0)
        write(out,block,nread);

    exit(0);
}
```

Now try the program, first removing the old output file:

```
$ rm file.out
$ time copy_block
0.01user 1.09system 0:01.90elapsed 57%CPU
...
$ ls -ls file.in file.out
1029 -rw-r--r--    1 neil      users       1048576 Sep 17 10:46 file.in
1029 -rw-------    1 neil      users       1048576 Sep 17 10:57 file.out
```

The revised program took under two seconds to do the copy.

Other System Calls for Managing Files

Here are some system calls that operate on these low-level file descriptors.

lseek

```
#include <unistd.h>

#include <sys/types.h>

off_t lseek(int fildes, off_t offset, int whence);
```

The **lseek** system call sets the read/write pointer of a file descriptor, **fildes**. You use it to set where in the file the next read or write will occur.

The **offset** parameter is used to specify the position and the **whence** parameter specifies how the offset is used.

whence can be one of the following:

- `SEEK_SET` `offset` is an absolute position
- `SEEK_CUR` `offset` is relative to the current position
- `SEEK_END` `offset` is relative to the end of the file

dup and dup2

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

The **dup** system calls provide a way of duplicating a file descriptor, giving two or more, different descriptors that access the same file.

File Status Information-Stat Family: fstat, stat and lstat

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

int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Note that the inclusion of `sys/types.h` *is deemed 'optional, but sensible'.*

The **fstat** system call returns status information about the file associated with an open file descriptor.

The members of the structure, **stat**, may vary between UNIX systems, but will include:

stat Member	Description
st_mode	File permissions and file type information.
st_ino	The inode associated with the file.
st_dev	The device the file resides on.
st_uid	The user identity of the file owner.
st_gid	The group identity of the file owner.
st_atime	The time of last access.
st_ctime	The time of last change to mode, owner, group or content.
st_mtime	The time of last modification to contents.
st_nlink	The number of hard links to the file.

The permissions flags are the same as for the **open** system call above. File-type flags include:

- **S_IFBLK** Entry is a block special device.
- **S_IFDIR** Entry is a directory.
- **S_IFCHR** Entry is a character special device.
- **S_IFIFO** Entry is a FIFO (named pipe).
- **S_IFREG** Entry is a regular file.
- **S_IFLNK** Entry is a symbolic link.

Other mode flags include:

- **S_ISUID** Entry has **setUID** on execution.
- **S_ISGID** Entry has **setGID** on execution.

Masks to interpret the **st_mode** flags include:

- **S_IFMT** File type.
- **S_IRWXU** User read/write/execute permissions.
- **S_IRWXG** Group read/write/execute permissions.
- **S_IRWXO** Others read/write/execute permissions.

There are some macros defined to help with determining file types. These include:

- **S_ISBLK** Test for block special file.
- **S_ISCHR** Test for character special fi
- **S_ISDIR** Test for directory.
- **S_ISFIFO** Test for FIFO.
- **S_ISREG** Test for regular file.
- **S_ISLNK** Test for symbolic link.

To test that a file doesn't represent a directory and has execute permisson set for the owner and no other permissions, we can use the test:

```
struct stat statbuf;
mode_t modes;

stat("filename",&statbuf);
modes = statbuf.st_mode;

if(!S_ISDIR(modes) && (modes & S_IRWXU) == S_IXUSR)
    ...
```

File and record locking-fcntl function

- File locking is applicable only for regular files.
- It allows a process to impose a lock on a file so that other processes can not modify the file until it is unlocked by the process.
- Write lock: it prevents other processes from setting any overlapping read / write locks on the locked region of a file.
- Read lock: it prevents other processes from setting any overlapping write locks on the locked region of a file.

- Write lock is also called a exclusive lock and read lock is also called a shared lock.
- fcntl API can be used to impose read or write locks on either a segment or an entire file.
- Function prototype:

```
#include<fcntl.h>

int fcntl (int fdesc, int cmd_flag, ….);
```

- All file locks set by a process will be unlocked when the process terminates.

File Permission-chmod

You can change the permissions on a file or directory using the **chmod** system call. Tis forms the basis of the **chmod** shell program.

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

int chmod(const char *path, mode_t mode);
```

chown

A superuser can change the owner of a file using the **chown** system call.

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

Links-soft link and hard link

Soft link(symbolic links): Refer to a symbolic path indicating the abstract location of another file.

- Used to provide alternative means of referencing files.
- Users may create links for files using **ln** command by specifying **–s** option.

hard links : Refer to the specific location of physical data.

- A hard link is a UNIX path name for a file.
- Most of the files have only one hard link. However users may create additional hard links for files using **ln** command.

Limitations:

- Users cannot create hard links for directories unless they have super user privileges.
- Users cannot create hard links on a file system that references files on a different systems.

unlink, link, symlink

We can remove a file using **unlink**.

```
#include <unistd.h>

int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

The **unlink** system call decrements the link count on a file.

The **link** system call cretes a new link to an existing file.

The **symlink** creates a symbolic link to an existing file.

Directories

As well as its contents, a file has a name and 'administrative information', i.e. the file's creation/modification date and its permissions.

The permissions are stored in the **inode**, which also contains the length of the file and where on the disc it's stored.

A directory is a file that holds the inodes and names of other files.

mkdir, rmdir

We can create and remove directories using the **mkdir** and **rmdir** system calls.

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

int mkdir(const char *path, mode_t mode);
```

The mkdir system call makes a new directory with **path** as its name.

```
#include <unistd.h>

int rmdir(const char *path);
```

The **rmdir** system call removes an empty directory.

chdir

A program can naviagate directories using the **chdir** system call.

```
#include <unistd.h>

int chdir(const char *path);
```

Current Working Directory- getcwd

A program can determine its current working directory by calling the **getcwd** library function.

```
#include <unistd.h>

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

The **getcwd** function writes the name of the current directory into the given buffer, **buf**.

Scanning Directories

The directory functions are declared in a header file, **dirent.h**. They use a structure, **DIR**, as a basis for directory manipulation.

Here are these functions:

- opendir, closedir
- readdir
- telldir
- seekdir

opendir

The **opendir** function opens a directory and establishes a directory stream.

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

DIR *opendir(const char *name);
```

readdir

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

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

The **readdir** function returns a pointer to a structure detailing the next directory entry in the directory stream **dirp**.

The **dirent** structure containing directory entry details included the following entries:

- ino_t d_ino The inode of the file.
- char d_name[] The name of the file.

telldir

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

long int telldir(DIR *dirp);
```

The **telldir** function returns a value that records the current position in a directory stream.

seekdir

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

void seekdir(DIR *dirp, long int loc);
```

The **seekdir** function sets the directory entry pointer in the directory stream given by **dirp**.

closedir

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

int closedir(DIR *dirp);
```

The **closedir** function closes a directory stream and frees up the resources associated with it.

Try It Out - A Directory Scanning Program

1. The **printdir**, prints out the current directory. It willrecurse for subdirectories.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>

void printdir(char *dir, int depth)
{
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;

    if((dp = opendir(dir)) == NULL) {
        fprintf(stderr,"cannot open directory: %s\n", dir);
        return;
    }
    chdir(dir);
    while((entry = readdir(dp)) != NULL) {
        stat(entry->d_name,&statbuf);
```

```
        if(S_ISDIR(statbuf.st_mode)) {
            /* Found a directory, but ignore . and .. */
            if(strcmp(".",entry->d_name) == 0 ||
                strcmp("..",entry->d_name) == 0)
                continue;
            printf("%*s%s/\n",depth,"",entry->d_name);
            /* Recurse at a new indent level */
            printdir(entry->d_name,depth+4);
        }
        else printf("%*s%s\n",depth,"",entry->d_name);
    }
    chdir("..");
    closedir(dp);
}
```

2. Now we move onto the **main** function:

```
int main()
{
    printf("Directory scan of /home/neil:\n");
    printdir("/home/neil",0);
    printf("done.\n");

    exit(0);
}
```

After some initial error checking, using **opendir**, to see that the directory exists, **printdir** makes a call to **chdir** to the directory specified. While the entries returned by **readdir** aren't null, the program checks to see whether the entry is a directory. If it isn't, it prints the file entry with indentation **depth**.

The program produces output like this (edited for brevity):How It Works

```
$ printdir
Directory scan of /home/neil:
.less
.lessrc
.term/
    termrc
.elm/
    elmrc
Mail/
    received
    mbox
.bash_history
.fvwmrc
.tin/
    .mailidx/
    .index/
        563.1
        563.2
posted
attributes
active
tinrc
done.
```

Here is one way to make the program more general.

```
int main(int argc, char* argv[])
{
    char *topdir, pwd[2]=".";
    if (argc != 2)
        topdir=pwd;
    else
        topdir=argv[1];

    printf("Directory scan of %s\n",topdir);
    printdir(topdir,0);
    printf("done.\n");

    exit(0)
}
```

You can run it using the command:

$ printdir /usr/local | more

UNIT-III

Processes and signals form a fundamental part of the UNIX operating environment, controlling almost all activities performed by a UNIX computer system.

Here are some of the things you need to understand.

- Process structure, type and scheduling
- Starting new processes in different ways
- Parent, child and zombie processes
- What signals are and how to use them

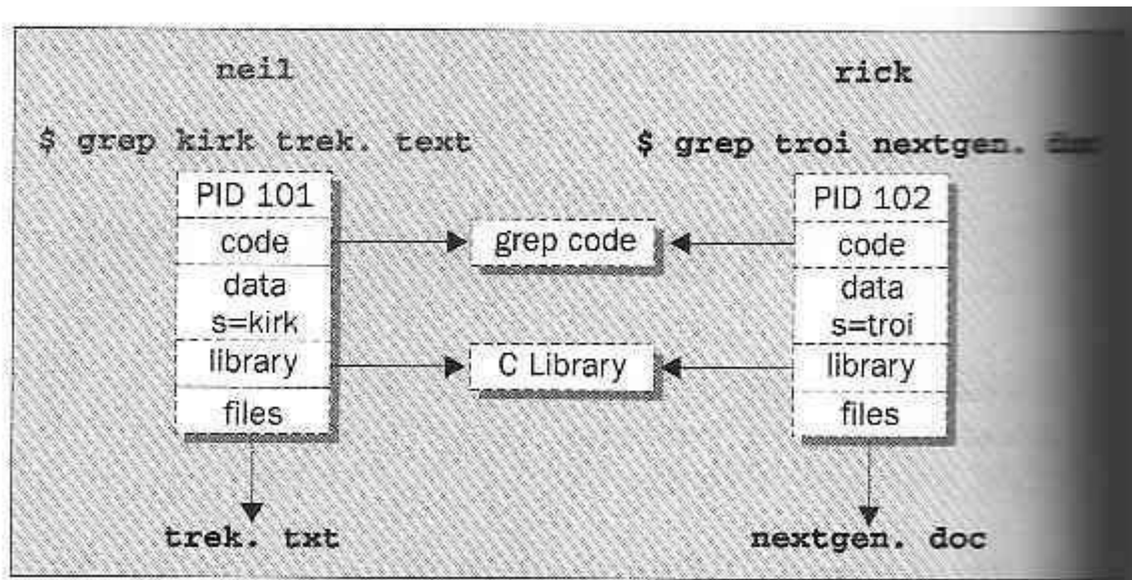
What is a Process

The X/Open Specification defines a process as an address space and single thread of control that executes within that address space and its required system resources.

