Linux Programming

"Unit-I - Linux Utilities"

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.

A distribution oriented toward desktop use will typically include the X Window System and an accompanying desktop environment such as GNOME or KDE Plasma. Some such distributions may include a less resource intensive desktop such as LXDE or Xfce for use on older or less powerful computers. A distribution intended to run as a server may omit all graphical environments from the standard install and instead include other software such as the Apache HTTP Server and an SSH server such as OpenSSH. Because Linux is freely redistributable, anyone may create a distribution for any intended use. Applications commonly used with desktop Linux systems include the Mozilla Firefox web browser, the LibreOffice office application suite, and the GIMP image editor.

Since the main supporting user space system tools and libraries originated in the GNU Project, initiated in 1983 by Richard Stallman, the Free Software Foundation prefers the name *GNU/Linux*.

Linux Programming

History

Unix

The Unix operating system was conceived and implemented in 1969 at AT&T's Bell Laboratories in the United States by Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna. It was first released in 1971 and was initially entirely written in assembly language, a common practice at the time. Later, in a key pioneering approach in 1973, Unix was re-written in the programming language C by Dennis Ritchie (with exceptions to the kernel and I/O). The availability of an operating system written in a high-level language allowed easier portability to different computer platforms.

Today, Linux systems are used in every domain, from embedded systems to supercomputers, and have secured a place in server installations often using the popular LAMP application stack. Use of Linux distributions in home and enterprise desktops has been growing. They have also gained popularity with various local and national governments. The federal government of Brazil is well known for its support for Linux. News of the Russian military creating its own Linux distribution has also surfaced, and has come to fruition as the G.H.ost Project. The Indian state of Kerala has gone to the extent of mandating that all state high schools run Linux on their computers.

Design

A Linux-based system is a modular Unix-like operating system. It derives much of its basic design from principles established in Unix during the 1970s and 1980s. Such a system uses a monolithic kernel, the Linux kernel, which handles process control, networking, and peripheral and file system access. Device drivers are either integrated directly with the kernel or added as modules loaded while the system is running.

Separate projects that interface with the kernel provide much of the system's higher-level functionality. The GNU userland is an important part of most Linux-based systems, providing the most common implementation of the C library, a popular shell, and many of the common Unix tools which carry out many basic operating system tasks. The graphical user interface (or GUI) used by most Linux systems is built on top of an implementation of the X Window System.

Linux Programming

Programming on Linux

Most Linux distributions support dozens of programming languages. The original development tools used for building both Linux applications and operating system programs are found within the GNU toolchain, which includes the GNU Compiler Collection (GCC) and the GNU build system. Amongst others, GCC provides compilers for Ada, C, C++, Java, and Fortran. First released in 2003, the Low Level Virtual Machine project provides an alternative open-source compiler for many languages. Proprietary compilers for Linux include the Intel C++ Compiler, Sun Studio, and IBM XL C/C++ Compiler. BASIC in the form of Visual Basic is supported in such forms as Gambas, FreeBASIC, and XBasic.

Most distributions also include support for PHP, Perl, Ruby, Python and other dynamic languages. While not as common, Linux also supports C# (via Mono), Vala, and Scheme. A number of Java Virtual Machines and development kits run on Linux, including the original Sun Microsystems JVM (HotSpot), and IBM's J2SE RE, as well as many open-source projects like Kaffe and JikesRVM.

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

Linux Programming

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. **Multitasking:** 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.

The   difference   between   Linux   and   UNIX   operating   systems?

UNIX is copyrighted name only big companies are allowed to use the UNIX copyright and name, so IBM AIX and Sun Solaris and HP-UX all are UNIX operating systems. The Open Group holds the UNIX trademark in trust for the industry, and manages the UNIX trademark licensing program.

Most UNIX systems are commercial in nature.

Linux is a UNIX Clone

But if you consider Portable Operating System Interface (POSIX) standards then Linux can be considered as UNIX. To quote from Official Linux kernel README file:

Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX compliance.

However, "Open Group" do not approve of the construction "Unix-like", and consider it misuse of their UNIX trademark.

Linux Programming

Linux Is Just a Kernel

Linux is just a kernel. All Linux distributions includes GUI system + GNU utilities (such as cp, mv, ls,date, bash etc) + installation & management tools + GNU c/c++ Compilers + Editors (vi) + and various applications (such as OpenOffice, Firefox). However, most UNIX operating systems are considered as a complete operating system as everything come from a single source or vendor.

As I said earlier Linux is just a kernel and Linux distribution makes it complete usable operating systems by adding various applications. Most UNIX operating systems comes with A-Z programs such as editor, compilers etc. For example HP-UX or Solaris comes with A-Z programs.

License and cost

Linux is Free (as in beer [freedom]). You can download it from the Internet or redistribute it under GNU licenses. You will see the best community support for Linux. Most UNIX like operating systems are not free (but this is changing fast, for example OpenSolaris UNIX). However, some Linux distributions such as Redhat / Novell provides additional Linux support, consultancy, bug fixing, and training for additional fees.

User-Friendly

Linux is considered as most user friendly UNIX like operating systems. It makes it easy to install sound card, flash players, and other desktop goodies. However, Apple OS X is most popular UNIX operating system for desktop usage.

Security Firewall Software

Linux comes with open source netfilter/iptables based firewall tool to protect your server and desktop from the crackers and hackers. UNIX operating systems comes with its own firewall product (for example Solaris UNIX comes with ipfilter based firewall) or you need to purchase a 3rd party software such as Checkpoint UNIX firewall.

Backup and Recovery Software

UNIX and Linux comes with different set of tools for backing up data to tape and other backup media. However, both of them share some common tools such as tar, dump/restore, and cpio etc.

File Systems

Linux Programming

- Linux by default supports and use ext3 or ext4 file systems.
- UNIX comes with various file systems such as jfs, gpfs (AIX), jfs, gpfs (HP-UX), jfs, gpfs (Solaris).

System Administration Tools

1. UNIX comes with its own tools such as SAM on HP-UX.
2. Suse Linux comes with Yast
3. Redhat Linux comes with its own gui tools called redhat-config-*.

However, editing text config file and typing commands are most popular options for sys admin work under UNIX and Linux.

System Startup Scripts

Almost every version of UNIX and Linux comes with system initialization script but they are located in different directories:

1. HP-UX - /sbin/init.d
2. AIX - /etc/rc.d/init.d
3. Linux - /etc/init.d

End User Perspective

The differences are not that big for the average end user. They will use the same shell (e.g. bash or ksh) and other development tools such as Perl or Eclipse development tool.

System Administrator Perspective

Again, the differences are not that big for the system administrator. However, you may notice various differences while performing the following operations:

1. Software installation procedure
2. Hardware device names
3. Various admin commands or utilities
4. Software RAID devices and mirroring
5. Logical volume management
6. Package management
7. Patch management

Linux Programming

UNIX Operating System Names

A few popular names:

1. HP-UX
2. IBM AIX
3. Sun Solairs
4. Mac OS X
5. IRIX

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

10 fundamental differences between Linux and Windows

#1: Full access vs. no access

Having access to the source code is probably the single most significant difference between Linux and Windows. The fact that Linux belongs to the GNU Public License ensures that users (of all sorts) can access (and alter) the code to the very kernel that serves as the foundation of the Linux operating system. You want to peer at the Windows code? Good luck. Unless you are a member of a very select (and elite, to many) group, you will never lay eyes on code making up the Windows operating system.

Linux Programming

You can look at this from both sides of the fence. Some say giving the public access to the code opens the operating system (and the software that runs on top of it) to malicious developers who will take advantage of any weakness they find. Others say that having full access to the code helps bring about faster improvements and bug fixes to keep those malicious developers from being able to bring the system down. I have, on occasion, dipped into the code of one Linux application or another, and when all was said and done, was happy with the results. Could I have done that with a closed-source Windows application? No.

#2: Licensing freedom vs. licensing restrictions

Along with access comes the difference between the licenses. I'm sure that every IT professional could go on and on about licensing of PC software. But let's just look at the key aspect of the licenses (without getting into legalese). With a Linux GPL-licensed operating system, you are free to modify that software and use and even republish or sell it (so long as you make the code available). Also, with the GPL, you can download a single copy of a Linux distribution (or application) and install it on as many machines as you like. With the Microsoft license, you can do none of the above. You are bound to the number of licenses you purchase, so if you purchase 10 licenses, you can legally install that operating system (or application) on only 10 machines.

#3: Online peer support vs. paid help-desk support

This is one issue where most companies turn their backs on Linux. But it's really not necessary. With Linux, you have the support of a huge community via forums, online search, and plenty of dedicated Web sites. And of course, if you feel the need, you can purchase support contracts from some of the bigger Linux companies (Red Hat and Novell for instance).

However, when you use the peer support inherent in Linux, you do fall prey to time. You could have an issue with something, send out e-mail to a mailing list or post on a forum, and within 10 minutes be flooded with suggestions. Or these suggestions could take hours of days to come in. It seems all up to chance sometimes. Still, generally speaking, most problems with Linux have been encountered and documented. So chances are good you'll find your solution fairly quickly.

On the other side of the coin is support for Windows. Yes, you can go the same route with Microsoft and depend upon your peers for solutions. There are just as many help sites/lists/forums for Windows as there are for Linux. And you can purchase support from Microsoft itself. Most corporate higher-ups easily fall victim to the safety net that having a support contract brings. But most higher-ups haven't had to depend up on said support contract. Of the various people I know who have used either a Linux paid support contract or a Microsoft paid support contract, I can't say one was more pleased than the other. This of course begs the question "Why do so many say that Microsoft support is superior to Linux paid support?"

#4: Full vs. partial hardware support

Linux Programming

One issue that is slowly becoming nonexistent is hardware support. Years ago, if you wanted to install Linux on a machine you had to make sure you hand-picked each piece of hardware or your installation would not work 100 percent. I can remember, back in 1997-ish, trying to figure out why I couldn't get Caldera Linux or Red Hat Linux to see my modem. After much looking around, I found I was the proud owner of a Winmodem. So I had to go out and purchase a US Robotics external modem because that was the one modem I *knew* would work. This is not so much the case now. You can grab a PC (or laptop) and most likely get one or more Linux distributions to install and work nearly 100 percent. But there are still some exceptions. For instance, hibernate/suspend remains a problem with many laptops, although it has come a long way.

With Windows, you know that most every piece of hardware will work with the operating system. Of course, there are times (and I have experienced this over and over) when you will wind up spending much of the day searching for the correct drivers for that piece of hardware you no longer have the install disk for. But you can go out and buy that 10-cent Ethernet card and know it'll work on your machine (so long as you have, or can find, the drivers). You also can rest assured that when you purchase that insanely powerful graphics card, you will probably be able to take full advantage of its power.

#5: Command line vs. no command line

No matter how far the Linux operating system has come and how amazing the desktop environment becomes, the command line will always be an invaluable tool for administration purposes. Nothing will ever replace my favorite text-based editor, ssh, and any given command-line tool. I can't imagine administering a Linux machine without the command line. But for the end user — not so much. You could use a Linux machine for years and never touch the command line. Same with Windows. You can still use the command line with Windows, but not nearly to the extent as with Linux. And Microsoft tends to obfuscate the command prompt from users. Without going to Run and entering cmd (or command, or whichever it is these days), the user won't even know the command-line tool exists. And if a user does get the Windows command line up and running, how useful is it really?