A process is, essentially, a running program.

Layout of a C program

Here is how a couple of processes might be arranged within the operationg system.



Each process is allocated a unique number, a **process identifier**, or PID.

The program code that will be executed by the **grep** command is stored in a disk file.

The system libraries can also be shared.

A process has its own stack space.

Image in main memory

The UNIX **process table** may be though of as a data structure describing all of the processes that are currently loaded.

Viewing Processes

We can see what processes are running by using the **ps** command.

Here is some sample output:

```
$ ps
  PID TTY STAT  TIME COMMAND
   87 v01 S     0:00 -bash
  107 v01 S     0:00 sh /usr/X11/bin/startx
  115 v01 S     0:01 fvwm
  119 pp0 S     0:01 -bash
  129 pp0 S     0:06 emacs process.txt
  146 v01 S     0:00 oclock
```

The **PID** column gives the PIDs, the **TTY** column shows which terminal started the process, the **STAT** column shows the current status, **TIME** gives the CPU time used so far and the **COMMAND** column shows the command used to start the process.

Let's take a closer look at some of these:

```
87 v01 S     0:00 -bash
```

The initial login was performed on virtual console number one (**v01**). The shell is running **bash**. Its status is **s**, which means sleeping. Thiis is because it's waiting for the X Windows sytem to finish.

```
107 v01 S     0:00 sh /usr/X11/bin/startx
```

X Windows was started by the command **startx**. It won't finished until we exit from X. It too is sleeping.

```
115 v01 S     0:01 fvwm
```

The **fvwm** is a window manager for X, allowing other programs to be started and windows to be arranged on the screen.

```
119 pp0 S     0:01 -bash
```

This process represents a window in the X Windows system. The shell, bash, is running in the new window. The window is running on a new pseudo terminal (/dev/ptyp0) abbreviated pp0.

```
129 pp0 S      0:06 emacs process.txt
```

This is the EMACS editor session started from the shell mentioned above. It uses the pseudo terminal.

```
146 v01 S      0:00 oclock
```

This is a clock program started by the window manager. It's in the middle of a one-minute wait between updates of the clock hands.

Process environment

Let's look at some other processes running on this Linux system. The output has been abbreviated for clarity:

```
$ ps -ax
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
   40  ?  S     0:01 /usr/sbin/syslogd
   46  ?  S     0:00 /usr/sbin/lpd
   51  ?  S     0:00 sendmail: accepting connections
   88 v02 S     0:00 /sbin/agetty 38400 tty2
  109  ?  R     0:41 X :0
  192 pp0 R     0:00 ps -ax
```

Here we can see one very important process indeed:

```
1  ?  S     0:00 init
```

In general, each process is started by another, known as its **parent process**. A process so started is known as a **child process**.

When UNIX starts, it runs a single program, the prime ancestror and process number one: **init**.

One such example is the login procedure **init** starts the **getty** program once for each terminal that we can use to long in.

These are shown in the **ps** output like this:

```
88 v02 S     0:00 /sbin/agetty 38400 tty2
```

When interacting with your server through a shell session, there are many pieces of information that your shell compiles to determine its behavior and access to resources. Some of these settings are contained within configuration settings and others are determined by user input.

One way that the shell keeps track of all of these settings and details is through an area it maintains called the environment. The environment is an area that the shell builds every time that it starts a session that contains variables that define system properties.

In this guide, we will discuss how to interact with the environment and read or set environmental and shell variables interactively and through configuration files. We will be using an Ubuntu 12.04 VPS as an example, but these details should be relevant on any Linux system.

Every time a shell session spawns, a process takes place to gather and compile information that should be available to the shell process and its child processes. It obtains the data for these settings from a variety of different files and settings on the system.

Basically the environment provides a medium through which the shell process can get or set settings and, in turn, pass these on to its child processes.

Environment List

The environment is implemented as strings that represent key-value pairs. If multiple values are passed, they are typically separated by colon (:) characters. Each pair will generally will look something like this:

KEY=value1:value2:...

If the value contains significant white-space, quotations are used:

KEY="value with spaces"

The keys in these scenarios are variables. They can be one of two types, environmental variables or shell variables.

Environmental variables are variables that are defined for the current shell and are inherited by any child shells or processes. Environmental variables are used to pass information into processes that are spawned from the shell.

Shell variables are variables that are contained exclusively within the shell in which they were set or defined. They are often used to keep track of ephemeral data, like the current working directory.

By convention, these types of variables are usually defined using all capital letters. This helps users distinguish environmental variables within other contexts.

Environment variables- getenv, setenv

Every process has an environment block that contains a set of environment variables and their values. There are two types of environment variables: user environment variables (set for each user) and system environment variables (set for everyone).

By default, a child process inherits the environment variables of its parent process. Programs started by the command processor inherit the command processor's environment variables. To specify a different environment for a child process, create a new environment block and pass a pointer to it as a parameter to the CreateProcess function.

The command processor provides the set command to display its environment block or to create new environment variables. You can also view or modify the environment variables by selecting System from the Control Panel, selectingAdvanced system settings, and clicking Environment Variables.

Each environment block contains the environment variables in the following format:

Var1=Value1\0

Var2=Value2\0

Var3=Value3\0

...

VarN=ValueN\0\0

The name of an environment variable cannot include an equal sign (=).

The GetEnvironmentStrings function returns a pointer to the environment block of the calling process. This should be treated as a read-only block; do not modify it directly. Instead, use the SetEnvironmentVariable function to change an environment variable. When you are finished with the environment block obtained from GetEnvironmentStrings,call the FreeEnvironmentStrings function to free the block. Calling SetEnvironmentVariable has no effect on the system environment variables.

Kernel support for process

The kernel runs the show, i.e. it manages all the operations in a Unix flavored environment. The kernel architecture must support the primary Unix requirements. These requirements fall in two categories namely, functions for process management and functions for file management (files include device files). Process management entails allocation of resources including CPU, memory, and offers services that processes may need. The file management in itself involves handling all the files required by processes, communication with device drives and regulating transmission of data to and from peripherals. The kernel operation gives the user processes a feel of synchronous operation, hiding all underlying asynchronism in peripheral and hardware operations (like the time slicing by clock). In summary, we can say that the kernel handles the following operations :

1. It is responsible for scheduling running of user and other processes.

2. It is responsible for allocating memory.

3. It is responsible for managing the swapping between memory and disk.

4. It is responsible for moving data to and from the peripherals.

5. it receives service requests from the processes and honors them.

Process Identification:

Every process has a unique process ID, a non-negative integer. There are two special processes. Process ID_0 is usually the schedule process and is often known as the 'swapper'. No program on disk corresponds to this process – it is part of the kernel and is known as a system process, process ID_1 is usually the 'init' process and is invoked by the kernel at the end of the bootstrap procedure. The program files for this process loss /etc/init in older version of UNIX and is /sbin/init is newer version. 'init' usually reads the system dependent initialization files and brings the system to a certain state. The 'init' process never dies. 'init' becomes the parent process of any orphaned child process.

Process control

One further **ps** output example is the entry for the **ps** command itself:

```
192 pp0 R       0:00 ps -ax
```

This indicates that process 192 is in a run state (**R**) and is executing the command **ps-ax**.

We can set the process priority using **nice** and adjust it using **renice**, which reduce the priority of a process by 10. High priority jobs have negative values.

Using the **ps -l** (forlong output), we can view the priority of processes. The value we are interested in is shown in the **NI** (nice) column:

```
$ ps -l
 F   UID   PID  PPID PRI NI SIZE  RSS WCHAN      STAT TTY   TIME COMM
 0   501   146     1   1  0   85  756 130b85     S     v01  0:00 oclo
```

Here we can see that the **oclock** program is running with a default nice value. If it had been stated with the command,

```
$ nice oclock &
```

it would have been allocated a nice value of +10.

We can change the priority of a ruinning process by using the **renice** command,

```
$ renice 10 146
146: old priority 0, new priority 10
```

So that now the clock program will be scheduled to run less often. We can see the modified nice value with the **ps** again:

```
F    UID   PID  PPID PRI NI SIZE  RSS WCHAN      STAT TTY   TIME COMM
 0   501   146     1  20 10   85  756 130b85     S N  v01   0:00 oclo
```

Notice that the status column now also contains **N**, to indicate that the nice value has changed from the default.

Process Creation Starting New Processes

We can cause a program to run from inside another program and thereby create a new process by using the **system**. library function.

```
#include <stdlib.h>

int system (const char *string);
```

The **system** function runs the command passed to it as **string** and waits for it to complete.

The command is executed as if the command,

```
$ sh -c string
```

has been given to a shell.

Try It Out - system

1. We can use **system** to write a program to run **ps** for us.

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

int main()
{
    printf("Running ps with system\n");
    system("ps -ax");
    printf("Done.\n");
    exit(0);
}
```

2. When we compile and run this program, **system.c**, we get the following:

```
$ ./system
Running ps with system
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
...
  146 v01 S N   0:00 oclock
  256 pp0 S     0:00 ./system
  257 pp0 R     0:00 ps -ax
Done.
```

3. The **system** function uses a shell to start the desiredprogram.

We could put the task in the background, by changing the function call to the following:

```
    system("ps -ax &");
```

Now, when we compile and run this version of the program, we get:

```
$ ./system2
Running ps with system
Done.
$   PID TTY STAT  TIME COMMAND
      1  ?  S     0:00 init
      7  ?  S     0:00 update (bdflush)
...
    146 v01 S N   0:00 oclock
    266 pp0 R     0:00 ps -ax
```

How It Works

In the first example, the program calls **system** with the string **"ps -ax"**, which executes the **ps** program. Our program returns from the call to **system** when the **ps** command is finished.

In the second example, the call to **system** returns as soon as the shell command finishes. The shell returns as soon as the **ps** program is started, just as would happen if we had typed,

```
$ ps -ax &
```

at a shell prompt.

Replacing a Process Image

There is a whole family of related functions grouped under the **exec** heading. They differ in the way that they start processes and present program arguments.