#6: Centralized vs. noncentralized application installation

The heading for this point might have thrown you for a loop. But let's think about this for a second. With Linux you have (with nearly every distribution) a centralized location where you can search for, add, or remove software. I'm talking about package management systems, such as Synaptic. With Synaptic, you can open up one tool, search for an application (or group of applications), and install that application without having to do any Web searching (or purchasing).

Windows has nothing like this. With Windows, you must know where to find the software you want to install, download the software (or put the CD into your machine), and run setup.exe or

Linux Programming

install.exe with a simple double-click. For many years, it was thought that installing applications on Windows was far easier than on Linux. And for many years, that thought was right on target. Not so much now. Installation under Linux is simple, painless, and centralized.

#7: Flexibility vs. rigidity

I always compare Linux (especially the desktop) and Windows to a room where the floor and ceiling are either movable or not. With Linux, you have a room where the floor and ceiling can be raised or lowered, at will, as high or low as you want to make them. With Windows, that floor and ceiling are immovable. You can't go further than Microsoft has deemed it necessary to go.

Take, for instance, the desktop. Unless you are willing to pay for and install a third-party application that can alter the desktop appearance, with Windows you are stuck with what Microsoft has declared is the ideal desktop for you. With Linux, you can pretty much make your desktop look and feel exactly how you want/need. You can have as much or as little on your desktop as you want. From simple flat Fluxbox to a full-blown 3D Compiz experience, the Linux desktop is as flexible an environment as there is on a computer.

#8: Fanboys vs. corporate types

I wanted to add this because even though Linux has reached well beyond its school-project roots, Linux users tend to be soapbox-dwelling fanatics who are quick to spout off about why you should be choosing Linux over Windows. I am guilty of this on a daily basis (I try hard to recruit new fanboys/girls), and it's a badge I wear proudly. Of course, this is seen as less than professional by some. After all, why would something worthy of a corporate environment have or need cheerleaders? Shouldn't the software sell itself? Because of the open source nature of Linux, it has to make do without the help of the marketing budgets and deep pockets of Microsoft. With that comes the need for fans to help spread the word. And word of mouth is the best friend of Linux.

Some see the fanaticism as the same college-level hoorah that keeps Linux in the basements for LUG meetings and science projects. But I beg to differ. Another company, thanks to the phenomenon of a simple music player and phone, has fallen into the same fanboy fanaticism, and yet that company's image has not been besmirched because of that fanaticism. Windows does not have these same fans. Instead, Windows has a league of paper-certified administrators who believe the hype when they hear the misrepresented market share numbers reassuring them they will be employable until the end of time.

#9: Automated vs. nonautomated removable media

I remember the days of old when you had to mount your floppy to use it and unmount it to remove it. Well, those times are drawing to a close — but not completely. One issue that plagues new Linux users is how removable media is used. The idea of having to manually "mount" a CD

Linux Programming

drive to access the contents of a CD is completely foreign to new users. There is a reason this is the way it is. Because Linux has always been a multiuser platform, it was thought that forcing a user to mount a media to use it would keep the user's files from being overwritten by another user. Think about it: On a multiuser system, if everyone had instant access to a disk that had been inserted, what would stop them from deleting or overwriting a file you had just added to the media? Things have now evolved to the point where Linux subsystems are set up so that you can use a removable device in the same way you use them in Windows. But it's not the norm. And besides, who doesn't want to manually edit the */etc/fstab* fle?

#10: Multilayered run levels vs. a single-layered run level

I couldn't figure out how best to title this point, so I went with a description. What I'm talking about is Linux' inherent ability to stop at different run levels. With this, you can work from either the command line (run level 3) or the GUI (run level 5). This can really save your socks when X Windows is fubared and you need to figure out the problem. You can do this by booting into run level 3, logging in as root, and finding/fixing the problem.

With Windows, you're lucky to get to a command line via safe mode — and then you may or may not have the tools you need to fix the problem. In Linux, even in run level 3, you can still get and install a tool to help you out (hello apt-get install APPLICATION via the command line). Having different run levels is helpful in another way. Say the machine in question is a Web or mail server. You want to give it all the memory you have, so you don't want the machine to boot into run level 5. However, there are times when you do want the GUI for administrative purposes (even though you can fully administer a Linux server from the command line). Because you can run the *startx*command from the command line at run level 3, you can still start up X Windows and have your GUI as well. With Windows, you are stuck at the Graphical run level unless you hit a serious problem.

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:

Linux Programming

-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:

1. 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.

2. 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).

3. To display a file:

   cat file1.txt

   This command displays the data in the file.

4. To concatenate several files and display:

Linux Programming

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. Some times the output may not fit the monitor screen. In such situation you can print those files in a new file or display the file using less command.

cat file1.txt file2.txt | less

5. 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.

Linux Programming

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.

Linux Programming

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

Linux Programming

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:

1. Copy two files:

   cp file1 file2

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

2. To backup the copied file:

   cp -b file1.php file2.php

   Backup of file1.php will be created with '~' symbol as file2.php~.

3. 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.

Linux Programming

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:

1. Display root directory contents:

   ls /

   lists the contents of root directory.

2. Display hidden files and directories:

   ls -a

   lists all entries including hidden files and directories.

3. Display inode information:

Linux Programming

ls -i

7373073 book.gif

7373074 clock.gif

7373082 globe.gif

7373078 pencil.gif

7373080 child.gif

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.

Linux Programming

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.

Linux Programming

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
Symbolic Mode	— — —

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

Linux Programming

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

Linux Programming

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.

Linux Programming

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.

Linux Programming

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.

Linux Programming

-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	
hscripts.com	HIOX TEST
	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.

Linux Programming

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.

Linux Programming

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.

Linux Programming

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

Linux Programming

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.

Linux Programming

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.

Linux Programming

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.

Linux Programming

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:

Linux Programming

```
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:

Linux Programming

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 | -   increment must be in the range 1-19; if not specified, an increment of 10 is
n increment   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

---

Linux Programming

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.

Linux Programming

-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

Linux Programming

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 <u>crontab command</u>.

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.

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 .

Linux Programming

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'.

Linux Programming

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

Linux Programming

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

Linux Programming

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.

Linux Programming

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

Linux Programming

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.

Linux Programming

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.

General Commands:

date COMMAND:

date command prints the date and time.

SYNTAX:

The Syntax is

date [options] [+format] [date]

OPTIONS:

	Slowly adjust the time by sss.fff seconds (fff represents fractions of a second).
-a	This adjustment can be positive or negative.Only system admin/ super user can adjust the time.
- s 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).

Linux Programming

%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:

1. date command

   date

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

2. 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

3. To know the week number of the year,

   date -V

Linux Programming

The above command will print 30

4. 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

Linux Programming

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

Linux Programming

\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

EXAMPLE:

1. echo command

   echo "hscripts Hiox India"

   The above command will print as hscripts Hiox India

2. To use backspace:

   echo -e "hscripts \bHiox \bIndia"

   The above command will remove space and print as hscriptsHioxIndia

3. 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]

Linux Programming

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]

Linux Programming

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

Linux Programming

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]]

Linux Programming

-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).

Linux Programming

-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   The nodename may be changed by specifying a system name argument. The
systemname   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.

Linux Programming

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
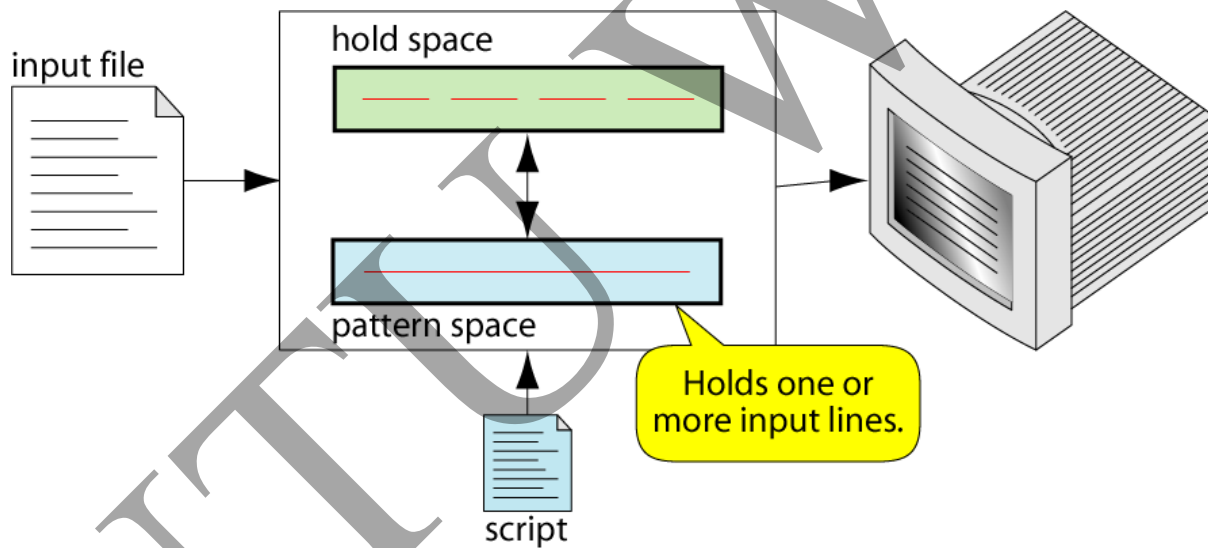 - Write conversion programs
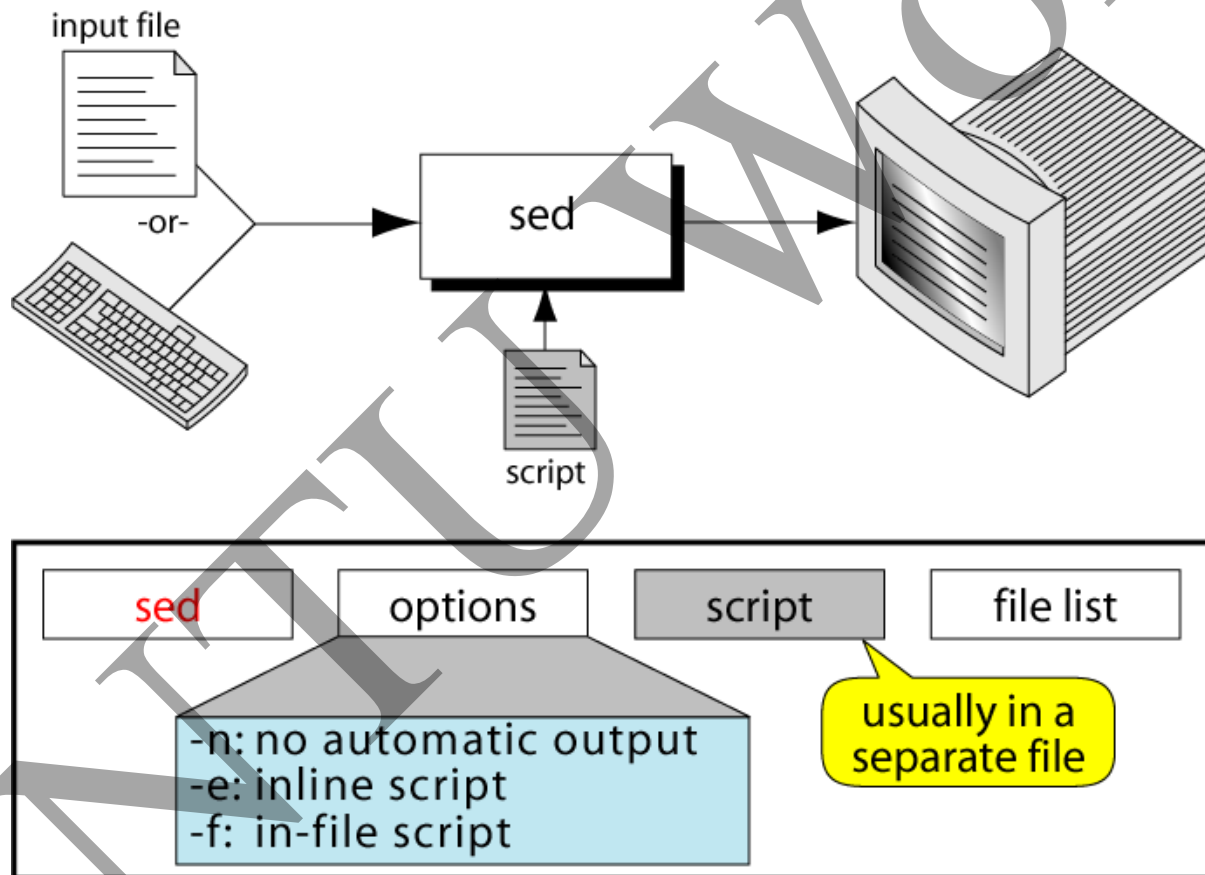
sed command syntax

Linux Programming

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

(a) Inline Script

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

(b) Script File

sed Operation

input file → hold space

pattern space

Holds one or more input lines.

script

How Does sed Work?

- O sed reads line of input
  - line of input is copied into a temporary buffer called pattern space
  - editing commands are applied
    - O subsequent commands are applied to line in the pattern space, not the original input line
    - O once finished, line is sent to output

Linux Programming

(unless –n option was used)

- line is removed from pattern space

  O sed reads next line of input, until end of file

<u>Note:</u> input file is unchanged

sed instruction format

- O 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
- O address types:
  - Single-Line address
  - Set-of-Lines address
  - Range address
  - Nested address

Single-Line Address

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

<u>Examples:</u>

- 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

Linux Programming

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

Linux Programming

- 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 (!):

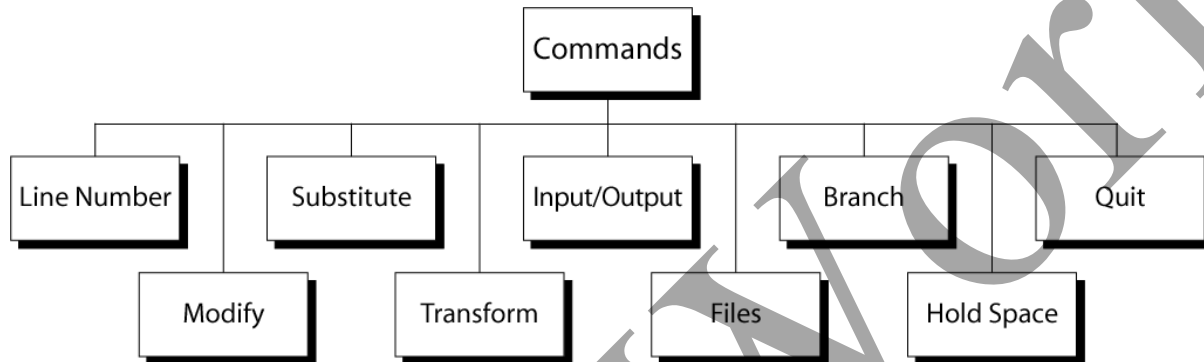 instruction will be applied to all lines that do not match the address

---

Linux Programming

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
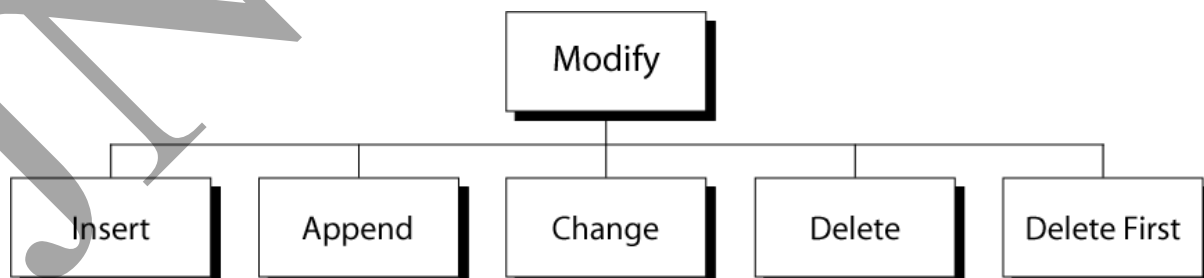
Insert Command: i

Linux Programming

- ○ 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

Linux Programming

- O 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]**

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

Regular Expressions: use with sed

Linux Programming

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
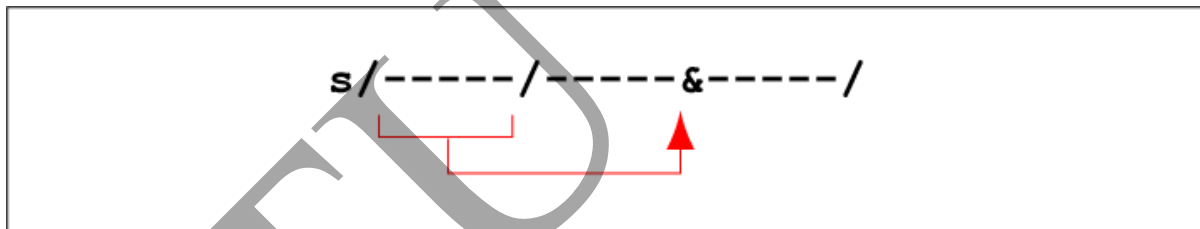
(a) Whole Pattern Substitution
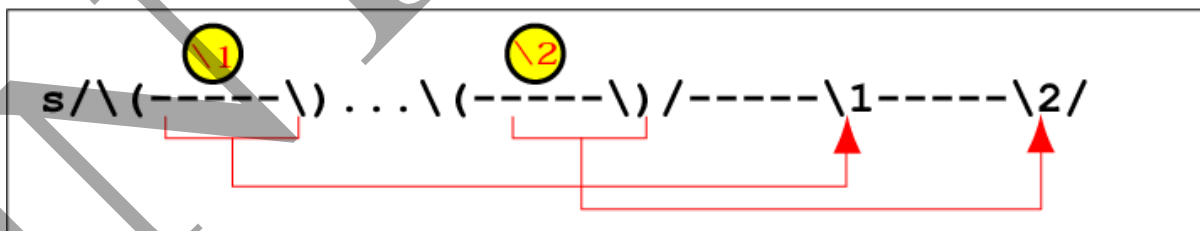
(b) Numbered Buffer Substitution

Example: Replacement String &

$ cat datafile

Linux Programming

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/
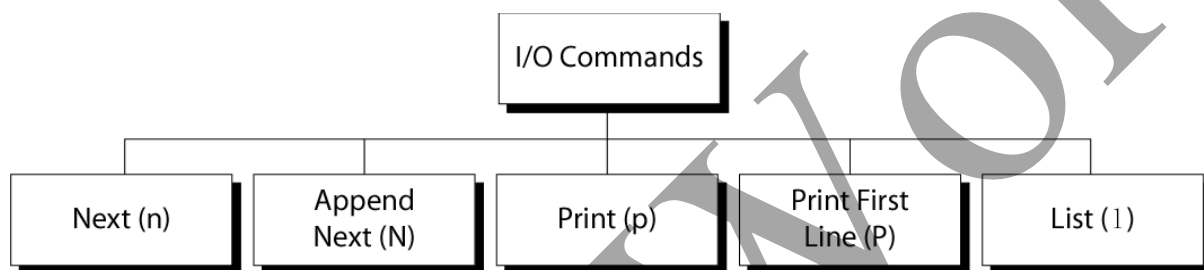
- translates one character 'a' to another 'b'

Linux Programming

- cannot use regular expression metacharacters
- cannot indicate a range of characters
- similar to "tr" command

<u>Example:</u>

$ sed -e '1,10y/abcd/wxyz/' datafile

sed i/o commands