```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ...,  (char *)0);
int execlp(const char *path, const char *arg0, ...,  (char *)0);
int execle(const char *path, const char *arg0, ...,  (char *)0, const char
*envp[]);
int execv(const char *path, const char *argv[]);
int execvp(const char *path, const char *argv[]);
int execve(const char *path, const char *argv[], const char *envp[]);
```

The **exec** family of functions replace the current process with another created according to the arguments given.

If we wish to use an **exec** function to start the **ps** program as in our previous examples, we have the following choices:

```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
const char *ps_argv[] =
    {"ps", "-ax", 0};

/* Example environment, not terribly useful */
const char *ps_envp[] =
    {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "-ax", 0);              /* assumes ps is in /bin */
execlp("ps", "ps", "-ax", 0);                  /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Try It Out - exclp

Let's modify our example to use an **exexlp** call.

Now, when we run this program, **pexec.c**, we get the usual **ps** output, but no **Done**. message at all.

Note also that there is no reference to a process called **pexec** in the output:

```
$ ./pexec
Running ps with execlp
  PID TTY STAT  TIME COMMAND
    1  ?  S     0:00 init
    7  ?  S     0:00 update (bdflush)
...
  146 v01 S N   0:00 oclock
  294 pp0 R     0:00 ps -ax
```

How It Works

The program prints its first message and then calls **execlp**, which searches the directories given by the **PATH** environment variable for a program called **ps**.

It then executes this program in place of our **pexec** program, starting it as if we had given the shell command:

```
$ ps -ax
```

Waiting for a Process

We can arrange for the parent process to wait until the child finishes before continuing by calling **wait**.

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

pid_t wait(int *stat_loc);
```

The **wait** system call causes a parent process to pause until one of its child processes dies or is stopped.

We can interrogate the status information using macros defined in **sys/wait.h**. These include:

Macro	Definition
`WIFEXITED(stat_val)`	Non-zero if the child is terminated normally.
`WEXITSTATUS(stat_val)`	If **`WIFEXITED`** is non-zero, this returns child exit code.
`WIFSIGNALED(stat_val)`	Non-zero if the child is terminated on an uncaught signal.
`WTERMSIG(stat_val)`	If **`WIFSIGNALED`** is non-zero, this returns a signal number.
`WIFSTOPPED(stat_val)`	Non-zero if the child has stopped on a signal.
`WSTOPSIG(stat_val)`	If **`WIFSTOPPED`** is non-zero, this returns a signal number.

Try It Out - wait

1. Let's modify our program slightly so we can wait for and examine the child process exit status. Call the new program **wait.c**.

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

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
```

2. This section of the program waits for the child process to finish:

```
    if(pid) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit (exit_code);
}
```

When we run this program, we see the parent wait for the child. The output isn't confused and the exit code is reported as expected.

```
$ ./wait
fork program starting
This is the parent
This is the child
This is the parent
```

```
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 410
Child exited with code 37
$
```

How It Works

The parent process uses the **wait** system call to suspend its own execution until status information becomes available for a child process.

Zombie Processes

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls **wait**.

This terminated child process is known as a **zombie process**.

Try It Out - Zombies

fork2.c is jsut the same as **fork.c**, except that the number of messages printed by the child and paent porcesses is reversed.

Here are the relevant lines of code:

```
switch(pid)
{
case -1:
    exit(1);
case 0:
    message = "This is the child";
    n = 3;
    break;
default:
    message = "This is the parent";
    n = 5;
    break;
}
```

How It Works

If we run the above program with **fork2 &** and then call the **ps** program after the child has finished but before the parent has finished, we'll see a line like this:

```
PID TTY STAT  TIME COMMAND

420 pp0 Z     0:00 (fork2) <zombie>
```

There's another system call that you can use to wail for child processes. It's called **waitpid** and you can use it to wait for a specific process to terminate.

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

If we want to have a parent process regularly check whether a specific child process had terminated, we could use the call,

```
waitpid(child_pid, (int *) 0, WNOHANG);
```

which will return zero if the child has not terminated or stopped or **child_pid** if it has.

Orphan Process

- When the parent dies first the child becomes **Orphan** .
- The kernel clears the process table slot for the parent.

System call interface for process management

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers

#incldue<sys/types.h>

#include<unistd.h>

pid_t getpid(void); Returns: process ID of calling process

pid_t geppid(void); Returns: parent process ID OF calling process

uid_t getuid(void); Returns: real user ID of calling process

uid_t geteuid(void);　　Returns: effective user ID of calling process

gid_t getgid(void);　　Returns: real group ID of calling process

gid_t getegid(void);　　Returns: effective group ID of calling process

fork Function

The only way a new process is created by the UNIX kernel is when an existing process calls the fork function.

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

#include<unistd.h>

pid_t fork(void);
```

Return: 0 is child, process ID of child in parent, -1 on error

The new process created by fork is called child process. This is called once, but return twice that is the return value in the child is 0, while the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process IDs of its children. The reason fork return 0 to the child is because a process can have only a single parent, so that child can always call getppid to obtain the process ID of its parent.

Both the child and parent contain executing with the instruction that follows the call to fork. The child is copy of the parent. For example, the child gets a copy of the parent's data space, heap and stack. This is a copy for the child the parent and children don't share these portions of memory. Often the parent and child share the text segment, if it is read-only.

There are two users for fork:

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers_ the parent waits for a service requests from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.

   When a process wants to execute a different program, this is common for shells. In this case the child does an exec right after it returns from the fork.

vfork Function

The function vfork has the same calling sequence and share return values as fork. But the semantics of the two functions differ. vfork is intended to create a new process when the purpose of the new process is to exec a new program. vfork creates the new process, just like fork, without fully copying the address space of the parent into the child, since the child won't reference the address space – the child just calls exec right after the vfork. Instead, while the child is running, until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual memory implementations of UNIX.

Another difference between the two functions is that vfork guarantees that the child runs first, until the parent resumes.

exit Function

There are three ways for a process to terminate normally, and two forms of abnormal termination.

1. Normal termination:
    a. Executing a return from the main function. This is equivalent to calling exit
    b. Calling the exit function
    c. Calling the _exit function
2. Abnormal termination
    a. Calling abort: It generates the SIGABRT signal
    b. When the process receives certain signals. The signal can be generated by the process itself

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and the like.

For any of the preceding cases we want the terminating process to be able to notify its parent how it terminated. For the exit and _exit functions this is done by passing an exit status as the argument to these two functions. In the case of an abnormal termination however, the kernel generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the wait or waitpid function.The exit status is converted into a termination status by the kernel when _exit is finally called. If the child terminated normally, then the parent can obtain the exit status of the child.

If the parent terminates before the child, then init process becomes the parent process of any process, whose parent terminates; that is the process has been inherited by init. Whenever a process terminates the kernel goes through all active processes to see if the terminating process is the parent of any process that still exists. If so, the parent process ID of the still existing process is changed to be 1 to assume that every process has a parent process.

When a child terminates before the parent, and if the child completely disappeared, the parent wouldn't be able to fetch its termination status, when the parent is ready to seek if the child had terminated. But parent get this information by calling wait and waitpid, which is maintained by the kernel.

wait and waitpid Functions

When a process terminates, either normally or abnormally, the parent is notified by the kernel sending the parent SIGCHLD signal. Since the termination of a child is an asynchronous event, this signal is the asynchronous notification from the kernel to the parent. The default action for this signal is to be ignored. A parent may want for one of its children to terminate and then accept it child's termination code by executing wait.

A process that calls wait and waitpid can

1. block (if all of its children are still running).
2. return immediately with termination status of a child (if a child has terminated and is waiting for its termination status to be fetched) or
3. return immediately with an error (if it down have any child process).

If the process is calling wait because it received SIGCHLD signal, we expect wait to return immediately. But, if we call it at any random point in time, it can block.

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

#include<sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, o or -1 on error

The difference between these two functions is

1. wait can block the caller until a child process terminates, while waitpid has an option that prevents it from blocking.
2. waitpid does not wait for the first child to terminate, it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates: if the caller blocks and has multiple children, wait returns when one terminates, we can know this process by PID return by the function.

For both functions, the argument statloc is pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

If we have more than one child, wait returns on termination of any of the children. A function that waits for a specific process is waitpid function.

The interpretation of the pid argument for waitpid depends on its value:

pid == -1	waits for any child process. Here, waitpid is equivalent to wait
pid > 0	waits for the child whose process ID equals pid
pid == 0	waits for any child whose process group ID equals that of the calling process
pid < -1	waits for any child whose process group ID equals the absolute value of pid

waitpid returns the process ID of the child that terminated, and its termination status is returned through statloc. With wait the only error is if the calling process has no children. With waitpid however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The *options* argument lets us further control the operation of waitpid. This argument is either 0 or is constructed from the bitwise OR of the following constants.

WNOHANG	waitpid will not blink if a child specified by pid is not immediately available. In this case, the return value is 0.
WUNTRACED	if the status of any child specified by pid that has stopped, and whose

status has not been updated since it has stopped, is returned

The waitpid function provides these features that are not provided by the wait function are:

1. waitpid lets us to wait for one particular process
2. waitpid provides a non-blocking version of wait
3. waitpid supports job control (wit the WUNTRACED option)

exec Function

The fork function can create a new process that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program and the new program starts executing at its main function. The process ID doesn't change across an exec because a new process is not created. exec merely replaces the current process with a brand new program from disk.

There are six different exec functions. These six functions round out the UNIX control primitives. With fork we can create new processes, and with the exec functions we can initiate new programs. The exit function and the two wait functions handle termination and waiting for termination. These are the only process control primitives we need.

```
#include<unistd.h>

int execl(const char *pathname, const char *arg0, . . . /*(char *) 0*/

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, . . . /* (char *) 0, char envp[]*/);

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

int execlp(const char *pathname, const char *arg0, . . . /* (char *) 0*/);

int execvp(const char *filename, char *const argv[]);
```

All six returns: -1 on error, no return on success.

The first difference in these functions is that the first four take a pathname argument, while the last two take a filename argument. When a filename argument is specified:

- if filename contains a slash, it is taken as a pathname.
- Otherwise, the executable file is a searched for in directories specified by the PATH

environment variable.

The PATH variable contains a list of directories (called path prefixes) that are separated by colors. For example, the name=value environment string

PATH=/bin:/usr/bin:usr/local/bin/:.

Specifies four directories to search, where last one is current working directory.

If either of the two functions, execlp or execvp finds an executable file using one of the path prefixes, but the file is not a machine executable that was generated by the link editor, it assumes the file is a shell script and tries to invoke /bin/sh with filename as input to the shell.

The next difference concerns the passing of argument list. The function execl, execlp and execle require each of the command-line arguments to the new program to be specified as separate arguments. The end of the argument should be a null pointer. For the other three functions execv, execvp and execve, we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

The final difference is the passing of the environment list to the new program. The two functions execle and execve allow us to pass a pointer to an array of pointer to an array of pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

Differences Between Threads and Processes

UNIX processes can cooperate; they can send each other messages and they can interrupt one another.

There is a class of process known as a **thread** which are distinct from processes in that they are separate execution streams within a single process.

Signals

A **signal** is an event generated by the UNIX system in response to some condition, upon receipt of which a process may in turn take some action.

Signal names are defined in the header file **signal.h**. They all begin with **SIG** and include:

Signal Name	Description
SIGABORT	*Process abort
SIGALRM	Alarm clock
SIGFPE	*Floating point exception
SIGHUP	Hangup
SIGILL	*Illegal instruction
SIGINT	Terminal Interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGQUIT	Terminal Quit
SIGSEGV	*Invalid memory segment access
SIGTERM	Termination
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

Additional signals include:

Signal Name	Description
SIGCHLD	Child process has stopped or exited
SIGCONT	Continue executing, if stopped
SIGSTOP	Stop executing (can't be caught or ignored)
SIGTSTP	Terminal stop signal
SIGTTIN	Background process trying to read
SIGTTOU	Background process trying to write

If the shell and terminal driver are configured normally, typing the interrupt character (Ctrl-C) at the keyboard will result in the **SIGINT** signal being sent to the foreground process. This will cause the program to terminate.

We can handle signals using the **signal** library function.

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

The **signal** function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of these tow special values:

- **SIG_IGN** Ignore the signal.
- **SIG_DFL** Restore default behavior.

Signal generation & Handling

1. We'll start by writing the function which reacts to the signal which is passed in the parameter **sig**.

Let's call it **ouch**:

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}
```

2. The **main** function has to intercept the **SIGINT** signal generated when we type Ctrl-C.

For the rest of the time, it just sits in an infinite loop, printing a message once a second:

```
int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

3. While the program is running, typing Ctrl-C causes it to react and then continue.

When we **type** Ctrl-C again, the program ends:

```
$ ./ctrlc
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

How It Works

The program arranges for the function **ouch** to be called when we type Ctrl-C, which gives the **SIGINT** signal.

Kernel support for Signals-Sending Signals

A process may send a signal to itself by calling **raise**.

```
#include <signal.h>

int raise(int sig);
```

A process may send a signal to another process, including itself, by calling **kill**.

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

int kill(pid_t pid, int sig);
```

Signals provide us with a useful alarm clock facility.

The **alarm** function call can be used by a process to schedule a **SIGALRM** signal at some time in the future.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Try It Out - An Alarm Clock

1. In **alarm.c**, the first function, **ding**, simulates an alarm clock:

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

void ding(int sig)
{
    printf("alarm has gone off\n");
}
```

2. In **main**, we tell the child process to wait for five seconds before sending a **SIGALRM** signal to its parent:

```
int main()
{
    int pid;

    printf("alarm application starting\n");

    if((pid = fork()) == 0) {
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
```

3. The parent process arranges to catch **SIGALRM** with a call to **signal** and then waits for the inevitable.

```
    printf("waiting for alarm to go off\n");
    (void) signal(SIGALRM, ding);

    pause();

    printf("done\n");
    exit(0);
}
```

When we run this program, it pauses for five seconds while it waits for the simulated alarm clock.

```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
alarm has gone off
done
$
```

This program introduces a new function, **pause**, which simply causes the program to suspend execution until a signal occurs.

It's declared as,

```
#include <unistd.h>

int pause(void);
```

How It Works

The alarm clock simulation program starts a new process via **fork**. This child process sleeps for five seconds and then sends a **SIGALRM** to its parent.

A Robust Signals Interface

X/Open specification recommends a newer programming interface for signals that is more robust: **sigaction**.

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

The **sigaction** structure, used to define the actions to be taken on receipt of the signal specified by **sig**, is defined in **signal.h** and has at least the following members:

`void (*) (int) sa_handler`	function, `SIG_DFL` or `SIG_IGN`
`sigset_t sa_mask`	signals to block in `sa_handler`
`int sa_flags`	signal action modifiers

Try It Out - sigaction

Make the changes shown below so that **SIGINT** is intercepted by **sigaction**. Call the new

program **ctrlc2.c**.

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Running the program, we get a message when we type Ctrl-C because **SIGINT** is handled repeated;y by **sigaction**.

Type Ctrl-\ to terminate the program.

```
$ ./ctrlc2
Hello World!
Hello World!
```

```
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^\
Quit
$
```

How It Works

The program calls **sigaction** instead of **signal** to set the signal handler for Ctrl-C (**SIGINT**) to the function **ouch**.

Signal Sets

The header file **signal.h** defines the type **sigset_t and functions used to manipulate sets of signals.**

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

The function **sigismember** determines whether the given signal is amember of a signal set.

```
#include <signal.h>

int sigismember(sigset_t *set, int signo);
```

The process signal mask is set or examined by calling the function **sigprocmask**.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

sigprocmask can change the process signal mask in a number of ways according to the **how** argument.

The **how** argument can be one of:

- **SIG_BLOCK** The signals in **set** are added to the signal mask.
- **SIG_SETMASK** The signal mask is set from **set**.
- **SIG_UNBLOCK** The signals in **set** are removed from the signal mask.

If a signal is blocked by a process, it won't be delivered, but will remain pending.

A program can determine which of its blocked signals ar pending by calling the function **sigpending**.

```
#include <sigpending>

int sigpending(sigset_t *set);
```

A process can suspend execution until the delivery of one of a set of signals by calling **sigsuspend**.

This is a more general form of the **pause** function we met earlier.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Signal Functions

The system calls related to signals are explained in the following sections.

Unreliable signals

The signals could get lost – a signal could occur and the process would never know about it. Here, the process has little control over a signal, it could catch the signal or ignore it, but blocking of a signal is not possible.

Reliable signals

Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

Signal dispositions

Each signal has a current *disposition*, which determines how the process behaves when it is delivered the signal.
The entries in the "Action" column of the tables below specify the default disposition for each signal.

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.

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

#include<signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Both return: 0 if OK, -1 on error
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.

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.

pid = -1 unspecified.

alarm and pause Functions

The alarm function allows us to get 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. There is only one of the alarm clocks per process. If, when we call alarm, there is a previously registered alarm clock for the process that has not yet expired, the number of seconds left for that alarm clock to return as the value of this function. That previously registered alarm clock is replaced by the new value.

If there is a previously registered alarm clock for the process that has not yet expired and if the *seconds* value is 0, the previous alarm clock is cancelled. 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 terminating the process, most processes use an alarm clock catch this signal.

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.

abort Function

abort function causes abnormal program termination.

#include<stdlib.h>

void abort(void);

This function never returns.

This function sends the SIGABRT signal to the process. A process should not ignore this signal. abort overrides the blocking or ignoring of the signal by the process.

sleep Function

#include<unistd.h>

unsigned int sleep(unsigned int seconds);

Returns: 0 or number of unslept seconds.

This function causes the calling process to be suspended until either:

1. The amount of clock that is specified by *seconds* has elapsed or
2. A signal is caught by the process and the signal handler returns.

In case 1 the return value is 0 when sleep returns early, because of some signal being caught case 2, the return value is the number of unslept seconds.

Sleep can be implemented with an alarm function. If alarm is used, however, there can be interaction between the two functions.

Unit IV
Introduction to IPC

Interprocess communication (IPC) includes thread synchorization and data exchange between threads beyond the process boundaries. If threads belong to the same process, they execute in the same address space, i.e. they can access global (static) data or heap directly, without the help of the operating system. However, if threads belong to different processes, they cannot access each others address spaces without the help of the operating system.

There are two fundamentally different approaches in IPC:

- processes are residing on the same computer
- processes are residing on different computers

The first case is easier to implement because processes can share memory either in the user space or in the system space. This is equally true for uniprocessors and multiprocessors.

In the second case the computers do not share physical memory, they are connected via I/O device(for example serial communication or Ethernet). Therefore the processes residing in different computers can not use memory as a means for communication.

IPC between processes on a Single System

Most of this chapter is focused on IPC on a single computer system, including four general approaches:

- Shared memory
- Messages
- Pipes
- Sockets

The synchronization objects considered in the previous chapter normally work across the process boundaries (on a single computer system). There is one addition necessary however: the synchronization objects must be named. The handles are generally private to the process, while the object names, like file names, are global and known to all processes.

```
h = init_CS("xxx");
h = init_semaphore(20,"xxx");
h = init_event("xxx");
h = init_condition("xxx");
h = init_message_buffer(100,"xxx");
```

IPC between processes on different systems

IPC between processes on different systems

IPC is Inter Process Communication, more of a technique to share data across different processes **within** one machine, in such a way that data passing binds the coupling of different processes.

The first, is using memory mapping techniques, where a memory map is created, and others

open the memory map for reading/writing...

The second is, using sockets, to communicate with one another...this has a high overhead, as each process would have to open up the socket, communicate across... although effective

The third, is to use a pipe or a named pipe, a very good example

PIPES:

A pipe is a ***serial*** communication device (i.e., the data is read in the order in which it was written),which allows a ***unidirectional communication***. The data written to end isreadbackfromtheotherend.

The pipe is mainly used to communicate between two threads in a single process or between parent and child process. Pipes can only connect the related process. In shell, thesymbolcanbeusedtocreateapipe.

In pipes the ***capacity of data is limited***. (i.e.) If the writing process is faster than the reading process which consumes the data, the pipe cannot store the data. In this situation the writer process will block until more capacity becomes available. Also if the reading process tries to read data when there is no data to read, it will be blocked until the data becomes available. By this, pipes ***automatically synchronize the two process***.

Creatingpipes:

The *pipe()* function provides a means of passing data between two programs and also allows to read and write the data.

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

pipe()function is passed with an array of file descriptors. It will fill the array with new file descriptors and returns zero. On error, returns -1 and sets the errno to indicate the reason of failure.

The file descriptors are connected in a way that is data written to file_ descriptor[1] can be read back from the file_descriptor[0].

(**Note**: As this uses file descriptors and not the file streams, we must use read and write system calls to access the data.)

Pipes are originally used in UNIX and are made even more powerful in Windows 95/NT/2000.

Pipes are implemented in file system. Pipes are basically files with only two file offsets: one for reading another for writing. Writing to a pipe and reading from a pipe is strictly in FIFO manner. (Therefore pipes are also called FIFOs).

For efficiency, pipes are in-core files, i.e. they reside in memory instead on disk, as any other global data structure. Therefore pipes must be restricted in size, i.e. number of pipe blocks must be limited. (In UNIX the limitation is that pipes use only direct blocks.)Since the pipes have a limited size and the FIFO access discipline, the reading and writing processes are synchronized in a similar manner as in case of message buffers. The access functions for pipes are the same as for files: WriteFile() and ReadFile().

Pipes used as standard input and output:

We can invoke the standard programs, ones that don't expect a file descriptor as a parameter.

```
#include<unistd.h>
int dup(int file_descriptor);
int  dup2(int  file_descriptor_1,
int file_descriptor_2);
```

The purpose of dup call is to open a new file descriptor, which will refer to the same file as an existing file descriptor. In case of dup, the value of the new file descriptor is the lowest number available. In dup2 it is same as, or the first available descriptor greater than the parameter file_descriptor_2.