```
                          ┌─────────────────┐
                          │  I/O Commands   │
                          └─────────────────┘
    ┌──────────┬──────────────┬──────────┬──────────────┬──────────┐
┌────────┐ ┌────────────┐ ┌──────────┐ ┌────────────┐ ┌─────────┐
│Next (n)│ │  Append    │ │Print (p) │ │ Print First│ │List (1) │
│        │ │  Next (N)  │ │          │ │  Line (P)  │ │         │
└────────┘ └────────────┘ └──────────┘ └────────────┘ └─────────┘
```

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

Linux Programming

- 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

---

Linux Programming

○ 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

Linux Programming

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

Linux Programming
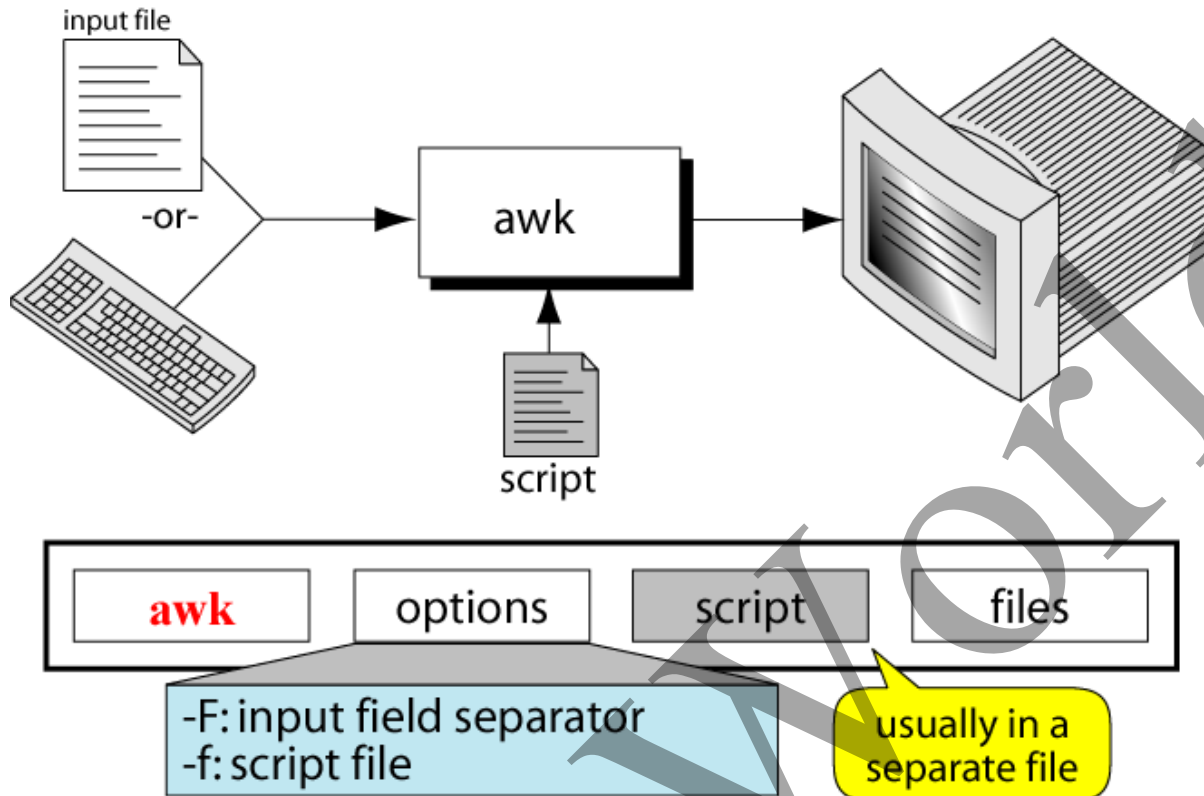
Basic awk Syntax

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

Options:

-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

Linux Programming

- if action is missing, the matched line is printed
- must have either pattern or action

Example:

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

A file with 10 records, each with four fields

Buffers

Linux Programming
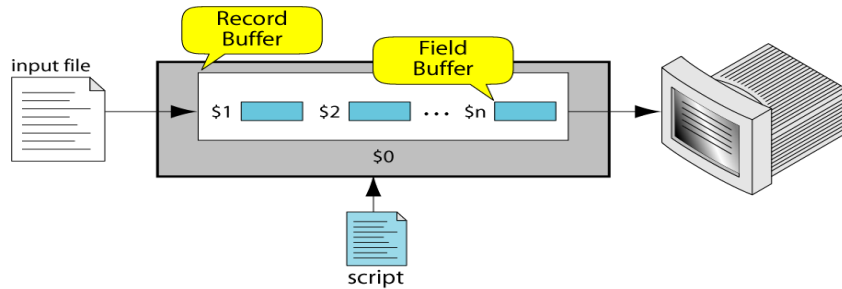
- ○ awk supports two types of buffers:

  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

Linux Programming

```
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
```

Linux Programming

4 Billy Black 336500

Example: Colon as Field Separator

% cat em2

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

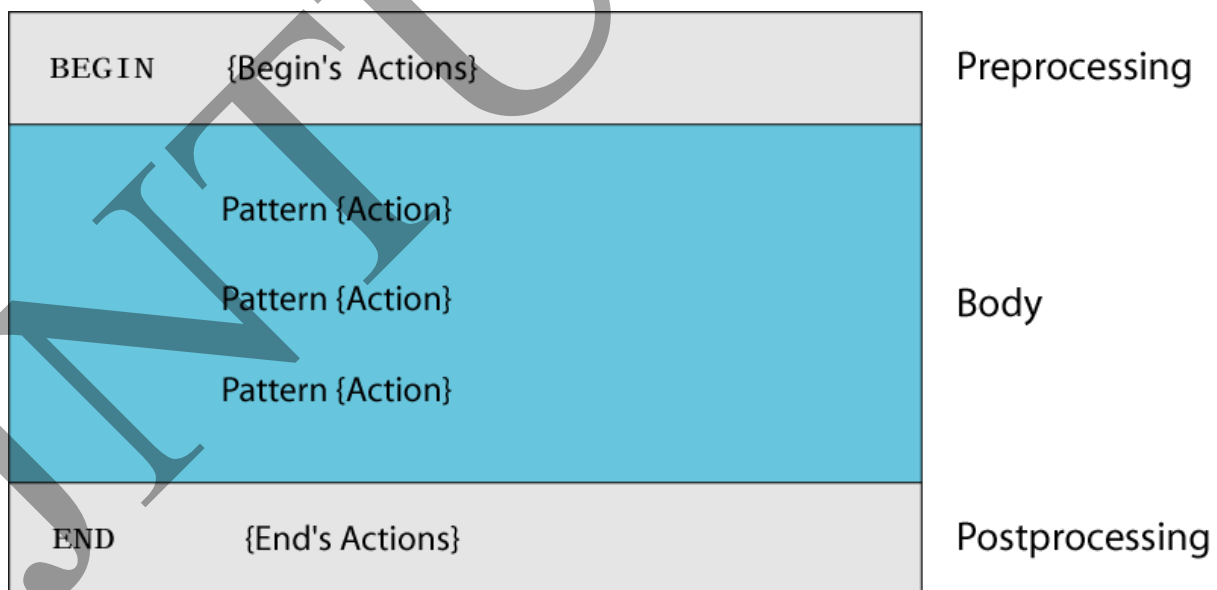**Mary Adams:5346:11/4/63:28765**

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 #

Linux Programming

awk Scripts

- **O** 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
- **O** BODY: Processing
  - contains main processing logic to be applied to input records
  - like a loop that processes input data one record at a time:
    - **O** if a file contains 100 records, the body will be executed 100 times, one for each record
- **O** 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}
```
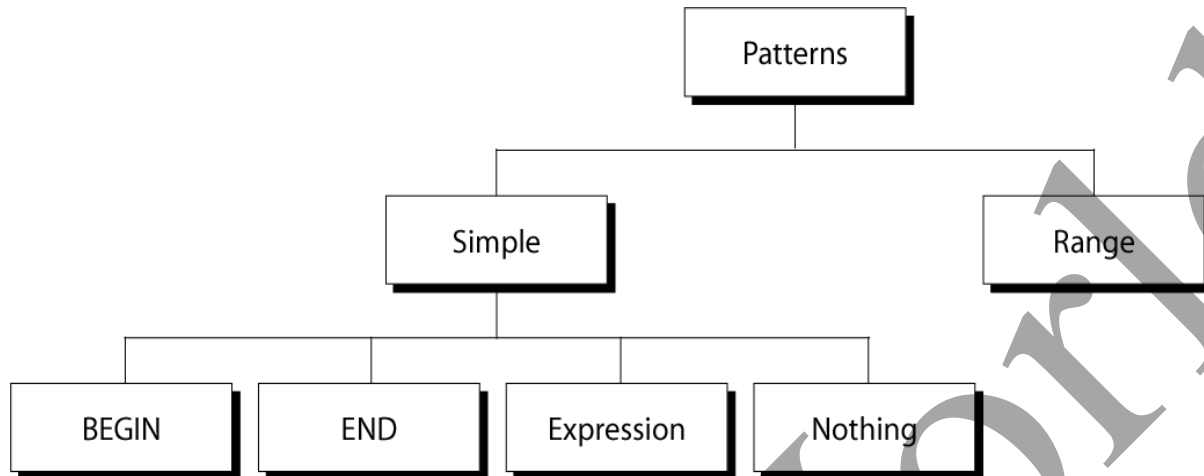
(b) Multiple Statements Separated by Semicolons

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

(c) Multiple Statements Separated by Newlines

Linux Programming

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

Example: match input record

% cat employees2

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

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

Linux Programming

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

Linux Programming

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

Linux Programming

/	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

Linux Programming

!	NOT	! a

<u>Examples:</u>

% awk '($2 > 5) && ($2 <= 15)   {print $0}' file

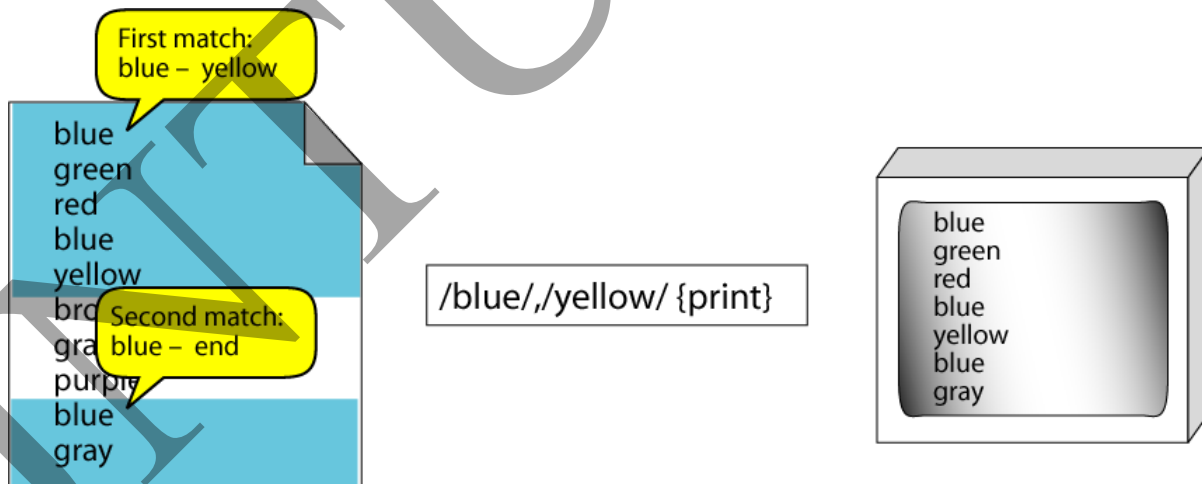**% awk '$3 == 100 || $4 > 50' file**

Range Patterns

- Matches ranges of consecutive input lines

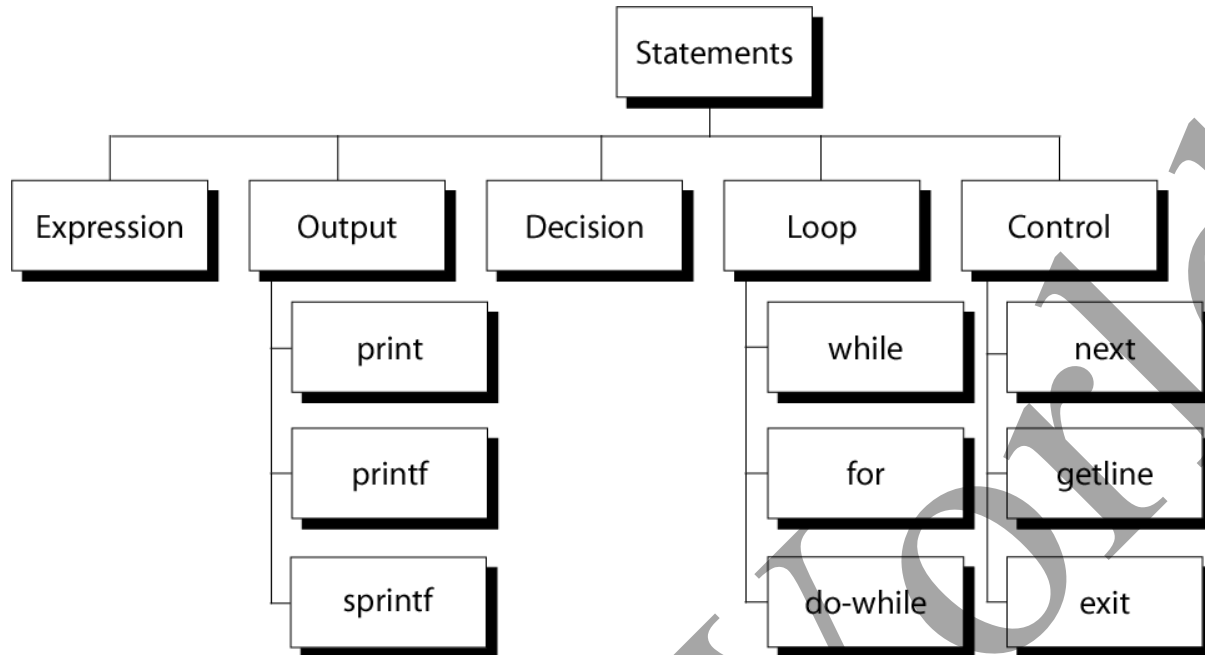<u>Syntax:</u>

      pattern1 , pattern2 {action}

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

Range Pattern Example

awk Actions

Linux Programming

awk expressions

- O Expression is evaluated and returns value
  - consists of any combination of numeric and string constants, variables, operators, functions, and regular expressions
- O Can involve variables
  - As part of expression evaluation
  - As target of assignment