We can pass data between process by first closing the file descriptor 0 and call is made to dup. By this the new file descriptor will have the number 0.As the new descriptor is the duplicate of an existing one, standard input is changed to have the access. So we have created two file descriptors for same file or pipe, one of them will be the standard input.

(**Note:** The same operation can be performed by using the fcntl() function. But compared to this dup and dup2 are more efficient)

```
//pipes.c
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

int main()
{
int   data_processed;
int file_pipes[2];
const char some_data[]= "123";
pid_t fork_result;

if(pipe(file_pipes)==0)
{
fork_result=fork();
if(fork_result==(pid_t)-1)
{
fprintf(stderr,"fork failure");
exit(EXIT_FAILURE);
}
if(fork_result==(pid_t)0)
{
close(0);
dup(file_pipes[0]);
close(file_pipes[0]);
```

```
close(file_pipes[1]);
execlp("od","od","-c",(char *)0);
exit(EXIT_FAILURE);
}
else
{
close(file_pipes[0]);
data_processed=write(file_pipes[1],
some_data,strlen(some_data)); close(file_pipes[1]);
printf("%d -wrote %d bytes\n",(int)getpid(),data_processed);
}
} exit(EXIT_SUCCESS);
}
```

The program creates a pipe and then forks. Now both parent and child process will have its own file descriptors for reading and writing. Therefore totally there are four file descriptors.

The child process will close its standard input with *close(0)* and calls duo(*file_pipes[0])*. This will duplicate the file descriptor associated with the read end. Then child closes its original file descriptor. As child will never write, it also closes the write file descriptor, *file_pipes[1]*. Now there is only one file descriptor 0 associated with the pipe that is standard input. Next, child uses the *exec* to invoke any program that reads standard input.
The *od* command will wait for the data to be available from the user terminal.

Since the parent never read the pipe, it starts by closing the read end that is file_pipe[0]. When writing process of data has been finished, the write end of the parent is closed and exited. As there are no file descriptor open to write to pipe, the od command will be able to read the three bytes written to pipe, meanwhile the reading process will return 0 bytes indicating the end of the file.

There are two types of pipes:

- Namedpipes.
- Unnamed pipes (Anonymous pipes)

Named pipes (FIFOs)

Similar to pipes, but allows for communication between unrelated processes.
This is done by naming the communication channel and making it permanent.

Like pipe, FIFO is the unidirectional data stream.

FIFO creation:

int mkfifo (const char *pathname, mode_t mode);

- makes a FIFO special file with name pathname.
  (mode specifies the FIFO's permissions, as common in UNIX-like file systems).
- A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo()

Once a FIFO special file has been created, any process can open it for reading or writing, in the same way as an ordinary file.

A First-in, first-out(FIFO) file is a pipe that has a name in the filesystem. It is also called as med pipes.

Creation of FIFO:
We can create a FIFO from the command line and within a program.

To create from **command line** we can use either *mknod* or *mkfifo* commands.

```
$ mknod filename p
$ mkfifo filename
```

(**Note**: The mknod command is available only n older versions, you can make use of mkfifo in new versions.)

To create FIFO **within the program** we can use two system calls. They are,

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

int mkfifo(const char
*filename,mode_t mode);
int  mknod(const  char  *filename,
mode_t mode|S_IFIFO,(dev_t) 0);
```

If we want to use the mknod function we have to use ORing process of fileaccess mode with S_IFIFO and the dev_t value of 0.Instead of using this we can use the simple mkfifo function.

Accessing FIFO:
Let us first discuss how to access FIFO in command line using file commmands. The useful feature of named pipes is, as they appear in the file system, we can use them in commands.

We can read from the FIFO(empty)

```
$ cat < /tmp/my_fifo
```

Now, let us write to the FIFO.

```
$ echo "Simple!!!" > /tmp/my_fifo
```

(**Note:** These two commands should be executed in different terminals because first command will be waiting for some data to appear in the FIFO.)

FIFO can also be accessed as like a file in the program using low-level I/O functions or C library I/O functions.

The only difference between opening a regular file and FIFO is the use of *open_flag* with the option*O_NONBLOCK.* The only restriction is that we can’t open FIFO for reading and writing with O_RDWR mode.

//fifo1.c

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

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

#define  FIFO_NAME  "/tmp/my_fifo"
#define   BUFFER_SIZE   PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
int pipe_fd;
int res;
int open_mode = O_WRONLY;
int bytes_sent = 0;
char buffer[BUFFER_SIZE + 1];
if (access(FIFO_NAME, F_OK) == -1) {
res = mkfifo(FIFO_NAME, 0777);
if (res != 0) {
fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
exit(EXIT_FAILURE);
}
}
printf("Process %d opening FIFO O_WRONLY\n", getpid());
pipe_fd = open(FIFO_NAME, open_mode);
printf("Process %d result %d\n", getpid(), pipe_fd); if
(pipe_fd != -1) {
while(bytes_sent < TEN_MEG) {
res = write(pipe_fd, buffer, BUFFER_SIZE); if
(res == -1) {
fprintf(stderr, "Write error on pipe\n");
exit(EXIT_FAILURE);
}
}
(void)close(pipe_fd);
}
else { exit(EXIT_FAILURE);
}
printf("Process %d finished\n", getpid());
exit(EXIT_SUCCESS);
}
```

//fifo2.c

```
#include     <unistd.h>
#include      <stdlib.h>
#include       <stdio.h>
#include      <string.h>
#include       <fcntl.h>
#include      <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define  FIFO_NAME  "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF int
```

```
main()
{
int pipe_fd; int
res;
int open_mode = O_RDONLY; char
buffer[BUFFER_SIZE + 1]; int
bytes_read = 0;
memset(buffer, '\0', sizeof(buffer));
printf("Process %d opening FIFO O_RDONLY\n", getpid());
pipe_fd = open(FIFO_NAME, open_mode);
printf("Process %d result %d\n", getpid(), pipe_fd);
if (pipe_fd != -1) {
do {
res = read(pipe_fd, buffer, BUFFER_SIZE);
bytes_read += res;
} while (res > 0);
(void)close(pipe_fd);
}
else {
exit(EXIT_FAILURE);
}
printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
exit(EXIT_SUCCESS);
}
```

Both *fifo1.c* and *fifo2.c* programs use the FIFO in blocking mode.

First fifo1.c is executed .It blocks and waits for reader to open the named pipe. Now writer unblocks and starts writing data to pipe. At the same time, the reader starts reading data from the pipe.

Unnamed pipes (Anonymous Pipes)

Anonymous pipes don't have names, therefore they can be used only between related processes which can inherit the file handles (file descriptors).

Anonymous pipes are typically used to "pipe" two programs: standard output from one program is redirected to the pipe input (write handle), while the standard input of the second program is redirected to from the pipe output (read handle). The pipe is created by the parent (usually the login shell), and the pipe handles are passed to children through the inheritance mechanism.

Anonymous pipes cannot be used across a network. Also, anonymous pipes are unidirectional- in order to communicate two related processes in both directions, two anonymous pipes must be created.

Example of Win32 anonymous pipes used for program piping:

```
//*******************************************************************
// This program implements piping of programs p1.exe and p2.exe
// through an anonymous pipe. The program creates two child processes
// (which execute images p1.exe and p2.exe) and a pipe, then passes
// the pipe handles to the children.
```

```
//
// The program is invoked as: pipe p1 p2 (no command line arguments)
//*****************************************************************
#include <windows.h>
#include <iostream.h>
int main(int argc, char *argv[])
{
// Create anonymous (unnamed) pipe
SECURITY_ATTRIBUTES sa;
sa.nLength  =  sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = 0;
sa.bInheritHandle = TRUE; // Handles are inheritable (default is FALSE)
HANDLE rh,wh;           //  Read  and  write  handles  of  the  pipe
if(!CreatePipe(&rh,&wh,&sa,0))
{
cout  <<  "Couldn't  create  pipe  "  <<  GetLastError()<<  endl;
return (1);
}
//  Create  the  first  child  process  p1
PROCESS_INFORMATION       pi1;
STARTUPINFO si1;
GetStartupInfo(&si1);          // Get default startup structure
si1.hStdInput       =       GetStdHandle(STD_INPUT_HANDLE);
si1.hStdOutput = wh;          // Std output of p1 is input to the pipe
si1.dwFlags = STARTF_USESTDHANDLES;
CreateProcess( argv[1],      // Name of the p1's image (without ".exe."
0,0,0,
TRUE,                          // Each open inheritable handle of the
                               // parent will be inherited by the child
0,0,0,
&si1,&pi1);
CloseHandle(wh);          // Pipe handle no longer needed
 //  Create  the  second  child  process  p2
PROCESS_INFORMATION            pi2;
STARTUPINFO si2;
GetStartupInfo(&si2);         //    Get    default    startup    structure
si2.hStdInput = rh;             // Std input of p2 is otput from the pipe
si2.hStdOutput    =    GetStdHandle(STD_OUTPUT_HANDLE);
si2.dwFlags = STARTF_USESTDHANDLES;
CreateProcess( 0,argv[2],          // Name of the p1's image (without ".exe."
0,0,
TRUE,                              // Each open inheritable handle of the
                                    // parent will be inherited by the child
0,0,0,
&si2,&pi2);
WaitForSingleObject(pi1.hProcess,INFINITE);
CloseHandle(pi1.hProcess);
WaitForSingleObject(pi2.hProcess,INFINITE);
CloseHandle(pi2.hProcess);
CloseHandle(rh);
```

```
return(0);
}
```

Comment:
In order to communicate two processes (**P**1 and **P**2) through anonymous pipes by redirecting the standard I/O, the processes don't have to be aware of the existence of pipes, i.e. their sources and images don't have to be modified.

Pipe processing:(popen &pclose library functions)

The process of passing data between two programs can be done with the help of popen() and pclose() functions.

```
#include<stdio.h>
FILE *popen(const char *command ,
const char *open-mode);
int pclose(FILE *stream_to_close);
```

popen():
The popen function allows a program to invoke another program as a new process and either write the data to it or to read from it. The parameter command is the name of the program to run. The open_mode parameter specifies in which mode it is to be invoked, it can be only either "r" or "w". On failure popen() returns a NULL pointer. If you want to perform bi-directional communication you have to use two pipes.

pclose():
By using pclose(), we can close the filestream associated with popen() after the process started by it has been finished. The pclose() will return the exit code of the process, which is to be closed. If the process was already executed a *wait* statement before calling pclose, the exit status will be lost because the process has been finished. After closing the filestream, pclose() will wait for the child process to terminate.

Messagequeue:
This is an easy way of passing message between two process. It provides a way of sending a *block of data* from one process to another. The main advantage of using this is, each block of data is considered to have a type, and a receiving process receives the blocks of data having different type values independently.

Creation and accessing of a message queue:
You can create and access a message queue using the *msgget()* function.

```
#include<sys/msg.h>
int msgget(key_t key,int msgflg);
```

The first parameter is the key value, which specifies the particular message queue. The special constant IPC_PRIVATE will create a private queue. But on some Linux systems the message queue may not actually be private.

The second parameter is the flag value, which takes nine permission flags.

Adding a message:
The *msgsnd()* function allows to add a message to a message queue.
#include<sys/msg.h>
int msgsnd(int msqid,const void *msg_ptr ,size_t msg_sz,int msgflg);

The first parameter is the message queue identifier returned from an msgget function.

The second parameter is the pointer to the message to be sent. The third parameter is the size of the message pointed to by msg_ptr. The fourth parameter, is the flag value controls what happens if either the current message queue is full or within the limit. On success, the function returns 0 and a copy of the message data has been taken and placed on the message queue, on failure -1 is returned.

Retrieving a message:
The *smirch()* function retrieves message from the message queue.
#include<sys/msg.h>
int msgsnd(int msqid,const void *msg_ptr
,size_t msg_sz,long int msgtype ,int msgflg);

The second parameter is a pointer to the message to be received.

The fourth parameter allows a simple form of reception priority. If its value is 0,the first available message in the queue is retreived. If it is greater than 0,the first message type is retrived. If it is less than 0,the first message that has a type the same a or less than the absolute value of msgtype is retrieved.

On success, msgrcv returns the number on bytes placed in the receive buffer, the message is copied into the user-allocated buffer and the data is deleted from the message queue. It returns -1 on error.

Controlling the message queue:
This is very similar that of control function of shared memory.

#include<sys/msg.h>
int msgctl(int msgid,int command,
struct msqid_ds *buf);

The second parameter takes the values as given below:

1.) **IPC_STAT** - Sets the data in the msqid_ds to reflect the values associated with the message queue.

2.) **IPC_SET** - If the process has the permission to do so, this sets the values associated with the message queue to those provided in the msgid_ds data structure.

3.) **IPC_RMID**-Deletes the message queue.

(Note: If the message queue is deleted while the process is writing in a msgsnd or msgrcv function, the send or receive function will fail.

Client /server Example:

//msgq1.c

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

#include<sys/msg.h>

struct my_msg_st
{
long   int   my_msg_type;
char some_text[BUFSIZ];
};

int main()

{
int running = 1;
int msgid;
struct my_msg_st some_data;
long int msg_to_receive = 0;



 msgid = msgget( (key_t)1234,
0666 | IPC_CREAT);
if (msgid == -1)
{
fprintf(stderr, "failed to get:\n");
exit(EXIT_FAILURE);
}
while (running)
{
if(msgrcv(msgid, (void *)&some_data,
BUFSIZ,msg_to_receive,0) == -1)

{
fprintf(stderr, "failedto receive: \n");
exit(EXIT_FAILURE);
}

printf("You                Wrote:               %s",
some_data.some_text);
```

```
if(strncmp(some_data.some_text, "end", 3)
== 0)
{
running = 0;

}
}
if (msgctl(msgid, IPC_RMID, 0) == -1)
{
fprintf(stderr, "failed to delete\n");
exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
```

//msgq2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/msg.h>
#define MAX_TEXT 512

struct my_msg_st
{
long int my_msg_type;
char some_text[MAX_TEXT];
};

int main()
{
int running = 1;
struct my_msg_st some_data;
int msgid;
char buffer[BUFSIZ];

msgid = msgget( (key_t)1234,
0666 | IPC_CREAT);
if (msgid == -1)
{
fprintf(stderr, "failed to create:\n");
exit(EXIT_FAILURE);
}
while(running)
{
printf("Enter Some Text: ");
fgets(buffer, BUFSIZ, stdin);
some_data.my_msg_type = 1;
strcpy(some_data.some_text, buffer);
```

```
if(msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1)
{
fprintf(stderr, "msgsnd failed\n");
exit(EXIT_FAILURE);
}
if(strncmp(buffer, "end", 3) == 0)
{
running = 0;
}
}
exit(EXIT_SUCCESS);
}
```

The *msgq1.c* program will create the message queue using msgget() function. The *msgid* identifier is returned by the msgget().The message are received from the queue using *msgrcv()* function until the string "end" is encountered. Then the queue is deletedusing *msgctl()* function.

The *msgq2.c* program uses the *msgsnd()* function to send the entered text to the queue.

Semaphore:

While we are using threads in our programs in multi-user systems, multiprocessing system, or a combination of two, we may often discover critical sections in the code. This is the section where we have to ensure that a single process has exclusive access totheresource.

For this purpose the semaphore is used. It allows in managing the access to resource.
To prevent the problem of one program accessing the shared resource simultaneously, we are in Need to generate and use a token which guarantees the access to only one threadofexecutioninthecriticalsectionatatime.

It is counter variable, which takes only the positive numbers and upon which programs can only act atomically. The positive number is the value indicating the number of units of the shared resources are available for sharing.

The common form of semaphore is the *binary semaphore*, which will control a single resource, and its value is initialized to 0.

Creation of semaphore:

The *shmget()* function creates a new semaphore or obtains the semaphore key of an existing semaphore.

```
#include<sys/sem.h>
intshmget(key_tkey,intnum_sems,
intsem_flags);
```

The first parameter, key, is an integral value used to allow unrelated process to access the same semaphore. The semaphore key is used only by semget. All others use the identifier return by the semget(). There is a special key value IPC_PRIVATE whichallows to create the semaphore and to be accessed only by the creating process.

The second parameter is the number of semaphores required, it is almost always 1. The third parameter is the set of flags. The nine bits are the permissions for the semaphore.

On success it will return a positive value which is the identifier used by the other semaphore functions. On error, it returns -1.

Changing the value:
The function semop() is used for changing the value of the semaphore.

```
#include<sys/sem.h>
int semop(int sem_id,struct sembuf
*sem_ops,size_t num-_sem_ops);
```

The first parameter is the shmid is the identifier returned by the semget().

The second parameter is the pointer to an array of structure. The structure may contain at least the following members:

```
struct   sembuf{
short sem_num;
short    sem_op;
short sem_flg;
}
```

The first member is the semaphore number, usually 0 unless it is an array of semaphore. The sem_op is the value by which the semaphore should be changed. Generally it takes -1,which is operation to wait for a semaphore and +1, which is the operation to signal the availability of semaphore.
The third parameter, is the flag which is usually set to SET_UNDO. If the process terminates without releasing the semaphore, this allows to release it automatically.

Controlling the semaphore:
The semctl() function allows direct control of semaphore information.

```
#include<sys/sem.h>
int   semctl(int   sem_id,int   sem_num,
int command,.../*union semun arg */);
```

The third parameter is the command, which defines the action to be taken. There are two common values:

1.) **SETVAL**: Used for initializing a semaphore to a known value.
2.) **IPC_RMID**:Deletes the semaphore identifier.

File locking with semaphores

```
//sem.c
#include     <unistd.h>
#include      <stdlib.h>
#include       <stdio.h>
#include   <sys/sem.h>
```

```
#include<sys/ipc.h>
#include<sys/types.h>
union semun
{
int val;
struct      semid_ds      *buf;
unsigned short *array;
};

static void    del_semvalue(void);
static int     set_semvalue(void);
static int     semaphore_p(void);
static int     semaphore_v(void);
static int     sem_id;
//--------------------------------
static int set_semvalue()
{
union semun sem_union;
sem_union.val = 1;
if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
return(1);
}
//------------------------------------------

static void del_semvalue()
{
union semun sem_union;
if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
fprintf(stderr, "Failed to delete semaphore\n");
}
//-------------------------------------------
static int semaphore_p()
{
struct            sembuf            sem_b;
sem_b.sem_num           =           0;
sem_b.sem_op   =   -1;   /*   P()   */
sem_b.sem_flg = SEM_UNDO;
if (semop(sem_id, &sem_b, 1) == -1)
{
fprintf(stderr, "semaphore_p failed\n");
return(0);
}
return(1);
}
//-----------------------------------------------
static int semaphore_v()
{
struct            sembuf            sem_b;
sem_b.sem_num           =           0;
sem_b.sem_op   =   1;   /*   V()   */
sem_b.sem_flg = SEM_UNDO;
```

```
if (semop(sem_id, &sem_b, 1) == -1) {
fprintf(stderr, "semaphore_v failed\n");
return(0);
}
return(1);
}
int main(int argc, char *argv[])
{
int i;
int     pause_time;
char op_char = 'O';
srand((unsigned int)getpid());
sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
if (argc > 1)
{
if (!set_semvalue())
{
fprintf(stderr, "Failed to initialize semaphore\n");
exit(EXIT_FAILURE);
}
op_char = 'X';
sleep(2);
}

for(i = 0; i < 10; i++)
{
if     (!semaphore_p())     exit(EXIT_FAILURE);
printf("%c", op_char);
fflush(stdout);
pause_time = rand() % 3;
sleep(pause_time);
printf("%c", op_char);fflush(stdout);

if (!semaphore_v()) exit(EXIT_FAILURE);
pause_time     =     rand()     %     2;
sleep(pause_time);
}
printf("\n%d - finished\n", getpid());
if (argc > 1)
{
sleep(10);
del_semvalue();
}
exit(EXIT_SUCCESS);
}
```

The function *set_semvalue()* initializes the semaphore using the *SETVAL* command in *semctl()*function. But this is to be done before the usage of semaphore.

The function *del_semvalue()* is used to delete the semaphore by using the command *IPC_RMID* in the *semctl()* function. The function *semaphore_p()* changes the

semaphore value to -1, which is used to make the process to wait.

In the function *semaphore_v()*,the *semop* member of the structure *sembuf* is set to 1.By this the semphore becomes available for the other processes because it is released.

UNIT V

Shared Memory:

Shared memory is a highly efficient way of data sharing between the running programs. It allows two unrelated processes to access the same logical memory. It is the fastest form of IPC because all processes share the same piece of memory. It also avoidscopyingdataunnecessarily.

As kernel does not synchronize the processes, it should be handled by the user. Semaphore can also be used to synchronize the access to shared memory.

Usageofsharedmemory:

To use the shared memory, first of all one process should allocate the segment, and then each process desiring to access the segment should attach the segment. After accessing the segment, each process should detach it. It is also necessary to deallocate thesegmentwithoutfail.

Allocating the shared memory causes virtual pages to be created. It is important to note that allocating the existing segment would not create new pages, but will return theidentifierfortheexistingpages.

All the shared memory segments are allocated as the integral multiples of the system's page size, which is the number of bytes in a page of memory.