awk variables

- O A user can define any number of variables within an awk script
- O The variables can be numbers, strings, or arrays
- O Variable names start with a letter, followed by letters, digits, and underscore
- O Variables come into existence the first time they are referenced; therefore, they do not need to be declared before use
- O All variables are initially created as strings and initialized to a null string ""

awk Variables

<u>Format:</u>

variable = expression

<u>Examples:</u>

% 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

Linux Programming

- 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

Linux Programming

- 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"

Linux Programming

>> "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

Linux Programming

Wed Nov 19 14:40:07 CST 2008

% date |

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

Month: 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, %idecimal 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 "%"

Linux Programming

Format specifier examples

Given: x = 'A', y = 15, z = 2.3, and $1 = Bob Smith	
Printf Format Specifier	**What it Does**
%c	*printf("The character is %c \n", x)* **output: The character is A**
%d	*printf("The boy is %d years old \n", y)* **output: The boy is 15 years old**
%s	*printf("My name is %s \n", $1)* **output: My name is Bob Smith**
%f	*printf("z is %5.3f \n", z)* **output: z is 2.300**

Format specifier modifiers

- between "%" and letter

       %10s

       %7d

       %10.4f

       %-20s

Linux Programming

- ○ 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 builtin functions

tolower(string)

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

    <u>Example:</u> tolower("MiXeD cAsE 123")

        returns "mixed case 123"

toupper(string)

Linux Programming

- O 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

Linux Programming

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

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= ":"

Linux Programming

```
        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"

}
```

awk Array

O   awk allows one-dimensional arrays

    to store strings or numbers

O   index can be number or string

O   array need not be declared

    - its size
    - its elements

Linux Programming

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

Arrays in awk

<u>Syntax:</u>

arrayName[index] = value

<u>Examples:</u>

list[1] = "one"

list[2] = "three"

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

Illustration: Associative Arrays

- awk arrays can use string as index

Name	Age	Department	Sales
"Robert"	46	"19-24"	1,285.72
"George"	22	"81-70"	10,240.32
"Juan"	22	"41-10"	3,420.42
"Nhan"	19	"17-A1"	46,500.18
"Jonie"	34	"61-61"	1,114.41

Index   Data   Index   Data

Awk builtin split function

split(string, array, fieldsep)

Linux Programming

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

<u>Example:</u>

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

O sets the contents of the array a as follows:

   a[1] = "auto"

   a[2] = "da"

   a[3] = "fe"

Example: process sales data

O input file:

Sales

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

O output:
- summary of category sales

Linux Programming

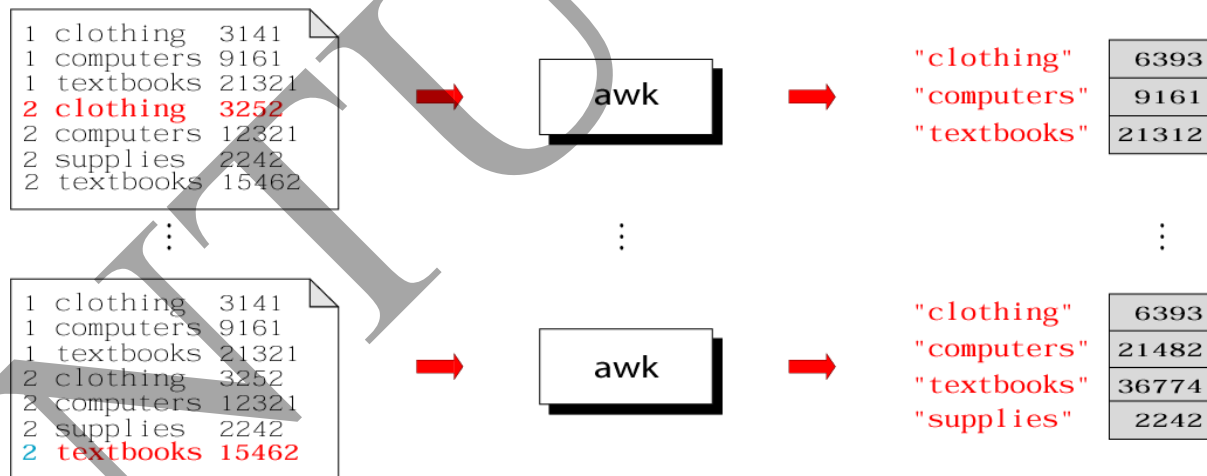Illustration: process each input line
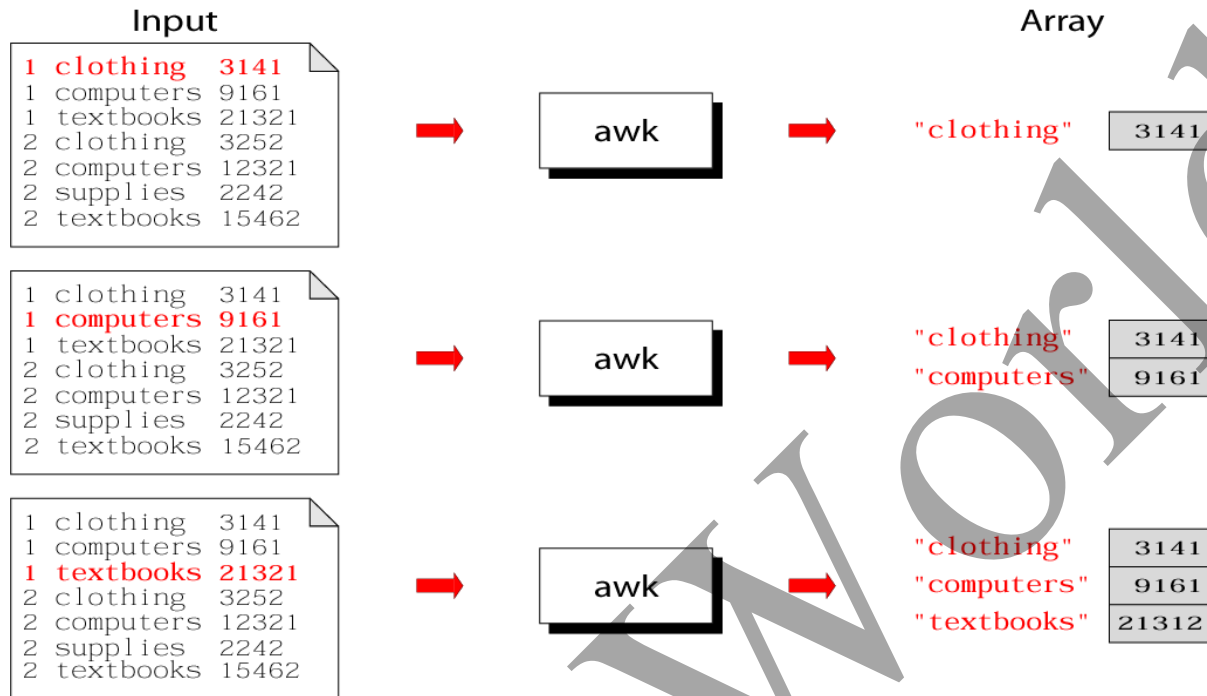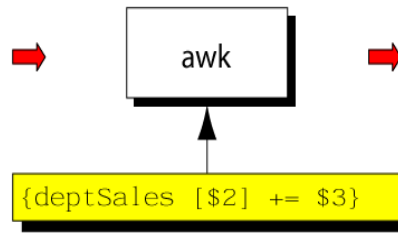

Illustration: process each input line

Summary: awk program

Linux Programming

Example: complete program

% cat sales.awk

{

    deptSales[$2] += $3

}

END {

    for (x in deptSales)

        print x, deptSales[x]

}

% awk –f sales.awk sales

Awk control structures

- Conditional
  - if-else
- Repetition
  - for
    - with counter
    - with array index

Linux Programming

- while
- do-while
- also: break, continue

if Statement

<u>Syntax:</u>

if (conditional expression)

      statement-1

else

      statement-2

<u>Example:</u>

if (NR < 3)

      print $2

else

      print $3

for Loop

<u>Syntax:</u>

for (initialization; limit-test; update)

     statement

<u>Example:</u>

for (i = 1; i <= NR; i++)

Linux Programming

```
{

    total += $i

    count++

}
```

for Loop for arrays

<u>Syntax:</u>

for (var in array)

    statement

<u>Example:</u>

for (x in deptSales)

{

    print x, deptSales[x]

}

while Loop

<u>Syntax:</u>

while (logical expression)

    statement

<u>Example:</u>

i = 1

Linux Programming

while (i <= NF)

{

    print i, $i

    i++

}

do-while Loop

<u>Syntax:</u>

    do

        statement

    while (condition)

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

<u>Example:</u>

 i = 1

  do {

  print $0

   i++

  } while (i <= 10)

loop control statements

   O **break**

Linux Programming

exits loop

○ **continue**

skips rest of current iteration, continues with next iteration

"Unit-II - Shell Programming"

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.
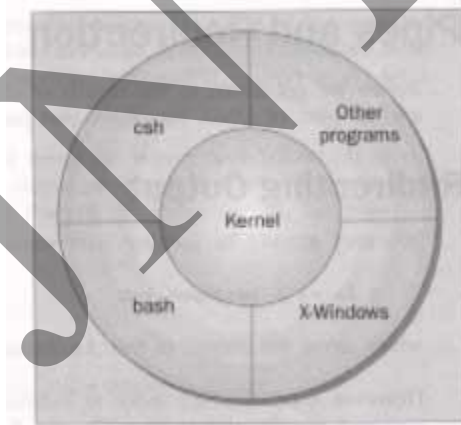
Linux Programming

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.

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 append 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.

Linux Programming

The Shell as a Programming Language

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
$
```

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

Linux Programming

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.

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.

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

Linux Programming

```
#!/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.