Unix kernel support for shared memory

- There is a shared memory table in the kernel address space that keeps track of all shared memory regions created in the system.
- Each entry of the tables store the following data:

1. Name
2. Creator user ID and group ID.
3. Assigned owner user ID and group ID.
4. Read-write access permission of the region.
5. The time when the last process attached to the region.
6. The time when the last process detached from the region.
7. The time when the last process changed control data of the region.
8. The size, in no. of bytes of the region.

UNIX APIs for shared memory shmget

- Open and create a shared memory.
- Function prototype:

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

#include<sys/ipc.h>

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

```
int shmget ( key_t key, int size, int flag );
```

- Function returns a positive descriptor if it succeeds or -1 if it fails.

Shmat

- Attach a shared memory to a process virtual address space.
- Function prototype:

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

#include<sys/ipc.h>

#include<sys/shm.h>

void * shmat ( int shmid, void *addr, int flag );
```

- Function returns the mapped virtual address of he shared memory if it succeeds or -1 if it fails.

Shmdt

- Detach a shared memory from the process virtual address space.
- Function prototype:

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

#include<sys/ipc.h>

#include<sys/shm.h>

int shmdt ( void *addr );
```

- Function returns 0 if it succeeds or -1 if it fails.

Shmctl

- Query or change control data of a shared memory or delete the memory.
- Function prototype:

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

#include<sys/ipc.h>

#include<sys/shm.h>

int shmctl ( int shmid, int cmd, struct shmid_ds *buf );
```

- Function returns 0 if it succeeds or -1 if it fails.

Shared memory Example

//shmry1.c

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<sys/shm.h>

#define TEXT_SZ 2048

struct shared_use_st
{
int written_by_you;
char some_text[TEXT_SZ];
};

int main()
{
int running = 1;
void *shared_memory = (void *)0;
struct shared_use_st *shared_stuff;
int shmid;
srand( (unsigned int)getpid() );
shmid = shmget( (key_t)1234,
sizeof(struct     shared_use_st),
0666 |IPC_CREAT );

if (shmid == -1)
{
fprintf(stderr, "shmget failed\n");
exit(EXIT_FAILURE);
}
shared_memory = shmat(shmid,(void *)0, 0);
if (shared_memory == (void *)-1)
{
fprintf(stderr,        "shmat        failed\n");
exit(EXIT_FAILURE);
}

printf("Memory Attached at %x\n",
(int)shared_memory);

 shared_stuff = (struct shared_use_st *)
shared_memory;
shared_stuff->written_by_you      =      0;
while(running)

{
```

```
if(shared_stuff->written_by_you)
{

printf("You  Wrote:  %s",
shared_stuff->some_text);

sleep( rand() %4 );
shared_stuff->written_by_you = 0;

if (strncmp(shared_stuff->some_text,
"end", 3)== 0)
{
running = 0;
}
}
}
if (shmdt(shared_memory) == -1)

{
fprintf(stderr, "shmdt failed\n");
exit(EXIT_FAILURE);
}
if (shmctl(shmid, IPC_RMID, 0) == -1)
{
fprintf(stderr, "failed to delete\n");
exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);

}
```

//shmry2.c

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

#include<sys/shm.h>

#define TEXT_SZ 2048
struct shared_use_st
{
int written_by_you;
char some_text[TEXT_SZ];
};

int main()
{
int running =1
```

```
void *shared_memory = (void *)0;
struct shared_use_st *shared_stuff;
char buffer[BUFSIZ];
int shmid;
shmid =shmget( (key_t)1234, sizeof(struct
shared_use_st),
0666 | IPC_CREAT);
if (shmid == -1)
{
fprintf(stderr, "shmget failed\n");
exit(EXIT_FAILURE);
}

shared_memory=shmat(shmid,
(void *)0, 0);
if (shared_memory == (void *)-1)
{
fprintf(stderr, "shmat failed\n");
exit(EXIT_FAILURE);
}
printf("Memory Attached at %x\n", (int) shared_memory);
shared_stuff = (struct shared_use_st *)shared_memory;
while(running)
{
while(shared_stuff->written_by_you== 1)
{
sleep(1);
printf("waiting for client....\n");
}
printf("Enter Some Text: ");
fgets (buffer, BUFSIZ, stdin);
strncpy(shared_stuff->some_text, buffer,
TEXT_SZ);
shared_stuff->written_by_you = 1;
if(strncmp(buffer, "end", 3) == 0)
{
running = 0;
}
}
if (shmdt(shared_memory) == -1)
{
fprintf(stderr, "shmdt failed\n");
exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);

}
```

The *shmry1.c* program will create the segment using *shmget()* function and returns the identifier shmid. Then that segment is attached to its address space using *shmat()* function.

The structure *share_use_st* consists of a flag *written_by_you* is set to 1 when data is available. When it is set, program reads the text, prints it and clears it to show it has read the data. The string end is used to quit from the loop. After this the segment is detached and deleted.

The *shmry2.c* program gets and attaches to the same memory segment. This is possible with the help of same key value *1234* used in the shmget() function. If the written_by_you text is set, the process will wait until the previous process reads it. When the flag is cleared, the data is written and sets the flag. This program too will use the string "*end*" to terminate. Then the segment is detached.

Sockets

A *socket* is a bidirectional communication device that can be used to communicate withanother process on the same machine or with a process running on other machines.Sockets are the only interprocess communication we'll discuss in this chapter thatpermit communication between processes on different computers. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

For example, you can obtain the WWW page from a Web server using theTelnet program because they both use sockets for network communications.To open a connection to a WWW server at www.codesourcery.com, use telnet www.codesourcery.com 80.The magic constant 80 specifies a connection to the Web server programming running www.codesourcery.com instead of some other process.Try typing GET / after the connection is established.This sends a message through the socket to the Web server, which replies by sending the home page's HTML source and then closing the connection—for example:

```
% telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
Escape character is '^]'.
GET /
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
...
```

3. Note that only Windows NT can create a named pipe;Windows 9*x* programs can form only client connections.
4. Usually, you'd use telnet to connect a Telnet server for remote logins. But you can also use telnet to connect to a server of a different kind and then type comments directly at it.

Introduction to Berkeley sockets

Berkeley sockets (or BSD sockets) is a computing library with an application programming interface (API) for internet sockets and Unix domain sockets, used for inter-process communication (IPC).

This list is a summary of functions or methods provided by the Berkeley sockets API library:

- socket() creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
- bind() is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
- listen() is used on the server side, and causes a bound TCP socket to enter listening state.
- connect() is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
- send() and recv(), or write() and read(), or sendto() and recvfrom(), are used for sending and receiving data to/from a remote socket.
- close() causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
- gethostbyname() and gethostbyaddr() are used to resolve host names and addresses. IPv4 only.
- select() is used to pend, waiting for one or more of a provided list of sockets to be ready to read, ready to write, or that have errors.
- poll() is used to check on the state of a socket in a set of sockets. The set can be tested to see if any socket can be written to, read from or if an error occurred.
- getsockopt() is used to retrieve the current value of a particular socket option for the specified socket.
- setsockopt() is used to set a particular socket option for the specified socket.

IPC over a network Socket Concepts

When you create a socket, you must specify three parameters:

- communication style,
- namespace,
- protocol.

A communication style controls how the socket treats transmitted data and specifies the number of communication partners.When data is sent through a socket, it is ackaged into chunks called *packets*.The communication style determines how these packets are handled and how they are addressed from the sender to the receiver.

Connection styles guarantee delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender.
A connection-style socket is like a telephone call: The addresses of the sender and receiver are fixed at the beginning of the communication when the connection is established.

Datagram styles do not guarantee delivery or arrival order. Packets may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered.The system

guarantees only “best effort,” so packets may disappear or arrive in a different order than shipping.
A datagram-style socket behaves more like postal mail.The sender specifies the receiver’s address for each individual message.

A socket namespace specifies how *socket addresses* are written. A socket address identifies one end of a socket connection. For example, socket addresses in the “local namespace” are ordinary filenames. In “Internet namespace,” a socket address is composed of the Internet address (also known as an *Internet Protocol address* or *IP address*) of a host attached to the network and a port number.The port number distinguishes among multiple sockets on the same host.
A protocol specifies how data is transmitted. Some protocols are TCP/IP, the primary networking protocols used by the Internet; the AppleTalk network protocol; and the UNIX local communication protocol. Not all combinations of styles, namespaces,and protocols are supported.

Client-server datagram socket — example

To experiment with datagram sockets in the UNIX domain we will write a client/server application where:

- the client takes a number of arguments on its command line and send them to the server using separate datagrams
- for each datagram received, the server converts it to uppercase and send it back to the client
- the client prints server replies to standard output

For this to work we will need to bind all involved sockets to pathnames.

Client-server datagram socket example — protocol

```
#include <ctype .h>
#include <sys/un.h>
#include <sys/socket .h>
#include <unistd .h>
#include " helpers .h"
#define SRV_SOCK_PATH " /tmp/uc_srv_socket "
#define CLI_SOCK_PATH " /tmp/ uc_cl i_socket .%ld "
#define MSG_LEN 10
#include "uc�proto .h"
int main( int argc , char *argv [ ] ) {
struct sockaddr_un srv_addr , cl i_addr ;
int srv_fd , i ;
ssize_t bytes ;
socklen_t len ;
char buf [MSG_LEN] ;
i f ( ( srv_fd = socket (AF_UNIX , SOCK_DGRAM, 0) ) < 0)
err_sys ( " socket error " ) ;
memset(&srv_addr , 0, sizeof ( struct sockaddr_un ) ) ;
srv_addr . sun_family = AF_UNIX ;
strncpy ( srv_addr . sun_path , SRV_SOCK_PATH,
sizeof ( srv_addr . sun_path ) �1) ;
i f ( access ( srv_addr . sun_path , F_OK) == 0)
unlink ( srv_addr . sun_path ) ;
i f ( bind ( srv_fd , ( struct sockaddr * ) &srv_addr ,
```