Environment Variables

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

Linux Programming

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.

The following shows the difference between using the variable $* and $@

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

notice that the first line of the above has a space between the firsr ' and the second '.

Now try your hand at typing a shell script

Carefully type the following into a file called: try_variables

Linux Programming

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"

echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"
exit 0
```

make sure to make it executable by typing the command:

< chmod a+x try_variables

Execute the file with parameters by typing:

try_variables foo bar baz

The results of executing the file is:

```
$ try_variables foo bar baz
Hello
The program ./try_variables is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The users home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

How It Works

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

Conditions

All programming languages have the ability to test conditions and perform different actions based on those conditions. A shell script can test the exit code of any command.

The test, or []Command

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

Linux Programming

```
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.

Linux Programming

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 [ $timeofday = "yes" ]
then
        echo "Good morning"
elif [ $timeofday = "no" ]; then
        echo "Good afternoon"
else
```

Linux Programming

```
          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
          fud
          43
```

Linux Programming

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
done
```

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 [ "$foo" -le 20 ]
do
        echo "Here we go again"
        foo=$(($foo+1))
done

exit 0
```

How It Works

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

```
foo=$(($foo+1))
```
increments the value of foo each time the loop executes..

Linux Programming

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
        sleep 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";;
```

Linux Programming

```
            "y" ) echo "Good Morning";;
            "n" ) echo "Good Afternoon";;
            * ) echo "Soory, answer not recognized";;
    esac

    exit 0
```

How It Works

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";;
    esac
```

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

Linux Programming

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

Lists

To test for multiple conditions, we can use nested if or if/elif.

The AND List

Alolows us to execute a series of command. Each command is only execute if the previous commands have succeeded. An AND list joins conditions by using &&.

```
            statement1 && statement2 && statement3 && ...
```
Her is a sample AND list:
```
            #!/bin/sh

            touch fine_one
            rm -f file_two

            if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
            then
                    echo "in if"
            else
                    echo "in else"
            fi

            exit 0
```

How It Works

The touch command creates an empty file. the rm come remove a file. So, before we start, file_one exists and file_two doesn't. The AND list finds the file_one, and echos the word hello, but it doesn't find the file file_two. Therefore the overall if fails and the else clause is executed.

The OR List

The OR list construct allows us to execute a series of commands until one succeeds!

```
            statement1 || statement2 || statement3 || ...
```
Here is a sample Or list
```
            rm -f file_one
```

Linux Programming

```
if [ -f file_one ] || echo "hello" || echo " there"
then
        echo "in if"
else
        echo "in else"
fi

exit 0
```

How It Works

The above script removes the file file_one, then test for and fails to find the file_one, but does successfully echo hello. It then executes the then statement echoing in if.

Statement Blocks

Multiple statements can be placed inside of { } to make a statement block.

Functions

You can define functions inthe shell. The syntax is:

```
function_name () {
        statements
}
```
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.

Linux Programming

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

```
#!/bin/sh

yes_or_no() {
        echo "Parameters are $*"
        while true
        do
                echo -n "Enter yes or no"
                read x
                case "$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

exit 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

Linux Programming

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.

Linux Programming

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.

$x =$ `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.

Linux Programming

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.

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.

Linux Programming

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.

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.

Arithmetic Expansion

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

Linux Programming

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.

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.

Debugging 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.

Linux Programming

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.

Putting It All Together

The rest of this chapter is about designing a CD database application.

Requirements

The system should store basic information about each CD, search for CDs, and update or add new CDs.

Design

The three requirements--updating, searching and displaying the CD data--suggest that a simple menu willbe adequate. Here is the example titles file.

Catalog	Title	Type	Composer
CD123	Cool sax	Jazz	Bix
CD234	Classic violin	Classical	Bach
CD345	Hits91	Pop	Various

Here is the associated track file.

Catalog	Track no	Title
CD123	1	Some jazz
CD123	2	More jazz
CD345	1	Dizzy
CD234	1	Sonata in D minor

Notes

The code for the CD database is included in the try it out section. The trap command allows the user to use Ctrl-C.

Linux Programming

Summary

By the time you enter the CD database application, you will know that programs can be written using just the shell language. The shell is used for much of Linux system administration.

"Unit Three - Working with Files"

Chapter Outline

Working with Files
    UNIX File Structure
        Directories
        Files and Devices
    System Calls and Device Drivers
    Library Functions
    Low-level File Access
        Other System Calls for Managing Files
    The Standard I/O Library
        Formatted Input and Output
        Other Stream Functions
        Stream Errors
        Stream and File Descriptors
    File and Directory Maintenance
    Scanning Directories
    Errors
    Advanced
    Summary

Lecture Notes

Working with Files

In this chapter we learn how to create, open, read, write, and close files.

UNIX File Structure

Linux Programming

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.

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.

Files are arranged in directories, which also contain subdirectories.

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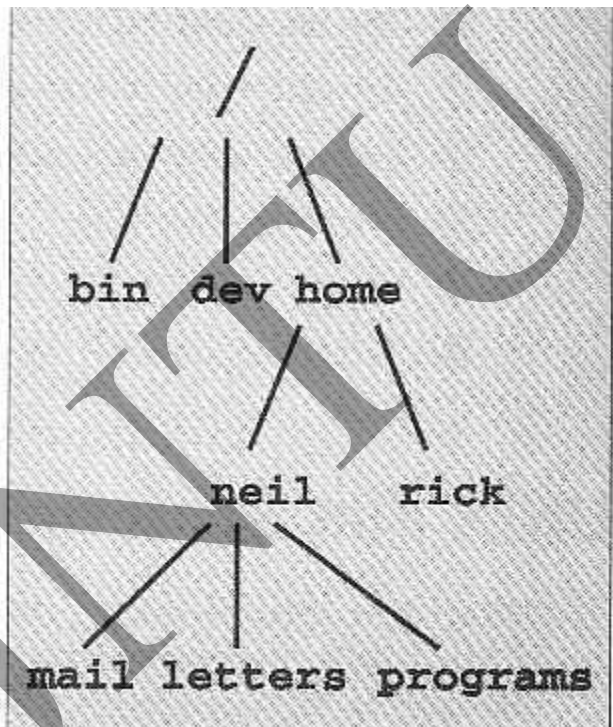
Files and Devices

Linux Programming

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.

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.

Linux Programming
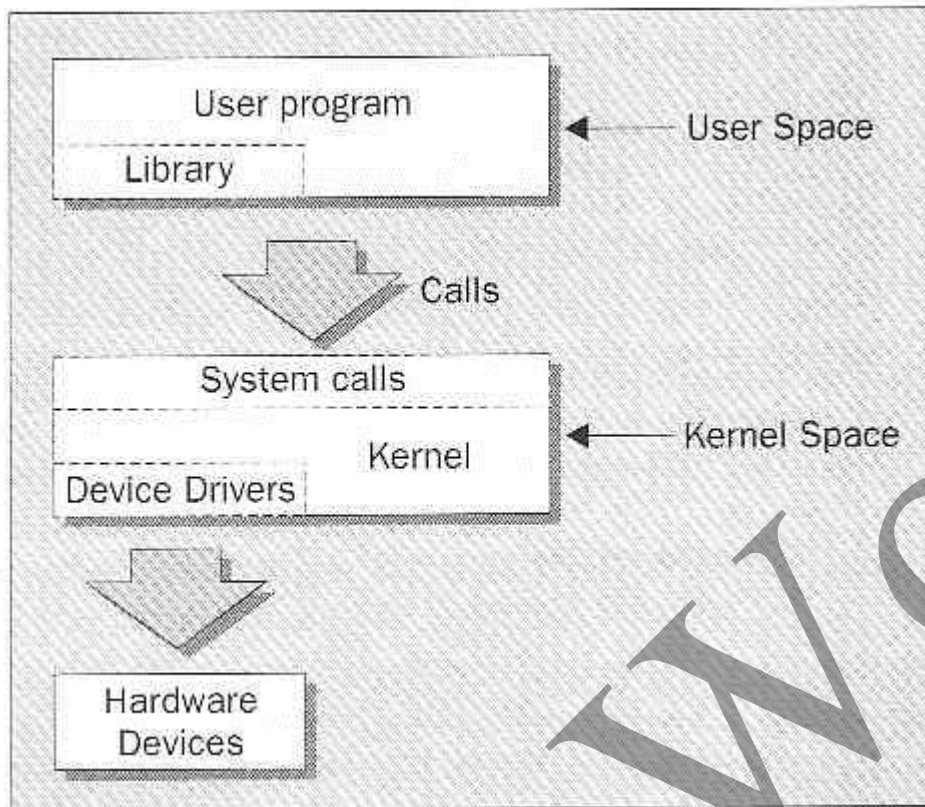
Low-level File Access

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**.

Linux Programming

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

```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"

    exit(0);
}
```

Here is how to run the program and its output.

> $ simple_write
> Here is some data
> $

read

```c
#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.

Linux Programming

```c
#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.

```c
#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);
```

FYI Strictly speaking, we don't need to include sys/types.h and sys/s
on POSIX systems, but they may be necessary on some UNIX syste

Linux Programming

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. Th atomic, i.e. it's performed with just one function call. This pro two programs creating the file at the same time. If the file alr 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:

Linux Programming

▶	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.

Foe 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.

Linux Programming

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

Linux Programming

```
#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!

Linux Programming

```
#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.

Linux Programming

```
#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

Linux Programming

```
#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

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:

Linux Programming

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:

Linux Programming

➤	**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)
    ...
```

dup and dup2

```
#include <unistd.h>

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

Linux Programming

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

The Standard I/O Library

The standard I/O library and its header file **stdio.h**, provide a versatile interface to low-level I/O system calls.

Three file streams are automatically opened when a program is started. They are **stdin**, **stdout**, and **stderr**.

Now, let's look at:

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc, getc, getchar
- fputc, putc, putchar

- fgets, gets
- printf, fprintf and sprintf
- scanf, fscanf and sscanf

fopen

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

The **fopen** library function is the analog of the low level **open** system call.

fopen opens the file named by the **filename** parameter and associates a stream with it. The **mode** parameter specifies how the file is to be opened. It's one of the following strings:

Linux Programming

"r" or "rb"	Open for reading only
"w" or "wb"	Open for writing, truncate to zero length
"a" or "ab"	Open for writing, append to end of file
"r+" or "rb+" or "r+b"	Open for update (reading and writing)
"w+" or "wb+" or "w+b"	Open for update, truncate to zero length
"a+" or "ab+" or "a+b"	Open for update, append to end of file

If successful, **fopen** returns a non-null **FILE *** pointer.

fread

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

The **fread** library function is used to read data from a file stream. Data is read into a data buffer given by **ptr** from the stream, **stream**.

fwrite

```
#include <stdio.h>

size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *st
```

The **fwrite** library call has a similar interface to **fread**. It takes data records from the specified data buffer and writes them to the output stream.

Note that `fread` *and* `fwrite` *can cause problems if used indiscriminately where structured data is being transferred between dissimilar machines. We'll discuss further issues of portability in Appendix A.*

fclose

Linux Programming

```
#include <stdio.h>

int fclose(FILE *stream);
```

The **fclose** library function closes the specified **stream**, causing any unwritten data to be written.

fflush

```
#include <stdio.h>

int fflush(FILE *stream);
```

The **fflush** library function causes all outpstanding data on a file stream to be written immediately.

fseek

```
#include <stdio.h>

int fseek(FILE *stream, long int offset, int whence);
```

The **fseek** function is the file stream equivalent of the **lseek** system call. It sets the position in the stream for the next read or write on that stream.

fgetc, getc, getchar

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar();
```

The **fgetc** function returns the next byte, as a character, from a file stream. When it reaches the end of file, it returns **EOF**.

The **getc** function is equivalent to **fgetc**, except that you can implement it as a macro.

The **getchar** function is equivalent to **getc(stdin)** and reads the next character from the standard input.

Linux Programming

fputc, putc, putchar

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

The **fputc** function writes a character to an output file stream. It returns the value it has written, or **EOF** on failure.

The function **putc** is quivalent to **fputc**, but you may implement it as a macro.

The **putchar** function is equivalent to **putc(c,stdout)**, writing a single character to the standard output.

fgets, gets

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

The **fgets** function reads a string from an input file **stream**. It writes characters to the string pointed to by **s** until a newline is encountered, **n-1** characters have been transferred or the end of file is reached.

Formatted Input and Output

There are library functions for producing output in a controlled fashion.

printf, fprintf and sprintf

```
#include <stdio.h>

int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

The **printf** family of functions format and output a variable number of arguments of different types.

Linux Programming

Ordinary characters are passed unchanged into the output. Conversion specifiers cause **printf** to fetch and format additional argumetns passed as parameters. They are start with a **%**.

For example

```
printf("Some numbers: %d, %d, and %d\n", 1, 2, 3);
```

which produces, on the standard output:

       Some numbers: 1, 2, and 3

Here are some of the most commonly used conversion specifiers:

▶	%d, %i	Print an integer in decimal.
▶	%o, %x	Print an integer in octal, hexadecimal.
▶	%c	Print a character.
▶	%s	Print a string.
▶	%f	Print a floating point (single precision) number.
▶	%e	Print a double precision number, in fixed format.
▶	%g	Print a double in a general format.

Here's another example:

```
char initial = 'A';
char *surname = "Matthew";
double age = 6.5;

printf("Hello Miss %c %s, aged %g\n", initial, surname, age);
```

This produces:

       Hello Miss A Mathew, aged 6.5

Field specifiers are given as numbers immediatley after the % character in a conversion specifier. Theya re used to make things clearer.

Linux Programming

Format	Argument	Output
%10s	"Hello"	`      Hello`
%-10s	"Hello"	`Hello      `
%10d	1234	`      1234`
%-10d	1234	`1234      `
%010d	1234	`0000001234`
%10.4f	12.34	`   12.3400`
%*s	10, "Hello"	`     Hello`

The **printf** function returns an integer, the number of characters written.

scanf, fscanf and sscanf

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

The **scanf** family of functions work in a similar way to the **printf** group, except that thye read items from a stream and place vlaues into variables.

The format string for **scanf** and friends contains both ordinary characters and conversion specifiers.

Here is a simple example:

```
int num;
scanf("Hello %d", &num);
```

The call to **scanf** will succeed and place **1234** into the variable **num** given either if the following inputs.

```
Hello    1234
Hello1234
```

Linux Programming

Other conversion specifiers are:

▶	%d	Scan a decimal integer.
▶	%o, %x	Scan an octal, hexadecimal integer.
▶	%f, %e, %g	Scan a floating point number.
▶	%c	Scan a character (whitespace not skipped).
▶	%s	Scan a string.
▶	%[]	Scan a set of characters (see below).
▶	%%	Scan a % character.

Given the input line,

```
Hello, 1234, 5.678, X, string to the end of the line
```

this call to **scanf** will correctly scan four items:

```
char s[256];
int n;
float f;
char c;

scanf("Hello,%d,%g, %c, %[^\n]", &n,&f,&c,s);
```

In general, **scanf** and friends are not highly regarded, for three reasons:

▶ Traditionally, the implementations have been buggy.

▶ They're inflexible to use.

▶ They lead to code where it's very difficult to work out what is being parse

Other Stream Functions

Other library functions use either stream paramters or the standard streams **stdin, stdout, stderr**:

Linux Programming

▶	fgetpos	Get the current position in a file stream.
▶	fsetpos	Set the current position in a file stream.
▶	ftell	Return the current file offset in a stream.
▶	rewind	Reset the file position in a stream.
▶	freopen	Reuse a file stream.
▶	setvbuf	Set the buffering scheme for a stream.
▶	remove	Equivalent to **unlink**, unless the **path** parameter is a directory in case it's equivalent to **rmdir**.

You can use the file stream functions to re-implement the file copy program, by using library functions.

Try It Out - Another File Copy Program

This program does the character-by-character copy is accomplished using calls to the functions referenced in **stdio.h**.

```c
#include <stdio.h>

int main()
{
    int c;
    FILE *in, *out;

    in = fopen("file.in","r");
    out = fopen("file.out","w");

    while((c = fgetc(in)) != EOF)
        fputc(c,out);

    exit(0);
}
```

Running this program as before, we get:

Linux Programming

         $ time copy_stdio
         1.69user 0.78system 0:03.70elapsed 66%CPU
This time, the program runs in 3.7 seconds.

Stream Errors

To indicate an error, many of the **stdio** library functions return out of range values, such as null pointers or the constant **EOF**.

In these cases, the error is indicated in the external variable **errno:**

```
#include <errno.h>

extern int errno;
```

Note that many functions may change the value of **errno.** *Its value is only valid when a function has failed. You should inspect it immediately after a function has indicated failure. You should always copy it into another variable before using it, because printing functions, such as* **fprintf,** *might alter* **errno** *themselves.*

You can also interrogate the state of a file stream to determine whether an error has occurred, or the end of file has been reached.

```
#include <stdio.h>

int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

The **ferror** function tests the error indicator for a stream and returns non-zero if its set, zero otherwise.

The **feof** function tests the end-of-file indicator within a stream and returns non-zero if it is set zero otherwise.

You use it like this:

Linux Programming

```
if(feof(some_stream))
      /* We're at the end */
```

The **clearerr** function clears the end-of-file and error indicators for the stream to which **stream** points.

Streams and File Descriptors

Each file stream is associated with a low level file descriptor.

You can mix low-level input and output operations with higher level stream operations, but this is generally unwise.

The effects of buffering can be difficult to predict.

```
#include <stdio.h>

int fileno(FILE *stream);
FILE *fdopen(int fildes, const char *mode);
```

File and Directory Maintenance

The standard libraries and system calls provide complete control over the creation and maintenance of files and directories.

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.

Linux Programming

```
#include <unistd.h>

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

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 edcrements 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.

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, getcwd

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

Linux Programming

```
#include <unistd.h>

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

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

Linux Programming

```
#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

Linux Programming

```
#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 will recurse for subdirectories.

Linux Programming

```c
#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);
```

```c
        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:

Linux Programming

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

        exit(0);
    }
```

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

```
$ 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.
```

How It Works

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**.

Linux Programming

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

Errors

System calls and functions can fail. When they do, they indicate the reason for their failure by setting the value of the external varaible **errno**.

The values and meanings of the errors are listed in the header file **errno.h**. They include:

Linux Programming

▶	**EPERM**	Operation not permitted
▶	**ENOENT**	No such file or directory
▶	**EINTR**	Interrupted system call
▶	**EIO**	I/O Error
▶	**EBUSY**	Device or resource busy
▶	**EEXIST**	File exists
▶	**EINVAL**	Invalid argument
▶	**EMFILE**	Too many open files
▶	**ENODEV**	No such device
▶	**EISDIR**	Is a directory
▶	**ENOTDIR**	Isn't a directory

There are a couple of useful functions for reporting errors when they occur: **strerror** and **perror**.

```
#include <string.h>

char *strerror(int errnum);
```

The **strerror** function maps an error number into a string describing the type of error that has occurred.

```
#include <stdio.h>

void perror(const char *s);
```

The **perror** function also maps the current error, as reported in **errno**, into a string and prints it on the standard error stream.

It's preceded by the message given in the string **s** (if not **null**), followed by a colon and a space. For example:

Linux Programming

```
    perror("program");
```

might give the following on the standard error output:

```
    program: Too many open files
```

Advanced Topics

fcntl

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**.

Linux Programming

The calls,

```
fcntl(fildes, F_GETFL)
fcntl(fildes, F_SETFL, flags)
```

respectively get and set the file status flags and access modes.

mmap

The **mmap** function creates a pointer to a region of memory associated with the contents of the file accessed through an open file descriptor.

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

void *mmap(void *addr, size_t len, int prot, int flags, int fildes,
```

You can use the **addr** parameter to request a particular memory address.

The **prot** parameter is used to set access permissions for the memory segment. This is a bitwise OR of the following constant values.

PROT_READ	The segment can be read.
PROT_WRITE	The segment can be written.
PROT_EXEC	The segment can be executed.
PROT_NONE	The segment can't be accessed.

The **flags** parameter controls how changes made to the segment by the program are reflected elsewhere.

MAP_PRIVATE	The segment is private, changes are local.
MAP_SHARED	The segment changes are made in the file.
MAP_FIXED	The segment must be at the given address, **addr**.

The **msync** function causes the changes in part or all of the memory segment to be written back to (or read from) the mapped file.

Linux Programming

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

int msync(void *addr, size_t len, int flags);
```

The part of the segment to be updated is given by the passed start address, **addr**, and length, **len**. The **flags** parameter controls how the update should be performed.

▶ MS_ASYNC   Perform asynchronous writes.

▶ MS_SYNC   Perform synchronous writes.

▶ MS_INVALIDATE   Read data back in from the file.

The **munmap** function releases the memory segment.

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

int munmap(void *addr, size_t len);
```

Try It Out - Using mmap

1. The following program, **mmap_eg.c** shows a file of structures being updated using **mmap** and array-style accesses.

Here is the definition of the **RECORD** structure and the create **NRECORDS** versions each recording their number.

Linux Programming

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

typedef struct {
    int integer;
    char string[24];
} RECORD;

#define NRECORDS (100)

int main()
{
    RECORD record, *mapped;
    int i, f;
    FILE *fp;

    fp = fopen("records.dat","w+");
    for(i=0; i<NRECORDS; i++) {
        record.integer = i;
        sprintf(record.string,"RECORD-%d",i);
        fwrite(&record,sizeof(record),1,fp);
    }
    fclose(fp);
```

2. We now change the integer value of record 43 to 143, and write this to the 43rd record's string:

```c
    fp = fopen("records.dat","r+");
    fseek(fp,43*sizeof(record),SEEK_SET);
    fread(&record,sizeof(record),1,fp);

    record.integer = 143;
    sprintf(record.string,"RECORD-%d",record.integer);

    fseek(fp,43*sizeof(record),SEEK_SET);
    fwrite(&record,sizeof(record),1,fp);
    fclose(fp);
```

Linux Programming

3. We now map the records into memory and access the 43rd record in order to change the integer to 243 (and update the record string), again using memory mapping:

```
f = open("records.dat",O_RDWR);
read(f,&record,sizeof(record));
mapped = (RECORD *)mmap(0, NRECORDS*sizeof(record),
                        PROT_READ|PROT_WRITE, MAP_SHARED, f, 0);

mapped[43].integer = 243;
sprintf(mapped[43].string,"RECORD-%d",mapped[43].integer);

msync((void *)mapped, NRECORDS*sizeof(record), MS_ASYNC);
munmap((void *)mapped, NRECORDS*sizeof(record));
close(f);

exit(0);
}
```

Summary

This chapter showed how LINUX provides direct access to files and devices..