```
sizeof ( struct sockaddr_un ) ) < 0)
err_sys ( " bind error " ) ;
for ( ; ; ) {
len = sizeof ( struct sockaddr_un ) ;
i f ( ( bytes = recvfrom( srv_fd , buf , MSG_LEN, 0,
( struct sockaddr * ) &cl i_addr , &len ) ) < 1)
err_sys ( " recvfrom error " ) ;
pr int f ( " server received %ld bytes from %s\n" ,
( long) bytes , cl i_addr . sun_path ) ;
for ( i = 0; i < bytes ; i ++)
buf [ i ] = toupper ( ( unsigned char ) buf [ i ] ) ;
i f ( sendto ( srv_fd , buf , bytes , 0,
( struct sockaddr * ) &cl i_addr , len ) != bytes )
err_sys ( " sendto error " ) ;
}
}
#include "uc�proto .h"
int main( int argc , char *argv [ ] ) {
struct sockaddr_un srv_addr , cl i_addr ;
int srv_fd , i ;
size_t     len     ;
ssize_t bytes ;
char resp [MSG_LEN] ;
i f ( argc < 2)
er r_qui t ( "Usage : uc�c l ient MSG. . . " ) ;
i f ( ( srv_fd = socket (AF_UNIX , SOCK_DGRAM, 0) ) < 0)
err_sys ( " socket error " ) ;
memset(&cl i_addr , 0, sizeof ( struct sockaddr_un ) ) ;
cl i_addr . sun_family = AF_UNIX ;
snpr int f ( cl i_addr . sun_path , sizeof ( cl i_addr . sun_path ) ,
CLI_SOCK_PATH, ( long) getpid ( ) ) ;
i f ( bind ( srv_fd , ( struct sockaddr * ) &cl i_addr ,
sizeof ( struct sockaddr_un ) ) == �1)
err_sys ( " bind error " ) ;
```

Notes:
the server is persistent and processes one datagram at a time, no matter the client rocess, i.e. there is no notion of connection messages larger than 10 bytes are silently truncated

Socket address structures(UNIX domain & Internet domain) UNIX

domain Sockets:

We now want to give an example of stream sockets. To do so, we can longer remain in the abstract of general sockets, but we need to pick a domain. We pick the UNIX domain. In the UNIX domain, addresses are pathnames. The corresponding Cstructure is sockaddr_un:

```
struct sockaddr_un {
sa_fami ly_t sun_family ; /* = AF_UNIX */
```

char sun_path[108] ; /* *socket pathname,*
NULL�terminated */
}
The field sun_path contains a regular pathname, pointing to a special file of type socket (. pipe) which will be created at bind time.
During communication the file will have no content, it is used only as a *rendez-vous* point between processes.

Internet-Domain Sockets

UNIX-domain sockets can be used only for communication between two processes on the same computer. *Internet-domain sockets*, on the other hand, may be used to connect processes on different machines connected by a network.
Sockets connecting processes through the Internet use the Internet namespace represented by PF_INET.The most common protocols are TCP/IP.The *Internet Protocol (IP)*, a low-level protocol, moves packets through the Internet, splitting and rejoining the packets, if necessary. It guarantees only "best-effort" delivery, so packets may vanish or be reordered during transport. Every participating computer is specified using a unique IP number.The *Transmission Control Protocol (TCP)*, layered on top of IP, provides reliable connection-ordered transport. It permits telephone-like connections to be established between computers and ensures that data is delivered reliably and inorder.

DNS Names

Because it is easier to remember names than numbers, the *Domain Name Service (DNS)* associates names such as www.codesourcery.com with computers' unique IP numbers. DNS is implemented by a worldwide hierarchy of name servers, but you don't need to understand DNS protocols to use Internet host names in your programs.
Internet socket addresses contain two parts: a machine and a port number.This information is stored in a struct sockaddr_in variable. Set the sin_family field to AF_INET to indicate that this is an Internet namespace address.The sin_addr field stores the Internet address of the desired machine as a 32-bit integer IP number.A *port number* distinguishes a given machine's different sockets. Because different machines store multibyte values in different byte orders, use htons to convert the port number to
network byte order. See the man page for ip for more information.To convert human-readable hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such as www.codesourcery.com) into 32-bit IP numbers, you can use gethostbyname.This returns a pointer to the struct hostent structure; the h_addr field contains the host's IP number.

System Calls

Sockets are more flexible than previously discussed communication techniques.These are the system calls involving sockets:

socket—Creates a socket
closes—Destroys a socket
connect—Creates a connection between two sockets
bind—Labels a server socket with an address

listen—Configures a socket to accept conditions
accept—Accepts a connection and creates a new socket for the connection
Sockets are represented by file descriptors.

Creating and Destroying Sockets

Sockets are IPC objects that allow to exchange data between processes running:
either on the same machine (*host*), or on different ones over a network.
The UNIX socket API first appeared in 1983 with BSD 4.2. It has been finally standardized for the first time in POSIX.1g (2000), but has been ubiquitous to every UNIX implementation since the 80s.
The socket API is best discussed in a network programming course,which this one is *not*. We will only address enough general socketconcepts to describe how to use a specific socket family: UNIXdomain sockets.

Connection Oriented Protocol

Server

socket()

bind()

listen()

accept()

Client

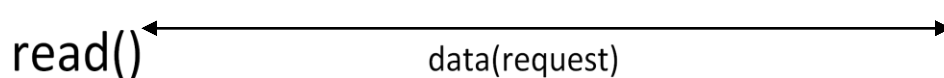
blocks until connection from client

socket()

connection establishment

connect()

read()



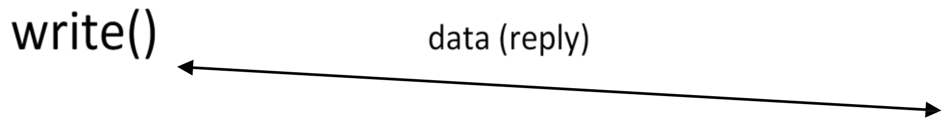
data(request)

write()

process request

write()

data (reply)



read()

Connectionless Protocol

Server

socket()

bindConnectionless Protocol ()

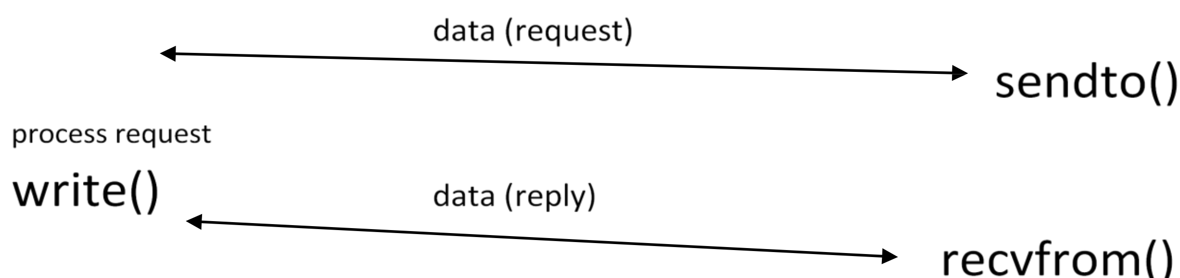
recvfrom()

Client

blocks until connection from client

socket()

bind()



Client-server setup

Let’s consider a typical client-server application scenario — no matter if they are located on the same or different hosts.
Sockets are used as follows:
each application: create a socket
idea: communication between the two applications will flow through an imaginary “pipe” that *will* connect the two sockets together
server: bind its socket to a well-known address
we have done the same to set up *rendez-vous* points for other IPC objects.
e.g. FIFOs
client: locate server socket (via its well-known address) and “initiate communication”1 with the server.

Socket options:

In order to tell the socket to get the information about the packet destination, we should call setsockopt().

setsockopt()* and *getsockopt() - set and get options on a socket. Both methods return 0 on success and -1 on error.

Prototype: int setsockopt(int sockfd, **int level**, int optname,...
There are two levels of socket options:
To manipulate options at the sockets API level:
SOL_SOCKET
To manipulate options at a protocol level, that protocol number should be used;

for example, for UDP it is IPPROTO_UDP or SOL_UDP (both are equal 17) ; see include/linux/in.h and include/linux/socket.h

- SOL_IP is 0.
- There are currently 19 Linux socket options and one another on option for BSD compatibility.
- There is an option called IP_PKTINFO.

We will set the IP_PKTINFO option on a socket in the following example.

```
// from /usr/include/bits/in.h
#define IP_PKTINFO 8 /* bool */
/* Structure used for IP_PKTINFO. */
struct in_pktinfo
{
int ipi_ifindex; /* Interface index */
struct in_addr ipi_spec_dst; /* Routing destination address */
struct in_addr ipi_addr; /* Header destination address */
};
const int on = 1;
sockfd = socket(AF_INET, SOCK_DGRAM,0);
if (setsockopt(sockfd, SOL_IP, IP_PKTINFO, &on,
sizeof(on))<0)
perror("setsockopt");
...
...
...
```

When calling recvmsg(), we will parse the msghr like this:

```
for (cmptr=CMSG_FIRSTHDR(&msg); cmptr!=NULL;
cmptr=CMSG_NXTHDR(&msg,cmptr))
{
if (cmptr->cmsg_level == SOL_IP && cmptr->cmsg_type ==IP_PKTINFO)
{
pktinfo = (struct in_pktinfo*)CMSG_DATA(cmptr);
printf("destination=%s\n", inet_ntop(AF_INET, &pktinfo->ipi_addr,str, sizeof(str)));
}
}
```

In the kernel, this calls *ip_cmsg_recv()* in net/ipv4/ip_sockglue.c. (which eventually calls *ip_cmsg_recv_pktinfo()*).

- You can in this way retrieve other fields of the ip header:

For getting the TTL:

- setsockopt(sockfd, SOL_IP, IP_RECVTTL, &on, sizeof(on))<0).
- But: cmsg_type == IP_TTL.

For getting ip_options:

- setsockopt() with IP_OPTIONS.

fcntl system calls

The **fcntl** system call provides further ways to manipulate low level file descriptors.

```
#include <fcntl.h>

int fcntl(int fildes, int cmd);
int fcntl(int fildes, int cmd, long arg);
```

It can perform miscellaneous operations on open file descriptors.

The call,

```
fcntl(fildes, F_DUPFD, newfd);
```

returns a new file descriptor with a numerical value equal to or greater than the integer **newfd**.

The call,

```
fcntl(fildes, F_GETFD)
```

returns the file descriptor flags as defined in **fcntl.h**.

The call,

```
fcntl(fildes, F_SETFD, flags)
```

is used to set the file descriptor flags, usually just **FD_CLOEXEC**.

The calls,

```
fcntl(fildes, F_GETFL)
fcntl(fildes, F_SETFL, flags)
```

respectively get and set the file status flags and access modes.

Comparision of IPC mechanisms.

IPC mechanisms are mianly 5 types
1.pipes:it is related data only send from one pipe output is giving to another pipe input to share resouses pipe are used drawback:itis only related process only communicated
2.message queues:message queues are un related process are also communicate with message queues.
3.sockets:sockets also ipc it is comunicate clients and server

with socket system calls connection oriented and connection less also
4.PIPE: Only two related (eg: parent & child) processess can be communicated. Data reading would be first in first out manner.
Named PIPE or FIFO : Only two processes (can be related or unrelated) can communicate. Data read from FIFO is first in first out manner.
5.Message Queues: Any number of processes can read/write from/to the queue. Data can be read selectively. (need not be in FIFO manner)
6.Shared Memory: Part of process's memory is shared to other processes. other processes can read or write into this shared memory area based on the permissions. Accessing Shared memory is faster than any other IPC mechanism as this does not involve any kernel level switching(Shared memory resides on user memory area).
7.Semaphore: Semaphores are used for process synchronisation. This can't be used for bulk data transfer between processes.