"Unit Four - Processes and Signals"

Chapter Outline

Processes and Signals
     What is a Process
     Process Structure
          The Process Table
          Viewing Processes
          System Processes
          Process Scheduling
     Starting New Processes
          Waiting for a Process
          Zombie Processes
          Input and Output Redirection
          Threads
     Signals

Linux Programming

Sending Signals
Signal Sets
Summary

Lecture Notes

Processes and Signals

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.

Process Structure

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

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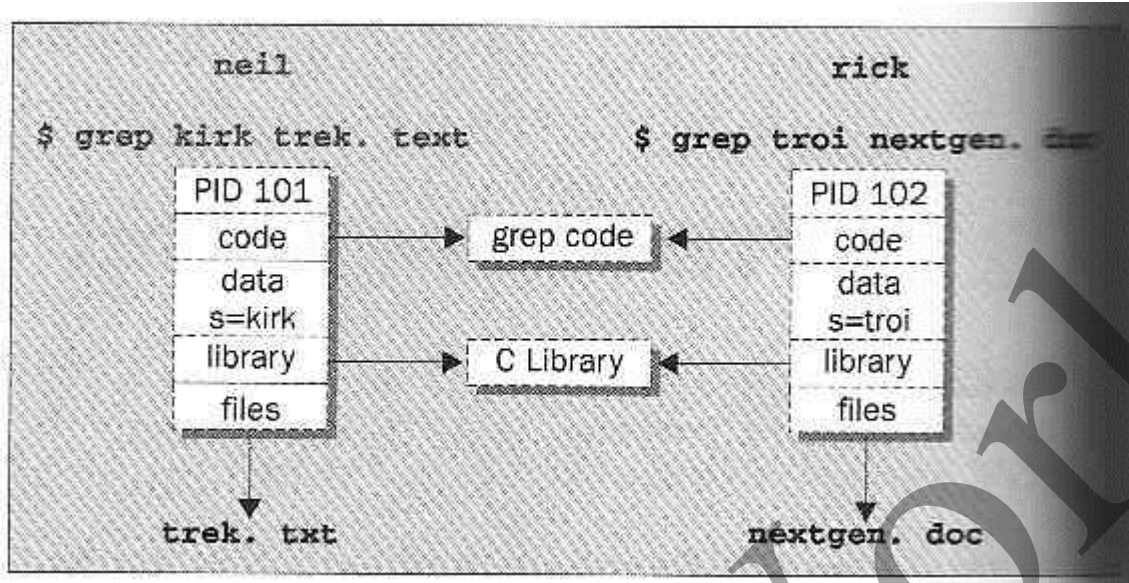
Each process is allocated a unique number, a **process identifier**, or PID.

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.

The Process Table

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:

Linux Programming

```
$ 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. This 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.

Linux Programming

```
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.

System Processes

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.

Linux Programming

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

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

Process Scheduling

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:

Linux Programming

```
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.

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.

Linux Programming

```
#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 desired program.

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:

Linux Programming

```
$ ./system2
Running ps with system
Done.
S    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, cons
*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[])
```

Linux Programming

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 /bi
```

```
execlp("ps", "ps", "-ax", 0);                          /* assumes /bin is in PA
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /* passes own environmen

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.

Linux Programming

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

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", 0);
    printf("Done.\n");
    exit(0);
}
```

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
```

Duplicating a Process Image

To use processes to perform more than one function at a time, we need to create an entirely separate process from within a program.

Linux Programming

We can create a new process by calling **fork**. This system call duplicates the current process.

Combined with **exec, fork** is all we need to create new processes to do our bidding.

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

pid_t fork(void);
```

The **fork** system call creates a new child process, identical to the calling process except that the new process has a unique process ID and has the calling process as its parent PID.

A typical code fragment using **fork** is:

```
pid_t new_pid;

new_pid = fork();

switch(new_pid) {
case -1 : /* Error */
    break;
case 0  : /* We are child */
    break;
default : /* We are parent */
    break;
}
```

Try It Out - fork

Let's look at a simple example, **fork.c**:

Linux Programming

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

int main()
{
    pid_t pid;
    char *message;
    int n;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        break;
    default:
        message = "This is the parent";
        n = 3;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}
```

This program runs as two process. A child prints a message five times. The parent prints a message only three times.

Linux Programming

```
$ ./fork
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
```

How It Works

When the call to **fork** is made, this program divides into two separate processes.

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:

Linux Programming

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 co
WIFSIGNALED(stat_val)	Non-zero if the child is terminated on an uncaught
WTERMSIG(stat_val)	If WIFSIGNALED is non-zero, this returns a signal r
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 nu

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**.

Linux Programming

```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:

Linux Programming

```
    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

Linux Programming

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

   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 youu can use it to wait for a specific process to terminate.

Linux Programming

```
#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.

Input and Output Redirection

We can use our knowledge of processes to alter the behavior of programs by exploiting the fact that open file descriptors are preserved across calls to **fork** and **exec**.

Try It Out - Redirection

1. Here's a very simple filter program, **upper.c**, to convert all characters to uppercase:

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

int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

When we run this program, it reads our input and converts it:

Linux Programming

```
$ ./upper
hello THERE
HELLO THERE
^D
$
```

We can, of course, use it to convert a file to uppercase by using the shell redirection:

$ cat file.txt
this is the file, file.txt, it is all lower case.
$ upper < file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.

2. What if we want to use this filter fromwithin another program? This code, **useupper.c**, accepts a file name as an argument and will respond with an error if called incorrectly:

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

int main(int argc, char *argv[])
{
    char *filename;

    if(argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
        exit(1);
    }

    filename = argv[1];
```

3. The done, we reopen the standard input, again checking for any errors as we do so, and then use **execl** to call **upper**:

Linux Programming

```
if(!freopen(filename, "r", stdin)) {
    fprintf(stderr, "could not redirect stdin to file %s\n", fil
```

```
    exit(2);
}

execl("./upper", "upper", 0);
```

4. don't forget that **execl** replaces the current process; provided there is no error, the remaining lines are not executed:

```
    fprintf(stderr, "could not exec upper!\n");
    exit(3);
}
```

How It Works

when we run this program, we can give it a file to convert to uppercase. The job is done by the program **upper**. The program is executed by:

```
 ./useupper file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

Because open file descriptors are preserved across the call to **execl**,
the **upper** program runs exactly as it would have under the shell command:

```
 upper < file.txt
```

Threads

UNIX processes can cooperate; they can send each other messages and they can interrupt one another.

Linux Programming

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:

Linux Programming

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.

Try It Out - Signal 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**:

Linux Programming

```
#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:

Linux Programming

```
$ ./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.

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.

Linux Programming

```
#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:

```c
#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:

```c
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.

Linux Programming

```
        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**.

Linux Programming

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction
```

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**.

Linux Programming

```
#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.

Linux Programming

```
$ ./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.

Linux Programming

```
#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.

Linux Programming

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

sigaction Flags

The **sa_flags** field of the **sigaction** structure used in **sigaction** may contain the following values to modify signal behavior

▶	**SA_NOCLDSTOP**	Don't generate **SIGCHLD** when child processes stop.
▶	**SA_RESETHAND**	Reset signal action to **SIG_DFL** on receipt.
▶	**SA_RESTART**	Restart interruptible functions rather than error with **EIN**
▶	**SA_NODEFER**	Don't add the signal to the signal mask when caught.

Functions that are safe to call inside a signal handler, those guaranteed by the X/Open specification either to be re-entrant or not to raise signals themselves include:

Linux Programming

access	fstat	read	sysconf
alarm	getegid	rename	tcdrain
cfgetispeed	geteuid	rmdir	tcflow
cfgetospeed	getgid	setgid	tcflush
cfsetispeed	getgroups	setpgid	tcgetattr
cfsetospeed	getpgrp	setsid	tcgetpgrp
chdir	getpid	setuid	tcsendbreak
chmod	getppid	sigaction	tcsetattr
chown	getuid	sigaddset	tcsetpgrp
close	kill	sigdelset	time
creat	link	sigemptyset	times
dup2	lseek	sigfillset	umask
dup	mkdir	sigismember	uname
execle	mkfifo	signal	unlink
execve	open	sigpending	utime
_exit	pathconf	sigprocmask	wait
fcntl	pause	sigsuspend	waitpid
fork	pipe	sleep	write
stat			

Common Signal Reference

Here we list the signals that UNIX programs typically need to get involved with,
including the default behaviors:

Linux Programming

Signal Name	Description
SIGALRM	Generated by the timer set by the **alarm** function.
SIGHUP	Sent to the controlling process by a disconnecting terminal, or by t controlling process on termination to each foreground process.
SIGINT	Typically raised from the terminal by typing *Ctrl-C* or the configur interrupt character.
SIGKILL	Typically used from the shell to forcibly terminate an errant proces signal can't be caught or ignored.
SIGPIPE	Generated if a pipe with no associated reader is written to.
SIGTERM	Sent as a request for a process to finish. Used by UNIX when shutt to request that system services stop. This is the default signal sent command.
SIGUSR SIGUSR2	May be used by processes to communicate with each other, possib cause them to report status information.

The default action signals is abnormal termination of the process.

Signal Name	Description
SIGFPE	Generated by a floating point arithmetic exception.
SIGILL	An illegal instruction has been executed by the processor. Usually c corrupt program or invalid shared memory module.
SIGQUIT	Typically raised from the terminal by typing *Ctrl-* or the configured
SIGSEGV	A segmentation violation, usually caused by reading or writing at a location in memory either by exceeding array bounds or de-referen pointer. Overwriting a local array variable and corrupting the stack **SIGSEGV** to be raised when a function returns to an illegal address

By default, these signals also cause abnormal termination. Additionally, implementation-dependent actions, such as creation of a core file, may occur.

Linux Programming

Signal Name	Description
SIGSTOP	Stop executing (can't be caught or ignored).
SIGTSTP	Terminal stop signal, often raised by typing *Ctrl-Z*.
SIGTTIN	Used by the shell to indicate that background jobs have stopped bec
SIGTTOU	to read from the terminal or produce output.

A process is stopped by default on receipt of one of the above signals.

Signal Name	Description
SIGCONT	Continue executing, if stopped.

SIGCONT restarts a stopped process and is ignored if received by a process which is not stopped.

Signal Name	Description
SIGCHLD	Raised when a child process stops or exits.

The **SIGCHLD** signal is ignored by default.

Summary

We have seen how processes are a fundamental part of the LINUX operation system.

We have also learned to start, terminate, and signal between processes.